How to Use DLPI

Neal Nuckolls

Internet Engineering

1.0 Status

This paper, manual pages, and all example programs are available via anonymous ftp from newstop.EBay.Sun.COM as the file *dltest.tar.Z* .

2.0 Purpose

This paper shows how user-level programs in SunOS 5.x can use the STREAMS Data Link Provider Interface to transmit and receive raw datalink-level frames. Examples are used to demonstrate the use of the interface. This paper *does not* describe how to write DLPI device drivers. The purpose of this paper is to provide the applications programmer within enough information to allow him/her to write programs which directly access the raw datalink drivers in the system to send and receive raw datalink frames.

Few programs need to access the raw datalink device drivers. Most networking applications should use the Transport Layer Interface (TLI), socket, or Transport-Independent RPC interface in SunOS 5.x. But a certain class of applications, such as network monitoring programs, require access to the underlying datalink interface and must use DLPI for this.

3.0 Background

The Data Link Provider Interface specifies a STREAMS kernel-level implementation of the ISO Data Link Service Definition (ISO 8886) and Logical Link Control (ISO 8802/2) LLC.

DLPI enables a data link service user to access and use any of a variety of conforming data link service providers without special knowledge of the provider's protocol. Specifically, the interface is intended to support X.25 LAPB, BX.25 level2, SDLC, ISDN LAPD, Ethernet, FDDI, Token Ring, Token Bus, BISYNC, and other datalink-level protocols.

The interface specifies access to data link service providers, and does not define a specific protocol or protocol implementation.

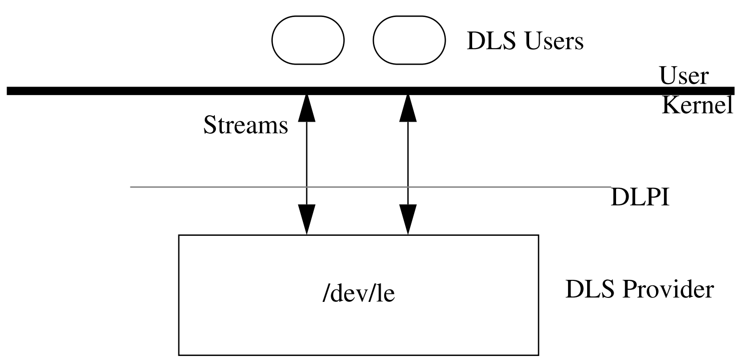
SunOS 5.0 (Solaris 2.0) includes support for DLPI Version 2. Most networking datalink-level device drivers for SunOS 5.x support DLPI Version 2 as their primary interface. A user-level DLPI library interface (DLLI) is not currently available. Again, since the number of programs which should access the raw datalink layer is limited, supporing a formal user-level lirbrary seemed unnecessary. This paper and the example demo programs contain more than sufficient code examples for most DLPI users.

The DLPI header file shipped with SunOS 5.x is *<sys/dlpi.h>*.

The DLPI Version 2 specification can be ordered from Sun. Don't try to derive the interface from the header file - get a copy:

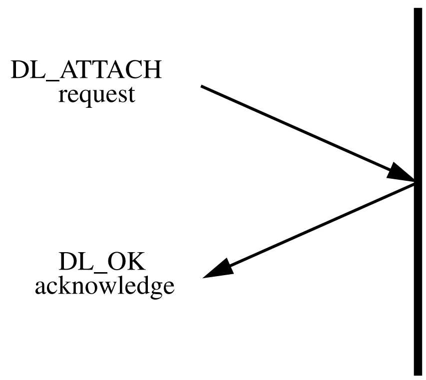
STREAMS DLPI Spec., Sun Microsystems, 800-6915-01.

DLPI defines a STREAMS message set, state table, and conventions to allow the interaction between a Data Link Service User (DLS user) and Data Link Service Provider (DLS provider). Within SunOS 5.x, the DLS users include the kernel IP and ARP protocols and the DLS providers are the datalink drivers such as the *le* (LANCE Ethernet) driver. Other DLS users include *in.rarpd*, the user-level reverse ARP daemon program and *snoop*, a networking monitor program which has replaced *etherfind* . Additionally, the *ifconfig* program is a DLS user when it reports or configures certain datalink-level values of the system networking interfaces.



4.0 DLPI Quick Tutorial

The DLPI primitives are defined in terms of STREAMS messages. All DLPI messages are either M_PROTO or M_PCPROTO type and contain the appropriate DLPI message structure in the leading mblk. The DLPI primitives are generally issued in a request/acknowledge scenerio where the user issues a request message and waits for an acknowledge or error response message from the provider.



DLPI provides support for connectionless and connection-oriented datalink protocols. This paper will focus on providing examples of how to use connectionless datalink providers.

The DLPI Version 2 connectionless primitives allow the user to cause the provider to perform the following operations:

- attach a Stream to a particular addressable physical device
- bind a datalink-level service access point address to the Stream
- transmit datalink frames
- receive datalink frames
- enable reception of multicast group addresses
- enable physical and datalink-level “promiscuous” reception
- get/set physical address of attached device
- transmit/receive LLC XID/TEST messages

DLPI supports the concept of a DLSAP address which consists of a physical address plus a datalink service access point (SAP) address. For example, the DLSAP address format for a datalink provider for Ethernet is the 6 byte Ethernet physical address followed immediately by the 2 byte Ethernet type to create an 8 byte Ethernet DLSAP address. The DLSAP address format differs from provider to provider. Information included in the DL_INFO_ACK primitive allows the user to compose and decompose the physical and datalink portions to/from a DLSAP address for that particular provider. There are examples of how to do this later.

Before transmitting and receiving datalink frames, the user must open the datalink provider, attach the file descriptor to a "Physical Point of Attachment" (PPA), and bind a Service Access Point (SAP) address to the file descriptor.

Two styles of DLS provider are defined by DLPI, distinguished by the way they enable a DLS user to choose a particular physical point of attachment (PPA). Opening the datalink provider creates a new Stream to the device and returns a file descriptor.

The *style1* provider assigns a PPA to the Stream based on the major/minor device numbers the user opened. Style1 providers "implicitly" attach the Stream to a device and do not require the user to explicitly send a DL_ATTACH_REQ message downstream to the provider for this. An example of a style1 provider would probably be a serial line multiplexer driver which allowed the user to open the minor device which corresponded to the particular serial port number. Opening the special device file */dev/serial000* would probably open minor device zero and associate this Stream with port zero of the device.

The *style2* provider requires a DLS user to explicitly identify the desired PPA using the attach service primitive. Most DLS providers which provide shared access for multiple Streams to a particular device port will be implemented as style2 providers. An example of a style2 provider is the le0 (LANCE Ethernet) device driver shipped with the SunOS 5.x system. Opening the special file */dev/le* creates a new Stream but does not select a particular ethernet interface. The user must then send a DL_ATTACH_REQ message downstream and receive a DL_OK_ACK response to associate the Stream with a particular le interface.

After the file descriptor has been attached to a PPA, the user must issue the DL_BIND_REQ primitive to associate the attached file descriptor with a full DLSAP address.

After the file descriptor has been attached and bound, the user can send and receive raw datalink frames or issue additional, optional DLPI primitives to access optional features of the DLS provider.

5.0 How to Use DLPI

The examples below demonstrate the use of the connectionless DLPI version 2 message set.

5.1 Example DLPI message

The typical DLPI streams message consists of an M_PROTO or M_PCPROTO type mblk_t followed by zero or more M_DATA type mblk's. Encapsulated within the leading M_PROTO/M_PCPROTO m_blk will be a longword aligned data structure defining one of the DLPI primitives. Here is the data structure for the DL_ATTACH_REQ primitive:

```
/*
 * DL_ATTACH_REQ, M_PROTO type
 */
typedef struct {
        ulong   dl_primitive;  /* set to DL_ATTACH_REQ */
        ulong   dl_ppa;        /* id of the PPA */
} dl_attach_req_t;
```

5.2 Attaching and Binding

The program below demonstrates how to open, attach, and bind a datalink provider file descriptor, issue the inforeq primitive, and print some information about the provider.

The program takes three command line arguments. The *device* argument specifies the full path of the device driver. The *ppa* argument is the id of the point of physical attachment interpreted by the driver. It identifies the unit number or port number of the physical device within the system. The third argument, *sap*, is an integer representing the datalink layer service access point and also interpreted by the driver. This tends to idenfy one of the protocol entities which will use this provider.

An example of the arguments and output of this program follows the source listing.

```
/*
 * NAME
 * dlinfo - print information about the datalink provider
 * SYNOPSIS:
 *     dlinfo device ppa sap
 *
 * Open datalink provider.
 * Attach to PPA.
 * Bind to sap
 * Get DL_INFO_ACK.
 * Print DL_INFO_ACK information.
 */

#include<sys/types.h>
#include<sys/stropts.h>
#include<sys/dlpi.h>
#include<sys/signal.h>
#include<fcntl.h>
#include<stdio.h>
#include"dltest.h"

main(ac, av)
int     ac;
char    *av[];
{
        long    buf[MAXDLBUF];/* aligned on long */
        union   DL_primitives*dlp;
        char    *device;
        int     ppa;
        int     fd;
        int     sap;

        dlp = (union DL_primitives*) buf;

        if (ac != 4)
                usage(av[0]);

        device = av[1];
        ppa = atoi(av[2]);
        sap     = atoi(av[3]);

        /*
         * Open the device.
         */
        if ((fd = open(device, O_RDWR)) < 0)
                syserr(device);

        /*
         * Attach.
         */
        dlattachreq(fd, ppa);
        dlokack(fd, buf);

        /*
         * Bind.
         */
        dlbindreq(fd, sap, 0, DL_CLDLS, 0, 0);
        dlbindack(fd, buf);
```

```
        /*
         * Issue DL_INFO_REQ.
         */
        dlinforeq(fd);
        dlinfoack(fd, buf);

        /*
         * Print DL_INFO_ACK values.
         */
        printdlprim(dlp);

        return (0);
}

usage(av0)
char    *av0;
{
        (void) fprintf(stderr, "%s: ", av0);
        (void) fprintf(stderr, "Usage: %s device ppa sap\n", av0);
        (void) exit(1);
}


dlattachreq(fd, ppa)
int     fd;
u_long  ppa;
{
        dl_attach_req_tattach_req;
        struct  strbuf  ctl;
        int     flags;

        attach_req.dl_primitive = DL_ATTACH_REQ;
        attach_req.dl_ppa = ppa;

        ctl.maxlen = 0;
        ctl.len = sizeof (attach_req);
        ctl.buf = (char *) &attach_req;

        flags = 0;

        if (putmsg(fd, &ctl, (struct strbuf*) NULL, flags) < 0)
                syserr("dlattachreq: putmsg");
}

dlbindreq(fd, sap, max_conind, service_mode, conn_mgmt, xidtest)
int     fd;
u_long  sap;
u_long  max_conind;
u_long  service_mode;
u_long  conn_mgmt;
u_long  xidtest;
{
        dl_bind_req_tbind_req;
        struct  strbuf  ctl;
        int     flags;

        bind_req.dl_primitive = DL_BIND_REQ;
        bind_req.dl_sap = sap;
        bind_req.dl_max_conind = max_conind;
```

```
        bind_req.dl_service_mode = service_mode;
        bind_req.dl_conn_mgmt = conn_mgmt;
        bind_req.dl_xidtest_flg = xidtest;

        ctl.maxlen = 0;
        ctl.len = sizeof (bind_req);
        ctl.buf = (char *) &bind_req;

        flags = 0;

        if (putmsg(fd, &ctl, (struct strbuf*) NULL, flags) < 0)
               syserr("dlbindreq: putmsg");
}

dlbindack(fd, bufp)
int     fd;
char    *bufp;
{
        union   DL_primitives*dlp;
        struct  strbuf  ctl;
        int     flags;

        ctl.maxlen = MAXDLBUF;
        ctl.len = 0;
        ctl.buf = bufp;

        strgetmsg(fd, &ctl, (struct strbuf*)NULL, &flags, "dlbindack");

        dlp = (union DL_primitives *) ctl.buf;

        expecting(DL_BIND_ACK, dlp);

        if (flags != RS_HIPRI)
               err("dlbindack: DL_OK_ACK was not M_PCPROTO");

        if (ctl.len < sizeof (dl_bind_ack_t))
               err("dlbindack: short response ctl.len: %d", ctl.len);
}

dlokack(fd, bufp)
int     fd;
char    *bufp;
{
        union   DL_primitives*dlp;
        struct  strbuf  ctl;
        int     flags;

        ctl.maxlen = MAXDLBUF;
        ctl.len = 0;
        ctl.buf = bufp;

        strgetmsg(fd, &ctl, (struct strbuf*)NULL, &flags, "dlokack");

        dlp = (union DL_primitives *) ctl.buf;

        expecting(DL_OK_ACK, dlp);

        if (ctl.len < sizeof (dl_ok_ack_t))
               err("dlokack: response ctl.len too short: %d", ctl.len);
```

```
        if (flags != RS_HIPRI)
                err("dlokack: DL_OK_ACK was not M_PCPROTO");

        if (ctl.len < sizeof (dl_ok_ack_t))
                err("dlokack: short response ctl.len: %d", ctl.len);
}


dlinforeq(fd)
int     fd;
{
        dl_info_req_tinfo_req;
        struct  strbuf  ctl;
        int     flags;

        info_req.dl_primitive = DL_INFO_REQ;

        ctl.maxlen = 0;
        ctl.len = sizeof (info_req);
        ctl.buf = (char *) &info_req;

        flags = RS_HIPRI;

        if (putmsg(fd, &ctl, (struct strbuf*) NULL, flags) < 0)
                syserr("dlinforeq: putmsg");
}

dlinfoack(fd, bufp)
int     fd;
char    *bufp;
{
        union   DL_primitives*dlp;
        struct  strbuf  ctl;
        int     flags;

        ctl.maxlen = MAXDLBUF;
        ctl.len = 0;
        ctl.buf = bufp;

        strgetmsg(fd, &ctl, (struct strbuf*)NULL, &flags, "dlinfoack");

        dlp = (union DL_primitives *) ctl.buf;

        expecting(DL_INFO_ACK, dlp);

        if (ctl.len < sizeof (dl_info_ack_t))
                err("dlinfoack: response ctl.len too short: %d", ctl.len);

        if (flags != RS_HIPRI)
                err("dlinfoack: DL_INFO_ACK was not M_PCPROTO");

        if (ctl.len < sizeof (dl_info_ack_t))
                err("dlinfoack: short response ctl.len: %d", ctl.len);
}
```

When executed, the program produces the following output:

```
# dlinfo
dlinfo: Usage: dlinfo device ppa sap
# dlinfo /dev/le 0 0
DL_INFO_ACK: max_sdu 1500 min_sdu 0
addr_length 8 mac_type DL_CSMACD current_state DL_IDLE
sap_length -2 service_mode DL_CLDLS qos_length 0
qos_offset 0 qos_range_length 0 qos_range_offset 0
provider_style DL_STYLE2 addr_offset 76 version 2
brdcst_addr_length 6 brdcst_addr_offset 84
addr 8:0:20:10:ab:ed:0:0
brdcst_addr ff:ff:ff:ff:ff:ff
#
```

Note that the DLSAP address length for the le driver is 8 and the sap_length is -2 which means that the first 6 bytes are the physical address (Ethernet address) and the final 2 bytes are the SAP (Ethernet type) for this particular provider.

5.3 Transmitting messages

Once the file descriptor has been attached to a physical interface and a SAP has been bound, you can send and receive datalink messages using the DL_UNITDATA_REQ and DL_UNITDATA_IND dlpi primitives.

```
/*
 * Xmit data buffer. Fill with whatever pattern you like ...
 */
char    xmitbuf[MAXDLBUF];

main(ac, av)
int     ac;
char    *av[];
{
        char    *device;
        int     ppa;
        int     fd;
        int     localsap;
        int     sapval;
        int     n;
        int     size;
        int     physlen, saplen;
        u_char  phys[MAXDLADDR], sap[MAXDLADDR];
        u_char  addr[MAXDLADDR];
        int     addrlen;
        long    buf[MAXDLBUF];
        union   DL_primitives*dlp;
        int     i;


        ...

        if (ac != 8)
                usage(av[0]);

        device = av[1];
        ppa = atoi(av[2]);
        localsap = atoi(av[3]);
        n = atoi(av[6]);
        size = atoi(av[7]);

        ...

        /*
         * Convert destination address string to address.
         */
        physlen = stringtoaddr(av[4], phys);
        sapval = atoi(av[5]);
        for (i = 0; i < sizeof (long); i++)
                sap[i] = (sapval >> (i * BITSPERBYTE)) & 0xff;

        ...

        /*
         * Get info.
         */
        dlinforeq(fd);
```

```
        dlinfoack(fd, buf);

        ...

        saplen = ABS(dlp->info_ack.dl_sap_length);
        addrlen = saplen + physlen;

        /*
         * Construct destination address.
         */
        if (dlp->info_ack.dl_sap_length > 0) {/* order is sap+phys */
                (void) memcpy((char*)addr, (char*)sap, saplen);
                (void) memcpy((char*)addr+saplen, (char*)phys, physlen);
        } else {/* order is phys+sap */
                (void) memcpy((char*)addr, (char*)phys, physlen);
                (void) memcpy((char*)addr+physlen, (char*)sap, saplen);
        }

        /*
         * Transmit 'size' packet 'n' times.
         */
        for (i = 0; i < n; i++)
                dlunitdatareq(fd, addr, addrlen, 0, 0, xmitbuf, size);
        ...



dlunitdatareq(fd, addrp, addrlen, minpri, maxpri, datap, datalen)
int     fd;
u_char  *addrp;
int     addrlen;
u_long  minpri, maxpri;
u_char  *datap;
int     datalen;
{
        long    buf[MAXDLBUF];
        union   DL_primitives*dlp;
        struct  strbuf  data, ctl;

        dlp = (union DL_primitives*) buf;

        dlp->unitdata_req.dl_primitive = DL_UNITDATA_REQ;
        dlp->unitdata_req.dl_dest_addr_length = addrlen;
        dlp->unitdata_req.dl_dest_addr_offset = sizeof (dl_unitdata_req_t);
        dlp->unitdata_req.dl_priority.dl_min = minpri;
        dlp->unitdata_req.dl_priority.dl_max = maxpri;

        (void) memcpy(OFFADDR(dlp, sizeof (dl_unitdata_req_t)), addrp, addrlen);

        ctl.maxlen = 0;
        ctl.len = sizeof (dl_unitdata_req_t) + addrlen;
        ctl.buf = (char *) buf;

        data.maxlen = 0;
        data.len = datalen;
        data.buf = (char *) datap;

        if (putmsg(fd, &ctl, &data, 0) < 0)
                syserr("dlunitdatareq: putmsg");
}
```

5.4 Receiving messages

Received datalink packets are subject to two levels of address matching. Packets which do not match the enabled physical address for the file descriptor are rejected. Packets which do not match the enabled SAP address(es) for the file descriptor are rejected. Physical and SAP level address matching are independent on a per-packet and per-file descriptor basis. The physical and SAP addresses enabled on one opened file descriptor have no effect on the packets received on another file descriptor. Only those received packet which match both the enabled physical address(es) and enabled SAP addresses(s) are sent upstream to the user.

Attaching to a PPA enables that physical address on the file descriptor. Binding a sap enables that sap address on the file descriptor. Additional physical and sap addresses can be enabled for the file descriptor using the promiscuous and multicast primitives.

```
...
data.buf = (char *) databuf;
data.maxlen = MAXDLBUF;
data.len = 0;

while (getmsg(fd, NULL, &data, &flags) == 0) {

        if (data.len) {
                for (i = 0; i < data.len; i++)
                        printf("%02x ", data.buf[i] & 0xff);
                printf("\n");
        }

        data.len = 0;
}
...
```

5.5 RAW mode

One feature which is not included in the USL DLPI version 2 specification but is frequetly necessary for several types of applications is “raw” mode. Support for this mode has been added to SunOS 5.x . The user enables raw mode by issuing the DLIOCRAW ioctl which is defined in the version of dlpi.h shipped with SunOS 5.x . After this, “raw” ethernet frames which include the 14 byte ethernet header may be sent downstream in M_DATA type messages and all received frames will be sent upstream with ethernet header included in M_DATA type messages.

The DLIOCRAW ioctl takes no arguments.

5.6 Getting and Setting the physical address

Use the DL_PHYS_ADDR_REQ primitive to request the current or default factory physical (ethernet) address which is returned in the DL_PHYS_ADDR_ACK message.

```
        ...
        /*
         * Get our current physical address.
         */
        dlphysaddrreq(fd, DL_CURR_PHYS_ADDR);
        dlphysaddrack(fd, buf);

        dlp = (union DL_primitives*) buf;

        /*
         * Create raw ethernet header.
         */
        ehp = (struct ether_header*) xmitbuf;
        memcpy(&ehp->ether_dhost, phys, ETHERADDRL);
        memcpy(&ehp->ether_shost,
               OFFADDR(dlp, dlp->physaddr_ack.dl_addr_offset), ETHERADDRL);
        ehp->ether_type = (u_short) sapval;

        /*
         * Put file descriptor in "raw mode".
         */
        if (strioctl(fd, DLIOCRAW, -1, 0, 0) < 0)
               syserr("ioctl DLIOCRAW");

        /*
         * Transmit it as an M_DATA msg.
         */
        if (write(fd, xmitbuf, size) < 0)
               syserr("write");
        ...
```

```
dlphysaddrreq(fd, addrtype)
int     fd;
u_long  addrtype;
{
        dl_phys_addr_req_tphys_addr_req;
        struct  strbuf  ctl;
        int     flags;

        phys_addr_req.dl_primitive = DL_PHYS_ADDR_REQ;
        phys_addr_req.dl_addr_type = addrtype;

        ctl.maxlen = 0;
        ctl.len = sizeof (phys_addr_req);
        ctl.buf = (char *) &phys_addr_req;

        flags = 0;

        if (putmsg(fd, &ctl, (struct strbuf*) NULL, flags) < 0)
               syserr("dlphysaddrreq: putmsg");
}
```

```
dlphysaddrack(fd, bufp)
int     fd;
char    *bufp;
{
        union   DL_primitives*dlp;
        struct  strbuf  ctl;
        int     flags;

        ctl.maxlen = MAXDLBUF;
        ctl.len = 0;
        ctl.buf = bufp;

        strgetmsg(fd, &ctl, (struct strbuf*)NULL, &flags, "dlphysaddrack");

        dlp = (union DL_primitives *) ctl.buf;

        expecting(DL_PHYS_ADDR_ACK, dlp);

        if (flags != RS_HIPRI)
                err("dlbindack: DL_OK_ACK was not M_PCPROTO");

        if (ctl.len < sizeof (dl_phys_addr_ack_t))
                err("dlphysaddrack: short response ctl.len: %d", ctl.len);
}
```

Use the DL_SET_PHYS_ADDR_REQ primitive to set the physical address of the attached interface. Note: this is a destructive operation and unlike all other primitives will affect other file descriptors for this physical interface. The new physical address for the interface will be retained until the next DL_SET_PHYS_ADDR_REQ is issued or the system is rebooted. Only the superuser may issue this primitive.

5.7 Multicast addressing

Use DL_ENABMULTI_REQ and DL_DISABMULTI_REQ to enable/disable specific multicast group addresses. These primitives allow the user to iteratively build a set of enabled multicast addresses on a per-file-descriptor basis.

5.8 Promiscuous reception

DL_PROMISCON_REQ and DL_PROMISCOFF_REQ allow the user to enable/ disable promiscuous-mode reception. These two primitives take a *level* argument which may be one of:

DL_PROMISC_PHYS

DL_PROMISC_SAP

DL_PROMISC_MULTI

Each of the promiscuous levels acts like a "wildcard" for the particular level. Enabling DL_PROMISC_PHYS will match all physical addresses. Enabling DL_PROMISC_SAP will match all sap addresses. Enabling all DL_PROMISC_MULTI will Attaching to a PPA enables that physical address on the file descriptor. Binding a sap enables that sap address on the file descriptor. Additional physical and sap addresses can be enabled for the file descriptor using the promiscuous and multicast primitives.

match all multicast-group addresses.

The following example demonstrates how one would enable physical-level promiscuous mode:

```
        ...

        /*
         * Enable promiscuous mode.
         */
        dlpromisconreq(fd, DL_PROMISC_PHYS);
        dlokack(fd, buf);

        ...

dlpromisconreq(fd, level)
int     fd;
u_long  level;
{
        dl_promiscon_req_tpromiscon_req;
        struct  strbuf  ctl;
        int     flags;

        promiscon_req.dl_primitive = DL_PROMISCON_REQ;
        promiscon_req.dl_level = level;

        ctl.maxlen = 0;
        ctl.len = sizeof (promiscon_req);
        ctl.buf = (char *) &promiscon_req;

        flags = 0;

        if (putmsg(fd, &ctl, (struct strbuf*) NULL, flags) < 0)
                syserr("dlpromiscon: putmsg");

}
```

6.0 Converting NIT programs to use DLPI

The Network Interface Tap (NIT) STREAMS pseudo driver is not included in SunOS 5.x . In previous releases of SunOS the underlying datalink device drivers were not directly accessible as UNIX special files therefore a pseudo driver, NIT, was supported to allow those few user-level programs which required access to the raw datalink drivers to access the devices.

In 5.x, all networking drivers are real STREAMS hardware drivers accessible in the file system namespace. To access the "le0" networking interface you now open /dev/le instead of /dev/nit and issue DLPI primitives instead of NIT primitives. DLPI provides a superset of the features previously provided by the NIT pseudo driver.

Applications which used NIT in SunOS 4.x will require some source-level modifications to run in SunOS 5.x . With the information provided below, the application programmer should find that these changes relatively simple and straightforward.

SunOS 4.x included three NIT components:

1. nit_if(4M) pseudo driver.
2. nit_pf(4M) packet filter module.
3. nit_buf(4M) buffer module.

The nit_if pseudo driver is not included in 5.x . The packet filter and buffer modules have been ported to 5.x and minor enhancements made. Refer to the pfmod(1M) and bufmod(1M) manual pages for the complete programmatic interfaces for these two general-purpose STREAMS modules.

6.1 Open driver directly

Open the datalink driver instead of /dev/nit:

```
/*
 * Open the device.
 */
if ((fd = open("/dev/le", 2)) < 0)
        syserr("/dev/le");
```

6.2 NIT ioctls are now DLPI primitives or Buffer module ioctls

The NIT ioctl commands have changed slightly:

Replace use of the SIOCGIFADDR ioctl with the DL_PHYS_ADDR_REQ dlpi primitive to get the current or default physical ethernet address.

Replace use of the SIOCSIFADDR ioctl with the DL_SET_PHYS_ADDR_REQ dlpi primitive to set the current physical ethernet address. You must be superuser to do this and this operation is destructive and permanently changes the individual ethernet address of the interface for all open Streams until it is changed again or the system rebooted.

Replace use of the SIOCADDMULTI and SIOCDELMULTI ioctls with the DL_ENABMULTI_REQ and DL_DISABMULTI_REQ dlpi primitives. Each datalink provider will support a fixed number of specific multicast addresses. Use the DL_PROMISCON_REQ with DL_PROMISC_MULTI dl_level value to enable *all* multicast group addresses.

Replace use of the NIOCBIND ioctl command with the DL_ATTACH_REQ and DL_BIND_REQ dlpi primitives. The attach primitive performs the equivalent function of the NIOCBIND ioctl and the bind primitive then labels the attached interface with a service access point (sap) address. The bundled ethernet drivers implement the sap address as the ethernet type. You must bind a sap address to the interface before you can send or receive; this is not an optional command.

Replace use of the NIOCSSNAP ioctl command with the SBIOCSSNAP buffer module ioctl command. You must push “bufmod” before issuing the SBIOCSSNAP in the same manner as the old NIT NIOCSSNAP ioctl.

The NIOCSFLAGS ioctl command has changed. Instead of setting the NI_PROMISC flag you must now issue the DL_PROMISCON_REQ dlpi primitive setting the DL_PROMISC_PHYS value in the dl_level field. Instead of setting the NI_TIMESTAMP and NI_DROPS flags you must push the buffer module; timestamp and dropcounts are included in the default buffer module chunk message headers which also includes the original and truncated packet lengths.

6.3 Attach and Bind

Use the DLPI primitives DL_ATTACH_REQ and DL_BIND_REQ to first associate the file descriptor with a particular physical interface (like NIOCBIND) and then label the attached file descriptor with a sap address. In the bundled 4.x ethernet drivers implement the sap as the ethernet type. So if you wanted to receive IP (0x0800) type ethernet packets only you'd do this:

```
...
/*
 * Attach.
 * Arguments: file descriptor, ppa identifier.
 */
dlattachreq(fd, ppa);
dlokack(fd, buf);


/*
 * Bind.
 * Arguments: file descriptor, sap, max conind, service mode, conn_mgmt, xidtest.
 */
dlbindreq(fd, 0x800, 0, DL_CLDLS, 0, 0);
dlbindack(fd, buf);
...
```

The complete example routines for attaching and binding are included in the first example in the previous section.

6.4 Filtering on ethernet type comes free

After binding to a specific type, only packets of that ethertype will be received on that file descriptor. You don't need to push and configure the packet filter module to filter on ethertype since you get that "for free". If you want to enable multiple ethernet types on the same file descriptor you'll have to enable promiscuous sap-level service using the DL_PROMISCON_REQ primitive and setting the DL_PROMISC_SAP value in the dl_level field and then push and configure the packet filter module to filter out the packets with ethertype you're not interested in receiving.

6.5 The DLIOCRAW ioctl sets "raw" mode

The standard connectionless DLPI primitives to send and receive datalink packets are DL_UNITDATA_REQ and DL_UNITDATA_IND. Each of these primitives is implemented as an M_PROTO STREAMS message type followed by one or more M_DATA message types.

If you just want to send and receive "raw" ethernet frames including the ethernet header you can do this by enabling "raw mode" on the particular file descriptor. Issue the DLIOCRAW ioctl:

```
        ...
        /*
         * Issue DLIOCRAW ioctl.
         */
        if (strioctl(fd, DLIOCRAW, -1, 0, NULL) < 0)
                syserr("DLIOCRAW");
        ...
```

```
strioctl(fd, cmd, timout, len, dp)
int     fd;
int     cmd;
int     timout;
int     len;
char    *dp;
{
        struct  strioctlsioc;
        int     rc;

        sioc.ic_cmd = cmd;
        sioc.ic_timout = timout;
        sioc.ic_len = len;
        sioc.ic_dp = dp;
        rc = ioctl(fd, I_STR, &sioc);

        if (rc < 0)
                return (rc);
        else
                return (sioc.ic_len);
}
```

Now all received frames will be passed up without changes within M_DATA messages and you can send raw ethernet frames downstream in M_DATA messages.

6.6 Enabling promiscuous mode is slightly different

Use the DL_PROMISCON_REQ with the dl_level field set to DL_PROMISC_PHYS to enable physical-level promiscuous mode:

```
        ...
        /*
         * Enable promiscuous mode.
         */
        dlpromisconreq(fd, DL_PROMISC_PHYS);
        dlokack(fd, buf);
        ...


dlpromisconreq(fd, level)
int     fd;
u_long  level;
{
        dl_promiscon_req_tpromiscon_req;
        struct  strbuf  ctl;
        int     flags;

        promiscon_req.dl_primitive = DL_PROMISCON_REQ;
        promiscon_req.dl_level = level;

        ctl.maxlen = 0;
        ctl.len = sizeof (promiscon_req);
        ctl.buf = (char *) &promiscon_req;

        flags = 0;

        if (putmsg(fd, &ctl, (struct strbuf*) NULL, flags) < 0)
                syserr("dlpromiscon: putmsg");

}
```

6.7 Packet filter module is the same

The packet filter module, now called “pfmod” is almost identical to the SunOS 4.x version. 1GYou can push it and program it as before.

6.8 Buffer module has some of the old nit_if features

The buffer module has had a few additions and includes a few features that were a part of the 4.x nit_if pseudo driver. In addition to the standard *chunksize* and *timeout* values for defining when the “chunk” is passed upstream, the 5.x bufmod also includes a *snaplength* parameter to truncate uninteresting parts of the packet, *dropcount* of number of received packets dropped due to read-side flow-control, and *timestamp* of each received packet. You push “bufmod” on any Stream in which you want one or more of these generic features.

The buffer module header data structure has changed slightly:

```
struct sb_hdr {
        u_int   sbh_origlen;
        u_int   sbh_msglen;
        u_int   sbh_totlen;
        u_int   sbh_drops;
        struct  timeval sbh_timestamp;
};
```

The names of the buffer module ioctl commands have all changed to take on the new prefix (“SB” stands for STREAMS Buffer Module):

NIOCSTIME is now SBIOCSTIME

NIOCGTIME is now SBIOCGTIME

NIOCCTIME is now SBIOCCTIME

NIOCSCHUNK is now SBIOCSCHUNK

NIOCGCHUNK is now SBIOCGCHUNK

Additional ioctls are documented in the bufmod(1M) manual page.