# LLA to DLPI Migration Guide

**Edition 4**


**B2355-90138**
**HP 9000 Networking**
**E0497**

Printed in: United States

# Legal Notices

The information in this document is subject to change without notice.

*Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.* Hewlett-Packard shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

**Warranty.** A copy of the specific warranty terms applicable to your Hewlett- Packard product and replacement parts can be obtained from your local Sales and Service Office.

**Restricted Rights Legend.** Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DOD agencies, and subparagraphs (c) (1) and (c) (2) of the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

HEWLETT-PACKARD COMPANY 3000 Hanover Street Palo Alto, California 94304 U.S.A.

Use of this manual and flexible disk(s) or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs may be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

# Contents

# Contents

## Printing History

The manual printing date and part number indicate its current edition. The printing date will change when a new edition is printed. Minor changes may be made at reprint without changing the printing date. the manual part number will change when extensive changes are made.

Manual updates may be issued between editions to correct errors or document product changes. To ensure that you receive the updated or new editions, you should subscribe to the appropriate product support service. See your HP sales representative for details.

First Edition: February 1991

Second Edition: July 1992

Third Edition: January 1995

Fourth Edition: April 1997

# Preface

Link Level Access for the HP 9000 (LLA/9000) is a Hewlett-Packard data communications and data management product supported on earlier HP-UX releases. The Data Link Provider Interface (DLPI) is an industry standard which defines a STREAMS-based interface to the Logical Link Control (LLC) 802.3 services.

The *LLA to DLPI Migration Guide* provides information about migrating LLA programs to DLPI programs.

This manual is organized as follows:

| | |
|---|---|
| Chapter 1 | **LLA to DLPI Migration** provides information about migrating programs from the HP proprietary LLA to the industry standard DLPI. |
| Chapter 2 | **LLA and DLPI Example Programs** includes example programs that compare LLA and DLPI. |

# 1 LLA to DLPI Migration

As part of Hewlett-Packard's movement toward industry standard networking, HP has discontinued the LLA/9000 product with the HP-UX 10.30 release. HP recommends that you migrate all existing applications that use LLA to the industry standard Data Link Provider Interface (DLPI). HP provides DLPI with the LAN/9000 product.

Before you begin the process of migrating your application, you may need to review the *DLPI Programmer's Guide*.

The following information explains the basic differences between LLA and DLPI. This information is the basis for performing migration.

# Device Files

Device files are used to identify the LAN driver, Ethernet/IEEE 802.3 interface card, and protocol to be used. Each LAN driver/interface card and protocol combination (Ethernet or IEEE 802.3) is associated with a device file.

A network device file is like any other HP-UX device file. When you write to a network device file after opening it, the data goes out on the network, just as when you write to a disk drive device file, the data goes out onto the disk.

By convention, device files are kept in a directory called `/dev`. When the LAN/9000 product is installed, several special device files are created. Among these files are the network device files associated with the LAN interface. If default names are used during installation, these files are called `/dev/lan0` and `/dev/ether0` for IEEE 802.3 and Ethernet, respectively.

LLA requires a separate device file for every LAN interface in the system. This device file is used by LLA to uniquely identify a specific device (e.g. `/dev/lan0`).

DLPI only requires one device file (`/dev/dlpi`) to access all supported LAN interfaces. In addition, there are other device files (`/dev/dlpiX`, where `X` is 0-100), used by DLPI, to access all supported LAN interfaces. The difference between `/dev/dlpi` and `/dev/dlpiX` is clone vs. non-cloneable devices. Basically, cloneable devices give you a separate stream for each open request.

Non-cloneable devices only give you one stream no matter how many times you open the device. All of the LAN interfaces supported by HP DLPI support both cloneable and non-cloneable access.

# ioctl Requests

All general control requests (i.e. protocol logging, destination addresses, multicast addresses, etc.) for LLA are issued via the `ioctl` system call.

The HP-UX `ioctl` call is used to construct, inspect, and control the network environment in which an LLA application will operate. All LLA applications must use the `ioctl` call to configure source and destination addresses before data can be sent or received using the HP-UX `read` and `write` calls.

`ioctl` requests are used in DLPI only for device specific control requests. These `ioctl` requests are not interpreted by DLPI, but passed directly to the driver for processing. All general control requests in DLPI are defined with a standard DLPI 2.0 primitive or extension. These primitives are passed to DLPI via the `putmsg` system call only.

All of the standard DLPI primitives are defined in `<sys/dlpi.h>`. The *DLPI Programmer's Guide* provides detailed descriptions of all the primitives. All HP DLPI extensions (denoted in the following table with an *) are defined in `<sys/dlpi_ext.h>`.

Table 1-1 lists LLA ioctl request types and their corresponding DLPI primitives.

**Table 1-1 LLA ioctls and Corresponding DLPI Primitives**

| LLA ioctl (req type) | DLPI Primitive |
|---|---|
| LOG_TYPE_FIELD | DL_BIND_REQ or DL_SUBS_BIND_REQ |
| LOG_SSAP | DL_BIND_REQ or DL_SUBS_BIND_REQ |
| LOG_DSAP | Not required with DLPI. The destination address is specified with each data request (see Transmitting data). |
| LOG_DEST_ADDR | Not required with DLPI. The destination address is specified with each data request (see Transmitting data). |
| LOG_READ_CACHE | Not defined |

| LLA ioctl (req type) | DLPI Primitive |
|---|---|
| LOG_READ_TIMEOUT | Not defined |
| LLA_SIGNAL_MASK | Not defined |
| FRAME_HEADER | Frame headers are delivered with each individual packet via the control portion of the message. |
| LOCAL_ADDRESS | DL_PHYS_ADDR_REQ |
| DEVICE_STATUS | DL_HP_HW_STATUS_REQ* |
| MULTICAST_ADDRESSES | DL_HP_MULTICAST_LIST_REQ* |
| MULTICAST_ADDR_LIST | DL_HP_MULTICAST_LIST_REQ* |
| RESET_STATISTICS | DL_HP_RESET_STATS_REQ* |
| READ_STATISTICS | DL_GET_STATISTICS_REQ. This primitive returns mib and extended mib statistics for the device in one request. |
| LOG_CONTROL | Not required with DLPI. The control value (if any) is determined from the primitive. |
| RESET_INTERFACE | DL_HP_HW_RESET_REQ* |
| ENABLE_BROADCAST | Not defined |
| DISABLE_BROADCAST | Not defined |
| ADD_MULTICAST | DL_ENABMULTI_REQ |
| DELETE_MULTICAST | DL_DISABMULTI_REQ |

# Transmitting Data

LLA requires the user to log a destination address (LOG_DEST_ADDR) and a destination service access point (LOG_DSAP) prior to sending any data.

DLPI requires the user to specify the destination address and destination service access point (dsap) as part of the data transfer request. The combination of destination MAC address and dsap is referred to as the DLSAP address.

The DLSAP address format is basically the destination MAC address followed by the LLC protocol value. A complete description of the DLSAP address format is described in the *DLPI Programmer's Guide*.

LLA supports the `write` system call for sending data requests.

DLPI only supports the `putmsg` system call for sending data over RAW (see the *DLPI Programmer's Guide*) and connectionless mode streams. The `write` system call is only supported over connection oriented streams in the DATA_XFER state (i.e. a connection must be established).

# Receiving Data

LLA does not automatically return LLC header information when packets are read by the user. The user is required to issue a separate control request (FRAME_HEADER) to get the LLC header information for the last packet received.

DLPI returns the LLC header information in the control portion of each individually received packet (i.e. DL_UNITDATA_IND, DL_XID_IND, DL_TEST_IND, etc). The user is not required to issue a separate control request to get LLC header information.

LLA only allows a maximum of 16 packets (for normal users and 64 for super users) to be queued before it starts dropping data.

DLPI will read as many packets as possible until both the stream head read queue (default is about 10k bytes) and DLPI read queue (default is about 60K bytes) fill. When both these queues are full, DLPI will begin dropping data until the queues start draining.

# 2 LLA and DLPI Example Programs

This chapter contains two example programs.

The first example shows a data transfer program using DLPI. The second example shows the same type of program using LLA for comparison.

# DLPI Example Program

```
/*********************************************************************
        (C) COPYRIGHT HEWLETT-PACKARD COMPANY 1992. ALL RIGHTS
        RESERVED. NO PART OF THIS PROGRAM MAY BE PHOTOCOPIED,
        REPRODUCED, OR TRANSLATED TO ANOTHER PROGRAM LANGUAGE WITHOUT
        THE PRIOR WRITTEN CONSENT OF HEWLETT PACKARD COMPANY
*********************************************************************/


/*********************************************************************
        The main part of this program is composed of two parts.
        The first part demonstrates data transfer over a connectionless
        stream with LLC SAP headers. The second part of this program
        demonstrates data transfer over a connectionless stream with
        LLC SNAP headers.
*********************************************************************/


#include <stdio.h>
#include <fcntl.h>
#include <memory.h>
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/dlpi.h>
#include <sys/dlpi_ext.h>

#define SEND_SAP        0x80            /* sending SAP */
#define RECV_SAP        0x82            /* receiving SAP */
#define SNAP_SAP        0xAA            /* SNAP SAP */

/*********************************************************************
        SNAP protocol values.
*********************************************************************/
u_char SEND_SNAP_SAP[5] = {0x50, 0x00, 0x00, 0x00, 0x00};
u_char RECV_SNAP_SAP[5] = {0x60, 0x00, 0x00, 0x00, 0x00};

/*********************************************************************
        global areas for sending and receiving messages
*********************************************************************/
#define AREA_SIZE        5000   /* bytes; big enough for largest possible msg */

#define LONG_AREA_SIZE  (AREA_SIZE / sizeof(u_long)) /* AREA_SIZE / 4 */

u_long  ctrl_area[LONG_AREA_SIZE];/* for control messages */
u_long  data_area[LONG_AREA_SIZE];/* for data messages */

struct strbuf ctrl_buf = {
         AREA_SIZE,      /* maxlen = AREA_SIZE */
         0,              /* len gets filled in for each message */
         ctrl_area       /* buf = control area */
};
```

```
struct strbuf data_buf = {
          AREA_SIZE,      /* maxlen = AREA_SIZE */
          0,              /* len gets filled in for each message */
          data_area       /* buf = data area */
};


/*****************************************************************
          get the next message from a stream; get_msg() returns one of the
          following defines
*****************************************************************/
#define GOT_CTRL          1        /* message has only a control part */
#define GOT_DATA          2        /* message has only a data part */
#define GOT_BOTH          3        /* message has control and data parts */

int
get_msg(fd)
          int     fd;             /* file descriptor */
{
          int     flags = 0;      /* 0 ---> get any available message */
          int     result = 0;     /* return value */
          /*
          zero first byte of control area so the caller can call check_ctrl
          without checking the get_msg return value; if only data was
          in the message and the user was expecting control or control +
          data, then when he calls check_ctrl it will compare the expected
          primitive zero and print information about the primitive
          that it got.
          */
          ctrl_area[0] = 0;

          /* call getmsg and check for an error */
          if(getmsg(fd, &ctrl_buf, &data_buf, &flags) < 0) {
                  printf("error:  getmsg failed, errno = %d\n", errno);
                  exit(1);
          }
          if(ctrl_buf.len > 0) {
                  result |= GOT_CTRL;
          }
          if(data_buf.len > 0) {
                  result |= GOT_DATA;
          }
          return(result);
}

/*****************************************************************
          check that control message is the expected message
*****************************************************************/
void
check_ctrl(ex_prim)
          int     ex_prim; /* the expected primitive */
{
          dl_error_ack_t*err_ack = (dl_error_ack_t *)ctrl_area;

          /* did we get the expected primitive? */
          if(err_ack->dl_primitive != ex_prim) {
```

```
                /* did we get a control part */
                if(ctrl_buf.len) {
                        /* yup; is it an ERROR_ACK? */
                        if(err_ack->dl_primitive == DL_ERROR_ACK) {
                                /* yup; format the ERROR_ACK info */
                                printf("error: expected primitive
                                                0x%02x, ", ex_prim);
                                printf("got DL_ERROR_ACK\n");
                                printf("   dl_error_primitive =
                                                0x%02x\n", err_ack->
                                                dl_error_primitive);
                                printf("   dl_errno = 0x%02x\n",
                                                err_ack->dl_errno);
                                printf("   dl_unix_errno = %d\n",
                                                err_ack->dl_unix_errno);
                                exit(1);
                        } else {
                                /*
                                didn't get an ERROR_ACK either; print
                                whatever primitive we did get
                                 */
                                printf("error:  expected primitive
                                                0x%02x, ", ex_prim);
                                printf("got primitive 0x%02x\n",
                                                err_ack->dl_primitive);
                                exit(1);
                        }
                  } else {
                        /* no control; did we get data? */
                        if(data_buf.len) {
                                /* tell user we only got data */
                                printf("error:  check_ctrl found only
                                                data\n");
                                exit(1);
                        } else {
                                /*
                                no message???; well, it was probably an
                                interrupted system call
                                */
                                printf("error:  check_ctrl found no
                                                message\n");
                                exit(1);
                        }
                }
           }
}

/*************************************************************
          put a message consisting of only a data part on a stream
*************************************************************/
void
put_data(fd, length)
          int     fd;          /* file descriptor */
          int     length;      /* length of data message */
{
          /* set the len field in the strbuf structure */
```

```
        data_buf.len = length;

        /* call putmsg and check for an error */
        if(putmsg(fd, 0, &data_buf, 0) < 0) {
                printf("error:  put_data putmsg failed, errno = %d\n",  errno);
                exit(1);
        }
}
/*****************************************************************
        put a message consisting of only a control part on a stream
*****************************************************************/
void
put_ctrl(fd, length, pri)
        int     fd;          /* file descriptor */
        int     length;      /* length of control message */
        int     pri;         /* priority of message: either 0 or RS_HIPRI */
{
        /* set the len field in the strbuf structure */
        ctrl_buf.len = length;

        /* call putmsg and check for an error */
        if(putmsg(fd, &ctrl_buf, 0, pri) < 0) {
                printf("error:  put_ctrl putmsg failed, errno = %d\n",
                                errno);
                exit(1);
        }
}

/*****************************************************************
        put a message consisting of both a control part and a control
        part on a stream
*****************************************************************/
void
put_both(fd, ctrl_length, data_length, pri)
        int     fd;              /* file descriptor */
        int     ctrl_length;     /* length of control part */
        int     data_length;     /* length of data part */
        int     pri;             /* priority of message: either 0
                                        or RS_HIPRI */
{
        /* set the len fields in the strbuf structures */
        ctrl_buf.len = ctrl_length;
        data_buf.len = data_length;

        /* call putmsg and check for an error */
        if(putmsg(fd, &ctrl_buf, &data_buf, pri) < 0) {
                printf("error:  put_both putmsg failed, errno = %d\n",
                                errno);
                exit(1);
        }
}
/*****************************************************************
        open the DLPI cloneable device file, get a list of available
        PPAs, and attach to the first PPA; returns a file descriptor
        for the stream
*****************************************************************/
```

```
int
attach() {
        int     fd;              /* file descriptor */
        int     ppa;             /* PPA to attach to */
        dl_hp_ppa_req_t    *ppa_req = (dl_attach_req_t *)ctrl_area;
        dl_hp_ppa_ack_t *ppa_ack = (dl_hp_ppa_ack_t *)ctrl_area;
        dl_hp_ppa_info_t *ppa_info;
        dl_attach_req_t *attach_req = (dl_attach_req_t *)ctrl_area;
        char *mac_name;

        /* open the device file */
        if((fd = open("/dev/dlpi", O_RDWR)) == -1) {
                printf("error:  open failed, errno = %d\n", errno);
                exit(1);
        }

        /*
        find a PPA to attach to;  we assume that the first PPA on the
        remote is on the same media as the first local PPA
        */
        /* send a PPA_REQ and wait for the PPA_ACK */
        ppa_req->dl_primitive = DL_HP_PPA_REQ;
        put_ctrl(fd, sizeof(dl_hp_ppa_req_t), 0);
        get_msg(fd);
        check_ctrl(DL_HP_PPA_ACK);
        /* make sure we found at least one PPA */
        if(ppa_ack->dl_length == 0) {
                printf("error:  no PPAs available\n");
                exit(1);
        }
        /* examine the first PPA */
        ppa_info = (dl_hp_ppa_info_t *)((u_char *)ctrl_area +
                         ppa_ack->dl_offset);
        ppa = ppa_info->dl_ppa;
        switch(ppa_info->dl_mac_type) {
                case DL_CSMACD:
                case DL_ETHER:
                        mac_name = "Ethernet";
                        break;
                case DL_TPR:
                        mac_name = "Token Ring";
                        break;
                case DL_FDDI:
                        mac_name = "FDDI";
                        break;
                default:
                        printf("error:  unknown MAC type in ppa_info\n");
                        exit(1);
         }
         printf("attaching to %s media on PPA %d\n", mac_name, ppa);

         /*
         fill in ATTACH_REQ with the PPA we found, send the ATTACH_REQ,
         and wait for the OK_ACK
         */
         attach_req->dl_primitive = DL_ATTACH_REQ;
```

```
          attach_req->dl_ppa = ppa;
          put_ctrl(fd, sizeof(dl_attach_req_t), 0);
          get_msg(fd);
          check_ctrl(DL_OK_ACK);

          /* return the file descriptor for the stream to the caller */
          return(fd);
}

/*****************************************************************
          bind to a sap with a specified service mode and max_conind;
          returns the local DLSAP and its length
*****************************************************************/
void
bind(fd, sap, max_conind, service_mode, dlsap, dlsap_len)
          intfd;/* file descriptor */
          intsap;/* 802.2 SAP to bind on */
          intmax_conind;/* max # connect indications to accept */
          intservice_mode;/* either DL_CODLS or DL_CLDLS */
          u_char*dlsap;/* return DLSAP */
          int*dlsap_len;/* return length of dlsap */
{
          dl_bind_req_t*   bind_req = (dl_bind_req_t *)ctrl_area;
          dl_bind_ack_t*   bind_ack = (dl_bind_ack_t *)ctrl_area;
          u_char*          dlsap_addr;

          /* fill in the BIND_REQ */
          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 = 0;/*      conn_mgmt is NOT supported */
          bind_req->dl_xidtest_flg = 0;      /* user handles TEST/XID pkts */

          /* send the BIND_REQ and wait for the OK_ACK */
          put_ctrl(fd, sizeof(dl_bind_req_t), 0);
          get_msg(fd);
          check_ctrl(DL_BIND_ACK);

          /* return the DLSAP to the caller */
          *dlsap_len = bind_ack->dl_addr_length;
          dlsap_addr = (u_char *)ctrl_area + bind_ack->dl_addr_offset;
          memcpy(dlsap, dlsap_addr, *dlsap_len);
}
/*****************************************************************
          bind to a SNAP sap via the DL_PEER_BIND, or DL_HIERARCHICAL_BIND
          subsequent bind class;  returns the local DLSAP and its length
*****************************************************************/
void
subs_bind(fd, snapsap, snapsap_len, subs_bind_class, dlsap, dlsap_len)
int       fd;
u_char*       snapsap;
int           subs_bind_class;
u_char        *dlsap;
int           *dlsap_len;
{
```

```
            dl_subs_bind_req_t *subs_bind_req = (dl_subs_bind_req_t*)ctrl_area;
            dl_subs_bind_ack_t *subs_bind_ack = (dl_subs_bind_ack_t*)ctrl_area;
            u_char  *dlsap_addr;

            /* Fill in Subsequent bind req */
            subs_bind_req->dl_primitive = DL_SUBS_BIND_REQ;
            subs_bind_req->dl_subs_sap_offset = DL_SUBS_BIND_REQ_SIZE;
            subs_bind_req->dl_subs_sap_length = snapsap_len;
            subs_bind_req->dl_subs_bind_class = subs_bind_class;
            memcpy((caddr_t)&subs_bind_req[1], snapsap, snapsap_len);

            /* send the SUBS_BIND_REQ and wait for the OK_ACK */
            put_ctrl(fd, sizeof(dl_subs_bind_req_t)+snapsap_len, 0);
            get_msg(fd);
            check_ctrl(DL_SUBS_BIND_ACK);

            /* return the DLSAP to the caller */
            *dlsap_len = subs_bind_ack->dl_subs_sap_length;
            dlsap_addr = (u_char *)ctrl_area +subs_bind_ack->dl_subs_sap_offset;
            memcpy(dlsap, dlsap_addr, *dlsap_len);

}
/*********************************************************************
            unbind, detach, and close
*********************************************************************/
void
cleanup(fd)
        int     fd;          /* file descriptor */
{
        dl_unbind_req_t*unbind_req = (dl_unbind_req_t *)ctrl_area;
        dl_detach_req_t*detach_req = (dl_detach_req_t *)ctrl_area;

        /* unbind */
        unbind_req->dl_primitive = DL_UNBIND_REQ;
        put_ctrl(fd, sizeof(dl_unbind_req_t), 0);
        get_msg(fd);
        check_ctrl(DL_OK_ACK);

        /* detach */
        detach_req->dl_primitive = DL_DETACH_REQ;
        put_ctrl(fd, sizeof(dl_detach_req_t), 0);
        get_msg(fd);
        check_ctrl(DL_OK_ACK);

        /* close */
        close(fd);
}

/*********************************************************************
        receive a data packet;
*********************************************************************/
int
recv_data(fd)
       int     fd;       /* file descriptor */
{
       dl_unitdata_ind_t *data_ind = (dl_unitdata_ind_t *)ctrl_area;
```

```
        char    *rdlsap;
        int     msg_res;

        msg_res = get_msg(fd);
        check_ctrl(DL_UNITDATA_IND);
        if(msg_res != GOT_BOTH) {
                printf("error:  did not receive data part of message\n");
                exit(1);
        }
        return(data_buf.len);
}


/*****************************************************************
        send a data packet; assumes data_area has already been filled in
*****************************************************************/
void
send_data(fd, rdlsap, rdlsap_len, len)
        int      fd;             /* file descriptor */
        u_char*  rdlsap;/* remote dlsap */
        int      rdlsap_len;/* length of rdlsap */
        int      len;/* length of the packet to send */
{
        dl_unitdata_req_t *data_req = (dl_unitdata_req_t *)ctrl_area;
        u_char*out_dlsap;


        /* fill in data_req */
        data_req->dl_primitive = DL_UNITDATA_REQ;
        data_req->dl_dest_addr_length = rdlsap_len;
        data_req->dl_dest_addr_offset = sizeof(dl_unitdata_req_t);
        /* copy dlsap */
        out_dlsap = (u_char *)ctrl_area + sizeof(dl_unitdata_req_t);
        memcpy(out_dlsap, rdlsap, rdlsap_len);

        put_both(fd, sizeof(dl_unitdata_req_t) + rdlsap_len, len, 0);
}


/*****************************************************************
        print a string followed by a DLSAP
*****************************************************************/
void
print_dlsap(string, dlsap, dlsap_len)
        char     *string;         /* label */
        u_char   *dlsap;          /* the DLSAP */
        int      dlsap_len;       /* length of dlsap */
{
        int      i;

        printf("%s", string);
        for(i = 0; i < dlsap_len; i++) {
                printf("%02x", dlsap[i]);
}
        printf("\n");
        }
```

```
/*********************************************************************
        main
*********************************************************************/
main() {
        int     send_fd, recv_fd;          /* file descriptors */
        u_char  sdlsap[20];                /* sending DLSAP */
        u_char  rdlsap[20];                /* receiving DLSAP */
        int     sdlsap_len, rdlsap_len;  /* DLSAP lengths */
        int     i, j, recv_len;

        /*
        PART 1 of program.  Demonstrate connectionless data
        transfer with LLC SAP header.
        */

        /*
        First, we must open the DLPI device file, /dev/dlpi, and attach
        to a PPA.  attach() will open /dev/dlpi, find the first PPA
        with the DL_HP_PPA_INFO primitive, and attach to that PPA.
        attach() returns the file descriptor for the stream.  Here we
        do an attach for each file descriptor.
        */
        send_fd = attach();
        recv_fd = attach();

        /*
        Now we have to bind to a IEEESAP.  We will ask for connectionless
        data link service with the DL_CLDLS service mode.  Since we are
        connectionless, we will not have any incoming connections so we
        set max_conind to 0.  bind() will return our local DLSAP and its
        length in the last two arguments we pass to it.
        */
        bind(send_fd, SEND_SAP, 0, DL_CLDLS, sdlsap, &sdlsap_len);
        bind(recv_fd, RECV_SAP, 0, DL_CLDLS, rdlsap, &rdlsap_len);

        /* print the DLSAPs we got back from the binds */
        print_dlsap("sending DLSAP   = ", sdlsap, sdlsap_len);
        print_dlsap("receiving DLSAP = ", rdlsap, rdlsap_len);

        /*
        Time to send some data.  We'll send 5 data packets in sequence.
        */
        for(i = 0; i < 5; i++) {
                /* send (i+1)*10 data bytes with the first byte = i */
                data_area[0] = i;
                /* Initialize data area */
                for (j = 1; j < (i+1)*10; j++)
                        data_area[j] = "a";
                print_dlsap("sending data to ",rdlsap, rdlsap_len);
                send_data(send_fd, rdlsap, rdlsap_len, (i + 1) * 10);
                /* receive the data packet */
                recv_len = recv_data(recv_fd);
                printf("received %d bytes, first word = %d\n", recv_len,
                                (u_int)data_area[0]);
```

```
}

/*
We're finished with PART 1.  Now call cleanup to unbind, then
detach, then close the device file.
*/
cleanup(send_fd);
cleanup(recv_fd);

/*
PART 2 of program.  Demonstrate connectionless data transfer
with LLC SNAP SAP header.
*/

/*
As demonstrated in the first part of this program we must first
open the DLPI device file, /dev/dlpi, and attach to a PPA.
*/
send_fd = attach();
recv_fd = attach();

/*
The first method for binding a SNAP protocol value (which is
demonstrated below) requires the user to first bind the SNAP
SAP 0xAA, then issue a subsequent bind with class
DL_HIERARCHICAL_BIND with the 5 bytes of SNAP information.

The second method (which is not demonstrated in this program) is
to bind any supported protocol value (see section 5) and then
issue a subsequent bind with class DL_PEER_BIND.  The data area
area of the subsequent bind should include 6 bytes of data, the
first byte being the SNAP SAP 0xAA followed by 5 bytes of SNAP
information.
*/
bind(send_fd, SNAP_SAP, 0, DL_CLDLS, sdlsap, &sdlsap_len);
bind(recv_fd, SNAP_SAP, 0, DL_CLDLS, rdlsap, &rdlsap_len);

/*
Now we must complete the binding of the SNAP protocol value
with the subsequent bind request and a subsequent bind class
of DL_HIERARCHICAL_BIND.
*/

subs_bind(send_fd, SEND_SNAP_SAP, 5, DL_HIERARCHICAL_BIND,
                                        sdlsap,&sdlsap_len);
subs_bind(recv_fd, RECV_SNAP_SAP, 5, DL_HIERARCHICAL_BIND,
                                        rdlsap,&rdlsap_len);
/* print the DLSAPs we got back from the binds */
print_dlsap("sending DLSAP   = ", sdlsap, sdlsap_len);
print_dlsap("receiving DLSAP = ", rdlsap, rdlsap_len);

/*
Time to send some data.  We'll send 5 data packets in sequence.
*/
for(i = 0; i < 5; i++) {
         /* send (i+1)*10 data bytes with the first byte = i */
```

```
                data_area[0] = i;
                /* Initialize data area */
                for (j = 1; j < (i+1)*10; j++)
                        data_area[j] = ”a”;
                print_dlsap(”sending data to ”,rdlsap, rdlsap_len);
                send_data(send_fd, rdlsap, rdlsap_len, (i + 1) * 10);
                /* receive the data packet */
                recv_len = recv_data(recv_fd);
                printf(”received %d bytes, first word = %d\n”, recv_len,
                                data_area[0]);
        }

/*
        We're finished.  Now call cleanup to unbind, then detach,
        then close the device file.
        */
        cleanup(send_fd);
        cleanup(recv_fd);

        }
```

# LLA Example Program

```
/*************************************************************************
  (C) COPYRIGHT HEWLETT-PACKARD COMPANY 1992. ALL RIGHTS
  RESERVED. NO PART OF THIS PROGRAM MAY BE PHOTOCOPIED,
  REPRODUCED, OR TRANSLATED TO ANOTHER PROGRAM LANGUAGE WITHOUT
  THE PRIOR WRITTEN CONSENT OF HEWLETT PACKARD COMPANY
*************************************************************************/


/*************************************************************************
         The main part of this program is composed of two parts.
         The first part demonstrates data transfer over LLA
         with LLC SAP headers.  The second part of this program
         demonstrates data transfer over LLA with LLC SNAP headers.
*************************************************************************/


#include <stdio.h>
#include <fcntl.h>
#include <memory.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/netio.h>

#define SEND_SAP        0x80            /* sending SAP */
#define RECV_SAP        0x82            /* receiving SAP */
#define SNAP_SAP        0xAA            /* SNAP SAP */

/*************************************************************************
        SNAP protocol values.
*************************************************************************/
u_char SEND_SNAP_SAP[5] = {0x50, 0x00, 0x00, 0x00, 0x00};
u_char RECV_SNAP_SAP[5] = {0x60, 0x00, 0x00, 0x00, 0x00};


/*************************************************************************
        global areas for sending and receiving messages
*************************************************************************/
#define MAX_PKT_SIZE     1500     /* Maximum packet size for Ethernet */

u_long   data_area[MAX_PKT_SIZE];          /* for data messages */

struct fis ctrl_buf;

/*************************************************************************
     Read a packet on LLA file descriptor fd.
*************************************************************************/
int
get_pkt(fd)
        int     fd;             /* file descriptor */
{
        int recv_cnt;
```

```
        /*
        * Read a packet from the device.
        */
        /* call read and check for an error */
        if((recv_cnt = read(fd, data_area, MAX_PKT_SIZE)) < 0) {
                printf("error:  read failed, errno = %d\n", errno);
                exit(1);
        }
        return(recv_cnt);
}

/*****************************************************************************
     Send a packet over LLA
*****************************************************************************/
void
put_data(fd, length)
     int     fd;             /* file descriptor */
     int     length;         /* length of data message */
{

     /* call putmsg and check for an error */
     if(write(fd, data_area, length) < 0) {
             printf("error:  put_data putmsg failed, errno = %d\n", errno);
             exit(1);
     }
}


/*****************************************************************************
     Send a control request to the driver.
*****************************************************************************/
void
put_ctrl(fd, cmd)
        int      fd;         /* file descriptor */
        int      cmd;        /* NETCTRL or NETSTAT */
{

        /* Send control request to driver */
        if(ioctl(fd, cmd, &ctrl_buf) < 0) {
                printf("error:  put_ctrl putmsg failed, errno = %d\n", errno);
                exit(1);

        }
}


/*****************************************************************************
     Open an LLA device.  The device file specifies which device you
     attaching to.  There is no need to issue a seperate attach control
     request to designate which device you are using.  In this example
     we will default to /dev/lan0.
*****************************************************************************
*/
int
attach() {
```

```
        intfd;              /* file descriptor */
        char *mac_name;


        /* open the device file */
        if((fd = open("/dev/lan0", O_RDWR)) == -1) {
                printf("error:  open failed, errno = %d\n", errno);
                exit(1);
        }

        /* return the file descriptor for the LLA device to the caller */
        return(fd);
}

/**************************************************************************
     Bind to a sap.  LLA does not automatically return the local MAC
     address and local sap information when binding a protocol value.
     You must explicitly request the local MAC address via the
     LOCAL_ADDRESS control request.

**************************************************************************/
void
bind(fd, sap)
        int     fd;             /* file descriptor */
        int     sap;            /* 802.2 SAP to bind on */
{

     ctrl_buf.reqtype = LOG_SSAP;
     ctrl_buf.vtype = INTEGERTYPE;
     ctrl_buf.value.i = sap;

        /* send the LOG_SSAP request.  LLA will return success or
           failure when the ioctl completes, so there is no need to
           wait for an acknowledgement.
         */
         put_ctrl(fd, NETCTRL);

}


/**************************************************************************
     Get the local MAC address.
**************************************************************************/
void
get_local_address(fd, ret_addr)
    int         fd;             /* file descriptor */
    caddr_tret_addr;/* return local address here */
{
    ctrl_buf.reqtype = LOCAL_ADDRESS;

    /* send the LOCAL_ADDRESS request.  LLA will return success or
       failure when the ioctl completes, so there is no need to
       wait for an acknowledgement.
    */
   put_ctrl(fd, NETSTAT);
```

```
   /* Copy the address to ret_addr */
   memcpy(ret_addr, (caddr_t)ctrl_buf.value.s, 6);
}

/*****************************************************************************
     Set the destination MAC and SAP address.
*****************************************************************************/
void
set_dst_address(fd, dest_addr, dsap, length)
        int     fd;                     /* file descriptor */
        caddr_t dest_addr;/* return local address here */
        int     dsap;/* destination sap */
        int     length;/* destination sap length */
{

        ctrl_buf.reqtype = LOG_DEST_ADDR;
        ctrl_buf.vtype = 6;
        memcpy((caddr_t)ctrl_buf.value.s, dest_addr, 6);
       /* send the LOG_DEST_ADDR request.  LLA will return success or
          failure when the ioctl completes, so there is no need to
          wait for an acknowledgement.
       */
       put_ctrl(fd, NETCTRL);

       /* Only log sap addresses, SNAP addresses do not need to
          be logged twice.
        */
        if (length == INTEGERTYPE) {
           ctrl_buf.reqtype = LOG_DSAP;
           ctrl_buf.vtype = INTEGERTYPE;
           ctrl_buf.value.i = dsap;
           put_ctrl(fd, NETCTRL);
        }
}


/*****************************************************************************
     bind to a SNAP sap.
*****************************************************************************/
void
bind_snap(fd, snapsap)
int     fd;
u_char  *snapsap;
{

        /* Fill in SNAP req */
        ctrl_buf.reqtype = LOG_SNAP_TYPE;
        ctrl_buf.vtype = 5;
        memcpy((caddr_t)ctrl_buf.value.s, snapsap, 5);

        /* send the SNAP request. */
        put_ctrl(fd, NETCTRL);

}
```

```
/*****************************************************************************
     Close the file descriptor.  This will automatically unbind the
     protocol.
*****************************************************************************/
void
cleanup(fd)
        int     fd;             /* file descriptor */
{

        /* close */
        close(fd);
}


/*****************************************************************************
     receive a data packet;
*****************************************************************************/
int
recv_data(fd)
        int     fd;             /* file descriptor */
{
        int     length;

        length = get_pkt(fd);
        if(length == 0) {
                printf("error:  did not receive any data part \n");
                exit(1);
        }
        return(length);
}


/*****************************************************************************
     send a data packet; assumes data_area has already been filled in
     and a destination address has already been logged.
*****************************************************************************/
void
send_data(fd, len)
        int     fd;             /* file descriptor */
        int     len;            /* length of the packet to send */
{
        put_data(fd, len);
}


/*****************************************************************************
     print a string followed by a destination MAC and SAP address.
*****************************************************************************/
void
print_dest_addr(string, dest_addr, dest_addr_len)
        char     *string;        /* label */
        u_char   *dest_addr;     /* the destination address */
```

```
        int      dest_addr_len;  /* length of dest_addr */
{
        int      i;

        printf("%s", string);
        for(i = 0; i < dest_addr_len; i++) {
                printf("%02x", dest_addr[i]);
        }
        printf("\n");
}


/*************************************************************************
     main
*************************************************************************/
main() {
       int     send_fd, recv_fd;       /* file descriptors */
       u_char  local_addr[20];         /* local MAC address */
       int     i, j, recv_len;

       /*
       PART 1 of program.  Demonstrate connectionless data transfer with
       LLC SAP header.
       */

       /*
       First, we must open the LLA device file, /dev/lan0.  LLA does
       not require a seperate control request to specify which device
       you want to use, it is explicit in the open request (via the
       device file minor number).
       */
       send_fd = attach();
       recv_fd = attach();

       /*
       Now we have to bind to a IEEESAP.  Since LLA only supports
       connectionless services there is no need to specify a specific
       service mode.  LLA also does not return the local MAC address
       automatically when binding, so we need to issue a seperate control
       request (LOCAL_ADDRESS)to get this information (see below).
       */
       bind(send_fd, SEND_SAP);
       bind(recv_fd, RECV_SAP);

       /*
       The following calls to get_local_address and set_dst_address
       are required for LLA because of one primary difference in sending
       data over LLA and DLPI.  The difference is that DLPI
       requires you to specify the destination address as part of the
       data request and LLA requires the destination address to be
       logged prior to the data request.

       Get the local MAC address so that we can send loopback packets.
       */
       get_local_address(send_fd, local_addr);
```

```
/*
Set the destination MAC and SAP address to the local address.
This will allow us to send loopback packets.
*/
set_dst_address(send_fd, local_addr, RECV_SAP, INTEGERTYPE);

/* print the MAC and SAP addresses we are sending and receiving on */
local_addr[6] = SEND_SAP;
print_dest_addr("sending too   = ", local_addr, 7);
local_addr[6] = RECV_SAP;
print_dest_addr("receiving on = ", local_addr, 7);

/*
Time to send some data.  We'll send 5 data packets in sequence.
*/
for(i = 0; i < 5; i++) {
        /* send (i+1)*10 data bytes with the first byte = i */
        data_area[0] = i;
        /* Initialize data area */
        for (j = 1; j < (i+1)*10; j++)
                data_area[j] = "a";
        print_dest_addr("sending data to ",local_addr, 7);
        send_data(send_fd, (i + 1) * 10);
        /* receive the data packet */
        recv_len = recv_data(recv_fd);
        printf("received %d bytes, first word = %d\n", recv_len,
                        (u_int)data_area[0]);
 }

 /*
 We're finished with PART 1.  Now call cleanup to close the device file.
 */
 cleanup(send_fd);
 cleanup(recv_fd);

 /*
 PART 2 of program.  Demonstrate connectionless data transfer with
 LLC SNAP SAP header.
 */

 /*
 As demonstrated in the first part of this program we must first
 open the DLPI device file, /dev/dlpi, and attach to a PPA.
 */

send_fd = attach();
recv_fd = attach();

/*
Bind the send and recv SNAP protocols.  When binding SNAP over
LLA the SNAP address will be used as both the sending and receiving
protocol address.  Therefore, there is no need to issue a seperate
request to log the destination SNAP protocol.  However, we still need
to set the destination MAC address.
*/
bind_snap(send_fd, SEND_SNAP_SAP);
```

```
        /*
        The following bind is not needed because we are running in loopback
        mode with only one LAN interface.  Since the sending LLA device
        will use the same SNAP address for sending and receiving we'll
        just loopback on the same LLA file descriptor.
        bind_snap(recv_fd, RECV_SNAP_SAP);
        */
        get_local_address(send_fd, local_addr);

        /*
        Set the destination MAC and SAP address to the local address.
        This will allow us to send loopback packets.  As mention above,
        the SNAP address does not need to be logged, it is used here
        only to distinguish SAPs and SNAP values.
        */
        set_dst_address(send_fd, local_addr, RECV_SNAP_SAP, 6);

        /* print the MAC and SAP addresses we are sending and receiving on */
        memcpy((caddr_t)&local_addr[6], SEND_SNAP_SAP, 5);
        print_dlsap("sending too   = ", local_addr, 11);
        print_dlsap("receiving on   = ", local_addr, 11);

        /*
        Time to send some data.  We'll send 5 data packets in sequence.
        */
        for(i = 0; i < 5; i++) {
                /* send (i+1)*10 data bytes with the first byte = i */
                        data_area[0] = i;
                /* Initialize data area */
                for (j = 1; j < (i+1)*10; j++)
                        data_area[j] = "a";
        print_dlsap("sending data to ",local_addr, 11);
        send_data(send_fd, (i + 1) * 10);

        /* receive the data packet.  Since we are sending
           to the SNAP address we enabled on the send_fd we
           must also receive on this file descriptor.
        */
       recv_len = recv_data(send_fd);
       printf("received %d bytes, first word = %d\n", recv_len,
                        data_area[0]);
       }

       /*
       We're finished.  Now call cleanup to then close the device file.
       */
       cleanup(send_fd);
       cleanup(recv_fd);

}
```

# Index