HP 9000 Networking

LLA Programming and Migration Guide

HP Part No. 98194-90053
Printed in U.S.A.
E0195

Edition 3

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.

This software is based in part on the Fourth Berkeley Software Distribution

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

First Edition: July 1992 (HP-UX Release 9.0)

First Edition: January 1995 (HP-UX Release 10.0)

Preface

Link Level Access for the HP 9000 (LLA/9000) is one of Hewlett-Packard's data communications and data management products. 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 Programming and Migration Guide* provides information about migrating LLA programs to DLPI programs. This guide also contains reference information for programmers who write or maintain programs that access the LAN link driver provided by Hewlett-Packard's LAN/9000 product.

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. This chapter also includes example programs that compare LLA and DLPI.

Chapter 2 **LLA Concepts** provides an overview of the LLA/9000 product.

Chapter 3 **Using LLA** explains how to use standard HP-UX file system calls to access the LAN drivers.

Chapter 4 **Network I/O Control Commands** describes the special I/O control (*ioctl*) commands provided with LLA.

Contents

Contents

Contents

Contents

1

LLA to DLPI Migration

As part of Hewlett-Packard's movement toward industry standard networking, HP will be discontinuing the LLA/9000 product following the HP-UX 10.0 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

  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.

  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>`**. All HP DLPI extensions (denoted in the following table with an *) are defined in **`<sys/dlpi_ext.h>`**. The *DLPI Programmer's Guide* provides detailed descriptions of all the primitives listed in table 1.

Table 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
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

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

LLA ioctl (req type)	DLPI Primitive
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 sap (LOG_DSAP) prior to sending any data.

  DLPI requires the user to specify the destination address and sap as part of the data transfer request. The combination of destination MAC address and destination sap 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 LLC header information

  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.

- Read cache

  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 ~10k bytes) and DLPI read queue (default is ~ 60K bytes) fill. When both these queues are full, DLPI will begin dropping data until the queues start draining.

LLA and DLPI 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.

```
/**************************************************************************
        (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)
        int     fd;             /* file descriptor */
        int     sap;            /* 802.2 SAP to bind on */
        int     max_conind;     /* max # connect indications to accept */
        int     service_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);

}
```

```
/*****************************************************************************
  (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() {
        int     fd;             /* 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_t  ret_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);

}
```

2

LLA Concepts

LLA and the OSI Model

NOTE: The information contained in this manual applies to HP 9000 Series 700 and Series 800 computer systems only.

A network architecture is a structured, modular design for networks. The Reference Model of Open Systems Interconnection (OSI) is a network architecture model developed by the International Standards Organization (ISO). HP based the development of the LAN/9000 product on the OSI model.

In the OSI model, communication tasks are assigned to seven logically distinct modules called **layers**. Each layer performs a specific data communication function. Interfaces between each layer allow each layer to communicate with the layers directly above and below it. Each layer may also communicate with its peer layer on a remote computer.

LLA (Link Level Access) allows you to access the LAN/9000 device driver at Layer 2 (Data Link Layer) in the OSI architecture. This driver controls the Ethernet/IEEE 802.3 LAN interface card at Layer 1 (Physical Layer). The portions of the LAN/9000 that implement the Ethernet and IEEE 802.3 protocols are, at Layer 2, the driver and, at Layer 1, the interface card and the remaining hardware that connects the HP 9000 computer to the LAN cable.

Because it provides access to Layer 2, LLA allows you to create applications that communicate with other vendors that also implement IEEE 802.3/Ethernet at Layer 1 and Layer 2, but that do not implement the same protocols as HP at higher layers. LLA also provides an alternative to using the other process-to-process communication services provided by the LAN/9000 product.

NOTE: Refer to the *Networking Overview* for a complete description of the OSI model. Refer to *Installing and Administering LAN/9000 Software* for a complete description of how the LAN/9000 product relates to the OSI model.

OSI Layer 2

The purpose of Layer 2 (Data Link Layer) is to provide reliable transmission of data over the physical media. Layer 2 accomplishes this by packing raw bits into **message frames** for transmission, detecting transmission errors and controlling access to the physical media. Layer 1 transmits the frames.

IEEE 802.3 and Ethernet

IEEE 802.3 is a standard data link protocol defined by the Institute of Electrical and Electronic Engineers (IEEE) and adopted by the International Standards Organization (ISO) for Layer 1 and Layer 2. IEEE 802.3 defines a baseband coaxial bus media with a media speed of 10 Megabits per second, a Media Access protocol Carrier Sense Multiple Access/Collision Detection (CSMA/CD), and the IEEE 802.2 Logical Link Control protocol.

Ethernet is a de-facto standard link level protocol that was developed before IEEE 802.3 was defined. IEEE 802.3 is a standard that evolved from Ethernet. Ethernet is not as precisely defined as IEEE 802.3, either electrically or in the frame header. Like IEEE 802.3, Ethernet also defines a baseband, coaxial, bus media, and the Media Access Method CSMA/CD.

IEEE 802.3 and Ethernet nodes can coexist on the same cable, but cannot communicate with each other.

Ethernet Frame Structure

The Ethernet packet contains the following information:

- **Preamble**. The preamble is a 64-bit (8 byte) field that contains a synchronization pattern consisting of alternating ones and zeros and ending with two consecutive ones. After synchronization is established, the preamble is used to locate the first bit of the packet. The preamble is generated by the LAN interface card.
- **Destination Address**. The destination address field is a 48-bit (6 byte) field that specifies the station or stations to which the packet should be sent. Each station examines this field to determine whether it should accept the packet.
- **Source Address**. The source address field is a 48-bit (6 byte) field that contains the unique address of the station that is transmitting the packet.
- **Type field**. The type field is 16-bit (2 byte) field that identifies the higher-level protocol associated with the packet. It is interpreted at the data link level.
- **Data Field**. The data field contains 46 to 1500 bytes. Each octet (8-bit field) contains any arbitrary sequence of values. The data field is the information received from Layer 3 (Network Layer). The information, or packet, received from Layer 3 is broken into frames of information of 46 to 1500 bytes by Layer 2.
- **CRC Field**. The Cyclic Redundancy Check (CRC) field is a 32-bit error checking field. The CRC is generated based on the destination address, type and data fields.

The packet is transmitted from the first byte of the preamble to the last byte of the CRC. Each byte is transmitted least significant bit first to most significant bit last.

Ethernet Destination Address

The destination address field in the Ethernet frame is a 48-bit (6 byte) address that contains the station address of the Ethernet/IEEE 802.3 interface card to which the packet is directed.

The first bit (Bit 1) of the destination address indicates the type of address. If it is set to zero, the field contains the unique address of one of the stations. If it is set to one, the field specifies a logical group of stations. If the address field contains all ones, the packet is broadcast to all stations.

IEEE 802.3 Frame Structure

The 802.3 packet is very similar to the Ethernet packet. It contains the following information:

- **Preamble**. The preamble field consists of seven bytes of alternating ones and zeros. After synchronization is established, the preamble is used to locate the first bit of the packet. The preamble is generated by the LAN interface card.
- **Start Frame Delimiter (SFD)**. The SFD is the 8-bit sequence 10101011 that is the same as the eighth byte of the Ethernet preamble. Together the 802.3 preamble and the SFD are identical to the Ethernet preamble.
- **Destination Address**. The 802.3 protocol gives the manufacturer the option of implementing either 16 or 48 bit addresses. HP implements the 48-bit (6 byte) address to be compatible with Ethernet's 48-bit (6 byte) address. The destination address specifies the station or stations to which a packet should be sent. Each station examines this field to determine whether or not it should accept the packet.
- **Source Address**. The source address field is a 48-bit (6 byte) field that contains the unique address of the station that is transmitting the packet.
- **Length Field**. The 2-byte length field is equal to the number of bytes in the LLC field plus the number of bytes in the pad field. If the LLC is less than 46 bytes, then the size of the pad field is 46 minus the size of the LLC. The LLC plus pad must be a minimum of 46 bytes, but no greater than 1500 bytes.
- **LLC Field**. The LLC field contains the 802.2 packet that becomes part of the 802.3 packet.
- **Pad Field**. The LLC and pad fields must be between 46 and 1500 bytes in length. If the data is not a minimum of 43 bytes, the field is padded with undefined characters or groups of bytes. The pad is automatically stripped off by the LAN interface card.
- **CRC Field**. The Cyclic Redundancy Check (CRC) field is a 32-bit error checking field. The CRC is generated based on the destination address, source address, type and data fields.

IEEE 802.3 Address Field Structures

The source and destination address fields of the IEEE 802.3 contain 48 bits (6 bytes) each. The source address is the address of the station sending the packet; the destination address is the address of the station to which the packet is directed.

The first bit (least significant bit) of the first byte of the destination address is used to distinguish between an individual and a group address. A zero indicates individual access; a one indicates group access. The second bit of the first byte distinguishes between globally and locally administered addresses. A zero indicates global and a one indicates local. All ones in the destination field indicates a broadcast address; therefore, all active stations will receive the packet.

LLC Structure

The LLC is the 802.2 packet that becomes part of the 802.3 packet. The 802.2 packet consists of four fields.

The information field is an integral number of bytes in the range of 0 to 1497. The information field, combined with the control, DSAP and SSAP fields, must be 3 to 1500 bytes. The control field is 16 bits in length when it is used for formats using sequence numbers, and 8 bits when it is used for formats not using sequence numbers. Type 1 service uses an 8-bit control field. Since HP implements Type 1, HP uses the 8-bit control field.

DSAP Address Field The DSAP field contains a Destination Service Access Point. A DSAP is a unique user-level address that identifies the higher-level protocol used on the destination machine.

The DSAP address is one byte in length. The least significant bit in the DSAP identifies whether an individual or a group of individuals should receive the packet. The remaining seven bits, or the most significant bits of the DSAP, are the address.

When the DSAP is all ones, broadcasting is enabled. An individual address indirectly identifies the higher-level protocol implemented on the destination node. Group DSAPs are reserved for future use.

SSAP Address Field The SSAP field contains a Source Service Access Point. An SSAP is a unique user-level address that identifies the higher-level protocol used on the source machine. The SSAP and the DSAP must be the same in order for two nodes to communicate.

The SSAP is one byte in length. The least significant bit of the SSAP indicates whether the packet is a command or a response. All zeroes in the SSAP indicates a null address.

Ethernet and IEEE 802.3 Packet Comparison

The two types of packets are the same through the preamble, destination and source fields. The type and length fields are also the same number of bytes in length (two bytes each). Ethernet uses the type field to convey the protocol used at higher levels; IEEE 802.3 uses the Destination Service Access Point (DSAP) for that purpose. Ethernet has no Source Service Access Point (SSAP) or control fields. Because Ethernet does not have the DSAP, SSAP or control fields, there are three extra bytes available for data.

Implementing Two Protocols

Since LLA allows implementation of both the IEEE 802.3 and Ethernet protocols, it must distinguish between the two types of packets. LLA does this by assuming that all packets are 802.2/3 packets and then checking the length field. If the value in the length field is less than 1536 bytes, the packet is processed as an 802.2/3 packet. Otherwise, the packet is assumed to be an Ethernet packet. Once this assumption is made, the length field is assumed to be the type field.

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.

This manual assumes that the LAN/9000 product has already been installed. Before you begin using LLA, you should verify that the network device files exist. If the device file directory was named **`/dev`**, use the following commands:

```
ls -l /dev/lan0
ls -l /dev/ether0
```

The following listing shows an example of the major number definition on a Series 800 computer only:

```
crw-rw-rw- 1 bin bin 50   0x000000 Jan 28 08:58 lan0
crw-rw-rw- 1 bin bin 50   0x000001 Jan 28 08:58 ether0
```

The fifth column is the major number, the sixth column is the minor number, and the final column is the name of the device file. In the previous example, the major number is 50. Bits 16 through 23 of the minor number (00 in the example) represent the instance number of LAN interface. The last bit, bit 32, specifies the protocol. A value of 1 signifies Ethernet; a value of 0 signifies IEEE 802.3. As shown in the example, a given LAN interface has one instance (in this case it is zero) but is associated with two device files: one for the Ethernet protocol and one for the IEEE 802.3 protocol.

For Series 700 computers, the major number definition is the same as on a Series 800 computer with the exception of the minor number which is bits 8 through 15. For the Series 700, the minor number for an Ethernet device file would be 0x202001. The minor number for an IEEE device file would be 0x202000.

NOTE: For complete information about LAN/9000 product installation and network device file creation, refer to *Installing and Administering LAN/9000 Software*. For complete information on device files, refer to *System Administration Tasks*.

HP-UX Calls

LLA uses six standard HP-UX file system calls to access the drivers that control the Ethernet/IEEE 802.3 interface cards:

- *open(2)*
- *close(2)*
- *read(2)*
- *write(2)*
- *select(2)*
- *ioctl(2)*

NOTE: This manual provides brief descriptions of the *open(2)*, *close(2)*, *read(2)*, *write(2)*, *select(2)*, and *ioctl(2)*calls. For complete information about these or any HP-UX calls, refer to the appropriate man page. The file system call, *fstat()*, is not supported for LAN device files. EINVAL will be returned. Use the *stat()* system call instead.

open(2) and close(2) Calls

The HP-UX *open(2)* call is used to open a device file associated with a LAN driver. The HP-UX *close(2)* command is used to close a network device file.

read(2) and write(2) Calls

The HP-UX *read(2)* call is used to read data from the network. The HP-UX *write(2)* call is used to write data out to the network.

select(2) Call

The HP-UX *select(2)* call can be used before *read(2)* or *write(2)* calls to help an application synchronize its I/O operations.

ioctl(2) Call

The HP-UX *ioctl(2)* 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(2)* call to configure source and destination addresses before data can be sent or received using the HP-UX *read(2)* and *write(2)* calls. The *ioctl(2)* call syntax that is used for LLA is described later in this chapter.

Other System Calls

The HP-UX *stat(2)* call is used to obtain information about a device file, such as the device number, access control, user ID of the file owner, and group ID of the file group. The *fstat (2)* call is not supported for LAN device files.

NETCTRL and NETSTAT Commands

LLA defines two types of network I/O control commands:

- NETCTRL commands are used to set up device-specific parameters prior to read and write operations and to reset the network I/O card and its statistical registers. There are two types of NETCTRL commands:
  - those which affect the network I/O cards, and
  - those which affect a particular connection to the network I/O card.
- NETSTAT commands are used to obtain device-dependent status and statistical information.

NETCTRL and NETSTAT commands are specified using the *ioctl(2)* command. Both types of commands are explained in chapter 3, “Using LLA,” and chapter 4, “Network I/O Control Commands.”

LLA Header File

A special C header file, **`/usr/include/netio.h`**, is provided with the LLA software. This file contains definitions of all the data structures and macros (including NETSTAT and NETCTRL) that are used to interface with LLA.

ioctl(2) Syntax

The following is a description of the *ioctl(2)* call syntax that is used for LLA. (The LLA data structures and macros used below are defined in the header file **/usr/include/netio.h**.)

```
int ioctl(fildes, request, arg)
intfildes, request;
struct fis *arg;
```

fildes Specifies on which device the *ioctl* operation is to be performed. This is the file descriptor of a successfully opened network device file.

request Specifies which type of LLA command to perform. This parameter must be either NETSTAT or NETCTRL.

arg The *arg* structure contains the address of an instance of the fis data structure. The *fis* data structure contains information necessary to perform a specific NETCTRL or NETSTAT command. The *arg* parameter must be set to the address of a *fis* structure before an *ioctl* call is made. The type of information stored in *arg* is:

```
structfis{intreqtype;
intvtype;
union{floatf;
inti;
unsignedchars[100];
}value;
};
```

reqtype Contains the name of the NETCTRL or NETSTAT command to be executed.

vtype Identifies the type of value in the value union:

vtype = INTEGERTYPE
indicates that the value is in value.i.

vtype = FLOATTYPE
indicates that the value is in value.f.

vtype = a non-negative integer ($0 \leq$ vtype ≤ 99)
indicates that the value is a character string in value.s.

This integer also specifies the length of the string.

NOTE: No LLA operations use FLOATTYPE values.

If successful, *ioctl(2)* returns a value of 0; if an error occurs, -1 is returned. Actual error values are returned to the HP-UX external variable *errno*. An *ioctl(2)* call will fail if:

- *fildes* is not a valid file descriptor.
- *request* is not appropriate for the selected device.
- *request* or *arg* are invalid.
- Resources are not available to service the request at this time.

Address Conversion Routines

LLA provides two special library routines that allow you to translate station addresses between ASCII and binary formats. These library routines, called *net_aton(3n)* and *net_ntoa(3n)*, are explained in chapter 3, “Using LLA.” Both routines are located in **`/usr/lib/libn.a`**.

LLA Error Codes

The HP-UX file system calls utilized by LLA (*open(2)*, *close(2)*, *read(2)*, *write(2)*, *select(2)*, and *ioctl(2)*) are integer functions that return -1 when an error is encountered. Actual error values are returned to the HP-UX external variable *errno*. The values for *errno* are defined in the file **`/usr/include/sys/errno.h`** and in the man page for *errno(2)*.

3

Using LLA

WARNING: **LLA is a utility for sophisticated users. Because LLA can have potentially destructive or catastrophic effects on your network, only programmers with experience with networking, the Ethernet and IEEE 802.3 protocols and I/O device drivers should use LLA.**

You must perform the following steps in order to transmit and receive data over a network using LLA:

1 Open a network device file.

2 Log a user-level address.

3 Log a destination address (this step is only required for writing data).

4 Read or write data.

5 Close the network device file.

This chapter describes the standard HP-UX file system calls and LLA NETCTRL commands that are used to perform these steps. Additional NETCTRL commands are described in chapter 4, “Network I/O Control Commands.”

NOTE: The behavior of Ethernet/IEEE 802.3 device file descriptors is similar to that of other file descriptors: multiple processes sharing a file descriptor can interfere with each other. You should be particularly aware of this when using the NETCTRL commands described in this chapter and when performing *read(2)* operations.

Step 1: Open a Network Device File

You must use the HP-UX *open(2)* call to open the network device file before performing *read(2)* and *write(2)* operations. The following is a brief description of the *open(2)* call.

```
int open(path, oflag)
char *path;
int oflag;
```

path Points to a path name that identifies the device.

oflag Constructed by using the OR symbol (‘|’desired flag options.

The *open(2)* call returns a file descriptor for the file that was opened. The only applicable option flags are the delay flag, O_NDELAY, the read only flag, O_RDONLY, and the read/write flag, O_RDWR. If O_NDELAY is set and no data is available, a *read(2)* call returns immediately. If you wish to use only the NETSTAT commands, specify the O_RDONLY flag. For other uses, you **must** specify the O_RDWR flag.

The first example below shows a device file being opened without specifying the delay flag:

```
open(“/dev/lan0”, O_RDWR);
```

The next example shows a device file being opened with the delay flag specified:

```
open(“/dev/lan0”, O_RDWR|O_NDELAY);
```

The following error values may be returned to *errno*:

- EINVAL—This value is returned if neither O_RDWR, O_RDONLY, nor O_WRONLY was specified, or if an option other than O_RDWR, O_RDONLY, O_WRONLY, or O_NDELAY was specified.
- ENXIO—This value is returned if the device specified does not exist, the device file has an invalid logical unit number or unsupported protocol.
- ENOBUFS—This value is returned if no network memory is available (not enough memory) to set up the data link structures. Refer to *Installing and Administering LAN/9000 Software* for more information about network memory.

Step 2: Log a User-Level Address

Before you can perform *read(2)* or *write(2)* operations to a network interface, you must log a user-level address. A **type field** represents a user-level address if the device is Ethernet. A **source service access point**, or **ssap**, represents a user-level address if the device is IEEE 802.3.

The following sections describe how to log a type field or a ssap using the HP-UX *ioctl(2)* call with NETCTRL commands.

For Ethernet Device

If you perform read or write operations to an Ethernet device, you must specify a user-level address by logging a type field of the Ethernet header with the driver.

LOG_TYPE_FIELD Command

To log a type field using an *ioctl(2)* call, you must specify NETCTRL in the *ioctl(2)* call's *request* parameter and initialize the *arg* parameter to contain the LOG_TYPE_FIELD command.

Initialization of *arg* for a LOG_TYPE_FIELD command is:

```
arg.reqtype = LOG_TYPE_FIELD
arg.vtype   = INTEGERTYPE
arg.value.i = type field
```

The type field is the user-level address for the network connection being established. The format of the type field is an integer in the range of 1536 to 65535. Using values outside of this range results in an EINVAL error.

A LOG_TYPE_FIELD command fails with an EBUSY error if the type field is already logged or in use by another file descriptor on the same device file.

WARNING: **DO NOT assign the following type field values, as they are reserved addresses: 2048, 2053, 2054, 32773. Using them may adversely affect operation of the HP network and will result in an EBUSY error. Other specifically reserved addresses include 4096 through 4111. These types are reserved for use by Berkeley Trailer Protocols. If your network is a multivendor network or an internetwork system, authorization to use specific type field values should be obtained from Xerox Corporation.**

Only one type field per network interface can be declared per open file descriptor. The type field cannot be changed once it is logged, and cannot be shared among other open file descriptors.

The driver uses the type field during read and write operations. The device header attached to the data on a *write(2)* call contains the type field. The *read(2)* call returns the data from a packet only if the type field on the packet header matches the logged type field.

For IEEE 802.3 Device

If you perform read or write operations to an IEEE 802.3 device, you must specify a user-level address by logging a source service access point (ssap) with the driver.

LOG_SSAP Command

To log the ssap using an *ioctl(2)* call, you must specify NETCTRL in the *ioctl(2)* call's *request* parameter and initialize the *arg* parameter to contain the LOG_SSAP command.

Initialization of *arg* for a LOG_SSAP command is:

```
arg.reqtype = LOG_SSAP
arg.vtype   = INTEGERTYPE
arg.value.i = ssap
```

The ssap is the user-level address for the network connection being established, and it must be a unique address. The format of the ssap is an **even integer** in the range of 2 to 254. Using odd values or values outside of this range will result in an EINVAL error. (Odd values are reserved by the IEEE.) Only one ssap per network interface can be declared per open file descriptor. Once an ssap has been logged, it cannot be changed without closing and reopening the device file.

NOTE: DO NOT assign the following ssap values, as they are reserved addresses: **6, 252, 248**. Using them will adversely affect operation of the HP network.

LOG_SSAP fails with an EBUSY error if the ssap value is already logged or in use by another file descriptor on the same device file.

LOG_DSAP Command

The dsap is the user address of the remote protocol with which communication is desired. The driver uses the ssap/dsap fields in read and write operations. The link level header attached to the data on a *write(2)* call contains the ssap/dsap values. *read(2)* calls will return the data from a packet only if the dsap value on the packet header of incoming IEEE 802.3 packets matches the logged ssap value.

Unlike the ssap, which cannot be changed without closing and reopening the device file, a dsap can be changed as often as necessary. If you want to change the dsap, you must execute a LOG_DSAP command.

To log a dsap using an *ioctl(2)* call, you must specify NETCTRL in the *ioctl(2)* call's *request* parameter and initialize the *arg* parameter to contain the LOG_DSAP command.

Initialization of *arg* for a LOG_DSAP command is:

```
arg.reqtype = LOG_DSAP
arg.vtype   = INTEGERTYPE
arg.value.i = dsap
```

The format of the dsap field follows the same conventions and restrictions described above for the ssap field, although odd dsaps and a dsap of zero may be logged. The dsap value can be changed as many times as necessary. LOG_DSAP must be executed after the LOG_SSAP operation.

Step 3: Log a Destination Address

Before writing to a network device, a destination address should be declared. This is done using an HP-UX *ioctl(2)* call.

LOG_DEST_ADDR Command

To declare a destination address using an *ioctl(2)* call, you must specify NETCTRL in the *ioctl(2)* call's *request* parameter and initialize the *arg* parameter to contain the LOG_DEST_ADDR command.

Initialization of *arg* for the LOG_DEST_ADDR command is:

```
arg.reqtype = LOG_DEST_ADDR
arg.vtype   = length of arg.value.s = 6
arg.value.s = destination address
```

The destination address is the **station address**, in binary form, of the remote Ethernet/IEEE802.3 device that is to receive the data. The device header attached to the data packets on *write(2)* calls contains the destination address. LOG_DEST_ADDR can be called as often as necessary.

A station address (also referred to as an Ethernet address, LAN address, IEEE 802.3 address or network station address) is a link-level address that is the unique address of an Ethernet/IEEE 802.3 interface card. This value is set at the factory and cannot be changed. To find out what the station address is for a particular card, you can run the *lanscan(1M)* command or refer to the Network Map for your network. Since the LOG_DEST_ADDR requires that you specify the station address in binary form, you must convert the hexadecimal address before executing this command. LLA provides two address conversion routines for this purpose.

Address Conversion

Two address conversion routines, *net_aton(3n)* and *net_ntoa(3n)*, are provided to help you translate station addresses between hexadecimal, octal or decimal and binary formats. The *net_aton(3n)* library routine converts a hexadecimal, octal or decimal address to a binary address; the *net_ntoa(3n)* library routine converts a binary address to an ASCII hexadecimal address. Both routines are provided in **`/usr/lib/libc.a`**.

net_aton(3n)

The *net_aton(3n)* routine converts an Ethernet or IEEE 802.3 station address to binary form. The function is:

```
char *net_aton(dstr, sstr, size)
char *dstr;
char *sstr;
int size;
```

dstr Pointer to the binary address returned by the function.

sstr Pointer to a null-terminated ASCII form of a station address (Ethernet or IEEE 802.3). This address may be an octal, decimal or hexadecimal number as used in the C language. In other words, a leading 0x or 0X implies hexadecimal; a leading 0 implies octal. Otherwise, the number is interpreted as decimal.

size Length of the binary address to be returned in **`dstr`**. The length is 6 for Ethernet/IEEE 802.3 addresses.

A NULL value is returned if any error occurs, otherwise **`dstr`** is returned.

net_ntoa(3n)

The *net_ntoa(3n)* routine converts a 48-bit binary address to its ASCII hexadecimal equivalent. The function is:

```
char *net_ntoa(dstr, sstr, size)
char *dstr;
char *sstr;
int size;
```

dstr Pointer to the ASCII hexadecimal address returned by the function. **`dstr`** is null-terminated and padded with leading zeroes if necessary. **`dstr`** must be at least (2 * size + 3) bytes long to accommodate the size of the converted address.

sstr Pointer to a station address in its binary form.

size Length of **`sstr`**.

A NULL value is returned if any error occurs, otherwise **`dstr`** is returned.

Step 4: Read or Write Data

You must use the HP-UX *read(2)* call to read data from the network. You must use the HP-UX *write(2)* call to send data out to the network.

NOTE: Before attempting to read or write data, you must declare a user-level address. Before attempting to write data, you must declare a destination address. These tasks are described earlier in steps 2 and 3. An attempt to read or write data without having logged a user-level address or an attempt to write data prior to logging a destination address will return the error EDESTADDRREQ.

Reading Data

The following is a brief description of the HP-UX *read(2)* call.

```
int read(fildes, buf, nbytes)
int fildes;
char *buf;
unsigned nbytes;
```

fildes Specifies which device the data is to be read from. *read* fails if **`fildes`** is not a valid file descriptor.

buf Buffer into which data read from the network is placed.

nbytes **`nbytes`** should be greater than or equal to zero. A negative number returns a -1 with EINVAL in the *errno* variable. Maximum number of bytes of data to be read.

Upon successful completion, *read(2)* returns the number of bytes actually read and placed in the buffer. If an error occurs, *read(2)* returns a -1. If a packet (the data message and its Ethernet/IEEE 802.3 header) is not immediately available, the process is blocked until a packet with the proper user-level address (specified by LOG_TYPE_FIELD for Ethernet and by LOG_SSAP for IEEE 802.3) arrives, or until a timeout occurs (EIO is returned on timeout). However, if the O_NDELAY flag is set, the process is NOT blocked, but returns -1 with EWOULDBLOCK in the *errno* variable.

Blocked read operations will terminate upon delivery of signals to the calling process, and the error EINTR is returned to the process.

Read and write operations may only address a single packet of data appropriate for the protocol being used.

The link level frame header is not returned with the read, only user data will be placed in the user's buffer. The frame header for the last read packet may be obtained with the *ioctl* NETSTAT FRAME_HEADER call.

The **maximum number of data bytes** that can be transferred per *read(2)* call is:

- 1500 bytes for Ethernet.
- 1497 bytes for IEEE 802.3.

The **minimum number of data bytes** that can be transferred per *read(2)* call is:

- 46 data bytes for Ethernet.
- 0 data bytes for IEEE 802.3.

NOTE: A packet is truncated to fit in the user buffer if the allocated buffer (**`buf`**) is too small. Since the packet size is usually not known before it is received, it is recommended that you always use a buffer size of 1500 bytes when reading.

A received data packet cannot be less than the minimum data packet size because the sending node pads such packets. For IEEE 802.3, the receiving node detects and strips off any padding characters. They are not stripped from Ethernet packets. The actual data delivered is equal to or less than the user buffer size. If the received data packet is greater than the user-specified buffer size, then the actual data delivered will be truncated. The user program should compare the amount of bytes read with the amount requested.

Padded characters are not stripped off by the Ethernet drivers. Usually, the user program is expecting data to always be a certain size and can ignore the padded characters.

For example:

- User buffer is 1400 bytes.
- Minimum number is 46 data bytes for Ethernet and 0 data bytes for

IEEE 802.3.
- Inbound packet contains 40 data bytes.
- For IEEE 802.3, 40 bytes are returned.
- For Ethernet, 46 bytes (40 + 6 pad characters) are returned.

NOTE: The LAN drivers do not guarantee data delivery. On a successful *write(2)*, the only guarantee is that the data has been queued for transmission by the LAN interface card. Likewise, there is no guarantee that, once transmitted, data will be received by the target computer. The desired degree of reliability must be coded into your program using acknowledgment or sequencing algorithms.

Managing the Packet Receive Cache

By default, only one packet received for an active type field or destination sap (dsap) is cached prior to a read of the associated file descriptor. Subsequent packets received for that file descriptor are discarded. This one-packet cache may be suitable for request/reply protocols, but may not be suitable for applications that communicate with more than one host or where windowing protocols are used. The NETCTRL command LOG_READ_CACHE can be used to increase the receive caching for up to 16 packets for normal users and 64 packets for super users.

The following section describes how to specify the LOG_READ_CACHE command using the *ioctl(2)* call.

LOG_READ_CACHE Command To alter the read cache, you must specify NETCTRL in the *ioctl(2)* call's *request* parameter and initialize the *arg* parameter to contain the LOG_READ_CACHE command.

Initialization of *arg* for the LOG_READ_CACHE command is:

```
arg.reqtype = LOG_READ_CACHE
arg.vtype   = INTEGERTYPE
arg.value.i = number of packets £ 16 (normal user) or
              64 (super user) to be added to cache
```

If you assign arg.value.i a value greater than 16 (64, super user), it is interpreted as 16 (64, super user). LOG_READ_CACHE returns an ENOBUFS error to *errno* if the requested memory is unavailable.

Altering the I/O Timeout Interval

The default timeout value for *read(2)* is zero. A timeout value of zero causes an executing *read(2)* operation to be blocked indefinitely until data is available. The NETCTRL command LOG_READ_TIMEOUT is provided to set the timeout value for read operations.

The following section describes how to specify the LOG_READ_TIMEOUT command using the *ioctl(2)* call.

LOG_READ_TIMEOUT Command To alter the I/O timeout interval using an *ioctl(2)* call, you must specify NETCTRL in the *ioctl(2)* call's *request* parameter and initialize the *arg* parameter to contain the LOG_READ_TIMEOUT command.

Initialization of *arg* for the LOG_READ_TIMEOUT command is:

```
arg.reqtype = LOG_READ_TIMEOUT
arg.vtype   = INTEGERTYPE
arg.value.i = read timeout value in milliseconds
```

A positive timeout value causes a *read(2)* to fail if no data is available and the specified time has elapsed. If a read timeout occurs, read will return a -1 with EIO placed in *errno*. A negative timeout value will fail with EINVAL returned. The *read(2)* option O_NDELAY overrides the timeout mechanism; if data is not immediately available, a *read(2)* returns a -1 with an EWOULDBOCK error in *errno* immediately.

NOTE: Due to race conditions caused by asynchronous interrupts, the accuracy of the timer is guaranteed only to the extent that it does not timeout sooner than the assigned value.

Writing Data

The following is a brief description of the HP-UX *write(2)* call.

```
int write(fildes, buf, nbytes)
int fildes;
char *buf;
unsigned nbytes;
```

fildes Specifies which device the data is to be written to. A *write(2)* call fails if fildes is not a valid file descriptor.

buf Pointer to a buffer that holds the data to be written.

nbytes Number of bytes of data to be written.

Upon successful completion, *write(2)* returns the number of bytes actually written. If an error occurs, *write(2)* returns a -1. The *write(2)* call transfers packets to an internal transmit queue, from which they are sent out on the network. If a write is performed when the transmit queue is exhausted or if network memory allocated to this connection is insufficient to handle the write request, ENOBUFS is returned.

Read and write operations can only address a single packet of data appropriate for the protocol being used.

The **maximum number of data bytes** that can be transferred per *write(2)* call is:

- 1500 bytes for Ethernet.
- 1497 bytes for IEEE 802.3.

The **minimum number of data byte**s that can be transferred per *write(2)* call is:

- 46 data bytes for Ethernet.
- 0 data bytes for IEEE 802.3.

If a *write(2)* packet is smaller than the minimum size, it is padded with undefined characters. These are removed by a receiving IEEE 802.3 driver, but not by a receiving Ethernet driver. If a *write(2)* packet is greater than the maximum number of bytes, 0 bytes are written, and the error EMSGSIZE is returned.

NOTE: The network drivers do not guarantee data delivery. On a successful *write(2)*, the only guarantee is that the data has been queued for transmission by the LAN interface card. Likewise, there is no guarantee that, once transmitted, data will be received by the target computer. The desired degree of reliability must be coded into your program using acknowledgment or sequencing algorithms.

Synchronizing I/O Operations

You can use the HP-UX *select(2)* call before performing *read(2)* or *write(2)* operations to help an application synchronize its I/O operations. *select(2)* is not supported for exceptional conditions. The following is a brief description of the *select(2)* call.

```
int select (nfds, readfds, writefds,
execptfds, timeout)
int nfds, *readfds, *writefds, *execptfds;
struct timeval *timeout;
```

nfds Specifies the maximum number of file descriptors for which to check.

readfds Pointer to a bit-mapped integer that specifies which file descriptors are to be checked for reading.

writefds Pointer to a bit-mapped integer that specifies which file descriptors are to be checked for writing.

execptfds File descriptor for pending exceptional conditions. **This not supported by LLA.** Use a value of 0 for the bit which refers to the network device.

timeout If a non-zero pointer, this parameter specifies a maximum interval to wait for the selection to complete. If it is a zero pointer, the *select(2)* waits until an event causes one of the masks to be returned with a valid (non-zero) value.

A *select(2)* call returns on a *read(2)* operation when a packet is available for the correct user-level address. The *select(2)* call returns on a *write(2)* operation when there is room for the packet in the transmit queue.

Because *select(2)* does not reserve resources, it does not guarantee uninterrupted completion of a subsequent I/O operation.

Setting Up Asynchronous Signals

As a companion to *select(2)*, the user may set up a file descriptor to receive signals asynchronously. This is done with the *ioctl(2)* command, using the NETCTRL request type LLA_SIGNAL_MASK. If this mask is set to LLA_PKT_RECV, a SIGIO signal is generated on the user process when a packet arrives for a file descriptor associated with that process. If the mask is set to LLA_Q_OVERFLOW, a SIGIO signal is generated on the user process when the inbound queue for an associated file descriptor overflows, which causes a packet to be dropped. These two options may be combined in the mask, so the SIGIO signal is generated by either condition. If signals are used with more than one LLA file descriptor, *select(2)* may be used to help determine which file descriptor generated the signal.

The NETCTRL command LLA_SIGNAL_MASK is provided to allow the user to request the generation of a SIGIO signal to the user process upon certain events.

LLA_SIGNAL_MASK Command

Initialization of *arg* for the LLA_SIGNAL_MASK command is:

```
arg.reqtype = LLA_SIGNAL_MASK
arg.vtype   = INTEGERTYPE
arg.value.i = LLA_NO_SIGNAL      Do not generate any signals
                                 (default).
              LLA_PKT_RECV       SIGIO generated when packet
                                 has arrived on queue.
              LLA_Q_OVERFLOW     SIGIO generated when inbound
                                 queue has overflowed, results
                                 in a dropped packet.
```

If signal disabling is desired, set **`value.i`** to LLA_NO_SIGNAL:

```
arg.value.i = LLA_NO_SIGNAL
```

If one of, but not both of LLA_PKT_RECV or LLA_Q_OVERFLOW is desired, assign the appropriate value to **`value.i`**:

```
arg.value.i = LLA_PKT_RECV
```

or

```
arg.value.i = LLA_Q_OVERFLOW
```

If both LLA_PKT_RECV and LLA_Q_OVERFLOW are desired, OR the values together:

```
arg.value.i = LLA_Q_OVERFLOW | LLA_Q_OVERFLOW
```

The only case in which a signal will not be generated despite the appropriate event occurring is if the process is already blocked on a read to the LLA connection.

NOTE: Combining mask values results in an ambiguous cause of a received signal, since it could be generated either by the arrival of a packet or by inbound queue overflow. Also, the driver will only signal the process which last configured the LLA_SIGNAL_MASK. Processes that share file descriptors can potentially interfere with the intended use of LLA SIGIO.

Step 5: Close the Network Device File

You must use the HP-UX *close(2)* call to close a network device file. The following is a brief description of *close(2)* call.

```
int close(fildes)
int fildes;
```

fildes Specifies which Ethernet/IEEE802.3 device file is to be closed.

The operation fails if **`fildes`** is not a valid open file descriptor.

4

Network I/O Control Commands

This chapter describes the NETCTRL and NETSTAT commands provided by LLA to perform the following activities:

- Collect and Reset Interface Statistics.
- Manage Network Addresses.
- Reset an Interface.
- Manage Broadcast Packets.
- Manage Multicast Packets.

The commands described in this chapter are organized according to these activities. All of these activities are accomplished using the standard HP-UX *ioctl(2)* call.

The NETCRTL and NETSTAT commands may be executed anytime after you have successfully opened an LLA device file.

Collecting and Resetting Interface Statistics

Commands are provided for collecting and resetting interface statistics. The following commands are used as NETSTAT commands only.

- FRAME_HEADER.
- LOCAL_ADDRESS.
- DEVICE_STATUS.
- MULTICAST_ADDRESSES.
- MULTICAST_ADDR_LIST.

Several other commands, referred to as **Reset and Read Statistics Commands**, can be used as either NETCTRL or NETSTAT *ioctl(2)* commands. The meaning of each of these commands is different depending on which *request* value (NETCTRL or NETSTAT) is used.

FRAME_HEADER Command

This command returns the Ethernet/IEEE 802.3 device header associated with the last *read(2)* call. The header contains the target computer's station address (the destination address), the transmitting computer's station address (the source address), and the user-level address.

NOTE: The FRAME_HEADER command returns unpredictable information if there has not been a previous *read(2)*.

Initialization of *arg* for an Ethernet FRAME_HEADER command is:

```
arg.reqtype = FRAME_HEADER
```

FRAME_HEADER returns:

```
arg.vtype = 14
```

`arg.value.s =` `s[0] to s[5] =` destination address

The destination address is the sender's destination address, which could be the local device's station address, a multicast address or the broadcast address.

s[6] to s[11] = source address

The source address is the station address of the sender's device.

s[12] to s[13] = type field

The type field is the user-level address, specified as a 2 byte unsigned integer.

Initialization of *arg* for an IEEE802.3 FRAME_HEADER command is:

```
arg.reqtype = FRAME_HEADER
```

FRAME_HEADER returns:

```
arg.vtype   = 17
arg.value.s =
```

s[0] to s[5] = destination address

s[6] to s[11] = source address

s[12] to s[13] = received packet's length, including data, dsap/ssap and control field

s[14] = dsap value

s[15] = ssap value

s[16] = control field value

Use the *net_ntoa(3n)* routine to convert the returned destination addresses to ASCII form.

LOCAL_ADDRESS Command

This command returns the station address of the local Ethernet/IEEE 802.3 device.

Initialization of *arg* for the LOCAL_ADDRESS command is:

```
arg.reqtype = LOCAL_ADDRESS
```

LOCAL_ADDRESS returns:

```
arg.vtype   = 6
arg.value.s  = local station address
```

If necessary, use the *net_ntoa(3n)* routine to convert the returned address to ASCII form.

DEVICE_STATUS Command

This command returns the value of the current status of the local Ethernet/IEEE 802.3 device.

Initialization of *arg* for the DEVICE_STATUS command is:

```
arg.reqtype = DEVICE_STATUS
```

DEVICE_STATUS returns:

```
arg.vtype   = INTEGERTYPE
arg.value.i = INACTIVE
              INITIALIZING
              ACTIVE
              FAILED
```

The constants returned to **arg.value.i** are defined in the LLA header file **/usr/include/netio.h**. These constants have the following meanings:

- **INACTIVE**—the driver is “alive” but not currently active.
- **INITIALIZING**—the driver is processing an initialization request.
- **ACTIVE**—the driver is “alive,” and a request is active on the card.
- **FAILED**—the driver is in a “dead” state. A reset is required.

MULTICAST_ ADDRESSES Command

This command returns the current number of accepted multicast addresses.

Initialization of *arg* for the MULTICAST_ADDRESSES command is:

```
arg.reqtype = MULTICAST_ADDRESSES
```

MULTICAST_ADDRESSES returns:

```
arg.vtype   = INTEGERTYPE
arg.value.i = number of multicast addresses
```

MULTICAST_ADDR_LIST Command

This command returns the current list of accepted multicast addresses.

Initialization of *arg* for the MULTICAST_ADDR_LIST command is:

```
arg.reqtype = MULTICAST_ADDR_LIST
```

MULTICAST_ADDR_LIST returns:

```
arg.vtype   = length of arg.value.s
arg.value.s = list of multicast addresses
```

The value in **arg.vtype** represents the number of bytes used for the contiguous address list in **arg.value.s**. Each address is six bytes long. The maximum number of bytes that can be returned is 96.

RESET_STATISTICS Command

The RESET_STATISTICS command can be used as a NETCTRL *ioctl(2)* command to reset all interface statistics that are kept by the interface card. When request equals NETCTRL and **arg.reqtype** is RESET_STATISTICS, all statistics counters are reset to zero. No operands are necessary. The NETCTRL reset statistics command requires **super-user** capability.

An unrecognized request type will return an *errno* value of EINVAL. A NETCTRL request without super-user capability will return the error EPERM.

READ_STATISTICS Command

When *request* equals NETSTAT, the current value of the statistic specified in **arg.reqtype** is returned.

The value returned from a statistics counter represents the value since the last reset of that counter. The value of the statistic applies to the device, as opposed to an open file descriptor associated with the device. The result is returned in the appropriate field of the **arg.value** union.

An unrecognized request type will return an *errno* value of EINVAL.

Interface Statistics

The following NETSTAT commands are used to collect interface statistics that are kept by the interface card.

RESET_STATISTICS	NETSTAT: Not applicable. Will return EINVAL if used.
RX_FRAME_COUNT	NETSTAT: Returns the number of packets received without error.
TX_FRAME_COUNT	NETSTAT: Returns the number of packets transmitted without error.
UNTRANS_FRAMES	NETSTAT: Returns the number of packets that, due to some error, could not be transmitted.
UNDEL_RX_FRAMES	NETSTAT: Returns the number of packets which were received, but due to some error, could not be delivered to an appropriate network connection.
RX_BAD_CRC_FRAMES	NETSTAT: Returns the number of packets received with a bad CRC.
NO_HEARTBEAT	This is a hardware-dependent statistic that indicates problems with the Medium Attachment Unit (MAU) cabling. NETSTAT: Returns the number of transmit packets for which no heartbeat was detected.
MISSED_FRAMES	NETSTAT: Returns the number of times that the card missed packets due to lack of resources.
ALIGNMENT_ERRORS	NETSTAT: Returns the number of packets received with an alignment error and a bad CRC. NOTE: These packets are also counted by the RX_BAD_CRC_FRAMES counter.
DEFERRED	NETSTAT: Returns the number of packets that had to defer before transmission.
ONE_COLLISION	NETSTAT: Returns the number of transmissions completed with one collision.
MORE_COLLISIONS	NETSTAT: Returns the number of transmissions completed with more than one collision.

LATE_COLLISIONS	NETSTAT: Returns the number of transmit packets for which the card detected a late collision.
EXCESS_RETRIES	NETSTAT: Returns the number of packets that were not transmitted due to an excessive number of retries (16 or more).
CARRIER_LOST	NETSTAT: Returns the number of transmit packets that failed due to the loss of the carrier. This is a hardware-dependent statistic that indicates problems with the Medium Attachment Unit (MAU) cabling.
BAD_CONTROL_FIELD	NETSTAT: Returns the number of IEEE 802.3 packets received with an invalid control field.
UNKNOWN_PROTOCOL	NETSTAT: Returns the number of packets dropped because the type field or dsap referenced an unknown protocol.
TDR	NETSTAT returns the time (in bit times) from when a frame started to transmit until a collision occurred. This statistic can be useful for grossly determining where on the cable a problem is located. This statistic is not updated after an external loopback frame is transmitted.
RX_XID	NETSTAT: Returns the number of IEEE 802.3 XID packets that were received.
RX_TEST	NETSTAT: Returns the number of IEEE 802.3 TEST packets that were received.
RX_SPECIAL_DROPPED	NETSTAT: Returns the number of IEEE 802.3 XID or TEST packets that were received but not responded to due to lack of resources.
ILLEGAL_FRAME_SIZE	NETSTAT: Returns the numbers of times the card received and discarded packets that were illegal in size (greater than 1514 bytes). Not supported on Series 700.
NO_TX_SPACE	NETSTAT: Returns the number of times that the card exhausted its transmit buffer space. Not supported on Series 700 or Model 8x7S systems.

LITTLE_RX_SPACE	NETSTAT: Returns the number of times the card had one or no buffers to accept incoming packets. Not supported on Series 700 or Model 8x7S systems.

Managing Network Addresses

Five NETCTRL commands are provided to manage network addresses. These commands are:

- LOG_TYPE_FIELD—(Ethernet) Log type field of the Ethernet header.
- LOG_SSAP—(IEEE 802.3) Log source service access point.
- LOG_DEST_ADDR—(Ethernet or IEEE 802.3) Log destination network station address.
- LOG_DSAP—(IEEE 802.3) Change destination service access point.
- LOG_CONTROL—(IEEE 802.3; requires **super-user** capability) Override Unnumbered Information control field of IEEE 802.3 header.

The first four commands, LOG_TYPE_FIELD, LOG_SSAP, LOG_DEST_ADDR, and LOG_DSAP, are described in chapter 3, “Using LLA.” Refer to that chapter for information on these commands. The remaining command, LOG_CONTROL, is described below.

NOTE: The LOG_CONTROL command is **only applicable to the IEEE 802.3 protocol** and conforms to its specification. Refer to the IEEE 802.3 specification for detailed information about the UI, XID and TEST control fields mentioned below.

LOG_CONTROL Command

You can call LOG_CONTROL after you have logged a ssap. The Unnumbered Information (UI) control field of the IEEE 802.3 header is the default used for normal communication. With super-user capability, you can override this default with XID_CONTROL or TEST_CONTROL.

- XID control field: Any data written to the network device is ignored. An XID request packet is transmitted instead, and any network responses will be returned through a subsequent *read(2)* call.
- TEST control field: Data written to the network device causes a TEST packet containing the data to be transmitted. Any network responses will be returned through a subsequent *read(2)* call.

Initialization of *arg* for the LOG_CONTROL command is:

arg.reqtype = LOG_CONTROL

arg.vtype = INTEGERTYPE

arg.value.i = UI_CONTROL for normal data frame (default) = 3
XID_CONTROL for XID frame = 0xBF
TEST_CONTROL for TEST frame = 0xF3

Resetting an Interface

The NETCTRL command RESET_INTERFACE is provided to reset the Ethernet/IEEE 802.3 device. This command forces a complete hardware self-test. It also resets all interface statistics counters. The RESET_INTERFACE command requires **super-user** capability.

NOTE: A reset can drop packets or impair any currently active network connections at the local computer.

RESET_INTERFACE Command

Initialization of *arg* for the RESET_INTERFACE command is:

```
arg.reqtype = RESET_INTERFACE
```

Managing Broadcast Packets

Two NETCTRL commands, ENABLE_BROADCAST and DISABLE_BROADCAST, are provided to control the reception of broadcast packets. Broadcast packets are packets with the destination address field containing all 1s. These commands require **super-user** capability.

ENABLE_ BROADCAST Command

ENABLE_BROADCAST allows broadcast packets to be received by the local network device.

Initialization of *arg* for the ENABLE_BROADCAST command is:

```
arg.reqtype = ENABLE_BROADCAST
```

DISABLE_ BROADCAST Command

DISABLE_BROADCAST prohibits broadcast packets from being received.

CAUTION: Use of the DISABLE_BROADCAST command may be catastrophic to an active HP network.

Initialization of *arg* for the DISABLE_BROADCAST command is:

```
arg.reqtype = DISABLE_BROADCAST
```

Managing Multicast Packets

Two NETCTRL commands, ADD_MULTICAST and DELETE_MULTICAST, are provided to control multicast packets. Both commands require **super-user** capability.

ADD_MULTICAST Command

The ADD_MULTICAST command adds the multicast address specified in **`arg.value.s`** to the device's list of accepted multicast addresses. This multicast address list is maintained inside the LAN card. If a packet is received with a multicast destination address, this address is compared to the receiving device's current list. If the address is not in the list, the packet is discarded. This operation is performed by the LAN card, not by the device driver.

Initialization of *arg* for the ADD_MULTICAST command is:

```
arg.reqtype  = ADD_MULTICAST
arg.vtype    = length of arg.value.s = 6
arg.value.s  = multicast address
```

A multicast address is defined by the user and is not tied to the physical station address of a computer. After such address is defined, any node in the network that has added this address to its device multicast address list (by issuing the ADD_MULTICAST command) will receive any packet with its destination field equal to this multicast address. A valid multicast address is a 48-bit value with the least significant bit turned on to indicate a group address. Up to 16 multicast addresses can be supported simultaneously.

The following errors can be returned:

- EPERM—Indicates that the application is not running under super-user capabilities.
- EINVAL—Indicates that the multicast list is full; an improper address size was used; the group address bit was not set (not a multicast address); or the specified address is already in the list.

DELETE_MULTICAST Command

The DELETE_MULTICAST command removes the multicast address specified in **`arg.value.s`** from the device's current list of accepted multicast addresses.

Initialization of *arg* for the DELETE_MULTICAST command is:

```
arg.reqtype  = DELETE_MULTICAST
arg.vtype    = length of arg.value.s = 6
arg.value.s  = multicast address
```

CAUTION: Deletion of an HP special multicast address may be catastrophic to an active HP network. These addresses are: 0x090009000001, 0x090009000002.

A valid multicast address is a 48-bit value with the least significant bit turned on to indicate a group address.

The following errors can be returned:

- EPERM—Indicates that the application is not running under super-user capabilities.
- EINVAL—Indicates that the multicast list is empty; an improper address size was specified; the group address bit was not set (not a multicast address); or the specified address is not in the list.

You can use *net_aton(3n)* to translate the ASCII form of the multicast address into its network-internal form.

Index

Index

Index