HEWLETT PACKARD

HP 9000 Networking

LLA Programmer's Guide

HP 9000 Networking

LLA Programmer's Guide

HEWLETT PACKARD

Edition 2
E0792

98194-60534
Printed in U.S.A. 07/92

Preface

Link Level Access for the HP 9000 (LLA/9000) is one of Hewlett-Packard's data communications and data management products.

The *LLA Programmer's Guide* is the primary reference manual 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 Concepts," provides an overview to the LLA/9000 product.
Chapter 2	"Using LLA," explains how to use standard HP-UX file system calls to access the LAN drivers.
Chapter 3	"Network I/O Control Commands," describes the special I/O control (*ioctl*) commands provided with LLA.
Chapter 4	"LLA Examples," provides LLA programming examples.
Appendix A	"Implementation Differences," lists and explains the differences between the HP 9000 Series 300/400 and HP 9000 Series 600/700/800 LLA products.
Appendix B	"LLA Layer 2 Protocols," contains diagrams and text that explain Ethernet and IEEE 802.3 protocol components.
Appendix C	"Error Messages," lists and describes the error messages produced by Link Level Access.

Documentation Map

The following documentation map lists the manuals containing information related to the product described in this manual. You may need information from one or all of these manuals.

Installing and Administering LAN/9000

Installing and Administering Network Services

Administering ARPA Services

Installing and Administering NFS Services

NetIPC Programmer's Guide

Berkeley IPC Programmer's Guide

For more information on Ethernet:

The Ethernet, A LAN: Data Link Layer and Physical Layer Specification, Version 2.0, November 1982, Digital Equipment Corporation, Intel Corporation, Xerox Corporation

For more information on IEEE 802.3:

CSMA/CD Access Method and Physical Specification, October 1984, Institute of Electrical and Electronic Engineers

Contents

Chapter 3 Network I/O Control Commands

Chapter 4 LLA Examples

Appendix A Implementation Differences

Appendix B LLA Layer 2 Protocols

Appendix C Error Messages

Figures

Tables

1

LLA Concepts

LLA and the OSI Model

Note The information contained in this manual applies to HP 9000 Series 300/400/700 and Series 600/800 computer systems. Any differences in installation, configuration, or operation are specifically noted.

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. Figure 1-1 shows how the LLA/9000 product relates to the OSI model.

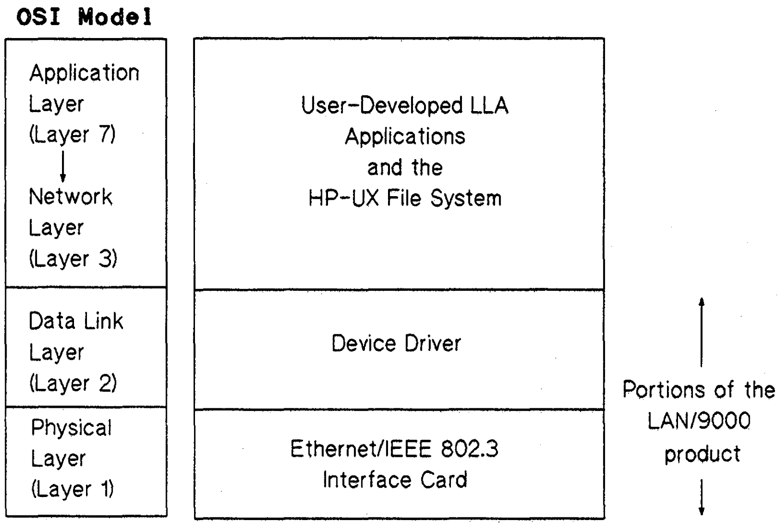


Figure 1-1. LLA and the OSI Model

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 *Networking Overview: LAN, NS, ARPA and NFS*, 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.

For complete information about the Ethernet and IEEE 802.3 protocols, refer to the documentation map at the beginning of this manual.

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.

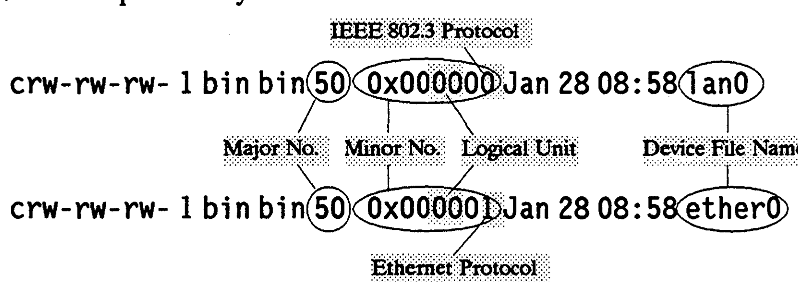
By convention, device files are kept in a directory called `/dev`. When the LAN/9000 product is installed, several special device files are created. Among these files are the network device files associated with the LAN interface. If default names are used during installation, these files are called `/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 600/800 computer only:



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 logical unit (LU) 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 LU (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.

Series 700 only: The major number definition is the same as on a Series 600/800 computer with the exception of the minor number which is bits 8 through 15. In Table 1-1 below, Hw_Loc identifies the hardware location of the IO card on the Series 700. This field is always initialized to 0x202 for core LAN.

Table 1-1. Device File Bit Conventions

Series	Protocol Bit	LU Number	Select Code	Hw_Loc
300/400	-	16-23	8-15	-
600/800	31	16-23	-	-
700	31	-	-	8-19

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.

Table 1-2 lists the major numbers for the HP 9000 LAN interfaces.

Table 1-2. Major Number of LAN/9000 Device Files

Protocol	Computer	Interface	Major Number	Minor Number
Ethernet	Series 300/400	98171A LAN Card	19	encoded select code
IEEE 802.3	Series 300/400	98171A LAN Card	18	encoded select code
Ethernet and IEEE 802.3	815 and 8x2	36967A-20N HP-PB LAN Card	51	encoded logical unit and protocol
Ethernet and IEEE 802.3	8x7S	J2146A HP-PB LAN Card A1703-60003 SCSI Console LAN Card	32	encoded logical unit and protocol
Ethernet and IEEE 802.3	600/800 computers	36967A-20C CIO LAN Card (TurboLAN)	50	encoded logical unit and protocol
Ethernet and IEEE 802.3	Series 700	A1094-66530 CORE IO Card 25567A Add-On EISA Card	52	encoded protocol, device lu, hardware part

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 Series 600/800 device files, refer to the *HP 9000 Series 600/800 System Administrator's Manual*. For Series 300/400 device file information, refer to the *HP 9000 Series 300/400 System Administrator's Manual.*

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 call, refer to the *HP-UX Reference* manual. 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 in the "ioctl(2) Syntax" section 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 2, "Using LLA," and Chapter 3, "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 complete 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)
int fildes, 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:

```
struct fis { int reqtype;
             int vtype;
             union  { float f;
                      int i;
                      unsigned char s[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 ≤ `vtype` ≤ 99)
indicates that the value is a character string in `value.s`.
This integer also specifies the length of the string.

Note At present, 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 detail in Chapter 2, "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 *HP-UX Reference* manual entry for *errno(2)*. Certain *errno* return values are also described in this manual.

2

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.
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 3, "Network I/O Control."

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.

Opening 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 ('|') to combine the 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.

Logging 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 an *ssap* using the HP-UX *ioctl(2)* call with NETCTRL commands. Complete syntax for the *ioctl(2)* call is provided in Chapter 1, "LLA Concepts."

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.

The LOG_SSAP command also sets the destination service access point, or *dsap*, to the same value as the *ssap*. The *dsap* is discussed in the following section.

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.

Logging a Destination Address (Ethernet/IEEE 802.3)

Before writing to a network device, a destination address should be declared. This is done using an HP-UX *ioctl(2)* call. Complete *ioctl(2)* syntax is described in Chapter 1, "LLA Concepts."

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/IEEE 802.3 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, the *landiag (1M)* command, or refer to the Network Map for your network. Both describe the station address in hexadecimal form. Since the LOG_DEST_ADDR requires that you specify the station address in binary form, you must convert the 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.

Reading Data

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

Note Before attempting to read data, you must declare a user-level address as described in "Logging a User-Level Address" earlier in this chapter. An attempt to read data without having logged a user-level address will return the error EDESTADDRREQ.

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 of data bytes is 46 bytes for Ethernet and 0 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 acknowledgement 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

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

Note Before attempting to write data, you must declare a user-level address and a destination address. Declaring a user-level address is described in "Logging a User-Level Address" earlier in this chapter. Declaring a destination address is described in "Logging a Destination Address," also earlier in this chapter. Attempting to write data prior to logging a destination address or user level address returns the error EDESTADDRREQ.

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 bytes** 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 acknowledgement 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 option is 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.

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.

Setting Up Asynchronous Signals

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, resulting 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 ambiguity as to the 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.

Closing a 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/IEEE 802.3 device file is to be closed.

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

3

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 *ioctl(2)* syntax used for LLA is described in Chapter 1, "LLA Concepts."

Collecting and Resetting Interface Statistics

Many commands are provided for collecting and resetting interface statistics. Several of these 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.

The following commands are used as **NETSTAT commands only**; these commands are described first:

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

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 IEEE 802.3 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. (See Chapter 2, "Using LLA," for an explanation of the *net_atoa(3n)* routine.)

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. (See Chapter 1, "Using LLA," for an explanation of the *net_ntoa(3n)* routine.)

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.

This statistic is kept by the **Series 600/700/800 only.**

RESET_STATISTICS Command

The **RESET_STATISTICS** command can be used as a **NETCTRL** ioctl(2) command. It is used to reset 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. 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.**

RESET_STATISTICS	NETCTRL: Resets all statistics counters to zero. No operands are necessary.

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 Series 600/800 only	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 Series 600/800 only	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 Series 600/800 only	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 Link Level Protocol

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 2, "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*. (See LOG_SSAP in the Chapter 2, "Using LLA.") 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. (The *net_aton(3n)* routine is described in Chapter 2, "Using LLA.")

4

LLA Examples

Note **These programs are provided only as examples of LLA usage and are not Hewlett-Packard supported products.**

The source code for the example programs is located on your system in the directory `/usr/netdemo/LLA`.

There are two examples with the following files:

- File transfer
  - *nserver.c*
  - *nget.c*
  - *nput.c*
  - *ncopy.h*
- Network interface statistics report
  - *nstatus.c*

File Transfer Program

The file transfer example consists of the three programs listed in the introduction of this chapter.

Each program must be compiled with the file *ncopy.h*.

nserver.c — A file server program. Find out the station address of the LAN card for the computer on which you plan to run this program. You can do this by running the LAN display portion of the *landiag (1M)* command or the LANDAD portion of the Online Diagnostic Subsystem (Series 800 only), or by referring to the Network Map for your network. Once you know the station address, compile *nserver.c* and run it. You can then run the requester programs.

nget.c — A requester program *nget.c* asks *nserver.c* to get a file, which is displayed on *stdout*. To use *nget.c*, you must know the remote file name and the station address of the computer on which *nserver.c* is running.

nput.c — A requester program *nput.c* asks *nserver.c* to put a file in the server's current working directory, taking input from *nput.c*'s *stdin*. To use *nput.c*, you must know the remote file name and the station address of the computer on which *nserver.c* is running. If you use the wrong station address, *nput.c* will time out.

If necessary, all three programs can be run on the same computer. If you choose to put *nserver.c* on the same computer as *nput.c* and/or *nget.c*, run *nserver* in the background and redirect its output to a file.

Figure 4-1 and Figure 4-2 show the communication between these processes in chronological order.

Source code for the sample programs follows Figures 4-1 and 4-2.

Follow the procedure below to run the example programs.

1. Run *landiag* on the machine on which you are going to run the server program. Select `lan` from the first menu and `display` from the second menu, and note the station address to be used when running the `nget` and `nput` programs.

2. Move to the directory containing the LLA example programs.

   ```
   cd /usr/netdemo/LLA
   ```

3. Start the file server program.

   ```
   ./nserver > /tmp/serverlog &
   ```

4. Run the `nget` program to get file `/etc/passwd` from the server.

   ```
   ./nget /etc/passwd 0x08000900abcd | more
   ```

5. Alternatively, run the `nput` program to put the local file `/etc/passwd` on the server as `/tmp/remfile`.

   ```
   cat /etc/passwd |./nput  /tmp/remfile 0x08000900abcd
   ```

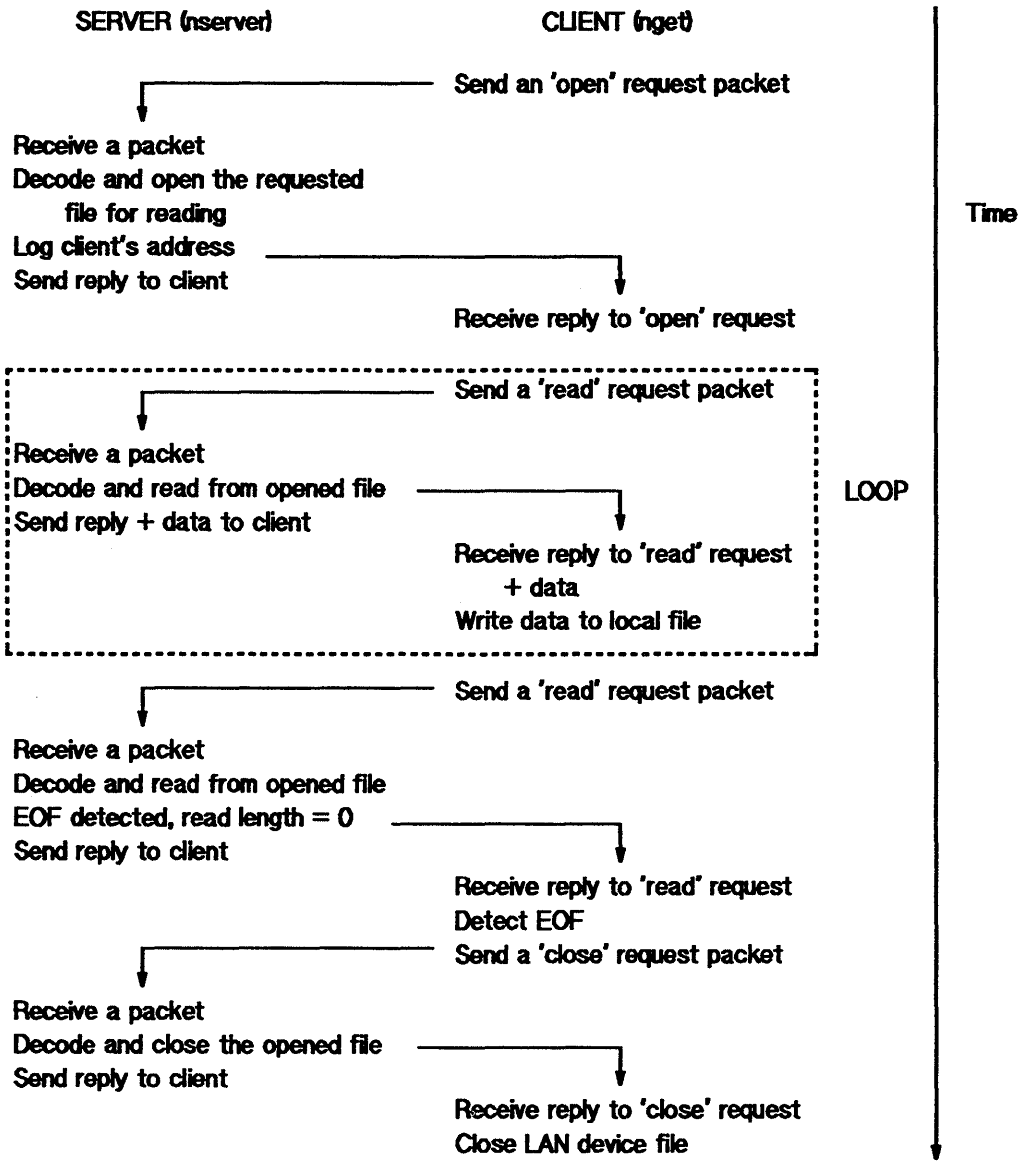


Figure 4-1. nserver nget

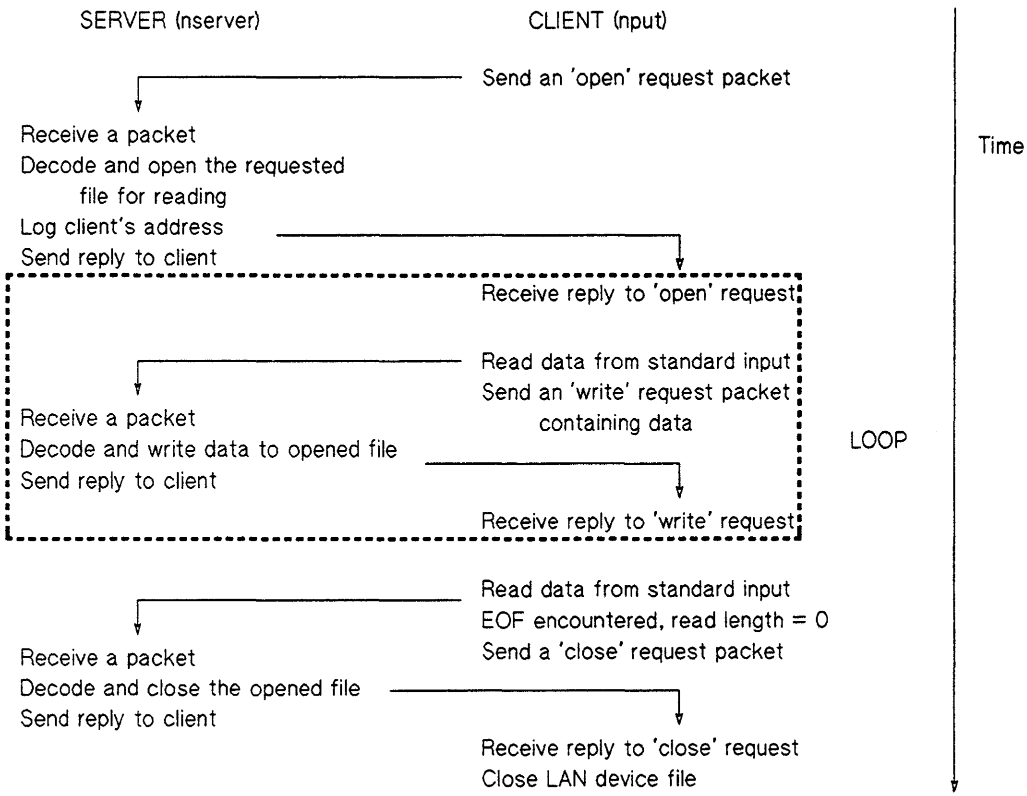


Figure 4-2. nserver nput

```
/**********************************************************************
 *      ncopy.h                                                       *
 *                                                                    *
 *      This is the include file needed for the example file transfer *
 *      programs: nget, nput, and nserver.                             *
 **********************************************************************/

/*
 *      The following is the packet format for the transfers
 *
 *      +--------+--------+--------+--------+
 *      |                 id                |
 *      +--------+--------+--------+--------+
 *      |       mode      |       len       |
 *      +--------+--------+--------+--------+
 *      | data[0]| data[1]| data[2]| data[3]|
 *      +--------+--------+--------+--------+
 *      | data[4]| data[5]| data[6]| data[7]|
 *      +--------+--------+--------+--------+
 *      | ...                               |
 *      :                                   :
 *      |                                   |
 *      +--------+--------+--------+--------+
 *
 *      id      --  integer (four bytes) describing the file
 *      mode    --  describes the action to be performed (if request)
 *                  describe the result of the action (if reply)
 *                  (2 byte integer)
 *      len     --  length of the data field
 *                  (2 byte integer)
 *      data    --  the data to be transferred
 *                  or the filename (if RDOPEN or WROPEN).
 */

#define MAXDATA         1400

/*
 *      The following structure is a structure overlay for the packet
 *      format described above.  This structure is dependent on the
 *      compiler to generate the alignment as shown above.
 */


struct packet_format {
    int         id;
    short       mode;           /* RDOPEN,WROPEN,READ,WRITE,CLOSE,OK,ERR */
    short       len;            /* length of data */
                                /* or length of request READ */
    char        data[MAXDATA];
};

#define OVHEAD_SIZE     (sizeof(struct packet_format) - MAXDATA)
#define PACKET_SIZE     sizeof(struct packet_format)
```

```
/*
 *      The following are enumerated types for above 'mode' field.
 */

#define RDOPEN          0       /* possible REQUEST modes */
#define WROPEN          1
#define READ            2
#define WRITE           3
#define CLOSE           4

#define OK              5       /* possible REPLY modes */
#define ERR             6

/*
 *      The link level access parameters.
 *      The requester SAP and the server SAP are different so that
 *      the server and the requester can both be active at the same
 *      time.
 */

#define REQ_SAP         0x10            /* SAP for requester            */
#define SER_SAP         0x12            /* SAP for server               */
#define TIMEOUT_VALUE   10000           /* allow 10 seconds for reply   */


static char Uni_id[] =  @(#)1.3";
```

```
/*********************************************************************
 *  nserver.c -- this program handles requests from nget and nput   *
 *               programs.                                            *
 *                                                                    *
 *  This program will run until stopped by a signal.                 *
 *  It will log requests to stdout.                                   *
 *********************************************************************/

#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <netio.h>

#include ncopy.h

extern char *net_aton();
extern errno;

/*--------------------------------------------------------------
 *  Algorithm:
 *
 *      check arguments.
 *      set up connection:
 *          Open device file
 *          log SSAP/DSAP
 *      REPEAT
 *          Read(request packet)
 *          CASE packet type OF
 *              OPEN:
 *                      if (cannot open)
 *                          mode = ERR;
 *                      else
 *                          mode = OK;
 *                          id = open file descriptor;
 *              READ:
 *                      if (cannot read)
 *                          mode = ERR;
 *                      else
 *                          mode = OK;
 *                          data = read data;


 *              WRITE:
 *                      if (cannot write)
 *                          mode = ERR;
 *                      else
 *                          mode = OK;
 *              CLOSE:
 *                      if (cannot close)
 *                          mode = ERR;
 *                      else
 *                          mode = OK;
 *          log destination address
```

```
 *          Write(reply packet)
 *      FOREVER
 *
 *-------------------------------------------------------------*/

main ( argc, argv )
int     argc;
char    *argv[];
{
    struct packet_format rxbuf, txbuf;
    int f;
    int res;
    struct fis arg;
    short size;

/*-------------------------------------------------------------
 *      rxbuf     buffer to receive
 *      txbuf     buffer to transmit
 *      f         device file
 *      res       result of system calls
 *      arg       used for ioctl calls
 *      size      size of the reply packet
 *-------------------------------------------------------------*/

    if (argc != 1) {
        fprintf(stderr,  Usage: %s\n , argv[0]);
        exit(__LINE__);
    }

/*-------------------------------------------------------------
 *  Set up the connection
 *-------------------------------------------------------------*/

    f = open(  /dev/lan0", O_RDWR );
        if (f < 0) {
            fprintf(stderr,  Cannot open, f = %d\n", f);
            fprintf(stderr,  Errno = %d\n , errno );
            exit(__LINE__);
        }

/*
 *  Setup the local/remote SAP
 */

    arg.reqtype = LOG_SSAP;
    arg.vtype = INTEGERTYPE;
    arg.value.i = SER_SAP;
    res = ioctl( f, NETCTRL, &arg );
        if (res != 0) {
            fprintf(stderr,  Cannot control(LOG_SSAP), res = %d\n , res);
            fprintf(stderr,  Errno = %d\n , errno );
            exit(__LINE__);
```

```
        }

    arg.reqtype = LOG_DSAP;
    arg.vtype = INTEGERTYPE;
    arg.value.i = REQ_SAP;
    res = ioctl( f, NETCTRL, &arg );
        if (res != 0) {
            fprintf(stderr,  Cannot control(LOG_DSAP), res = %d\n , res);
            fprintf(stderr,  Errno = %d\n , errno );
            exit(__LINE__);
        }
/*-------------------------------------------------------------
 *  LOOP
 *      Read(request)
 *      service request
 *      Write(reply)
 *  FOREVER
 *-------------------------------------------------------------*/


while (1) {
        res = read( f, &rxbuf, PACKET_SIZE );
            if (res <= 0) {
                fprintf(stderr,  Cannot read, res = %d\n , res);
                fprintf(stderr,  Errno = %d\n , errno );
                exit(__LINE__);
            }

        switch ( rxbuf.mode )
        {
        case WROPEN:
            printf("Servicing open for write: %s\n", rxbuf.data );
            res = open( rxbuf.data, O_WRONLY|O_CREAT, 0644 );
            if (res < 0) {
                printf("   cannot open, res = %d\n", res);
                printf("   errno = %d\n", errno );
                txbuf.mode = ERR;
            } else {
                printf("   returned file descriptor %d\n", res );
                txbuf.mode = OK;
            }
            txbuf.id = res;
            txbuf.len = 0;
            size = OVHEAD_SIZE;
            break;

        case RDOPEN:
            printf("Servicing open for read: %s\n", rxbuf.data );
            res = open( rxbuf.data, O_RDONLY );
            if (res < 0) {
                printf("   cannot open, res = %d\n", res);
                printf("   errno = %d\n", errno );
                txbuf.mode = ERR;
```

```
        } else {
            printf("   returned file descriptor %d\n", res );
            txbuf.mode = OK;
        }
        txbuf.id = res;
        txbuf.len = 0;
        size = OVHEAD_SIZE;
        break;


case READ:
        printf("Servicing read: %d\n", rxbuf.id );
        res =  read( rxbuf.id, txbuf.data, rxbuf.len );
        if (res < 0) {
            printf("   cannot read, res = %d\n", res);
            printf("   errno = %d\n", errno );
            txbuf.mode = ERR;
            txbuf.len = 0;
            size = OVHEAD_SIZE;
        } else {
            printf("   read %d bytes\n", res);
            txbuf.mode = OK;
            txbuf.len = res;
            size = OVHEAD_SIZE + res;
        }
        txbuf.id = rxbuf.id;
        break;

    case WRITE:
        printf("Servicing write: %d\n", rxbuf.id );
        res =  write( rxbuf.id, rxbuf.data, rxbuf.len );
        if (res < 0) {
            printf("   cannot write, res = %d\n", res);
            printf("   errno = %d\n", errno );
            txbuf.mode = ERR;
        } else {
            printf("   write %d bytes\n", res);
            txbuf.mode = OK;
        }
        txbuf.len = res;
        txbuf.id = rxbuf.id;
        size = OVHEAD_SIZE;
        break;


case CLOSE:
        printf("Servicing close: %d\n", rxbuf.id );
        res = close( rxbuf.id );
        if (res != 0) {
            printf("   cannot close, res = %d\n", res);
            printf("   errno = %d\n", errno );
            txbuf.mode = ERR;
        } else {
```

```
                printf("   closed file\n");
                txbuf.mode = OK;
            }
            txbuf.len = 0;
            txbuf.id = -1;
            size = OVHEAD_SIZE;
            break;

        default:
            printf("Unrecognized request %d\n", rxbuf.mode);
            txbuf.mode = ERR;
            txbuf.len = 0;
            txbuf.id = -1;
            size = OVHEAD_SIZE;
            break;
        }

/*
 *      Setup the destination address by reading source address of the
 *      request packet.
 */

        arg.reqtype = FRAME_HEADER;
        res = ioctl(f, NETSTAT, &arg);
          if (res != 0) {
          fprintf(stderr,  Cannot status(FRAME_HEADER), res = %d\n , res);
                fprintf(stderr,  Errno = %d\n , errno );
                exit(__LINE__);
            }

        arg.reqtype = LOG_DEST_ADDR;
        arg.vtype = 6;
        copy( arg.value.s, &arg.value.s[6], 6 );
        res = ioctl( f, NETCTRL, &arg );
          if (res != 0) {
          fprintf(stderr,"Cannot control(LOG_DEST_ADDR), res = %d\n",res);
                fprintf(stderr,"Errno = %d\n", errno );
                exit(__LINE__);
            }


/*
 *      write reply packet
 */

        res = write( f, &txbuf, size );
            if (res <= 0) {
                fprintf(stderr,  Cannot write, res = %d\n , res);
                fprintf(stderr,  Errno = %d\n , errno );
                exit(__LINE__);
            }

    }
```

```
}
/*
 *  copy exactly 'num' bytes
 */

copy( to, from, num )
register char *to;
register char *from;
register int num;
{
    while( num  > 0 )
        *to++ = *from++;
}

static char Uni_id[] =  @(#)1.3";
```

```
/*********************************************************************
 *  nget.c -- get a file from the remote machine                    *
 *                                                                   *
 *  This program asks for a file to be transferred over the network. *
 *  The required parameters are the name of the remote file and the  *
 *  link address of the remote machine.                              *
 *  The remote machine must be running the  nserver  program.        *
 *  The file will be printed to stdout, which can be redirected by   *
 *  the shell.                                                       *
 *********************************************************************/

#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <netio.h>

#include ncopy.h

extern char *net_aton();
extern errno;

/*------------------------------------------------------------
 *  Algorithm:
 *
 *      check arguments.
 *      set up connection:
 *          Open device file
 *          log SSAP/DSAP
 *          log destination address
 *          log timeout value
 *      Write(open packet request)
 *      Read(reply to open packet)
 *      REPEAT
 *          Write(read packet request)
 *          Read(reply to read packet)
 *          Write data to output
 *      UNTIL data length received != data length transmitted
 *      Write(close packet request)
 *      Read(reply to close packet)
 *      tear down the connection:
 *          Close device file
 *
 *------------------------------------------------------------*/

main ( argc, argv )
int     argc;
char    *argv[];
{
    struct packet_format rxbuf, txbuf;
    int f;
    int res;
    struct fis arg;
```

```
/*-----------------------------------------------------------
 *      rxbuf      buffer to receive
 *      txbuf      buffer to transmit
 *      f          device file
 *      res        result of system calls
 *      arg        used for ioctl calls
 *------------------------------------------------------------*/

    if (argc != 3) {
        fprintf(stderr,  Usage: %s remote-file remote-addr\n , argv[0]);
        exit(__LINE__);
    }

/*-----------------------------------------------------------
 *  Set up the connection
 *------------------------------------------------------------*/

    f = open(  /dev/lan0", O_RDWR );
        if (f < 0) {
            fprintf(stderr,  Cannot open, f = %d\n", f);
            fprintf(stderr,  Errno = %d\n , errno );
            exit(__LINE__);
        }

/*
 *  Setup the local/remote SAP
 */

    arg.reqtype = LOG_SSAP;
    arg.vtype = INTEGERTYPE;
    arg.value.i = REQ_SAP;

res = ioctl( f, NETCTRL, &arg );
        if (res != 0) {
            fprintf(stderr,  Cannot control(LOG_SSAP), res = %d\n , res);
            fprintf(stderr,  Errno = %d\n , errno );
            exit(__LINE__);
        }

    arg.reqtype = LOG_DSAP;
    arg.vtype = INTEGERTYPE;
    arg.value.i = SER_SAP;
    res = ioctl( f, NETCTRL, &arg );
        if (res != 0) {
            fprintf(stderr,  Cannot control(LOG_DSAP), res = %d\n , res);
            fprintf(stderr,  Errno = %d\n , errno );
            exit(__LINE__);
        }


/*
 *  Setup the destination address
```

```
 */

    arg.reqtype = LOG_DEST_ADDR;
    arg.vtype = 6;
    net_aton( arg.value.s, argv[2], 6 );
    res = ioctl( f, NETCTRL, &arg );
        if (res != 0) {
            fprintf(stderr,  Cannot control(LOG_DEST_ADDR), res = %d\n , res);
            fprintf(stderr,  Errno = %d\n , errno );
            exit(__LINE__);
        }

/*
 *  Setup the timeout value.
 *      If a timeout occurs in a subsequent read, then this program
 *      will be aborted.  In this case the remote nserver program
 *      will need to be stopped and restarted.
 */

    arg.reqtype = LOG_READ_TIMEOUT;
    arg.vtype = INTEGERTYPE;
    arg.value.i = TIMEOUT_VALUE;


res = ioctl( f, NETCTRL, &arg );
      if (res != 0) {
      fprintf(stderr,"Cannot control(LOG_READ_TIMEOUT), res = %d\n", res);
            fprintf(stderr,"Errno = %d\n", errno );
            exit(__LINE__);
        }

/*------------------------------------------------------------
 *  Network Open
 *------------------------------------------------------------*/

    txbuf.mode = RDOPEN;
    txbuf.id   = 0;
    txbuf.len  = strlen(argv[1]) + 1;   /* add 1 for null terminator */
    strcpy( txbuf.data, argv[1] );
    res = write( f, &txbuf, txbuf.len+OVHEAD_SIZE );
        if (res <= 0) {
            fprintf(stderr,  Cannot write, res = %d\n , res);
            fprintf(stderr,  Errno = %d\n , errno );
            exit(__LINE__);
        }

    res = read( f, &rxbuf, PACKET_SIZE );
        if (res <= 0) {
            fprintf(stderr,  Cannot read, res = %d\n , res);
            fprintf(stderr,  Errno = %d\n , errno );
            exit(__LINE__);
        }
    if (rxbuf.mode != OK) {
```

```
        fprintf(stderr,  Remote open problem\n );
        exit(__LINE__);
    }

/*
 *  Set up transmit frames
 */

    txbuf.mode = READ;
    txbuf.id   = rxbuf.id;

/*------------------------------------------------------------
 *  REPEAT
 *      Network read
 *      write(stdout)
 *  UNTIL EOF(stdin)
 *------------------------------------------------------------*/

    do {
        txbuf.len = MAXDATA;
        res = write( f, &txbuf, OVHEAD_SIZE );
            if (res <= 0) {
                fprintf(stderr,  Cannot write, res = %d\n , res);
                fprintf(stderr,  Errno = %d\n , errno );
                exit(__LINE__);
            }

        res = read( f, &rxbuf, PACKET_SIZE );
            if (res <= 0) {
                fprintf(stderr,  Cannot read, res = %d\n , res);
                fprintf(stderr,  Errno = %d\n , errno );
                exit(__LINE__);
            }
        if (rxbuf.mode != OK) {
            fprintf(stderr,  Remote read problem\n );
            exit(__LINE__);
        }


        if (rxbuf.len > 0) {
            res = write( 1, rxbuf.data, rxbuf.len );
                if (res < 0) {
                    fprintf(stderr,  Cannot write stdout, res = %d\n , res);
                    fprintf(stderr,  Errno = %d\n , errno );
                    exit(__LINE__);
                }
        }
    } while ( rxbuf.len > 0 );
```

```
/*-------------------------------------------------------------
 *  Network Close
 *-------------------------------------------------------------*/

    txbuf.mode = CLOSE;
    txbuf.len  = 0;
    res = write( f, &txbuf, txbuf.len+OVHEAD_SIZE );
        if (res <= 0) {
            fprintf(stderr,  Cannot write, res = %d\n , res);
            fprintf(stderr,  Errno = %d\n , errno );
            exit(__LINE__);
        }

    res = read( f, &rxbuf, PACKET_SIZE );
        if (res <= 0) {
            fprintf(stderr,  Cannot read, res = %d\n , res);
            fprintf(stderr,  Errno = %d\n , errno );
            exit(__LINE__);
        }
    if (rxbuf.mode != OK) {
        fprintf(stderr,  Remote close problem\n );
        exit(__LINE__);
    }

/*-------------------------------------------------------------
 *  Tear down the connection
 *-------------------------------------------------------------*/

    res = close( f );
        if (res != 0) {
            fprintf(stderr,  Cannot close, res = %d\n , res);
            fprintf(stderr,  Errno = %d\n , errno );
            exit(__LINE__);
        }

    exit(0);
}

static char Uni_id[] =  @(#)1.3";
```

```
/**********************************************************************
 *  nput.c --  put a file on the remote system.                      *
 *                                                                    *
 *  This program puts a new file on the remote system.               *
 *  The input to the remote file will be read from stdin, which can be  *
 *  redirected from the shell.                                        *
 *  The required parameters are the name of the remote file and the   *
 *  link address of the remote machine.                               *
 *  The remote machine must be running the  nserver  program.         *
 **********************************************************************/

#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <netio.h>

#include ncopy.h

extern char *net_aton();
extern errno;

/*------------------------------------------------------------
 *  Algorithm:
 *
 *      check arguments.
 *      set up connection:
 *          Open device file
 *          log SSAP/DSAP
 *          log destination address
 *          log timeout
 *      Write(open packet request)
 *      Read(reply to open packet)
 *      REPEAT
 *          Read data from input
 *          Write(write packet request)
 *          Read(reply to write packet)
 *      UNTIL eof(input)
 *      Write(close packet request)
 *      Read(reply to close packet)
 *      tear down the connection:
 *          Close device file
 *
 *------------------------------------------------------------*/

main ( argc, argv )
int     argc;
char    *argv[];
{
    struct packet_format rxbuf, txbuf;
    int f;
    int res;
    struct fis arg;
    int more;
```

```
/*----------------------------------------------------------
 *      rxbuf     buffer to receive
 *      txbuf     buffer to transmit
 *      f         device file
 *      res       result of system calls
 *      arg       used for ioctl calls
 *      more      boolean if not at EOF
 *----------------------------------------------------------*/

    if (argc != 3) {
        fprintf(stderr,  Usage: %s remote-file remote-addr\n , argv[0]);
        exit(__LINE__);
    }

/*----------------------------------------------------------
 *  Set up the connection
 *----------------------------------------------------------*/

    f = open(  /dev/lan0", O_RDWR );
        if (f < 0) {
            fprintf(stderr,  Cannot open, f = %d\n", f);
            fprintf(stderr,  Errno = %d\n , errno );
            exit(__LINE__);
        }



/*
 *  Setup the local/remote SAP
 */
    arg.reqtype = LOG_SSAP;
    arg.vtype = INTEGERTYPE;
    arg.value.i = REQ_SAP;
    res = ioctl( f, NETCTRL, &arg );
        if (res != 0) {
            fprintf(stderr,  Cannot control(LOG_SSAP), res = %d\n , res);
            fprintf(stderr,  Errno = %d\n , errno );
            exit(__LINE__);
        }

    arg.reqtype = LOG_DSAP;
    arg.vtype = INTEGERTYPE;
    arg.value.i = SER_SAP;
    res = ioctl( f, NETCTRL, &arg );
        if (res != 0) {
            fprintf(stderr,  Cannot control(LOG_DSAP), res = %d\n , res);
            fprintf(stderr,  Errno = %d\n , errno );
            exit(__LINE__);
        }


/*
```

```
 *  Setup the destination address
 */

    arg.reqtype = LOG_DEST_ADDR;
    arg.vtype = 6;
    net_aton( arg.value.s, argv[2], 6 );
    res = ioctl( f, NETCTRL, &arg );
        if (res != 0) {
            fprintf(stderr,  Cannot control(LOG_DEST_ADDR), res = %d\n , res);
            fprintf(stderr,  Errno = %d\n , errno );
            exit(__LINE__);
        }




/*
 *  Setup the timeout value.
 *      If a timeout occurs in a subsequent read, then this program
 *      will be aborted.  In this case the remote nserver program
 *      will need to be stopped and restarted.
 */

    arg.reqtype = LOG_READ_TIMEOUT;
    arg.vtype = INTEGERTYPE;
    arg.value.i = TIMEOUT_VALUE;
    res = ioctl( f, NETCTRL, &arg );
        if (res != 0) {
            fprintf(stderr,"Cannot control(LOG_READ_TIMEOUT), res = %d\n", res);
            fprintf(stderr,"Errno = %d\n", errno );
            exit(__LINE__);
        }

/*------------------------------------------------------------
 *  Network Open
 *------------------------------------------------------------*/

    txbuf.mode = WROPEN;
    txbuf.id   = 0;
    txbuf.len  = strlen(argv[1]) + 1;   /* add 1 for null terminator */
    strcpy( txbuf.data, argv[1] );
    res = write( f, &txbuf, txbuf.len+OVHEAD_SIZE );
        if (res <= 0) {
            fprintf(stderr,  Cannot write, res = %d\n , res);
            fprintf(stderr,  Errno = %d\n , errno );
            exit(__LINE__);
        }

    res = read( f, &rxbuf, PACKET_SIZE );
        if (res <= 0) {
            fprintf(stderr,  Cannot read, res = %d\n , res);
```

```
            fprintf(stderr,  Errno = %d\n , errno );
            exit(__LINE__);
        }
    if (rxbuf.mode != OK) {
        fprintf(stderr,  Remote open problem\n );
        exit(__LINE__);
    }

/*
 *  Set up transmit frames
 */

    txbuf.mode = WRITE;
    txbuf.id   = rxbuf.id;

/*------------------------------------------------------------
 *  LOOP
 *      read(stdin)
 *      EXIT IF EOF(stdin)
 *      Network write
 *  FOREVER
 *------------------------------------------------------------*/

    while (1) {
        res = read( 0, txbuf.data, MAXDATA );
            if (res < 0) {
                fprintf(stderr,  Cannot read stdin, res = %d\n , res);
                fprintf(stderr,  Errno = %d\n , errno );
                exit(__LINE__);
            }

        if (res == 0) break;    /* loop exit at eof(stdin) */

        txbuf.len = res;
        res = write( f, &txbuf, res+OVHEAD_SIZE );
            if (res <= 0) {
                fprintf(stderr,  Cannot write, res = %d\n , res);
                fprintf(stderr,  Errno = %d\n , errno );
                exit(__LINE__);
            }







        res = read( f, &rxbuf, PACKET_SIZE );
            if (res <= 0) {
                fprintf(stderr,  Cannot read, res = %d\n , res);
                fprintf(stderr,  Errno = %d\n , errno );
                exit(__LINE__);
```

```
        }
    if (rxbuf.mode != OK) {
        fprintf(stderr,  Remote write problem\n );
        exit(__LINE__);
    }
}

/*----------------------------------------------------------
 *  Network Close
 *----------------------------------------------------------*/

    txbuf.mode = CLOSE;
    txbuf.len  = 0;
    res = write( f, &txbuf, txbuf.len+OVHEAD_SIZE );
        if (res <= 0) {
            fprintf(stderr,  Cannot write, res = %d\n , res);
            fprintf(stderr,  Errno = %d\n , errno );
            exit(__LINE__);
        }

    res = read( f, &rxbuf, PACKET_SIZE );
        if (res <= 0) {
            fprintf(stderr,  Cannot read, res = %d\n , res);
            fprintf(stderr,  Errno = %d\n , errno );
            exit(__LINE__);
        }
    if (rxbuf.mode != OK) {
        fprintf(stderr,  Remote close problem\n );
        exit(__LINE__);
    }
/*----------------------------------------------------------
 *  Tear down the connection
 *----------------------------------------------------------*/



    res = close( f );
        if (res != 0) {
            fprintf(stderr,  Cannot close, res = %d\n , res);
            fprintf(stderr,  Errno = %d\n , errno );
            exit(__LINE__);
        }

    exit(0);
}
```

Network Interface Statistics Report Program

This example program, *nstatus.c*, dumps the status of the driver for a Series 600/800 computer. The source code follows.

```
/**************************************************************
 * -------------------------- nstatus.c ------------------        *
 *  This is an example program which dumps out the status of the  *
 *  lan driver.                                                   *
 ***************************************************************/

#include <stdio.h>
#include <fcntl.h>
#include <netio.h>

extern int errno;               /* system errno */

int map[] = {
    RX_FRAME_COUNT,
    TX_FRAME_COUNT,
    UNDEL_RX_FRAMES,
    UNTRANS_FRAMES,
    RX_BAD_CRC_FRAMES,
    ONE_COLLISION,
    MORE_COLLISIONS,
    EXCESS_RETRIES,
    DEFERRED,
    CARRIER_LOST,
    NO_HEARTBEAT,
    ALIGNMENT_ERRORS,
    LATE_COLLISIONS,
    MISSED_FRAMES,
    UNKNOWN_PROTOCOL,
    BAD_CONTROL_FIELD,
    NO_TX_SPACE,
    LITTLE_RX_SPACE,
    TDR,
    RX_XID,
    RX_TEST,
    RX_SPECIAL_DROPPED,
    MULTICAST_ADDR_LIST,
    ILLEGAL_FRAME_SIZE
};

#define NUM_STAT        (sizeof(map) / sizeof(int))

char *descript[] = {
     RX_FRAME_COUNT ,
     TX_FRAME_COUNT ,
     UNDEL_RX_FRAMES ,
     UNTRANS_FRAMES ,
     RX_BAD_CRC_FRAMES ,
```

```
        ONE_COLLISION ,
        MORE_COLLISIONS ,
        EXCESS_RETRIES ,
        DEFERRED ,
        CARRIER_LOST ,
        NO_HEARTBEAT ,
        ALIGNMENT_ERRORS ,
        LATE_COLLISIONS ,
        MISSED_FRAMES ,
        UNKNOWN_PROTOCOL ,
        BAD_CONTROL_FIELD ,
        NO_TX_SPACE ,
        LITTLE_RX_SPACE ,
        TDR ,
        RX_XID ,
        RX_TEST ,
        RX_SPECIAL_DROPPED ,
        MULTICAST_ADDR_LIST ,
        ILLEGAL_FRAME_SIZE
};

char *desc_status[] = {
        INACTIVE ,
        INITIALIZING ,
        ACTIVE ,
        FAILED
};

#define UNKNOWN ((FAILED)+1)

main( argc, argv )
int  argc;
char *argv[];
{
    int f;              /* file descriptor */
    int res;            /* result of ioctl */
    int stat;           /* index for the array map */
    struct fis arg;     /* used to pass parameters to driver */
    char addr[15];      /* used to hold local address */

/*
 *  check the arguments to the program
 */

    if ( argc != 2 ) {
        fprintf(stderr,  Usage: %s device-file\n , argv[0]);
        exit(-1);
    }

    f = open( argv[1], O_RDWR );
    if ( f < 0 ) {
        fprintf(stderr,  Cannot open device file: %s\n , argv[1]);
        fprintf(stderr,      errno = %d\n , errno);
        exit(-1);
```

```
    }

/*
 *  print out the local address
 */

    arg.reqtype = LOCAL_ADDRESS;
    res = ioctl(f, NETSTAT, &arg);
    if (res != 0) {
        fprintf(stderr,  Cannot read local address\n );
        fprintf(stderr,      errno = %d\n , errno);
        exit(-1);
    }

    if (net_ntoa(addr, arg.value.s, 6) == NULL) {
        fprintf(stderr,  Error in converting address\n );
        exit(-1);
    }

    printf("%30s : %s\n",  LOCAL_ADDRESS , addr);

/*
 *  print out the state of the card
 */

    arg.reqtype = DEVICE_STATUS;
    res = ioctl(f, NETSTAT, &arg);
    if (res != 0) {
        fprintf(stderr,  Cannot read device status\n );
        fprintf(stderr,      errno = %d\n , errno);
        exit(-1);
    }

    printf("%30s : %s\n",  DEVICE_STATUS , desc_status[arg.value.i]);
/*
 *  print out all the statistics
 */

    for ( stat = 0; stat < NUM_STAT; stat++ ) {
        arg.reqtype = map[stat];
        res = ioctl(f, NETSTAT, &arg);
        if (res != 0) {
            fprintf(stderr,  Cannot read statistic %s\n , descript[stat]);
            fprintf(stderr,      errno = %d\n , errno);
            exit(-1);
        }
        printf("%30s : %d\n", descript[stat], arg.value.i);
    }
    exit(0);
}
```

A

Implementation Differences

This appendix compares the HP 9000 Series 600/700/800 and HP 9000 Series 300/400 LLA implementations. You should refer to this appendix if you plan to port applications written for the Series 300/400 to the Series 600/700/800 programming environment.

Certain network I/O control commands are unique to either the Series 600/800 or Series 300/400. The following Series 600/800 commands are **not supported** on the Series 300/400:

- NO_TX_SPACE
- LITTLE_RX_SPACE
- TDR
- RX_XID
- RX_TEST
- RX_SPECIAL_DROPPED
- MULTICAST_ADDR_LIST
- ILLEGAL_FRAME_SIZE

The following Series 600/800 commands are **not** supported on the Series 700:

- ILLEGAL_FRAME_SIZE
- NO_TX_SPACE
- LITTLE_RX_SPACE

B

LLA Layer 2 Protocols

This appendix contains diagrams and text that explain the following Ethernet and IEEE 802.3 protocol components:

- Ethernet Frame Structure.
- Ethernet Destination Address Structure.
- IEEE 802.3 Frame Structure.
- IEEE 802.3 Address Field Structures.
- Ethernet and IEEE 802.3 Packet Comparison.

Ethernet Frame Structure

The Ethernet packet contains the following information:

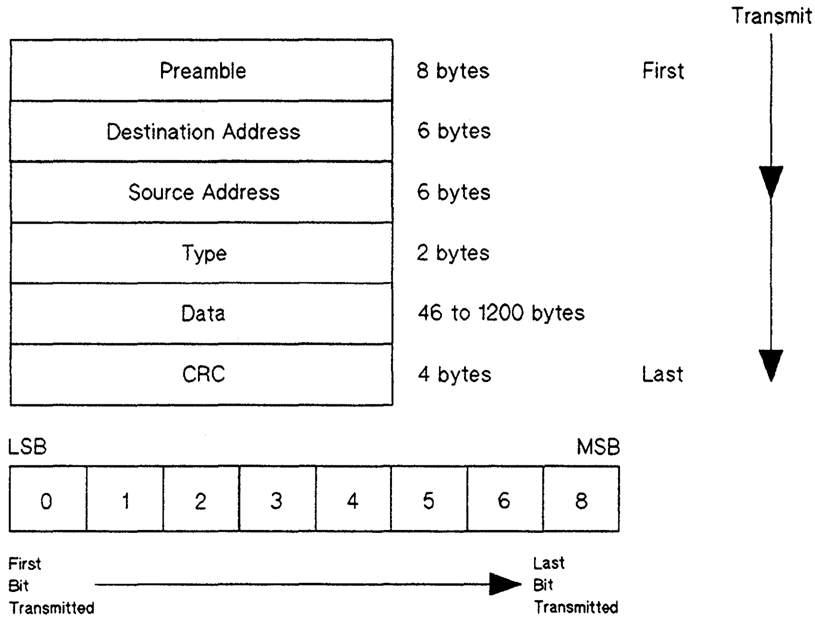


Figure B-1. Ethernet Frame Structure

- **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.

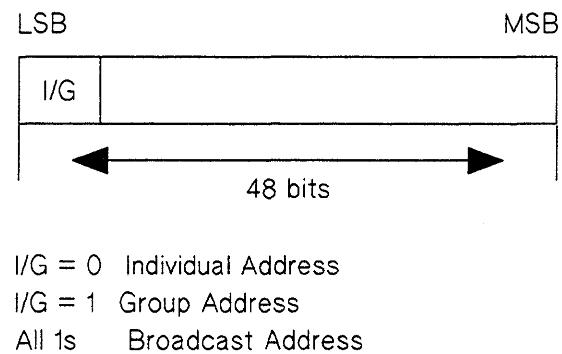


Figure B-2. Ethernet Destination Address

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:

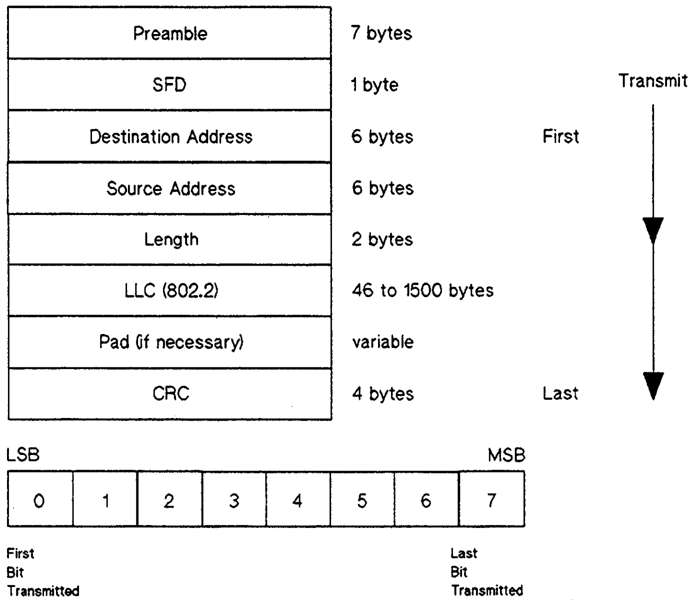


Figure B-3. IEEE 802.3 Frame Structure

- **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 Delimeter (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.

I/G	UPC/L	22 Bit Manuf. Address	24 Bit Address

I/G = 0 Individual Address
I/G = 1 Group Address

UPC/L = 0 Globally Administered Address
UPC/L = 1 Locally Administered Address

Figure B-4. IEEE 802.3 Address Fields

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 as shown in Figure B-5.

DSAP ADDRESS	SSAP ADDRESS	CONTROL	INFORMATION
8 bits	8 bits	Y bits	8*m*bits

DSAP: Destination Service Access Point
SSAP: Source Service Access Point

CONTROL: 16 bits for formats using Sequence numbers
8 bits for formats not using Sequence numbers

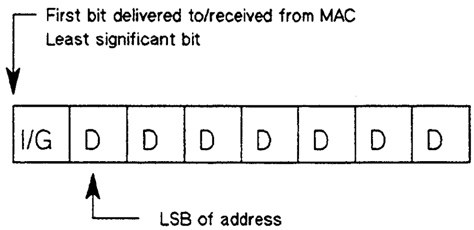
INFORMATION: Integral number of bytes

Figure B-5. IEEE 802.3 LLC Packet Structure

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.



I/G = 0 Individual DSAP
I/G = 1 Group DSAP

X0DDDDDD	Legal DSAP address
X1DDDDDD	Reserved for 802 definition
11111111	Global DSAP
00000000	Null DSAP (addressed to MAC)
01000000	Individual LLC management function
11000000	Group LLC management function

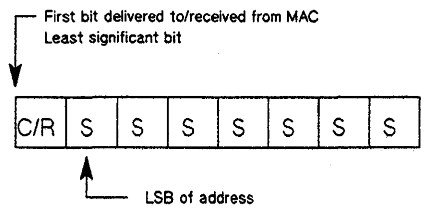
Figure B-6. IEEE 802.3 DSAP Structure

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.



C/R = 0 Command
C/R = 1 Response

X0SSSSSS	Legal SSAP address
X1SSSSSS	Reserved for 802 definition
00000000	Null SSAP (originated from MAC sublayer)

Figure B-7. IEEE 802.3 SSAP Structure

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

Figure B-8 illustrates the differences between the Ethernet and IEEE 802.3 packet structures.

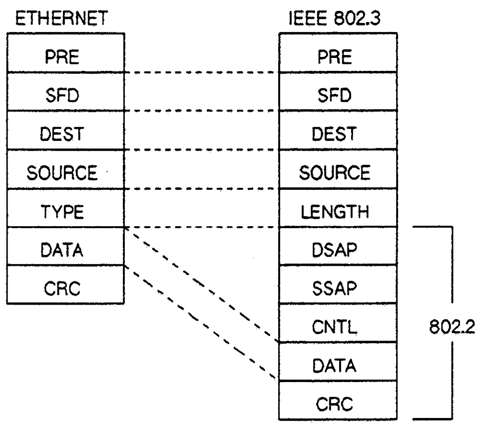


Figure B-8. IEEE 802.3 and Ethernet Packet Comparison

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.

C

Error Messages

This appendix lists and describes the error messages produced by Link Level Access that apply to HP 9000 Series 300/400/700 and Series 600/800 computer systems.

If an error occurs, the error value is given in *errno*. The values for *errno* are defined in the file `/usr/include/sys/errno.h` and in the *HP-UX Reference* manual entry for *errno(2)*.

A list of the error messages, causes, and action to be taken follows. If there is more than one cause for an error message, each cause is given with a corresponding numbered action for it.

MESSAGE	**EBADF**
CAUSE	A NETCTRL *ioctl* or *write* was attempted on an LLA device that was opened with *read* only (O_RDONLY) access permission.
ACTION	Open the device with *read/write* permission (O_RDWR).

MESSAGE	**EBUSY**
CAUSE	An attempt was made to log a user-level address that is already in use.
ACTION	Select a different SSAP or TYPE.

MESSAGE	**EDESTADDRREQ**
CAUSE	1. A *read* or *write* call preceded a LOG_TYPE_FIELD or LOG_SSAP call. 2. A *write* call preceded a LOG_DEST_ADDR call.
ACTION	Establish the LLA connection in proper sequence.

MESSAGE	**EINTR**
CAUSE	During a blocked *read*, the calling process was delivered a software interrupt prior to receiving a packet on its inbound queue.
ACTION	No action is required.

MESSAGE	**EINVAL**
CAUSE	1. An attempt was made to *write* or *read* a negative number of bytes. 2. An attempt was made to *open* with a bad *oflag* value. 3. LOG_DSAP call preceded a LOG_SSAP call. 4. LOG_TYPE_FIELD call was sent to an IEEE 802.3 device. 5. LOG_SSAP, LOG_DSAP, LOG_CONTROL, or RX_XID, RX_TEST, RX_SPECIAL_DROPPED, BAD_CONTROL_FIELD calls were sent to an Ethernet device. 6. An attempt was made to log a user address, and the SSAP or TYPE was out of range.

7. An attempt was made to change a type field or SSAP (user-level address).
8. An improper address format exists in an *ioctl* call involving an address.
9. An ADD_MULTICAST call was attempted, but the supplied address was already in the list.
10. An ADD_MULTICAST call was attempted, but 16 multicast addresses were already logged. (The list is full.)
11. A DELETE_MULTICAST call was attempted, but the supplied address was not in the list.
12. A DELETE_MULTICAST call was attempted, but no multicast addresses have been logged. (The list is empty.)
13. An ADD_MULTICAST or DELETE_MULTICAST call was attempted, but the multicast bit was not set in the address operand.
14. The timeout value passed to LOG_READ_TIMEOUT was negative.
15. An unknown `arg.reqtype`.
16. Incorrect `arg.vtype`.
17. `Fildes` does not specify an active network I/O device.
18. An attempt was made to set LLA_SIGNAL_MASK with undefined events set in the mask operand.

ACTION

1. *Read* and *write* calls require a message length greater or equal to zero.

2. Check options flag in *open* call.

3. A LOG_SSAP command must be successfully completed prior to initiating a change of the DSAP.

4. Invalid command for this protocol.

5. Do not use these calls with an Ethernet device.

6. Refer to proper ranges of SSAP or TYPE given in manual.

7. After an SSAP or TYPE has been logged, reopen the file to change the SSAP or TYPE.

8. Physical address must be 6-byte entities. Check value in vtype field.

9. No action is required. Address is already in list.

10. A multicast address must be deleted before a different one can be inserted.

11. No action is required. Address is not in list.

12. No action is required. Address is not in list.

13. The multicast list must be set in operand.

14. The timeout value must be greater than or equal to zero.

15. Check the file `/usr/include/netio.h` for the proper req types.

16. The user-supplied variable type must match the variable type defined for the *ioctl* request.

17. The process has not successfully opened the LAN device. Retry the file *open* call.

18. The process specified a signal flag value that is not defined. Recheck the flag value.

MESSAGE	**EIO**
CAUSE	A *read* or *write* failure occurred (includes timeout conditions).
ACTION	Retry *read* or *write* call at a later time.

MESSAGE	**EMSGSIZE**
CAUSE	An attempt was made to *write* more than the maximum bytes specified by the selected protocol.
ACTION	Set the message size within the limits of the protocol.

MESSAGE	**ENETUNREACH**
CAUSE	A protocol was not configured for this interface, or the interface was not configured as "up." Refer to the *ifconfig(1m)* entry in the *HP-UX Reference Manual* for more detailed information.
ACTION	Use *ifconfig* to reconfigure the interface.

MESSAGE	**ENOBUFS**
CAUSE	An *open*, *write*, or *ioctl* call could not get enough memory.
ACTION	If an *open* or *ioctl* call: need to configure more networking memory. If a *write* call: the process has exceeded the allocated outbound network memory. The *write* may be retried later when the system has had time to return used memory to the process memory pool.

MESSAGE	**ENOSPC**
CAUSE	An attempt to *write* a packet failed as a result of an outbound queue overflow on the interface card.
ACTION	Retry output. Check the LAN volume; it may be necessary to add additional LAN cards to handle constant high-volume traffic.

MESSAGE	**ENXIO**
CAUSE	1. An attempt was made to *open* a LAN device with an incorrect select code (**Series 300/400 only**) or with an incorrect logical unit or protocol (**Series 600/700/800 only**). 2. The specified driver call could not complete because the interface card was found to be in a **dead** state (that is, the driver was unable to communicate with the interface card). The card must be reset before any further interface activity can resume.
ACTION	1. Check the device file definitions for proper values. 2. Check hardware and I/O configuration.

MESSAGE	**EPERM**
CAUSE	A non-super-user attempted to call a super-user-only command.
ACTION	Set the effective user ID of the process to super-user to successfully complete the call.

MESSAGE	**EWOULDBLOCK**
CAUSE	The LLA connection was opened with the O_NDELAY option, and a subsequent *read* was performed when data have not been queued for this connection.
ACTION	Retry *read* later; it may be advantageous to use SIGNALS to notify the process of the packet arrival.

Index

!

F

I

L

U

W

X

HEWLETT® PACKARD

Printed in USA 07/92 English

Customer Order No.
98194-60534

Manufacturing No.
98194-90034
Mfg. number is for HP internal use only

98194-90034