SX.25 NLI Programmer's Guide

Document Number 007-2268-002

CONTRIBUTORS

Written by Susan Ellis
Edited by Christina Cary
Production by Gloria Ackley
Engineering contributions by Inna Liou, Bob Horen, Irene Kuffel, John Ng, Jay Lan, Jay McCauley

SX.25 NLI Programmer's Guide
Document Number 007-2268-002

Contents

About This Guide ix

1. Introduction to NLI 1
Include Files 2

2. Data Structures 5
Addresses 5
Quality of Service and X.25 Facilities 7
CONS Quality of Service Parameters 8
Non-OSI Facilities 11

3. Listens 21
Listening for Incoming Calls 21
Call User Data Matching 22
Address Matching 23
Priority 24

4. NLI Message Primitives 27
Connect Request/Indication 29
Connect Response/Confirmation 30
Data 31
Data Acknowledgment 31
Expedited Data 32
Expedited Data Acknowledgment 33
Reset Request/Indication 33
Reset Response/Confirmation 34
Disconnect Request/Indication 34
Disconnect Confirmation 36
Abort Indication 36
Listen Request/Response 37

Listen Cancel Request/Response 38
PVC Attach 38
PVC Detach 40

5. Programming Examples 43
Using the NLI Conversion Module 44
Opening a Connection 46
CONS Calls 47
Non-CONS Calls 50
Data Transfer 51
Sending Data 52
Receiving Data 53
Expedited Data 54
Resets 56
Closing a Connection 58
Remote Disconnect 58
Local Disconnect 60
Listening 61
Listening for Incoming Connections 61
Constructing the Listen Message 62
Handling the Connect Indication 65
Acceptance 65
Rejection 65
Negotiation of QOS Parameters 66
Reusing the Listen Stream 67
PVC Operation 68
Attaching a PVC 68
PVC Data Transfer 70
Detaching a PVC 70
Remote Detach 71
Local Detach 71

A. NLI Messages 75

B. Error Codes 79

Glossary 83

Index 87

Tables

Table 2-1 **aflags** Values 6

Table 2-2 **protection_type** Values 10

Table 2-3 **pwoptions** Values 13

Table A-1 Downstream Messages and Associated Outgoing X.25 Packets 75

Table A-2 Upstream Messages and Associated Incoming X.25 Packets 75

Table B-1 N_RI and N_DI Originator Codes 79

Table B-2 N_DI Reason Codes for Network Service Providers 79

Table B-3 N_DI Reason Codes for Network Users 80

Table B-4 N_RI Reason Codes for Network Service Providers 80

Table B-5 N_RI Reason Code For Network Service Users 81

About This Guide

This guide describes IRIS® SX.25 for programmers. It describes the Network Layer Interface (NLI), which the programmer can use to access the X.25 Packet Layer Protocol (PLP) driver. It is assumed that the programmer is familiar with ISO (International Organization for Standardization) 8208, *X.25 Packet Level Protocol for Data Terminal Equipment*.

This guide has the following chapters:

Chapter 1, "Introduction to NLI"
: provides a brief overview of the Network Layer Interface and a glossary of terms. This section also lists the include files that a programmer needs.

Chapter 2, "Data Structures"
: details the function and use of the data structures used across the NLI for addressing, quality of service and facility negotiation.

Chapter 3, "Listens"
: shows how to set up an application to listen for incoming calls.

Chapter 4, "NLI Message Primitives"
: describes the interface message formats and parameters that the X.25 driver supports.

Chapter 5, "Programming Examples"
: provides programming examples using the NLI.

This guide should be used in conjunction with *STREAMS Modules and Drivers, UNIX® SVR4.2*, UNIX Press, 1992.

Other manuals that cover IRIS SX.25 are the following:

- The *SX.25 User Guide* describes the packet assembler/disassembler user interface. It is available for online viewing with *insight*(1).
- The *SX.25 Administrator's Guide* describes the installation and operation of the networking software. It is available for online viewing with *insight*(1).

The IRIS SX.25 reference pages (manual pages) describe all the programs, application utilities, and system files available to programmers, users, and the system administrator.

Chapter 1

Introduction to NLI

This chapter presents a brief introduction to NLI and the system header files required to use it.

Chapter 1

Introduction to NLI

This chapter contains the following section:

- "Include Files" on page 2

IRIS SX.25 supports a Network Layer Interface (NLI) to the X.25 Packet Layer Protocol (PLP) for use by applications. This NLI was developed by Spider Systems, Ltd., and is widely available on many platforms. This interface is not provided as a programming library, but by the standard STREAMS mechanisms for communicating with a stream head. In this way, application programs in user space interact with the in-kernel PLP Driver by exchanging STREAMS messages, using the *getmsg* and *putmsg* system calls. The NLI has been designed so that user level library software can be easily constructed.

Messages passed in this way have both a control part and a data part. Primitives and associated parameters are passed to the X.25 driver by using the control part of messages. If data is to be passed with a primitive, it is contained in the data part of the message.

This means that the application must always provide a control part in messages when using the STREAMS routines *getmsg* and *putmsg*, and whether data is present in the message or not.

Using this message type, the packet structure and parameters necessary for a general X.25 driver can be mapped into the STREAMS environment very easily.

Include Files

Applications using the IRIS SX.25 NLI need to include several system header files:

- *<errno.h>* contains standard error codes.
- *<sys/types.h>* contains type definitions used by STREAMS.
- *<sys/stropts.h>* defines the message structures used in STREAMS system calls.
- *<sys/snet/uint.h>* defines types used by the data structures passed across the NLI.
- *<sys/snet/x25_proto.h>* defines the data structures that must be included.

Since only standard system calls are used, no special library needs to be linked with applications using the NLI. There are, however, some potentially useful service functions in the library *libsx25.a*.

Chapter 2

Data Structures

This chapter describes the data structures used by NLI primitives to specify X.25 addresses and facilities.

Chapter 2

Data Structures

This chapter contains the following sections:

- "Addresses" on page 5
- "Quality of Service and X.25 Facilities" on page 7

This chapter describes the data structures used by NLI primitives to specify X.25 addresses and facilities. These data structures are defined in the file *<sys/snet/x25_proto.h>*. (This file is included in the subsystem *eoe2.sw.dlpi*, which is a prerequisite for IRIS SX.25.)

Addresses

In call requests and responses, it is usually necessary to specify the X.25 addresses associated with the connection—the **called**, **calling** and **responding** addresses. A common structure is used for these addresses. The addressing format used by this structure provides the following information:

- the subnetwork on which outgoing Connect Requests are to be sent and on which Connect Indications arrive
- NSAPs (Network Service Access Points) and SNPAs (Subnetwork Point of Attachments), or DTE (Data Terminal Equipment) addresses and LSAPs (Link Service Access Points)
- options in encoding of addresses (NSAPs)

The addressing format is:

```
#define        NSAPMAXSIZE           20

struct xaddrf {
    unsigned long         sn_id;
    unsigned char         aflags;
    struct lsapformat     DTE_MAC;
    unsigned char         nsap_len;
    unsigned char         NSAP[NSAPMAXSIZE];
}
```

The fields in this structure are:

sn_id The subnetwork identifier, selected by the system administrator. It identifies the subnetwork required for a Connect Request, or on which a Connect Indication arrived. The **sn_id** field holds a representation of the one byte string subnetwork identifier as an unsigned long. The X.25 library routine **snidtox25** can be used to convert the character subnetwork identifier to an unsigned long.

aflags Specifies the options required (or used) by the subnetwork to encode and interpret addresses. When there is a value in the **NSAP** field, **aflags** takes one of the three values listed in Table 2-1. When the **NSAP** field is empty, aflags has the value 0. See the *x25addr*(5), *stox25*(3N), *x25tos*(3N), and *getxhostent*(3N) reference pages for details about the X.25 address format.

Table 2-1 **aflags** Values

Value	Meaning
0	NSAP field contains an OSI (Open Systems Interconnection) encoded NSAP address
1	NSAP field contains a non-OSI encoded extended address
2	DTE_MAC field contains the logical channel identifier (LCI) of a permanent virtual circuit (PVC)

DTE_MAC Holds the DTE address, the MAC+SAP (medium access control+service access point) address or the LCI. This is binary. The **lsapformat** structure is described below.

nsap_len This indicates the length of the NSAP, if any (and where appropriate), in semi-octets.

NSAP This carries the NSAP or address extension (see field **aflags**) when present as indicated by **nsap_len**. This is binary.

The format of the lsapformat structure is as follows:

```
#define    LSAPMAXSIZE    9

struct     lsapformat {
               unsigned char     lsap_len;
               unsigned char     lsap_add[LSAPMAXSIZE];
};
```

The fields in this structure are defined as follows:

lsap_len This gives the length of the DTE address, the MAC+SAP address, or the LCI in semi-octets. For example, for Ethernet, the length is always 14 to indicate the MAC (12) plus SAP (2). The SAP always follows the MAC address. The DTE can be up to 15 decimal digits unless X.25(88) and TOA/NPI (Type Of Address/Numbering Plan Identification) addressing is being used, when it can be up to 17 decimal digits. For an LCI the length is 3.

lsap_add This holds the DTE, MAC+SAP or LCI, when present, as indicated by **lsap_len**. This is binary.

Quality of Service and X.25 Facilities

Negotiable X.25 facilities are supported by the PLP driver. This section describes the request and negotiation of these facilities, and the data structures used by the NLI primitives. Refer to the *SX.25 Administrator's Guide* for details on the options selected for a particular subnetwork. The facility set can be broken down into two main groups—those required for Connection-Oriented Network Service (CONS) support and those for non-OSI procedures (X.29, for example).

Note: CONS can also use the non-OSI procedures.

CONS Quality of Service Parameters

The CONS quality of service (QOS) parameters supported are the following:

- Throughput Class
- Minimum Throughput Class
- Target Transit Delay
- Maximum Acceptable Transit Delay
- Use of Expedited Data
- Protection
- Priority
- Receipt Acknowledgment

CONS-related quality of service parameters are defined in this structure:

```
#define MAX_PROT 32

struct qosformat {
    unsigned char  reqtclass;
    unsigned char  locthroughput, remthroughput;
    unsigned char  reqminthruput;
    unsigned char  locminthru, remminthru;
    unsigned char  reqtransitdelay;
    unsigned short transitdelay;
    unsigned char  reqmaxtransitdelay;
    unsigned short acceptable;
    unsigned char  reqpriority;
    unsigned char  reqprtygain;
    unsigned char  reqprtykeep;
    unsigned char  prtydata;
    unsigned char  prtygain;
    unsigned char  prtykeep;
    unsigned char  reqlowprtydata;
    unsigned char  reqlowprtygain;
    unsigned char  reqlowprtykeep;
    unsigned char  lowprtydata;
    unsigned char  lowprtygain;
    unsigned char  lowprtykeep;
    unsigned char  protection_type;
    unsigned char  prot_len;
    unsigned char  lowprot_len;
```

```
    unsigned char  protection[MAX_PROT];
    unsigned char  lowprotection[MAX_PROT];
    unsigned char  reqexpedited;
    unsigned char  reqackservice;

    struct extraformat xtras;
};
```

The fields in this structure are defined as follows:

Throughput Class
: **reqtclass** is nonzero if the throughput negotiation parameter is selected. The fields **locthroughput** and **remthroughput** contain the four-bit throughput encoding for local-to-remote and remote-to-local, respectively.

Minimum Throughput Class
: **reqminthruput** is nonzero if the minimum throughput negotiation parameter is selected. In this case, the fields **locminthru** and **remminthru** contain the four-bit throughput encoding for the directions local-to-remote and remote-to-local, respectively.

Target Transit Delay
: In Connect Requests and Indications, **reqtransitdelay** is nonzero if the transit delay parameter is selected. In this case **transitdelay** contains the 16-bit value. In a Connect Confirmation, the value of the selected transit delay is placed in the **transitdelay** field and is nonzero.

Maximum Acceptable Transit Delay
: If the calling NLI application specifies a maximum acceptable value for the transit delay parameter (lowest quality acceptable), the field **reqmaxtransitdelay** is nonzero and **acceptable** contains the 16-bit value of the maximum acceptable.

  Note: Transit delay selection applies only to Connect Requests. There is no transit delay QOS parameter in a Connect Response. The correct response when the indicated QOS is unattainable is to make a Disconnect Request. In a Connect Confirmation, the value of the selected transit delay is placed in the **transitdelay** field when such negotiation takes place.

Priority

The **reqpriority** field is used to request/indicate priority on a connection. The mandatory field **prty_data** contains the 8-bit value for the priority of data on the connection. The **reqprtygain** and **reqprtykeep** fields can be optionally set to indicate that the fields **prty_gain** and **prty_keep** contain, respectively, the 8-bit values for the priority to gain and keep a connection.

On N-CONNECT requests, the calling NS_user can also specify a lowest acceptable value for priority. The fields **reqlowprtydata**, **reqlowprtygain**, and **reqlowprtykeep** can be set to indicate that the fields **lowprtydata**, **lowprtygain**, and **lowprtykeep** contain, respectively, the 8-bit values for the lowest acceptable priority of data on connection, to gain a connection, and to keep a connection.

Protection

If the protection negotiation parameter is selected, the **protection_type** is nonzero and indicates the type of protection required. In this case the mandatory fields **prot_len** and **protection** contain, respectively, the length and value for the target protection. On N-CONNECT requests, the calling NS_user can optionally specify a lowest acceptable protection. In this case, the fields **lowprot_len** and **lowprotection** contain, respectively, the length and value for the lowest acceptable protection. Values for **protection_type** are listed in Table 2-2.

Table 2-2 **protection_type** Values

Value	Name	Meaning
1	PRT_SRC	Source address specific
2	PRT_DST	Destination address specific
3	PRT_GLB	Globally unique

Use of Expedited Data

If Expedited Data is required/selected, the field **reqexpedited** is non- zero. For Connect Indications, a value of 1 implies that the Expedited Data negotiation facility was present in the incoming call packet, and that its use was requested.

Note: Negotiation is a CONS procedure. When the facility is present and indicates non-use, use cannot be negotiated by Connect Responses. For a description of the use of the **CONS_call** field in Connect Requests and Connect Responses, see the sections "Connect Request/Indication" and "Connect Response/Confirmation" in Chapter 4.

For incoming or outgoing non-CONS calls (denoted by the **CONS_call** flag set to 0), Expedited Data negotiation is not required—interrupt data is always available in X.25. This means that this field is ignored on Connect Requests and Responses for non-CONS calls.

Receipt Acknowledgment Service

If the receipt acknowledgment service is to be used, the field **reqackservice** is nonzero. Setting **reqackservice** to 1 signifies receipt confirmation by the remote DTE. Setting **reqackservice** to 2 signifies receipt confirmation by the remote application.

In the case of receipt confirmation by the remote DTE, no acknowledgments are expected or given over the X.25 interface. In the case of receipt confirmation by the remote application, there is a one-to-one correspondence between D-bit data and acknowledgments, with one data acknowledgment being received or sent for each D-bit data packet sent or received over the X.25 interface.

Non-OSI Facilities

Note: The non-OSI facilities are also negotiable by CONS.

For those NLI applications that require them, the non-OSI facilities supported are as follows:

- Non-OSI extended addressing
- X.25 fast select request/indication with no restriction on response
- X.25 fast select request/indication with restriction on response
- X.25 reverse charging
- X.25 packet size negotiation

- X.25 window size negotiation
- X.25 network user identification
- X.25 recognized private operating agency selection
- X.25 closed user groups
- X.25 call deflection
- X.25 programmable facilities

Facilities and QOS parameters are defined in the following structure:

```
#define MAX_NUI_LEN    64
#define MAX_RPOA_LEN    8
#define MAX_CUG_LEN     2
#define MAX_FAC_LEN    32
#define MAX_TARIFFS     4
#define MAX_CD_LEN     MAX_TARIFFS * 4
#define MAX_SC_LEN     MAX_TARIFFS * 8
#define MAX_MU_LEN     16

struct extraformat {
    unsigned char     fastselreq;
    unsigned char     restrictresponse, reversecharges;
    unsigned char     pwoptions;
    unsigned char     locpacket, rempacket;
    unsigned char     locwsize, remwsize;
    int               nsdulimit;
    unsigned char     nui_len;
    unsigned char     nui_field[MAX_NUI_LEN];
    unsigned char     rpoa_len;
    unsigned char     rpoa_field[MAX_RPOA_LEN];
    unsigned char     cug_type;
    unsigned char     cug_field[MAX_CUG_LEN];
    unsigned char     reqcharging;
    unsigned char     chg_cd_len;
    unsigned char     chg_cd_field[MAX_CD_LEN];
    unsigned char     chg_sc_len;
    unsigned char     chg_sc_field[MAX_SC_LEN];
    unsigned char     chg_mu_len;
    unsigned char     chg_mu_field[MAX_MU_LEN];
    unsigned char     called_add_mod;
    unsigned char     call_redirect;
    struct lsapformat called;
    unsigned char     call_deflect;
```

```
    unsigned char     x_fac_len;
    unsigned char     cg_fac_len;
    unsigned char     cd_fac_len;
    unsigned char     fac_field[MAX_FAC_LEN];
};
```

The fields in this structure are:

Fast Select
: For non-OSI services like X.29, if the X.25 facility fast select is to be requested or indicated, the field **fastselreq** is nonzero.

  Note: For CONS, the use of fast select need not be requested.

Fast Select with Restricted Response
: If the response to a Connect Request or Indication is to be a Disconnect Indication, the field **restrictresponse** is nonzero.

Reverse Charging
: If reverse charging is requested or indicated for a connection, the field **reversecharges** is non-zero.

  Note: The configuration mode bit SUB_REVCHARGE—see the *SX.25 Administrator's Guide*—has an impact on whether reverse charging is indicated, since it is possible to select a "per subnetwork status" for receipt of reverse charging.

Packet Concatenation, Packet Size, and Window Size Negotiation
: The **pwoptions** field is used to indicate per circuit options. The field is a bit map interpreted as shown in Table 2-3.

Table 2-3 **pwoptions** Values

Bit	Values	Meaning
bit 0	0	Packet size negotiation NOT permitted
	1	Packet size negotiation permitted
bit 1	0	Window size negotiation NOT permitted
	1	Window size negotiation permitted
bit 2	0	No concatenation limit asserted
	1	Assert concatenation limit

The field is defined as follows:

```
#define NEGOT_PKT 0x01  /* packet size is
                           negotiable */
#define NEGOT_WIN 0x02  /* window size is
                           negotiable */
#define ASSERT_HWM 0x04 /* assert concatenation
                           limit */
```

This field is used for two reasons:

1. The X.25 software always indicates the values of the window and packet sizes operating on the virtual circuit. The field **pwoptions** for an incoming call indicates whether these values are negotiable.
2. In Connect Requests and Connect Responses, the NLI user can set **nsdulimit**, the limit value for packet concatenation by the X.25 level, to a value different from the limit in the subnetwork configuration database. It is not a negotiable option, so whatever the user requests is used.

Packet Size

If the fields **locpacket** and **rempacket** are nonzero, **locpacket** contains indicated or negotiated encoded packet sizes for the direction local-to-remote and **rempacket** contains indicated or negotiated encoded packet sizes for the direction remote-to-local.

Note: Actual packet size is 2 to the power of the value.

```
#define DEF_X25_PKT 7 /* the standard default
                         X.25 packetsize */
```

Window Size

If the fields **locwsize** and **remwsize** are nonzero, they contain indicated or negotiated window sizes for the directions local-to-remote and remote-to-local, respectively.

```
#define DEF_X25_WIN 2 /* the standard default
                         X.25 windowsize */
```

Packet Concatenation

If the field **nsdulimit** is nonzero and the appropriate bit is set in the **pwoptions** field described above, the **nsdulimit** specified is used as the concatenation limit.

Network User Identification

The network user identification (NUI) is used in Connect Requests and Responses. It is not available on X.25(80) networks. If the field **nui_len** is nonzero, the network user identification is supplied in **nui_field** and is of length **nui_len** octets.

RPOA Selection

Recognized private operating agency (RPOA), used in Connect Requests only. If the field **rpoa_len** is nonzero, the RPOA DNIC information is supplied in **rpoa_field** and is of length **rpoa_len** semi-octets.

For an X.25(80) network, this is restricted to one RPOA of length 4 semi-octets. The basic format encoding is used for the RPOA selected.

For an X.25(84) or X.25(88) network, one or more RPOAs can be selected. The extended format encoding is used only if the number of RPOAs selected is greater than 1. The maximum number of RPOAs selected is restricted to 4. Valid values for **rpoa_len** are 0, 4, 8, 12, and 16.

Closed User Groups

This field is used in Connect Requests and Indications only. If the field **cug_type** is nonzero, the CUG information is supplied right-justified in **cug_field**. Values for **cug_type** are:

- CUG—closed user group, up to four semi-octets
- BCUG—bilateral CUG (two members only), four semi-octets

Note: Incoming CUG facilities are assumed to have been validated by the network. No further checking is performed.

Charging Information

If the field **reqcharging** is nonzero in a Connect Request or Connect Indication, Call Charging is requested. In a Disconnect Indication or Disconnect Confirmation, the following three fields give the lengths of the charging information:

- **chg_cd_len** is the length of **chg_cd_field**—call duration
- **chg_sc_len** is the length of **chg_sc_field**—segment count
- **chg_mu_len** is the length of **chg_mu_field**—monetary unit

A zero length field means no charging information is supplied for the relevant charging category.

Called Address Modification

A nonzero **called_add_mod** field holds the reason for any address modification.

Call Redirection

A nonzero **call_redirect** field holds the reason for the call redirection. The field **called** supplies the originally-called DTE address.

Call Deflection

A nonzero **call_deflection** field holds the reason for the call deflection. The **deflected** field in the Disconnect Request contains the DTE address, and if required, the NSAP address that the call is to be deflected to.

Programmable X.25 Facilities

This field is used in Connect Requests and Connect Indications only. Provision is made for the passing of explicit facility encoded strings for X.25 facilities, and non-X.25 facilities for calling and called networks.

The fields **x_fac_len, cg_fac_len,** and **cd_fac_len** denote the lengths of the facilities in the field **fac_field** relating to, respectively, X.25 facilities, non-X.25 facilities for the calling network, and non-X.25 facilities for the called network.

If a length field is zero, this denotes that no facilities are supplied for the corresponding facility category.

Note: The contents of this field, if supplied, are not validated or acted upon by the code. The X.25 facilities are inserted at the end of any other X.25 facilities that are passed in the Connect Request/Indication (for example, packet/window sizes). If any non-X.25 facilities are supplied, the appropriate marker is inserted before the supplied facilities.

Chapter 3

Listens

This chapter describes the features of listening.

Chapter 3

Listens

This chapter contains the following sections:

- "Listening for Incoming Calls" on page 21
- "Call User Data Matching" on page 22
- "Address Matching" on page 23
- "Priority" on page 24

The major features of listening are the following:

- Any number of processes can listen simultaneously, subject to resource constraints imposed by the system administrator. Moreover, any number of these processes can listen at the same (set of) **called addresses**. Note that there are no means of listening for a particular **calling address**.
- An application can elect to listen and handle one or more Connect Indications at a time. The most likely use of this feature is when the application wants to make use of the next facility.
- An incoming connection may be accepted on a stream other than the one that received the Connect Indication (the listening stream).

Listening for Incoming Calls

When an application wishes to listen for incoming calls, it must specify the (called) address(es) and Call User Data (CUD) field values for which it is prepared to accept calls. The data that does this is passed as part of a Listen Request.

The control part of the message is accompanied by a data part containing the addresses to be registered for incoming calls. The data portion is treated as a byte stream of CUD and addresses conforming to the following definition:

```
unsigned char  l_cumode;
unsigned char  l_culength;
unsigned char  l_cubytes [l_culength];
unsigned char  l_mode;
unsigned char  l_type;
unsigned char  l_length;
unsigned char  l_add[(l_length+1)>>1];
```

It is important to note that, depending on both the value of the "mode" bytes and the lengths, not all fields need to be present. Refer to the individual field descriptions below for more details.

Call User Data Matching

The fields **l_cumode, l_culength**, and **l_cubytes** are used to match the CUD field of the incoming call, if any, against that specified in the Listen Request.

l_cumode This field defines the type of matching. Three cases are possible:

- X25_DONTCARE

  The listener ignores the CUD—**l_culength** and **l_cubytes** are omitted.

- X25_IDENTITY

  The listener match is made only if all bytes of the CUD field are the same as the supplied **l_cubytes**.

- X25_STARTSWITH

  The listener match is made only if the leading bytes of the CUD field are the same as the supplied **l_cubytes**.

  The last two are intended to distinguish X.29, for example, from other higher level protocols.

l_culength This is the length of the CUD in octets for an X25_IDENTITY or X25_STARTSWITH CUD field match. If **l_culength** is zero, the **l_cubytes** are omitted. Currently, the range for **l_culength** is zero to 16 inclusive. The application still has to check the full CUD field.

l_cubytes This is the string of bytes sought in the CUD field when **l_cumode** is X25_IDENTITY or X25_STARTSWITH.

Address Matching

The fields **l_mode, l_type, l_length** and **l_add** are used to match the address field(s) of the incoming call against that specified in the Listen Request.

l_mode This field defines the type of matching to be done. Three cases are possible:

- X25_DONTCARE

  The listener ignores the address—**l_type, l_length**, and **l_add** are omitted.

- X25_IDENTITY

  The listener match is made only if all semi-octets of the address are the same as the supplied **l_add.**

- X25_STARTSWITH

  The listener match is made only if the leading semi-octets of the address are the same as the supplied **l_add.**

l_type This is the type of the address entry, and it can have two values: X25_DTE or X25_NSAP. It denotes the important addressing quantity. For X.25(84) and X.25(88), for example, NSAPs (or extended addresses) are the important addresses, while for X.25(80), where there is no NSAP, the DTE is the important quantity. Various applications can be distinguished by X.25 DTE subaddress where necessary.

	On many X.25(84) and X.25(88) networks, it is possible to listen on either X25_DTE or X25_NSAP addresses. This is not possible when running X.25(84) or X.25(88) over LLC-2 (Logical Link Control-Class II) on the LAN. In this case, the DTE address field is NULL and the X25_NSAP field is used.
l_length	This is the length of the address **l_add** in semi-octets—the common format for X.25 DTE addresses and NSAPs. If **l_length** is zero, then **l_add** is omitted. The maximum values for **l_length** are 15 for X25_DTE and 40 for X25_NSAP.
l_add	This contains the address. **l_add** is omitted when **l_length** is zero.

Priority

The Listen Request queue is ordered in terms of the amount of listen data supplied. The more a Listen Request asks for, the higher its place in the queue. Connect Indications are sent to the listener whose listening criteria are best matched.

Privileged users can ask for a Listen Request to be placed at the front of the queue, regardless of the amount of listen data supplied. To do this, the Listen Request should be sent as a M_PCPROTO message. This is achieved by setting the RS_HIPRI flag in *putmsg*. Such requests are searched in the order in which they arrive.

The system administrator controls whether or not listening for incoming calls is a privileged operation. If listening is privileged, incoming calls will be sent only on listen streams opened by a user with superuser privilege. This prevents other users accepting calls that may contain private information, passwords, and so on.

In systems where privileged and non-privileged listens are allowed:

- Privileged listens have priority.
- A matching but busy privileged listen prevents a search of any non-privileged listens.

Chapter 4

NLI Message Primitives

This chapter describes NLI message primitives and their data structures.

Chapter 4

NLI Message Primitives

This chapter contains the following sections:

- "Connect Request/Indication" on page 29
- "Connect Response/Confirmation" on page 30
- "Data" on page 31
- "Data Acknowledgment" on page 31
- "Expedited Data" on page 32
- "Expedited Data Acknowledgment" on page 33
- "Reset Request/Indication" on page 33
- "Reset Response/Confirmation" on page 34
- "Disconnect Request/Indication" on page 34
- "Disconnect Confirmation" on page 36
- "Abort Indication" on page 36
- "Listen Request/Response" on page 37
- "Listen Cancel Request/Response" on page 38
- "PVC Attach" on page 38
- "PVC Detach" on page 40

The control part of the messages passed across the NLI has a format defined by structures in the following **C** union:

```
union X25_primitives {
    struct xcallf   xcall;   /* Connect Request/Indication */
    struct xccnff   xccnf;   /* Connect Confirm/Response */
    struct xdataf   xdata;   /* Normal, Q-bit or D-bit data*/
    struct xdatacf  xdatac;  /* Data ack */
    struct xedataf  xedata;  /* Expedited data */
    struct xedatacf xedatac; /* Expedited data ack */
    struct xrstf    xrst;    /* Reset Request/Indication */
    struct xrscf    xrscf;   /* Reset Confirm/Response */
    struct xdiscf   xdisc;   /* Disconnect
                                Request/Indication */
    struct xdcnff   xdcnf;   /* Disconnect Confirm */
    struct xabortf  abort;   /* Abort Indication */
    struct xlistenf xlisten; /* Listen Command/Response */
    struct xcanlisf xcanlis; /* Cancel Command/Response */
    struct pvcattf  pvcatt;  /* PVC Attach */
    struct pvcdetf  pvcdet;  /* PVC Detach */
};
```

The above messages have common fields, which can be accessed by the following type:

```
typedef struct xhdrf {
    unsigned char xl_type;     /* XL_CTL/XL_DAT */
    unsigned char xl_command;  /* Command */
} S_X25_HDR;
```

The messages to and from the application are classified as *control* or *data*, depending on the value of **xl_type**, which is either XL_CTL (*control*) or XL_DAT (*data*). Within each classification, the exact message identity is determined by the **xl_command** qualifier, and it is important to ensure that the combination of **xl_type** and **xl_command** is consistent. Each of these cases is described below.

Connect Request/Indication

The control part of a Connect Request or Indication message has a format defined in the following structure:

```
struct xcallf {
    unsigned char xl_type;     /* Always XL_CTL */
    unsigned char xl_command;  /* Always N_CI */
    int conn_id;               /* The connection id returned
                                  in Connect Response or
                                  Disconnect */
    unsigned char CONS_call;   /* When set, indicates a CONS
                                  call */
    unsigned char negotiate_qos;
                               /* When set, negotiate
                                  facilities etc. or else
                                  use defaults */
    struct xaddrf calledaddr;  /* The called and */
    struct xaddrf callingaddr; /* calling addresses */
    struct qosformat qos;      /* Facilities and CONS qos: if
                                  negotiate_qos is set */
};
```

This structure is used when calls are requested or indicated across the X.25 interface. The data part of the message contains the Call User Data (CUD), if any. Other components are:

conn_id For incoming calls, an attempt is made to match the called address and CUD with that of one of the listening applications. If a match is found, then the indication is passed to that application with a **conn_id** identifier, which must be returned in the Connect Response or Disconnect Request to accept or reject the connection.

CONS_call When this field is set in Connect Requests, it indicates that CONS procedures should be used for the call.

negotiate_qos A nonzero value shows that facilities and quality of service (QOS) are being negotiated. A zero value for the flag means the initiator is requesting all default values.

qos This structure holds the facilities requested/indicated. See the section “Opening a Connection” in Chapter 5 for more information on QOS negotiation.

calledaddr Holds the called address.

callingaddr Holds the calling address.

Connect Response/Confirmation

The control part of a Connect Response or Confirmation message is defined in the following structure:

```
struct xccnff {
    unsigned char xl_type;      /* Always XL_CTL */
    unsigned char xl_command;   /* Always N_CC */
    int      conn_id;           /* The connection id quoted
                                   on the associated
                                   indication */
    unsigned char CONS_call;    /* When set, indicates CONS
                                   call */
    unsigned char negotiate_qos;/* When set, negotiate
                                   facilities etc. else use
                                   indicated values */
    struct xaddrf responder;    /* Responding address */
    struct qosformat rqos;      /* Facilities and CONS qos */
                                /* if negotiate_qos is set */
};
```

This structure is used when calls are being accepted. The data part of the message contains the CUD, if any. The components are:

conn_id The connection identifier **conn_id** must be returned in the Connect Response so that the procedures described in the section “Listening” in Chapter 5 can be guaranteed to operate properly.

CONS_call When this field is set in Connect Responses, it indicates that CONS procedures should be used for the call.

negotiate_qos A nonzero value shows that facilities and quality of service(QOS) are being negotiated. A zero value for the flag means the initiator is requesting all default values.

responder Holds the responding address.

qos Holds selected facilities and CONS QOS parameters to be passed to the initiator.

Data

The control part of a Data message is defined in the following structure:

```
struct xdataf {
    unsigned char xl_type;         /* Always XL_DAT */
    unsigned char xl_command;      /* Always N_Data */
    unsigned char More,            /* Set when more data is
                                      required to complete
                                      the nsdu */
              setDbit,             /* Set when data carries
                                      X.25 D-bit */
              setQbit;             /* Set when data carries
                                      X.25 Q-bit */
};
```

This structure is used when data crosses the X.25 interface.

More Shows whether there is more of this network service data unit to be received or sent.

setQbit Used to request or indicate that the Q-bit is set when user data is transmitted or received.

setDbit Used to request or indicate that the D-bit is set when user data is transmitted or received.

The following M_DATA portion contains the user data.

Note: No acknowledgment for this data is given to, or expected from, the application unless the D-bit is set and application to application receipt confirmation is being used.

Data Acknowledgment

This following structure is associated with Data Acknowledgment messages:

```
struct xdatacf {
    unsigned char xl_type;      /* Always XL_DAT */
    unsigned char xl_command;   /* Always N_DAck */
};
```

This structure is used when an N-DATA-ACK request or an N-DATA-ACK indication crosses the X.25 interface.

When receipt confirmation from the remote application is active on a virtual circuit, this structure is used to acknowledge a previous N-DATA request/indication which had the D-bit set. There is a one to one correspondence between D-bit data and acknowledgments, with one Data Acknowledgment being received/sent for each D-bit data packet sent/received. It is always the oldest outstanding D-bit packet that is being acknowledged.

For CONS calls, if receipt acknowledgment has been negotiated on the connection, then the above procedures should apply for any D-bit data sent or received. However to be compatible with previous releases of the NLI, the value of the **reqackservice** field in the qos structure can be set to request that the D-bit signifies receipt confirmation by the remote DTE only, thus ensuring no acknowledgment is expected or given.

For non CONS calls, only if the **reqackservice** field in the qos structure has been set to the appropriate value will the above procedures apply for any D-bit data sent or received. Otherwise no acknowledgment is expected or given.

Expedited Data

The control part of an Expedited Data message has a format defined in the following structure:

```
struct xedataf {
    unsigned char xl_type;      /* Always XL_DAT */
    unsigned char xl_command;   /* Always N_EData */
};
```

This structure is used when Expedited Data, carried by an X.25 interrupt packet, crosses the X.25 interface. No parameters are required.

The following M_DATA portion of the message contains the user data. The Expedited Data is a confirmed primitive and must be acknowledged (see the next section) before another Expedited Data unit can be requested or indicated.

Expedited Data Acknowledgment

The control part of the Expedited Data Acknowledgment message has a format defined in the following structure:

```
struct xedatacf {
    unsigned char xl_type;      /* Always XL_DAT */
    unsigned char xl_command;   /* Always N_EAck */
};
```

This structure is used when Expedited Data needs to be, or is being, acknowledged.

No parameters or user data are required.

Reset Request/Indication

The control part of a Reset Request or an Indication message has a format defined in the following structure:

```
struct xrstf {
    unsigned char xl_type;     /* Always XL_CTL */
    unsigned char xl_command;  /* Always N_RI */
    unsigned char originator,  /* Originator and Reason
                                  mapped */
                  reason,      /* from X.25 cause/diag in
                                  indications */
                  cause,       /* X.25 cause byte */
                  diag;        /* X.25 diagnostic byte */
};
```

This structure is used when a Reset Request/Indication crosses the X.25 interface. Data is never associated with the primitive.

The X.25 cause and diagnostic bytes, **cause** and **diag**, are presented, as well as the CONS **originator** and **reason** codes, which are mapped from these. For a Reset Request on a non-CONS call, the user can specify a nonzero **cause** code. This has no effect for a CONS call; the value is set to zero by the system.

Note: A Reset Request primitive is an acknowledged service (see the associated structure **xrscf**). A collision between a Reset Indication and a Reset Request is taken to acknowledge the Reset Request—no Reset Confirmation is then required.

Reset Response/Confirmation

The control part of a Reset Response or Confirmation message has a format defined in the following structure:

```
struct xrscf {
    unsigned char xl_type;      /* Always XL_CTL */
    unsigned char xl_command;   /* Always N_RC */
};
```

This structure is used when a Reset Response/Confirmation to a previous Reset crosses the X.25 interface. There are no parameters or data associated with the primitive. The comments above on Reset collision also apply here.

Disconnect Request/Indication

The control part of a Disconnect Request or Indication message has a format defined in the following structure:

```
struct xdiscf {
    unsigned char xl_type;      /* Always XL_CTL */
    unsigned char xl_command;   /* Always N_DI */
    unsigned char originator,   /* Originator and Reason
                                   mapped */
                  reason,       /* from X.25 cause/diag in
                                /* indications */
                  cause,        /* X.25 cause byte */
                  diag;         /* X.25 diagnostic byte */
    int           conn_id;      /* The connection id (for
                                   reject only) */
    unsigned char indicated_qos;/* When set, facilities
                                   indicated */
    struct xaddrf responder;    /* CONS responder address */
    struct xaddrf deflected;    /* Deflected address */
```

```
    struct qosformat qos;       /* If indicated_qos is set,
                                   holds facilities and CONS
                                   qos */
};
```

This structure is used when a Disconnect Request/Indication crosses the X.25 interface. The data part of the message contains the clear user data, if any.

The X.25 cause and diagnostic bytes, **cause** and **diag**, are presented, as well as the CONS **originator** and **reason** codes mapped from these. For a Disconnect Request on a non-CONS call, the user can specify a nonzero **cause** code. This has no effect for a CONS call; the value is set to zero by the system.

Other parameters are:

indicated_qos A nonzero value shows that facilities and QOS are being indicated.

responder This field contains the responding address.

deflected This field is used in conjunction with the **call_deflect** facility in the qos structure, to convey the address of the remote DTE that the call is to be deflected to.

qos Contains the facilities indicated. Currently, this is used with the charging information facility and the call deflection facility.

The Disconnect Request from an application is confirmed unless it is a rejection of a previous Connect Indication. When it is not a rejection, the X.25 driver sends a Disconnect Confirmation to the application when the Disconnect Request is received. This guarantees that, once the Disconnect Confirmation is observed by the application, no more messages are sent on this stream. For this reason, after requesting disconnection, the application should read and discard all messages from the stream until the Disconnect Confirmation is received.

For call rejection, no “acknowledgment” is sent. However, the application must supply the connection identifier presented in the Connect Indication so that the appropriate circuit is cleared.

In the case of a Disconnect Indication, all messages sent downstream except connect messages are discarded silently.

Note: A disconnect collision can occur. If it does, the "acknowledgment" can be taken to be complete.

Disconnect Confirmation

The control part of a Disconnect Confirmation message has a format defined in the following structure:

```
struct xdcnff {
    unsigned char xl_type;       /* Always XL_CTL */
    unsigned char xl_command;    /* Always N_DC */
    unsigned char indicated_qos; /* When set, facilities
                                    indicated */
    struct qosformat rqos;       /* If indicated_qos is set,
                                    holds facilities and
                                    CONS qos */
};
```

This structure is used when a Disconnect Confirmation crosses the X.25 interface. There is no data associated with this primitive. The components of the structure are:

indicated_qos
: A nonzero value shows that facilities and QOS are being indicated.

rqos
: Contains the facilities indicated. Currently, this is used only with the charging information facility.

Abort Indication

The control part of an Abort Indication message has a format defined in the following structure:

```
struct xabortf {
    unsigned char xl_type;       /* Always XL_CTL */
    unsigned char xl_command;    /* Always N_Abort */
};
```

This structure is used when the X.25 driver needs to send a Disconnect Indication to the application, but there is no resource available in the system to construct a full Disconnect Indication message. For this reason, this message should rarely be received.

Note: This message is used only in the upstream direction—never downstream.

Listen Request/Response

The control part of a Listen Request or Response message has a format defined in the following structure:

```
struct xlistenf {
    unsigned char xl_type;     /* Always XL_CTL */
    unsigned char xl_command;  /* Always N_Xlisten */
    int lmax;                  /* Maximum number of CI's at a
                                  time */
    int l_result;              /* Result flag */
};
```

This structure is used when an NLI application wants to register interest in incoming calls. The components are:

lmax This is the maximum number of Connect Indications that the listener is willing to handle at one time. The data part of the message carries the address(es) in which the listener is interested (see also Chapter 3, "Listens").

Note: Listen Requests are cumulative, but the **lmax** value (number of simultaneously handled Connect Indications) is not. This means that several Listen Requests can be made on a single Stream, in which case the **lmax** value contained in the last Listen Request message specifies the number of simultaneously handled Connect Indications.

l_result The result of the Listen Request is acknowledged upstream with the same message. An error in the parameters or a lack of resources to set up the listen results in this flag being set to a nonzero value.

For more information, see Chapter 3, "Listens."

Listen Cancel Request/Response

The control part of a Listen Cancel Request or Response message has a format defined in the following structure:

```
struct xcanlisf {
    unsigned char xl_type;      /* Always XL_CTL */
    unsigned char xl_command;   /* Always N_Xcanlis */
    int c_result;               /* Result flag */
};
```

This structure is used to cancel an interest in incoming calls. Like the Listen Request message described above, this request is confirmed. In this case, a nonzero value of the **c_result** flag indicates failure of the operation to cancel a Listen Request. For example, the Listen Request was not present or some connect event is outstanding. Naturally, the closure of a stream on which there is a Listen Request also cancels the Listen Request, but in the case of the Listen Cancel Request message, the stream remains open.

Note: The Listen Cancel Request removes all listen addresses from the stream. There is no way of cancelling a Listen Request on a particular address; this message is probably used when the *use* of the stream is about to be changed by the application.

PVC Attach

The control part of a PVC Attach message has a format defined in the following structure:

```
struct pvcattf {
    unsigned char xl_type;        /* Always XL_CTL */
    unsigned char xl_command;     /* Always N_PVC_ATTACH */
    unsigned short lci;           /* Logical channel */
    unsigned long sn_id;          /* Subnetwork identifier */
    unsigned char reqackservice;  /* Receipt Acknowledgement
                                     0 for next parameter
                                     implies use of
                                     default */
```

```
    unsigned char reqnsdulimit;
    int nsdulimit;
    int result_code;              /* Nonzero - error */
};
```

This structure is used when a PVC Attach crosses the X.25 interface.

This message is used when a user wants to "attach" to a PVC. The components are:

lci Contains the logical channel identifier (LCI) of the required PVC.

sn_id Denotes the particular subnetwork for the PVC.

reqackservice If nonzero, denotes that the receipt acknowledgment service is requested by use of the D-bit. Setting **reqackservice** to 1 signifies receipt confirmation by the remote DTE. Setting **reqackservice** to 2 signifies receipt confirmation by the remote application.

In the case of receipt confirmation by the remote DTE, no acknowledgments are expected or given over the X.25 interface. In the case of receipt confirmation by the remote application, there is a one-to-one correspondence between D-bit data and acknowledgments, with one data acknowledgment being received or sent for each D-bit data packet sent or received over the X.25 interface.

reqnsdulimit If this is non-zero, look at the field **nsdulimit**.

nsdulimit Specifies the packet concatenation limit for network service data units (NSDU).

result_code In the attach message sent to the user as acknowledgment, this field denotes whether or not the attach was successful.

PVC Detach

The control part of a PVC Detach message has a format defined in the following structure:

```
struct pvcdetf {
    unsigned char xl_type;     /* Always XL_CTL */
    unsigned char xl_command;  /* Always N_PVC_DETACH */
    int reason_code;           /* Reports why */
};
```

This structure is used when a PVC Detach crosses the X.25 interface. This message is used when a user wants to “detach” from the PVC. This allows the use of the stream to be changed.

The PVC Detach message is acknowledged to the user by returning a PVC Detach message, in which the field **reason_code** denotes whether or not the PVC Detach was successful.

This message is also used by the X.25 driver to inform the user of some failure of the PVC. These include link down, remote end not responding, and so on. When the message is sent by the X.25 driver, the field **reason_code** gives the reason for the PVC Detach.

Chapter 5

Programming Examples

This chapter provides examples of various operations: opening a connection, data transfer, closing a connection, listening, and PVC operations.

Chapter 5

Programming Examples

This chapter contains the following sections:

- "Using the NLI Conversion Module" on page 44
- "Opening a Connection" on page 46
- "Data Transfer" on page 51
- "Closing a Connection" on page 58
- "Listening" on page 61
- "PVC Operation" on page 68

To perform any of the operations described in this section, the application must **open** a stream to the X.25 PLP driver. Once the stream has been opened, it can be used for initiating, listening for, or accepting a connection. There is a one-to-one mapping between X.25 virtual circuits and PLP driver streams. Once a connection has been established on a stream, the stream cannot be used other than for passing data and protocol messages for that connection.

Such a stream is opened on **/dev/x25**, the major device, as follows:

```
if ((x25_fd = open("/dev/x25", O_RDWR))  <  0)
{
        perror("Opening Stream");
        exit(1);
}
```

Using the NLI Conversion Module

IRIS SX.25 provides an "NLI Conversion Module" which allows older NLI applications to run, without modification, over a new version of the NLI. This situation might arise in porting an application from another platform that supports the older NLI version. The module ensures binary compatibility between applications that utilize an older version of the NLI than that supported by the current release of IRIS SX.25. Newly developed applications for IRIS SX.25 have no need for the conversion module.

The IRIS SX.25 conversion module provided for this release, **s_nli3**, converts version 3 applications to the future versions of the NLI. An **ioctl** command, *N_getnliversion*, is used to get the current version of the NLI. If the NLI applications are older than the current network, then the appropriate conversion module can be directly pushed onto the protocol stack. An example is given below.

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stream.h>
#include <stropts.h>
#include <sys/snet/uint.h>
#include <sys/snet/x25_proto.h>
#include <sys/snet/ll_proto.h>
#include <sys/snet/x25_control.h>

#define OPENFLAGS O_RDWR

int              fd;
struct nliformat versioninfo;
struct strioctl  nioctl;

/*
    Open a stream to the device using the flags
    passed  to  the  open routine.
*/
if ((fd = open("/dev/x25", OPENFLAGS)) < 0)
{
    perror("failed to open stream");
    return(-1);
}
```

```
/*
    Send an ioctl  message requesting the current NLI
    version number.

    If the ioctl fails, then return -1 to fail the open.

    If the ioctl succeeds, then compare the returned version
    number with the library version number. There are 3
    possible cases:

        i) if a value less than the library NLI version
           number is returned then an attempt has been made
           to run an application over an X.25 driver which
           does not support multiple NLI versions.
       ii) if a value equal to the library NLI version number
           is returned then no further action is required.
      iii) otherwise a version conversion is required, so
           push on a module that converts between the
           library NLI version and the X.25 driver version.
*/

versioninfo.version = 0;

nioctl.ic_cmd = N_getnliversion;
nioctl.ic_timout = 0;
nioctl.ic_len = sizeof(struct nliformat);
nioctl.ic_dp = (char *)&versioninfo;

if (ioctl(fd, I_STR, &nioctl) < 0)
{
    perror("N_getnliversion ioctl failed");
    return(-1);
}

if (versioninfo.version < NLI_VERSION)
{
   fprintf(stderr,"X.25 driver is older than application\n”);
   return(-1);
}
else
if (versioninfo.version > NLI_VERSION)
{
    if (ioctl(fd, I_PUSH, "s_nli3") < 0)
    {
        perror("Failed to push conversion module");
        return(-1);
    }
}
```

```
/*
    If neither of the above cases is TRUE then there is no
    need to  PUSH the module as the application has the same
    NLI version as the network
*/

return(fd);
}
```

Also provided is an NLI library, which allows application software to have access to the X.25 PLP driver without having detailed knowledge of the operation of the network providers services. The **n_open** routine ensures that the correct conversion module is pushed onto the protocol stack.

The **n_open** routine is used as follows:

```
if ((x25_fd = n_open("/dev/x25", O_RDWR, NULL)) < 0)
{
    perror("Opening Stream");
    exit(1);
}
```

The first two parameters are the same as for any STREAMS **open** routine, namely the device name and the open flags. The third is the **service** argument, which is used to return the service characteristics of the network provider. This argument should be of the type:

```
(struct n_info *)
```

This service facility is currently unsupported. Setting the parameter to NULL means the parameter is ignored. If the conversion fails, –1 is returned. Any application using the **n_open** routine should "link in" the appropriate NLI library for the release of the NLI they are using.

Opening a Connection

To establish a connection on an open stream, an application must do the following:

1. Allocate a Connect Request structure.
2. Supply the Connect Request with the quality of service and facilities parameters.

3. Set the called (and optionally calling) addresses.
4. Pass the Connect Request down to the X.25 driver.
5. Wait for the connect confirmation or rejection.

The following sections describe the procedures for opening a connection for a CONS call and for a non-CONS call, respectively.

CONS Calls

The following example opens a connection for a CONS call:

```
#define FALSE   0
#define TRUE    1

#include <memory.h>
#include <sys/snet/x25_proto.h>

struct xaddrf called =
    { 0, 0, {14, { 0x23,  0x42,  0x31, 0x56, 0x56, 0x56,
    0x56 }}, 0};
    /* Subnetwork "A" (filled in later), no flags,
       DTE = "23423156565656", null NSAP */

struct xcallf   conreq;

/* Convert sn_id to internal format */
called.sn_id = snidtox25("A");   /* snidtox25  only  fails
                                    if a NULL string is
                                    passed to it */
conreq.xl_type = XL_CTL;
conreq.xl_command = N_CI;
conreq.CONS_call = TRUE;         /* This is a CONS call */
conreq.negotiate_qos = TRUE;     /* Negotiate requested */

memset(&conreq.qos, 0, sizeof(struct qosformat));
conreq.qos.reqexpedited = TRUE;  /* Expedited requested */
conreq.qos.xtras.locpacket = 8;  /* 256  bytes  */
conreq.qos.xtras.rempacket = 8;  /* 256 bytes */
memcpy(&conreq.calledaddr, &called, sizeof(struct xaddrf));
memset(&conreq.callingaddr, 0, sizeof(struct xaddrf));
```

Note: When **negotiate_qos** is true (nonzero), setting the fields to zero means that the connection uses defaults for QOS and facilities. If required, these can be set to different values, but it is recommended that the **whole** QOS structure be zeroed first as shown. This is preferable to setting each field individually, as it allows for any future additions to this structure. Setting the calling address to null leaves the network to fill in this value.

The message is then sent on the stream using the *putmsg* system call, with any Call User Data (CUD) being passed in the data part of the message:

```
#define CUDFLEN  4

struct strbuf ctlblk, datblk;
char          cudf[CUDFLEN] = { 1, 0, 0, 0 };

ctlblk.len = sizeof(struct  xcallf);
ctlblk.buf = (char *) &conreq;
datblk.len = CUDFLEN;
datblk.buf = cudf;

if (putmsg(x25_fd, &ctlblk, &datblk, 0) < 0 )
{
    perror("Call putmsg");
    exit(1);
}
```

At this stage, the application should wait for a response to the call request. The response may be either a Connect Confirmation or a Disconnect Indication (rejection) message. (The second #define below may be shown wrapped; it must be on one line in the source file.)

```
#define    DBUFSIZ    128
#define CBUFSIZ MAX(sizeof(struct xccnff),sizeof(struct
xdiscf))

int              getflags = 0;
S_X25_HDR        *ind_msg;
char             ctlbuf[CBUFSIZ], datbuf[DBUFSIZ];

struct xccnff    *ccnf;
struct qosformat qos;
```

```
ctlblk.maxlen = CBUFSIZ;
ctlblk.buf    = ctlbuf;
datblk.maxlen = DBUFSIZ;
datblk.buf    = datbuf;

for(;;)
{
    if (getmsg(x25_fd, &ctlblk, &datblk, &getflags) < 0)
    {
        perror("Getmsg fail");
        exit(1);
    }
    ind_msg = (S_X25_HDR *) ctlbuf;
    if (ind_msg->xl_type != XL_CTL)
        continue;
    switch (ind_msg->xl_command)
    {
        case N_CC:
            /* ........ Process the Connect Confirmation */
            ccnf = ((struct xccnff *) ind_msg;
            if (ccnf -> negotiate_qos)
            {
                bcopy (&qos, ccuf->qos,
                    sizeof (struct qosformat));
                if (qos->reqexpedited)
                    printf("Request Expedited set\n");
                else
                    printf("Request Expedited not set\n");
            }
            else
            {
            /* indicated values have been accepted */
            }
            return;
        case  N_DI:
            perror("Connection rejected");
            exit(1);
        default:
            continue;
    }
}
```

In the example, *getmsg* is used to retrieve the next message from the stream head. This is done in a loop, until the application receives either a Connect Confirmation message, indicating successful completion, or a Disconnect Indication, showing that the connect attempt was rejected.

Note: The facility and QOS values indicated in the Connect Confirmation are those that are used for the duration of the connection.

It is possible to abort the connect request before a response is received. The application can do this by sending a Disconnect Request message (see the section "Closing a Connection" in this chapter). If this is done, the application should read and discard all messages from the stream until it receives the disconnect acknowledgment (described in the section "Disconnect Request/Indication" in this chapter).

After a rejection or connect abort the stream remains open, and can be used, for example, to make further connection attempts.

Non-CONS Calls

The following example opens a connection for a non-CONS call:

```
#define FALSE   0
#define TRUE    1

#include <memory.h>
#include <sys/snet/x25_proto.h>

struct xaddrf   called =
    { 0, 0, { 14, { 0x23, 0x42, 0x31, 0x56, 0x56, 0x56,
    0x56  }}, 0 };
    /* Subnetwork "A" (filled in later), no flags,
       DTE = "23423156565656", null NSAP */
struct xcallf   conreq;

/* Convert sn_id to internal format */
called.sn_id = snidtox25("A");
conreq.xl_type = XL_CTL;
conreq.xl_command = N_CI;
conreq.CONS_call = FALSE;      /* This is not a CONS call */
conreq.negotiate_qos = FALSE;  /* Just  use  default  */
```

```
memset(&conreq.qos, 0, sizeof(struct qosformat));
memcpy(&conreq.calledaddr, &called, sizeof(struct xaddrf));
memset(&conreq.callingaddr, 0, sizeof(struct xaddrf));
```

Note: When **negotiate_qos** is true (nonzero), setting the fields to zero means that the connection uses defaults for QOS and Facilities. If required, these can be set to different values (see the sections "Quality of Service and X.25 Facilities" in Chapter 2 and "Connect Request/Indication" in Chapter 4 for more details). However, it is recommended that the **whole** QOS structure be zeroed first, as shown. This is preferable to setting each field individually, as it allows for any future additions to this structure. Setting the calling address to null leaves the network to fill this value in.

The message is sent on the stream using the *putmsg* system call, with any CUD being passed in the data part of the message:

```
#define CUDFLEN  4

struct strbuf   ctlblk, datblk;
char            cudf[CUDFLEN] = { 1, 0, 0, 0 };

ctlblk.len = sizeof(struct  xcallf);
ctlblk.buf = (char *) &conreq;

datblk.len = CUDFLEN;
datblk.buf = cudf;

if (putmsg(x25_fd, &ctlblk, &datblk, 0) < 0 )
{
    perror("Call putmsg");
    exit(1);
}
```

Data Transfer

In the data transfer phase, access is given to:

- The Q-bit—to support X.29-like services
- The M-bit—to signal packet fragmentation
- The D-bit—to request confirmation of data delivery
- Expedited Data—to support X.29 and CONS

Normal and Q-bit data is sent and received using the Data (**N_Data**) message and may be acknowledged using the Data Acknowledgment (**N_DAck**) message. Expedited Data uses the Expedited Data (**N_EData**) message, and is acknowledged using the Expedited Data Acknowledgment (**N_EAck**) message.

The following sections show examples of code for data transfer.

Sending Data

Once a connection has been successfully opened on a stream, sending a data packet is straightforward:

```
#define  DBUFSIZ  128

struct xdataf data;
char          datbuf[DBUFSIZ];
int           retval;

/* Copy data into datbuf[] here */
data.xl_type = XL_DAT;
data.xl_command = N_Data;
data.More = data.setQbit = data.setDbit = FALSE;

ctlblk.len = sizeof(struct xdataf);
ctlblk.buf = (char *) &data;
datblk.len = DBUFSIZ;
datblk.buf = datbuf;

retval = putmsg(x25_fd, &ctlblk, &datblk, 0);
```

Normally, the call to *putmsg* is blocked if there are flow control conditions in the connection, which lead to either a full queue at the stream head, or a lack of STREAMS resources. Blocking due to a full queue can be avoided if the stream is opened with the option O_NDELAY flagged. In this case, *putmsg* returns immediately, and the failure is signalled by a return value (**retval**) of EAGAIN.

This procedure allows the application to carry out other processing (for example, receiving data) before trying again. The best method to use depends on the nature of the application.

Receiving Data

In the same way, data reception is straightforward. When data is received with the D-bit set, action may be required by the application. When the initial Connect Request is sent, it may request that data confirmation be at the application-to-application level. If application-to-application confirmation is agreed upon, then on receiving a packet with the D-bit set, the application must acknowledge the packet by sending a Data Acknowledgment message.

This example prints out incoming data as a string, if the Q-bit is not set:

```
S_X25_HDR      *hdrptr;
struct xdataf  *dat_msg;
struct xdatacf *dack;

for (;;)
{
    if (getmsg(x25_fd, &ctlblk, &datblk, &getflags) < 0)
    {
        perror("Getmsg fail");
        exit(1);
    }
    hdrptr = (S_X25_HDR *) ctlbuf;
    if (hdrptr->xl_type == XL_CTL)
    {
        /* Deal with protocol message as required -
           see below */
    }
    if (hdrptr->xl_type == XL_DAT)
    {
        dat_msg = (struct xdataf *) ctlbuf;
        switch (dat_msg->xl_command)
        {
            case N_Data:
                if (dat_msg->More)
                    printf("M-bit set\n");
                if (dat_msg->setQbit)
                    printf("Q-bit set\n");
                else
                {
                    if (dat_msg- >setDbit)
                        printf("D-bit set\n");
                    for (i =  1;i<datblk.len;  i++)
```

```
                        printf("%c", datbuf[i]);
                    /* If application to application Dbit
                       confirmation was negotiated
                       at call setup time, send an N_DAck */
                    if (app_to_app && dat_msg->setDbit)
                    {
                        dack = (struct xdatacf  *)
                            malloc(sizeof(struct xdatacf));
                        bzero((char *)dack, sizeof(struct
                            xdatacf));
                        dack->xl_command = N_DAck;
                        dack->xl_type = XL_DAT;
                        ctlblk->len = sizeof(struct xdatacf);
                        ctlblk->buf = (char *)dack;
                        datblk->len = 0;
                        datblk->buf = (char *)0;
                        putmsg(x25_fd, &ctlblk, &datblk,
                            &getflags);
                    }
                }
                break;
            case N_EData:
                printf("***Expedited data received\n");
                /* Must deal with */
                break;
            case N_DAck:
               printf("***Data Acknowledgement received\n");
               break;
            default:
                break;
        }
    }
}
```

Expedited Data

The above example allows for the possibility of receiving Expedited Data messages (which are carried in X.25 interrupt packets). These must be dealt with appropriately. Since only one Expedited Data packet can be outstanding in the connection at any time, its sender is prevented from sending any further such messages until the receiver has acknowledged it. It does this by sending an Expedited Data Acknowledgment message.

This is sent in much the same way as an ordinary Data packet, but with no data part. If the application does not need to use the Expedited Data capability, then other appropriate responses to receiving an Expedited Data message are to reset or to close the connection (see the sections “Resets” and “Closing a Connection” in this chapter).

When sending Expedited Data, the application must wait for an acknowledgment before requesting further expedited transmissions.

```
#include        <sys/snet/x25_proto.h>
#define         EXPLEN  4

struct xedataf exp;
char   expdata[]= {1, 2, 3, 4};

exp.xl_type = XL_CTL;
exp.xl_command  = N_Edata;
ctlblk.len      = sizeof (struct xedataf);
ctlblk.buf      = (char *) &exp;
datblk.len      = EXPLEN;
datblk.buf      = expdata;

if (putmsg(x25_fd, &ctlblk, &datblk, 0) < 0)
{
    error("Exp putmsg");
    exit(1);
}
for (;;)
{
    if (getmsg(x25_fd, &ctlblk, &datblk, &getflags) < 0)
    {
        perror("Getmsg fail");
        exit(1);
    }
    hdrptr = (S_X25_HDR *) ctlbuf;
    if (hdrptr->xl_type == XL_CTL)
    {
        /* Deal with protocol message as required  */
    }
    if (hdrptr->xl_type == XL_DAT)
    {
        dat_msg = (struct xdataf *) ctlbuf;
```

```
        switch (dat_msg->xl_command)
        {
            case N_Data:
                /* process more data */
                break;
            case N_EData:
                printf("***Expedited data received \n");
                /* Must deal with */
                .... send N_EAck ....
                break;
            case N_EAck:
                /* Expedited data received */
                /* Further  N_Edata  can  now be sent */
                break;
            default:
                break;
        }
    }
}
```

Resets

These can be dealt with in a way similar to the way interrupts are dealt with, except that there is no data passed with a Reset Request. When a Reset Request is issued, the application must wait for the acknowledgment, as for an Expedited Data request. However, until this is received, the **only** action that can be taken is to issue a Disconnect Request.

The diagnostic field in a Reset Request should be filled in with the reason for issuing the reset. Standard values for this are defined in the include file <*sys/snet/x25_proto.h*>, although the application can set any value. See Appendix B, “Error Codes,” for more details.

When a Reset Indication is received, there are only two valid actions that may be taken:

- Send a Reset Confirmation message to acknowledge the reset.
- Send a Disconnect Request. In this situation, pending data is flushed from the queue.

Reset Indications can be dealt with as part of the general processing of incoming messages—see the Disconnect handling example below.

```
#include        <sys/snet/x25_proto.h>

struct xrstf    rst;
S_X25_HDR       *hdrptr;

rst.xl_type     = XL_CTL;
rst.xl_command  = N_RI;
rst.cause       = 0;
rst.diag        = NU_RESYNC;
ctlblk.len      = sizeof (struct rstf);
ctlblk.buf      = (char *) &rst;

if (putmsg(x25_fd, &ctlblk, 0, 0) < 0)
{
    perror(" putnmsg");
    exit(1);
}
for (;;)
{
    if (getmsg(x25_fd, &ctlblk, &datblk, &getflags) < 0)
    {
        perror("Getmsg fail");
        exit(1);
    }
    hdrptr = (S_X25_HDR *) ctlbuf;
    if (hdrptr->xl_type  ==  XL_CTL)
    {
        continue;
    }
    switch (hdrptr->xl_command)
    {
        case N_RC:
            /* Reset complete */
            /* Enter data transfer */
            break;
        default:
            break;
    }
}
```

Control messages like resets and interrupts take higher priority than normal data messages, both internally in the PLP driver and across the network.

However, it is important to note that the NLI does not use the mechanism for priority processing of STREAMS messages (by setting the RS_HIPRI flag in *putmsg*). There are two reasons for this:

- The stream head can hold only one incoming priority message (the first). This is inappropriate in certain situations where several of these messages may follow each other in quick succession. For example, a Reset may be followed immediately by a Disconnect.
- An outgoing priority message would overtake any data that is queued, waiting to be sent. It is possible that data could then be sent *after* the priority message (for example, a reset), which would lead to an NLI protocol violation.

Closing a Connection

This section covers remote and local disconnects.

Remote Disconnect

If, during a connection, the remote end initiates a disconnect, then a Disconnect Indication message is received at the NLI (or possibly an Abort Indication message—see the section "Abort Indication" in Chapter 4). The application need not acknowledge this message since, after sending a Disconnect Indication, the X.25 driver silently discards all messages received except for connect and accept messages. These are the only meaningful X.25 messages on the stream after disconnection.

The receiver of a Disconnect Indication should ensure that enough room is available in the *getmsg* call to receive all parameters and, when present, up to 128 bytes of clear user data.

Handling such a disconnect event would normally be part of the general processing of incoming messages.

The example below could be combined with the code from the data transfer example shown above.

```
struct xdiscf *dis_msg;

if (hdrptr->xl_type == XL_CTL)
{
    switch (hdrptr->xl_command)
    {
        /* Other  events/indications  dealt  with
           here - e.g. Reset Indication (N_RI) */
        case N_DI:
            dis_msg = (struct xdiscf *) hdrptr;
            printf("Remote disconnect,
                cause = %x, diagnostic = %x \n",
                dis_msg->cause, dis_msg->diag);
            /* Any other processing needed here -
               e.g. change connection state */
            return;
        case N_Abort:
            printf("***Connection\n");
            /* etc. */
            return;
        default:
            break;
    }
}
```

Note: It is **guaranteed** that no X.25 interface messages are sent to the application once a disconnect message has been passed up to it, wherever the message came from. That is, it can be a Disconnect Indication or the "response" described in the section "Local Disconnect" in this chapter).

Although at this stage the stream is idle, it is in an open state and remains so until some user action. This could be to close the stream, or to initiate a new Listen or Connect Request on it.

Local Disconnect

To initiate a disconnect on a connection, the application should send a Disconnect Request message on the stream. Unless this is being used to reject an incoming call (see the section "Handling the Connect Indication" in this chapter), the X.25 driver signals that it has observed the message. It does this by sending a Disconnect Confirmation upstream when it receives the Disconnect Request. In this way, the upper components can be certain that no messages will follow the Disconnect Request.

In the case of rejection, the connection identifier supplied on the Connect Indication must be returned in the Disconnect Indication message. The Disconnect Request (reject) is not acknowledged in this case.

As in the case of a remote disconnection, once the response has been received the stream becomes idle, and remains in this state until the application sends out another control message. This may be to close the stream, or to initiate a new Listen or Connect Request on it. The application should, however, not send any of these messages until it receives the Disconnect Indication.

As described in the section "Disconnect Request/Indication" in Chapter 4, a disconnect collision may occur. If this happens, no Disconnect Confirmation is sent.

```
/* Coded and sent disconnect request, process response */
struct xdiscf *dis_ind;
struct xdcnff *dis_cnf;
struct extraformat *xqos = (struct extraformat *)0;

if (hdrptr->xl_type == XL_CTL)
{
    switch (hdrptr->xl_command)
    {
        /* Disconnect Collision */
        case N_DI:
            dis_ind = (struct xdiscf*) hdrptr;
            xqos = &dis_ind->indicatedqos.xtras;
            break;
```

```
        /* Disconnect Confirmation */
        case N_DC:
            dis_cnf = (struct xdcnff*)hdrptr;
            xqos = &dis_cnf->indicatedqos.xtras;
            break;
        default:
            return;
    }
    if (xqos)
    {
        /* Print  any  charging  information  returned */
        if (xqos->chg_cd_len)
        {
            /* Print out Call Duration from chg_cd_field */
        }
        if (xqos->chg_mu_len)
        {
            /* Print out Monetary Unit from chg_mu_field */
        }
        if (xqos->chg_sc_len)
        {
            /* Print out Segment Count from chg_sc_field */
        }
    }
}
```

Listening

For more information on listening, see Chapter 3, “Listens.”

Listening for Incoming Connections

Before an incoming call can be received from the X.25 driver, there must be at least one **listener**. Moreover, as mentioned in the section “Priority” in Chapter 3, listening for incoming connections may be a privileged operation—that is, the stream must have been opened by a process with *superuser* privilege.

To listen for an incoming connection, the application does the following:

1. Sends a Listen Request message carrying the called address list that the application is interested in to the X.25 driver (see Chapter 3, “Listens”). After this, the application waits for the response to the Listen Request.
2. When the Listen Response is received (and the **l_result** flag indicates success), wait for Connect Indication messages from the X.25 driver. If the **l_result** flag indicates failure, the application can decide either to close the stream or to try again later.
3. When a Connect Indication is passed up, the application can decide whether to accept on this or a different stream.
4. At this point, the facilities and QOS are negotiated if required. A Connect Confirmation message carrying the appropriate connection identifier is passed down on the stream on which the connection is being accepted.

Constructing the Listen Message

As described in Chapter 3, “Listens,” the listen message has two parts. The construction of the control part of the message is straightforward:

```
struct xlistenf     lisreq;

lisreq.xl_type = XL_CTL;
lisreq.xl_command = N_XListen;
lisreq.lmax = 1;
```

In this example, **lmax** has the value of 1, indicating that only one Connect Indication is to be handled at a time.

The data part of the message should be filled with the sequence of bytes that specifies the CUD string and address(es) which are to be listened for. The simplest case for this would be to set “Don't Care” values for both the CUD and address:

```
int lislen;
char        lisbuf[MAXLIS];

lisbuf[0] = X25_DONTCARE; /* l_cumode */
lisbuf[1] = X25_DONTCARE; /* l_mode   */
lislen = 2;
```

Alternatively, to set the CUD to match exactly the (X.29) value defined in the array *cudf[]* earlier (0x01000000), and the NSAP to match any sequence starting "0x80", "0x00", the following would be used:

```
lislen = 0;

lisbuf[lislen++] = X25_IDENTITY;          /* l_cumode   */
lisbuf[lislen++] = CUDFLEN;               /* l_culength */
memcpy(&(lisbuf[lislen]), cudf, CUDFLEN); /* l_cubytes  */
lislen += CUDFLEN;
lisbuf[lislen++] = X25_STARTSWITH;  /* l_mode   */
lisbuf[lislen++] = X25_NSAP;        /* l_type   */
lisbuf[lislen++] = 4;               /* l_length */
lisbuf[lislen++] = 0x80;            /* l_add    */
lisbuf[lislen++] = 0x00;
```

Or, to accept any CUD field, with a DTE of "2342315656565":

```
#define     MY_DTE_LEN      13
#define     MY_DTE_OCTETS   7

char my_dte[MY_DTE_OCTETS] =
    {0x23,0x42,0x31,0x56,0x56,0x56,0x50};

lislen = 0;
lisbuf[lislen++] = X25_DONTCARE;    /* l_cumode */
lisbuf[lislen++] = X25_IDENTITY;    /* l_mode   */
lisbuf[lislen++] = X25_DTE;         /* l_type   */
lisbuf[lislen++] = MY_DTE_LEN;      /* l_length */
memcpy(&(lisbuf[lislen]), my_dte, MY_DTE_OCTETS);  /* l_add*/
lislen += MY_DTE_OCTETS;
```

Note: The **l_add** field uses packed hexadecimal digits and the **l_length** value is actually the number of *semi-octets*, whereas the **l_culength** field specifies the length of the **l_cubytes** field in *octets*.

Next, send the Listen Request down the open stream:

```
ctlblk.len = sizeof(struct xlistenf);
ctlblk.buf = (char *) &lisreq;
datblk.len = lislen;
datblk.buf = lisbuf;
```

```
if (putmsg(x25_fd, &ctlblk, &datblk, 0) < 0)
{
    perror("Listen putmsg failure");
    return -1;
}
```

Finally, wait for the Listen Response—the result flag indicates success or failure (the second #define below may be shown wrapped; it must be on one line in the source file):

```
#define    DBUFSIZ    128
#define    CBUFSIZ    MAX(sizeof(struct
xccnff),sizeof(struct xdiscf))

struct xlistenf *lis_msg;

ctlblk.maxlen = CBUFSIZ; /* See 4.1 above for declarations */
ctlblk.buf  =  ctlbuf;
datblk.maxlen  =  DBUFSIZ;
datblk.buf  = datbuf;

for (;;)
{
    if (getmsg(x25_fd, &ctlblk, &datblk, &getflags) < 0)
    {
        perror("Listen getmsg failure");
        return -1;
    }
    lis_msg = (struct xlistenf *) ctlbuf;
    if ((lis_msg->xl_type  ==  XL_CTL) &&
        (lis_msg->xl_command == N_XListen))
        if (lis_msg->l_result != 0)
        {
            printf ("Listen command failed\n");
            return -1;
        }
        else
        {
            printf("Listen command succeeded\n");
            return 0;
        }
}
```

Cancelling a Listen Request can be done in the same way, except that no data is passed with the request—it simply cancels all successful Listen Requests that have been made on that stream.

Handling the Connect Indication

Once the listening application has received a Listen Response indicating success, it should wait for incoming Connect Indications.

When a Connect Indication message arrives, the application should inspect its parameters—address, CUD, facilities, quality of service, and so on, then decide whether to accept or reject the connection.

Acceptance

If accepting, it can do so either on the stream the indication arrived on, or on some other stream. This other stream can be one that is already open and free, or it can be newly opened.

Whatever method is used for the accept, the identifier **conn_id** in the Connect Indication message **must** be copied into the accept message for matching by the X.25 driver. If this identifier in the accept message does not match, a Disconnect Request is sent to the accepting application. This causes the resource to hang on the stream on which the incoming call was sent, since the connection is never accepted.

Rejection

The call can be rejected by sending a Disconnect Request message down the stream on which the Connect Indication arrived. A Connect Indication cannot be rejected on a different stream. Again, the connection identifier **must** be quoted in the message for matching, since there may be several Connect Indications passed to the listening application. If there is no match for the rejection, the message is silently discarded.

The rejecting listener can request one of two actions in response to the disconnect:

- Request immediate disconnect. Set the reason field to NU_PERMANENT (0xF5).

- Search for further matching listeners. Set the reason field to any value except 0xF5.

The following code example shows how to reject an incoming call:

```
struct xcallf *conind;
struct xdiscf disc_msg;

/* Use getmsg to receive the Connect Indication,
   use conind to point to it */
disc_msg.xl_type = XL_CTL;
disc_msg.xl_command = N_DI;
disc_msg.conind = conind->conind;
disc_msg.cause = cause;    /* cause to be returned */
disc_msg.diag = diag;      /* diagnostic to be returned */

if (disc_immed)            /* no more searches */
    disc_msg.reason = NU_PERMANENT;    /* 0xF5 */

/* Send Rejection down stream with putmsg */
```

Note: The application must not accept a connection on a listening stream that is capable of handling more than one Connect Indication at one time if there could subsequently be other Connect Indications to be handled on that stream. For example, suppose the application issues a Listen Request to handle three Connect Indications at one time. A Connect Indication is received and sent to the application on the listen stream. The application must **not** accept this connection on the listen stream because there could be two more Connect Indications that could be sent subsequently.

Negotiation of QOS Parameters

The Connect Indication message passed contains X.25 facility values, and CONS QOS parameters, if appropriate. The application may want to negotiate these values. This is done by setting the **negotiate_qos** flag in the Connect Response message. The values received should then be copied into the response, and those facilities and/or parameters (and any related flags) for which a different value is desired should then be altered (see the section "Quality of Service and X.25 Facilities" in Chapter 2). It is recommended that the **whole** QOS structure be copied from the indication to the response. This is preferable to copying each field individually, as it allows for any future additions to this structure.

An example of negotiation is shown below. Here all the values are copied as indicated, except the packet size, which is negotiated down to 256 if it is flagged as negotiable, and is greater than 256:

```
struct xcallf   *conind;
struct xccnff   conresp;

/* Do a getmsg etc to receive the Connect Indication,
   assign conind to point to it.*/
conresp.xl_type = XL_CTL;
conresp.xl_command = N_CC;
conresp.conn_id = conind->conn_id; /* Connection identifier*/
conresp.CONS_call = TRUE           /* This is a CONS call */

memset(&conresp.responder, 0, sizeof(struct xaddrf));
                   /* Let network fill in responding addr */
conresp.negotiate_qos = TRUE;
memcpy(&conresp.rqos, &conind->qos,sizeof(struct qosformat));
if (conind->qos.xtras.pwoptions & NEGOT_PKT)
{
    if (conind->qos.xtras.rempacket > 8)
        conresp.rqos.xtras.rempacket = 8;   /* 256 = 28 */
    if (conind->qos.xtras.locpacket > 8)
        conresp.rqos.xtras.locpacket = 8; }

/* Set any other values to be negotiated here,
   then send the response down with a putmsg. */
```

Alternatively, the application may decide to accept (agree with) the indicated values, in which case the **negotiate_qos** flag is set to zero.

Reusing the Listen Stream

If a connection is never established on a listening stream (using a matching accept) then that stream remains listening on the address list supplied. On the other hand, once an established connection has been disconnected, the stream does not return to a listening state. Instead, it remains open in an idle state. If the application needs to listen again, then the listen message must be re-sent. Rejection does not alter the listening state of the stream.

PVC Operation

The following subsections describe the procedures necessary for an application to operate a PVC on the X.25 PLP driver.

Attaching a PVC

To attach a PVC on an open stream, an application must:

1. Allocate a PVC Attach structure.
2. Supply the structure with the appropriate **reqackservice** and **reqnsdulimit** parameters. These parameters are used for the duration of the connection.
3. Set the appropriate subnetwork and Logical Channel Identifiers.
4. Pass the attach request down to the X.25 driver.
5. Wait for the attach accept or rejection.

For example:

```
#include <sys/stropts.h>
#include <sys/snet/x25_proto.h>

struct pvcattf attach = {XL_CTL, N_PVC_ATTACH,1,0,0,0,0};
    /* Subnetwork "A" (filled in later), Logical Channel 1
       No request for Receipt Ack or nsdulimit */
struct strbuf ctlblk;

/* Convert sn_id to internal format */
attach.sn_id = snidtox25("A");
ctlblk.len = sizeof(struct pvcattf);
ctlblk.buf = (char *) &attach;
```

The message is then sent on the stream using the *putmsg* system call:

```
if (putmsg(x25_fd, &ctlblk, 0, 0) < 0)
{
    perror("Attach putmsg");
    exit(1);
}
```

At this stage, the application should wait for a response to the attach. The response may indicate either a successful attachment or a rejection.

```
#define    DBUFSIZ     128
#define    CBUFSIZ     sizeof(struct pvcattf)

int             getflags;
struct pvcattf  *ind_msg;
char            ctlbuf[CBUFSIZ], datbuf[DBUFSIZ];

ctlblk.maxlen = CBUFSIZ;
ctlblk.buf    = ctlbuf;
datblk.maxlen = DBUFSIZ;
datblk.buf    = datbuf;

for (;;)
{
    if (getmsg(x25_fd, &ctlblk, &datblk, &getflags) < 0)
    {
        perror("Getmsg fail");
        exit(1);
    }
    ind_msg = (struct pvcattf *) ctlbuf;
    if (ind_msg->xl_type != XL_CTL)
        continue;
    switch (ind_msg->xl_command)
    {
        case N_PVC_ATTACH:
            switch (ind_msg->result_code)
            {
                case PVC_SUCCESS:
                    /* ...... Process the attach */
                    return(1);
                case PVC_NOSUCHSUBNET:
                case PVC_CFGERROR:
                case PVC_PARERROR:
                case PVC_BUSY:
                    /*  ...... Process the reject */
                    return(0);
                default:
                    printf("Unknown PVC message\n");
                    exit(1);
            }
    }
}
```

In this example, *getmsg* is used to retrieve the next message from the stream head. This is done in a loop, until the attach is either confirmed successful or rejected. Although the processing of the attach is not shown here, it is recommended that the application send a Reset Request (see the section "Reset Request/Indication" in Chapter 4) and wait for the Reset Confirmation (see the section "Reset Response/Confirmation" in Chapter 4) before proceeding with the data transfer. The example given in the section "Resets" in this chapter shows the code used to send a Reset Request and handle the acknowledgment. This synchronizes the X.25 PLP drivers at each end of the PVC. The example does not illustrate all possible **result_code** cases.

It is possible to abort the Attach Request before a response is received. The application can do this by sending a PVC Detach message (see the section "Detaching a PVC" below). If this is done, the application should read and discard all messages from the stream until it receives the detach acknowledgment.

After a rejection or an attach abort the stream remains open and can be used, for example, to make further attach attempts.

PVC Data Transfer

The transfer of data over a permanent virtual circuit is exactly the same, to the application, as for virtual circuits. See the section "Data Transfer" in this chapter for a description of the procedures involved.

Detaching a PVC

The procedure used to detach a PVC differs for the remote and local cases, so these are described separately here.

Remote Detach

If, during a connection, the remote end initiates a detach, then a Reset Indication (see the section “Reset Request/Indication” in Chapter 4) message is received at NLI. The application should acknowledge this with a Reset Response (see the section “Reset Response/Confirmation” in Chapter 4).

Handling such an event would normally be part of the general processing of incoming messages.

After sending the Reset Response, the application is still attached to its PVC and remains so until it initiates a local detach.

Local Detach

To initiate a detach on a connection, the application should send a PVC Detach message on the stream. The X.25 driver signals that it has observed the message by sending a PVC Detach upstream. In this way, the upper component can be certain that no messages follow the PVC Detach.

For example:

```
struct pvcdetf detach = { XL_CTL, N_PVC_DETACH, 0 };

ctlblk.len = sizeof(struct pvcdetf);
ctlblk.buf = (char *) &detach;

if (putmsg(x25_fd, &ctlblk, 0, 0) < 0)
{
    perror("Detach putmsg");
    exit(1);
}
```

As is the case for a remote detach, the stream becomes idle once the response has been received. It enters an open state, in which it remains until the application commands otherwise. This could be to close the stream, or to initiate a new PVC Attach on it. The application should, however, wait until it receives the PVC Detach.

Appendix A

NLI Messages

This appendix provides tables of the NLI Messages and their associated downstream and upstream X.25 packets.

Appendix A

NLI Messages

Table A-1 lists the downstream messages and associated outgoing X.25 packets.

Table A-1 Downstream Messages and Associated Outgoing X.25 Packets

NLI Message	X.25 Packet
N_CI	Connect Request
N_CC	Connect Response
N_Data	Data
N_DAck	Data Acknowledgment
N_EData	Expedited Data
N_EAck	Expedited Data Acknowledgment
N_RI	Reset Request
N_RC	Reset Response
N_DI	Disconnect Request

Table A-2 lists the upstream messages and associated incoming X.25 packets.

Table A-2 Upstream Messages and Associated Incoming X.25 Packets

NLI Message	X.25 Packet
N_CI	Connect Indication
N_CC	Connect Confirmation
N_Data	Data
N_DAck	Data Acknowledgment

Table A-2 (continued) Upstream Messages and Associated Incoming X.25

NLI Message	X.25 Packet
N_EData	Expedited Data
N_EAck	Expedited Data Acknowledgment
N_RI	Reset Indication
N_RC	Reset Confirmation
N_DI	Disconnect Indication
N_DC	Disconnect Confirmation

Note: The NLI PVC messages PVC Attach and PVC Detach do not have corresponding X.25 packets.

Appendix B

Error Codes

This appendix lists the OSI error codes that can be used by NLI application programs.

Appendix B

Error Codes

This section lists the OSI codes defined in <*sys/snet/x25_proto.h*> that can be used by NLI application programmers.

Table B-1 lists the originator codes in Disconnect Request/Indication and Reset Request/Indication messages.

Table B-1 N_RI and N_DI Originator Codes

Originator Code	Value
N_USER	1
N_PROVIDER	2

Table B-2 lists the reason codes when the originator is the Network Service provider in Disconnect Request/Indication messages.

Table B-2 N_DI Reason Codes for Network Service Providers

Reason Code	Value
NS_GENERIC	0xE0
NS_DTRANSIENT	0xE1
NS_DPERMANENT	0xE2
NS_TUNSPECIFIED	0xE3
NS_PUNSPECIFIED	0xE4
NS_QOSNATRANSIENT	0xE5
NS_QOSNAPERMANENT	0xE6
NS_NSAPTUNREACHABLE	0xE7

Table B-2 (continued) N_DI Reason Codes for Network Service Providers

Reason Code	Value
NS_NSAPPUNREACHABLE	0xE8
NS_NSAPPUNKNOWN	0xEB

Table B-3 lists the reason codes when the originator is the Network Service user in Disconnect Request/Indication messages.

Table B-3 N_DI Reason Codes for Network Users

Reason Code	Value
NU_GENERIC	0xF0
NU_DNORMAL	0xF1
NU_DABNORMAL	0xF2
NU_DINCOMPUSERDATA	0xF3
NU_TRANSIENT	0xF4
NU_PERMANENT	0xF5
NU_QOSNATRANSIENT	0xF6
NU_QOSNAPERMANENT	0xF7
NU_INCOMPUSERDATA	0xF8
NU_BADPROTID	0xF9

Table B-4 lists the reason codes when the originator is the Network Service provider in Reset Request/Indication messages.

Table B-4 N_RI Reason Codes for Network Service Providers

Reason Code	Value
NS_RUNSPECIFIED	0xE9
NS_RCONGESTION	0xEA

Table B-5 lists the reason codes when the originator is the Network Service user in Reset Request/Indication messages.

Table B-5 N_RI Reason Code For Network Service Users

Reason Code	Value
NU_RESYNC	0xFA

Note: These codes are found in ISO 8208 and are mapped from X.25 cause and diagnostic codes, as described in ISO 8878.

Glossary

CONS

Connection-Oriented Network Service.

CUD

Call User Data. Carried with a Connect Request.

DTE

Data Terminal Equipment. Used in this guide for the X.121 address of the line connecting the equipment to a packet-switched network.

ISO

International Organization for Standardization.

ISO 8208

Standard for X.25 Packet Level Protocol for Data Terminal Equipment.

ISO 8878

Standard that defines the use of X.25 to provide the OSI connection-mode network service.

LCI

Logical Channel Identifier.

LLC-2

Logical Link Control-Class II.

LSAP

Link Service Access Point.

MAC

Medium Access Control. Used in this guide to refer to the physical Ethernet address.

NLI

Network Layer Interface.

NSAP

Network Service Access Point. An ISO address, a string of 40 hexadecimal digits or semi-octets.

NSDU

Network Service Data Unit—the unit of data passed across the interface to the network layer.

NUI

Network User Identification.

Octet

Eight bits, a byte.

OSI

Open Systems Interconnection. The ISO definition of a communications system providing reliable, data transparent, host-independent communications service.

PVC

Permanent Virtual Circuit.

PLP

Packet Layer Protocol—X.25 Level III.

QOS

Quality of Service.

RPOA

Recognized Private Operating Agency.

SAP

Service Access Point.

Semi-Octet

Four bits, one hexadecimal digit.

SNPA

Subnetwork Point of Attachment.

STREAMS

STREAMS is a set of system calls, kernel resources, and utility routines that create, use and dismantle a stream.

Subnetwork

In this guide, this term refers to the network accessed via a single physical link or a single Ethernet interface.

TOA/NPI

Type Of Address/Numbering Plan Identification. An address format that provides increased addressing capacity suitable for communication with ISDNs.

Index

A

Abort Indication messages, 36
acceptable field, 9
addresses
  called, 5, 16, 21, 30, 47
  calling, 5, 21, 30, 47, 48, 51
  DTE, 6, 7
  format, 6
  Logical Channel Identifier, 6, 7
  MAC+SAP, 6, 7
  matching, 23
  modification, 16
  options to encode and interpret, 6
  responding, 5, 30, 35
  type, 23
  X.25, 5
aflags field, 6

C

c_result field, 38
call_deflection field, 16
call_redirect field, 16
Call Deflection facility, 12, 16, 35
called_add_mod field, 16
calledaddr field, 30
called field, 16
callingaddr field, 30
call redirection, 16
Call User Data. *See* CUD.
cause field, 33, 35
cd_fac_len field, 16
Charging Information facility, 16, 35, 36
chg_ fields, 16
Closed User Groups, 12, 15
closing a connection, 58
conn_id field, 29, 30, 65
Connect Confirmation
  facilities and QOS negotiation, 62
  messages, 30, 48, 75
  target transit delay, 9
Connect Indication
  and Expedited Data, 10
  and listening, 21, 24, 62
  handling, 65-67
  maximum number, 37
  messages, 29, 75
  rejection, 35
  subnetwork, 6
connection
  addresses, 5
  closing, 58-61
  priority, 10
connections
  identifiers, 29, 30, 35, 60, 65
  listening for, 61-67
  opening, 46-51
Connect Request
  Call Charging, 16
  limit for packet concatenation, 14

message, 29, 75
Network User Identification, 15
opening a connection, 46
subnetwork, 6
transit delay, 9
Connect Response
connection identifier, 29
messages, 30, 75
packet concatenation, 14
QOS parameters negotiation, 66
transit delay, 9
CONS
CONS_call, 11, 29, 30
Expedited Data, 51
facilities, 7
fast select, 13
negotiation, 11
opening a connection, 47-50
originator and **reason** codes, 33
quality of service. *See* quality of service.
reqackservice, 32
CONS_call field, 11, 29, 30
control messages, 28
conversion module, 44
CUD field
constructing the listen message, 62
listening for incoming calls, 21
matching Listen Request, 22
cug_field field, 15
cug_type field, 15
CUG. *See* Closed User Groups

D

Data Acknowledgment message, 31, 52, 53, 75
Data message, 31, 52, 75
data messages, 28
data structures, 5-17
data transfer phase, 51, 70
D-bit data and acknowledgments, 11, 31, 39, 51, 53
deflected field, 16, 35
diag field, 33, 35
diagnostic bytes, 33, 35, 56, 81
disconnect
collision, 36, 60
local, 60
rejecting calls, 65
remote, 58
See also Disconnect Confirmation, Disconnect Indication, Disconnect Request.
Disconnect Confirmation
charging information, 16
messages, 36, 60, 76
rejection, 35
Disconnect Indication
closing a connection, 58
example, 48
messages, 16, 34, 76
Disconnect Indication messages, 79, 80
Disconnect Request, 79
and resets, 56
call deflection, 16
local disconnects, 60
messages, 34, 75
reason codes, 80
when QOS is unattainable, 9
DTE
addressing format, 5
address length, 7
call deflection, 16, 35
call redirection, 16
DTE_MAC field, 6
remote receipt confirmation, 11, 39
X25_DTE addresses, 23

E

error codes, 79
Expedited Data
  acknowledging, 33
  data transfer, 54
  field in qosformat structure, 10
  message, 32, 52, 75, 76
  negotiation, 10
Expedited Data Acknowledgment
  data transfer, 52
  example, 54
  messages, 33, 75, 76
extended addresses, 6, 11, 23
extraformat structure, 12

F

fac_field field, 16
facilities
  Call Deflection, 12, 16, 35
  Closed User Groups, 12, 15
  CONS, 7
  extended addressing, 11
  fast select, 11, 13
  Network User Identification, 12, 15
  non-OSI procedures, 7, 11
  packet size negotiation, 11, 13, 67
  programmable, 12, 16
  Recognized Private Operating Agency, 12, 15
  request and negotiation, 7-17
  reverse charging, 11, 13
  window size negotiation, 12, 13
fast select facility, 11, 13
fastselreq field, 13

G

getmsg system call, 1, 50, 58, 70

I

include files, 2
incoming calls
  address matching, 23
  Call User Data matching, 22
  cancelling interest, 38
  Expedited Data, 11
  listening and privileged operation, 24
  listening for, 21, 61
  pwoptions field, 14
  registering interest, 37
  rejecting, 60, 66
indicated_qos field, 35, 36
interrupt
  control messages, 58
  data, 11
  packets, 32, 54
IRIS SX.25
  documentation, x
  Network Layer Interface, 1
  NLI Conversion Module, 44
  reference (manual) pages, x

L

l_ address fields, 23, 63
l_cu fields, 22
l_result field, 37, 62
lci field, 39

listen
  accepting calls, 65
  address matching, 23
  called addresses, 21
  Call User Data matching, 22
  cancelling an interest, 38
  constructing the listen message, 62
  for incoming calls, 21
  handling the connect indication, 65
  listening for incoming calls, 61
  major features, 21
  maximum number of Connect Indications, 37
  negotiating QOS parameters, 66
  priority, 24
  private information, 24
  privileged users, 24
  registering interest, 37
  rejecting calls, 65
  reusing the listen stream, 67
  See also Listen Request, Listen Response messages, Listen Cancel Request messages, Listen Cancel Response messages.
Listen Cancel Request messages, 38
Listen Cancel Response messages, 38
Listen Request
  address matching, 23
  Call User Data matching, 22
  cancelling, 38, 65
  listening for an incoming connection, 62
  messages, 37
  queue priority, 24
Listen Response messages, 37, 62
lmax field, 37
locminthru field, 9
locpacket field, 14
locthroughput field, 9
locwsize field, 14
Logical Channel Identifier (LCI), 6, 39, 68
lowprot fields, 10
lowprty field, 10
lsap_add field, 7
lsap_len field, 7
lsapformat structure, 7

M

M_DATA portion, 31, 32
maximum acceptable transit delay, 9
M-bit data and acknowledgments, 51
message structure, 1
minimum thoughput class, 9
More field, 31

N

N_Abort. *See* Abort Indication messages.
N_CC. *See* Connect Confirmation messages, Connect Response.
N_CI. *See* Connect Indication, Connect Request.
N_DAck. *See* Data Acknowledgment message.
N_Data. *See* Data message, 31
N_DC. *See* Disconnect Confirmation.
N_DI. *See* Disconnect Indication, Disconnect Request.
N_EAck. *See* Expedited Data Acknowledgment.
N_EData. *See* Expedited Data.
N_getnliversion **ioctl** command, 44
N_PVC_ATTACH. *See* PVC Attach messages.
N_PVC_DETACH. *See* PVC Detach messages.
N_RI. *See* Reset Indication, Reset Request.
N_Xcanlis. *See* Listen Cancel Request message, Listen Cancel Response messages.

N_XListen. *See* Listen Request, Listen Response message.
N-CONNECT requests, 10
negotiate_qos field, 29, 30, 48, 51, 66
Network User Identification facility, 12, 15
non-CONS calls
  cause code, 33
  negotiation, 11
  opening a connection, 50
non-OSI facilities, 11
nsap_len field, 7
NSAP field, 7
nsdulimit field, 14, 39
nui_field field, 15
nui_len field, 15

O

opening a connection, 46
originator field, 33, 35
OSI codes, 79

P

packet size negotiation, 11, 13, 67
PLP driver
  and STREAMS, 1
  higher priority messages, 58
  mapping to X.25 Virtual Circuits, 43
  negotiable X.25, 7
  NLI library, 46
  opening a stream, 43
  operating a PVC, 68
priority, 10, 24
privileged users and operations, 24, 61
programmable facilities, 12, 16
protection fields, 10
prty_ fields, 10
putmsg system call, 1, 24, 48, 51, 52, 58, 68
PVC Attach messages, 38, 68, 76
pvcattf structure, 38
PVC Detach messages, 40, 71, 76
pvcdetf structure, 40
pwoptions field, 13

Q

Q-bit data and acknowledgments, 31, 51, 53
qosformat structure, 8
QOS. *See* quality of service.
qos structure, 29, 30, 35
quality of service
  indicated, 35, 36
  negotiating, 48, 51, 66
  negotiation, 29, 30
  parameters, 8, 12
  qosformat structure, 8

R

reason_code field, 40
reason field, 33, 35
receipt acknowledgment service, 11, 32, 39
receiving data, 53
Recognized Private Operating Agency, 12, 15
remminthru field, 9
rempacket field, 14
remthroughput field, 9
remwsize field, 14
reqackservice field, 11, 32, 39, 68
reqcharging field, 16

reqexpedited field, 10
reqlowprty fields, 10
reqmaxtransitdelay field, 9
reqminthruput field, 9
reqnsdulimit field, 39, 68
reqpriority field, 10
reqprtygain field, 10
reqprtykeep field, 10
reqtclass field, 9
reqtransitdelay field, 9
Reset Confirmation messages, 34, 56, 76
Reset Indication
  error codes, 79
  example, 56
  messages, 33, 76
  PVC detach, 71
Reset Request
  error codes, 79, 80
  example, 70
  messages, 33, 75
  resets, 56
Reset Response messages, 34, 71, 75
responder field, 30, 35
restrictresponse field, 13
result_code field, 39, 70
reversecharges field, 13
reverse charging facility, 11, 13
rpoa_ fields, 15
RPOA. *See* Recognized Private Operating Agency
rqos field, 36

S

sending data, 52
setDbit field, 31
setQbit field, 31
sn_id field, 6, 39
Spider Systems, Ltd., 1
STREAMS messages, 1-2, 58
subnetwork identification, 6
system header files, 2

T

target transit delay, 9
throughput class, 9
transitdelay field, 9

U

user data, 29, 30, 31, 32, 35, 48

W

window size negotiation, 12, 13

X

x_fac_len field, 16
X.25
  addresses, 5
  D-bit data, 11
  driver, 1
  facilities. *See* facilities
  interrupt data, 11
  Packet Layer Protocol (PLP), 1
X25_primitives union, 28
X.29, 7, 13, 22, 51, 63
xabortf structure, 36
xaddrf structure, 6
xcallf structure, 29

xcanlisf structure, 38
xccnff structure, 30
xdatacf structure, 31
xdataf structure, 31
xdcnff structure, 36
xdiscf structure, 34
xedatacf structure, 33
xedataf structure, 32
xhdrf typedef, 28
xl_command field, 28
XL_CTL value, 28
XL_DAT value, 28
xl_type field, 28
xlistenf structure, 37
xrscf structure, 34
xrstf structure, 33

We'd Like to Hear From You

As a user of Silicon Graphics documentation, your comments are important to us. They help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics to comment on:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please include the title and part number of the document you are commenting on. The part number for this document is 007-2268-002.

Thank you!

Three Ways to Reach Us

The **postcard** opposite this page has space for your comments. Write your comments on the postage-paid card for your country, then detach and mail it. If your country is not listed, either use the international card and apply the necessary postage or use electronic mail or FAX for your reply.

If **electronic mail** is available to you, write your comments in an e-mail message and mail it to either of these addresses:

- If you are on the Internet, use this address: techpubs@sgi.com
- For UUCP mail, use this address through any backbone site: *[your_site]*!sgi!techpubs

You can forward your comments (or annotated copies of manual pages) to Technical Publications at this **FAX** number:

415 965-0964