Solstice X.25 9.2 Developer's Guide

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part No: 806-1235–10
October 1999

Contents

Tables

Figures

Preface

This guide describes the programming interfaces provided as part of the Solstice™ X.25 9.2 product. It does not cover the installation or configuration of the product. For this, refer to the installation instructions and *Solstice X.25 9.2 Administration Guide.*

How This Book Is Organized

This book contains the following parts and chapters:

Part I, Network Layer Interface, covers the network layer interface.

Chapter 1, provides a brief overview of STREAMS programming.

Chapter 2, provides some background information on the NLI programming interface.

Chapter 3, contains example programs for making and receiving calls.

Chapter 4, contains example programs for listening for incoming calls.

Chapter 5, contains an example program for collecting statistics.

Chapter 6, provides reference material on NLI commands and structures.

Chapter 7, provides reference material on network layer ioctls.

Chapter 8, provides reference material on the available library routines.

Chapter 9, provides information on NLI error codes.

Part II, Data Link Protocol Interface (DLPI), covers the DLPI programming interface.

Chapter 10, provides background information on DLPI.

Chapter 11, provides reference information on DLPI.

Part III, Sockets Interface, covers the Sockets programming interface.

Chapter 12, provides reference material on sockets programming.

Chapter 13, contains a sockets programming example.

What Typographic Changes Mean

The following table describes the typographic changes used in this book

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-programlisting computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`
`AaBbCc123`	What you type, contrasted with on-programlisting computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename.*
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

PART I Network Layer Interface (NLI)

CHAPTER 1

STREAMS Overview

STREAMS defines a standard interface for character I/O within the kernel, and between the kernel and the rest of the system. STREAMS creates, uses and dismantles *streams*. A stream is a full-duplex processing and data transfer path between a driver in kernel space and a process in user space.

1.1 Overview

A stream is made up of three parts—a stream head, optionally one or more modules, and a driver. The stream structure is summarized in Figure 1–1. The stream head provides the interface between the stream and the user processes. The module processes data travelling between the stream head and the driver. The driver can be a device driver, which has associated hardware, or a software driver.

STREAMS facilities are available using a series of system calls, which interact with the driver.

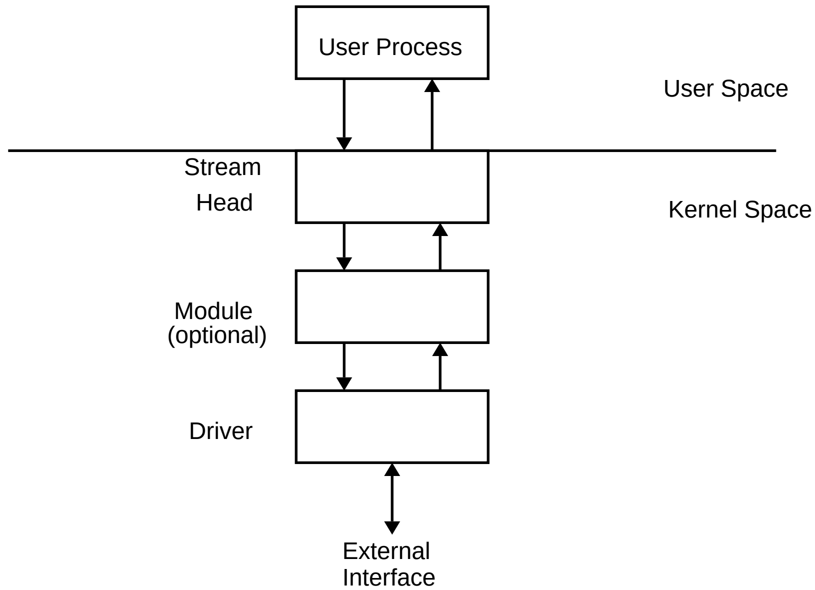


Figure 1–1 STREAMS Format

For detailed information on STREAMS, refer to the *STREAMS Programming Guide*.

CHAPTER **2**

About NLI

The Solstice X.25 Network Layer Interface (NLI) provides acces to the X.25 Packet Layer Protocol (PLP). The NLI defines the format that STREAMS messages must take when interfacing to the network layer. This allows for the easy construction of user level library software, and means that applications map conveniently onto the STREAMS format.

2.1 NLI Overview

The Solstice X.25 Network Layer Interface provides access to the X.25 Packet Layer Protocol (PLP). The NLI defines the format that STREAMS messages must take when interfacing to the network layer. This allows for the easy construction of user level library software, and means that applications map conveniently onto the STREAMS format.

Solstice X.25 applications use the `putmsg` and `getmsg` system calls to interact with the PLP driver.

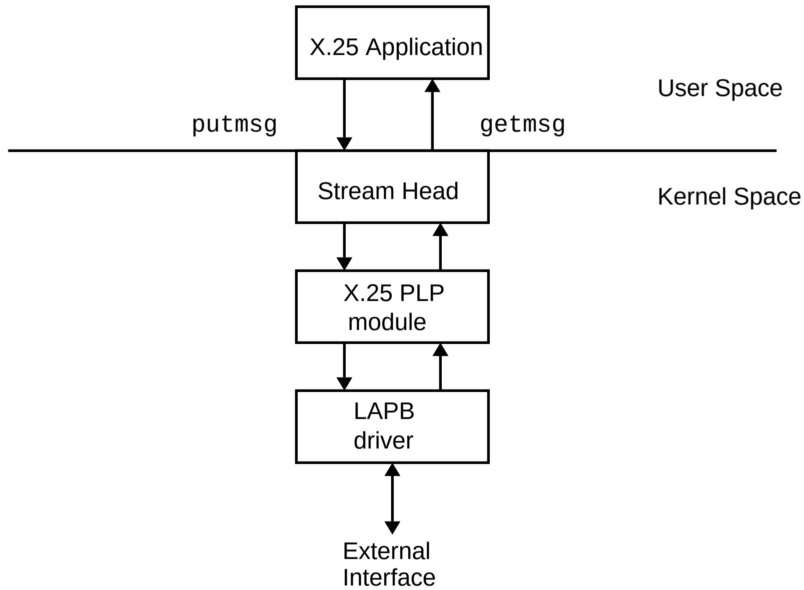


Figure 2–1 NLI and STREAMS

Messages passed using NLI have both a control and a data part. Primitives and associated parameters are passed to the X.25 driver using the control part of the message. Data, if there is any, is contained in the data part of the message.

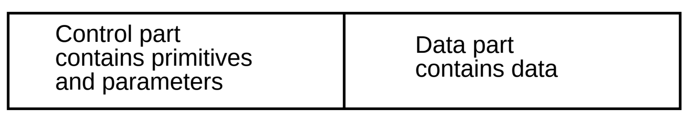


Figure 2–2 NLI Message Format

Solstice X.25 NLI provides the following:

- NLI messages

  These determine the format of the control parts of `putmsg` and `getmsg`, and are used to communicate with the network.

- A series of ioctls

  These communicate with the Solstice X.25 code, rather than with the network.

- A series of library functions

  These are not part of the NLI, but can be used along with it.

2.2 NLI Commands

Solstice X.25 NLI provides a series of NLI commands contained within C structures. These determine the format of the control parts of putmsg and getmsg. The NLI commands correspond to X.25 packet types, and are used to communicate with the network. For example, when an application passes down the NLI command N_CI to a stream using putmsg, this is translated into an X.25 Call Connect by the PLP module. When the PLP module receives a Connect Indication, it translates it into an N_CI command which is passed up to the application using a getmsg system call.

Table 2–1 summarizes the structures and their corresponding Packet Types and NLI Commands. Refer to the sections indicated for detailed information.

TABLE 2–1 NLI Commands and Structures

NLI Command	X.25 Packet	NLI Structure	See section
N_Abort	Abort Indication	xabortf	Section 6.4.1 "N_Abort—Abort Indication" on page 48
N_CC	Call Response/ Confirmation	xccnff	Section 6.4.2 "N_CC—Call Response/Confirmation" on page 48
N_CI	Call Request/ Indication	xcallf	Section 6.4.3 "N_CI—Call Request/Indication" on page 49
N_DAck	Data Acknowledgment Request/Indication	xdatacf	Section 6.4.4 "N_DAck—Data Ack Request/Indication" on page 51
N_Data	Data	xdataf	Section 6.4.5 "N_Data—Data" on page 52

TABLE 2–1 NLI Commands and Structures *(continued)*

NLI Command	X.25 Packet	NLI Structure	See section
N_DC	Clear Confirm	xdcnff	Section 6.4.6 "N_DC—Clear Confirm" on page 53
N_DI	Clear Request/Indication	xdiscf	Section 6.4.7 "N_DI—Clear Request/ Indication " on page 54Section 6.4.13 "N_RI—Reset Request/ Indication" on page 61
N_EAck	Expedited Data Acknowledgment	xedatacf	Section 6.4.8 "N_EAck—Expedited Data Acknowledgement" on page 56
N_EData	Expedited Data	xedataf	Section 6.4.9 "N_EData—Expedited Data" on page 57
N_RC	Reset Response/Confirm	xrscf	Section 6.4.12 "N_RC—Reset Response/ Confirm" on page 60
N_RI	Reset Request/Indication	xrstf	Section 6.4.13 "N_RI—Reset Request/ Indication" on page 61

The following commands and structures are also provided. They do not correspond to X.25 packet types:

TABLE 2–2 PVC and Listening Commands and Structures

NLI Command	NLI Structure	Description	See section
N_PVC_ATTACH	pvcattf	Specify X.25 service to use with PVC	Section 6.4.10 "N_PVC_ATTACH—PVC Attach " on page 58
N_PVC_DETACH	pvcdetf	Specify X.25 service to stop using with PVC	Section 6.4.11 "N_PVC_DETACH—PVC Detach " on page 59
N_Xcanlis	xcanlisf	Cancel listening	Section 6.4.14 "N_Xcanlis—Listen Cancel Command / Response" on page 62
N_Xlisten	xlistenf	Listen for incoming calls	Section 6.4.15 "N_Xlisten—Listen Command / Response" on page 63

2.3 NLI ioctls

Solstice X.25 NLI provides a series of ioctls. These are used to communicate with the Solstice X.25 code itself, rather than with the network. For example, ioctls are used to gather statistics about a link, or to set the parameters to be used on a link. This distinguishes them from the NLI Commands given in Table 2–1. To use them together, you might use an ioctl to set the parameters to be used on a link and then use an NLI command to initiate a call over the link. The NLI ioctls can be used for the following purposes:

- Operating on the Network User Identifiers mapping table
- Reading and resetting statistics
- Getting status information
- Operating on the X.25 Routing Table
- Overriding settings made using `x25tool`

2.4 Support Functions

Solstice X.25 also includes a library of Sun-specific support functions that you can use when writing applications. These are not part of the NLI, but can be used in NLI applications.

2.5 Support for OSI Connection-Mode Network Service (OSI CONS)

The Solstice X.25 NLI can support applications which use the OSI Connection-Mode Network Service, as defined in X.223 and ISO 8878. To be consistent with these documents, this service is referred to as OSI CONS in this guide. This service provides the mapping between the OSI CONS primitives and the elements of the X.25 Packet Layer Protocol.

2.6 Addressing

When making straightforward X.25 calls, you need to work with DTE and LSAP addresses. When making OSI CONS calls, you use NSAP and LSAP addresses. See Section 6.3.1 “`xaddrf`—Define Addressing” on page 37 for more information.

2.7 Facilities and QOS Parameters

The X.25 Recommendations allow service providers to offer a number of optional facilities, that affect the way that calls are made and handled.

- 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 programmable facilities
- X.25 call deflection.

The following Quality of Service (QOS) parameters are available when writing OSI CONS applications:

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

2.8 Operating System Support

Solstice X.25 9.2 will run on Solaris 7 or Solaris 8. It is not compatible with earlier versions of the Solaris operating system.

CHAPTER **3**

Making and Receiving Calls

This chapter contains examples of how to make and receive calls. All of the examples involve the application opening 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.

Sample code for making OSI CONS calls, dealing with Expedited Data and Resets and receiving a remote disconnect is given at the end of this section.

Note - There are copies of the code samples referred to in this chapter in the `/opt/SUNWconn/x25/samples.nli` directory.

3.1 Making a Single Call

This section shows the process for making a single, straightforward call. The call being made is a standard X.25 call. It does not have to deal with Expedited Data or Resets. The disconnect is initiated locally. The steps for making a standard X.25 call are:

1. **Open a stream on the `/dev/x25` device:**

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

2. **Open a connection to the open stream.**

1. Allocate a Connect Request structure.
2. Supply any quality of service and facilities parameters that are required.
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

```
#define FALSE 0
    #define TRUE 1
    #define CUDFLEN 4
    #define DBUFSIZ 128
    #include <memory.h>
    #include <netx25/x25_proto.h>
    struct xaddrf called = { 0, 0, { 14, { 0x23, 0x42, 0x31,
    0x56, 0x56, 0x56, 0x56 }}, 0 };
     /* no flags
      * DTE = "23423156565656", null NSAP
      */
    struct xcallf conreq;
    struct strbuf ctlblk, datblk;
    struct xdataf data;

    main ()
    {
       .
       /* Convert link to internal format */
       called.link_id = 0;
       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));
     }
```

In the example, the entire QOS field is zeroed, allowing for future additions to the structure. Setting the calling address to null, as shown, leaves the network to fill in this value. For more information on QOS and Facilities, see Section 2.7 “Facilities and QOS Parameters” on page 10.

3. **Send the message on the stream using the** `putmsg` **system call, passing any call user data in the data part of the message:**

```
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);
     }
```

4. **Transfer the data.**

   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 OSI CONS.

     Normal and Q-bit data is sent and received using the N_Data message and may be acknowledged using the N_DAck message. Expedited data uses the N_EData message, and is acknowledged using an N_EAck message.

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

```
char datbuf[DBUFSIZ];
    /* 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 blocks 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. To avoid blocking due to a full queue, open the stream 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.

5. **Close the connection.**

   In this example, closure is initiated locally. The application sends a Disconnect Request (N_DI) message on the stream. Unless this is being used to reject an incoming call the X.25 driver signals that it has observed the message. It does this by sending a Disconnect Confirm upstream when it receives the Clear Confirm. In this way, the upper components can be certain that no messages will follow the Disconnect.

   In the case of rejection, the connection identifier supplied on the Connect Indication must be returned in the disconnect message. The disconnect (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 Response.

As described in Section 6.4.7 "N_DI—Clear Request/Indication " on page 54, a disconnect collision may occur. If this happens, no Clear Confirm 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 */
       }
      }  /* end if (xqos) */
     }  /* end if (hdrptr->xl_type==XL_CTL) */
```

3.2 Receiving Data

In the same way as sending data, data reception is straightforward. When data is received with the D-bit set, action may be required by the application. When the initial Call 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, an application must send a Data Acknowledgment (N_DAck) 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));
      memset((char *)dack 0, 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);
      }
     } /* end else */
    break;
   case N_EData:
    printf(‘‘***Expedited data received \n’’);
 /* Must deal with it */
    break;
   case N_DAck:
    printf(‘‘***Data Acknowledgment received \n’’);
    break;
   default:
    break;
   } /* end switch */
  } /* end if */
 } /* end for */
```

3.3 Additional Call Information

The example in Section 3.1 “Making a Single Call” on page 13, is of a relatively straightforward call. Procedures for making a call using OSI CONS, for receiving expedited data, for dealing with resets and for receiving remotely initiated disconnects are given in the following sections. These can be integrated into the example above, as required.

3.3.1 Opening connections for OSI CONS Calls

The following example opens a connection for an OSI CONS call:

```
#define FALSE 0
 #define TRUE 1
 #define CUDFLEN 4
 #define EXPLEN 4
 #include <memory.h>
 #include <netx25/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;
 struct strbuf, ctlblk, datblk;
 struct xedataf exp;

 main ()
 {
    .
    .
    .
    called.link_id = 0;
    /*
     * 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 (non-zero), setting the `QOS` 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 *entire* 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 call user data being passed in the data part of the message:

```
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 (rejection) message.

3.3.2 Receiving Expedited Data

The preceding 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. The receiver does this by sending an Expedited Acknowledgment (`EAck`) message. The `EAck` is sent in the same way as an ordinary data packet, but with no data part.

If an application does not need to use the expedited data capability, then it responds to receiving an `EData` by resetting or closing the connection.

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

```
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;
  }
 }
```

3.3.3 Dealing with Resets and Interrupts

Resets and Interrupts are dealt with in a similar way, except that there is no data passed with a Reset Request. When a Reset Request or Interrupt is issued, the application must wait for the acknowledgment, as for an expedited 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 or Interrupt contains the reason for issuing the reset. Standard values for this are defined in the include file `<netx25/x25_proto.h>`, although the application can set any value. See Chapter 9 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 either case, pending data is flushed from the queue.

Reset Indications can be dealt with as part of the general processing of incoming messages, as shown in the following disconnect handling example.

```
#include<netx25/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;
    }  /* end switch */
   } /* end for */
```

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

3.3.3.1 Receiving a Remote Disconnect

If the remote end initiates a Disconnect, then a Disconnect Indication (`N_DI`) message (or possibly an `N_Abort` message, see Section 6.4.1 “`N_Abort`—Abort Indication” on page 48) is received at the NLI. The application need not acknowledge this message since, after sending a Disconnect, 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 following example could be combined with the code from the data transfer example in the previous section.

```
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 Aborted \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.

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.

CHAPTER **4**

Listening for Calls

This chapter contains examples of how to listen for single or multiple incoming calls and then accept or reject the call.

Note - There are copies of the code samples referred to in this chapter in the `/opt/SUNWconn/x25/samples.nli` directory.

4.1 Listening for a Single Call

The steps for listening for a single incoming call are:

1. **Send an `N_Xlisten` message to the X.25 driver.**

   This should carry the called address list in which the application is interested.

2. **Wait for the response to the Listen Request.**

   The `l_result` flag will indicate success or failure. If the `l_result` flag indicates failure, the application can decide either to close the stream or to try again later.

3. **Wait for Connect Indication messages from the X.25 driver.**

4. **Decide whether to accept on this or a different stream.**

5. **Negotiate facilities and QOS, if required.**

   A Connect Confirmation message carrying the appropriate connection identifier is then passed down on the stream on which the connection is being accepted.

6. **Construct the listen message.**

The listen message has two parts. Construct the control part of the message like this:

```
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 contains the sequence of bytes that specify the Call User Data 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[]` (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.

---

7. **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;
     }
```

8. **Wait for the listen response; the result flag indicates success or failure:**

```
#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 is done in the same way, except that no data is passed with the request. It cancels all successful Listens that have been made on that stream.

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

When an N_CI message arrives, the application should inspect its parameters—address, call user data, facilities, quality of service, and so on—then decide whether to accept or reject the connection.

A listening application can accept a call either on the stream the indication arrived on, or on some other stream. This other stream can be one which is already open and free, or the application can open it.

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 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.

A listening application can reject the call by sending a N_DI message down the stream on which the Connect Indication arrived. A Connect Indication cannot be rejected on a different stream. 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. The application sets the reason field to NU_PERMANENT (0xF5).
- Search for further matching listeners. The application 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, 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 can be sent subsequently.

---

10. Negotiate any facilities or OSI CONS QOS parameters required.

To do this, set 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 Section 2.7 “Facilities and QOS Parameters” on page 10). Copy the entire QOS structure from the indication to the response. This allows for 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;
     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.

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.

4.2 Listening for Multiple Incoming Calls

Sample code for a listener that can handle several incoming PAD calls simultaneously is provided in the file `/opt/SUNWconn/x25/samples.nli/listen.c`. Listeners to handle other types of incoming calls are similar. The steps are:

1. **Define the values you need.**

   Specify the maximum number of simultaneous calls you want to allow. Set the maximum number of simultaneous calls depending on the processor power available to you and the number of calls you expect to need to handle.

2. **Open the X.25 device.**

   The open_stream() function does this. It requests notification from the kernel when there is incoming data on the stream.

3. **Listen for incoming data.**

   The do_listen() function specifies the information used to decide what to do with an incoming call. The example shows two ways of doing this, one simple, the other more complex. In the simple example, the program listens for any call user data beginning with the four BCD digits 1234.

4. **Wait for incoming calls.**

   To do this, call the getmsg() function. If an incoming call arrives that matches the criteria that you specified in step 1, the X25 driver will send an N_CI indication. At this point, you could choose to do some more sophisticated checking. The example program includes a function called try_next() that tells the X.25 driver to see if the connect message is destined for another application, and a function called reject_call() that tells the X.25 driver to reject the call.

5. **Accept the incoming call.**

   Assuming the call is valid, the accept_call() function is used to accept it. Note that when accepting incoming data, the application *must* copy the call indication identifier into the connect confirm sent to the kernel.

6. **Handle the incoming call as appropriate.**

   The sample code contains an example of a call that is handled by printing a message and closing the device (which closes the connection).

7. **Exit the program when finished.**

CHAPTER **5**

Getting Statistics

This chapter contains an example of a program for gathering statistics. By using the ioctls described in Chapter 7, you can write programs that specify more precisely what kind of statistics you want to gather and whether they should apply to a particular link or virtual circuit.

5.1 Sample Program

Note - There is a copy of this code sample in the `/opt/SUNWconn/x25/samples.stats` directory.

The steps for writing this kind of program are:

1. **Include the streams and X.25 header files.**

2. **Specify the location of the X.25 devices file descriptor.**

3. **Define a structure for containing the statistics.**
   In the example these are per-link X.25 statistics.

4. **Open the X.25 driver.**

5. **Define the fields required by the ioctl(s) you are using.**
   In the example, this is `N_getlinkstats`, which retrieves per link X.25 statistics.

6. **Specify where you want to gather statistics from.**
   For example, `N_getlinkstats` requires you to give a link number.

7. **Gather the statistics.**

   How you do this depends on which ioctls you are using.

8. **Display, or otherwise make use of, the statistics that have been gathered.**

```
/************************************************************
    * Copyright 20 Apr 1995 Sun Microsystems, Inc. All Rights Reserved
    *
    * This example shows how to open X25 driver,
    * get X25 per_link statistics,
    * and display (as an example) the number of transmit and
    * received CALL packets.
    *
    * Many more information are available for X25 through this ioctl.
    * Many other ioctls permit to get :
    * - Global stats for : IXE, X25 packet layer protocol, LAPB, LLC2, and MLP
    * - Per-link stats for : X25, LAPB, LLC2 and MLP
    * - Per-VC stats
    * Some other ioctls permit to reset those statistics.
    *
    ************************************************************/

   /*
    * General includes for streams
    */
   #include <fcntl.h>
   #include <stdio.h>
   #include <sys/types.h>
   #include <sys/stat.h>
   #include <sys/stropts.h>
   #include <sys/stream.h>

   /*
    * the following includes are specific to X25
    */
   #include <netx25/uint.h>
   #include <netx25/x25_proto.h>
   #include <netx25/x25_control.h>

   /*
    * location of x25 device file descriptor
    */
   #define X25_DEV ‘‘/dev/x25’’

   /* used to open X25 device */
   int x25_fd;

   /*
    * io control structure used for all stream ioctl
    * see STREAM programmer’s guide for more information about this structure.
    */
   struct strioctl      ioc;

   /*
    * this structure to be used to collect X25 per-link stats
    */
   struct perlinkstats  x25_s;
```

```
main()
{
 /*
  * open x25 driver
  */
 x25_fd = open(X25_DEV, O_RDONLY);
 if (x25_fd == -1)
        {
   perror(‘‘Failed to access X25 driver.\n’’);
   exit(1);
 }

 /*
  * set the general info for ioctl
  */
 ioc.ic_cmd = N_getlinkstats;
 ioc.ic_len = sizeof(x25_s);
 ioc.ic_dp = (char *) &x25_s;

 /*
  * Set the link id.
  * Specify the link number where to gather statistics.
  * (2 in that particular case)
  */
 x25_s.linkid = 2;

 /*
  * perform the STREAMS ioctl
  */
 if (ioctl(x25_fd, I_STR, &ioc) < 0)
   {
   perror(‘‘Failed to gather X25 per-link statistics’’);
   exit(2);
 }

 /*
  * display some statistics
  */
 printf(‘‘X25 statistics for link number 2\n’’);
 printf(‘‘Number of CALL transmitted %10ld\n’’,
   x25_s.mon_array[cll_out_s]);
 printf(‘‘Number of CALL received %10ld   \n’’,
   x25_s.mon_array[cll_in_s]);
}
```

CHAPTER 6

NLI Commands and Structures

Solstice X.25 NLI provides a set of NLI commands. These are contained within C structures, which determine the format of the control parts of putmsg and getmsg. The NLI commands correspond to X.25 packet types. NLI commands are used to communicate with the network. For example, when an application passes down the NLI command N_CI to a stream using putmsg, this is translated into an X.25 Call Request by the PLP module. When the PLP module receives a Connect Indication, it translates it into an N_CI message which is passed up to the application using a getmsg system call.

The header files used by the NLI commands and structures are contained in the /usr/include/netx25 directory.

6.1 Commands and Structures Tables

Table 6–1 summarizes the NLI commands and their corresponding Packet Types and C structures:

TABLE 6–1 NLI Commands and Structures

NLI Command	X.25 Packet	NLI Structure
N_Abort	Abort Indication	xabortf
N_CC	Call Response/ Confirmation	xccnff
N_CI	Call Request/ Indication	xcallf

TABLE 6–1 NLI Commands and Structures *(continued)*

NLI Command	X.25 Packet	NLI Structure
N_DAck	Data Acknowledgment Request/Indication	xdatacf
N_Data	Data	xdataf
N_DC	Clear Confirm	xdcnff
N_DI	Clear Request/Indication	xdiscf
N_EAck	Expedited Data Acknowledgement	xedatacf
N_EData	Expedited Data	xedataf
N_RC	Reset Response/Confirm	xrscf
N_RI	Reset Request/Indication	xrstf

These commands and structures in do not correspond to X.25 packet types:

TABLE 6–2 PVC and Listening Commands and Structures

NLI Command	NLI Structure	Description
N_PVC_ATTACH	pvcattf	Specify X.25 service to use with PVC
N_PVC_DETACH	pvcdetf	Specify X.25 service to stop using with PVC
N_Xcanlis	xcanlisf	Cancel listening
N_Xlisten	xlistenf	Listen for incoming calls

All of the structures listed in Table 6–1 and Table 6–2 are defined in the `x25_primitives` C union.

In addition to the structures that have a one-to-one mapping with NLI commands, Solstice X.25 provides a number of structures that are used by several of the commands. These are related to addressing and facilities, and are listed in Table 6–3:

TABLE 6–3 Generic Structures

structure	function
xaddrf	contains addressing information
lsapformat	defines the LSAP
extraformat	defines optional X.25 facilities
qosformat	defines OSI CONS Quality of Service (QOS) parameters

6.2 x25_primitives C Union

The Solstice X.25 software provides a series of data structures that determine the control part of messages passed across the NLI. The format of the control part of messages passed across the NLI is 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 */
  };
```

All structures begin with the same members, as shown below:

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

Messages to and from applications are classified as control messages or data messages. `xl_type` indicates whether a message is control or data using the values `XL_CTL` for control and `XL_DAT` for data. Within each classification, the message identity is indicated by the `xl_command` qualifier. The combination of `xl_type` and `xl_command` must be consistent.

When sending an NLI command to the `x25` driver using `putmsg`, the size of the data structure is determined by the command, and clearly is known in advance. The `.len` member of the control buffer is used to hold this value, and the `.maxlen` member is not used.

When reading a message with the `getmsg` call, the type of message cannot be known before it is received, so a buffer large enough to hold any message should be supplied. Put the size of this buffer in the `.maxlen` member of the control buffer structure. The actual size of the message received will be placed in the `.len` member on return from the `getmsg` call. To ensure that the buffer will be large enough, declare it as being of type `union X25_primitives`.

Code Example 6–1 shows how a `getmsg` can be constructed.

CODE EXAMPLE 6–1 Constructing a `getmsg`

```
#include <stream.h>
 #include <netx25/dx25_proto.h>


 struct strbuf ctlb;
 struct strbuf datab;

 union X25_primitives buffer;
 char data_buf[DATALEN];

   .
   .
   .

 ctlb.maxlen = sizeof (union X25_primitives);
 ctlb.buf = (char *)buffer;

 flag = MSG_ANY;
 datab.maxlen = DATALEN;
 datab.buf = data_buf;

 getmsg (x25_fd, &ctlb, &datab, flag);

 switch ((S_X25_HDR *)&buffer->xl_type) {
  case N_Abort:
    /* treat 'buffer' as an Abort message
     * datab.len should be 0
     */
   break;

  case N_CI:
    /* Treat 'buffer' as a Connect Indication
     * data_buf[] contains Call User Data
     * datab.len equals length of Call User Data
     */
    break;

   .
```

```
.
.
};
```

6.3 Generic Structures

The structures described in this section define addressing, facilities and QOS and are used by a number of the commands described in Section 6.4 "NLI Commands" on page 47.

6.3.1 xaddrf—Define Addressing

Addressing is defined by the xaddrf structure:

CODE EXAMPLE 6–2 xaddrf Structure

```
#define NSAPMAXSIZE 20

 struct xaddrf {
        uint32_t           link_id;
        unsigned char           aflags;
        struct lsapformat       DTE_MAC;
        unsigned char           nsap_len;
        unsigned char           NSAP[NSAPMAXSIZE];
}
```

The members in the xaddrf structure are:

TABLE 6–4 Members of xaddrf Structure

Member	Description
link_id	Holds the link number as an uint32_t. By default, link_id has a value of 0xFF. When link_id is 0xFF, Solstice X.25 attempts to match the called address with an entry in a routing configuration file. If it cannot find a match, it routes the call over the lowest numbered WAN link.
aflags	Specifies the options required or used by the subnetwork to encode and interpret addresses. Takes one of these values: NSAP_ADDR 0x00 NSAP field contains OSI-encoded NSAP address EXT_ADDR 0x01 NSAP field contains non-OSI-encoded extended address PVC_LCI 0x02 NSAP field contains a PVC number.
DTE_MAC	The DTE address, or LSAP as two BCD digits per byte, right justified, or the PVC_LCI as three BCD digits with two digits per byte, right justified.
nsap_len	The length in semi-octets of the NSAP as two BCD digits per byte, right justified.
NSAP	The NSAP or address extension (see aflags) as two BCD digits per byte, right justified.

6.3.2 lsapformat—Define an LSAP

The LSAP is defined by the lsapformat structure:

CODE EXAMPLE 6–3 lsapformat Structure

```
#define LSAPMAXSIZE 9

 struct lsapformat {
    uint8      lsap_len;
    uint8      lsap_add[LSAPMAXSIZE];
 };
```

The members of the lsapformat structure are:

TABLE 6–5 Members of `lsapformat` Structure

Member	Description
`lsap_len`	The length of the DTE address or LSAP as two BCD digits per byte, right justified. An LSAP is always 14 digits long. A DTE address can be up to 15 decimal digits unless X.25 (88) and TOA / NPI addressing is used, in which case it can be up to 17 decimal digits. A PVC_LCI is 3 digits long.
`lsap_add`	The DTE address, LSAP or PVC_LCI as two BCD digits per byte, right justified.

6.3.3 `extraformat`—Define Standard X.25 Facilities

Standard X.25 facilities are defined by the `extraformat` structure:

CODE EXAMPLE 6–4 `extraformat` Structure

```
#define MAX_NUI_LEN   64
 #define MAX_RPOA_LEN  8
 #define MAX_CUG_LEN   2
 #define MAX_FAC_LEN   109
 #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 {
 /* extraformat structure */
         unsigned char     fastselreq;
         unsigned char     restrictresponse;
         unsigned char     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 members of this structure are defined as follows:

TABLE 6–6 Members of `extraformat` Structure

Member	Description
`fastselreq`	Applies only to non-OSI applications, for example X.29. A non-zero value means the X.25 facility fast select is to be requested or indicated.
`restrictresponsee`	Sets response to be a Clear Request.
`reversecharges`	A non-zero value means that reverse charging is requested or indicated for this connection
`pwoptions`	Indicates per virtual-circuit options. The field is a bit map with the following interpretation: bit 0: 0 - Packet size negotiation NOT permitted. 1 `NEGOT_PKT` - Packet size negotiation permitted. bit 1: 0 - Window size negotiation NOT permitted. 1 `NEGOT_WIN` - Window size negotiation permitted. bit 2: 0 - No concatenation limit asserted. 1 `ASSERT_HWM` - 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. However, the field `pwoptions` for an incoming call indicates whether these values are negotiable. 2) In Call Requests and Call Responses the NLI user can set a limit value, `nsdulimit`, for packet concatenation by the X.25 level that differs from the limit in the subnetwork configuration database. It is not a negotiable option, so that whatever the user has requested is used.
`locpacket`	Contains the packet size for local-to-remote calls, using the following notation: the actual packet size is 2 to the power of the value specified. For example if the field `locpacket` is set to 7, the actual packet size is 2^7 or 128.

TABLE 6–6 Members of `extraformat` Structure *(continued)*

Member	Description
`rempacket`	Contains the packet size for remote-to-local calls, using the following notation: the actual packet size is 2 to the power of the value specified. For example if the field `rempacket` is set to 7, the actual packet size is 2^7 or 128.
`locwsize`	The window sizes for local-to-remote calls.
`remwsize`	The window sizes for remote-to-local calls.
`nsdulimit`	Specifies the packet concatenation limit.
`nui_len`	The length of any Network User Identification used in Call Requests and Responses.
`nui_field`	Network User Identification used in Call Requests and Responses. This is not available on X.25 (80) networks.
`rpoa_len`	The length of any RPOA DNIC information used in Call Requests. Valid values for `rpoa_len` are 0, 4, 8, 12 and 16.
`rpoa_field`	Any Recognized Private Operating Agency (RPOA) DNIC information. This is used in Call Requests only. It is stored as two BCD digits per byte, right justified. On X.25 (80) networks, this is restricted to one RPOA of 4 BCD digits. Basic format encoding is used. On X.25 (84) and X.25 (88) networks, there can be one or more RPOAs. Extended format encoding is used if there is more than one RPOA. The maximum number of RPOAs is 4.
`cug_type`	Possible values are: CUG — Closed User Group BCUG — Bilateral CUG 0—No CUG used
`cug_field`	Any applicable CUG information, stored as two BCD digits per byte, right justified. Note: Incoming Closed User Group facilities are assumed to have been validated by the network. No further checking is performed.
`reqcharging`	Requests call charging in a Call Request or Connect Accept.

TABLE 6–6 Members of `extraformat` Structure *(continued)*

Member	Description
chg_cd_len	Gives length of chg_cd_field.
chg_cd_field	Specifies duration of the call if call charging is in use. Used in a Disconnect Indication or Confirm.
chg_sc_len	Gives length of chg_sc_field.
chg_sc_field	Specifies segment count if call charging is in use. Used in a Disconnect Indication or Confirm.
chg_mu_len	Gives length of chg_mu_field.
chg_mu_field	Specifies monetary unit if call charging is in use. Used in a Disconnect Indication or Confirm.
called_add_mod	A one byte field holding the reason for any address modification as defined in the X.25 Recommendation, encoded as follows: X0000001—Called DTE busy. Call redirected. X0000111—Call distribution within hunt group. X0001001—Called DTE out of order. Call redirected. X0001111—Called DTE has requested systematic redirection. 11000000—Called DTE deflected call. 11000001—Called DTE busy. Gateway redirected call. 11001001—Called DTE out of order. Gateway redirected call. 11001111—Called DTE has requested systematic redirection. Gateway redirected call. X indicates that this bit is 0 if the address modification occurred in a public data network and 1 if it occurred in a private network.
call_redirect	A one byte field holding the reason for a call redirection as defined in the X.25 Recommendation, encoded as follows: 00000001—Called DTE busy. Call redirected. 00000111—Call distribution within hunt group. 00001001—Called DTE out of order. Call redirected. 00001111—Called DTE has requested systematic redirection. 11000000—Called DTE deflected call. 11000001—Called DTE busy. Gateway redirected call. 11001001—Called DTE out of order. Gateway redirected call. 11001111—Called DTE has requested systematic redirection. Gateway redirected call.

TABLE 6–6 Members of `extraformat` Structure *(continued)*

Member	Description
`called`	Supplies the originally-called DTE address.
`call_deflect`	A one byte field holding the reason for a call deflection as defined in the X.25 Recommendation, encoded as follows: `11000000`—Called DTE deflected call. `11000001`—Called DTE busy. Gateway redirected call. `11001001`—Called DTE out of order. Gateway redirected call. `11001111`—Called DTE has requested systematic redirection. Gateway redirected call.
`deflected`	In a Clear Request, contains the DTE address, and if required, the NSAP that a call is to be deflected to.
`x_fac_len`	Indicates the length of a `fac_field` relating to X.25 facilities.
`cg_fac_len`	Indicates the length of a `fac_field` relating to non-X.25 facilities for the calling network.
`cd_fac_len`	Indicates the length of a `fac_field` relating to non-X.25 facilities for the called network.
`fac_field`	This field is used in Call Requests and Connect Accepts only. It allows for the passing of explicit facility encoded strings for X.25 facilities, and non-X.25 facilities for calling and called networks. **Note -** The contents of this field, are not validated or acted upon by the code. The X.25 facilities are inserted at the end of any other X.25 facilities which are passed in the Call Request/Accept (for example, packet/window sizes). If any non-X.25 facilities are supplied the appropriate marker is inserted before the supplied facilities. Take care not to duplicate any facilities.

6.3.4 `qosformat`—Define OSI CONS QOS Parameters

OSI CONS-related quality-of-service parameters are defined in the `quosformat` 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 members of the `qosformat` structure are defined as follows:

TABLE 6–7 QOS Parameters

Member	**Description**
`reqtclass`	Indicates whether the throughput negotiation parameter is selected. 0 indicates that it is not selected.
`locthroughput`	Contains four-bit throughput encoding for local-to-remote calls.
`remthroughput`	Contains four-bit throughput encoding for remote-to-local calls.
`reqminthruput`	Indicates `whether` the minimum throughput negotiation parameter is selected.
`locminthru`	Contains four-bit throughput encoding for local to remote calls.
`remminthru`	Contains four-bit throughput encoding for remote to local calls.

TABLE 6–7 QOS Parameters *(continued)*

Member	Description
`reqtransitdelay`	Indicates whether the transit delay parameter is selected. 0 indicates that it is not selected.
`transitdelay`	Contains the transit delay parameter as a 16-bit value. It is used in Call Requests and Indications and Confirms.
`reqmaxtransitdelay`	Indicates whether the calling NLI application specifies a maximum acceptable value for the transit delay parameter (“Lowest Quality Acceptable”). Note: The transit delay selection relates only to Call Requests and there is no transit delay QOS parameter in a Call Response primitive. The correct response when the indicated QOS is unattainable is to make a Clear Request. Also, in a Connect Confirm, the value of the selected transit delay will be placed in the `transitdelay` field when such negotiation takes place.
`acceptable`	Contains the maximum acceptable transit delay parameter, if this is specified by the calling NLI application.
`reqpriority`	Requests or indicates priority on a connection. 0 indicates that priority is not used.
`prtydata`	Contains the 8-bit value for the priority of data on the connection.
`reqprtygain`	Indicates that the field `prtygain` is used.
`reqprtykeep`	Indicates that the field `prtykeep` is used.
`prtygain`	Contains an 8-bit value for the priority to gain a connection.
`prtykeep`	Contains the 8-bit value priority to keep a connection.
`reqlowprtydata`	Indicates the field `lowprtydata` is used.
`reqlowprtygain`	Indicates the field `lowprtygain` is used.
`reqlowprtykeep`	Indicates the field `lowprtykeep` is used.
`lowprtydata`	Contains the lowest acceptable priority value. Used on `N-CONNECT` requests by the calling `NS_user`.

TABLE 6–7 QOS Parameters *(continued)*

Member	Description
lowprtygain	Indicates the priority of data on a connection. Used on N-CONNECT requests by the calling NS_user.
lowprtykeep	Indicates priority for gaining a connection. Used on N-CONNECT requests by the calling NS_user.
protection_type	Indicates the type of protection required. Values are: PRT_SRC Source address specific PRT_DST Destination address specific PRT_GLB Globally unique 0 indicates that protection is not required. On N-CONNECT requests the calling NS_user may optionally specify a lowest acceptable level of protection.
prot_len	The length of the target protection.
protection	The value of target protection.
lowprot_len	The length of the lowest acceptable level of protection.
lowprotection	The lowest acceptable level of protection.

TABLE 6–7 QOS Parameters *(continued)*

Member	Description
reqexpedited	Indicates whether expedited data is required/selected. 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. 0 indicates that expedited data is not used. Note: Negotiation is an OSI CONS procedure. When the facility is present and indicates non-use, use cannot be negotiated by Connect responses. See Section 6.4.3 "N_CI—Call Request/Indication" on page 49and Section 6.4.2 "N_CC—Call Response/Confirmation" on page 48for a description of the use of the CONS_call field in Call Requests and Call Responses. If the CONS_call flag is 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 Call Requests and Responses.
reqackservice	Indicates whether the acknowledgement service is to be used. Allowed values are: 0 indicates the service is not used. 1 signifies acknowledgment confirmation by the remote DTE. In the case of acknowledgment confirmation by the remote application, 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 over the X.25 interface. 2 signifies acknowledgment confirmation by the remote application. In this case of acknowledgment confirmation by the remote DTE, no acknowledgments are expected or given over the X.25 interface. Any non-zero value causes negotiation in the call setup phase for use of the D-bit on the connection.

6.4 NLI Commands

This section describes the available NLI commands in alphabetical order. Refer to Table 6–1 for a summary of the available commands and related structures.

6.4.1 N_Abort—Abort Indication

Description

N_Abort is used when the X.25 driver needs to send a Disconnect 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. The control part of an Abort Indication message has a format defined in the xabortf structure. There is no data part.

Note - This message only appears in a getmsg, never in a putmsg. Code Example 6–1 shows how a getmsg can be constructed.

The xabortf structure is shown below:

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

6.4.2 N_CC—Call Response / Confirmation

Synopsis

```
#include <stream.h>
 #include <netx25/x25_proto.h >

 struct strbuf ctlb;

 struct xccnff confirm;
 .
 .
 .
 ctlb.len = sizeof(struct xccnff);
 ctlb.buf = (char *)confirm;

 putmsg(x25_fd, &ctlb, NULL,0);
```

Description

N_CC is used when calls are being accepted. When used with putmsg, N_CC is a Call Response, when used with getmsg, it is a Call Confirmation. Code Example 6–1 shows how a getmsg can be constructed. When used with getmsg, ctlb.len is replaced by cltb.maxlen. The control part of the Call Request or Indication is defined by the xccnff structure. There is no data part.

The xccnff structure is shown below:

```
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 */
};
```

The members of the `xccnff` structure are:

TABLE 6–8 Call Response / Confirmation Message

Member	Description
`conn_id`	Connection identifier. `conn_id` must be returned in the Call Response so that listening operates properly. This must be the same connection identifier as was included in the Connection Request or Indication.
`CONS_call`	Indicates that OSI CONS procedures should be used for responses. If you are not using OSI CONS, this value should be 0.
`negotiate_qos`	A non-zero value shows that facilities and quality of service (QOS) are being negotiated. A zero value means the initiator is requesting all default values.
`responder`	The responding address.
`rqos`	Selected facilities and OSI CONS QOS parameters to be passed to the initiator.

6.4.3 N_CI—Call Request / Indication

Synopsis

```
#include <stream.h>
#include <netx25/x25_proto.h >

struct strbuf ctlb;
struct strbuf datab;

struct xcallf call;
char          cud[MAX_LENGTH];
```

```
.
.
.
ctlb.len = sizeof(struct xcallf);
ctlb.buf = (char *)call;

datab.len = cudlen;
datab.buf = cud;

putmsg(x25_fd, &ctlb, &datab, 0);
```

Description

N_CI is used when calls are requested or indicated across the X.25 interface. When used with putmsg, N_CI is a Call Request, when used with getmsg, it is a Connect Indication. Code Example 6–1shows how a getmsg can be constructed. The control part of the Call Request or Indication is defined by the xcallf structure. The data part of the message will contain any call user data.

The xcallf structure is shown below:

```
struct xcallf {
  unsigned char xl_type; /* Always XL_CTL */
  unsigned char xl_command; /* Always N_CI */
  int conn_id; /*connection id returned in Call 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 address  */
  struct xaddrf callingaddr; /* The calling address */
  struct qosformat qos; /* Facilities and CONS qos: if
                              negotiate_qos is set */
 };
```

The members of the xcallf structure are:

TABLE 6–9 Call Request/Indication Message

Member	Description
conn_id	For incoming calls, an attempt is made to match the called address and call user data 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 Call Response or Clear Request to accept or reject the connection. Leave this value as 0.
CONS_call	Indicates that OSI CONS procedures should be used for the call.
negotiate_qos	A non-zero value shows that facilities and quality of service (QOS) are being negotiated. A zero value means the initiator is requesting all default values.

TABLE 6–9 Call Request/Indication Message *(continued)*

Member	Description
calledaddr	Holds the called address.
callingaddr	The calling address.
qos	Any facilities requested or indicated. To use the qos member, you must set negotiate_qos.

6.4.4 N_DAck—Data Ack Request/Indication

Synopsis

```
#include <stream.h>
 #include <netx25/x25_proto.h >

 struct strbuf ctlb;

 struct xdatacf dack;
 .
 .
 .
 ctlb.len = sizeof(struct xdatacf);
 ctlb.buf = (char *)dack;

 putmsg(x25_fd, &ctlb, NULL, 0);
```

Description

N_DAck acknowledges a previous Data Acknowledgment Request or Indication which had the D-bit set. The D-bit requests end-to-end, as opposed to local, acknowledgment. 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.

Refer to Section 6.3.4 "qosformat—Define OSI CONS QOS Parameters" on page 43 for details of requesting acknowledgment using the reqackservice member of the qosformat structure. For OSI CONS calls, Data Acknowledgment must be negotiated on the connection.

When used with putmsg, N_DAck is a Data Acknowledgment Request, when used with getmsg, it is a Data Acknowledgment Indication. Code Example 6–1 shows how a getmsg can be constructed. The control part of the Data Acknowledgment Request or Indication is defined by the xdatacf structure. There is no data part.

The xdatacf structure is shown below:

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

6.4.5 N_Data—Data

Synopsis

```
#include <stream.h>
 #include <netx25/x25_proto.h >

 struct strbuf ctlb;
 struct strbuf datab;

 struct xdataf control;
 char          data[MAX_LENGTH];
 .
 .
 .
 ctlb.len = sizeof(struct xdataf);
 ctlb.buf = (char *)control;

 datab.len = MAX_LENGTH;
 datab.buf = data;

 putmsg(x25_fd, &ctlb, &datab, 0);
```

Description

N_Data is used to transfer data across the X.25 interface. The synopsis shows a putmsg. Code Example 6–1 shows how a getmsg can be constructed. The control part of the Data packet is defined by the xdataf structure. The data part of the message contains the user data.

The xdataf structure is shown below:

```
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 */
  unsigned char setDbit, /* Set when data carries X.25 D-bit */
  unsigned char setQbit; /* Set when data carries X.25 Q-bit */
 };
```

The members used by xdataf are.

TABLE 6–10 Data Message

Member	Description
`More`	Shows whether there is more of this network service data unit to be received / sent.
`setQbit`	Requests or indicates that the Q-bit is set when user data is transmitted / received. The Q-bit indicates that the data is intended for a device attached to the DTE and not for the DTE itself.
`setDbit`	Requests or indicates that the D-bit is set when user data is transmitted / received. The D-bit requests end-to-end acknowledgement.

Note - No acknowledgement 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.

6.4.6 N_DC—Clear Confirm

Synopsis

```
#include <stream.h>
 #include <netx25/x25_proto.h >

 struct strbuf ctlb;
 struct strbuf datab;

 struct xdcnff disc;
 char          cud[MAX_LENGTH];
 .
 .
 .
 ctlb.len = sizeof(struct xdcnff);
 ctlb.buf = (char *)disc;

 datab.len = cudlen;
 datab.buf = cud;

 putmsg(x25_fd, &ctlb, NULL, 0);
```

Description

`N_DC` is used to confirm a previous clear indication (N_DI). The example shows a `putmsg`. Code Example 6–1 shows how a `getmsg` can be constructed. The control part of the Data packet is defined by the xdcnff structure. If Fast Select is in use, the data part of the message contains any clear user data.

The xdcnff structure is shown below:

```
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 */
 };
```

The members of the xdcnff structure are:

TABLE 6–11 Clear Confirm Parameters

Member	Description
indicated_qos	Non-zero value shows that facilities and QOS are being indicated.
rqos	Contains the facilities indicated. This is only used with the Charging Information facility.

6.4.7 N_DI—Clear Request / Indication

Synopsis

```
#include <stream.h>
 #include <netx25/x25_proto.h >

 struct strbuf ctlb;
 struct strbuf datab;

 struct xdiscf disc;
 char          cud[MAX_LENGTH];
 .
 .
 .
 ctlb.len = sizeof(struct xdiscf);
 ctlb.buf = (char *)disc;

 datab.len = cudlen;
 datab.buf = cud;

 putmsg(x25_fd, &ctlb, &datab, 0);
```

Description

N_DI is used when a Clear Request/Indication crosses the X.25 interface. When used with putmsg, N_DI is a Clear Request, when used with getmsg, it is a Disconnect Indication. Code Example 6–1shows how a getmsg can be constructed. The control part of the Call Request or Indication is defined by the xdiscf structure. If Fast Select is in use, the data part of the message contains any clear user data.

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 Clear Request the user can specify a non-zero cause code. This has no effect for an OSI CONS call; the value is set to zero by the system.

The Clear 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 Clear Confirm to the application when the Clear Confirmation is received. This guarantees that once the Clear Confirm is read 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 Clear Confirm 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.

The xdiscf structure is shown below:

```
struct xdiscf {
  unsigned char xl_type; /* Always XL_CTL */
  unsigned char xl_command; /* Always N_DI */
  unsigned char originator, /* Originator and Reason mapped
                                  from */
   reason, /* 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 */
 };
```

The members of the xdiscf structure are.

TABLE 6–12 Clear Request/Indication Parameters

member	Description
`originator`	OSI CONS mapping of the X.25 cause byte.
`reason`	OSI CONS mapping of the X.25 diagnostic byte. Refer to Chapter 9 for further information.
`cause`	The X.25 cause byte.
`diag`	The X.25 diagnostic byte.
`indicated_qos`	Non-zero value shows that facilities and QOS are being indicated.
`responder`	Contains the responding address.
`deflected`	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. This is used with the Charging Information facility and the Call Deflection facility.

Note - If a disconnect collision occurs, acknowledgement is taken to be complete.

6.4.8 `N_EAck`—Expedited Data Acknowledgement

Synopsis

```
#include <stream.h>
 #include <netx25/x25_proto.h >

 struct strbuf ctlb;

 struct xedatacf eack;
 .
 .
 .
 ctlb.len = sizeof(struct xedatacf);
 ctlb.buf = (char *)eack;

 putmsg(x25_fd, &ctlb, NULL, 0);
```

Description

N_EAck is used to acknowledge expedited data, carried by an X.25 Interrupt packet. The example shows a putmsg. Code Example 6–1 shows how a getmsg can be constructed. The control part of the Interrupt packet is defined by the xedatacf structure. There is no data part. An acknowledgment must be sent immediately on receipt of an Interrupt packet.

The xedatacf structure is shown below:

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

6.4.9 N_EData—Expedited Data

Synopsis

```
#include <stream.h>
 #include <netx25/x25_proto.h >

 struct strbuf ctlb;
 struct strbuf datab;

 struct xedataf edata;
 char          data[MAX_LENGTH];
 .
 .
 .
 ctlb.len = sizeof(struct xedataf);
 ctlb.buf = (char *)edata;

 datab.len = cudlen;
 datab.buf = cud;

 putmsg(x25_fd, &ctlb, &datab, 0);
```

Description

N_EData is used when expedited data, carried by an X.25 Interrupt packet, crosses the X.25 interface. The example shows a putmsg. Code Example 6–1 shows how a getmsg can be constructed. The control part of the Interrupt packet is defined by the xedataf structure. The data part of the message contains the user data. The expedited data is a confirmed primitive and must be acknowledged (see Section 6.4.8 “N_EAck—Expedited Data Acknowledgement” on page 56) before another expedited data unit can be requested or indicated.

The xedataf structure is shown below:

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

6.4.10 N_PVC_ATTACH—PVC Attach

Synopsis

```
#include <stream.h>
 #include <netx25/x25_proto.h >

 struct strbuf ctlb;

 struct pvcattf attach;
 .
 .
 .
 ctlb.len = sizeof(struct pvcattf);
 ctlb.buf = (char *)attach;

 putmsg(x25_fd, &ctlb, NULL, 0);
```

Description

N_PVC_ATTCH is used when an application wants to attach to a PVC.The control part of the PVC Attach is defined by the pvcattf structure. The example shows a putmsg. Code Example 6–1 shows how a getmsg can be constructed.

The pvcattf structure is shown below:

```
struct pvcattf {
        unsigned char     xl_type;      /* Always XL_CTL */
        unsigned char     xl_command;   /* Always N_PVC_ATTACH */
        unsigned short    lci;          /* Logical channel */
        uint32_t     link_id;      /* Link # */
        /* 0 for next 3 parameters implies use of default */
        unsigned char     reqackservice;
        unsigned char     reqnsdulimit;
        int               nsdulimit;
        int               result_code;     /* Non-zero - error */
};
```

The members used by pvcattf are:

TABLE 6–13 PVC Attach Parameters

Member	Description
lci	Contains the logical channel identifier of the required PVC.
link_id	Denotes the particular link identifier for the PVC.

TABLE 6–13 PVC Attach Parameters *(continued)*

Member	Description
reqackservice	If non-zero, denotes that the receipt acknowledgement 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 acknowledgements 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 acknowledgements with one data acknowledgement being received / sent for each D-bit data packet sent / received over the X.25 interface.
reqnsdulimit	If this is non-zero, use value in nsdulimit.
nsdulimit	Specifies the packet concatenation limit for NSDUs. If you want to use this parameter, reqnsdulimit must be non-zero. (The X.25 driver does not look at reqnsdulimit if nsdulimit is zero.)
result_code	In the attach message sent to the user as acknowledgment, this member denotes whether the attach was successful. The possible values are defined in the netx25/x25_proto.h file.

6.4.11 N_PVC_DETACH—PVC Detach

Synopsis

```
#include <stream.h>
 #include <netx25/x25_proto.h >

 struct strbuf ctlb;
 struct strbuf datab;

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

 datab.len = cudlen;
 datab.buf = cud;

 putmsg(x25_fd, &ctlb, &datab, 0);
```

Description

N_PVC_DETACH is used when an application wants to detach from a PVC. This allows the use of a stream to be changed. The control part of the PVC Detach is defined by the pvcdetf structure. The data part of the message contains any call user data. The synopsis shows a putmsg. Code Example 6–1 shows how a getmsg can be constructed.

The pvcdetf structure is shown below:

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

This structure has the following member:

TABLE 6–14 Listen Cancel Command/Response Parameters

Member	Description
reason_code	The reason for the detach, or a code indicating that a previous PVC Detach was successful.

Note - The PVC Detach message is acknowledged to the user by returning another PVC Detach message.

6.4.12 N_RC—Reset Response/Confirm

Synopsis

```
#include <stream.h>
 #include <netx25/x25_proto.h >

 struct strbuf ctlb;

 struct xrscf rc;
 .
 .
 .
 ctlb.len = sizeof(struct xrscf);
 ctlb.buf = (char *)rc;

 putmsg(x25_fd, &ctlb, NULL, 0);
```

Description

N_RC is used to respond to a previous reset. When used in a putmsg it is a Reset Response. In a getmsg it is a Reset Confirm. Code Example 6–1 shows how a getmsg can be constructed. The control part of the Reset Response or Confirm is defined by the xrscf structure. There is no data part.

The xrscf structure is shown below:

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

Note - A Reset 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—no Reset Confirmation is then required before another Reset Request can be sent. Normally, Resets are handled by the application.

6.4.13 N_RI—Reset Request/Indication

Synopsis

```
#include <stream.h>
 #include <netx25/x25_proto.h >

 struct strbuf ctlb;

 struct xrstf reset;
 .
 .
 .
 ctlb.len = sizeof(struct xrstf);
 ctlb.buf = (char *)reset;

 putmsg(x25_fd, &ctlb, NULL, 0);
```

Description

N_RI is used for resets. When used in a putmsg it is a Reset Request. In a getmsg it is a Reset Indication. Code Example 6–1 shows how a getmsg can be constructed. The X.25 cause and diagnostic bytes, cause and diag, are presented as well as the CONS originator and reason codes that are mapped from these. Refer to Chapter 9 for further information.

For a Reset Request, the user can specify a non-zero cause code. This has no effect for an OSI CONS call; the value is set to zero by the system.

The control part of the Reset Request or Indication is defined by the `xrstf` structure. There is no data part.

The `xrstf` structure is shown below:

```
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 */
 };
```

Note - A Reset primitive is an acknowledged service. It must be acknowledged before another Reset can be requested. A collision between a Reset Indication and a Reset Request is taken to acknowledge the Reset—no Reset Confirmation is then required before another Reset Request can be sent. Normally, Resets are handled by the application.

6.4.14 N_Xcanlis—Listen Cancel Command/Response

Synopsis

```
#include <stream.h>
 #include <netx25/x25_proto.h >

 struct strbuf ctlb;

 struct xcanlisf canlis;
 .
 .
 .
 ctlb.len = sizeof(struct xcanlisf);
 ctlb.buf = (char *)canlis;

 putmsg(x25_fd, &ctlb, NULL, 0);
```

Description

`N_Xcanlis` is used to cancel an interest in incoming calls.

When used with `putmsg`, `N_Xcanlis` is a Listen Cancel Command, when used with `getmsg`, it is a Listen Cancel Response. Code Example 6–1 shows how a `getmsg` can be constructed. The control part of the Listen Command or Response is defined by the `xcanlisf` structure. There is no data part.

Note - The Cancel Request removes all listen addresses from the stream. There is no way of cancelling a Listen on a particular address, for example, when the use of the stream is about to be changed by the application.

The control part of a Listen Cancel Command or Response message has a format defined in the `xcanlisf` structure:

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

This structure has the following member:

TABLE 6–15 Listen Cancel Command / Response Parameters

Member	Description
`c_result`	A non-zero value of the `c_result` flag indicates failure of the operation to cancel a Listen. This may indicate that the Listen was not present or that some connect event is outstanding. The closure of a stream on which there is a Listen also cancels the Listen, but in the case of the cancel command message, the stream remains open.

6.4.15 `N_Xlisten`—Listen Command / Response

Synopsis

```
#include <stream.h>
 #include <netx25/x25_proto.h >

 struct strbuf ctlb;
 struct strbuf datab;

 struct xlistenf listen;
 char          lisbuf[MAXLIS];
 .
 .
 .
 ctlb.len = sizeof(struct xlistenf);
 ctlb.buf = (char *)listen;

 datab.len = lislen;
 datab.buf = lisbuf;
```

```
putmsg(x25_fd, &ctlb, &datab, 0);
```

Description

N_Xlisten is used to listen for incoming calls. When used with putmsg, N_Xlisten is a Listen Command, when used with getmsg, it is a Listen Response. Code Example 6–1 shows how a getmsg can be constructed. The control part of the Listen Command or Response is defined by the xlistenf structure.

The data part 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];
```

Not all variables need be present. Refer to the individual variable descriptions below for more details.

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

TABLE 6–16 Variables for CUD matching

Variable Name	Description
l_cumode	Defines the type of matching: X25_DONTCARE The listener ignores the CUD; l_culength and l_cubytes are omitted. X25_IDENTITY The listener match is only made if all bytes of the CUD field are the same as the supplied l_cubytes. X25_STARTSWITH The listener match is only made if the leading bytes of the CUD Field are the same as the supplied l_cubytes.
l_culength	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. The range for l_culength is zero to 16 inclusive. If more than 16 bytes are present, the application still has to check the full CUD Field.
l_cubytes	String of bytes sought in the call user data field when l_cumode is X25_IDENTITY or X25_STARTSWITH.

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

TABLE 6–17 Variables for address matching

Variable Name	Description
l_mode	Defines the type of matching: X25_DONTCARE The listener ignores the address; l_type, l_length, and l_add are omitted. X25_IDENTITY The listener match is only made if all digits of the address are the same as the supplied l_add. X25_STARTSWITH The listener match is only made if the leading digits of the address are the same as the supplied l_add. X25_PATTERN The listener match is made on partial addresses, allowing the use of wildcard digits.
l_type	The type of the address entry; l_type 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, NSAP addresses (or extended addresses) are the important addresses, while for X.25 (80), where there is no NSAP address, the DTE address is the important quantity. 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 LLC2 on the LAN. In this case, the DTE address field is NULL and the X25_NSAP field is used.
l_length	Length of the address l_add in BCD digits—the common format for X.25 DTE and NSAP addresses. 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	Contains the address to be compared with the called address field of the incoming call packet. l_add is omitted when l_length is zero.

Note - To use wildcards, represent * by 0x0F and ? by 0x0E. * represents 0 or more characters. ? represents a single character.

The xlistenf structure is shown below:

```
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 */
 };
```

The members of the xlistenf structure are.

TABLE 6–18 Listen Command/Response Parameters

Member	Description
lmax	Maximum number of Connect Indications that the listener can handle at one time. 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 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 causes this flag to be set to a non-zero value.

Note - For example code using listening, refer to Chapter 4.

CHAPTER **7**

Network Layer ioctls

This chapter describes the Network Layer ioctls alphabetically. Refer to the tables below for functional groupings of Network Layer ioctls. Use the ioctls in this chapter to communicate with the Solstice X.25 software. To communicate with the network, for example to initiate calls, use the NLI commands described in Chapter 6.

Note - Note that some ioctls allow changes to be made to connections that may currently be in use—potentially disrupting users.

The header files used by the NLI ioctls are contained in the `/usr/include/netx25` directory.

7.1 ioctls Functional Grouping

These ioctls are related to NUI mapping. The NUI mapping table maps Network User Identifiers to particular facilities. The ioctls described in this section let you operate on NUI mappings. Note that any changes you make could disrupt other users. For this reason you require root access to use the ioctls that let you change the current settings.

TABLE 7–1 NUI mapping icotls

ioctl	description	access level
N_nuidel	delete specified NUI mapping	root only
N_nuiget	read specified NUI mapping	unrestricted
N_nuimget	read all NUI mappings	unrestricted
N_nuiput	store a set of NUI mappings	root only
N_nuireset	delete all NUI mappings	root only

These ioctls operate on a per multiplexor basis:

TABLE 7–2 Multiplexor ioctls

ioctl	description	access level
N_getstats	read X.25 multiplexor statistics	unrestricted
N_zerostats	reset X.25 multiplexor statistics to zero	root only

These ioctls operate on a per virtual circuit basis:

TABLE 7–3 Virtual circuit ioctls

ioctl	description	access level
N_getoneVCstats	get status and statistics for VC associated with current stream	unrestricted
N_getpvcmap	get default packet and window sizes	unrestricted
N_getVCstats	get per VC statistics	unrestricted

TABLE 7–3 Virtual circuit ioctls *(continued)*

ioctl	description	access level
N_getVCstatus	get per VC state and statistics	unrestricted
N_putpvcmap	change per VC packet and window sizes	unrestricted

These ioctls start and stop packet level tracing:

TABLE 7–4 Packet level tracing ioctls

ioctl	description	access level
N_traceon	start packet level tracing	root only
N_traceoff	stop packet level tracing	root only

These ioctls manage the X.25 routing table. Using them may override values set using x25tool. The ioctls are:

TABLE 7–5 Routing ioctls

ioctl	description	access level
N_X25_ADD_ROUTE	add a new route or update an existing route.	root only
N_X25_FLUSH_ROUTE	clear all entries from table.	root only
N_X25_GET_ROUTE	obtain routing information for specified address	unrestricted
N_GET_NEXT_ROUTE	obtain routing information for the next route in the table	unrestricted
N_RM_ROUTE	remove the specified route	root only

These ioctls operate on a link:

TABLE 7–6 Link ioctls

Header	**Header**	**Header**
N_getlinkstats	retreive per link statistics	unrestricted
N_linkconfig	configure wlcfg database for a link	unrestricted
N_linkent	configure a newly linked driver	unrestricted
N_linkread	read the wlcfg database	unrestricted

7.2 N_getlinkstats—Retrieve Per-Link Statistics

Retrieve statistics for a particular link.

Associated Structure

The following structure is associated with this ioctl:

```
struct perlinkstats {
        uint32_t linkid;                   /* link id (ppa)           */
        int      network_state;                 /* Network State           */
        uint32_t mon_array[link_mon_size]; /* L3perlinkmonarray       */
};
```

The members of the perlinkstats structure are:

TABLE 7–7 perlinkstats fields

Member	Description
`linkid`	the number of the link.
`network_state`	a code defining the network state. The codes are as follows: 1 Connecting to DXE 2 Connected resolving DXE 3 Random wait started 4 Connected and resolved DXE 5 DTE RESTART REQUEST 6 Waiting link disc reply 7 Buffer to enter `WtgRES` 8 Buffer to enter `L3restarting` 9 Buffer to enter `L_disconnect` 10 Registration request
`mon_array`	the array containing the statistics. `mon_array` is defined in the file `x25_control.h.`
`N_getnliversion`	read current NLI version Read which version of the Network Layer Interface is supported by the X.25 multiplexor. Solstice X.25 supports version 3.

Associated Structure

The following structure is associated with this ioctl:

```
struct nliformat {
     unsigned char  version;    /* NLI version number */
 };
```

The members of the `nliformat` structure are:

TABLE 7–8 `nliformat` fields

Member	Description
`version`	the version of NLI supported by the X.25 multiplexor.

7.3 N_getoneVCstats —Retrieve Per-Virtual-Circuit Statistics

This ioctl is used to retrieve per-virtual circuit state and statistics for the virtual circuit associated with the stream on which the ioctl is made.

Associated Structure

The vcinfo structure is shown below:

```
struct vcinfo {
        struct xaddrf     rem_addr;     /* = called for outward calls */
        /* = caller for inward calls  */
        uint32_t     xu_ident;     /* link id                    */
        uint32_t     process_id;   /* effective user id          */
        unsigned short    lci;          /* Logical Channel Identifier */
        unsigned char     xstate;       /* VC state                   */
        unsigned char     xtag;         /* VC check record            */
        unsigned char     ampvc;        /* =1 if a PVC                */
        unsigned char     call_direction;
        /* in=0, out=1                */
        unsigned char     domain;       /* was in 8.0, not in R7. Put it back */
        int                perVC_stats[perVCmon_size];
};
```

The members of the vcinfo structure are:

TABLE 7–9 vcinfo structure fields

Member	Description
rem_addr	The called address if its an outgoing call, or the calling address for incoming calls.
xu_ident	The link identifier.
process_id	The relevant user id.
lci	The logical channel identifier.
xstate	The VC state.
xtag	The VC check record.

TABLE 7–9 `vcinfo` structure fields *(continued)*

Member	Description
`ampvc`	Set to 1 if this is a PVC.
`call_direction`	0 indicates in incoming call, 1 an outgoing call.
`perVC_stats`	An array containing the per-virtual circuit statistics. The array is defined in the `x25_control.h` file.

7.4 `N_getpvcmap`—Get PVC Default Packet/Window Sizes

This ioctl is used to read the default packet and window sizes of active PVCs.

Associated Structure

The following structure is associated with this ioctl:

```
struct pvcmapf {
   struct pvcconff entries[MAX_PVC_ENTS]; /* Data buffer          */
   int             first_ent;             /* Where to start search   */
   unsigned char   num_ent;               /* Number entries returned */
 };
```

The members of the `pvcmapf` structure are:

TABLE 7–10 getpvcmap fields

Member	Description
`entries`	Contains the structure for the returned mapping entries.
`first_ent`	Informs the X.25 multiplexor where to start or restart the table read. It should initially be set to 0, to indicate starting at the beginning of the table. On return, it points to the next entry.
`num_ent`	Indicate the number of mapping entries returned in the entries member. It should be set to 0 before making the ioctl.

7.5 N_getstats—Get X.25 Multiplexor Statistics

This ioctl is used to read the statistics counts for the X.25 multiplexor since network start-up or since they were last reset by an N_zerostats ioctl (see below). Statistics are maintained an a multiplexor basis—separate link statistics are not available.

Associated Structure

The N_getstats structure associated with this ioctl is an integer array of size mon_size, defined in the file x25_control.h. Entries and meanings include the following:

TABLE 7–11 N_getstats structure

Entry	Description
cll_in_g	Calls received and indicated
caa_in_gc	Call established for outgoing
caa_out_g	Call established for incoming
ed_in_g	Interrupts received
ed_out_g	Interrupts sent
rnr_in_g	Receiver not ready received
rnr_out_g	Receiver not ready sent
rr_in_g	Receiver ready rvcd
rr_out_g	Receiver ready sent

TABLE 7–11 N_getstats structure *(continued)*

Entry	Description
rst_in_g	Resets received
rst_out_g	Resets sent
rsc_in_g	Restart confirms received
rsc_out_g	Restart confirms sent
clr_in_g	Clears received
clr_out_g	Clears sent
clc_in_g	Clear confirms received
clc_out_g	Clear confirms sent
cll_coll_g	Call collision count (not rjc)
cll_uabort_g	Calls aborted by user b4 sent
rjc_buflow_g	Calls rejd no buffs b4 sent
rjc_coll_g	Calls rejd - collision DCE mode
rjc_failNRS_g	Calls rejd negative NRS resp
rjc_lstate_g	Calls rejd link disconnecting
rjc_nochnl_g	Calls rejd no lcns left
rjc_nouser_g	In call but no user on NSAP
rjc_remote_g	Call rejd by remote responder
rjc_u_g	Call rejd by NS user
dg_in_g	DIAG packets in

TABLE 7–11 N_getstats structure *(continued)*

Entry	Description
dg_out_g	DIAG packets out
p4_ferr_g	Format errors in P4
rem_perr_g	Remote protocol errors
res_ferr_g	Restart format errors
res_in_g	Restarts received (inc DTE/DXE)
res_out_g	Restarts sent (inc DTE/DXE)
vcs_labort_g	Circuits aborted via link event
r23exp_g	Circuits hung by r23 expiry
l2conin_g	Link level connect established
l2conok_g	LLC connections accepted
l2conrej_g	LLC connections rejd
l2refusal_g	LLC connect requests refused
l2lzap_g	Oper requests to kill link
l2r20exp_g	R20 retransmission expiry
l2dxeexp_g	DXE/connect expiry
l2dxebuf_g	DXE resolv abort - no buffers
l2noconfig_g	No config base - error
xiffnerror_g	Upper i/f bad M_PROTO type
xintdisc_g	Internal disconnect events

TABLE 7–11 `N_getstats` structure *(continued)*

Entry	Description
`xifaborts_g`	Interface abort_vc called
`PVCusergone_g`	Count of non-user interactions
`max_opens_g`	highest no. simul. opens so far
`vcs_est_g`	VCs established since reset
`bytes_in_g`	Total data bytes received
`bytes_out_g`	Total data bytes sent
`dt_in_g`	Count of data packets sent
`dt_out_g`	Count of data packets received
`res_conf_in_g`	Restart Confirms received
`res_conf_out_g`	Restart Confirms sent
`reg_in_g`	Registration requests received
`reg_out_g`	Registration requests sent
`reg_conf_in_g`	Registration confirms received
`reg_conf_out_g`	Registration confirms sent
`l2r28exp_g`	R28 retransmission expiry

7.6 N_getVCstats—Get Per-Virtual-Circuit Statistics

This ioctl is used to retrieve per-virtual circuit state and statistics, for all virtual circuits currently active over all configured links.

Associated Structure

The vcstatsf structure, defined in x25_control.h, takes this format:

```
struct vcstatsf {
        int first_ent;          /* Where to start search   */
        unsigned char num_ent   /* Number entries returned */
        struct pervcinfo vc;    /* Data buffer, extendable by*/
                                /* malloc overlay*/
 };
```

The members of the vcstatsf structure are:

TABLE 7–12 `vcstatsf` fields

Member	Description
`first_ent`	Informs the X.25 multiplexor where to start or restart the table read. On return, it is set to point the next entry.
`num_ent`	Indicates the number of virtual circuit entries returned in the `vc` member.
`vc`	This is either a single `pervcinfo` structure or an array of `pervcinfo` structures, of size `MAX_VC_ENTRIES`, each containing the state and statistics of an individual virtual circuit. If a single `pervcinfo` structure is used, and `num_ent` is not 0, and statistics are returned of the virtual circuit specified in the `lci` member of the `pervcinfo` structure, with a link identifier specified using `xu_ident`. If a single `pervcinfo` structure is used, and `num_ent` is 0, the number of open virtual circuits is returned in `first_ent`. If an array of pervcinfo structures is used, and num_ent is set to 0, statistics are returned for the Logical Channel Number set using the `lci` member. If an array of `pervcinfo` structures is used, and num_ent is set to 1, statistics are returned for all virtual circuits on the link specified using xu_ident. If an array of `pervcinfo` structures is used, and `num_ent` is set to `MAX_VC_ENTRIES`, statistics are returned for all virtual circuits on all links.

The contents of the `pervcinfo` structure are:

```
struct pervcinfo {
        struct xaddrf     rem_addr;     /* = called for outward calls */
        /* = caller for inward calls  */
        uint32_t     xu_ident;     /* link id                    */
        uint32_t     process_id;   /* effective user id          */
        unsigned short    lci;          /* Logical Channel Identifier */
        unsigned char     xstate;       /* VC state                   */
        unsigned char     xtag;         /* VC check record            */
        unsigned char     ampvc;        /* =1 if a PVC                */
        unsigned char     call_direction;
        /* DIRECTION_xxx (see mib)    */
        unsigned char     domain;       /* was in 8.0, not in R7. Put it back */
        uint32_t     perVC_stats[perVCstat_size];
        /* Per-VC statistics array    */
        /*
         * move these to the end, so that the first bit of the struct is
         * identical to the 8.0 one
        */
        unsigned char     vctype;       /* what _is_ this? */
        struct xaddrf     loc_addr;     /* = caller for outward calls */
        /* = called for inward calls  */
```

```
        uint32_t      start_time;   /* time the VC was created */
};
```

`xstate` contains the state of the VC. Possible states and meanings are:

TABLE 7–13 `xstate` summary

Entry	Description
`Idle`	Record is not in use
`AskingNRS`	CR is being validated by NRS
`P1`	VC state is READY
`P2`	VC in DTE CALL REQUEST
`P3`	VC in DXE INCOMING CALL
`P5`	VC in CALL COLLISION
`DataTransfer`	VC in P4 (see xflags
`DXEbusy`	VC in P4, DXE sent RNR
`D2`	VC in DTE RESET REQUEST
`D2pending`	Wanting buffer for RESET
`WtgRCU`	Waiting U RSC to int.err.
`WtgRCN`	Waiting X.25 RSC for user
`WtgRCNpending`	Buffer reqd to enter state
`P4pending`	Buffer reqd for X.25 RSC
`pRESUonly`	Buffer for user rst only
`RESUonly`	User only being reset

TABLE 7–13 `xstate` summary *(continued)*

Entry	Description
`pDTransfer`	Buffer for RSC to user
`WRCUpending`	Buffer reqd internal RST
`DXErpending`	Buffer reqd RST indication
`DXEresetting`	Waiting U RSC to X.25 RI
`P6`	VC in DTE CLEAR REQUEST
`P6pending`	Wanting buffer for CLEAR
`WUcpending`	Buffer reqd DI no netconn
`WUNcpending`	Buffer reqd internal DI
`DXEcpending`	Buffer reqd CLR REQ->User
`DXEcfpending`	Buffer reqd CLC to User

`perVC_stats` contains statistics counts, as follows:

TABLE 7–14 `perVC_stats` summary

Entry	Description
`cll_in_v`	Calls received and indicated
`cll_out_v`	Calls sent
`caa_in_v`	Call established for outgoing
`caa_out_v`	Call established for incoming
`dt_in_v`	Data packets received

TABLE 7–14 perVC_stats summary *(continued)*

Entry	Description
dt_out_v	Data packets sent
ed_in_v	Interrupts received
ed_out_v	Interrupts sent
rnr_in_v	Receiver not ready received
rnr_out_v	Receiver not ready sent
rr_in_v	Receiver ready rvcd
rr_out_v	Receiver ready sent
rst_in_v	Resets received
rst_out_v	Resets sent
rsc_in_v	Restart confirms received
rsc_out_v	Restart confirms sent
clr_in_v	Clears received
clr_out_v	Clears sent
clc_in_v	Clear confirms received
clc_out_v	Clear confirms sent

7.7 N_getVCstatus—Get Per-Virtual-Circuit Statistics

Note - This ioctl has been superceded by the `N_getVCstats` ioctl. It is retained for backward compatibility with Solstice X.25 8.*x*. When writing new applications, use `N_getVCstats`.

This ioctl is used to retrieve per-virtual circuit state and statistics, for all virtual circuits currently active over all configured links.

Associated Structure

The `vcstatusf` structure takes this format:

```
struct vcstatusf {
   struct vcinfo       vcs[MAX_VC_ENTS];  /* Data buffer      */
   int                 first_ent;         /* Where to start search */
   unsigned char       num_ent;           /* Number entries returned */
 };
```

The members of the `vcstatusf` structure are:

TABLE 7–15 `vcstatusf` fields

Member	Description
`vcs`	An array of `vcinfo` structures, each of which contains the state and statistics for an individual virtual circuit.
`first_ent`	Informs the X.25 multiplexor where to start or restart the table read. It should initially be set to 0, to indicate starting at the beginning of the table. On return, it will be set to point to the next entry to be retrieved.
`num_ent`	Indicates the number of virtual circuit entries returned in the `vcs` member. It should be set to 0 before making the ioctl.

The contents of the `vcinfo` structure are:

```
struct vcinfo {
        struct xaddrf     rem_addr;     /* = called for outward calls */
        /* = caller for inward calls  */
        uint32_t     xu_ident;     /* link id                    */
        uint32_t     process_id;   /* effective user id          */
        unsigned short    lci;          /* Logical Channel Identifier */
```

```
        unsigned char     xstate;       /* VC state                    */
        unsigned char     xtag;         /* VC check record             */
        unsigned char     ampvc;        /* =1 if a PVC                 */
        unsigned char     call_direction;
        /* in=0, out=1                 */
        unsigned char     domain;       /* was in 8.0, not in R7. Put it back */
        int                perVC_stats[perVCmon_size];
};
```

The `xstate` member contains the state of the VC. Possible states and meanings are:

TABLE 7–16 xstate summary

Entry	Description
`Idle`	Record is not in use
`AskingNRS`	CR is being validated by NRS
`P1`	VC state is READY
`P2`	VC in DTE CALL REQUEST
`P3`	VC in DXE INCOMING CALL
`P5`	VC in CALL COLLISION
`DataTransfer`	VC in P4 (see xflags
`DXEbusy`	VC in P4, DXE sent RNR
`D2`	VC in DTE RESET REQUEST
`D2pending`	Wanting buffer for RESET
`WtgRCU`	Waiting U RSC to int.err.
`WtgRCN`	Waiting X.25 RSC for user
`WtgRCNpending`	Buffer reqd to enter state
`P4pending`	Buffer reqd for X.25 RSC

TABLE 7–16 xstate summary *(continued)*

Entry	Description
pRESUonly	Buffer for user rst only
RESUonly	User only being reset
pDTransfer	Buffer for RSC to user
WRCUpending	Buffer reqd internal RST
DXErpending	Buffer reqd RST indication
DXEresetting	Waiting U RSC to X.25 RI
P6	VC in DTE CLEAR REQUEST
P6pending	Wanting buffer for CLEAR
WUcpending	Buffer reqd DI no netconn
WUNcpending	Buffer reqd internal DI
DXEcpending	Buffer reqd CLR REQ->User
DXEcfpending	Buffer reqd CLC to User

The perVC_stats member contains statistics. Entries are statistics counts, as follows:

TABLE 7–17 perVC_stats summary

Entry	Description
cll_in_v	Calls received and indicated
cll_out_v	Calls sent
caa_in_v	Call established for outgoing

TABLE 7–17 `perVC_stats` summary *(continued)*

Entry	Description
caa_out_v	Call established for incoming
dt_in_v	Data packets received
dt_out_v	Data packets sent
ed_in_v	Interrupts received
ed_out_v	Interrupts sent
rnr_in_v	Receiver not ready received
rnr_out_v	Receiver not ready sent
rr_in_v	Receiver ready rvcd
rr_out_v	Receiver ready sent
rst_in_v	Resets received
rst_out_v	Resets sent
rsc_in_v	Restart confirms received
rsc_out_v	Restart confirms sent
clr_in_v	Clears received
clr_out_v	Clears sent
clc_in_v	Clear confirms received
clc_out_v	Clear confirms sent

7.8 N_linkconfig—Configure the wlcfg Database

This ioctl is used to configure the wlcfg database for a link. The wlcfg database appropriate to a link is carried as the M_DATA part of the ioctl N_linkconfig. The U_LINK_ID member of the wlcfg structure specifies the link to be configured. The wlcfg database structure is defined in the /usr/include/netx25/x25_control.h file.

Note - This ioctl affects currently open connections and could therefore disrupt users significantly. For this reason it can only be used by root.

The wlcfg database structure contains the members described below:

U_LINK_ID

The upper level link identifier which is quoted by upper level software in the xaddrf address structure to specify which link a call is to be sent on. It is also used to identify which link an incoming call arrived on.

NET_MODE

This determines the characteristics of the network protocol Possible values are:

TABLE 7–18 NET_MODE values

String	Value	Network, X.25 Type, or Country
X25_LLC	1	X.25(84/88)/LLC2
X25_88	2	X.25(88)
X25_84	3	X.25(84)
X25_80	4	X.25(80)
GNS	5	UK
AUSTPAC	6	Australia
DATAPAC	7	Canada

TABLE 7–18 NET_MODE values *(continued)*

String	Value	Network, X.25 Type, or Country
DDN	8	USA
TELENET	9	USA
TRANSPAC	10	France
TYMNET	11	USA
DATEX_P	12	Germany
DDX_P	13	Japan
VENUS_P	14	Japan
ACCUNET	15	USA
ITAPAC	16	Italy
DATAPAK	17	Sweden
DATANET	18	Holland
DCS	19	Belgium
TELEPAC	20	Switzerland
F_DATAPAC	21	Finland
FINPAC	22	Finland
PACNET	23	New Zealand
LUXPAC	24	Luxembourg
X25_Circuit	25	dialup call

X25_VSN

This determines the version of the X.25 protocol used over the network. Allowed values are:

- 0 indicating X.25(80)
- 1 indicating X.25(84)
- 2 indicating X.25(88)

Setting NET_MODE to X25_LLC overrides an X.25 (80) value.

L3PLPMODE

Indicates whether the link is DTE or DCE. Allowed values are:

- 0 indicating DCE
- 1 indicating DTE
- 2 indicating that this is to be resolved by following the procedures in ISO 8208 for DTE-DTE operation

LPC to HPC

Logical channel range assigned to PVCs.

LIC to HIC

Logical channel range assigned to one way incoming logical channels.

LTC to HTC

Logical channel range assigned to two-way logical channels.

LOC to HOC

Logical channel range assigned to one-way outgoing logical channels.

Note - In a DTE/DTE environment, one of the interacting pairs views these ranges as a DCE, for example, LIC to HIC are viewed as one-way *outgoing*. H*x*C = L*x*C = 0 denotes no channels in that grouping.

NPCchannels, NICchannels, NTCchannels, NOCchannels and Nochnls

The number of logical channels assigned. This is calculated from the logical channel ranges and can only be changed only by altering these ranges.

THISGFI

0x10 indicates Modulo 8. 0x20 indicates Modulo 128 sequence numbering operates on the network.

LOCMAXPKTSIZE

The maximum acceptable size of local to remote data packets, expressed as a power of 2.

`REMMAXPKTSIZE`

The maximum acceptable size of remote to local data packets expressed as a power of 2.

`LOCDEFPKTSIZE`

The default local-to-remote packet size expressed as a power of 2.

`REMDEFPKTSIZE`

The default remote-to-local packet size on a particular link, expressed as a power of 2.

`LOCMAXWSIZE`

The maximum acceptable local to remote X.25 window size.

`REMMAXWSIZE`

The maximum acceptable remote to local X.25 window size.

`LOCDEFWSIZE`

The local-to-remote default window size.

`REMDEFWSIZE`

The remote-to-local default window size.

`MAXNSDULEN`

The default maximum length beyond which concatenation is stopped and the data currently held is passed to the NS-user. This parameter can be overridden on a per-circuit basis using the `nsdulimit` parameter on `N-CONNECT` requests and `N-CONNECT` responses.

`ACKDELAY`

The maximum delay in ticks (0.1 second units) over which a pending acknowledgement will be withheld. The default value is 5, the permitted range 1—32000.

`T20value`

The length of DTE timer T20, the Restart Request Response Timer, in ticks (0.1 second units). The default value is 1800. The permitted range is 0—32000.

`T21value`

The length of DTE timer T21, the Call Request Response Timer, in ticks (0.1 second units). The default value is 2000. The permitted range is 0—32000.

`T22value`

The length of DTE timer T22, the Reset Request Response Timer, in ticks (0.1 second units). The default value is 1800. The permitted range is 0—32000.

`T23value`

The length of DTE timer T20, the Clear Request Response Timer, in ticks (0.1 second units). The default value is 1800. The permitted range is 0—32000.

`Tvalue`

The maximum time over which acknowledgments of data received from the remote transmitter will be withheld. After this timer expires any withheld acknowledgments are carried by a Receive Not Ready (RNR) packet. This timer ensures that non-receipt of acknowledgment by the remote transmitter does not cause resets within the virtual circuit. This timer does not cause transmission of window status every `Tvalue` ticks (0.1 second units). The default value is 750. The permitted range is 0—32000.

`T25value`

The length of DTE timer T25, the Window Rotation Timer, in ticks (0.1 second units). The default value is 1500, as specified in ISO 8208. The permitted range is 0—32000.

The code may be configured to be lenient in the case of flow control inhibition (see Section 11.2 of ISO 8208). That is, a decision has to be made in order to cater for the case when the remote station does not rotate the window fast enough to prevent expiration of T25. ISO 8208 recommends strongly that high level protocols be used to effect recovery, to achieve this, set T25 to either zero (implying infinite) or a very large value.

The timer `Tvalue`, should be set to a value approximately half `T25value`, in order to prevent the remote PLP from resetting on T25 expiration. The timer ACKDELAY should be approximately 0.5 seconds, although this recommendation may change after evaluation and experience is gained.

Finally, the `idlevalue` timer may be set according to how quickly the LAN administration wishes the resource to be reclaimed, while `connectvalue` should be about three times the T20 value.

Note also that ISO 8208 recommends that the retry values R20, R22 and R23 should never be set to zero in order to cater for the possibility of collisions (see footnote to Figure 6, ISO 8208).

`T26value`

The length of DTE timer T26, the Interrupt Response Timer, in ticks (0.1 second units). The default value is 1800. The permitted range is 0—32000.

`T28value`

The length of DTE timer T28, the Registration Request Timer, in ticks (0.1 second units). The default value is 1800. The permitted range is 0—32000.

`idlevalue`

The number of ticks (0.1 second units) over which a link-level connection associated with no connections is maintained. This timer is meaningful on a LAN or on a dial-up WAN connection. The default value is 600. The permitted range is 0—32000.

`connectvalue`

The number of ticks (0.1 second units) over which the DTE/DCE resolution phase must be complete. On expiration of this timer, the link connection is disconnected and all pending connections are aborted. The default value is 2000. The permitted range is 0—32000.

`R20value`

The DTE Restart Request Retransmission Count. The default value is 1. The permitted range is 1—255.

`R22value`

The DTE Restart Request Retransmission Count. The default value is 1. The permitted range is 1—255.

`R23value`

The DTE Restart Request Retransmission Count. The default value is 1. The permitted range is 1—255.

`localdelay` and `accessdelay`

In milliseconds, the values of the transit delay attributed to internal processing and the effect of the line transmission rate. These values are used to check whether any maximum acceptable end-to-end transit delay specified in an N-CONNECT request or an N-CONNECT indication is in fact available.

`locmaxthclass`

The maximum value of the throughput class quality of service parameter in the local-to-remote direction which is supported. According to ISO 8208 this parameter is bounded in the range 3 and ≤ 12 corresponding to a range 75 to 48000 bits/second.

`remmaxthclass`

The maximum value of the throughput class quality of service parameter in the remote-to-local direction which is supported. According to ISO 8208 this parameter is bounded in the range 3 and ≤ 12 corresponding to a range 75 to 48000 bits/second.

`locdefthclass`

In some PSDNs, for example, TELENET, negotiation of throughput class is constrained to be towards a configured default throughput class. In such cases the flag `thclass_neg_to_def` (see below) is non-zero and `locdefthclass` is the default for the local-to-remote direction. In other PSDNs, `locdefthclass` should be set equal to the value of `locmaxthclass` (see above).

Note that `locmaxthclass` must be greater than or equal to `locdefthclass`.

`remdefthclass`

In some PSDNs, for example, TELENET, negotiation of throughput class is constrained to be towards a configured default throughput class. In such cases the flag `thclass_neg_to_def` is non-zero and `remdefthclass` is the default for the remote-to-local direction. In other PSDNs, set `remdefthclass` equal to the value of `remmaxthclass` (see above).

Note that `remmaxthclass` must be greater than or equal to `remdefthclass`.

`locminthclass`

According to ISO 8208, the throughput class parameter must be greater than or equal to 3 and less than or equal to 12. Some PSDNs may provide a different mapping, in which case `locminthclass` is the minimum value in the local-to-remote direction. Note that `locmaxthclass` must be less than or equal to `locdefthclass` which must be greater than or equal to `locminthclass`.

`remminthclass`

According to ISO 8208, the throughput class parameter is defined in the range 3 and 12. Some PSDNs may provide a different mapping, in which case `remminthclass` is the minimum value in the remote-to-local direction. Note that `remmaxthclass` must be greater than or equal to `remdefthclass` which must be greater than or equal to `remminthclass`.

`CUG_CONTROL`

This member controls Closed User Group actions in two ways. Firstly, it describes the type, if any, of Closed User Group facilities subscribed to. This is used to choose the appropriate encoding for any closed user group facilities in `N-CONNECT` requests. Secondly, it specifies the action to be taken if the Closed User Group optional facility is present in an incoming call. It is a bit map where the bits have the following meanings:

TABLE 7–19 bit map summary

Bit	Description
0	subscription to CUGs with no Outgoing or Incoming Access
1	subscription to Preferential CUG
2	subscription to CUGs with Outgoing Access
3	subscription to CUGs with Incoming Access (For Information Only)
4	subscription to Basic Format CUGs
5	subscription to Extended format CUGs
6	reject incoming calls containing any Closed User Group facility
7	reserved

Bits 0 and 2 are mutually exclusive as are bits 4 and 5.

SUB_MODES

This member is a bit map, which contains information on the various subscription options for a particular PSDN link. The entries mean:

TABLE 7–20 SUB_MODES summary

Entry	Description
SUB_EXTENDED	Subscribe to extended call packets. This permits the use of extended CALL REQUEST and CALL ACCEPT packets.
BAR_EXTENDED	Treat incoming extended call packets as a procedure error.The use of extended call packets allows window and packet size negotiation. Not setting the BAR_EXTENDED member permits the use of extended INCOMING CALL and CALL CONFIRM packets.
SUB_FSELECT	Subscribe to fast select with no restriction on response. This permits the use of fast select on INCOMING CALL packets.
SUB_FSRRESP	Subscribe to fast select with restriction on response. This permits the use of fast select with restricted response on INCOMING CALL packets.
SUB_REVCHARGE	Subscribe to reverse charging. This permits the use of reverse charges on INCOMING CALL packets.
SUB_LOC_CHG_PREV	Subscribe to local charging prevention. If set, this member has two effects. It prevents the use of reverse charges on INCOMING CALL packets regardless of the setting of SUB_REVCHARGE, and any CALL REQUEST packet will have the reverse charges facility automatically inserted.
SUB_TOA_NPI_FMT	Subscribe to using TOA / NPI address format. This specifies that all call set-up and clearing packets transmitted will always use the TOA / NPI address format.
BAR_TOA_NPI_FMT	Treat incoming TOA / NPI address formats as a procedure error. The BAR_TOA_NPI_FMT entry if set specifies that any call set-up and clearing packets received employing the TOA / NPI address format will be treated as a procedure error.
BAR_CALL_X32_REG	Refuse to accept incoming calls while X.32 registration is incomplete.

TABLE 7–20 SUB_MODES summary *(continued)*

Entry	Description
SUB_NUI_OVERRIDE	Subscribe to NUI override.The SUB_NUI_OVERRIDE entry if set specifies that when an NUI is provided in a CALL REQUEST, then any associated subscription time options override the facilities which apply to the interface, for the duration of that particular call.
BAR_INCALL	Bar incoming calls.
BAR_OUTCALL	Bar outgoing calls.

Some PSDNs require certain procedures to be followed which are not standard for all X.25 networks. The structure psdn_local contains the flags used to tune the actions of the X.25 driver to the requirements of the particular network to which the configuration refers. The entries and values taken by the psdn_local structure are described below.

PSDN_MODES

This is used to tune the various options for a particular PSDN link. It is a bit map in which the various entries when set imply:

TABLE 7–21 PSDN Modes

Mode	Description
ACC_NODIAG	Allow the omission of the diagnostic byte in incoming RESTART, CLEAR and RESET INDICATION.
USE_DIAG	Use diagnostic packets.
CCITT_CLEAR_LEN	Restrict the length of a CLEAR INDICATION to 5 bytes and a CLEAR CONFIRM to 3 bytes.
BAR_DIAG	Disallow diagnostic packets.
DISC_NZ_DIAG	Discard diagnostic packets on a non-zero LCN.
ACC_HEX_ADD	Allow DTE addresses to contain hexadecimal digits.
BAR_NONPRIV_LISTEN	Disallow a non-privileged user (that is, one without superuser privilege) from listening for incoming calls.

TABLE 7–21 PSDN Modes *(continued)*

Mode	Description
INTL_PRIO	Prioritize international calls.
DATAPAC_PRIORITY	Use DATAPAC (1976) priority rules.
ISO_8882_MODE	Use strict ISO8882 conformance.
X121_MAC_OUT	Keep X.121 address in call packets to LAN.
X121_MAC_IN	Put X.121 address in call packets from LAN.

The BAR_DIAG and DISC_NZ_DIAG entries specify the treatment of incoming diagnostic packets. When BAR_DIAG is set, incoming diagnostic packets are handled as follows. If USE_DIAG is set, and the link is configured as a DCE, then a diagnostic packet is sent to the DTE. Otherwise, the incoming diagnostic packet is simply discarded. When DISC_NZ_DIAG is set, diagnostic packets will be discarded when received on non-zero logical channel numbers. If ACC_HEX_ADD is set, DTE addresses are not restricted to containing only BCD digits.

intl_addr_recogn

The main use of this feature is in conjunction with the intl_prioritised member discussed below. Possible values are:

TABLE 7–22 Intl_addr_recogn summary

Value	Description
0	International calls are not distinguished.
1	The DNIC of the called DTE address is examined and compared to that held in psdn_local members dnic1 and dnic2. A mismatch implies an international call.
2	International calls are distinguished by having a “1” prefix on the called DTE address; for example, DATAPAC has this feature.
3	International calls are distinguished by having a “0” prefix on the called DTE address.

dnic1, dnic2

The first four BCD digits of the DNIC and is only used when intl_addr_recogn has the value one.

intl_prioritised

This determines whether some prioritization method is to be used for international calls, and is used in conjunction with prty_encode_control and prty_pkt_forced_value.

intl_prioritised has two values: zero implying no priority, while non-zero implies an attempt to prioritize according to ptry_encode_control.

intl_addr_recogn has the value one.

prty_encode_control

This describes how the priority request is to be encoded for this PSDN. Values are:

TABLE 7–23 prty_encode_control values

Value	Description
0	The priority is encoded according to section 3.3.3 of Annex G, Blue Book Volume VIII, Fascicle VIII.3 (CCITT, 1988).
1	Encode the priority request using the DATAPAC Priority Bit (1976 version).
2	Encode the priority request using the DATAPAC Traffic Class (1980 version which uses the Calling Network facility marker).

prty_pkt_forced_value

If this entry is non-zero then it implies that all priority call requests and incoming calls should have the associated packet size parameter forced to this value (note that the actual packet size is two to the power of this parameter; for example, 7 implies 128 byte packets). A zero value implies no special action on packet size is required.

src_addr_control

This provides the means to override or set the calling address in outgoing call requests for this PSDN. It takes the following values:

TABLE 7–24 `src_addr_control` values

Value	Description
0	No special action. Calling DTE addresses are encoded as and if provided by the network service user.
1	Force omission of the calling DTE address, even if the network service user supplied one.
2	If the network service user does not supply a DTE address, use the configured DTE address (`local_address`) for this PSDN (which can, of course, be NULL).
3	Force the calling DTE address to that contained in `local_address`, even if the network service user supplied one.

`dbit_control`

This member specifies the action to be taken:

- during the call setup phase, where both parties do not agree on the use of the D-bit;
- during the data transfer phase, on receipt of a data packet with the D-bit set, where the use of the D-bit has not been agreed by both parties.

Actions which may be specified during the call setup phase are:

- Leave the D-bit set and pass the packet on.
- Zero the D-bit and pass the packet on.
- Clear the call.

Actions which may be specified during the data transfer phase are:

- Leave the D-bit set and pass the packet on.
- Zero the D-bit and pass the packet on.
- Reset the call.

`thclass_neg_to_def`

This accommodates certain network procedures which dictate that negotiation of throughput class must be towards the default value (for example, TELENET), the default value being configured into the member `defthclass`. A non-zero value in this member requests use of this option, zero implies non-use.

`thclass_type`

This provides the means by which throughput class encodings can be used to assign window and packet sizes (according to the arrays `thclass_wmap` and `thclass_pmap` described below). It should be noted that some implementations of

X.25 do not use the X.25 packet and window negotiation but instead rely on mapping the throughput class to these parameters (see thclass_type 1,2 and 3). Thclass_type should be used on such PSDNs. Note also that the values of locmaxthclass and remmaxthclass may have an effect on what is achieved through the mapping.

The values assigned to thclass_type to indicate the mapping are:

TABLE 7–25 thclass_type values

Value	Description
0	No special action is to be taken on throughput class.
1	Use only the low nibble of the throughput class parameter to map window and packet size for both directions and encode the high nibble as zero. Note that the window and packet sizes are intended to be asserted by the throughput class parameter.
2	Use only the high nibble of the throughput class parameter to map window and packet size for both directions and encode the low nibble as zero. Note that the window and packet sizes are intended to be asserted by the throughput class parameter.
3	Use both nibbles of the throughput class to map window and packet size for the appropriate directions. Note that the window and packet sizes are intended to be asserted by the throughput class parameter.

Values 1, 2 and 3 are intended for use on non-standard X.25 PSDN implementations. Note the following.

For the special values 1 and 2:

- Do not select these values when window and packet sizes can appear in call setup packets (that is, subscription to window and packet size negotiation) since this algorithm is designed for those PSDNs which support only the mapping procedure.
- In call requests, the network service user should specify equal values for locthroughput and remthroughput in the qosformat, to ensure that the correct behavior is obtained (see also high and low nibble usage for these two values).
- The user will be barred from negotiating window and packet sizes, and the throughput class will not be indicated in a connect indication.

For the value 3, window and packet sizes can be negotiated by the network service user only through the throughput class parameter. Negotiations through the flow negotiation parameters when subscribing to the extended facility option are overridden. However, as for values 1 and 2, this value is intended only for cases where this is the only means of negotiating window and packet sizes.

Since window and packet sizes can be mapped using these three values without the use of window and packet negotiation facilities, it is important that the map (thclass_wmap and thclass_pmap) is correct for the PSDN, in order to ensure that both called and calling parties agree on the values associated with a particular throughput class.

thclass_wmap, thclass_pmap

The mapping between the value of the throughput class (a number 0 to 15) and a window and packet parameter. Zero in this table indicates that the currently set or default value be used.

local_address

Holds the local DTE address for this X.25 link in a byte array, local_address.lsap_add, with an associated length byte local_address.lsap_len.

7.9 N_linkent—Configure a Newly Linked Driver

This ioctl is sent downstream by the x25netd process to configure a newly linked driver below the X.25 multiplexor. It supplies the parameters necessary to identify the link via the identifier and to register the mode of the lower driver.

Note - This ioctl is only used when X.25 is initializing. As it affects currently open connections and could therefore disrupt users significantly, it can only be used by root. It should *not* be used by user applications, as it may be withdrawn from future versions of Solstice X.25.

7.10 N_linkmode—Alter the Characteristics of a Link

This ioctl is used to read or change the SUB_MODES Member of a particular wlcfg database appropriate to a link. This configuration ioctl is used to alter characteristics of a link's operation, for example, to temporarily bar incoming calls.

Note - This ioctl affects currently open connections and could therefore disrupt users significantly. For this reason it can only be used by root.

Associated Structure

The parameters are carried as the M_DATA part of the N_linkmode ioctl as follows:

```
struct linkoptformat {
        uint32_t   U_LINK_ID;
        unsigned short  newSUB_MODES;
        unsigned char   rd_wr;
};
```

The members of the linkoptformat structure are:

TABLE 7–26 linkoptformat fields

Member	Description
newSUB_MODE	This is the new SUB_MODES value in a write ioctl or the current value in a read ioctl.
U_LINK_ID	This identifies the particular link and must match one of the wlcfg database entries.
rd_wr	This determines read or write mode. A value of zero indicates read while non-zero indicates write.

In the case of read, the same structure is returned with the current value of SUB_MODES for the link.

7.11 N_linkread —Read the wlcfg Database

This ioctl is used to extract the wlcfg database for a link in a running system for examination. The wlcfg database is returned within the M_DATA part of the N_linkread ioctl. Make sure that there is enough space in the data area to receive the copy of the structure.

Refer to Section 7.8 “N_linkconfig—Configure the wlcfg Database” on page 87 for a complete list of the fields contained in the wlcfg database structure.

7.12 N_nuidel—Delete Specified NUI Mapping

This ioctl deletes the mapping for a specified Network User Identifier (NUI).

Note - This ioctl can disrupt other users significantly. For this reason it can only be used by root.

Associated Structure

The following structure is associated with this ioctl:

```
struct nui_del {
     char    prim_class;          /* Always NUI_MSG             */
     char    op;                  /* Always NUI_DEL             */
     struct  nuiformat  nuid;     /* NUI  to delete             */
 };
```

The members of the nui_del structure are:

TABLE 7–27 nui_del fields

Member	Description
prim_class	The value of this member is always NUI_MSG.
op	The value of this member is always NUI_DEL.
nuid	The Network User Identifier of the entry to be deleted

7.13 N_nuiget—Read the Mapping for a Specified NUI

This ioctl is used to read the mapping for a specified Network User Identifier (NUI).

Associated Structure

The following structure is associated with this ioctl:

```
struct nui_get {
        char            prim_class;     /* Always NUI_MSG              */
        char            op;             /* Always NUI_GET              */
        struct nuiformat        nuid;           /* NUI to get          */
        struct facformat        nuifacility;    /* NUI facilities      */
```

The members of the nui_get structure are:

TABLE 7–28 nui_get fields

Member	Description
prim_class	The value of this member is always NUI_MSG.
op	The value of this member is always NUI_DEL.
nuid	The Network User Identifier of the entry you want to read.
nuifacility	The NUI facilities associated with the entry you want to read.

7.14 N_nuimget—Read all Existing NUI Mappings

This ioctl is used to read all existing mappings for Network User Identifiers (NUI).

Associated Structure

The following structure is associated with this ioctl:

```
struct nui_mget {
   unsigned int first_ent;          /* First entry required    */
   unsigned int last_ent;           /* Last entry required     */
   unsigned int num_ent;            /* No of entries required  */
   char         buf[MGET_NBUFSIZE]; /* Data Buffer             */
 };
```

The members of the nui_mget structure are:

TABLE 7–29 Members of the nui_mget structure

Member	Description
buf	Contains the structure for the returned mapping entries.
first_ent	Informs the X.25 multiplexor where to start or restart the table read. It should initially be set to 0, to indicate starting at the beginning of the table.
num_ent	Indicates the number of mapping entries returned in the buf member.
last_ent	Set on return to point past the last entry returned (that is, a subsequent N_nuimget ioctl should have first_ent set to the value returned here).

7.15 N_nuiput—Store a set of NUIs

This ioctl is used to store a set of Network User Identifiers (NUIs) and associated facilities mappings within the X.25 multiplexor. It is used in conjunction with the NUI override facility option.

Note - This ioctl affects currently open connections and could therefore disrupt users significantly. For this reason it can only be used by root.

Associated Structure

The following structures are associated with this ioctl:

```
struct nui_put {
         char    prim_class;     /* Always NUI_MSG                       */
         char    op;             /* Always NUI_ENT                       */
         struct nuiformat       nuid;           /* NUI                  */
        struct facformat      nuifacility;    /* NUI facilities    */
 };
```

The members of the nui_put structure are:

TABLE 7–30 nui_put fields

Member	Description
prim_class	This is always set to NUI_MSG.
op	This is always set to NUI_ENT.
nuid	The Network User Identifier of the entry you want to store. This is stored in the nuiformat structure.
nuifacility	Any relevant NUI facilities. These are stored in the facformat structure.

The nuiformat structure looks like this:

```
#define NUIMAXSIZE 64
 #define NUIFACMAXSIZE 32
 struct nuiformat {
   unsigned char nui_len;
   unsigned char nui_string[NUIMAXSIZE]; /* Network User Identifier */
 };
```

The members of the nuiformat structure are

TABLE 7–31 nuiformat fields

Member	Description
nui_len	The length of the NUI.
nui_string	The NUI itself.

The facformat structure looks like this:

```
struct facformat {
   unsigned short SUB_MODES; /* Mode tuning bits for net */
```

```
  unsigned char  LOCDEFPKTSIZE; /* Local default pkt size */
  unsigned char  REMDEFPKTSIZE; /* Local default pkt size */
  unsigned char  LOCDEFWSIZE; /* Local default window size */
  unsigned char  REMDEFWSIZE; /* Local default window size */
  unsigned char  locdefthclass; /* Local default value         */
  unsigned char  remdefthclass; /* Remote default value        */
  unsigned char  CUG_CONTROL; /* CUG facilities */
};
```

The members of the `facformat` structure are:

TABLE 7–32 facformat fields

Member	Description
SUB_MODES	The subscription options for a PSDN link. Possible values and meanings are: SUB_EXTENDED Subscribe to extended call packets.This allows for packet and window size negotiation. BAR_EXTENDED Treat incoming extended call packets as a procedure error. SUB_FSELECT Subscribe to fast select with no restriction on response. This applies to INCOMING CALL packets. SUB_FSRRESP Subscribe to fast select with restriction on response. This applies to INCOMING CALL packets. SUB_REVCHARGE Subscribe to reverse charging. This applies to INCOMING CALL packets. SUB_LOC_CHG_PREV Subscribe to local charging prevention. This overrides the setting of SUB_REVCHARGE. SUB_TOA_NPI_FMT Subscribe to using TOA / NPI address format. BAR_TOA_NPI_FMT Treat incoming TOA / NPI address formats as a procedure error. SUB_NUI_OVERRIDE Subscribe to NUI override. This specifies that when an NUI is provided in a CALL REQUEST, any associated subscription time options override the facilities which apply to the interface, for the duration of that particular call. BAR_INCALL Bar incoming calls. BAR_OUTCALL Bar outgoing calls.
LOCDEFPKTSIZE	Local default packets size

TABLE 7–32 `facformat` fields *(continued)*

Member	Description
`REMDEFPKTSIZE`	Remote default packet size
`LOCDEFWSIZE`	Local default window size
`REMDEFWSIZE`	Remote default window size
`locdefthclass`	Local default value
`remdefthclass`	Remote default value
`CUG_CONTROL`	CUG facilities

7.16 `N_nuireset` —Delete all Existing NUI Mappings

This ioctl is used to delete all existing mappings for Network User Identifiers (NUIs).

Note - This ioctl can disrupt other users significantly. For this reason it can only be used by root.

Associated Structure

The following structure is associated with this ioctl:

```
struct nui_reset {
        char           prim_class;     /* Always NUI_MSG       */
        char           op;             /* Always NUI_RESET     */
 };
```

The members of the `nui_reset` structure are:

TABLE 7–33 `nui_reset` fields

Member	Description
`prim_class`	The value of this member is always `NUI_MSG`.
`op`	The value of this member is always `NUI_DEL`.

7.17 N_putpvcmap—Change PVC Packet and Window Sizes

This ioctl is used to change the packet and window sizes of a PVC from the defaults configured for the link that the PVC is active on.

Note - This ioctl can disrupt other users significantly. For this reason it can only be used by root.

Associated Structure

The following structure is associated with this ioctl:

```
struct pvcconff {
        uint32_t     link_id;      /* Link #           */
        unsigned short    lci;          /* Logical channel  */
        unsigned char     locpacket;    /* Loc packet size  */
        unsigned char     rempacket;    /* Rem packet size  */
        unsigned char     locwsize;     /* Loc window size  */
        unsigned char     remwsize;     /* Rem window size  */
};
```

The members of the `pvcconff` structure are:

TABLE 7–34 `pvcconf` fields

Member	Description
`link_id`	The identifier of the PVC you want to change
`lci`	The logical channel identifier.
`locpacket`	The local packet size to use.
`rempacket`	The remote packet size to use
`locwsize`	The local window size.
`remwsize`	The remote window size.

7.18 `N_traceoff` ioctl—Cancel `N_traceon`

This ioctl is used to cancel a previously issued `N_traceon` ioctl.

Note - This ioctl affects currently open connections and could therefore disrupt users significantly. For this reason it can only be used by root.

7.19 `N_traceon` —Turn on Packet Level Tracing

This ioctl turns on packet level tracing for a particular link or all configured links. Each incoming and outgoing X.25 packet will be sent up the stream on which the `N_traceon` ioctl was made.

Note - This ioctl can have a serious impact on security. For this reason it can only be used by root.

Associated Structures

The following structures are associated with this ioctl:

```
struct trc_regioc {
    uint8     all_links;              /* Trace on all links          */
    uint8     spare[3];               /* for alignment               */
    uint32    linkid;                 /* Link                        */
    uint8     level;                  /* Level of tracing required   */
    uint8     spare2[3];              /* for alignment               */
    uint32    active[MAX_LINKS+1];    /* tracing actively on         */
};
```

The members of the `trc_regioc` structure are:

TABLE 7–35 `trc_regioc` fields

Member	**value**
`all_links`	Returns the `linkids` of all links for which tracing was activated in the `active` array.
`linkid`	Specify tracing for a particular link.
`level`	The level of tracing required.
`active`	Indicates that tracing is currently active.

Each X.25 packet is preceded by a `trc_ctl` structure:

```
/*
    Types of tracing message
*/
#define TR_CTL       100          /* Basic                   */
#define TR_LLC2_DAT  101          /* Basic + LLC2 parameters */
#define TR_LAPB_DAT  TR_CTL       /* Basic for now           */
#define TR_MLP_DAT   TR_CTL       /* Basic for now           */
#define TR_X25_DAT   TR_CTL       /* Basic for now           */
#define TR_DLPI      102          /* type used for tracing DLPI primitives */

/*
    Format for control part of trace messages
*/
struct trc_ctl {
    uint8          trc_prim;    /* Trace msg identifier     */
    uint8          trc_mid;     /* Id of protocol module    */
    uint16         trc_spare;   /* for alignment            */
    uint32         trc_linkid;  /* Link Id                  */
    uint8          trc_rcv;     /* Message tx or rx         */
    uint8          trc_spare2[3]; /* for alignment          */
    uint32         trc_time;    /* Time stamp               */
    uint16         trc_seq;     /* Message seq number       */
```

```
};
```

TABLE 7–36 `trc_ctrl` fields

Member	Description
`trc_prim`	Always set to `TR_X25_DAT`.
`trc_mid`	Always set to the module ID of the X.25 multiplexor (200).
trc_linkid	The link identifier
trc_rcv	Message receive or rx
trc_time	Time stamp
trc_seq	Message seq number

7.20 N_X25_ADD_ROUTE—Set Fields of X25_ROUTE Structure

Sets the fields in the `X25_ROUTE` structure to the desired values.

Note - This ioctl can disrupt other users significantly. For this reason it can only be used by root.

Associated Structure

The `x25_route_s` data structure takes the following form:

```
typedef struct x25_route_s {
        uint32_t        index;          /* used for reading next route */
        u_char          r_type;
#define R_NONE          0
#define R_X121_HOST     1
#define R_X121_PREFIX   2
#define R_AEF_HOST      3
#define R_AEF_PREFIX    4
#define R_AEF_SOURCE    5
        CONN_ADR        x121;
```

```
        u_char          pid_len;
#define MAX_PID_LEN     4
        u_char          pid[MAX_PID_LEN];
        AEF             aef;
        int             linkid;
        X25_MACADDR     mac;
        int             use_count;
        char            pstn_number[16];
} X25_ROUTE;
```

Example

```
#include <sys/strupts.h>
 struct strioctl ioc ;
 int         fd ;
 X25_ROUTE      r;

 fd = open(‘‘/dev/x25’’, O,RDW);
  /*prepare route*/
           initialize

 io.ic_cmd = N_X25_ADD_ROUTE;
 io.ic_timeout = 0; /*system default : 15 secs */
 io.ic_len = sizeof(X25_ROUTE);
 io.ic_dp = (char *)&r;

 if (ioctl (fd, I_STR, &ioc) <0) {
                  perror(‘‘ xxxioctl’’);
      }
 }
```

7.21 N_X25_FLUSH_ROUTES—Flush all Routes

Flushes all routes out of X25_ROUTE structure.

Note - This ioctl can disrupt other users significantly. For this reason it can only be used by root.

Associated Structure

The x25_route_s data structure takes the following form:

```
typedef struct x25_route_s {
        uint32_t        index;          /* used for reading next route */
        u_char          r_type;
#define R_NONE          0
```

```
#define R_X121_HOST     1
#define R_X121_PREFIX   2
#define R_AEF_HOST      3
#define R_AEF_PREFIX    4
#define R_AEF_SOURCE    5
        CONN_ADR        x121;
        u_char          pid_len;
#define MAX_PID_LEN     4
        u_char          pid[MAX_PID_LEN];
        AEF             aef;
        int             linkid;
        X25_MACADDR     mac;
        int             use_count;
        char            pstn_number[16];
} X25_ROUTE;
```

Example

```
#include <sys/strupts.h>
 struct strioctl ioc ;
 int          fd ;
 X25_ROUTE      r;

 fd = open(‘‘/dev/x25’’, O,RDW);
  /*prepare route*/
           initialize

  io.ic_cmd = N_X25_FLUSH_ROUTES;
        io.ic_timout = 0;
        io.ic_len = 0;
        io.ic_dp = (char *)NULL;


        if (ioctl(x25s, I_STR, &ioc))
                perror("ioctl(X25_FLUSH_ROUTES)");
      }
 }
```

7.22 N_X25_GET_ROUTE—Obtain Routing Information

Obtains the routing information for a given destination address.

Associated Structure

The x25_route_s data structure takes the following form:

```
typedef struct x25_route_s {
        uint32_t        index;          /* used for reading next route */
```

```
        u_char          r_type;
#define R_NONE          0
#define R_X121_HOST     1
#define R_X121_PREFIX   2
#define R_AEF_HOST      3
#define R_AEF_PREFIX    4
#define R_AEF_SOURCE    5
        CONN_ADR        x121;
        u_char          pid_len;
#define MAX_PID_LEN     4
        u_char          pid[MAX_PID_LEN];
        AEF             aef;
        int             linkid;
        X25_MACADDR     mac;
        int             use_count;
        char            pstn_number[16];
} X25_ROUTE;
```

Example

```
#include <sys/strupts.h>
 struct strioctl ioc ;
 int          fd ;
 X25_ROUTE      r;

 fd = open(''/dev/x25'', O,RDW);
  /*prepare route*/
           initialize

 io.ic_cmd = N_X25_GET_ROUTE;
 io.ic_timeout = 0; /*system default : 15 secs */
 io.ic_len = sizeof(X25_ROUTE);
 io.ic_dp = (char *)&r;

 if (ioctl (fd, I_STR, &ioc) <0) {
                  perror('' xxxioctl'');
      }
 }
```

7.23 N_X25_GET_NEXT_ROUTE—Get Next Routing Entry

Obtains routing information for the next entry in the routing table. When there are no routes left, `error` will be -1, and `errno` will be set to `ENOENT`.

Associated Structure

The `x25_route_s` data structure takes the following form:

```
typedef struct x25_route_s {
        uint32_t        index;          /* used for reading next route */
        u_char          r_type;
#define R_NONE          0
#define R_X121_HOST     1
#define R_X121_PREFIX   2
#define R_AEF_HOST      3
#define R_AEF_PREFIX    4
#define R_AEF_SOURCE    5
        CONN_ADR        x121;
        u_char          pid_len;
#define MAX_PID_LEN     4
        u_char          pid[MAX_PID_LEN];
        AEF             aef;
        int             linkid;
        X25_MACADDR     mac;
        int             use_count;
        char            pstn_number[16];
} X25_ROUTE;
```

Example

```
#include <sys/strupts.h>
 struct strioctl ioc ;
 int         fd ;
 X25_ROUTE      r;

 fd = open(‘‘/dev/x25’’, O,RDW);
  /*prepare route*/
          initialize

 io.ic_cmd = N_X25_GET_NEXT_ROUTE;
 io.ic_timeout = 0; /*system default : 15 secs */
 io.ic_len = sizeof(X25_ROUTE);
 io.ic_dp = (char *)&r;

 if (ioctl (fd, I_STR, &ioc) <0) {
                  perror(‘‘ xxxioctl’’);
      }
 }
```

7.24 N_X25_RM_ROUTE—Remove Route From X25_ROUTE

Removes the route for a given destination address.

Note - This ioctl can disrupt other users significantly. For this reason it can only be used by root.

Associated Structure

The `x25_route_s` data structure takes the following form:

```
typedef struct x25_route_s {
        uint32_t        index;          /* used for reading next route */
        u_char          r_type;
#define R_NONE          0
#define R_X121_HOST     1
#define R_X121_PREFIX   2
#define R_AEF_HOST      3
#define R_AEF_PREFIX    4
#define R_AEF_SOURCE    5
        CONN_ADR        x121;
        u_char          pid_len;
#define MAX_PID_LEN     4
        u_char          pid[MAX_PID_LEN];
        AEF             aef;
        int             linkid;
        X25_MACADDR     mac;
        int             use_count;
        char            pstn_number[16];
} X25_ROUTE;
```

Example

```
#include <sys/strupts.h>
 struct strioctl ioc ;
 int          fd ;
 X25_ROUTE      r;

 fd = open(‘‘/dev/x25’’, O,RDW);
  /*prepare route*/
           initialize

 io.ic_cmd = N_X25_RM_ROUTE;
 io.ic_timeout = 0; /*system default : 15 secs */
 io.ic_len = sizeof(X25_ROUTE);
 io.ic_dp = (char *)&r;

 if (ioctl (fd, I_STR, &ioc) <0) {
                  perror(‘‘ xxxioctl’’);
      }
 }
```

7.25 `N_zerostats`—Reset X.25 Multiplexor Statistics Count

This ioctl is used to reset the statistics counts for the X.25 multiplexor.

Note - This ioctl affects currently open connections and could therefore disrupt users significantly. For this reason it can only be used by root.

CHAPTER **8**

Support Functions

Solstice X.25 provides a library of functions that can be used in applications.

Many of the functions make use of the `padent` and `xhostent` structures, they are therefore described first. A description of the functions follows, in alphabetical order. The tables below group the functions together according to function. The PAD and `xhosts` related functions are based on similar functions that are available with IP.

8.1 Linking to the Support Library

The support library for use on 32–bit systems resides in `/opt/SUNWconn/x25/lib/libsx25.so`. To link against it, use a command like this:

```
hostname% cc -o test test.c -L /opt/SUNWconn/x25/lib -R /opt/SUNWconn/x25/lib -lsx25
```

The support library for use on 64–bit systems resides in `/opt/SUNWconn/x25/lib/sparcv9/libsx25.so`. To link against it, use a command like this:

```
hostname% cc -o test test.c -L /opt/SUNWconn/x25/lib -R /opt/SUNWconn/x25/lib -lsx25
```

8.2 Function Summary

The header files used by the NLI support functions are contained in the `/usr/include/netx25` directory.

These functions are related to the PAD Hosts Database:

TABLE 8–1 PAD related functions

function	description
`endpadent`	closes the PAD Hosts Database
`getpadbyaddr`	finds the PAD Hosts Database entry for a given address
`getpadent`	reads the next line in the PAD Hosts Database
`padtos`	converts a network PAD Hosts Database structure into a string
`setpadent`	opens and rewinds the PAD Hosts Database

These entries are related to the `xhosts` file:

TABLE 8–2 `xhosts` functions

function	description
`endxhostent`	closes the `xhosts` file
`getxhostbyaddr`	finds an entry in the `xhosts` file by address
`getxhostbyname`	finds an entry in the `xhosts` by name
`getxhostent`	reads the next line of the `xhosts` file
`setxhostent`	opens and rewinds the `xhosts` file

These functions are related to X.25 addressing:

TABLE 8–3 X.25 addressing functions

function	description
equalx25	compares two X.25 addresses
stox25	converts an X.25 dot format address to an X.25 xaddrf structure
x25tos	converts an X.25 xaddrf structure to an X.25 dot format address

These functions are related to configuration files:

TABLE 8–4 Configuration file functions

function	description
x25_find_link_parameters	finds link configuration files and builds a linked list of links.
x25_read_config_parameters	reads a configuration files into a data structure by link number
x25_read_config_parameters_file	reads a configuration file into a data structure by filename
x25_save_link_parameters	updates the configuration files
x25_write_config_parameters	writes a data structure into a configuration file identified by a link number
x25_write_config_parameters_file	writes a data structure into a configuration file identified by a filename
x25_set_parse_error_function	installs a function as the default error handler.

These functions are related to links:

TABLE 8–5 Link functions

function	description
getnettype	returns the type of network configured for a link
linkidtox25	converts a character format link identifier to numeric format
x25tolinkid	converts a numeric link identifier to a string

8.3 The padent Structure

The padent structure is defined in the /usr/include/netx25/xnetdb.h file. It has this format:

```
struct    padent {
           struct xaddrf       xaddr;
           unsigned char       x29;
           struct extraformat  xtras;
           unsigned char       cud[MAxnetdb.hXCUDFSIZE + 1];
      };
```

The padent structure contains a single entry from the /etc/SUNWconn/x25/padmapconf file. This contains information about facilities and so on to be used when making PAD calls to a particular address.

The members of the padent structure are:

TABLE 8–6 Members of padent structure

Member	Description
xaddr	The hosts X.25 address.
x29	The X.29 version specifier. Possible values are: 0—use the configured default X.29 address 1—use X.29(80) yellow book 2—use X.29(84) red book 3—use X.29(88) blue book

TABLE 8–6 Members of padent structure *(continued)*

Member	Description
xtras	Any facilities and QOS parameters defined for this entry
cud	Any Call User Data defined for this entry.

8.4 The xhostent Structure

The xhostent structure is defined in the /usr/include/netx25/xnetdb.h file. It has this format:

```
struct   xhostent {
          char *h_name;
          char **h_aliases;
          int  h_addrtype;
          int  h_length;
          char *h_addr;
     };
```

The xhostent structure contains a single entry from the xhosts file. This contains information mapping host names to X.25 addresses and is used when making PAD calls. By default this file is in the /etc/SUNWconn/x25 directory.

The members of the structure are:

TABLE 8–7 Members of xhostent structure

Member	Description
h_name	A pointer to the name of the X.25 host, as defined in the xhosts file.
h_aliases	A pointer to an array of character pointers that point to aliases for the X.25 host.
h_addrtype	The type of address being returned. This is always CCITT_X25.

TABLE 8–7 Members of xhostent structure *(continued)*

Member	Description
h_length	The length in bytes of the structure that contains the X.25 address.
h_addr	A pointer to an xaddrf structure containing the network address of the X.25 host.

8.5 endpadent—Closes the PAD Hosts Database

Synopsis

```
#include <netx25/x25_proto.h>
 #include <netx25/xnetdb.h>

    endpadent()
```

Description

endpadent closes the PAD Hosts Database.

Arguments

endpadent does not take any parameters.

8.6 endxhostent—Closes the xhosts File

Synopsis

```
#include <netx25/x25_proto.h>
 #include <netx25/xnetdb.h>

    endxhostent()
```

Description

`endxhostent` closes the `xhosts` file. By default, this file is located in the `/etc/SUNWconn/x25/` directory.

Arguments

There are no parameters.

Return Value

This function has no return values.

8.7 `equalx25`—Compares two X.25 `addresses`

Synopsis

```
#include <netx25/x25_proto.h>
 #include <netx25/x25db.h>

    int equalx25 (
      struct xaddrf *x1,
      struct xaddrf *x2
  );
```

Description

Compares two X.25 addresses by checking to see whether the two `xaddrf` structures holding them are the same.

Arguments

The members of the structure are:

TABLE 8–8 Members of `xaddrf` structure

Member	Description
`x1`	A pointer to the structure containing the first X.25 address for checking.
`x2`	A pointer to the structure containing the second X.25 address for checking.

Return Values

Returns 1 if the two structures are the same, and 0 if they are not.

8.8 `getnettype`—Get Type of Network for a Link

Synopsis

```
#include <netx25/x25_proto.h>
 #include <netx25/xnetdb.h>

    int getnettype (
      unsigned char *linkid
  );
```

Description

Determines the type of network referred to by a particular link identifier.

Arguments

The parameters are:

TABLE 8–9 `getnettype` parameters

Parameter	Description
`linkid`	A pointer to the link identifier.

Return Values

A negative value indicates an invalid link identifier. The possible network types are:

TABLE 8–10 Network Type

return value	network type
LAN	local area network
W80	wide area network conforming to 1980 X.25
W84	wide area network conforming to 1984 X.25
W88	wide area network conforming to 1988 X.25
MLP*n*	A multi-link connection with *n* links.

8.9 `getpadbyaddr`—Get PAD Database Entry for Address

Synopsis

```
#include <netx25/x25_proto.h>
 #include <netx25/xnetdb.h>

    struct padent * getpadbyaddr(
       char *addr
    );
```

Description

`getpadbyaddr` returns a pointer to the `padent` structure, containing the entry from the PAD Hosts Database for the specified address. See Section 8.3 “The `padent` Structure” on page 122 for a description of the `padent` structure.

Arguments

The parameters are:

TABLE 8–11 `getpadbyaddr` parameters

Parameter	Description
`addr`	A pointer to a structure containing the address of the host whose database entry you want.

Return Value

`getpadbyaddr` returns a pointer to static storage. You must copy the value in order to keep and reuse it. A return value of 0 indicates that no match was found.

8.10 `getpadent`—Get Next Line in PAD Hosts Database

Synopsis

```
#include <netx25/x25_proto.h>
 #include <netx25/xnetdb.h>

    struct padent *getpadent( )
```

Description

The `getpadent` subroutine returns a pointer to a `padent` structure, which contains the next entry from the PAD Hosts Database. If necessary `getpadent` opens the file.

Arguments

There are no parameters.

Return Value

A return value of 0 indicates an error.

8.11 getxhostbyaddr—Get X.25 Host Name by Address

Synopsis

```
#include <netx25/x25_proto.h>
 #include <netx25/xnetdb.h>

    struct xhostent *getxhostbyaddr(
      char *addr,
      int len, int type
  );
```

Description

getxhostbyaddr searches the xhosts file for an entry with a matching X.25 host address. By default, this file is located in the /etc/SUNWconn/x25 directory. It returns a pointer to a xhostent structure containing information on the entry.

Arguments

The parameters are:

TABLE 8–12 getxhostbyaddr parameters

Parameter	Description
addr	A pointer to an xaddrf structure containing the address of the host whose entry you want.
len	The length in bytes of addr.
type	The address type required. This is always CCITT_X25.

Return Value

getxhostbyaddr returns a pointer to static storage. You must copy the value in order to keep and reuse it. A return value of 0 indicates the address supplied is either invalid or unknown.

8.12 getxhostbyname—Get X.25 Address by Name

Synopsis

```
#include <netx25/x25_proto.h>
 #include <netx25/xnetdb.h>

    struct xhostent *getxhostbyname(
      char *name
  );
```

Description

getxhostbyname searches the xhosts file for an entry with a matching host name.By default, this file is located in the /etc/SUNWconn/x25 directory. It returns a pointer to a xhostent structure containing information on the entry.

Arguments

The parameters are:

TABLE 8–13 getxhostbyname parameters

Parameter	Description
name	A pointer to the address of a string containing the name of the host whose entry you want.

Return Value

A pointer to the xhostent structure. A return value of 0 indicates the name supplied is either invalid or unknown.

8.13 getxhostent—Reads Next Line of xhosts File

Synopsis

```
#include <netx25/x25_proto.h>
 #include <netx25/xnetdb.h>

    struct xhostent *getxhostent(
       );
```

Description

getxhostent reads the next line of the /etc/SUNWconn/x25/hosts file. It opens the file if necessary.

Arguments

There are no parameters.

Return Value

A pointer to an xhostent structure. A return value of 0 indicates an error.

8.14 linkidtox25—Convert Link Identifier to Numeric Form

Synopsis

```
#include <netx25/x25_proto.h>
 #include <netx25/xnetdb.h>

    uint32_t linkidtox25(linkid)
unsigned char *linkid;
```

Description

Converts a character string link identifier to the numeric format used in X.25 primitives, with a range of 0 - 254.

Arguments

The parameters are:

TABLE 8–14 `linkidtox25` parameters

Parameter	Description
`str_linkid`	A pointer to the string containing the character format link id.

Return Values

On success 0 is returned. On failure the value of `MAX_LINKID` is returned. By default, this is 255.

8.15 `padtos`—Convert PAD Database Structure Into String

Synopsis

```
#include <netx25/x25_proto.h>
 #include <netx25/xnetdb.h>

  int padtos (
     struct padent *p,
     unsigned char *strp
  );
```

Description

Converts a PAD structure into a string containing all the facilities, CUGs, RPOAs and call user data defined in the PAD structure. The validity of the structure is checked before conversion.

Arguments

The parameters are:

TABLE 8–15 `padtos` parameters

Parameter	Description
`p`	A pointer to the `padent` structure for conversion.
`strp`	A pointer to the character string that will hold the result.

The character string pointed to by `strp` takes this format:

```
~CUD facilities  year CUG RPOA
```

All of the values are optional:

TABLE 8–16 `strp` character string values

Value	Description
`CUD`	Call User Data. This is always preceded by a tilde (~).
`Facilities`	Holds the values for packet size, window size, fast select and reverse charging.
`Year`	Possible values are 80, 84 and 88. These correspond to the X.29(80) Yellow Book, X.29(84) Red Book and X.29(88) Blue Book.
`CUG`	Specifies any call user groups that apply to this call. Preceded by g, G, b or B. b and B signify bilateral CUGs.
`RPOA`	Signifies any Recognized Private Operating Agency. Always preceded by T or t.

For example this string:

```
~hello p7/9w4/2fr 80 B1234 T5678
```

has the following meaning:

The CUD is `hello`. There is a local-to-remote packet size of `7` a remote-to-local packet size of `9`, a local-to-remote window size of `4` a remote -to-local window size of `2`. Fast select and reverse charging are set. The X.29(80) Yellow Book recommendation is being used. The bilateral CUG is 1234 and the RPOA is 5678.

Return Values

On success this function returns 0. A negative return value indicates that the pad structure was invalid.

8.16 `setpadent`—Open and Rewind the PAD Hosts Database

Synopsis

```
#include <netx25/x25_proto.h>
 #include <netx25/xnetdb.h>

    void setpadent(
       int stayopen
    );
```

Description

`setpadent` opens and rewinds the PAD Hosts Database.

Arguments

The parameters are:

TABLE 8–17 `setpadent` parameters

Parameter	Description
`stayopen`	If this is set to 0, the PAD Hosts Database is closed after each `getpadent` call. Otherwise, the PAD Hosts Database is not closed.

8.17 setxhostent—Open and Rewind the xhosts File

Synopsis

```
#include <netx25/x25_proto.h>
 #include <netx25/xnetdb.h>

    void setxhostent(
      int stayopen
  );
```

Description

setxhostent opens the xhosts file and rewinds it. By default, this file is located in the /etc/SUNWconn/x25 directory.

Arguments

The parameters are:

TABLE 8–18 setxhostent parameters

Parameter	Description
stayopen	Determines whether the file is closed once it has been rewound. 0 indicates the file is to be closed.

8.18 stox25—Convert X.25 Address to xaddrf Structure

Synopsis

```
#include <netx25/x25_proto.h>
 #include <netx25/x25db.h>

  int stox25 (
    unsigned char *cp,       /*  X.25 dot format address  */
```

```
    struct xaddrf *xad,      /*  The returned structure  */
    int lookup
  );
```

Description

Converts an X.25 format address into an `xaddrf` structure. Can also be used as a validity check for X.25 addresses.

Arguments

The parameters are:

TABLE 8–19 `stox25` parameters

Parameter	**Description**
`cp`	Points to a character string containing the X.25 address for conversion.
`xad`	Points to the `xaddrf` structure containing the X.25 dot format address.
`lookup`	Determines the level of address checking carried out before the address is converted. 0 indicates no address checking is carried out. This allows for faster conversion, but means the address may not be valid for the type of network it is used on. A non-zero value means that the address format is checked with the configuration file for the link.

Return Values

0 indicates successful completion. A negative return value indicates that the X.25 address was invalid.

8.19 x25_find_link_parameters—Finds Link Configuration Files and Builds a Linked List of Links

Synopsis

```
#include <netx25/x25_proto.h>
 #include <netx25/x25db.h>
 #include <netx25/config_functions.h>

    int x25_find_link_parameters (
      struct link_data ** lptr
  );
```

Description

This function scans the directory containing the X.25 configuration files and builds a linked list of data structures.

Arguments

The members of the structure are:

TABLE 8–20 Members of `link_data` structure

Member	Description
`lptr`	Points to the address of a pointer to a `link_data` structure. Memory for these structures is dynamically allocated using `calloc()`.

Return Values

Returns 0 on success.

8.20 x25_read_config_parameters— Reads a Configuration File Into a Data Structure

Synopsis

```
#include <netx25/x25_proto.h>
 #include <netx25/xnetdb.h>
 #include <netx25/config_functions.h>

    int x25_read_config_parameters (
                         int linkid
                         struct config_ident     *ipt,
                         struct LINK_config_data *lpt,
                         struct X25_config_data  *xpt,
                         struct MLP_config_data  *mpt,
                         struct LAPB_config_data *lbp,
                         struct LLC2_config_data *l2p,
                         struct WAN_config_data  *wpt,
                         int *flags
     );
```

Description

`x25_read_config_parameters` reads the configuration file for the specified link into a data structure.

Arguments

The parameters are:

TABLE 8–21 `read_confing_parameters` parameters

Parameter	Description
`linkid`	The identifier of the link concerned.
`ipt`	A pointer to the `config_ident` structure containing the link identifier. Setting this variable is mandatory.
`lpt`	A pointer to the `link_item` structure containing link information. Setting this variable is mandatory.
`xpt`	A pointer to the `wlcfg` structure containing the layer 3 (X.25) parameters. If you set this variable to NULL, information on these parameters is omitted.

TABLE 8–21 `read_confing_parameters` parameters *(continued)*

Parameter	Description
`mpt`	A pointer to the `mlp_item` structure containing the MLP parameters. If you set this variable to NULL, information on these parameters is omitted. As the number of devices required by an MLP link is unknown, this routine allocates memory as required using calloc().
`lbp`	A pointer to the `lliun_t` structure containing the layer 2 LAPB parameters. If you set this variable to NULL, information on these parameters is omitted.
`l2p`	A pointer to the `lliun_t` structure containing the LLC2 parameters. If you set this variable to NULL, information on these parameters is omitted.
`wpt`	A pointer to the `wan_tnioc` structure containing the layer 1 (physical) parameters. If you set this variable to NULL, information on these parameters is omitted.
`flag`	Indicates whether data is being read for LLC2, LAPB or MLP.

Return Value

A return value of 0 indicates success.

8.21 `x25_read_config_parameters_file`— Reads a Configuration File Into a Data Structure

Synopsis

```
#include <netx25/x25_proto.h>
 #include <netx25/xnetdb.h>
 #include <netx25/config_functions.h>

    int X25_read_config_parameters_file (
                         char filename
                         struct config_ident     *ipt,
                         struct LINK_config_data *lpt,
                         struct X25_config_data  *xpt,
                         struct MLP_config_data  *mpt,
```

```
                                struct LAPB_config_data *lbp,
                                struct LLC2_config_data *l2p,
                                struct WAN_config_data  *wpt,
                                int *flag
);
```

Use

`x25_read_config_parameters` reads the specified configuration file into a data structure.

Description

The parameters are:

TABLE 8–22 `x25_read_config_parameters_file` parameters

Parameter	Description
`filename`	The name of the file concerned.
`ipt`	A pointer to the `config_ident` structure containing the link identifier. Setting this variable is mandatory.
`lpt`	A pointer to the `link_item` structure containing link information. Setting this variable is mandatory.
`xpt`	A pointer to the `wlcfg` structure containing the layer 3 (X.25) parameters. If you set this variable to NULL, information on these parameters is omitted.
`mpt`	A pointer to the `mlp_item` structure containing the MLP parameters. If you set this variable to NULL, information on these parameters is omitted. As the number of devices required by an MLP link is unknown, this routine allocates memory as required using calloc().
`lbp`	A pointer to the `lliun_t` structure containing the layer 2 LAPB parameters. If you set this variable to NULL, information on these parameters is omitted.
`l2p`	A pointer to the `lliun_t` structure containing the LLC2 parameters. If you set this variable to NULL, information on these parameters is omitted.

TABLE 8–22 `x25_read_config_parameters_file` parameters *(continued)*

Parameter	Description
`wpt`	A pointer to the `wan_tnioc` structure containing the layer 1 (physical) parameters. If you set this variable to NULL, information on these parameters is omitted.
`flag`	Indicates whether data is being read for LLC2, LAPB or MLP.

Return Value

A return value of 0 indicates success.

8.22 `x25_save_link_parameters`— Update Configuration Files

Synopsis

```
#include <netx25/x25_proto.h>
 #include <netx25/x25db.h>
 #include <netx25/config_functions.h>

    int x25_save_link_parameters (
      struct link_data * linkid
  );
```

Description

This function takes the information in the `LINK_config_data` structures and updates the X.25 configuration files as necessary.

Arguments

The parameters are:

TABLE 8–23 x25_save_link_parameters parameters

Parameter	Description
linkid	A pointer to the address of a link_data structure, which is the first in a linked list.

Return Values

Returns 0 on success.

8.23 x25_set_parse_error_function— Install a Function as Default Error Handler

Synopsis

```
#include <netx25/config_functions.h>

   int (*x25_set_parse_error_function(int (*func)(char *)))(char *)
```

Description

By default, errors are handled by printing a message to stderr and continuing. The x25_set_parse_error_function function allows a different function to be installed for use, for example with windowing programs.

Arguments

The parameters are:

TABLE 8–24 x25_set_parse_error_function parameter

Parameter	Description
func	A pointer to a function which is installed as the default error handler. This function will be called with a single argument, a pointer to the error string. If this is set to NULL, the default action if restored.

Return Values

The address of the previous error function is returned.

8.24 `x25_write_config_parameters`—Writes a Data Structure Into a Configuration File Identified by a Link Number

Synopsis

```
#include <netx25/x25_proto.h>
 #include <netx25/xnetdb.h>
 #include <netx25/config_functions.h>

   int x25_write_config_parameters (struct config_ident *idptr,
                       struct LINK_config_data *ptr,
                       struct X25_config_data  *xptr,
                       struct MLP_config_data  *mptr,
                       struct LAPB_config_data *lbptr,
                       struct LLC2_config_data *l2ptr,
                       struct WAN_config_data  *wptr);
```

Description

`x25_write_config_parameters` writes the specified data structure(s) into a configuration file identified by the number of the link it configures.

Arguments

The parameters are:

TABLE 8–25 x25_write_config_parameters parameters

Parameters	Description
idptr	A pointer to the config_ident structure containing the link identifier. Setting this variable is mandatory.
ptr	A pointer to the link_item structure containing link information. Setting this variable is mandatory.
xptr	A pointer to the wlcfg structure containing the layer 3 (X.25) parameters. This parameter is mandatory.
mptr	A pointer to the mlp_item structure containing the MLP parameters. If you set this variable to NULL, information on these parameters is omitted.
lbptr	A pointer to the lliun_t structure containing the layer 2 LAPB parameters. If you set this variable to NULL, information on these parameters is omitted.
l2ptr	A pointer to the lliun_t structure containing the LLC2 parameters. If you set this variable to NULL, information on these parameters is omitted.
wptr	A pointer to the wan_tnioc structure containing the layer 1 (physical) parameters. If you set this variable to NULL, information on these parameters is omitted.

Return Value

A return value of 0 indicates success.

8.25 x25_write_config_parameters_file—Writes a Data Structure Into a Configuration File Identified by a Filename

Synopsis

```
#include <netx25/x25_proto.h>
 #include <netx25/xnetdb.h>
 #include <netx25/config_functions.h>

    int x25_write_config_parameter_file (char *infname,
                       struct config_ident *idptr,
                       struct LINK_config_data *ptr,
                       struct X25_config_data  *xptr,
                       struct MLP_config_data  *mptr,
                       struct LAPB_config_data *lbptr,
                       struct LLC2_config_data *l2ptr,
                       struct WAN_config_data  *wptr);
```

Use

`x25_write_config_parameters_file` writes the specified data structure(s) into a configuration file identified by its filename.

Description

TABLE 8–26 `write_link_config_parameters_file` parameters

Parameter	**Description**
`filename`	The name of the file to contain the data structure.
`idptr`	A pointer to the `config_ident` structure containing the link identifier. Setting this variable is mandatory.
`ptr`	A pointer to the `link_item` structure containing link information. Setting this variable is mandatory.
`xptr`	A pointer to the `wlcfg` structure containing the layer 3 (X.25) parameters. This parameter is mandatory.

TABLE 8–26 `write_link_config_parameters_file` parameters *(continued)*

Parameter	Description
`mptr`	A pointer to the `mlp_item` structure containing the MLP parameters. If you set this variable to NULL, information on these parameters is omitted.
`lbptr`	A pointer to the `lliun_t` structure containing the layer 2 LAPB parameters. If you set this variable to NULL, information on these parameters is omitted.
`l2ptr`	A pointer to the `lliun_t` structure containing the LLC2 parameters. If you set this variable to NULL, information on these parameters is omitted.
`wptr`	A pointer to the `wan_tnioc` structure containing the layer 1 (physical) parameters. If you set this variable to NULL, information on these parameters is omitted.

Return Value

A return value of 0 indicates an error.

8.26 `x25tolinkid`—Convert Numeric Link Identifier to String

Synopsis

```
#include <netx25/x25_proto.h>
 #include <netx25/xnetdb.h>

    int x25tolinkid(linkid, str_linkid)
uint32_t linkid;
unsigned char *str_linkid;
```

Description

Converts a link identifier of the numeric format used in X.25 primitives to a character string.

Arguments

The parameters are:

TABLE 8–27 `x25tolinkid` parameters

Parameters	Description
`linkid`	The numeric format link identifier.
`str_linkid`	A pointer to the string that is to contain the character format link identifier

Return Values

On success 0 is returned. On failure -1 is returned.

8.27 `x25tos`—Convert `xaddrf` Structure to X.25 Address

Synopsis

```
#include <netx25/x25_proto.h>
 #include <netx25/x25db.h>

    int x25tos (
      struct xaddrf *xad,      /*  The X.25 structure   */
      unsigned char *cp,       /*  The returned string  */
      int lookup
  );
```

Description

Converts an `xaddrf` structure into an X.25 address. Before doing so, it checks the validity of the `xaddrf` structure.

Arguments

The parameters are:

TABLE 8–28 `x25tos` parameters

Parameters	Description
`xad`	Points to the `xaddrf` structure for conversion.
`cp`	Points to the string the X.25 address will be written to.
`lookup`	Determines the level of address checking carried out before the structure is converted. 0 indicates no checking is carried out. This allows for faster conversion, but means the structure may not be valid for the type of network it refers to. A non-zero value means that the structure is checked using the configuration files.

Return Values

0 indicates successful completion. A negative return value indicates that the structure was invalid.

CHAPTER 9

Error Codes

This chapter contains a summary of error codes returned by the NLI programming interface.

9.1 Originator and Reason Tables

The following tables list the OSI error codes defined in <netx25/x25_proto.h>.

To identify the *originator* in N_RI and N_DI messages:

NS_USER 1

NS_PROVIDER 2

To specify the *reason* when the originator is the Network Service provider in N_DI messages:

TABLE 9–1 Reason when Originator is NS Provider

Code	**Value**
NS_GENERIC	0xE0
NS_DTRANSIENT	0xE1
NS_DPERMANENT	0xE2
NS_TUNSPECIFIED	0xE3

TABLE 9–1 Reason when Originator is NS Provider *(continued)*

Code	Value
NS_PUNSPECIFIED	0xE4
NS_QOSNATRANSIENT	0xE5
NS_QOSNAPERMANENT	0xE6
NS_NSAPTUNREACHABLE	0xE7
NS_NSAPPUNREACHABLE	0xE8
NS_NSAPPUNKNOWN	0xEB

To specify the *reason* when the originator is the Network Service user in N_DI messages:

TABLE 9–2 Reason when Originator is NS User

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

TABLE 9–2 Reason when Originator is NS User *(continued)*

Code	Value
NU_INCOMPUSERDATA	0xF8
NU_BADPROTID	0xF9

To specify the *reason* when the originator is the Network Service provider in N_RI messages:

NS_RUNSPECIFIED 0xE9

NS_RCONGESTION 0xEA

To specify the *reason* when the originator is the Network Service user in N_RI messages:

NU_RESYNC 0xFA

Note - These codes are defined in ISO 8208 and are mapped from X.25 cause and diagnostic codes as described in ISO 8878.

9.2 Decoding Error Codes

You can decode the error codes listed in this chapter using the /opt/SUNWconn/x25/bin/x25diags utility. Enter x25diags followed by the hexadecimal value returned. For example:

```
hostname% x25diags E4
diag is 228 (decimal),  E4 (hexa) :

OSI Network service problem :
 Connection rejection -- reason unspecified (permanent condition)
```

PART II Data Link Protocol Interface (DLPI)

CHAPTER **10**

About DLPI

The Data Link Provider Interface (DLPI) is a standard defined by the Open Group. DLPI is defined by technical standard C614. Copies of this standard are available from the Open Group.

10.1 How DLPI Works

DLPI defines the format that STREAMS messages must take when interfacing to the datalink layer. The diagram below summarizes the way it works:

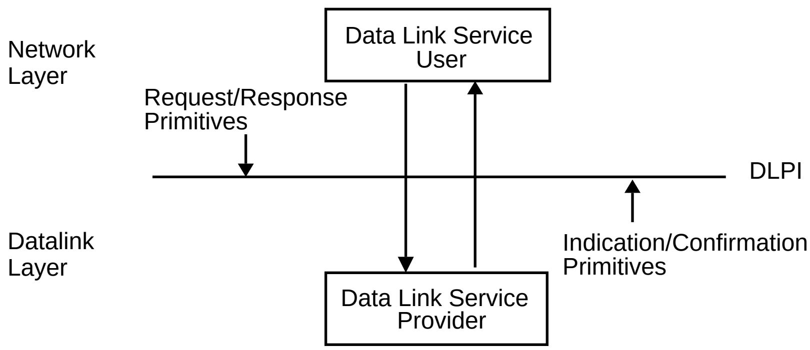


Figure 10–1 DLPI Summary

Like NLI, DLPI uses the `putmsg` and `getmsg` system calls and certain `ioctl` commands. See Chapter 11 for more information.

Note - The DLPI message primitives provided support LLC and LLC1 as well as LLC2. However, as LLC and LLC1 are not used by Solstice X.25, this is not documented here. Refer to *A STREAMS-based Data Link Provider Interface—Version 2* for information on working with LLC and LLC1.

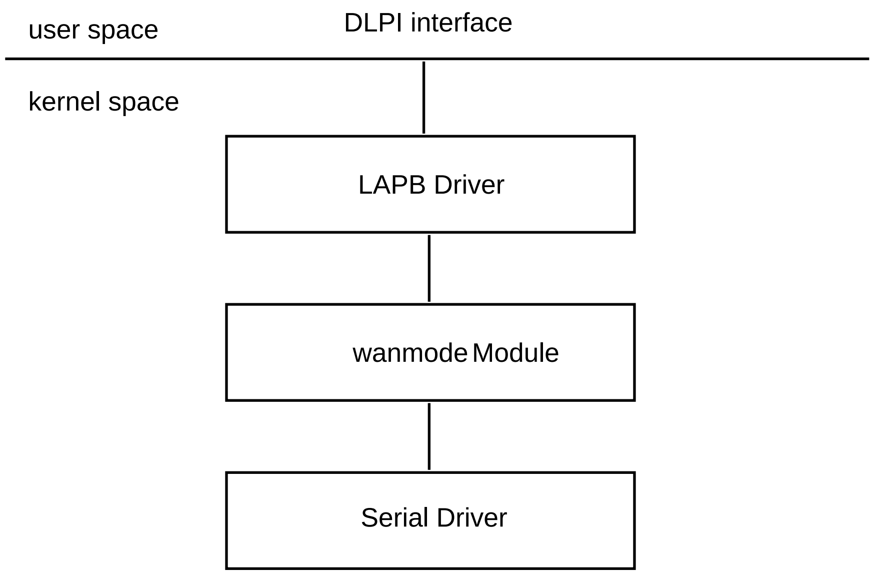
10.2 Addressing

A DLS User is identified by two pieces of information. The Physical Point of Attachment (PPA) defines the point at which the system is attached to a physical communications medium. The Datalink Service Access Point (DLSAP) identifies the service access point associated with a stream.

10.3 Running DLPI Over LAPB

You cannot use LAPB on a link that is already in use with X.25, so you need to build the stream architecture before you can run DLPI over the LAPB protocol.

The stream architecture for LAPB must be:

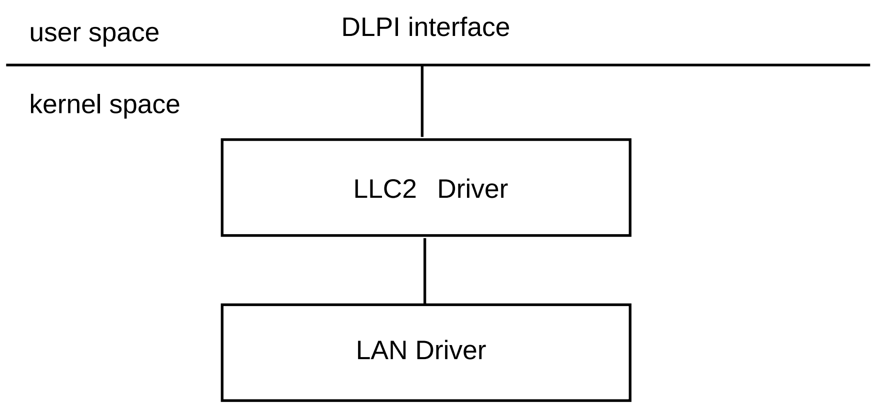


The serial driver manages the chip that controls the serial line. The `wanmod` module is installed between the serial driver and the LAPB driver. The `wanmod` driver controls the signals sent by the serial driver and informs the LAPB driver when the connection has been established at the physical level (i.e. when the cable is plugged in). The LAPB driver then implements the LAPB protocol and the DLPI interface.

See Section 11.1.2 “Message Primitive Sequence Summary” on page 163, for additional information.

10.4 Running DLPI Over LLC2

The stream architecture for LLC2 must be:



The previous version of the LLC2 driver did not carry out plumbing and PPA assignment, and applications that were designed to run directly over LLC2 were obliged to perform these steps themselves. The LLC2 driver supplied with Solstice X.25 9.2 now takes care of these tasks at system boot time, by means of `/etc/rc2.d/S40llc2`.

The relationship between the PPA and a particular LAN device is defined by the files in the directory `/etc/llc2/default`. Instead of choosing an arbitrary PPA and configuring LLC2 to use it, an application must find the PPA that is associated with the required LAN device and attach to it. Applications need now only do the following:

- determine the PPA associated with the required LAN device;
- open `/dev/llc2`;
- issue a DLPI attach request for the desired PPA.

Two routines have been added to the Solstice X.25 code to take account of the change in the LLC2 driver:

TABLE 10–1 Solstice X.25 routines to associate PPA with a LAN device

Routine	Description
`x25_device_instance()`	Finds the LAN device for a given X.25 link.
`x25_device_to_ppa()`	Finds the LLC2 PPA assigned to that LAN device.

See Section 11.1.2 “Message Primitive Sequence Summary” on page 163, for additional information.

CHAPTER **11**

DLPI Reference

The DLPI message primitives and Sun specific ioctls described in this chapter act as an interface to LAPB, or to both LAPB and LLC2. LAPB provides a connection-oriented service only, so the connectionless primitives cannot be used with LAPB. For information on ioctls exclusive to LLC2, refer to the `llc2` man page. LLC2 provides both connection-oriented and connectionless services. The connectionless service is generally referred to as LLC1 and is not a supported part of the Solstice X.25 product.

The primitives and ioctls are described in alphabetical order. Refer to the summary tables at the start of each section for functional groupings.

The header files used by the message primitives and ioctls described in this chapter are contained in the `/usr/include/netdlc`, `/usr/include/netx25` and `/usr/include/sys` directories.

All of the message primitives listed conform to the DLPI standard. They must be used with the getmsg(2) and putmsg(2) system calls. For more information, see the *STREAM's Programmer Guide.*

11.1 DLPI Specific Message Primitives

These message primitives are related to local management services:

TABLE 11–1 Local Management Service Message Primitives

name	summary
DL_INFO_REQ	requests information
DL_INFO_ACK	acknowledges a request for information
DL_ATTACH_REQ	identifies the physical link to attach to
DL_BIND_ACK	acknowledges a bind request
DL_DETACH_REQ	identifies the physical link to detach from
DL_BIND_REQ	specifies whether connectionless or connection oriented mode is to be use, and supplies the LSAP to bind to
DL_UNBIND_REQ	requests an unbind
DL_OK_ACK	positively acknowledges a previous primitive
DL_ERROR_ACK	negatively acknowledges a previous primitive

These message primitives are related to connection mode services:

TABLE 11–2 Connection Mode Service Message Primitives

name	summary
DL_CONNECT_REQ	establishes a connection
DL_CONNECT_IND	indicates that a remote user wants to establish a connection
DL_CONNECT_RES	accepts a connect request from a remote user
DL_CONNECT_CON	acknowledges a connect request
DL_TOKEN_REQ	determines the token associated with a stream (LLC2 only)
DL_TOKEN_ACK	acknowledges a token (LLC2 only)

These message primitives are related to connection release:

TABLE 11–3 Connection Release Message Primitives

name	summary
DL_DISCONNECT_REQ	disconnects a connection
DL_DISCONNECT_IND	informs that a connection has been disconnected or not established

The following non-DLPI message primitive is related to data transfer:

TABLE 11–4 Data Transfer Message Primitive

name	**summary**
M_DATA	carries data within a stream, and between a stream and a user process

These message primitives are related to connection resynchronization.

TABLE 11–5 Data Resynchronization Message Primitives

DL_RESET_REQ	resynchronizes a connection
DL_RESET_IND	indicates that the remote end is resynchronizing the connection
DL_RESET_RES	completes reset processing
DL_RESET_CON	confirms that reset processing is complete

11.1.1 Address Structures

DLPI uses data link service access point (DLSAP) addresses. These are used when connecting to a given address by the DL_CONNECT_REQ, DL_CONNECT_CON and DL_CONNECT_IND primitives. Addressing is handled differently for LLC2 and LAPB.

11.1.1.1 LLC2 Address Structure

The LLC2 DLSAP is contained in the following structure:

```
struct llc_dladdr {
    u_char   dl_mac[6];      /* MAC address */
    u_char   dl_sap;         /* LLC SAP */
 };
```

The file `/usr/include/netdlc/llc2.h` contains the structure definition.

The members of the `llc_dladdr` structure are:

TABLE 11–6 Members of `llc_dladdr` structure

Members	Description
`dl_mac`	The MAC address
`dl_sap`	The LLC SAP (service access point).

11.1.1.2 LAPB Address Structure

The address field is only required when LAPB is being used over a Public Switched Telephone Network (PSTN). In this case, the `dl_address` fields contain the PSTN address, in the format defined by the `pstnformat` structure:

```
struct pstnformat {
   uint8   pstn_len;               /* Address length in octets */
   uint8   pstn_add[20];  /* LAPB Address in hexadecimal */
 };
```

The members of the `pstnformat` structure are:

TABLE 11–7 Members of `pstnformat` structure

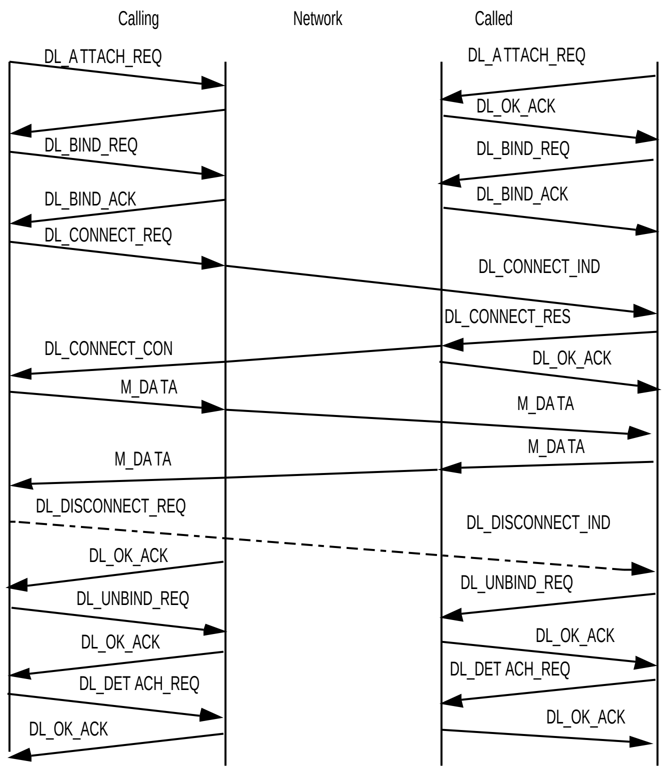
Members	Description
`pstn_len`	The length of the address as bytes.
`pstn_add`	The LSAP in hexadecimal format. This can be up to 20 digits long.

The file `/usr/include/netx25/sdlpi.h` contains the structure definition.

11.1.2 Message Primitive Sequence Summary

LLC2

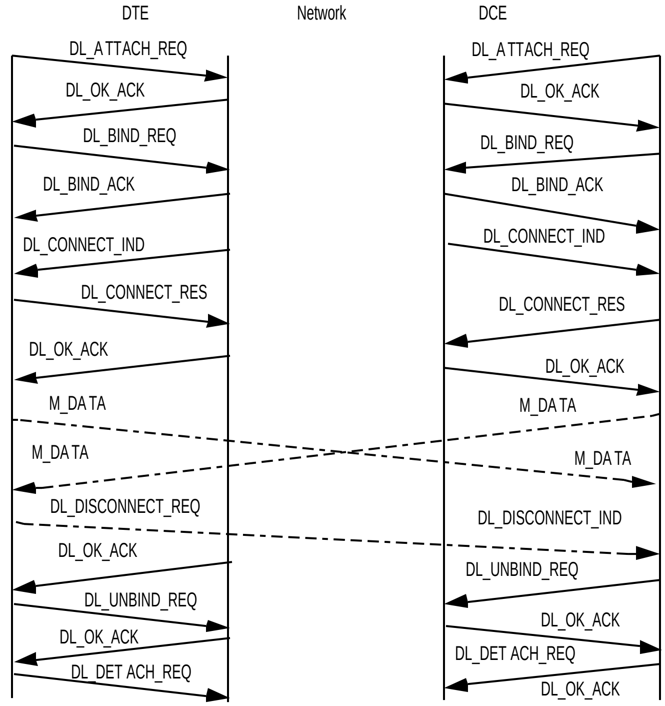
The diagram below summarizes the order that the DLPI message primitives are used to establish a connection and transfer data in connection oriented mode when LLC2 is used:



The file `/opt/SUNWconn/x25/samples.dlpi/llc2.c` contains an example.

LAPB

The diagram below summarizes the order that the DLPI message primitives are used to establish a connection and transfer data in connection oriented mode when LAPB is used:



When using LAPB, the connect phase is handled automatically by the network. The LAPB protocol automatically establishes the connection as soon as the cable has been plugged in. The connect indicator `CONNECT_IND` will automatically notify the user when the connection has been established.

The file `/opt/SUNWconn/x25/samples.dlpi/lapb.c` contains an example.

11.1.3 `DL_ATTACH_REQ`—Identifies Physical Link to use

This primitive is sent in an `M_PROTO` message block. It identifies the physical link to be used. In most cases this is a card or a port plus a card. The physical link is identified by a Physical Point of Attachment (PPA).

Associated Structure:

```
typedef struct {
        t_uscalar_t     dl_primitive;           /* set to DL_ATTACH_REQ */
        t_uscalar_t     dl_ppa;                 /* id of the PPA */
} dl_attach_req_t;
```

The members of the `dl_attach_req_t` structure are:

TABLE 11–8 Members of the dl_attach_req_t structure

Member	Description
dl_primitive	Should be set to the name of this primitive.
dl_ppa	Contains the PPA (or hardware device) the stream should be bound to. The PPA values are defined at system configuration time with the L_SETPPA ioctl. This applies to LAPB only. For LLC2 the PPA is associated with the hardware at system boot time.

Errors

TABLE 11–9 DL_ATTACH_REQ errors

Error	Description
DL_OUTSTATE	Primitive issued from an invalid state.
DL_BADPPA	The specified PPA was invalid (was not configured with the L_SETPPA ioctl).
DL_SYSERR	Could not allocate memory to handle the connection.

11.1.4 DL_BIND_ACK—Acknowledges Bind Request

If a bind request is successful, a DL_BIND_ACK message will be sent upstream to acknowledge the request. This message is sent in an M_PCPROTO message block.

Associated Structure

```
typedef struct {
        t_uscalar_t     dl_primitive;   /* DL_BIND_ACK */
        t_uscalar_t     dl_sap;         /* DLSAP addr info */
        t_uscalar_t     dl_addr_length; /* length of complete DLSAP addr */
        t_uscalar_t     dl_addr_offset; /* offset from start of M_PCPROTO */
        t_uscalar_t     dl_max_conind;  /* allowed max. # of con-ind */
        t_uscalar_t     dl_xidtest_flg; /* responses supported by provider */
} dl_bind_ack_t;
```

The members of the dl_bind_ack_t structure are:

TABLE 11–10 Members of the dl_bind_ack_t structure

Member	Description
dl_primitive	Should be set to the name of this primitive.
dl_sap	Contains the SSAP.
dl_addr_length	The length of the DLSAP. When interfacing to LLC2, this is 7. For LAPB it is 0.
dl_addr_offset	Offset of DLSAP address, in bytes, from the beginning of the M_PCPROTO message block. The DLSAP address is stored as an struct llc_dladdr structure when working with LLC2 and as a pstnformat structure when working with LAPB. See Section 11.1.1 "Address Structures" on page 161 for more information.
dl_max_conind	Equals the value of max_conind passed down in the DL_BIND_REQ message.
dl_xidtest_flg	Valid for LLC2 only. Contains the value (DL_AUTO_XID | DL_AUTO_TEST), because the LLC2 driver has the capability of responding automatically to TEST and XID commands.

11.1.5 DL_BIND_REQ—Specifies CLNS or CONS Service

This primitive is sent in an M_PROTO message block. It specifies whether the connectionless or connection oriented service should be used and provides the LSAP if required.

This message primitive must be set to connection oriented mode when used with either LAPB or LLC2. Connectionless mode is not supported.

The LSAP is one octet long and can be any value.

Associated Structure:

```
typedef struct {
        t_uscalar_t     dl_primitive;   /* set to DL_BIND_REQ */
        t_uscalar_t     dl_sap;         /* info to identify dlsap addr */
        t_uscalar_t     dl_max_conind;  /* max # of outstanding con_ind */
        uint16_t        dl_service_mode;        /* CO, CL or ACL */
        uint16_t        dl_conn_mgmt;   /* if non-zero, is con-mgmt stream */
```

```
        t_uscalar_t     dl_xidtest_flg; /* auto init. of test and xid */
} dl_bind_req_t;
```

The members of the `dl_bin_req_t` structure:

TABLE 11–11 Members of the `dl_bin_req_t` structure

Member	Description
`dl_primitive`	Should be set to the name of this primitive.
`dl_sap`	One byte SSAP. For LLC2, this parameter must be set to an even value other than 0. For LAPB, this parameter must be set to 0. The full DLSAP is returned in the DL_BIND_ACK response.
`dl_max_conind`	The maximum number of outstanding `DL_CONNECT_IND` messages allowed on the stream. A value of zero prevents the stream from accepting any `DL_CONNECT_IND` messages. When using LAPB, set this parameter to 1. When using with LLC2, set the calling side to 0 and the called side to >0.
`dl_service_mode`	Set to `DL_CODLS` to indicate that connection-oriented service (LLC2 or LAPB) is desired.
`dl_conn_mgmt`	Set to non-zero to use this stream as the “connection management” stream for the PPA. When set to a non-zero value, this handles incoming DL_CONNECT_IND messages that do not match any other stream, or where the maximum number of outstanding connection messages specified in `dl_max_conind` has been exceeded.
`dl_xidtest_flg`	Valid for LLC2 only. Specifies whether or not the LLC2 driver is to automatically reply to `XID`/`TEST` commands. It is a bit-mask of the following two flags: `DL_AUTO_XID`—Respond to XID commands. `DL_AUTO_TEST`—Respond to TEST commands.If this field is zero, the LLC2 client will receive all incoming TEST and XID commands, and will be expected to respond to them.

Note - Multiple LLC2 streams may be bound to the same SAP, but only one listen stream is allowed per SAP.

Errors

TABLE 11–12 DL_BIND_REQ errors

Error	Description
DL_OUTSTATE	Primitive issued from an invalid state.
DL_UNSUPPORTED	The requested service mode is not supported (only DL_CODLS and DL_CLDLS are supported).
DL_BOUND	Attempt to bind a second listen stream, or a second “connection management” stream.
DL_BADADDR	Attempt to bind to a zero or odd SAP.
DL_SYSERR	Could not allocate STREAMS resources.

11.1.6 DL_CONNECT_CON—Acknowledge DL_CONNECT_REQ

Positively acknowledges a previous DL_CONNECT_REQ primitive. Is sent upstream when a UA frame arrives to ack a previously sent SABME or SABM frame. This message consists of one M_PROTO message block containing the following structure:

```
typedef struct {
        t_uscalar_t     dl_primitive;           /* DL_CONNECT_CON */
        t_uscalar_t     dl_resp_addr_length;    /* responder’s address len */
        t_uscalar_t     dl_resp_addr_offset;    /* offset from start of block */
        t_uscalar_t     dl_qos_length;          /* length of qos structure */
        t_uscalar_t     dl_qos_offset;          /* offset from start of block */
        t_uscalar_t     dl_growth;              /* set to zero */
} dl_connect_con_t;
```

The members of the dl_connect_con_t structure are:

TABLE 11–13 Members of the `dl_connect_con_t` structure

Member	Description
`dl_primitive`	Should be set to the name of this primitive.
`dl_resp_addr_length`	The length of the DLSAP. This is 7 when working with LLC2 and 0 or 21 when working with LAPB.
`dl_resp_addr_offset`	Offset to the responder (destination) address, stored in struct `llc_dladdr` or struct `pstnformat` format.
`dl_qos_length`	Always set to 0.
`dl_qos_offset`	Always set to 0.

11.1.7 DL_CONNECT_IND—Indicate Incoming Connection

Indicates that a remote user wants to establish a connection. This primitive is sent upstream when a SABME or SABM is received from the network.

Associated Structure

This message consists of one M_PROTO message block containing the following structure:

```
typedef struct {
        t_uscalar_t     dl_primitive;           /* DL_CONNECT_IND */
        t_uscalar_t     dl_correlation;         /* provider’s correl. token */
        t_uscalar_t     dl_called_addr_length;  /* length of called address */
        t_uscalar_t     dl_called_addr_offset;  /* offset from start of block */
        t_uscalar_t     dl_calling_addr_length; /* length of calling address */
        t_uscalar_t     dl_calling_addr_offset; /* offset from start of block */
        t_uscalar_t     dl_qos_length;          /* length of qos structure */
        t_uscalar_t     dl_qos_offset;          /* offset from start of block */
        t_uscalar_t     dl_growth;              /* set to zero */
} dl_connect_ind_t;
```

The members of the `dl_connect_ind_t` structure are:

TABLE 11–14 Members of the dl_connect_ind_t structure

Member	Description
dl_primitive	Should be set to the name of this primitive.
dl_correlation	Unique identifier for the connection, to be passed back downstream in a DL_CONNECT_RES or DL_DISCONNECT_RES message later on. Can also be passed upstream in a subsequent DL_DISCONNECT_IND message.
dl_called_addr_length	Set to 7 when interfacing with LLC2 and 21 when interfacing with LAPB.
dl_called_addr_offset	Offset to the called (destination) address, which is stored in struct llc_dladdr or struct lapbformat format.
dl_calling_addr_length	Set to 7 when working with LLC2 and 0 when working with LAPB.
dl_calling_addr_offset	Offset to the calling (source) address, which is stored in struct llc_dladdr or struct pstnformat format.
dl_qos_length	Always set to 0.
dl_qos_offset	Always set to 0.

11.1.8 DL_CONNECT_REQ—Establish a Connection

Used to establish a connection. When the user issues this primitive, a SABME or SABM frame is sent across the network to the destination.

Associated Structure

This message consists of one M_PROTO message block containing the following structure:

```
typedef struct {
        t_uscalar_t     dl_primitive;           /* DL_CONNECT_REQ */
        t_uscalar_t     dl_dest_addr_length;    /* len. of dlsap addr */
        t_uscalar_t     dl_dest_addr_offset;    /* offset */
        t_uscalar_t     dl_qos_length;          /* len. of QOS parm val */
        t_uscalar_t     dl_qos_offset;          /* offset */
        t_uscalar_t     dl_growth;              /* set to zero */
```

```
} dl_connect_req_t;
```

The members of the dl_connect_req_t structure are:

TABLE 11–15 Members of the dl_connect_req_t structure

Member	Description
dl_primitive	Should be set to the name of this primitive.
dl_dest_addr_length	Set to 7 when working with LLC2 and 21 when working with LAPB.
dl_dest_addr_offset	Offset, in bytes, from beginning of M_PROTO message block. The destination DLSAP address should be encoded as a struct llc_dladdr. This field and dl_dest_addr_length combined give the remote address if you are working with LLC2 and the PSTN address if you are working with dial-up LAPB. They are not used if you are working with non dial-up LAPB.
dl_qos_length	Will be ignored.
dl_qos_offset	Will be ignored.

This primitive is positively acknowledged with a DL_CONNECT_CON primitive. If there is a local error, this primitive is nack'ed with a DL_ERROR_ACK, with the possible error codes listed below. If the destination cannot be reached, this primitive is nack'ed with a DL_DISCONNECT_IND primitive.

Errors

TABLE 11–16 DL_CONNECT_REQ errors

Error	Description
DL_OUTSTATE	Primitive issued from an invalid state.
DL_BADADDR	The destination DLSAP address was invalid, for one of the following reasons: dl_dest_addr_length is incorrect, zero or, odd SAP (when using LLC2) loopback connection to the same SAP

TABLE 11–16 DL_CONNECT_REQ errors *(continued)*

Error	Description
DL_ACCESS	Attempt to connect a second LLC2 stream, bound to the same SAP, to the same destination DLSAP (really need to return an "address already in use" error in this case, but no such error exists in DLPI).
DL_SYSERR	Could not allocate memory.

11.1.9 DL_CONNECT_RES—Accept a Connect Request

Accept a connect request from a remote user. Causes a UA frame to be sent over the network (to ack the SABME or SABM that was received earlier). This message consists of one M_PROTO message block containing the following structure:

```
typedef struct {
        t_uscalar_t     dl_primitive;   /* DL_CONNECT_RES */
        t_uscalar_t     dl_correlation; /* provider's correlation token */
        t_uscalar_t     dl_resp_token;  /* token of responding stream */
        t_uscalar_t     dl_qos_length;  /* length of qos structure */
        t_uscalar_t     dl_qos_offset;  /* offset from start of block */
        t_uscalar_t     dl_growth;      /* set to zero */
} dl_connect_res_t;
```

The members of the dl_connect_res_t structure are:

TABLE 11–17 Members of the dl_connect_res_t structure

Member	Description
dl_primitive	Should be set to the name of this primitive.
dl_correlation	Contains the correlation number passed upstream in the DL_CONNECT_IND message.
dl_resp_token	Contains the token of the stream that will accept the connection, if the accepting stream is not the listen stream (applies to LLC2 only).
dl_qos_length	Will be ignored.
dl_qos_offset	Will be ignored.

Errors

TABLE 11–18 DL_CONNECT_RES errors

Error	Description
DL_OUTSTATE	Primitive issued from an invalid state, or the accepting stream is not in state DL_IDLE (attached and bound) or is not attached to the same PPA.
DL_BADCORR	The dl_correlation parameter does not correspond to the ID of a pending connection.
DL_BADTOKEN	The dl_resp_token parameter does not correspond to a currently open stream.
DL_ACCESS	Accepting stream is not bound to the same SAP as the listen stream.
DL_PENDING	Attempt to accept a connection on the listen stream when there are other outstanding connect indications on the listen stream, or an attempt to accept a connection on the “connection management” stream.
DL_SYSERR	Could not allocate STREAMS resources.

11.1.10 DL_DETACH_REQ—Undoes a Previous DL_ATTACH_REQ

Undoes a previous DL_ATTACH_REQ. This primitive is sent in an M_PROTO message block.

Associated Structure

```
typedef struct {
  t_uscalar_t dl_primitive;   /* set to DL_DETACH_REQ */
 } dl_detach_req_t;
```

The members of the dl_detach_req_t structure are:

TABLE 11–19 Members of the `dl_detach_req_t` structure

Member	Description
`dl_primitive`	Should be set to the name of this primitive.

Errors:

TABLE 11–20 `DL_DETACH_REQ` errors

Error	Description
`DL_OUTSTATE`	Primitive issued from an invalid state.

11.1.11 DL_DISCONNECT_IND—Indicates Connection Disconnect

Informs the user that the connection has been disconnected, or that a pending connection has been aborted. This message is passed upstream when a DISC frame is received from the network, or if the ack timer expires (because either the remote end didn't respond to a SABME/ DL_CONNECT_REQ, or to a SABM, or because during data transfer the connection went down).

Associated Structure

This message consists of one M_PROTO message block containing the following structure:

```
typedef struct {
    t_uscalar_t   dl_primitive;    /* DL_DISCONNECT_IND */
    t_uscalar_t   dl_originator;   /* USER or PROVIDER */
    t_uscalar_t   dl_reason;       /* permanent or transient */
    t_uscalar_t   dl_correlation;  /* association with connect_ind */
 } dl_disconnect_ind_t;
```

The `dl_disconnect_ind_t` structure has the following members:

TABLE 11–21 Members of dl_disconnect_ind structure

Member	Description
dl_primitive	Should be set to the name of this primitive.
dl_originator	Set to either: DL_USER—a DISC frame was received from the network. DL_PROVIDER—the ack timer expired.
dl_reason	Set to: DL_CONREJ_DEST_UNREACH_TRANSIENT—a pending connection was aborted. DL_DISC_TRANSIENT_CONDITION—an active connection was disconnected. DL_DISC_PERMANENT_CONDITION— the physical level is not connected.
dl_correlation	If a pending incoming call is being aborted, this contains the correlation value that was passed in the DL_CONNECT_IND primitive.

11.1.12 DL_DISCONNECT_REQ—Disconnects a Connection

Used to disconnect a connection. Can be used to disconnect an active connection (in state DL_DATAXFER), to refuse an incoming connection (which was indicated by the reception of a DL_CONNECT_IND primitive), or to cancel a previous DL_CONNECT_REQ primitive before the DL_CONNECT_CON acknowledgment is received back from the other end.

When the user issues this primitive, one of two things will happen, depending on the state that the stream is in:

1. In state DL_DATAXFER or if cancelling a previous DL_CONNECT_REQ, a DISC command frame is sent across the network to the destination.
2. If refusing an incoming connection, a DM response frame is sent (in response to the SABME that was received earlier).

Associated Structure

This message consists of one M_PROTO message block containing the following structure:

```
typedef struct {
        t_uscalar_t     dl_primitive;   /* DL_DISCONNECT_REQ */
        t_uscalar_t     dl_reason;      /* norm., abnorm., perm. or trans. */
        t_uscalar_t     dl_correlation; /* association with connect_ind */
} dl_disconnect_req_t;
```

The fields associated with the `dl_disconnect_req_t` structure are:

TABLE 11–22 Members of the `dl_disconnect_req_t` structure

Member	Description
`dl_primitive`	Should be set to the name of this primitive.
`dl_reason`	Any value passed here will be ignored, as LLC2 and LAPB have no means of carrying a "disconnect reason" across the network.
`dl_correlation`	If the user is rejecting an incoming call, this needs to be set to the correlation value supplied in the received `DL_CONNECT_IND` primitive. Otherwise, this parameter should be set to 0.

Errors

TABLE 11–23 `DL_DISCONNECT_REQ` errors

Error	Description
`DL_OUTSTATE`	Primitive issued from an invalid state.
`DL_BADCORR`	Non-zero correlation value supplied when not rejecting an incoming call, or invalid correlation value supplied when rejecting an incoming call.

11.1.13 DL_ERROR_ACK—Negative Acknowledgment

This primitive is sent upstream to negatively acknowledge a previous primitive.

Associated Structure

It is sent in an M_PCPROTO message block, with the following structure:

```
typedef struct {
        t_uscalar_t     dl_primitive;           /* DL_ERROR_ACK */
        t_uscalar_t     dl_error_primitive;     /* primitive in error */
        t_uscalar_t     dl_errno;               /* DLPI error code */
        t_uscalar_t     dl_unix_errno;          /* UNIX system error code */
} dl_error_ack_t;
```

The members of the dl_error_ack_t structure are:

TABLE 11–24 Members of the dl_error_ack_t structure

Member	Description
dl_primitive	Should be set to the name of this primitive.
dl_error_primitive	Set to the name of the primitive in error.
dl_unix_errno	Unix error code, if dl_errno is equal to DL_SYSERR
dl_errno	Contains the DLPI error code.

11.1.14 DL_INFO_ACK—Convey Info Summary

The LLC2 and LAPB drivers respond to the DL_INFO_REQ with a DL_INFO_ACK message.

Associated Structure

This message consists of one M_PCPROTO message block, with the following structure:

```
typedef struct {
        t_uscalar_t     dl_primitive;           /* set to DL_INFO_ACK */
        t_uscalar_t     dl_max_sdu;             /* Max bytes in a DLSDU */
        t_uscalar_t     dl_min_sdu;             /* Min bytes in a DLSDU */
        t_uscalar_t     dl_addr_length;         /* length of DLSAP address */
        t_uscalar_t     dl_mac_type;            /* type of medium supported */
        t_uscalar_t     dl_reserved;            /* value set to zero */
        t_uscalar_t     dl_current_state;       /* state of DLPI interface */
        t_scalar_t      dl_sap_length;          /* length of dlsap SAP part */
        t_uscalar_t     dl_service_mode;        /* CO, CL or ACL */
        t_uscalar_t     dl_qos_length;          /* length of qos values */
```

```
        t_uscalar_t     dl_qos_offset;          /* offset from start of block */
        t_uscalar_t     dl_qos_range_length;    /* available range of qos */
        t_uscalar_t     dl_qos_range_offset;    /* offset from start of block */
        t_uscalar_t     dl_provider_style;      /* style1 or style2 */
        t_uscalar_t     dl_addr_offset;         /* offset of the dlsap addr */
        t_uscalar_t     dl_version;             /* version number */
        t_uscalar_t     dl_brdcst_addr_length;  /* length of broadcast addr */
        t_uscalar_t     dl_brdcst_addr_offset;  /* offset from start of block */
        t_uscalar_t     dl_growth;              /* set to zero */
} dl_info_ack_t;
```

The members of the `dl_info_ack_t` structure are:

TABLE 11–25 members of the `dl_info_ack_t` structure

Member	Description
`dl_primitive`	Should be set to the name of this primitive.
`dl_mac_type`	Valid for LLC2 only. Set to the value returned by the hardware driver underneath LLC2.
`dl_mdlc_type`	Valid for LAPB only. Set to the value returned by the hardware driver underneath LAPB.
`dl_current_state`	Indicates the current DLPI state of the interface.
`dl_sap_length`	For LLC2, set to 1, which indicates that the SAP is one byte long, and follows the physical address in the DLSAP address. For LAPB, set to 0.
`dl_addr_offset`	Offset to the DLSAP address of this stream.
`dl_brdcst_addr_off`	Offset to the hardware broadcast address.

The QOS fields are always set to 0 when used over Ethernet/802.3. (`dl_qos_length`, `dl_qos_offset`, `dl_qos_range_length`, `dl_qos_range_offset`). Their settings above other media (FDDI, Token Ring, etc.) is to be defined. For descriptions of the other parameters, refer to the DLPI specifications.

11.1.15 DL_INFO_REQ—Request Info Summary

This primitive requests information about the stream.

Associated Structure

This primitive is sent in an M_PROTO message block, with the following structure:

```
typedef struct {
        t_uscalar_t     dl_primitive;                   /* set to DL_INFO_REQ */
} dl_info_req_t;
```

The members of the dl_info_req_t structure are:

TABLE 11–26 Members of the dl_info_req_t structure

Member	Description
dl_primitive	The name of this primitive.

11.1.16 DL_OK_ACK—Acknowledge Previous Primitive

This primitive is sent upstream to positively acknowledge a previous primitive. It is sent in an M_PCPROTO message block

Associated Structure

```
typedef struct {
        t_uscalar_t     dl_primitive;           /* DL_OK_ACK */
        t_uscalar_t     dl_correct_primitive;   /* primitive acknowledged */
} dl_ok_ack_t;
```

The members of the dl_ok_ack_t structure are:

TABLE 11–27 Members of the dl_ok_ack_t structure

Members	Description
dl_primitive	Should be set to the name of this primitive.
dl_correct_primitive	Set to the name of the primitive being acknowledged.

11.1.17 DL_RESET_CON—Acknowledges DL_RESET_REQ

Positively acknowledges a previous DL_RESET_REQ primitive. This primitive is sent upstream when a UA frame arrives to ack a previously sent SABME frame.

Associated Structure

This message consists of one M_PROTO message block containing the following structure:

```
typedef struct {
        t_uscalar_t     dl_primitive;           /* DL_RESET_CON */
} dl_reset_con_t;
```

The members of the dl_reset_con_t structure are:

TABLE 11–28 Members of the dl_reset_con_t structure

Member	Description
dl_primitive	Should be set to the name of this primitive.

11.1.18 DL_RESET_IND—Indicates Remote Reset

Indicates that the remote user is resynchronizing the connection. This primitive is sent upstream when a SABME or SABM is received from the network, while in state DL_DATAXFER.

Associated Structure

This message consists of one M_PROTO message block containing the following structure:

```
typedef struct {
        t_uscalar_t     dl_primitive;   /* DL_RESET_IND */
        t_uscalar_t     dl_originator;  /* Provider or User */
        t_uscalar_t     dl_reason;      /* flow control, link error, resync */
} dl_reset_ind_t;
```

The members of the dl_reset_ind_structure are:

TABLE 11–29 Members of the `dl_reset_ind_structure`

Member	Description
`dl_primitive`	Should be set to the name of this primitive.
`dl_originator`	Always set to `DL_USER`.
`dl_reason`	Always set to `DL_RESET_RESYNCH` (there is no means of carrying a "reset reason" across the network).

11.1.19 DL_RESET_REQ—Request Connection Reset

Used to resynchronize a connection. When the user issues this primitive while the stream is in state `DL_DATAXFER`, a `SABME` or `SABM` frame is sent across the network to the destination.

Associated Structure

This message consists of one `M_PROTO` message block containing the following structure:

```
typedef struct {
        t_uscalar_t     dl_primitive;   /* DL_RESET_REQ */
} dl_reset_req_t;
```

This primitive is positively acknowledged with a `DL_RESET_CON` primitive. If there is a local error, this primitive is negatively acknowledged with a `DL_ERROR_ACK`, with the possible error codes listed below.

The members of the `dl_reset_req_t` structure are:

TABLE 11–30 Members of the `dl_reset_req_t` structure

Member	Description
`dl_primitive`	Should be set to the name of this primitive.

Errors

TABLE 11–31 DL_RESET_REQ errors

Error	Description
DL_OUTSTATE	Primitive issued from an invalid state.

11.1.20 DL_RESET_RES—Respond to Reset Request

Respond to a reset request (an incoming DL_RESET_IND). Causes a UA frame to be sent over the network (to ack the SABME or SABM that was received earlier).

Associated Structure

This message consists of one M_PROTO message block containing the following structure:

```
typedef struct {
        t_uscalar_t     dl_primitive;           /* DL_RESET_RES */
} dl_reset_res_t;
```

The members of the dl_reset_res_t structure are:

TABLE 11–32 Members of the dl_reset_res_t structure

Member	Description
dl_primitive	Should be set to the name of this primitive.

Errors

TABLE 11–33 DL_RESET_RES errors

Error	Description
DL_OUTSTATE	Primitive issued from an invalid state.
DL_SYSERR	Could not allocate STREAMS resources.

11.1.21 DL_TOKEN_ACK—Acknowledges DL_TOKEN_REQ

The DL_TOKEN_REQ primitive is positively acknowledged with a DL_TOKEN_ACK primitive, which is encoded in an M_PCPROTO message block containing the following structure:

```
typedef struct {
        t_uscalar_t     dl_primitive;   /* DL_TOKEN_ACK */
        t_uscalar_t     dl_token;       /* Connection response token */
}dl_token_ack_t;
```

The members of the dl_token_ack_t structure are:

TABLE 11–34 Members of the dl_token_ack_t structure

Member	Description
dl_primitive	Should be set to the name of this primitive.
dl_token	Contains the connection response token.

11.1.22 DL_TOKEN_REQ—Assigns Token to Stream

Used to determine the token associated with a LLC2 stream (LAPB does not support). This token can then be supplied in the DL_CONNECT_RES primitive to indicate that the connection should be accepted on a different stream from the listen stream. The accepting stream must be attached and bound to the same PPA and SAP as the listen stream.

Note - This primitive is not supported by LAPB.

Associated Structure

This message consists of one M_PROTO message block containing the following structure:

```
typedef struct {
        t_uscalar_t     dl_primitive;   /* DL_TOKEN_REQ */
} dl_token_req_t;
```

The members of the dl_token_req_t structure are:

TABLE 11–35 Members of the dl_token_req_t structure

Member	Description
dl_primitive	Should be set to the name of this primitive.

11.1.23 DL_UNBIND_REQ—Summary

This primitive unbinds the STREAM that was bound by a previous DL_BIND_REQ. It is sent in an M_PROTO message block.

Associated Structure:

```
typedef struct {
        t_uscalar_t     dl_primitive;   /* DL_UNBIND_REQ */
} dl_unbind_req_t;
```

The members of the dl_unbind_req_t structure are:

TABLE 11–36 Members of the dl_unbind_req_t structure

Member	Description
dl_primitive	Should be set to the name of this primitive.

11.2 Sun-Specific ioctls

The following ioctls are specific to Sun. Take care when using them in programs that interwork with other versions of X.25.

These ioctls must be used with the ioctl(2) system call.

The following ioctls are related to statistics:

TABLE 11–37 Statistics Ioctls

name	summary
L_GETSTATS	reads per-link statistics (LAPB only)
L_ZEROSTATS	zeros per-link statistics (LAPB only)
L_GETGSTATS	reads global layer two statistics (LAPB only)

These ioctls are related to configuring a stream:

TABLE 11–38 Stream Configuration Ioctls

name	summary
L_SETPPA	sets the PPA. (LAPB only)
L_GETPPA	retrieves the PPA (LAPB only)
L_SETTUNE	sets tunable parameters for a PPA
L_GETTUNE	retrieves the tunable parameters for a PPA
W_SETTUNE	sets wanmod tunable parameters for a PPA (LAPB only)

11.2.1 Common ioctls

The ioctles described in this section can be used over both LAPB and LLC2.

11.2.1.1 L_GETTUNE—Retrieves Tunable Parameters for a PPA

The L_GETTUNE ioctl retrieves the tunable parameters in the LLC2 and LAPB drivers for a given PPA.

Associated Structures

LLC2 uses the llc2_tnioc structure.

```
/* Ioctl block for LLC2 L_GETTUNE command */
 struct llc2_tnioc {
   u_char   lli_type;       /* Table type = LI_LLC2TUNE */
   u_char   lli_spare[3];   /*   (for alignment)        */
   u_int    lli_ppa;        /* PPA (0xff for all PPAs) */
   llc2tune_t    llc2_tune; /* Table of tuning values   */
 };

 /* LLC2 tuning structure */
 typedef struct llc2tune {
   u_short   N2;         /* Maximum number of retries */
   u_short   T1;         /* Acknowledgment time (unit 0.1 sec)  */
   u_short   Tpf;        /* P/F cycle retry time (unit 0.1 sec)  */
   u_short   Trej;       /* Reject retry time (unit 0.1 sec)  */
   u_short   Tbusy;      /* Remote busy check time   (unit 0.1 sec)  */
   u_short   Tidle;      /* Idle P/F cycle time (unit 0.1 sec)  */
   u_short   ack_delay;  /* RR delay time (unit 0.1 sec)  */
   u_short   notack_max; /* Maximum number of unack'ed Rx I-frames */
   u_short   tx_window;  /* Transmit window (if no XID received) */
   u_short   tx_probe;   /* P-bit position before end of Tx window  */
   u_short   max_I_len;  /* Maximum I-frame length */
   u_short   xid_window  /* XID window size (receive window) */
   u_short   xid_Ndup;   /* Duplicate MAC XID count (0 => no test) */
   u_short   xid_Tdup;   /* Duplicate MAC XID time   (unit 0.1 sec) */
 } llc2tune_t;
```

The members of the llc2_tnoic structure are:

TABLE 11–39 Members of the llc2_tnoic structure

Member	Description
lli_type	The table type
lli_ppa	The PPA
llc2_tune	A table of tuning values

LAPB uses the lapb_tnioc structure

```
/* Ioctl block for LAPB L_GETTUNE command */
 struct lapb_tnioc {
   u_char   lli_type;     /* Table type = LI_LLAPBTUNE */
   u_char   lli_spare[3]; /*   (for alignment)        */
   u_int    lli_ppa;      /* PPA (0xff for all PPAs) */
   lapbtune_t  lapb_tune; /* Table of tuning values   */
 };

 /* LAPB tuning structure */
 typedef struct lapb_tune {
   uint16  N2;         /* Maximum number of retries                 */
   uint16  T1;         /* Acknowledgment time     (unit 0.1 sec)   */
   uint16  Tpf;        /* P/F cycle retry time     (unit 0.1 sec)  */
   uint16  Trej;       /* Reject retry time        (unit 0.1 sec)  */
   uint16  Tbusy;      /* Remote busy check time   (unit 0.1 sec)  */
   uint16  Tidle;      /* Idle P/F cycle time      (unit 0.1 sec)  */
   uint16  ack_delay;  /* RR delay time            (unit 0.1 sec)  */
   uint16  notack_max; /* Maximum number of unack'ed Rx I-frames   */
   uint16  tx_window;  /* Transmit window size                     */
   uint16  tx_probe;   /* P-bit position before end of Tx window   */
   uint16  max_I_len;  /* Maximum I-frame length                   */
   uint16  llconform;  /* LAPB conformance                         */
 } lapbtune_t;
```

The members of the `lapb_tnoic` structure are:

TABLE 11–40 Members of the `lapb_tnoic` structure

Member	Description
`lli_type`	The table type
`lli_ppa`	The PPA
`lapb_tune`	A table of tuning values

11.2.1.2 `L_SETTUNE`—Sets Tunable Parameters for a PPA

The `L_SETTUNE` ioctl sets tunable parameters in the LLC2 and LAPB drivers for a given PPA.

Associated Structures

LLCS uses the `llc2_tnioc` structure

```
/* Ioctl block for LLC2 L_SETTUNE command */
 struct llc2_tnioc {
```

```
  u_char  lli_type;       /* Table type = LI_LLC2TUNE */
  u_char  lli_spare[3];   /*   (for alignment)        */
  u_int   lli_ppa;        /* PPA (0xff for all PPAs) */
  llc2tune_t  llc2_tune;  /* Table of tuning values   */
};

/* LLC2 tuning structure */
typedef struct llc2tune {
  u_short  N2;          /* Maximum number of retries */
  u_short  T1;          /* Acknowledgment time (unit 0.1 sec)  */
  u_short  Tpf;         /* P/F cycle retry time (unit 0.1 sec)  */
  u_short  Trej;        /* Reject retry time (unit 0.1 sec)  */
  u_short  Tbusy;       /* Remote busy check time   (unit 0.1 sec)  */
  u_short  Tidle;       /* Idle P/F cycle time (unit 0.1 sec)  */
  u_short  ack_delay;   /* RR delay time (unit 0.1 sec)  */
  u_short  notack_max;  /* Maximum number of unack'ed Rx I-frames */
  u_short  tx_window;   /* Transmit window (if no XID received) */
  u_short  tx_probe;    /* P-bit position before end of Tx window  */
  u_short  max_I_len;   /* Maximum I-frame length */
  u_short  xid_window;  /* XID window size (receive window) */
  u_short  xid_Ndup;    /* Duplicate MAC XID count (0 => no test) */
  u_short  xid_Tdup;    /* Duplicate MAC XID time   (unit 0.1 sec) */
} llc2tune_t;
```

The members of the `llc2_tnoic` structure are:

TABLE 11–41 Members of the `llc2_tnoic` structure

Member	Description
`lli_type`	The table type
`lli_ppa`	The PPA
`llc2_tune`	A table of tuning values

LAPB uses the `lapb_tnioc` structure

```
/* Ioctl block for LAPB L_SETTUNE command */
 struct lapb_tnioc {
  u_char  lli_type;       /* Table type = LI_LLAPBTUNE */
  u_char  lli_spare[3];   /*   (for alignment)        */
  u_int   lli_ppa;        /* PPA (0xff for all PPAs) */
  lapbtune_t    lapb_tune;/* Table of tuning values   */
 };

 /* LAPB tuning structure */
 typedef struct lapb_tune {
  uint16  N2;         /* Maximum number of retries               */
  uint16  T1;         /* Acknowledgment time     (unit 0.1 sec)  */
  uint16  Tpf;        /* P/F cycle retry time     (unit 0.1 sec)  */
  uint16  Trej;       /* Reject retry time        (unit 0.1 sec)  */
```

```
    uint16  Tbusy;      /* Remote busy check time   (unit 0.1 sec)  */
    uint16  Tidle;      /* Idle P/F cycle time      (unit 0.1 sec)  */
    uint16  ack_delay;  /* RR delay time            (unit 0.1 sec)  */
    uint16  notack_max; /* Maximum number of unack'ed Rx I-frames   */
    uint16  tx_window;  /* Transmit window size                     */
    uint16  tx_probe;   /* P-bit position before end of Tx window   */
    uint16  max_I_len;  /* Maximum I-frame length                   */
    uint16  llconform;  /* LAPB conformance                         */
  } lapbtune_t;
```

The members of the lapb_tnoic structure are:

TABLE 11–42 Members of the lapb_tnoic structure

Member	Description
lli_type	The table type
lli_ppa	The PPA
lapb_tune	A table of tuning values

11.2.2 LAPB ioctls

The ioctls described in this section can only be used over LAPB.

11.2.2.1 L_GETGSTATS—Reads Global Layer 2 Statistics

The L_GETGSTATS ioctl reads global layer 2 statistics from the LAPB driver.

Associated Structures

The lapb_gstioc structure is used.

```
/* Ioctl block for L_GETGSTATS ioctl */
 struct lapb_gstioc {
     uint8         lli_type;      /* Table type = LI_GSTATS  */
     uint8         lli_spare[3];  /*   (for alignment)   */
     uint32        lapbgstats[globstatmax];
        /* global statistics table           */
 };
```

```
/* Global L2 statistics */
#define frames_tx     0 /* frames transmitted  */
#define frames_rx     1 /* frames received  */
#define sabme_tx      2 /* SABMEs transmitted  */
#define sabme_rx      3 /* SABMEs received  */
#define bytes_tx      4 /* data bytes transmitted */
#define bytes_rx      5 /* data bytes received  */
#define globstatmax   6 /* size of global stats array */
```

The members of the `lapb_gstioc` structure are:

TABLE 11–43 Members of the `lapb_gstioc` structure

Member	Description
`lli_type`	The table type
`lapbgstats`	The global statistics table

11.2.2.2 `L_GETPPA`—Returns the PPA Associated With a Stream

This ioctl returns the PPA and link index associated with the stream.

Associated Structure:

```
/* Ioctl block for L_SETPPA and L_GETPPA commands */
 struct ll_snioc {
     uint8       lli_type;     /* Table type = LI_SPPA            */
     uint8       lli_class;    /* DTE/DCE/extended                */
     uint16      lli_slp_pri;  /* SLP priority                    */
     uint32      lli_ppa;      /* PPA/ Subnetwork ID character    */
     uint32      lli_index;    /* Link index                      */
 };
```

The members of the `ll_snioc` structure are:

TABLE 11–44 Members of the `ll_snioc` structure

Member	Description
`lli_type`	The table type. This should always be LI_SPPA.
`lli_class`	This indicates the type of link. `LC_LLC2` must be used for LLC2; `LC__LAPBDTE` or `LC_LAPBDCE` must be used for LAPB. The file `/usr/include/netdlc/ll_proto.h` contains a complete list of values.
`lli_slp_pri`	This determines the priority of SLP when MLP is used.
`lli_ppa`	The PPA identifier
`lli_index`	The link index. This must be set with the `muxid` value returned by the `I-LINK` ioctl when LAPB is placed over a serial driver.

Errors

TABLE 11–45 L_GETPPA errors

Error	Description
`ENODEV`	No such device or a `DL_ATTACH_REQ` has not been sent.

11.2.2.3 `L_GETSTATS`—Retrieves Per-Link Statistics

The `L_GETSTATS` ioctl reads per-link (i.e., per-PPA) statistics from the LAPB driver.

Associated Structures

The `lapb_stioc` structure is used

```
/* Ioctl block for L_GETSTATS ioctl */
 struct lapb_stioc {
     uint8          lli_type;      /* Table type = LI_STATS  */
     uint8          lli_spare[3];  /*   (for alignment)   */
     uint32         lli_ppa;       /* PPA    */
     lapbstats_t    lli_stats;     /* Table of stats values  */
 };
```

The `lapbstats_t` structure needed for `L_GETSTATS` is defined as follows:

CODE EXAMPLE 11–1 `lapbstats_t` structure

```
typedef struct lapb2_stats {
  uint32 lapbmonarray[laphstatmax]; /* array of LAPB stats */
 } lapbstats_t;

 /* Statistics table definitions  */
 #define tx_ign              0 /* no. ignored + not sent */
 #define rx_badlen           1 /* bad length frames received */
 #define rx_unknown          2 /* unknown frames received */
 #define t1_exp              3 /* no. of T1 timeouts  */
 #define t4_exp              4 /* no. of T4 timeouts  */
 #define t4_n2_exp           5 /* T4 timeouts after N2 times */

 #define RR_rx_cmd           6 /* RR = Receive Ready  */
 #define RR_rx_rsp           7 /* tx = transmitted  */
 #define RR_tx_cmd           8 /* rx = received  */
 #define RR_tx_rsp           9 /* cmd/rsp = command/response */
 #define RR_tx_cmd_p        10 /* p = p-bit set        */

 #define RNR_rx_cmd         11 /* RNR = Receive Not Ready */
 #define RNR_rx_rsp         12
 #define RNR_tx_cmd         13
 #define RNR_tx_rsp         14
 #define RNR_tx_cmd_p       15

 #define REJ_rx_cmd         16 /* REJ = Reject   */
 #define REJ_rx_rsp         17
 #define REJ_tx_cmd         18
 #define REJ_tx_rsp         19
 #define REJ_tx_cmd_p       20

 #define SABME_rx_cmd       21 /* SABME = Set Asynchronous  */
 #define SABME_tx_cmd       22 /*  Balanced Mode Extended */

 #define DISC_rx_cmd        23 /* DISC = Disconnect  */
 #define DISC_tx_cmd        24

 #define UA_rx_rsp          25 /* UA = Unnumbered   */
 #define UA_tx_rsp          26 /* Acknowledgment  */

 #define DM_rx_rsp          27 /*    */
 #define DM_tx_rsp          28

 #define I_rx_cmd           29 /* I = Information  */
 #define I_tx_cmd           30

 #define FRMR_rx_rsp        31 /* FRMR = Frame Reject  */
 #define FRMR_tx_rsp        32

 #define tx_rtr             33 /* no. of retransmitted frames  */
 #define rx_bad             34 /* erroneous frames received    */
 #define rx_dud             35 /* received and discarded      */
 #define rx_ign             36 /* received and ignored         */

 #define I_rx_rsp           37
 #define I_tx_rsp           38
```

```
#define UI_rx_cmd          39
#define UI_tx_cmd          40

#define XID_rx_cmd         41
#define XID_rx_rsp         42
#define XID_tx_cmd         43
#define XID_tx_rsp         44

#define TEST_rx_cmd        45
#define TEST_rx_rsp        46
#define TEST_tx_cmd        47
#define TEST_tx_rsp        48

#define llc2statmax        40
```

11.2.2.4 L_SETPPA—Associates a PPA With a Physical Device

This ioctl associates a PPA with a physical device underneath the layer two provider.

Associated Structure:

```
/* Ioctl block for L_SETPPA and L_GETPPA commands */
 struct ll_snioc {
     uint8        lli_type;     /* Table type = LI_SPPA            */
     uint8        lli_class;    /* DTE/DCE/extended                */
     uint16       lli_slp_pri;  /* SLP priority                    */
     uint32       lli_ppa;      /* PPA/ Subnetwork ID character    */
     uint32       lli_index;    /* Link index                      */
 };
```

The members are:

TABLE 11–46 Members of the `ll_snioc` structure

Member	Description
`lli_type`	The table type. This should always be LI_SPPA.
`lli_class`	This indicates the type of link. `LC_LAPBDTE` or `LC_LAPBDCE` must be used for LAPB. The file `/usr/include/netdlc/ll_proto.h` contains a complete list of values.
`lli_slp_pri`	This determines the priority of SLP when MLP is used.

TABLE 11–46 Members of the ll_snioc structure *(continued)*

Member	Description
lli_ppa	The PPA identifier
lli_index	The link index. This must be set with the muxid value returned by the I-LINK ioctl when LAPB is placed over a serial driver.

Errors

TABLE 11–47 L_SETPPA errors

Error	Description
EBUSY	The PPA is already being used by another stream.
ENODEV	The specified lli_index has not been found.

11.2.2.5 L_ZEROSTATS—Clears the Per-Link Statistics Count

The L_ZEROSTATS ioctl clears per-link statistics in the LLC2 and LAPB drivers.

Associated Structure

```
/* Ioctl block for L_ZEROSTATS ioctl */
 struct ll_hdioc {
     uint8           lli_type;      /* Table type = LI_PLAIN */
     uint8           lli_spare[3];  /*   (for alignment)  */
     uint32          lli_ppa;       /* PPA (0xff for all links) */
 };
```

11.2.2.6 W_SETTUNE—Sets wanmod Tunable Parameters

The W_SETTUNE ioctl sets the tunable parameters of the LAPB wanmod module. This controls physical parameters such as the maximum frame length and line speed.

Associated Structure

The following is from the file /usr/include/netx25/wan_control.h.

```
/*    Ioctl block for WAN W_SETTUNE command
*/
struct wan_tnioc {
   uint8     w_type;     /* Always = WAN_TUNE                      */
   uint8     w_spare[3]; /* (for alignment)                        */
   uint32    w_snid;     /* subnetwork id character ('*' => 'all') */
   wantune_t wan_tune;   /* Table of tuning values
         */
};
/*    WAN tuning structure  */
typedef struct wantune  {
   uint16           WAN_options;  /* WAN options       */
   struct WAN_hddef  WAN_hd;      /* HD information.   */
   } wantune_t;
/*
This is the structure which contains all tuneable information
*/
struct  WAN_hddef  {
   uint16      WAN_maxframe;    /* WAN maximum frame size       */
   int         WAN_baud;        /* WAN baud rate                */
   uint16      WAN_interface ;  /* WAN physical interface       */
}
union {
   uint16         WAN_cptype;      /* Variant type */
   struct WAN_x21 WAN_x21def;
   struct WAN_v25 WAN_v25def;
   } WAN_cpdef ;                   /* WAN call procedural definition *
                                          * for hardware interface.        */
};
/*
This contains all of the national network specific timeouts.
*/
struct  WAN_v25  {
   uint16 WAN_cptype;   /* Variant type.        */
   uint16 callreq;      /* Abort time for call request command  *
};
```

The members of the wan_tnioc structure are:

TABLE 11–48 Members of the wan_tnioc structure

Member	Description
w_snid	The link id. It should be set to the same value as lli_ppa in the L_SETPPA L_SETTUNE ioctls.
WAN_options	Reserved for future use. Must be set to 0.
WAN_maxframe	The maximum frame size to be used on this interface (unit is octet).

TABLE 11–48 Members of the wan_tnioc structure *(continued)*

Member	Description
WAN_baud	The speed of the line (unit is baud, 0 is used for external clocking).
WAN_interface	The type of interface. Should always be set to WAN_V28
WAN_cptype	The type of interface. Set this to WAN_NONE if no calling procedures are used (the most frequent case), or to WAN_V25bis if a calling procedure and V25bis modem are used. In this instance, the WAN_v25 structure must be filled.

PART III Socket Interface

CHAPTER **12**

Compatibility with SunNet X.25 7.0 Sockets-Based Packet Level Interface

This chapter describes the sockets-based interface to the Solstice X.25 Packet Layer interface. In the current release, the sockets-based interface has been replaced by a streams-based interface. *The sockets-based interface is supported for backward-compatibility with SunNet X.25 7.0 only*. We strongly encourage you modify your existing X.25 applications to run over the streams-based interface described in the chapters of this manual.

Note - The sockets-based interface is a source-compatible—not a binary-compatible—interface. Applications that used the socket interface in SunOS 4.x must be recompiled to run on SunOS™ 5.x. See Section 13.2 "Compilation Instructions and Sample Programs " on page 242for instructions on compiling programs to use the sockets-based interface on SunOS 5.x.

12.1 Introduction — The AF_X25 Domain

This chapter assumes some familiarity with SunOS sockets and address domains (families). Briefly, the socket layer of the network system deals with the interprocess communications provided by the system. A socket is a descriptor that acts as a bidirectional endpoint for communications and is "typed" by the semantics of the communications it supports. The type of the socket is defined at socket creation time and used in selecting those services which are appropriate to support it. The socket type SOCK_STREAM provides sequenced, reliable, two-way, connection-based byte streams with an out-of-band data transmission mechanism. An address domain specifies an address format which is used to interpret addresses specified in later operations using the socket.

Solstice X.25 defines an address domain, AF_X25. Within this domain only the socket type SOCK_STREAM is supported. Like other SOCK_STREAM sockets, an AF_X25 domain socket is composed of two byte streams: an in-band stream and an out-of-band stream. However, unlike other sockets, there are two different kinds of out-of-band messages: X.25 status and interrupt data.

12.2 AF_X25 Domain Addresses

Addresses in the AF_X25 domain consist of two parts: a DTE address of up to 15 BCD digits and Call User Data of up to 16 bytes. (The leading bytes of the Call User Data is often a protocol identifier [PID] used to identify a specific application using X.25.) You can use either subaddressing (part of 15-digit DTE address) or both subaddressing and Call User Data as part of the binding mechanism to match Incoming Call packets with a server process.

An AF_X25 domain address is described by a CONN_DB structure:

```
typedef struct conn_db_s {
    u_char   hostlen;         /*address length in BCD digits */
    u_char   host[(MAXHOSTADR+1)/2];   /* DTE address */
    u_char   datalen;         /* user data length in bytes */
    u_char   data[MAXDATA];   /* user data */
 } CONN_DB;
```

The constants MAXHOSTADR and MAXDATA are defined in the include file x25_pk.h. Currently, MAXHOSTADR is 15, so the length of the host field is 8, and MAXDATA is 102. Use these constants, whenever possible, instead of hard-coded values.

The 15-digit DTE address comprises three components: a Data Network Identification Code (DNIC), a Network Terminal Number (NTN), and a subaddress. A full X.121 address is the concatenation of a DNIC, NTN, and subaddress, in that order. For example, if the DNIC is 4042, the NTN is 3831, and subaddress is 06, the full X.121 address is 4042383106.

Note that only eight bytes are provided for the X.121 address, which could be up to 15 digits in length. This is because each byte holds two BCD digits in packed format (it takes only four bits to represent a BCD digit). Thus the address 4042383106 will be stored as five bytes, with hexadecimal values 0x40, 0x42, 0x38, 0x31, and 0x06, in that order.

The necessary include files are listed in Chapter 13. For more information on address binding, see Section 12.3.4 “Address Binding ” on page 204.

12.3 Creating Switched Virtual Circuits

To set up a switched virtual connection between a local and remote system, a socket in the AF_X25 domain is created using the standard socket call:

```
int s;   /* socket to be created */
 s = socket(AF_X25, SOCK_STREAM, 0);
```

If a signal handler routine is to be used, it is necessary to associate a proper process group ID with the socket. Refer to the section Section 12.5.4 “Out-of-Band Data ” on page 214 of this chapter to see how this is done. X.25 facility specification and negotiation may be done after creating a socket. See Section 12.7.1 “Facility Specification and Negotiation” on page 217 of this chapter for more information regarding facility specification.

After a socket has been created, the client executes one of the two sequences described in the following subsections to set up the virtual circuit.

12.3.1 Calling Side — Outgoing Call Setup

The calling side initiates a virtual circuit connection by calling connect, supplying the called (remote) DTE address (including subaddress, if any) and a user data field as arguments. After `connect` completes successfully, the socket may be used for data transfer.

```
int   s /* socket */, error;
 CONN_DB   addr;
 error = connect(s, &addr, sizeof(addr));
```

Solstice X.25 supports multiple physical interfaces (or links). A single link maps to a serial port device, such as `zsh0`.

A link is automatically selected for the outgoing call. Among multiple links, Solstice X.25 routes outgoing calls based on the called address. Calls are routed according to the full or partial addresses (X.121, or NSAP or non-NSAP extended addresses) you specify in a `routes` file, the syntax for which is described in *Solstice X.25 9.2 Administration Guide*. The lowest-numbered link is the default.

If the interface supports 1984 X.25, the user may also specify a Called Address Extension Facility (AEF). In this case, Solstice X.25 will use the Called AEF to route the call over a particular link, provided the user has not specified an X.121 address. If the user wants the call to be routed based on the Called AEF, the hostlen field should be set to zero:

```
addr.hostlen = 0;
```

Where AEFs are used for routing, Solstice X.25 will select the interface to use and will also supply the X.121 address (if any) for the Call Request packet. In addition, if it is a LAN interface, Solstice X.25 will supply the necessary LSAP address.

Called and Calling AEFs are described in the section Section 12.7.1 “Facility Specification and Negotiation” on page 217.

Note - error is used in most examples to indicate the return code. A value of zero indicates a successful operation. A non-zero value indicates an unsuccessful operation. The cause of the error is stored in a global variable errno which is used throughout this manual. Values of errno are enumerated in <errno.h>. These values are listed in intro(2) in the SunOS Reference Manual. Programmers may access errno by inserting the following line in their programs: extern int errno; Note that errno indicates the cause of the very last system call failure and is therefore invalid for operations returning an error value of zero. To get more information on the meaning of the error string printed, use the perror function.

12.3.2 Calling Side — Setting the Local Address

Often, the receiver of an Incoming Call needs to know the address of the caller in order to validate the call. By default, the calling address in the Call Request is set to the address (including the subaddress, if any) specified in the configuration file of the link over which the Call Request is sent. There are several parameters in the link configuration file, all described in the preceding subsection, that determine how Solstice X.25 preprocesses the calling address to satisfy the requirements of the interface.

You may specify a different address using the X25_WR_LOCAL_ADR ioctl. The address is specified in a CONN_ADR structure.

```
typedef struct conn_adr_s {
    u_char   hostlen;   /* length of BCDs */
    u_char   host[(MAXHOSTADR+1)/2];
 } CONN_ADR;
```

Here, as in the CONN_DB structure, hostlen is the length of the address in BCD digits, and host contains the address in packed BCD format. The X25_WR_LOCAL_ADR ioctl call is issued as follows:

```
CONN_ADR addr;
 int s, error;
 error = ioctl(s, X25_WR_LOCAL_ADR, &addr);
```

The setting of the source address—and whether the X25_WR_LOCAL_ADR ioctl has effect—is controlled by the setting of the Source Address Control parameter in the Link Mode Parameters window in `x25tool`. See *Solstice X.25 9.2 Administration Guide* for instructions on setting this parameter.

12.3.3 Called Side — Incoming Call Acceptance

The called side initiates listening for incoming calls by calling `bind`, supplying the called (local) DTE address (including subaddress, if any) and protocol identifier to be used for matching with incoming calls:

```
int s, error;
 CONN_DB bind_addr;
 error = bind(s, &bind_addr, sizeof(bind_addr));
```

Here, `bind_addr` contains the address and protocol identifier of the called side. The protocol identifier is specified in the data field of the CONN_DB structure and is matched with the user data in incoming calls. More information on how to specify the address and protocol identifier for the `bind` call, and how incoming calls are matched with bound addresses and protocol identifiers, follows.

After `bind` has been called, `listen` is called to begin waiting for incoming calls. Incoming calls will be queued until they are accepted by means of the `accept` call. `backlog` specifies the maximum number of incoming calls (no more than five) to queue (waiting for `accept)` before clearing additional incoming calls.

```
int s, backlog, error;
 error = listen(s, backlog);
```

Finally, `accept` is called to block until an incoming call is received that matches the address and protocol identifier specified in the `bind` call. `accept` is passed a pointer to a CONN_DB structure (and length), which will be filled in with the calling DTE's (remote) address and user data field. The user data field in an Incoming Call packet consists of a protocol identifier followed by any additional user data. After an incoming call matches the binding criteria, `accept` returns the socket `news`, to be used for data transfer. `news` inherits the process group ID from `s`.

```
int    s, news;
 int    from_addr_len;
 CONN_DB from;
 from_addr_len = sizeof(from);
 news = accept(s, &from, &from_addr_len);
```

The remote address returned in `from` will be exactly as received (that is, in exactly the same form as received in the calling address field in the Incoming Call packet).

Note that on entry into the accept call, from_addr_len should be set to the size of the CONN_DB structure. On return, it will be set to the length of the actual address returned in from.

A typical caller of accept would be a server process that forks a new process (after calling accept) to handle each new socket. The sample programs (see Chapter 13") provided with Solstice X.25 illustrate how this can be done.

12.3.4 Address Binding

When an Incoming Call packet is received by Solstice X.25, the called address and user data field are matched against all listening sockets. In addition, if the interface supports 1984 X.25, and if the listener has specified a value for the Called AEF, the Called AEF field in the Incoming Call (if any) will be matched with the Called AEF specified by the listener. If a match is found, the call is accepted and the user process associated with that socket will be notified when the user process does an accept. This permits incoming calls to be bound to the correct user process. X.25 supports binding by either address or by both address and protocol identifier. The method used is determined by the fields of the CONN_DB structure passed to bind.

The address a socket is bound to is specified in the host field of the CONN_DB parameter passed to the bind call. The address is specified in packed BCD format, and the hostlen field contains the length of the address in BCD digits.

You can specify the bound address in a number of ways, depending on whether you want to accept all calls (from any link, for any subaddress), or all calls for a specific subaddress (from any link, for a particular subaddress), or calls from a specific link for any subaddress, or calls for a specific address (from a specific link, for a specific subaddress).

If you want to accept all calls (from any link, for any subaddress), set the bits ANY_LINK (0x80) and ANY_SUBADDRESS (0x40) in the hostlen field and do not specify any address:

```
bind_addr.hostlen = ANY_LINK | ANY_SUBADDRESS;
```

If you want to accept calls from any link, but only for a specific subaddress, specify only the subaddress, and set the ANY_LINK bit in the hostlen field:

```
bind_addr.hostlen |= ANY_LINK;
```

If you want to accept calls from a specific link, but for any subaddress, specify the link address (without the subaddress) and set the `ANY_SUBADDRESS` bit in the `hostlen` field:

```
bind_addr.hostlen |= ANY_SUBADDRESS;
```

If you want to accept calls for a specific address (including subaddress) specify the exact address in the CONN_DB structure passed to bind. In this case, the address you specify must exactly match the called address field of the received Incoming Call packet. The address of a link may be obtained with an X25_RD_LINKADR ioctl call (see the section Section 12.7.6 “Accessing the Link (X.25) Address ” on page 231 of this chapter for details).

The sample programs provided with Solstice X.25 illustrate the above features.

12.3.5 Binding by PID / CUDF

To bind by protocol identifier (PID), you must specify a protocol identifier in the data field of the CONN_DB parameter passed to bind. The datalen field contains the length of the protocol identifier. You can specify up to 102 bytes of protocol identifier, but only the first 16 bytes will be used for matching with user data in Incoming Call packets.

The user data field in an Incoming Call may be longer than the protocol identifier specified in bind. The match will be considered successful if the protocol identifier specified in bind is an initial sub-string of the user data in an Incoming Call. Thus, if you specify a zero-length protocol identifier in bind, it will match the user data in any Incoming Call.

You can enforce exact matching of the protocol identifier with user data in Incoming Call packets by setting the bit EXACT_MATCH (0x80) in datalen:

```
bind_addr.datalen |= EXACT_MATCH;
```

In this case, user data in an Incoming Call packet should match the protocol identifier specified in bind exactly (in content and length) in order for the match to be considered successful.

See Chapter 13,” for references to sample code. A simple example is given below:

```
CONN_DB bind_addr;
 int s, error;
 /*We want to accept calls from any link, for the subaddress 01.
 * We must specify the two digit subaddress 01 and set the ANY_LINK
 * bit in the hostlen field.
 */
 bind_addr.hostlen = 2 | ANY_LINK;   /* there are 2 BCD digits */
 bind_addr.host[0] = 0x01;
 /* We will specify a protocol identifier consisting of a single byte
 * with value 0x02.
 */
 bind_addr.datalen = 1;
 bind_addr.data[0] = 0x02;
 error = bind(s, &bind_addr, sizeof(bind_addr));
```

12.3.6 Masking Incoming Protocol Ids at Bit Level

The user data in an Incoming Call may be masked (that is, bitwise ANDed), using a specified mask value, before it is matched with the protocol identifier specified in a bind call. The mask is specified in a MASK_DATA_DB structure using the X25_WR_MASK_DATA ioctl. Here is an example:

```
typedef struct mask_data_bd_s {
    u_char   masklen;
    u_char   mask[MAXMASK];
 } MASK_DATA_DB;

 MASK_DATA_DB m;
 int s, error;
```

```
m.masklen = 3;
m.mask[0] = 0xff;
m.mask[1] = 0x00;
m.mask[2] = 0xff;

error = ioctl(s, X25_WR_MASK_DATA, &m);
```

MAXMASK is currently 16. masklen holds the length of the mask data in bytes, and mask is the actual mask value. In the above example, the first three bytes of user data in an Incoming Call will be masked: the first byte with 0xff, the second with 0x00, and the third with 0xff. The masked user data will then be matched with the specified protocol identifier. Note that the specified protocol identifier will not be masked before matching occurs, so in the above example, the second byte of the specified protocol identifier must be zero if any match is to succeed.

12.3.7 AEF Matching Considerations

A listener may specify a Called AEF. In this case, the Incoming Call packet must have the Called AEF, and it should match the Called AEF specified by the listener exactly, in order for the match to succeed. If the listener has not specified a Called AEF, any Called AEF present in the Incoming Call packet will be accepted, provided the match succeeds in other ways (Called Address and PID).

12.3.8 Explicit Link Selection—Calling Side

As discussed in a previous subsection, Solstice X.25 automatically selects a link for an outgoing call if so requested by the caller. If you do nothing to call automatic link selection into play, the call is sent over the lowest numbered WAN link by default. The calling side can override automatic link selection, and specify a desired link using the X25_SET_LINK ioctl:

```
int s, error;
 int linkid;        /* id of desired link for outgoing call */
 CONN_DB addr;      /* destination address */
 linkid = 3;        /* want to send call over link 3 */
 error = ioctl(s, X25_SET_LINK, &linkid);

 /* check error here */

 error = connect(s, &addr, sizeof(addr));
```

Note that a full X.121 address must be specified (and so indicated by setting the ANY_LINK bit as described earlier) if you want Solstice X.25 to process the address as required by the PSDN, using the parameters specified in the link configuration file. Otherwise, the address set in the Call Request packet will be exactly what you specified, and so you must take care to provide the address in exactly the form required by the PSDN.

Since setting the link prevents Solstice X.25 from consulting the routing table, all the information required to establish connection with the remote user must be provided. For example, if the link selected supports 1984 X.25, Called and Calling AEFs may be required. If the link selected is a LAN interface, the LSAP address of the remote user must be provided. This is done as follows:

```
typedef struct {
    u_char   lsel;
    u_char   maclen;
 #define MACADDR_LEN   6
    u_char   macaddr[MACADDR_LEN];
 } X25_MACADDR;

 X25_MACADDR dst_mac;         /* LSAP address */
 int s;                        /* socket */

 /* set the lsel, maclen and macaddr fields here */

 error = ioctl(s, X25_WR_MACADDR, &dst_mac);
```

If the lsel field is set to zero, Solstice X.25 will use the value specified in the link configuration file. After connection is established, the LSAP address of the remote user can be read using the X25_RD_MACADDR command:

```
X25_MACADDR dst_mac;         /* LSAP address */
 int s;                       /*socket */

 error = ioctl(s, X25_RD_MACADDR, &dst_mac);
```

12.3.9 Explicit Link Selection—Called Side

The called side may restrict the calls it wishes to examine for a possible match to a particular link by means of the X25_SET_LINK ioctl.

```
int s, linkid, error;
 CONN_DB addr;      /* address and protocol identifier */

 linkid = 2;        /* restrict calls to link 2 */
 error = ioctl(s, X25_SET_LINK, &linkid);

 /* check error here */

 error = bind(s, &addr, sizeof(addr));
```

The ANY_SUBADDRESS and ANY_LINK bits can still be used in the same way as explained in the sectionSection 12.3.4 “Address Binding ” on page 204 of this chapter. The ANY_LINK bit, in this context, serves as an abbreviation for the link address, and you do not have to specify the link address explicitly. A zero-length address also works in the same way as described in the Section 12.3.4 “Address Binding ” on page 204 section. Otherwise, you must specify the address in exactly the form it will be received. That is, it must exactly match the called address field of the received Incoming Call packet.

12.3.10 Accessing the Local and Remote Addresses

Once a connection is established, the calling and called sides may use the getsockname and getpeername calls to obtain the local and remote X.121 addresses:

```
int s, error;
 CONN_DB local;       /* local address */
 int local_len;       /* local address length */
 CONN_DB remote;      /* remote address */
 int remote_len;      /* remote address length */

 /* get local address */
 local_len = sizeof(local);
 error = getsockname(s, &local, &local_len);

 /* get remote address */
 remote_len = sizeof(remote);
 error = getpeername(s, &remote, &remote_len);
```

The local and remote addresses can also be obtained using the X25_RD_LOCAL_ADR and X25_RD_REMOTE_ADR ioctl calls:

```
int s, error;
 CONN_ADR local;       /* local address */
 CONN_ADR remote;      /* remote address */

 /* get local address */
 error = ioctl(s, X25_RD_LOCAL_ADR, &local);

 /* get remote address */
 error = ioctl(s, X25_RD_REMOTE_ADR, &remote);
```

Note that for getsockname and getpeername, the CONN_DB structure is used, and for the ioctl calls, the CONN_ADR structure is used. In both cases, the host field will contain the address in packed BCD format, and the hostlen field will contain the address length in BCD digits.

For the called side, the remote address will be defined only after the connection is complete. The remote address obtained using either of the above two methods will be exactly as obtained from the Incoming Call packet. After the call is established, the local address (obtained by either method) will be exactly as received in the called address field in the Incoming Call packet.

For the calling side, the remote address will be exactly as specified in the connect call. If the ANY_LINK bit was set in the hostlen field, it will be also set when it is read by the user using either of the above methods. The source address for the calling side will be either a zero-length address (indicating that the appropriate link address was used), or exactly what the user specified using the X25_WR_LOCAL_ADR ioctl call (including the SUBADR_ONLY bit if it is used).

12.3.11 Finding the Link Used for a Virtual Circuit

If you let Solstice X.25 select the link for an outgoing call, or make an accept call that accepts incoming calls from any link, you may use the X25_GET_LINK ioctl to obtain the identifier of the link used for the call:

```
int s, error;
 int linkid; /* link identifier */

 error = ioctl(s, X25_GET_LINK, &linkid);
```

If this call is made before connection establishment and you have not explicitly selected a link, linkid will be set to -1 on return from the call. After connection establishment, linkid will have a value in the range zero through one less than the maximum number of links configured.

An important use for this ioctl arises when the called side determines the remote address in order to call back the remote DTE. In this situation, the remote address is presented in exactly the form it arrived in the Call Request. For some PSDNs, this may not contain a DNIC. Hence, the only way you can call the remote DTE back is by finding out the link id for the call using the X25_GET_LINK ioctl, and explicitly selecting this link using the X25_SET_LINK ioctl when calling the remote DTE back. In this situation, you should *not* set the ANY_LINK bit in the hostlen field of the CONN_DB parameter to the connect call.

12.3.12 Determining the LCN for a Connection

To find out which logical channel is associated with a connection, do the following:

```
int s, lcn;
 error = ioctl(s, X25_RD_LCGN, &lcn);
```

Here, s is the socket associated with the connection (or virtual circuit). On return from the call, lcn is set to the logical channel number associated with socket s. If the returned value of lcn is 0, there is no connected virtual circuit associated with the socket.

12.4 Sending Data

The `send` call is used to send data over a virtual circuit. `send` is passed the socket, a pointer to the data to be transmitted, the length of the data, and a flag indicating the type of data to be sent. Interrupt data is sent by setting `flags` to `MSG_OOB`. Otherwise, `flags` should be set to zero. The returned `count` indicates the number of bytes transmitted by `send`.

```
int count, len, flags, s;
 char *msg;
 count = send(s, msg, len, flags);
```

Note that for normal data, you can use the write system call instead of send. The call:

```
write(s, msg, len)
```

is equivalent to:

```
send(s, msg, len, 0)
```

The X.25 protocol has the concept of an X.25 message. A complete X.25 message is a sequence of one or more packets with the M-bit (More bit) set in all but the final packet. Normally, X.25 sends the data specified in a `send` call as a complete message. This means that the data will be segmented into packets as required by the PSDN, and the M-bit will be set in all but the final packet. If the user wishes to pass the data in a complete X.25 message in pieces (that is, using multiple send calls), the setting of the M-bit must be controlled using the X25_SEND_TYPE ioctl as described below.

Note - In the current release of Solstice X.25, `send()` returns a positive result after a virtual circuit is closed at the remote end. This behavior is different from SunNet X.25 7.0. To be notified when the virtual circuit has been closed, use the `X25_OOB_ON_CLEAR` ioctl, as described in Section 12.7.8 “Accessing the Diagnostic Code” on page 232.

12.4.1 Control of the M-, D-, and Q-bits

The settings of M-, D- and Q-bits in transmitted packets are changed by means of the X25_SEND_TYPE ioctl call.

```
ints, send_type;
 error = ioctl(s, X25_SEND_TYPE, &send_type);
```

send_type provides the new settings of the M-, D-, and Q-bits. The M-, D-, and Q-bits are encoded into the send_type field by bit shifting as shown below.

```
#define M_BIT 0       /* number of bits to shift to set "more"
        * bit */
 #define D_BIT 2       /* number of bits to shift to set end-to-end
        * acknowledge bit */
 #define Q_BIT 3       /* number of bits to shift to set qualified
        * data bit */
```

For example, to set the Q-bit in a packet:

```
intsend_type = (1 << Q_BIT), s;
 error = ioctl(s, X25_SEND_TYPE, &send_type);
```

M_BIT determines whether or not a packet is the final piece of a complete X.25 message. If M_BIT is set, subsequent send calls are treated as part of a single X.25 message. If M_BIT is not set, the next send ends the current X.25 message. For example, the following code allows a complete X.25 message to be sent in three pieces:

```
ints, send_type, error;
 /* Set M_BIT to indicate multiple pieces */
 send_type = (1 << M_BIT);
 error = ioctl(s, X25_SEND_TYPE, &send_type);
 /* send first piece */
 error = send(s, &first_piece, sizeof(first_piece), 0);
 /* send next piece */
 error = send(s, &second_piece, sizeof(second_piece), 0);
 /* Clear M_BIT to indicate end of message */
 send_type = 0;
 error = ioctl(s, X25_SEND_TYPE, &send_type);
 /* send final piece */
 error = send(s, &final_piece, sizeof(final_piece), 0);
```

If the M-bit is turned on using the X25_SEND_TYPE ioctl, it will stay turned on until it is turned off. The X.25 recommendation states that the M-bit shall be turned on only in packets that are "full"—that is, packets that have the maximum size for that virtual circuit. So if the M-bit is turned on, and the next send does not supply a full X.25 packet, X.25 will wait until enough send calls have been issued to build a full X.25 packet before transmitting the next packet with the M-bit turned on.

The Q-bit qualifies the data in Data packets. A local DTE sets the Q-bit to indicate that the data being sent is significant for a device connected to the remote DTE. It is often used by a remote host when sending control packets to a PAD, to distinguish the control packets from packets containing user data.

The D-bit allows a local DTE to specify end-to-end acknowledgment of data packets. Normally, a DTE receives acknowledgement only from its local DCE. The D-bit is significant only in call setup and data packets.

D_BIT and Q_BIT control the settings of those bits in an X.25 packet. These bits are manipulated in the same manner as the M_BIT was above. Since the X.25 recommendation states that the D_BIT and Q_BIT bits should remain constant for each packet in a complete X.25 message, D_BIT and Q_BIT should only be changed at the beginning of an X.25 message.

Unlike M_BIT, D_BIT and Q_BIT are turned off automatically after a complete X.25 message has been sent. Hence, to set these bits in a series of complete X.25 messages, you should turn them on at the start of each complete X.25 message. If the complete X.25 message is a sequence of full packets with the more bit turned on in all but the last packet in the sequence, the setting of D_BIT and Q_BIT will be the same for all the packets unless you explicitly change the setting in between.

12.4.2 Sending Interrupt and Reset Packets

An interrupt packet may be sent in the following manner. The interrupt user data is contained in intr:

```
int s;
 char intr = 0;       /* set this variable to contain the interrupt
                       * user data (in this case 0) */
 error = send(s, &intr, 1, MSG_OOB);
```

If the link supports 1984 X.25, you may send up to 32 bytes of interrupt data. On 1980 links, you may send only one byte.

A reset packet may be sent in the following manner:

```
X25_CAUSE_DIAG diag;
 int error, s;
 diag.flags = 0;
 diag.datalen = 2;
 diag.data[0] = 0;          /* cause */
 diag.data[1] = 67;         /* diagnostic */
 error = ioctl(s, X25_WR_CAUSE_DIAG, &diag);
```

This will cause a Reset to be sent with the cause code and diagnostic specified by the user. See Section 12.7.8 “Accessing the Diagnostic Code” on page 232 of this chapter for more information.

12.5 Receiving Data

To read data from an X.25 socket, call recv. Data may be either in-band (normal data) or out-of-band (interrupt data and status). To receive out-of-band data, set flags to `MSG_OOB`. To receive normal data, set flags to 0.

```
int s, len, flags, count;
 char *buf;
 count = recv(s, buf, len, flags);
```

Note that for normal data, you can use the read system call instead of recv. The call:

```
read(s, buf, len)
```

is equivalent to:

```
recv(s, buf, len, 0)
```

12.5.1 In-Band Data

Calling recv with flags set to zero reads in-band data. Normally, each recv returns one complete X.25 message. It is very important to note that if the size of the receive buffer is not sufficient to hold the entire X.25 message, the excess is discarded and no error indication is returned. This is a feature of SunOS sockets, not of Solstice X.25. `count` returns a count of the number of bytes returned by recv. If the user wishes to read an X.25 message in pieces smaller than a complete message, the X25_RECORD_SIZE ioctl should be used as described in the section Section 12.5.3 "Receiving X.25 Messages in Records " on page 214of this chapter.

Unless non-blocking I/O has been requested, the recv call will block unless there is some data that can be returned to the user. If the connection is cleared (due to normal or abnormal reasons) while recv is blocked, recv will return a count of zero. A return value of zero from recv is an indication that the connection has been cleared, and the user must close the socket at this point.

12.5.2 Reading the M-, D-, and Q-bits

To determine the values of the M-, D-, and Q-bits in received frames, call the X25_HEADER ioctl before the virtual circuit has been created.

```
ints, need_header;
 error = ioctl(s, X25_HEADER, &need_header);
```

If need_header is set to one, subsequent recvs will return the data preceded by a one-byte header that contains the values of the M-, Q-, and D-bits encoded as bit shifts as follows:

```
#define M_BIT 0      /* number of bits to shift for M-bit */
 #define D_BIT 2      /* number of bits to shift for D-bit */
 #define Q_BIT 3      /* number of bits to shift for Q-bit */
```

For example, to check for the presence of the Q-bit in a packet, the following sequence might be used:

```
char buf[1025];
 int s, need_header = 1, count, error;
 error = ioctl(s, X25_HEADER, &need_header);
    . . .
 count = recv(s, buf, sizeof(buf), 0);
 if (count > 0 && (buf[0] & (1 << Q_BIT)))
    /* then Q bit is on */
```

The `X25_HEADER` ioctl must be issued either before the connect call (for outgoing calls), or before the accept call (for incoming calls). For PVCs, the `X25_HEADER` ioctl must be issued before the `X25_SETUP_PVC` ioctl. For the duration of the call, the `X25_HEADER` ioctl must not be used to change the header setting. For example, if a message is received when the header setting is on and the user turns it off before reading the message, the user will receive a one-byte header along with the message, even though he is not expecting it.

If the header is requested, X.25 does not wait for a complete X.25 message to be assembled before returning any data to the user. Rather, partial messages (indicated by the presence of M_BIT) are returned to the user as they become available. Note that the buffer supplied in the recv call must be large enough to accommodate the extra byte of header information.

12.5.3 Receiving X.25 Messages in Records

By default, each recv returns a complete X.25 message. To force recv to return data before a complete X.25 message has been assembled, issue the X25_RECORD_SIZE ioctl after the socket is created:

```
int s, record_size, error;
 /* Set record_size to n, where n is the number of
  * maximum size packets with more bit turned on that
  * will be received before the accumulated data is
  * returned in a recv call.
  */
 error = ioctl(s, X25_RECORD_SIZE, &record_size);
```

Here, record_size specifies the number of full (maximum size) packets with M-bit turned on that X.25 will receive before the accumulated data is returned to the user as a record (or message). Thus, the maximum record size seen by the user will be record_size times the maximum packet size for the virtual circuit. If a complete X.25 message comprises less than record_size packets, it will be returned to the user as in the normal case.

The X25_RECORD_SIZE ioctl is useful when complete X.25 messages are potentially very long, so that either they cannot be buffered in the socket receive buffers (limited by the high water mark), or it is too much of a performance bottleneck for the application to wait for the whole message to be assembled before processing it, or the application does not wish to dedicate very large buffers for receiving data. If record boundaries (that is, message boundaries) are important, this method must not be used. Rather, the X25_HEADER ioctl must be used, as indicated earlier, to obtain a header byte for each packet that indicates whether or not the packet is the last one in a record (that is, message).

12.5.4 Out-of-Band Data

Out-of-band data is managed by a combination of ioctl calls, the passing of the MSG_OOB flag to recv, and an optional signal, SIGURG. To determine whether out-of-band data has been received, call the X25_OOB_TYPE ioctl:

```
ints, oob_type;
 error = ioctl(s, X25_OOB_TYPE, &oob_type);
```

If out-of-band data does not exist, oob_type is set to zero. Otherwise, oob_type is set to a value indicating the type of out-of-band data that has been received. The types of out-of-band data are:

```
#define INT_DATA      30   /* interrupt data */
 #define VC_RESET      32   /* virtual circuit reset */
```

INT_DATA indicates that interrupt data has been received. The interrupt data is read by calling recv with flags set to MSG_OOB. In general, the following sequence occurs upon receipt of an interrupt packet:

1. **X.25 receives an interrupt request packet. The interrupt is queued and causes a `SIGURG` signal.**

2. **The user reads the interrupt packet (with `recv`), automatically causing an Interrupt Confirmation packet to be sent.**

   Up to 32 bytes of interrupt data may be received if the interface supports 1984 X.25.

   It is not necessary to issue a recv call with flags set to MSG_OOB if the interrupt type is something other than INT_DATA.

   VC_RESET indicates that the virtual circuit associated with the socket has been reset.

   The SunNet X.25 7.0 interface had an additional type of out-of-band data, MSG_TOO_LONG, which indicated that a message was discarded because of the socket buffer limitations. This type of out-of-band data does not exist in the current release, because an X.25 message will not get discarded when it gets too long. “Too long” means that too many packets are received with the M-bit set to 1 and the user has not asked for individual packets with the `X25_HEADER` ioctl. Instead of getting discarded, the X.25 message will be sent upstream as soon as its length goes over `MAXNSDULEN`, whether or not the end of the message has been seen (that is, a packet with the M-bit set to 0). `MAXNSDULEN` is one of the configurable Layer 3 parameters described in *Solstice X.25 9.2 Administration Guide*.

   If this happens, there are three possible courses of action that may be taken:

   - Increase the socket high water mark using the X25_WR_SBHIWAT ioctl to a maximum of 32767.
   - Request a header on every packet using the X25_HEADER ioctl. This will result in every packet being returned to the user with an extra header byte.
   - Use the X25_RECORD_SIZE ioctl to specify the maximum number of full packets in a complete X.25 message that X.25 should receive before returning the accumulated data to the user as a record.

     Out-of-band messages are serialized in a FIFO (first in, first out) queue, except for interrupt data, which preempts all other out-of-band messages. If the ioctl call X25_OOB_TYPE indicates INT_DATA, then the interrupt packet will be the next packet read on the out-of-band channel, that is, when recv is called with flags set to MSG_OOB. The INT_DATA condition remains true until the out-of-band packet has been read.

     The following piece of code may be used to set up the function func as the signal handler for the SIGURG signal:

```
int func();
   (void) signal(SIGURG, func);
```

The signal SIGURG, which indicates an urgent condition present on a socket, may be enabled to indicate an abnormal condition or the arrival of abnormal data at an AF_X25 socket. The signal causes func, the signal handler procedure, to be called. The signal procedure must be called before connect on the calling side and listen on the called side.

A process receiving the SIGURG signal must examine all potential causes for the signal in order to identify the source of the signal. For example, if a process has multiple AF_X25 sockets open when it receives the SIGURG signal, each open AF_X25 socket will have to be queried with the X25_OOB_TYPE ioctl to determine the signal source. It could well be that the signal did not originate with X.25, but from some other source.

Upon socket creation, the socket is not associated with a process group ID. If a signal handler routine is used, the user should associate a proper process group ID with the socket as shown below:

```
int pgrp, error;
   pgrp = getpid(); /* get the current process id */
   error = ioctl(s, SIOCSPGRP, &pgrp);
```

When a signal handler routine is awakened, pending system calls, for example, recv, accept, connect, select, etc., will be aborted with errno set to EINTR (interrupted system call). The signal handler routine func may be disabled at any time by assigning a default action SIG_DFL to SIGURG:

```
(void) signal(SIGURG, SIG_DFL);
```

A more general explanation of signals is in the SunOS 4.x documentation on socket programming.

12.6 Clearing a Virtual Circuit

The close system call is used to discontinue use of a socket and all of the resources held by the socket, as follows:

```
int s, error;
 error = close(s);
```

The close call closes the virtual circuit associated with a socket and frees the resources used by the socket. More specifically, close will send a Clear Request packet and then wait for a Clear Confirmation packet if the socket has an active virtual circuit associated with it. An active virtual circuit is one that is either connected, or is in the early stages of connection (that is, Call Request has been sent, but Call Connected has not been received). In this case, if a Clear Confirmation packet is not received after the amount of time specified in the link configuration file, the socket will be closed and close will return. If the socket does not have an active virtual circuit associated with it, close will return immediately.

12.7 Advanced Topics

This section includes material on a variety of advanced topics.

12.7.1 Facility Specification and Negotiation

X.25 user facilities are specified on a per-call basis. The X25_SET_FACILITY ioctl is used to set facilities one at a time. The X25_GET_FACILITY ioctl is used to read facilities one at a time. These ioctl commands support all facilities (1980 and 1984 X.25).

Facilities are set in two places: before issuing a connect call, in order to request desired facilities in the Call Request packet; and before issuing a listen call, in order to negotiate the facilities proposed in an Incoming Call packet.

Facilities are usually read in two places: after a call to connect has succeeded, and after a call to accept has succeeded. This is done to determine the values of the facilities in effect for the resulting connection. Facilities can be read at any time, in general, to determine values which were previously set.

12.7.2 `X25_SET_FACILITY`/`X25_GET_FACILITY` ioctls

Note - The sockets-based interface provides access only to those facilities that were supported in SunNet X.25 7.0. These are a subset of the facilities supported in Solstice X.25 9.2.

The X25_SET_FACILITY ioctl command is used to set the following facilities:

```
reverse charge   (*)   (#)
 fast select   (*)   (#)
 non-default packet size   (*)
 non-default window size   (*)
 non-default throughput   (*)
 minimum throughput class   (#)
 closed user group   (*)   (#)
 RPOA selection   (*)   (#)
 network transit delay   (#)
```

```
end-to-end transit delay
network user identification   (#)
charging information request
expedited data negotiation
called AEF
calling AEF   (#)
non-X.25 facilities
```

All of the above facilities can be sent in a Call Request packet. The ones that can be used with a 1980 X.25 interface are marked with an (*), although only the basic forms of the closed user group facility and the RPOA selection can be used in this case. The ones that cannot be sent in a Call Accepted packet are marked with a (#). Solstice X.25 does not permit users to set facilities in Clear Request and Clear Confirm packets.

All of the above facilities can be read using the X25_GET_FACILITY ioctl command. In addition, the following can also be read:

```
charging information, monetary unit
 charging information, segment
 charging information, call duration
 called line address modified notification
 call redirection notification
```

Sample programs provided with Solstice X.25 illustrate the use of these facilities. Here, we discuss each of the above facilities in more detail and provide code segments to illustrate their use. For convenience, the variables used in the discussion below are declared here. (Chapter 13" has a listing of the relevant data structures used by the X25_SET_FACILITY and X25_GET_FACILITY ioctl commands.)

```
FACILITY    f;   /* facility structure */
 int s;           /* socket */
 int    error;    /* ioctl return value */
```

For brevity, the value returned by ioctl calls is not checked for error.

In the discussion that follows, we show how the user can send facilities in the Call Request packet. In order to send a facility in the Call Accepted packet, the listener should either set the facility before invoking listen, or should set it before causing the Call Accepted packet to be sent (that is, the listener should have used the X25_CALL_ACPT_APPROVAL ioctl command, described later, to cause Solstice X.25 to permit call approval by the user).

The exceptions to this are end-to-end transit delay, expedited data negotiation, Called AEF, and non-X.25 facilities. To send these in the Call Accepted packet, the listener must do call approval, and must set these facilities after accept returns, but before the X25_SEND_CALL_ACPT ioctl command is used to send the Call Accepted packet.

12.7.2.1 Reverse Charge

There are two possible values for this facility: 1 indicates reverse charging, and 0 indicates no reverse charging.

This is set as follows:

```
u_char    reverse_charge;
 reverse_charge = 1;
 f.type = T_REVERSE_CHARGE;
 f.f_reverse_charge = reverse_charge;
 error = ioctl(s, X25_SET_FACILITY, &f);
```

This facility is read as follows:

```
f.type = T_REVERSE_CHARGE;
 error = ioctl(s, X25_GET_FACILITY, &f);
 reverse_charge = f.f_reverse_charge;
```

Setting this facility before making the connect call causes this facility to be sent in the Call Request. Setting this facility before making the listen call causes Incoming Calls with the reverse charging facility to be accepted. (Calls that are not reverse-charged are always acceptable.) The listener should read the value of the facility after the accept call returns to find out if the call is reverse-charged.

Note - Reverse charging must be allowed for this ioctl to work. You allow for reverse charging in the `x25tool` CUG and Facilities window. To access the CUG and Facilities window, from the `x25tool` Link Editor window, select CUG and Facilities. Click on Incoming Reverse Charging. See *Solstice X.25 9.2 Administration Guide* for further details.

12.7.2.2 Fast Select

There are three possible values for this facility. FAST_OFF indicates that fast select is not in effect. FAST_CLR_ONLY indicates fast select with restriction on response, and FAST_ACPT_CLR indicates fast select with no restriction on response.

This is set as follows:

```
u_char fast_select_type;
 fast_select_type = FAST_CLR_ONLY;
 f.type = T_FAST_SELECT_TYPE;
 f.f_fast_select_type = fast_select_type;
 error = ioctl(s, X25_SET_FACILITY, &f);
```

This is read as follows:

```
f.type = T_FAST_SELECT_TYPE;
 error = ioctl(s, X25_GET_FACILITY, &f);
 fast_select_type = f.f_fast_select_type;
```

If this facility is set before making the connect call, the Call Request packet is sent out with this facility. If this facility is set before making the listen call, the behavior that follows will depend on whether or not restriction on response was indicated, and on whether the Incoming Call has this facility. In order for an Incoming Call bearing the fast select facility to be acceptable, the listener should have specified fast select (with or without restriction). However, an Incoming Call not bearing the fast select facility will still be acceptable to a listener who has specified fast select with no restriction on response. The type of fast select in effect will be either the type of fast select in the Incoming Call, or fast select with restriction on response if either end of the connection has specified fast select with restriction on response. If the Incoming Call does not specify fast select, and is accepted by a listener who has specified fast select with no restriction on response, fast select will not be in effect for the duration of the call.

A listener that has specified fast select (with or without restriction) must use the X25_SEND_CALL_ACPT ioctl to accept the call or use close to clear the call, after successful completion of the accept call, regardless of whether fast select is in effect for the call. If the type of fast select in effect after accept is either FAST_OFF or FAST_ACPT_CLR, the user may either accept or clear the call. If the type of fast select in effect is FAST_CLR_ONLY, the user cannot accept the call (it can only be cleared). The handling of user data in conjunction with fast select is described later.

12.7.2.3 Packet Size

Packet size is set in the Call Request packet as follows:

```
u_short sendpktsize, recvpktsize;
 /* set sendpktsize, recvpktsize to desired values */
 f.type = T_PACKET_SIZE;
 f.f_sendpktsize = sendpktsize;
 f.f_recvpktsize = recvpktsize;
 error = ioctl(s, X25_SET_FACILITY, &f);
```

It is read as follows:

```
f.type = T_PACKET_SIZE;
 error = ioctl(s, X25_GET_FACILITY, &f);
 sendpktsize = f.f_sendpktsize;
 recvpktsize = f.f_recvpktsize;
```

Setting packet size in the Call Request causes the values set to be proposed for the call (a zero value indicates the default for the link). Reading the value after the call is set up yields the result of negotiation.

Packet sizes are set and read in bytes, so that, for example, 128, 256, and 512 are legal values.

12.7.2.4 Window Size

Window size is set in the Call Request packet as follows:

```
u_short   sendwndsize, recvwndsize;
 /* set sendwndsize, recvwndsize to desired values */
 f.type = T_WINDOW_SIZE;
 f.f_sendwndsize = sendwndsize;
 f.f_recvwndsize = recvwndsize;
```

```
 error = ioctl(s, X25_SET_FACILITY, &f);
```

It is read as follows:

```
f.type = T_WINDOW_SIZE;
 error = ioctl(s, X25_GET_FACILITY, &f);
 sendwndsize = f.f_sendwndsize;
 recvwndsize = f.f_recvwndsize;
```

Setting the window size in the Call Request causes the values set to be proposed for the call (a zero value indicates the default for the link). Reading the value after the call is set up yields the result of negotiation.

12.7.2.5 Throughput

Throughput is set in the Call Request packet as follows:

```
u_char      sendthruput, recvthruput;
 /* set sendthruput, recvthruput to desired values */
 f.type = T_THROUGHPUT;
 f.f_sendthruput = sendthruput;
 f.f_recvthruput = recvthruput;
 error = ioctl(s, X25_SET_FACILITY, &f);
```

It is read as follows:

```
f.type = T_THROUGHPUT;
 error = ioctl(s, X25_GET_FACILITY, &f);
 sendthruput = f.f_sendthruput;
 recvthruput = f.f_recvthruput;
```

When throughput is set in the Call Request, the values set are proposed for the call (a zero value indicates the default for the link). Reading the value after the call is set up yields the result of negotiation.

12.7.2.6 Minimum Throughput Class

Minimum throughput class is set in the Call Request packet as follows:

```
u_char      min_sendthruput, min_recvthruput;
 /* set min_sendthruput, min_recvthruput to desired values */
 f.type = T_MIN_THRU_CLASS;
 f.f_min_sendthruput = min_sendthruput;
 f.f_min_recvthruput = min_recvthruput;
 error = ioctl(s, X25_SET_FACILITY, &f);
```

It is read as follows:

```
f.type = T_MIN_THRU_CLASS;
 error = ioctl(s, X25_GET_FACILITY, &f);
 min_sendthruput = f.f_min_sendthruput;
 min_recvthruput = f.f_min_recvthruput;
```

This facility may only be set in a Call Request packet, and read from an Incoming Call packet. The receiver of the Incoming Call packet should clear the call (with an appropriate diagnostic) if the proposed minimum throughput values cannot be supported.

12.7.2.7 Closed User Group

The user may set one of three types of Closed User Group facility: CUG_REQ (no outgoing access), CUG_REQ_ACS (with outgoing access), and CUG_BI (bilateral CUG). For CUG_REQ and CUG_REQ_ACS, the CUG is a decimal integer in the range 0-9999 (for 1980 X.25 interfaces, the valid range is 0-99). The extended form of the facility is used for CUG indices in the range 100-9999. This facility is set as follows:

```
u_short      cug_index;
 /* set cug_index to appropriate value */
 f.type = T_CUG;
 f.f_cug_req = CUG_REQ;    /* could be CUG_REQ_ACS or CUG_BI */
 f.f_cug_index = cug_index;
 error = ioctl(s, X25_SET_FACILITY, &f);
```

To read this facility:

```
f.type = T_CUG;
 error = ioctl(s, X25_GET_FACILITY, &f);
 cug_req = f.f_cug_req;
 cug_index = f.f_cug_index;
```

12.7.2.8 RPOA Selection

Solstice X.25 supports the setting of up to three (MAX_RPOA) RPOA transit networks (in the extended form). If only one is specified, the non-extended form of the facility is used. An RPOA transit network is specified as a decimal integer in the range 0-9999.

This facility is set as follows:

```
u_short      rpoa0, rpoa1, rpoa2;
 /* set rpoa0, rpoa1, rpoa2 */
 f.type = T_RPOA;
 f.f_nrpoa = 3;
 f.f_rpoa_index[0] = rpoa0;
 f.f_rpoa_index[1] = rpoa1;
 f.f_rpoa_index[2] = rpoa2;
 error = ioctl(s, X25_SET_FACILITY, &f);
```

To read this facility:

```
f.type = T_RPOA;
 error = ioctl(s, X25_GET_FACILITY, &f);
 rpoa0 = f.f_rpoa_index[0];
 rpoa1 = f.f_rpoa_index[1];
 rpoa2 = f.f_rpoa_index[2];
```

12.7.2.9 Network Transit Delay

The Transit Delay Selection and Indication facility (TDSAI) is set in the Call Request as follows:

```
u_short    tr_delay;   /* desired transit delay in milliseconds */
 /* set tr_delay */
 f.type = T_TR_DELAY;
 f.f_tr_delay = tr_delay;
 error = ioctl(s, X25_SET_FACILITY, &f);
```

This is read as follows:

```
f.type = T_TR_DELAY;
 error = ioctl(s, X25_GET_FACILITY, &f);
 tr_delay = f.f_tr_delay;
```

12.7.2.10 End-to-End Transit Delay

This is set in the Call Request as follows:

```
u_short   req_delay, desired_delay, max_delay;
 /* set the requested, desired, and maximum delays */
 f.type = T_ETE_TR_DELAY;
 f.f_req_delay = req_delay;
 f.f_desired_delay = desired_delay;
 f.f_max_delay = max_delay;
 error = ioctl(s, X25_SET_FACILITY, &f);
```

This is read as follows:

```
f.type = T_ETE_TR_DELAY;
 error = ioctl(s, X25_GET_FACILITY, &f);
 req_delay = f.f_req_delay;
 desired_delay = f.f_desired_delay;
 max_delay = f.f_max_delay;
```

If f_desired_delay is set, f_req_delay must be non-zero; if f_max_delay is set, f_desired_delay must be non-zero. Delay is specified in milliseconds.

12.7.2.11 Network User Identification

This is set as follows (in the example below, NUI is an ASCII string):

```
char   nui_str[] = "sunhost";
 f.type = T_NUI;
 f.f_nui.nui_len = strlen(nui_str);
 bcopy(nui_str, f.f_nui.nui_data, strlen(nui_str));
 error = ioctl(s, X25_SET_FACILITY, &f);
```

Solstice X.25 permits a maximum length of 64 (MAX_NUI) for Network User Identification facility.

To read this facility:

```
f.type = T_NUI;
 error = ioctl(s, X25_GET_FACILITY, &f);
 nui_str = f.f_nui.nui_data;
```

12.7.2.12 Charging Information Request

This write-only facility is set as follows:

```
f.type = T_CHARGE_REQ;
 f.f_charge_req = 1;
 error = ioctl(s, X25_SET_FACILITY, &f);
```

12.7.2.13 Charging Information

By setting `f.type` to T_CHARGE_REQ as specified above you make available the following read-only facilities. The facility types are T_CHARGE_MU, T_CHARGE_SEG, and T_CHARGE_DUR. For example, the Charging Information (monetary unit) is read as follows:

```
typedef struct charge_info_s {
    u_char   charge_len;
 #define MAX_CHARGE_INFO   64
    u_char   charge_data[MAX_CHARGE_INFO];
 } CHARGE_INFO;

 CHARGE_INFO charge_mu;
 f.type = T_CHARGE_MU;
 error = ioctl(s, X25_GET_FACILITY, &f);
 charge_mu = f.f_charge_mu;
```

The `T_CHARGE_SEG` and `T_CHARGE_DUR` facilities are read in a way similar to the `T_CHARGE_MU` example above; that is, by using `T_CHARGE_SEG` or `T_CHARGE_DUR` for the `f.type` value, and using `f_charge_seg` or `f_charge_dur` in place of `f_charge_mu`.

The maximum length for the charging information facility permitted by Solstice X.25 is 64 (`MAX_CHARGE_INFO`). This facility should be read after the call is cleared, but before the socket is closed, since it is received in the Clear Request or Clear Confirm packets.

12.7.2.14 Called Line Address Modified Notification

This is a read-only facility received in either the Call Accepted or Clear Indication packets. It is read as follows:

```
u_char   line_addr_mod;
 f.type = T_LINE_ADDR_MOD;
 error = ioctl(s, X25_GET_FACILITY, &f);
 line_addr_mod = f.f_line_addr_mod;
```

12.7.2.15 Call Redirection Notification

This is a read-only facility received in either the Call Accepted or Clear Indication packets. It is read as follows:

```
typedef struct call_redir_s {
    u_char   cr_reason;
    u_char   cr_hostlen;
    u_char   cr_host[(MAXHOSTADR+1)/2];
 } CALL_REDIR;

 CALL_REDIR call_redir;
 f.type = T_CALL_REDIR;
 error = ioctl(s, X25_GET_FACILITY, &f);
 call_redir = f.f_call_redir;
```

12.7.2.16 Expedited Data Negotiation

This facility is set as follows:

```
u_char expedited = 1;/* 0 indicates non-use of expedited data */
 f.type = T_EXPEDITED;
 f.f_expedited = expedited;
 error = ioctl(s, X25_SET_FACILITY, &f);
```

It is read as follows:

```
f.type = T_EXPEDITED;
 error = ioctl(s, X25_GET_FACILITY, &f);
 expedited = f.f_expedited;
```

12.7.2.17 Called / Calling AEF

There are three types of address extensions: OSI NSAP (AEF_NSAP), Partial OSI (AEF_PARTIAL_NSAP), and Non-OSI (AEF_NON_OSI). The Calling AEF may only be present in the Call Request packet.

Solstice X.25 9.2 Administration Guide describes how Solstice X.25 may be set up to automatically supply the Calling AEF (referred to as address extension) in a Call Request packet.

The Called AEF is set as follows:

```
typedef struct aef_s {
    u_char   aef_type;
 #define AEF_NONE      0
 #define AEF_NSAP      1
 #define AEF_PARTIAL_NSAP   2
 #define AEF_NON_OSI      3
    u_char   aef_len;
 #define MAX_AEF   40
    u_char   aef[(MAX_AEF+1)/2];
 } AEF;

 AEF   aef;
 aef.aef_type = AEF_NON_OSI;
 aef.aef_len = 7;        /* length in nibbles */
 aef.aef[0] = 0x12;
```

```
 aef.aef[1] = 0x34;
 aef.aef[2] = 0x56;
 aef.aef[3] = 0x70;      /* Note, unused nibble is zero */
 f.type = T_CALLED_AEF;
 f_called_aef = aef;
 error = ioctl(s, X25_SET_FACILITY, &f);
```

The Called AEF is read as follows:

```
f.type = T_CALLED_AEF;
 error = ioctl(s, X25_GET_FACILITY, &f);
 aef = f_called_aef;
```

The Calling AEF is set and read similarly (using T_CALLING_AEF in place of T_CALLED_AEF and f_calling_aef in place of f_called_aef).

12.7.2.18 Non-X.25 Facilities

These are for expert use only. Solstice X.25 permits a maximum of 64 (MAX_PRIVATE) bytes of non-X.25 facilities. These are not looked at by Solstice X.25, but just passed through. Non-X.25 facilities consist of a sequence of facility blocks, where each block begins with a facility marker indicating non-X.25 facilities supported by either the local or remote network, or some arbitrary facility marker. This is set as follows:

```
typedef struct private_fact_s {
    u_char   p_len;   /* total length of facilities*/
 #define MAX_PRIVATE  64
    u_char   p_fact[MAX_PRIVATE];
       /* facilities exactly as they
        * are present in Call Request or
        * Call Accept packets
        */
 } PRIVATE_FACT;

 PRIVATE_FACT  private;
 /* set the p_len and p_fact fields */
 f.type = T_PRIVATE;
 f.f_private = private;
 error = ioctl(s, X25_SET_FACILITY, &f);
```

It is read as follows:

```
f.type = T_PRIVATE;
 error = ioctl(s, X25_GET_FACILITY, &f);
 private = f.f_private;
```

12.7.2.19 Determining Which Facilities are Present

Since facilities can be read only one at a time, the user needs a way to determine which facilities are present. Solstice X.25 provides the following mechanism for doing this.

The user can read a bit mask that has one bit reserved for each of the facilities described above. This is read as:

```
u_int     fmask;
 f.type = T_FACILITIES;
 error = ioctl(s, X25_GET_FACILITY, &f);
 fmask = f.f_facilities;
```

The following mask bits are defined:

```
F_REVERSE_CHARGE    /* reverse charging */
 F_FAST_SELECT_TYPE  /* fast select */
 F_PACKET_SIZE       /* packet size */
 F_WINDOW_SIZE       /* window size */
 F_THROUGHPUT        /* throughput */
 F_MIN_THRU_CLASS    /* minimum throughput class */
 F_CUG               /* closed user group selection */
 F_RPOA              /* ROPA transit network */
 F_TR_DELAY          /* network transit delay */
 F_ETE_TR_DELAY      /* end to end transit delay */
 F_NUI               /* network user identification */
 F_CHARGE_REQ        /* charging information request */
 F_CHARGE_MU         /* charging information, monetary unit */
 F_CHARGE_SEG        /* charging information, segment */
 F_CHARGE_DUR        /* charging information, call duration */
 F_LINE_ADDR_MOD     /* called line address modified notification */
 F_CALL_REDIR        /* call redirection notification */
 F_EXPEDITED         /* expedited data negotiation */
 F_CALLED_AEF        /* called AEF */
 F_CALLING_AEF       /* calling AEF */
 F_PRIVATE           /* non-X.25 facilities */
```

For example, to determine if the Call Redirection facility has been received, the following segment of code could be used:

```
if ((fmask & F_CALL_REDIR) != 0) {
 /*
  * Read its value.
  */
 CALL_REDIR call_redir;
 f.type = T_CALL_REDIR;
 error = ioctl(s, X25_GET_FACILITY, &f);
 call_redir = f.f_call_redir;
 }
```

12.7.3 Fast Select User Data

The fast select facility is handled in the following way.

12.7.3.1 Calling Side

To send fast select data, `fast_select_type` must be set to the proper value (with the `X25_SET_FACILITY` ioctl) before `connect` is called (see the section Section 12.7.1 “Facility Specification and Negotiation” on page 217of this chapter for more information). Using the `CONN_DB` structure, a calling DTE can specify a user data field up to 102 bytes (including the optional protocol identifier). If 102 bytes of call user data are not enough for the current fast select message, use the X25_WR_USER_DATA ioctl before calling connect to pass the additional user data. The user data specified in connect will precede this additional user data. To write user data:

```
typedef struct user_data_db_s {
 u_char      datalen;
 u_char       data[MAX_USER_DATA];
 } USER_DATA_DB;
 int s, error;
 USER_DATA_DB user_data;
 error = ioctl(s, X25_WR_USER_DATA, &user_data)
```

Here, MAX_USER_DATA is 124.

If connect returns –1 and errno is EFASTDATA, the remote side has cleared the call by sending a Clear Indication packet with up to 32 bytes (1980) or 128 bytes (1984) of user data. At this time, the user can read the user data in the Clear Indication packet with calls to the X25_RD_USER_DATA ioctl until the returned datalen in USER_DATA_DB structure is 0 or less than MAX_USER_DATA, then close the socket with close.

To read user data:

```
USER_DATA_DB user_data;
 int s, error;
 error = ioctl(s, X25_RD_USER_DATA, &user_data);
```

If connect returns 0, it indicates that the connection has been set up successfully. If the connection is over an interface that supports 1984 X.25, the remote user may have sent user data in the Call Accepted packet. (This will happen only if the initiator of the connection has specified fast select with no restriction on response.) Thus the initiating user must repeatedly read any user data using the X25_RD_USER_DATA ioctl until the returned length in the USER_DATA_DB structure is less than MAX_USER_DATA.

When a call is cleared after being connected, the Clear Indication packet may contain user data if the interface supports 1984 X.25 and fast select is in effect for that call. Either the initiator of the connection or the responder can send user data in the Clear Request packet. Thus when a call with fast select is cleared by the remote user, user data must be read in the same way as for the other cases.

For 1980 X.25 interfaces, if the connection was accepted by the remote user, the Call Accepted and Clear Request packets will not have any user data; the only time that the Clear Request can have user data is when a fast select call is cleared immediately (this is detectable by means of the EFASTDATA error return).

12.7.3.2 Called Side

To receive a fast select incoming call, the called side must specify either FAST_ACPT_CLR or FAST_CLR_ONLY as the value for fast_select_type using the X25_SET_FACILITY ioctl, before issuing the listen call.

If the Incoming Call has the fast select facility, it will be accepted only if the listener has specified fast select. The incoming call will also be accepted if it does not have the fast select facility and the listener has specified FAST_ACPT_CLR.

The call will be rejected if there are more than 16 bytes of user data, and the called side has either not specified the fast select facility at all, or has specified FAST_OFF (which is equivalent to not specifying fast select).

After accept returns, the called side may use the X25_GET_FACILITY ioctl to determine the type of fast select in effect. For example, if the called side has specified FAST_ACPT_CLR and the calling side has specified FAST_CLR_ONLY, after accept returns, the type of fast select in effect will be FAST_CLR_ONLY. If fast select is indicated, the called side can read the user data that was received in the Call Request by looking at the CONN_DB structure returned by accept. If more than 102 bytes of user data were received, the extra bytes can be read with the X25_RD_USER_DATA ioctl.

The X25_WR_USER_DATA ioctl can be used to specify user data to be sent back in the response to the fast select Call Request. To write more than MAX_USER_DATA bytes of user data, a second X25_WR_USER_DATA ioctl can be used to append the additional data after that from the first X25_WR_USER_DATA ioctl (total length of all user data may not exceed 128 bytes).

If the type of fast select in effect is FAST_CLR_ONLY, the called side can only clear the fast select call by closing the socket (which causes the user data specified by X25_WR_USER_DATA to be sent in the Clear Request). If the type of fast select in effect after accept returns is FAST_ACPT_CLR, the called side has the option, after writing the reply message with the X25_WR_USER_DATA ioctl, of either sending a Clear Request packet with close or sending a Call Accepted packet with the X25_SEND_CALL_ACPT ioctl and thereby entering the normal data transfer state.

```
int news, error;
 error = ioctl(news, X25_SEND_CALL_ACPT);
```

When the value in effect is FAST_CLR_ONLY, the called side can only close the socket with the close system call after writing the reply message.

FAST_OFF is the type of fast select that will be in effect when the listener has specified FAST_ACPT_CLR and the incoming call does not have the fast select facility. Even in this case, the listener must use the X25_SEND_CALL_ACPT ioctl to put the connection into normal data transfer state.

Note - In the current release (and not in SunNet X.25 7.0), the listen socket should not be closed until after the incoming fast select call has been either cleared (with `close`) or accepted (with `X25_SEND_CALL_ACPT`).

12.7.4 Permanent Virtual Circuits

Since permanent virtual circuits are always in data transfer state, there is no need to issue a connect on the calling side, or bind, listen, and accept on the called side. Instead, use an ioctl call to bind the socket to a logical channel number and to specify other parameters.

```
typedef struct pvc_db_s {
 u_short lcn; /* lcn of PVC */
 u_short sendpktsize; /* Maximum packet size */
 u_short recvpktsize; /* Maximum packet size */
 u_char sendwndsize;  /* Output flow control window */
 u_char recvwndsize;  /* Input flow control window */
 } X25_PVC_DB;
 X25_PVC_DB pvc_parms;
 int pvc_so;
 pvc_so = socket(AF_X25, SOCK_STREAM, 0);
 error = ioctl(pvc_so, X25_SETUP_PVC, &pvc_parms);
```

In the current release, the `sendpktsize`, `recvpktsize`, `sendwndsize`, and recvwndsize parameters are ignored. The default value in the link configuration file is always used. By default, the lowest numbered WAN link is used for the permanent virtual circuit. If you desire some other link for the permanent virtual circuit, you must select the desired link using the X25_SET_LINK ioctl as described earlier, after the socket call, but before the X25_SETUP_PVC ioctl. Permanent virtual circuits are not supported over LAN interfaces.

12.7.5 Call Acceptance by User

Normally Incoming Call packets are examined and responded to by X.25. If the call is accepted, a Call Accepted packet is sent by X.25 directly. In the event a user process wants to have additional checks before sending a Call Accepted packet, an X25_CALL_ACPT_APPROVAL ioctl may be used.

```
int approved_by_user, s, error;
 error = ioctl(s, X25_CALL_ACPT_APPROVAL, &approved_by_user);
```

where approved_by_user = 0 means the approval is done by X.25, and approved_by_user = 1 means approval is done by the user process. By default (that is, if this call is not issued), approval is done by X.25. Note that if a user wants to do call approval, the X25_CALL_ACPT_APPROVAL ioctl must be issued before the listen call is issued.

Regardless of the value of approved_by_user, X.25 always performs address matching and facilities negotiation before notifying accept. If a user process assumes the final incoming call approval, accept will return without sending a Call Accepted packet. At this time, the user process should reply as soon as possible to avoid the Call Request timeout on the remote calling side. To accept the call, use:

```
int news, error;
 error = ioctl(news, X25_SEND_CALL_ACPT);
```

Here, news is the socket descriptor returned by accept.

The X25_SEND_CALL_ACPT ioctl call is also needed for fast select calls, as described in an earlier section. To reject the call, simply close the socket:

```
int news;
 close(news);
```

where news is the socket descriptor returned by accept.

12.7.6 Accessing the Link (X.25) Address

The X.25 client can set the local link X.121 (X.25) address through an X.25 socket owned by the superuser. (The default value is established in the Interface Configuration window in `x25tool`, as described in *Solstice X.25 9.2 Administration Guide*):

```
typedef struct link_adr_s {
    int      linkid;  /* id of link */
    u_char   hostlen; /* length of BCDs */
    u_char   host[(MAXHOSTADR+1)/2];
 } LINK_ADR;
 LINK_ADR addr;
 int so, error;
 error = ioctl(so, X25_WR_LINKADR, &addr);
```

Set linkid to the identifier of the desired link.

The local link X.121 address can be read at any time with:

```
LINK_ADR addr;
 int s;
 error = ioctl(s, X25_RD_LINKADR, &addr);
```

The returned addr is actually the link address specified in `x25tool` (for the link specified in the linkid field of the LINK_ADR structure) unless a new address has been assigned to the link.

The X25_WR_LINKADR ioctl can be used to assign new X.25 addresses to a link.

12.7.7 Accessing High Water Marks of Socket

The AF_X25 socket provides a flow control mechanism using high and low water marks on both the send and receive sides of an X.25 virtual circuit. When the amount of queued data goes above the high water mark, additional data is blocked until the queued data falls below the low water mark. Blocking received data is accomplished by not acknowledging receipt of packets until the user reads the data. Blocking send data is accomplished by blocking the user process invoking send or `write`.

The default high water mark for both sending and receiving is 2048 bytes. The low water mark is always set to half the high water mark. Note that the high water mark is only an approximation of the maximum amount of data allowed to be queued up.

A user process may set or read the high water mark as described below. To read:

```
typedef struct so_hiwat_db_s {
    short   sendhiwat;
    short   recvhiwat;
 } SO_HIWAT_DB;
 SO_HIWAT_DB hiwater;
 int s, error;
 error = ioctl(s, X25_RD_SBHIWAT, &hiwater);
```

To write:

```
error = ioctl(s, X25_WR_SBHIWAT, &hiwater);
```

12.7.8 Accessing the Diagnostic Code

The user may read the cause or diagnostic code in a Clear Indication or Reset Indication packet received from the remote end. The user may also write the cause or diagnostic code in Clear Request and Reset Request packets to be transmitted to the remote end.

```
typedef struct x25_cause_diag_s {
 u_char   flags;
 #   define RECV_DIAG    0
 #   define DIAG_TYPE   1
 #   define WAIT_CONFIRMATION 2
 /*  bit 0 (RECV_DIAG)=
  *  0: no cause and diagnostic codes
  *  1: receive cause and diagnostic codes.
  *  bit 1 (DIAG_TYPE)=
  *  0: reset cause and diagnostic codes in data array
  *  1: clear cause and diagnostic codes in data array
  *  bit 2 (WAIT_CONFIRMATION)=
  *  0: no wait after X25_WR_DIAG_CODE ioctl
  *  1: wait returned cause and diagnostic codes after
  *  X25_WR_DIAG_CODE ioctl.
  */
    u_char   datalen; /* byte count of data array */
    u_char   data[64];
 } X25_CAUSE_DIAG;
 X25_CAUSE_DIAG diag;
 int s, error;
```

To read:

```
error = ioctl(s, X25_RD_CAUSE_DIAG, &diag);
```

To write:

```
error = ioctl(s, X25_WR_CAUSE_DIAG, &diag);
```

The data field in X25_CAUSE_DIAG contains the cause and diagnostic code.

Upon receiving a Clear Indication or Reset Indication packet, the X25_RD_CAUSE_DIAG ioctl may be issued to determine the cause and diagnostic associated with the packet. The datalen field contains the length in bytes of the information in data. When reading the diagnostic, if bit RECV_DIAG (that is, bit 0) is set, it indicates that the information in data is valid. If bit DIAG_TYPE (that is, bit 1) is set, it indicates that the diagnostic was received in a Clear Indication; otherwise, it was received in a Reset Indication.

The X25_WR_CAUSE_DIAG ioctl enables the user to send a Clear Request or Reset Request packet with the desired cause and diagnostic codes. If the user supplies only one byte in the data field, X.25 will use the cause code DTE_ORIGINATED, and use the provided byte as the diagnostic.

The X25_WR_CAUSE_DIAG ioctl call will send a Clear Request or Reset Request. To send a Clear Request, set bit DIAG_TYPE (that is, bit 1) in flags:

```
X25_CAUSE_DIAG diag;
 int s, error;
 diag.flags = 1 << DIAG_TYPE; /* Clear Request */
 diag.datalen = 2;
 diag.data[0] = 0;
 diag.data[1] = 67;
 error = ioctl(s, X25_WR_CAUSE_DIAG, &diag);
```

To send a Clear Request and wait for confirmation, set bit WAIT_CONFIRMATION (that is, bit 2) in flags:

```
X25_CAUSE_DIAG diag;
 int s, error;
 diag.flags = (1 << DIAG_TYPE) | (1 << WAIT_CONFIRMATION);
 diag.datalen = 2;
 diag.data[0] = 0;
 diag.data[1] = 67;
 error = ioctl(s, X25_WR_CAUSE_DIAG, &diag);
```

To send a Reset Request and wait for confirmation:

```
X25_CAUSE_DIAG diag;
 int s, error;
 diag.flags = 1 << WAIT_CONFIRMATION;
 diag.datalen = 2;
 diag.data[0] = 0;
 diag.data[1] = 0;   /* can be any valid diagnostic */
 error = ioctl(s, X25_WR_CAUSE_DIAG, &diag);
```

A `close` is still necessary to free all resources held by this socket and the associated virtual circuit after a Clear Indication or Clear Confirmation packet is received. After the DTE receives a Clear Indication packet, `recv` will return zero bytes after all unread data has been read. Calling `send` after the Clear Indication packet is received will *not* return an error. Note that this behavior is different from that of SunNet X.25 7.0, in which `send` *does* return an error.

To be notified when a Clear Indication packet is received, so that you can use the `X25_RD_CAUSE_DIAG` ioctl, you can use the following mechanism: Enable a third type of out-of-band data (see Section 12.5.4 “Out-of-Band Data ” on page 214) and receive the SIGURG signal when this type of out-of-band data arrives. To enable the signalling of Clear Indication packets, use the following ioctl:

```
error = ioctl(s, X25_OOB_ON_CLEAR, 0);
```

This will enable the reception of the following type of out-of-band data, which can be read with the X25_OOB_TYPE ioctl:

```
#define VC_CLEARED 31  /* virtual circuit cleared */
```

See Section 12.5.4 “Out-of-Band Data ” on page 214 for a complete description of how to handle out-of-band data.

Note - If an X25_WR_CAUSE_DIAG ioctl is not issued before close, X.25 fills an appropriate cause and diagnostic code in any Clear Request packet sent as a result (this will not happen if the connection is inactive at the time the call is issued).

12.8 Routing ioctls

In this section, we describe the ioctls used to manage the Solstice X.25 routing function in the sockets-based interface. The Solstice X.25 routing function is described in detail in *Solstice X.25 9.2 Administration Guide*. The data structure used for routing is as follows:

```
typedef struct x25_route_s {
   uint32_t   index;
   u_char    r_type;
 #define     R_NONE        0
 #define     R_X121_HOST   1
 #define     R_X121_PREFIX 2
 #define     R_AEF_HOST    3
 #define     R_AEF_PREFIX  4
   CONN_ADR  x121;
   u_char    pid_len;
 #define     MAX_PID_LEN   4
   u_char    pid[MAX_PID_LEN];
   AEF       aef;
   int       linkid;
   X25_MACADDR  mac;
   int       use_count;
   char      reserved[16];
 } X25_ROUTE;
```

The following declarations will be used in the code segments used for illustration:

```
int s, error;
 X25_ROUTE r;
```

To add a route, set the fields in the X25_ROUTE structure to desired values, and execute the X25_ADD_ROUTE ioctl as follows:

```
error = ioctl(s, X25_ADD_ROUTE, &r);
```

To obtain the routing information for a given destination address, set the destination address in the X25_ROUTE structure and execute the X25_GET_ROUTE ioctl:

```
error = ioctl(s, X25_GET_ROUTE, &r);
```

To remove a route for a given destination address, set the destination address in the X25_ROUTE structure and execute the X25_RM_ROUTE ioctl:

```
error = ioctl(s, N_X25_RM_ROUTE, &r);
```

To flush all routes out, execute the X25_FLUSH_ROUTES ioctl:

```
error = ioctl(s, X25_FLUSH_ROUTES);
```

The following code segment illustrates how one may cycle through all the routes configured in the system and obtain the parameters for each of them:

```
r.index = 0;
 do {
     error = ioctl(s, X25_GET_NEXT_ROUTE, &r);
     if (error == 0)
         /* print the route */;
 while (error == 0);
```

When there are no routes left, error will be -1, and errno will be set to ENOENT.

The X25_ADD_ROUTE, X25_RM_ROUTE, and X25_FLUSH_ROUTES ioctls require superuser privilege; X25_GET_ROUTE and X25_GET_NEXT_ROUTE do not.

12.9 Miscellaneous ioctls

This section describes some miscellaneous ioctl calls that were either not covered in the previous sections, or are supported from previous releases for backward compatibility. This does not imply backward compatibility with all user-written software for previous releases of Solstice X.25.

12.9.1 Obtaining Statistics

Use the X25_GET_NLINKS ioctl to determine the number of links configured:

```
int s, error, nlinks;
 error = ioctl(s, X25_GET_NLINKS, &nlinks);
```

The X.25 software maintains statistics for levels 1, 2, and 3. The statistics are made available for any socket at any time (that is, the sockets over which the calls for reading statistics are issued need not have superuser privilege).

The X25_RD_LINK_STATISTICS ioctl is used to read statistics of levels 1 and 2:

```
struct ss_dstats {
    int32_t ssd_ipack;    /* input packets */
    int32_t ssd_opack;    /* output packets */
    int32_t ssd_ichar;    /* input bytes */
    int32_t ssd_ochar;    /* output bytes */
 };

 /* error stats */
 struct ss_estats {
    int32_t sse_abort;      /* abort received */
    int32_t sse_crc;        /* CRC error */
    int32_t sse_overrun;    /* receiver overrun */
    int32_t sse_underrun;   /* xmitter underrun */
 };
 typedef struct x25_link_stat_db_s {
    int linkid; /* link identifier */
    u_short state;
    /* 0: initial state
     * 1: SABM outstanding
     * 2: FRMR outstanding
     * 3: DISC outstanding
     * 4: information transfer state
     */
    u_short hs_sentsabms;       /* sabms sent */
    struct ss_dstats hs_data;   /* data stats */
    struct ss_estats hs_errors; /* error stats */
 } X25_LINK_STAT_DB;

 X25_LINK_STAT_DB link_stats;
 int s, error;
 error = ioctl(s, X25_RD_LINK_STATISTICS, &link_stats);
```

The linkid field in the X25_LINK_STAT_DB structure identifies the interface whose statistics are to be read.

The X25_RD_PKT_STATISTICS ioctl is used for reading packet-level statistics for a specified logical channel:

```
typedef struct x25_pkt_stat_db_s {
    int linkid;   /* link identifier */
    u_short lcn;  /* logical channel identifier */
    u_char state; /* level 3 lcn state */
 /* current state of virtual circuit
    ST_OFF (0): virtual circuit not active
    ST_LISTEN (1): passive wait for incoming call
    ST_READY (2): connection in process of being established
                  (connection NOT up yet)
    ST_SENT_CALL (3): wait for call connected packet
    ST_RECV_CALL (4): wait user to send call accepted packet
    ST_CALL_COLLISION (5): call collision state
    ST_RECV_CLR (6): unused (should indicate reception of a
                 clear packet)
    ST_SENT_CLR (7): wait for clear confirmation packet
```

```
   ST_DATA_TRANSFER (8): in normal data transfer
   ST_SENT_RES (9): wait reset confirmation packet
*/
   u_char sub_state; /* level 3 lcn sub_state */
   /* valid only when state is ST_DATA_TRANSFER
      bit 0 (RECV_RNR): remote busy
      bit 1 (RECV_INT): wait user to read interrupt data
      bit 2 (SENT_INT): wait for interrupt confirmation
      bit 3 (SENT_RNR): local busy
   */
   u_char intcnt;   /* number of received interrupt datum */
   u_char resetcnt; /* times of virtual circuit reset */
   int sendpkts;    /* number of output packets */
   int recvpkts;    /* number of input packets */
   short pgrp;      /* process group of socket, if not 0 */
   short flags;     /* flag bits. If bit 0 is set, it */
   /* indicates an incoming call. */
   /* Otherwise, it is an outgoing call. */
} X25_PKT_STAT_DB;

X25_PKT_STAT_DB pkt_stats;
int s, error;
error = ioctl(s, X25_RD_PKT_STATISTICS, &pkt_stats);
```

The linkid field in the X25_PKT_STAT_DB structure identifies a link, and lcn identifies the logical channel whose statistics are to be read. Note that pkt_stats.`lcn` needs to be set to the proper logical channel number before making the X25_RD_PKT_STATISTICS ioctl call.

Solstice X.25 also provides ioctl commands to read the status of all of the links currently active and all the virtual circuits currently active. Use the `X25_GET_NEXT_LINK_STAT` ioctl to obtain link status as follows:

```
/*  The following is used to cycle through all the interfaces -
  *  static HDLC links as well as links used for LLC2.
  */
 typedef struct x25_next_link_stat_s {
    u_char opt;            /* search option */
 #define  GET_FIRST  0      /* get first one */
 #define  GET_NEXT   1      /* get next one */
    u_char specific;       /* applies to specified interface */
    u_char link_type;      /* HDLC_TYPE, LLC_TYPE */
    int linkid;            /* interface id */
    X25_MACADDR mac;       /* always null in current release */
 /* Level 2 states */
 #define LINKSTATE_DOWN  0              /* initial state */
 #define LINKSTATE_SABM  1              /* SABM outstanding */
 #define LINKSTATE_FRMR  2              /* FRMR outstanding */
 #define LINKSTATE_DISC  3              /* DISC outstanding */
 #define LINKSTATE_UP    4              /* info transfer state */
    u_short state;             /* link state--see preceding defines */
    u_short hs_sentsabms;      /* sabms sent */
    struct ss_dstats hs_data;  /* data stats */
    struct ss_estats hs_errors; /* error stats */
 } X25_NEXT_LINK_STAT;

    int s;
    int error;
```

```
X25_NEXT_LINK_STAT lstats;

lstats.opt = GET_FIRST;
lstats.specific = 0;
do {
   error = ioctl(s, X25_GET_NEXT_LINK_STAT, &lstats);
   if (error == 0)
   /* print the statistics */;
   } while (error == 0);
```

If the statistics for a specific link are required, set specific to 1, and linkid to the id of the interface whose statistics are required. After the first call, the opt field will automatically be changed to GET_NEXT. When the statistics for all the links are returned, error will be -1, and errno will be set to ENOENT.

Use the X25_GET_NEXT_VC_STAT ioctl to obtain the status of all the virtual circuits as follows:

CODE EXAMPLE 12–1 Reading Virtual Circuit Status

```
/* X25_NEXT_VC_STAT is used to cycle through all virtual circuits,
  * over HDLC as well as LLC type links.
  */
 typedef struct x25_next_vc_stat_s {
    u_char    opt;       /* search option */
    u_char    specific;  /* applies to specified linkid */
    u_char    link_type; /* HDLC_TYPE, LLC_TYPE */
    int    linkid;       /* link id */
    u_short    lcn;      /* logical channel to return */
    u_char    state;     /* level 3 lcn state */
 #define ST_OFF             0
 #define ST_LISTEN          1
 #define ST_READY           2
 #define ST_SENT_CALL       3
 #define ST_RECV_CALL       4
 #define ST_CALL_COLLISION  5
 #define ST_RECV_CLR        6
 #define ST_SENT_CLR        7
 #define ST_DATA_TRANSFER   8
 #define ST_SENT_RES        9
    u_char    sub_state;  /* level 3 lcn sub_state */
 #define RECV_RNR  0
 #define RECV_INT  1
 #define SENT_INT  2
 #define SENT_RNR  3
    u_char    intcnt;    /* number of received interrupts */
    u_char    resetcnt;  /* times of virtual circuit reset */
    int    sendpkts;     /* number of output packets */
    int    recvpkts;     /* number of input packets */
    short    pgrp;       /* process group, if any */
    short    flags;      /* various flags for future */
 #define  INCOMING_CALL  0x01
 #define  IS_A_PVC  0x02
    struct sockaddr  sa;  /* Remote X.121/IP address */
    AEF    aef;           /* Remote AEF, if any */
    X25_MACADDR  mac;    /* Remote mac for LLC links */
 } X25_NEXT_VC_STAT;
```

```
int s;
int error;
X25_NEXT_VC_STAT vstats;

vstats.opt = GET_FIRST;
vstats.specific = 0;
do {
   error = ioctl(s, X25_GET_NEXT_VC_STAT, &vstats);
   if (error == 0)
   /* print the statistics */;
   } while (error == 0);
```

hoIf the statistics of virtual circuits for a specific link are required, set specific to 1, and linkid to the id of the desired interface. After the first call, the opt field will automatically be changed to GET_NEXT. When the statistics for all the virtual circuits are returned, error will be -1, and errno will be set to ENOENT.

12.9.1.1 Obtaining Version Number

The X25_VERSION ioctl returns the version number of the Solstice X.25 kernel code. You can issue this call on any socket. The version number returned for the current release of Solstice X.25 is 92.

```
int so, version, error;
 error = ioctl(s, X25_VERSION, &version);
```

CHAPTER 13

Sockets Programming Example

This chapter discusses include files and structures, and provides references to example code.

Note - The sockets-based interface is a source-compatible—not a binary-compatible—interface. Applications that used the socket interface in SunOS 4.x must be recompiled to run on SunOS 5.x. See Section 13.2 "Compilation Instructions and Sample Programs " on page 242" for instructions on compiling programs to use the sockets-based interface on SunOS 5.0.

13.1 Include Files for User Programs

Sockets-based Solstice X.25 application programs need to have the following include statements in addition to any standard SunOS system files that may be needed:

```
#include <sys/iocom.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sundev/syncstat.h>
#include <netx25/x25_pk.h>
#include <netx25/x25_ctl.h>
#include <netx25/x25_ioctl.h>
```

This is illustrated in the sample programs provided.

13.2 Compilation Instructions and Sample Programs

To use the 7.0 socket interface, user programs should be linked against `libsockx25` stored in `/opt/SUNWconn/lib`. Use the `-L` option to link the `/opt/SUNWconn/lib` directory into your program. A program named `test` can be linked against the socket library as follows:

```
hostname% cc -o test test.c -L/opt/SUNWconn/lib -lsockx25 -lsocket -lnsl
```

You can find sample programs for the 7.0 socket interface in `/opt/SUNWconn/x25/samples.socket`.

13.3 Structures Used by the X25_SET_FACILITY and X25_GET_FACILITY ioctl Commands

The following structures are referenced in Section 12.7.2 "`X25_SET_FACILITY/X25_GET_FACILITY` ioctls" on page 217.

CODE EXAMPLE 13–1 Structures Used by ioctls that Set and Get X.25 Facilities

```
/* Packet sizes allowed are 0 (default), 16, 32, 64,
  * 128, 256, 512, 1024,2048, 4096
  */

 typedef struct packet_size_s {
    u_short sendpktsize;
    u_short recvpktsize;
    } PACKET_SIZE;
 /* window sizes allowed are 0:
  * (default), 1-7 (normal), 1-127 (extended)
  */

 typedef struct window_size_s {
    u_char sendwndsize;
    u_char recvwndsize;
 } WINDOW_SIZE;
 /* throughput values allowed are
  *   0 (default), 3 (75) , 4 (150), 5 (300),
  *   6 (600), 7 (1200), 8 (2400), 9 (4800),
  *   10 (9600), 11 (19200), 12 (48000)
  */
```

```
typedef struct throughput_s {
   u_char sendthruput:4;
   u_char recvthruput:4;
} THROUGHPUT;

typedef struct cug_s {
   u_char cug_req;
#define CUG_NONE 0 /* no CUG */
#define CUG_REQ 1 /* CUG */
#define CUG_REQ_ACS 2 /* CUG with outgoing access */
#define CUG_BI 3 /* bilateral CUG */
   u_short cug_index;
} CUG;

typedef struct rpoa_s {
   u_char nrpoa;   /* number of RPOAs requested */
#define MAX_RPOA 3
   u_short rpoa_index[MAX_RPOA]; /* rpoas;
                   nrpoa = 1 => normal format */
} RPOA;
/* Zero value for a field means the field is not specified; if a
 * field has zero value, that and the foll. fields are not sent.
 */

typedef struct ete_tr_delay_s {
   u_short req_delay;
   u_short desired_delay;
   u_short max_delay;
} ETE_TR_DELAY;

typedef struct nui_s {
   u_char nui_len; /* NUI length */
#define MAX_NUI 64
   u_char nui_data[MAX_NUI]    /* NUI */
} NUI;

typedef struct charge_info_s {
   u_char charge_len;
#define MAX_CHARGE_INFO 64
   u_char charge_data[MAX_CHARGE_INFO];
} CHARGE_INFO;

typedef struct call_redir_s {
   u_char cr_reason;
   u_char cr_hostlen;
   u_char cr_host[(MAXHOSTADR+1)/2];
} CALL_REDIR;

typedef struct aef_s {
   u_char aef_type;
#define AEF_NONE 0
#define AEF_NSAP 1
#define AEF_PARTIAL_NSAP 2
#define AEF_NON_OSI 3
   u_char aef_len;
#define MAX_AEF 40
   u_char aef[(MAX_AEF+1)/2];
} AEF;
```

```
typedef struct precedence_s {
   u_char precedence_req;   /* no precedence when = 0
                                   * else precedence level
                                   */
   u_char precedence;   /* valid when precedence_req = 1 */
} PRECEDENCE;

typedef struct private_fact_s {
   u_char p_len; /* total length of facilities */
#define MAX_PRIVATE 64
   u_char p_fact[MAX_PRIVATE];
/* facilities exactly as they
 * are present in Call Request or
 * Call Accept packets
 */
} PRIVATE_FACT;

typedef struct facility_s {
   u_int type;
#define T_FACILITIES 0x00000001
#define T_REVERSE_CHARGE 0x00000002
#define T_FAST_SELECT_TYPE 0x00000003
#define T_PACKET_SIZE 0x00000004
#define T_WINDOW_SIZE 0x00000005
#define T_THROUGHPUT 0x00000006
#define T_CUG 0x00000007
#define T_RPOA 0x00000008
#define T_TR_DELAY 0x00000009
#define T_MIN_THRU_CLASS 0x0000000a
#define T_ETE_TR_DELAY 0x0000000b
#define T_NUI 0x0000000c
#define T_CHARGE_REQ 0x0000000d
#define T_CHARGE_MU 0x0000000e
#define T_CHARGE_SEG 0x0000000f
#define T_CHARGE_DUR 0x00000010
#define T_LINE_ADDR_MOD 0x00000011
#define T_CALL_REDIR 0x00000012
#define T_EXPEDITED 0x00000013
#define T_CALLED_AEF 0x00000014
#define T_CALLING_AEF 0x00000015
#define T_STDSERVICE 0x00000016
#define T_OSISERVICE 0x00000017
#define T_PRECEDENCE 0x00000018
#define T_PRIVATE 0x00000019

   union {
      u_intfacilities;/* quick way to check
                                 * if a facility is present
                                 */
#define F_REVERSE_CHARGE 0x00000001
#define F_FAST_SELECT_TYPE 0x00000002
#define F_PACKET_SIZE 0x00000004
#define F_WINDOW_SIZE 0x00000008
#define F_THROUGHPUT 0x00000010
#define F_MIN_THRU_CLASS 0x00000020
#define F_CUG 0x00000040
#define F_RPOA 0x00000080
#define F_TR_DELAY 0x00000100
#define F_ETE_TR_DELAY 0x00000200
#define F_NUI 0x00000400
```

```
#define F_CHARGE_REQ 0x00000800
#define F_CHARGE_MU 0x00001000
#define F_CHARGE_SEG 0x00002000
#define F_CHARGE_DUR 0x00004000
#define F_LINE_ADDR_MOD 0x00008000
#define F_CALL_REDIR 0x00010000
#define F_EXPEDITED 0x00020000
#define F_CALLED_AEF 0x00040000
#define F_CALLING_AEF 0x00080000
#define F_STDSERVICE 0x00100000
#define F_OSISERVICE 0x00200000
#define F_PRECEDENCE 0x00400000
#define F_PRIVATE 0x00800000
      u_char reverse_charge;
/* permit/request reverse charge */
      u_char fast_select_type;
#define FAST_OFF 0 /* don’t use fast select */
#define FAST_CLR_ONLY 1 /* restricted response */
#define FAST_ACPT_CLR 2 /* unrestricted response */
                    PACKET_SIZE packet_size; /* packet sizes */
                    WINDOW_SIZE window_size; /* window sizes */
                     THROUGHPUT throughput; /* used for throughput
                           negotiation */
         THROUGHPUT min_thru_class; /* minimum throughput class */
         CUG cug; /* closed user group */
         RPOA rpoa; /* RPOA specification */
         u_short tr_delay; /* network transit delay */
         ETE_TR_DELAY ete_tr_delay; /* end-to-end transit delay */
         NUI nui; /* network user identification */
         u_char charge_req; /* request charging info */
         CHARGE_INFO charge_mu; /* charging info, monetary unit */
         CHARGE_INFO charge_seg; /* charging info, segment */
         CHARGE_INFO charge_dur; /* charging info, call duration */
         u_char line_addr_mod; /* called line addr modified */
         CALL_REDIR call_redir; /* call redirect notification */
         u_char expedited; /* expedited data negotiation */
         AEF called_aef; /* called aef */
         AEF calling_aef; /* calling aef */
         u_char osiservice; /* set when VC carries CLNP data */
         u_char stdservice; /* set for DDN services */
         PRECEDENCE prec; /* precedence for standard services */
         PRIVATE_FACT private; /* non-X.25 local/rem facilities */
      } facility;
} FACILITY;

/* Some convenient definitions. */
#define f_facilities facility.facilities
#define f_reverse_chargefacility.reverse_charge
#define f_fast_select_typefacility.fast_select_type
#define f_packet_size facility.packet_size
#define f_recvpktsize facility.packet_size.recvpktsize
#define f_sendpktsize facility.packet_size.sendpktsize
#define f_window_size facility.window_size
#define f_recvwndsize facility.window_size.recvwndsize
#define f_sendwndsize facility.window_size.sendwndsize
#define f_throughput facility.throughput
#define f_recvthruput facility.throughput.recvthruput
#define f_sendthruput facility.throughput.sendthruput
#define f_min_thru_classfacility.min_thru_class
#define f_min_recvthruputfacility.min_thru_class.recvthruput
```

```
#define f_min_sendthruputfacility.min_thru_class.sendthruput
#define f_cug  facility.cug
#define f_cug_req facility.cug.cug_req
#define f_cug_index facility.cug.cug_index
#define f_rpoa  facility.rpoa
#define f_nrpoa  facility.rpoa.nrpoa
#define f_rpoa_req facility.rpoa.rpoa_req
#define f_tr_delay facility.tr_delay
#define f_ete_tr_delay facility.ete_tr_delay
#define f_req_delay facility.ete_tr_delay.req_delay
#define f_desired_delay facility.ete_tr_delay.desired_delay
#define f_max_delay facility.ete_tr_delay.max_delay
#define f_nui  facility.nui
#define f_charge_req facility.charge_req
#define f_charge_mu facility.charge_mu
#define f_charge_seg facility.charge_seg
#define f_charge_dur facility.charge_dur
#define f_line_addr_mod facility.line_addr_mod
#define f_call_redir facility.call_redir
#define f_cr_reason facility.call_redir.cr_reason
#define f_cr_hostlen facility.call_redir.cr_hostlen
#define f_cr_host facility.call_redir.cr_host
#define f_expedited facility.expedited
#define f_called_aef facility.called_aef
#define f_cd_aef_type facility.called_aef.aef_type
#define f_cd_aef_len facility.called_aef.aef_len
#define f_cd_aef facility.called_aef.aef
#define f_calling_aef facility.calling_aef
#define f_cg_aef_type facility.calling_aef.aef_type
#define f_cg_aef_len facility.calling_aef.aef_len
#define f_cg_aef facility.calling_aef.aef
#define f_osiservice facility.osiservice
#define f_stdservice facility.stdservice
#define f_prec  facility.prec
#define f_precedence_reqfacility.prec.precedence_req
#define f_precedence facility.prec.precedence
#define f_private facility.private
```

Index

D

E

F

G

H

I

L

M

N