HEWLETT® PACKARD

Programmer's guide

X.25/9000

Contents

Contents

Contents

Contents

Contents

1

Introduction to X.25 Programmatic Access

Overview

This chapter introduces the X.25 product with emphasis on its X.25 programmatic access (X.25/PA) interface, which this manual describes in detail. With X.25, application programs can control X.25 packet transmission (level 3 of the Open Systems Interconnectivity (OSI) model).

This chapter contains 3 parts:

- X.25 Product Description
- Using BSD IPC
- Getting Started with X.25 Programmatic Access

All X.25 programmatic access is through Berkeley Software Distribution Interprocess Communication (BSD IPC) facilities, which are also known as Berkeley Sockets.

X.25 Product Description

The X.25 Link software and hardware enable an HP system to communicate with other HP and non-HP hosts by way of an X.25 Packet Switching Network (PSN).

X.25 Configurations

The CCITT X.25 recommendations describe the packet interface between a DTE (Data Terminating Equipment) and a DCE (Data Communications Equipment). The DTE is a device (computer or terminal) which connects to an X.25 PSN. The DCE interface is supplied by the PSN to enable the DTE to connect to the network. This type of connection is usually made over telephone lines via a modem.

Figure 1 illustrates a typical X.25 network configuration between an HP 9000 and a remote host using an X.25 PSN.

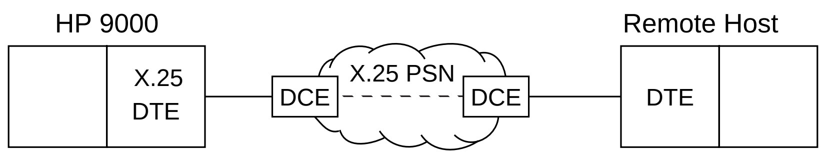

Figure 1 **Typical X.25 Network Connection**

A direct connection can be made via a modem eliminator. A modem eliminator provides the clocking mechanism necessary to synchronize signals between a DTE and a DCE. Two X.25 interfaces may be connected in this way, but one of them must be configured as a DCE. This type of configuration is called a “back-to-back configuration”.

Figure 2 illustrates a typical back-to-back configuration with a remote host. The remote host can be any type of processor, including another HP 9000 or any device capable of fulfilling the requirements of a DTE (or DCE in the back-to-back configuration).

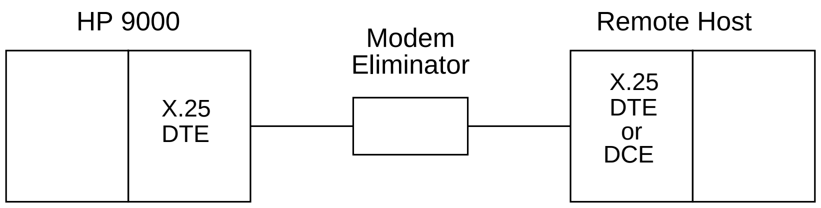

Figure 2 **Back-to -Back Configuration**

The term “remote” is used (in the relative sense) to designate a distant DTE which can be only a few feet, or thousands of miles away from the local DTE.

X.25 allows for the creation of multiple virtual connections over the same connection to a PSN. Each connection is called a virtual circuit (VC). X.25 can multiplex up to 4095 VCs (CCITT) over a single physical connection to a DCE. The X.25 PSN can route each VC to a different remote host. From the point of view of the application, each VC is directly connected to the remote host.

NOTE In a back-to-back configuration, all VCs must be connected to the same remote host.

Using BSD IPC

BSD IPC is a set of program development tools for interprocess communication. HP's implementation of BSD IPC is a subset of the networking services originally developed by the University of California at Berkeley. Before you attempt to use BSD IPC, you must be familiar with the C programming language.

The BSD IPC facility allows you to create distributed applications that pass data between processes (on the same computer or on different computers connected by a network) without requiring a complete understanding of the many layers of networking protocols. This is accomplished by using a set of system calls. These system calls, when used in the correct sequence, allow you to create communication endpoints called sockets and transfer data between them.

You will also find a description here of the steps involved in establishing and using X.25 programmatic access through BSD IPC connections. This manual also describes the protocols you must use and how the BSD IPC system calls interact.

The library routines and system calls that you need to implement a BSD IPC application are described throughout this manual.

You need not specify any special libraries when compiling or linking to use BSD IPC over X.25. The all required library routines are in the common C library (`libc.a`). The compiler uses `libc.a` automatically.

Details of each system call are described in the Section 2 entries of your man pages.

Using BSD IPC and X.25 with the Client/Server Model

In order to run X.25 applications over BSD IPC, two separate application processes must be running on both ends of a VC. The following is a sequential synopsis of BSD/X.25 communications:

1 The **client** process requests a connection by sending a CALL REQUEST packet.

2 The **server** process, receives the CALL INDICATION packet and accepts

it by sending a CALL ACCEPTED packet.

The server process:

a creates a socket, binds an address or range of addresses to it,
b sets up a **listen queue** for receiving connection requests,
c and then passively waits for connection requests until they arrive

When a request arrives the server process can either accept or reject the connection based on the information contained in the request.

3 The client process creates a socket and requests a connection to the remote server process, using as a destination one of the addresses to which the server has bound its socket.

Once the server process accepts a client process's request and a connection is established, full-duplex (two-way) communication can occur between the two sockets; the two processes are then **peers** and can exchange data as equals.

Getting Started with X.25 Programmatic Access

Before you begin designing your application:

1 Finish reading this book so you have a good idea as to how processes establish a connection, exchange data, handle asynchronous (out-of-band) events, and terminate connections.

2 Ensure that your node or system manager has installed and configured the X.25 product on your local host.

3 The `x25check(1M)` command can be used to test that X.25 is running and connected.

4 Obtain the X.121 addresses and interface names of the X.25 interfaces you intend to use.

5 The `x25stat(1)` command returns this information.

6 Determine which role your application will play in connection establishment: client or server.

7 If your application program will play the role of the client in connection establishment:

- Obtain the addressing information for the remote hosts and servers to which your application will establish a connection. To a large degree this information is application dependent and may only be available from an authority on the remote host.
- Obtain the strategy for information exchange between your process and the remote. This usually, but not necessarily, is an extension of the client/server model, with different formats for requests and responses.

8 If your application program will play the role of the server in connection establishment:

- Define the range of addresses at which you will receive connection requests and make them known to the designers of client processes. This is dependent on the X.25 interfaces which are connected to your local host, the addresses used by existing servers, and the flexibility and connectivity of the clients from which your program will be accepting connections.
- Develop a strategy for information exchange. Typically this implies that one side of the connection requests and the other services. Often the client/server model is retained during this phase of the connection but is not required.

Example Programs

Several pairs of programs are shipped with the X.25 product. These programs are in the `/usr/netdemo/x25` directory. Appendix B gives a brief description of the example files.

2

X.25 Addressing

Overview

This chapter discusses the issues associated with addressing over an X.25 interface. Addressing is used to define the particular interface and application that is used during a Switched Virtual Circuit (SVC) connection.

Note The information in this chapter applies only SVCs. This information does not apply to Permanent Virtual Circuits.

The issues discussed in this chapter include:

- Levels of Addressing
- Preparing Addressing Variables
- Addressing Options for Clients
- Addressing Options for Servers
- Call-Matching by X.25 Interface Name
- Call-Matching by Called X.121 Address Only
- Call-Matching by Subaddress
- Call-Matching by Protocol ID
- Using Wildcard Addressing
- Address Space Conflicts

Levels of Addressing

X.25 allows call addressing to use an interface's programmatic access name, the X.121 address and subaddress, and protocol ID. This information is contained in the `x25addrstr` structure, described below.

The application can specify which X.25 interface to use when receiving and connecting calls. This level of addressing is only useful when there is more than one X.25 interface (as with dual-port cards or systems with multiple cards) connected to the HP 9000 system.

note For the purposes of this discussion, and throughout this book, you should understand the distinction between the following terms: **Card**—refers to physical communications hardware, **Interface** (or **Port)**—used interchangeably to designate the physical point of connection for communications, and **Device**—a logical entity (internal to the communications software) that is logically associated with a particular interface.

The application can specify which X.121 address to use when the connection is established. Each interface connected to an X.25 PSN is assigned a unique X.121 address. When an interface is connected to a PSN, the subaddress also designates the X.25 interface (although for test purposes in back-to-back configurations, it is possible to use a different X.121 address than that specified for the PSN interface at configuration time).

The application can specify the X.121 subaddress to use in connecting the call. The subaddress may be used to select a particular type of application on the other end of the call.

The applications may also use the protocol ID to further select the type of application on the other end of the call. Protocol ID addressing is fully described in the CCITT X.244 (1984) Recommendations or chapter 6 of the X.25 (1980) Recommendations.

Preparing Address Variables

All addressing information for both the client and server is contained in the `x25addrstr` structure. This structure is defined in the include file `x25/x25addrstr.h`. It is used by the client in the `connect()` system call and by the server in the `bind()` system call. How the client and server use these calls is described in chapter 3.

The `x25addrstr` structure consists of the following declarations:

```
struct x25addrstr {
    unsigned short x25_family;
    unsigned char x25hostlen;
    unsigned char x25pidlen;
    unsigned char x25pid[8];
    unsigned char x25_host[16];
    char          x25ifname[13];
    } /* x25addrstr */
```

`x25_family`	Specifies the address family and must be set to `AF_CCITT`, which is defined in the `sys/socket.h` include file.
`x25hostlen`	Specifies the offset for the end of the numeric string in the X.121 address specified in the `x25_host` field described below. Range: 0 to 15 (including subaddress digits).
`x25pidlen`	Specifies the offset for the end of the character string that describes the protocol ID data. This field is not used in `connect()` system calls; the protocol ID must be explicitly set in the user data field by the application. Range: 0 to 8. Set this field to 0 in a `connect()` system call or (if protocol IDs are not used) in your application.
`x25pid[8]`	Specifies the protocol ID data in a `bind()` system call. The protocol ID data is located in the call-user data field of the CALL INDICATION packet. See “Addressing Options for Servers” below.
`x25_host[16]`	Specifies a destination X.121 address in a `connect()` system call with a decimal string (digits 0-9). In a `bind()` system call, this field specifies the range of X.121 addresses that it will receive with a decimal string and (optionally) with wildcard characters (“?” and “*”). See “Addressing Options for Servers” below. This field may also include a subaddress.

`x25ifname[13]`	Specifies the name of the X.25 interface set during X.25 configuration. The null string (“\0”) specifies the default interface in `connect()` and all interfaces in `bind()`. Range: 1 to 12 alphanumeric characters terminated by the null character (“\0”).

Refer to the `AF_CCITT(7F)` entry in your `man` pages for more information on the `x25addrstr` structure.

Addressing Options for Clients

The client process specifies the address to which it wants to connect an SVC. The client uses the `x25addrstr` structure in the `connect()` system call to specify most of the addressing information. If protocol IDs are used for call matching, the client process will also use `ioctl(X25_WR_USER_DATA)`.

In general clients have no real addressing options. The client must specify the addressing information that the network and server need to connect and handle the call properly. This information must be obtained from some authority for the server's host, such as the application designer or the system administrator.

The fields employed in the `x25addrstr` structure when a `connect()` system call is used are given below.

`x25_family`	Specifies the address family and must be set to `AF_CCITT`, which is defined in the `sys/socket.h` include file.
`x25hostlen`	Specifies the number of BCD digits in the X.121 address including the subaddress specified in the `x25_host field`. Range: 0 to 15.
`x25pidlen`	Is not used and should be set to 0.
`x25pid[8]`	Is not used and should be set to the null string (“\0”). If a protocol ID must be specified, the `ioctl(X25_WR_USER_DATA)` must be used.
`x25_host[16]`	Contains a character string of decimal digits (0-9) representing the remote host's X.121 address and subaddress if any.
`x25ifname[13]`	Specifies the name of the local X.25 interface to be used when sending a call request. The interface name is set during X.25 configuration. Specify the null string (“\0”) to use the default interface. Range: 1 to 12 alphanumeric characters terminated by the null character (“\0”). If the local host has more than one interface with equal connectivity the client may choose between them for reasons of throughput and response time.

Addressing Options for Servers

A server uses the X.25 socket address information to identify which calls it will process. Each server uses the `bind(2)` system call to define the addressing information for calls it will process. The `bind()` system call is described in chapter 3 and in your HP-UX `man` pages.

The discussion of incoming call-matching methods includes:

- Call-matching by interface name. An X.25 interface name is specified in the `x25ifname[]` field of the `x25addrstr` structure. Only calls arriving over that interface may be connected to the socket.
- Call-matching by called X.121 address. The called address is stored in the `x25_host` field of the `x25addrstr` structure. Only calls with the specified called address may be connected to the socket.
- Call-matching by called X.121 address and a subaddress. The subaddress is stored in the `x25_host` field of the `x25addrstr` structure. Only calls with the specified called address and subaddress may be connected to the socket.
- Call-matching by protocol ID. The protocol ID is set in the `x25_pid[]` field of the `x25addrstr` structure. Only calls with the correct protocol ID can be connected.
- Addressing conflicts in `bind()` calls.

When a CALL REQUEST packet arrives, three tests are performed to attempt to match the call to a listen socket:

- The name of the interface over which the call arrived is matched against the `x25ifname` field specified in the `bind()`.
- The called address field is matched against the `x25_host` field specified in the `bind()`.
- The first bytes of the user data field in the CALL REQUEST packet are matched against the `x25pid` field specified in the `bind()`.

If all of these tests succeeds, the call is connected to the socket. If the incoming call does not match with any of the specified addresses, the call is cleared.

Note If any of the three tests fail, the call is cleared before reaching the socket and the server application (above the socket) will never know anything about this incoming call request.

The matching tests for incoming calls and how a server controls calls are discussed below.

Call-Matching by X.25 Interface Name

The name of an X.25 interface is assigned during configuration. Refer to the *X.25/9000 User's Guide* for details on the interface name.

The `x25ifname[]` field of the `x25addrstr` structure may contain an interface name to designate a particular X.25 interface to be used for call connection. It can also specify that calls be connected from any X.25 interface on the system. If your application will receive calls from a single interface, the name must be specified in the `x25ifname` field. If your application shall receive calls arriving over any interface, no interface name can be specified in the `x25ifname[]` field (set to the null string, “\0”).

This field is of little importance when only one X.25 interface is in use. If more than one interface are in use, specify which interface connects to the network in the `x25ifname[]` field.

To accept calls from more than one X.25 interface (but not all interfaces) a separate socket must be created for each interfaces from which calls will be accepted. The resulting `listen()` sockets must be monitored with the `select()` call to determine when a matching incoming call arrives. Refer to “Using Nonblocking I/O” in chapter 4 of this manual.

Call-Matching by Called X.121 Address Only

The X.121 address for an X.25 interface is assigned during initialization. When an X.25 interface is connected to a PDN, the X.121 address is assigned to the interface by the network provider. The X.121 address is a string of decimal digits (0-9). Refer to the *X.25/9000 User's Guide* for details on X.121 address initialization.

When an interface is connected to a PDN, only CALL REQUEST packets with a called address field equal to the X.121 address assigned to the interface will be delivered to the interface. In this case specifying an X.121 address is synonymous with specifying an interface name.

In back-to-back configurations, a CALL REQUEST packet with any valid X.121 address can be received by the interface. Any CALL REQUEST packet, regardless of its X.121 address, is processed by the interface. If a socket with a matching X.121 address is found, a connection is made.

When issuing a `bind()` system call, the `x25_host` field of the `x25addrstr` can contain an X.121 address or be empty. If the `x25_host` field is not empty, then the specified address must exactly match the called address field of the CALL REQUEST packet. The called address field must exactly match (digit-for-digit) and be of equal length to the `x25_host[]` field in the `x25addrstr` structure specified in the `bind()` call. See "Using Wildcard Addressing".

If the `x25_host` field is empty, then the `x25hostlen` field of the bind address is zero (no X.121 address is specified), and the `x25_host` field will match the called address field of any incoming CALL REQUEST packet with no subaddress.

Call-Matching by Subaddress

Call matching by subaddress is actually an extension of call matching by X.121 address. The subaddress is appended to the called address field in the CALL REQUEST packet and the `x25_host[]` field in the `x25addrstr` structure. The subaddress, like the X.121 addresses, is a string of decimal digits (0-9). Not all PDNs support subaddresses, and some support a varying number of subaddress digits. Ask your node or network manager for configuration information concerning subaddresses.

Call-matching by subaddress is one method by which several servers may service different calls over the same interface. The programmer of the client process must know the subaddress as well as the X.121 address before connection begins. The X.121 addresses and the subaddresses must exactly match in order for an incoming request to be bound to a socket.

The combined length of the X.121 address and the subaddress must be less than 16 digits. The`x25hostlen` field must include the length of the X.121 address and the length of the subaddress.

Call-Matching by Protocol ID

Call-matching by protocol ID is a flexible way to allow multiple servers to service incoming calls over the same interface. First the X.121 address and subaddress is tested, and finally the protocol ID is tested for the incoming call. If the `x25pidlen` field is 0, the protocol ID is not used.

The protocol ID field is at the beginning of the call user data field of the CALL REQUEST packet. The server specifies the protocol ID in the `x25pid` field of the `x25addrstr` structure. The protocol ID may be from 1 to 9 bytes long. The CCITT X.244 Recommendations describes protocol ID addressing.

Client and server programmers must agree upon how many bits to specify for the protocol ID, but the length is not defined by the X.244 (1984) and X.25 (1980) Recommendations. HP suggests that you use protocol IDs to match incoming calls to sockets, because a single listen socket can be used for any number of X.25 interfaces (independent ports), and subaddresses are not always supported over PDNs.

You can also set a bit mask to specify a range of protocol IDs. The bit mask is described in “Using Wildcard Addressing”. Matching by protocol ID can identify higher-level protocols, such as those specified by PAD support.

Using Wildcard Addressing

Wildcard addresses are used in `bind()` calls only. They cannot be used in `connect()` calls. Wildcard addressing allows a single listen socket to connect to incoming calls using a variety of addresses and protocol IDs.

There are three types of wildcard addressing. The one which is implemented depends on the field in the `x25addrstr` structure that is being used. When an incoming CALL REQUEST packet is received, the `x25addrstr` structure fields are checked in the following order: `x25ifname`, `x25_host` and then `x25pid`.

Wildcard Addresses in the x25ifname[] Field

The `x25ifname[]` field has only one form of wildcard addressing. If you specify the null string ("\0"), the specified address will match the X.121 address of any interface connected to your system. If you specify an interface name, only calls from that one interface will match.

Wildcard Addresses in the x25_host[] Field

A wildcard address in the `x25_host[]` field may be the null string or use special wildcard characters. As described above, the null string will match only the X.121 address of the interface on the receiving end, but not the subaddress.

The valid wildcard characters are the question mark ("?"), and the asterisk ("*"). These characters may be used in combination with the decimal digits normally specified in this field. When they are specified in the `x25_host` field and are matched with an incoming call's called address field they have the properties described in table below.

Table 1 **Wildcard Characters**

Character	Meaning
?	Matches any single digit in the same position; for example, 1? matches 10, 11, 12, 13, 14, 15, 16, 17, 18, and 19.
*	Matches any decimal digit, including none; for example, 1* matches any address beginning with 1 including 1 itself. It can be used alone or as a suffix.

The table below illustrates the various possibilities of matching an non-matching addresses for a given number using wildcard characters.

Table 2 **Wildcard Address Matching for x25_host**

Example Address	Matching Addresses	Non-Matching Addresses
7234	7234	All addresses except 7234.
723?	7230, 7231, 7232, 7233,7234,7235, 7236,7237,7238, or 7239	All addresses that do not begin with 723, and all addresses that are not 4 digits long.
72?4	7204, 7214, 7224, 7234, 7244, 7254, 7264,7274, 7284, or 7294	All addresses that do not begin with 72 and end with 4, and all addresses that are not 4 digits long
*	All addresses are valid.	None.
??*	All addresses of 2 digits or more are valid.	Addresses with only 1 digit (0, 1, 2, 3, 4, 5, 6, 7, 8, or 9).
*??	Invalid syntax - nothing matches.	Invalid syntax.

Setting a Wildcard Protocol ID Local Address Mask

The server specifies the protocol ID for its listen socket in the `x25pid` field of the `x25addrstr` structure. The protocol ID is part of the call user data field in the CALL REQUEST packet. This field is added (ANDed) with the mask specified in the `ioctl(X25_WR_MASK_DATA)`. The `x25pid` field and the mask specified in the `ioctl(X25_WR_MASK_DATA)` can be combined to enable a certain degree of wildcard addressing.

The protocol ID masking match works as follows:

1 The first byte from the CALL INDICATION packet's call user data field is "masked" (that is, logically ANDed, bit-by-bit) with the first byte of the mask specified with the `ioctl(X25_WR_MASK_DATA)`.

2 The result is compared to the value specified in the first byte of the `x25pid` field in the `x25addrstr` structure specified in the `bind()` call.

3 If the result is unequal, the comparison fails. If equal, the comparison continues with the next byte of each field until a mismatch occurs or the number of bytes in the bind address's `x25_pidlen` field has been compared.

Use this wildcard method if the incoming protocol ID you need isn't a whole number of bytes, or there are bytes within the field that are not part of the protocol ID. For example, some systems place a length byte at the beginning of the call user data field, which should be ignored in protocol ID matching.

The bit-by-bit comparison is described in the following table:

Table 3 **x25pid and x25_mask Usage**

Call User Data Bit	x25_mask Data Bit	x25pid Data Bit
0 or 1	0	0 always matches
0 or 1	0	1 always fails
0	1	0 matches
1	1	1 matches
1	1	0 fails
0	1	1 fails

Syntax for ioctl (X25_WR_MASK_DATA)

The syntax for the `ioctl(X25_WR_MASK_DATA)` system call and its parameters are described below.

```
#include <x25/x25ioctls.h>
#include <x25/x25.h>
int err;
/*  DEFINE X25_MAX_PIDLEN 8
 *    struct x25_mask_data {
 *    u_char x25_masklen;
 *    u_char x25_mask[X25_MAX_PIDLEN];
 *    }
 */
int sd;
struct x25_mask_data mask;
err = ioctl(sd, X25_WR_MASK_DATA, &mask);
```

`sd`	A socket descriptor for a listen socket.
`X25_WR_MASK_DATA`	Indicates the type of `ioctl()` being performed. If the `X25_WR_MASK_DATA` value or `x25_mask_len` value is set to `0`, the `ioctl()` call returns no error and an empty mask is used. This has the same effect as if the call were not made.
`mask`	Indicates the mask to be ANDed with the protocol ID specified in the CALL REQUEST packet. The `x25_mask_len` field indicates the length of the mask, and the x25_mask indicates the mask to be used.

Address Space Conflicts

The X.25 subsystem’s programming access prevents any two sockets from binding to the same address structure. When a `bind()` call is made, the subsystem checks the specified address against addresses that are already associated with the socket. The `bind()` call is rejected if there is a conflict in the space allocation of components in the address structure.

The address structure is made up of three components:

- Interface name—the name of the interface or port
- Address/subaddress—the X.121 addresses
- PID—the Protocol Identification number

Address conflicts occur with `bind()` calls when the specified address structure occupies or overlaps into an address region that has already been assigned to another socket. In this instance the system returns one of two errors:

- EADDRNOTAVAIL—is returned when all of the addresses specified in the `bind()` call include all of the addresses specified in a previously bound socket.
- EADDRINUSE—is returned when the addresses specified in the bind include some of the addresses specified in a previously bound socket.

Table 4 **Addressing Conflict Errors**

Previous Bind	Current Bind	errno value
123*	12*	EADDRNOTAVAIL
12*	123*	EADDRINUSE
1?3*	123*	EADDRINUSE
123*	1?2?	EADDRNOTAVAIL

How to Avoid Address Conflicts

Avoid wildcard addresses with “*” and be cautious of all wildcard addressing. Avoid wildcards that specify large address spaces when specifying subaddresses. Specify an address space of exactly one address when specifying non-wildcard addresses. Each question mark increases the address space by a factor of 10; an asterisk increases the address space by several orders of magnitude.

The best way to avoid conflicts is to coordinate the use of address space with other servers, and write down the addresses that are in use. Check this list whenever a new server is installed. The first entry in the list should be for the `x25server` process which uses protocol ID `0xFCAA0A07`.

3

Establishing and Terminating a Socket Connection

Overview

This chapter describes the steps involved in establishing and terminating an X.25 switched virtual circuit (SVC) using a BSD IPC (socket). Topics include:

- Connection Establishment for the Server Process
- Connection Establishment for the Client Process
- Controlling Call Acceptance
- Terminating a Connection

Connection Establishment for the Server Process

This section describes the system calls and parameters that are executed by the server process to establish a connection.

In the simplest case, there are four steps that the server process must complete before a connection can be made with a client:

1. Create a socket with `socket()`.
2. Bind an address to the new socket with `bind()`.
3. Add a listen queue to the socket with `listen()`.
4. Wait for an incoming call with `accept()`.

Caution Programmers should take care to avoid issuing contradictory system calls when porting applications for operation with BSD IPC sockets. You cannot, for example, issue a `connect()` call on a socket on which you have previously issued a `bind()` call. Conflicting system calls will return the `EOPNOTSUPP (223)` error message.

Creating a Socket

The server process must call `socket()` to create a BSD IPC socket. This must be done before any other BSD IPC system call is executed.

Syntax for socket()

The `socket()` system call and its parameters are described below.

```
#include <sys/types.h>
#include <x25/ccittproto.h>
#include <sys/socket.h>


int sd;
int af, type, protocol;
sd = socket(af, type, protocol);
```

`af` Identifies the socket's address family. For X.25 programmatic access, AF_CCITT must be specified.

`type`	Identifies the type of socket. For X.25 programmatic access, SOCK_STREAM must be specified.
`protocol`	Identifies the underlying protocol to be used for the socket. For X.25 programmatic access, X25_PROTO_NUM should be specified. If 0 is specified the default protocol (X25_PROTO_NUM) is used.
`sd`	If the connection is successful, `sd` contains the socket descriptor for the newly-created socket. If the system call encountered an error, –1 is returned in `sd`, and `errno` contains the error code.

The socket descriptor returned by `socket()` references the newly-created socket. This descriptor is used for the subsequent system calls used to establish an SVC (`bind()`, `listen()` and `accept()`).

Refer to the `socket(2)` entry in your `man` pages for more information.

Binding an X.121 Address to a Socket

After your server process has created a socket and before a `listen()` system call is executed, the server must call `bind()` to associate an X.121 address to the socket. Until an address is bound to the server socket, X.25 cannot reach your server.

Syntax for bind()

The `bind()` system call and its parameters are described below.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <x25/x25addrstr.h>


int error;
int sd, addrlen;
struct x25addrstr bind_addr;
addrlen = sizeof(struct x25addrstr);
error = bind(sd, &bind_addr, addrlen)
```

`sd`	The socket (returned from a previous `socket()` system call) to which the address will be bound.

`bind_addr`	The `x25addrstr` structure which contains addressing information. The addressing information defines the types of CALL REQUEST packets that the server will handle. For a description of the issues associated with addressing, see chapter 2.
`addrlen`	The length of the `x25addrstr` structure in bytes.
`error`	If the call successfully completes, `error` contains a 0. If the system call encountered an error, –1 is returned in error, and `errno` contains the cause of the error.

Refer to the `bind(2` entry in your `man` pages for more information.

Preparing a Listen Socket

The `listen()` system call prepares a socket to receive CALL INDICATION packets whose address matches the address previously bound to the socket with a `bind()` call. All eligible CALL INDICATION packets are put into this queue. The server cannot receive a connection request until it has executed a `listen()`call.

Once a `listen()` call has been executed on a socket, calls that are correctly addressed are automatically accepted by the X.25 software. This prevents any time-outs from taking place while a client's request waits in the listen queue. A new socket is created along with all of the resources required to operate it including send and receive buffers.

Caution If the `bind_addr` parameter specifies a specific interface name (i.e. Call-Matching by X.25 Interface Name), the corresponding X.25 interface must be initialized *before* issuance of a `bind ()` call. Even if the `bind_addr` parameter does not specify an interface name (i.e. calls can be received from any interface), at least one X.25 interface must be initialized *before* the issuance of a `bind ()` call.

The new socket is:

- created with the same properties as the `listen()` socket (family = AF_CCITT, type = SOCK_STREAM).
- connected to the client process’s socket.

For more on this, see “Controlling Call Acceptance” on page 43.

Syntax for listen()

The `listen()` system call and its parameters are described below.

```
int error;
int sd, backlog;
error = listen(sd, backlog);
```

`sd`	The socket descriptor for a created and bound socket on which the process will wait for incoming CALL INDICATION packets.
`backlog`	The maximum length of the listen queue. Range: 1 to 20. Additional incoming CALL INDICATION packets are put into the queue regardless of the Range value. This allows the system to handle traffic surges without unexpected disconnection.
`error`	If the call successfully completes, `error` contains a 0. If an error is encountered, –1 is returned in `error`, and `errno` contains the cause of the error.

Incoming CALL INDICATION packets that match the socket's bind address (and the sockets created for them) are placed in the listen queue in the order in which they are received. Backlog requests can be waiting in the listen queue at the same time. You cannot send or receive data on a listen socket. Listen sockets only act as meeting points for incoming calls.

Closing the last active socket descriptor of a listen socket clears all pending requests and empties the listen queue. The socket is unusable after the `close()` call.

Refer to the `listen(2)` entry in your `man` pages for more information.

Accepting a Connection

The `accept()` system call returns a socket descriptor for a socket associated with an SVC connection. This call usually establishes a connection upon return, although this can also be controlled by the application. The transmission of the CALL ACCEPTED packet and its contents can be controlled with `ioctl(X25_CALL_ACPT_APPROVAL)` and `ioctl(X25_SEND_CALL_ACEPT)`. These `ioctl()` calls are described below.

The `accept()` call blocks the socket until a CALL REQUEST packet arrives (unless the listen socket is set to nonblocking mode).

Syntax for accept()

The `accept()` system call and its parameters are described below.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <x25/x25addrstr.h>
int sd, fromlen;
struct x25addrstr from;
int new_sd;
fromlen = sizeof(struct x25addrstr);
new_sd = accept(sd, &from, &fromlen);
```

`sd`	The socket descriptor used in a previous `listen()` call.
`from`	Upon successful completion, this `x25addrstr` structure will contain the name of the local interface that received the call, and the calling address and subaddress, if any, of the DTE which sent the CALL REQUEST packet. This information is useful when using wildcard addressing (see chapter 2).
`fromlen`	Upon successful completion, this integer will contain the length of the `x25addrstr` structure in bytes. Before calling `accept (0)`, this field must be initialized with the size declared in the `x25addrstr` structure.
`new_sd`	If the connection is successful, `new_sd` contains a socket descriptor for a new socket which is connected to the incoming call. If an error is encountered, –1 is returned in `new_sd` and `errno` contains the error code.

An `accept()` call usually returns a CALL ACCEPTED packet. However, the content and transmission of this packet can be controlled by the application. The `ioctl(X25_CALL_ACPT_APPROVAL)` and `ioctl(X25_SEND_CALL_ACEPT)` calls are used to control CALL ACCEPTED packets (see “Controlling Call Acceptance” on page 43).

If you set-up the listen socket to perform nonblocking I/O, your process will not block. Your request will return -1 and `errno` would contain `EWOULDBLOCK`. This means that there is no SVC connection request available at that time, but the `accept()` call is ready to process when it arrives. You can test the socket with `ioctl(X25_NEXT_MSG_STAT)`, (described in the next chapter) or with `select(2)`. The `select()` call allows you to specify when you want this test to take place.

Strategies for Server Design

HP suggests that you build a server process that creates a socket, binds an address, attaches a listen queue, and waits for the arrival of a CALL INDICATION packet with the `accept()` call. When the request packet arrives, the server process forks a child process to handle the newly established SVC.

The child process closes the socket descriptor for the listen socket, and the parent process closes the socket descriptor returned by `accept()`. The child process goes on to service the needs of the remote process. When the job is completed, it closes the connection and calls `exit(2)`. Meanwhile, the parent process calls `accept()` and waits for the next CALL INDICATION packet to service.

This technique may not suit all situations. If the server process will act upon one call request at a time, it can wait for a call, accept a call, execute a service request, close the call, and go back to wait for another call. In a database application, for example, it is not unusual for the server to accept only one incoming call at a time, completing the service request before accepting another.

In this case you would not fork a child process to accept the call. Instead the server might follow these steps:

1 Create a socket, bind an X.25 address to it, execute `listen()` on the socket.

2 Use `accept()` to obtain a connection.

3 Determine which service is requested.

4 Perform the requested service.

5 Terminate the connection (`close()`).

6 Go to step 2.

Notice that the listen socket is not closed, so incoming CALL REQUESTs are queued on the listen socket but not acted upon until the service request is completed.

Connection Establishment for the Client Process

This section discusses the system calls which the client process must make to establish an SVC with a server process. There are two mandatory steps:

1 Create a socket using the `socket()` call.

2 Make a connection request using the `connect()` call.

These steps are described below.

Creating a Socket

This is similar to the server process, the client process must also use the `socket()` call to create a BSD IPC socket (communications endpoint). The socket must be created before the `connect()` call is executed.

For a client, the `socket()` call and its parameters are identical to those used by the server when it creates a socket. See “Syntax for socket()” above.

The socket descriptor for the newly-created socket should be included in the `connect()` system call, and (after the connection is established) in all subsequent data transmission.

Refer to the `socket(2)` entry in your `man` pages for more information.

Requesting a Connection

The client process requests a connection with the `connect()` call. The server must be prepared to service a CALL INDICATION packet (with an active listen socket) when `connect()` is executed.

The client process specifies the X.121 address, subaddress, and protocol ID of the server with which it wants to establish an SVC with the `connect()` call. HP does not provide a programmatic method for obtaining this addressing information. It must be acquired from an authority associated with the remote host. When a `connect()` system call is issued, X.25 sends a CALL REQUEST packet with the specified addressing information.

Syntax for connect()

The `connect()` system call and its parameters are described below.

```
#include <sys/types.h>
#include <x25/x25addrstr.h>
#include <sys/socket.h>

int err;
int sd;

struct x25addrstr to_addr;
int to_addrlen;
err = connect(sd, &to_addr, to_addrlen);
```

`sd`	The socket descriptor returned by a previous `socket()` system call. It must use the AF_CCITT address family.
`to_addr`	The `x25addrstr` structure containing the local interface to be used in the call, as well as the X.121 address and subaddress of the remote server process with which the client process will establish an SVC.
`to_addrlen`	Contains the length of the `x25addrstr` struct (in bytes) which is pointed to by `to_addr`.
`err`	If the call successfully completes, `err` contains 0. If the system call encountered an error, –1 is returned in `err,` and `errno` contains the error number.

The `connect()` call transmits a CALL REQUEST packet and blocks the process until the connection is ready (unless you specify nonblocking mode).

If you place a call by issuing `connect()` on a socket which is nonblocking, your process will not block. Your request will return **`EINPROGRESS`** . This means that the process of connecting to the remote system has been initiated.

If your host system is connected to more than one X.25 interface and those interfaces do not have equal connectivity, you must specify the `x25ifname` field of the `x25addstr` structure. That field designates which interface must be used for the connection. If your system has only one interface, it is not necessary to designate the `x25ifname` field.

If the `x25ifname` field is null, the `connect ()` call sends outbound packets to the *first initialized interface* by default.

note The *first initialized interface* is the one that is first initialized when the sub-system is restarted after all cards are stopped.

You can use the `x25ifname` field to control performance. Each interface used to connect to a particular network can only support a fixed number of circuits. When selecting the network interface you should also consider bandwidth, throughput, and response-time factors.

note Facilities such as call user data, or (the circuit's D bit) must be specified before issuing the `connect()` call. Facilities are specified with the `ioctl(X25_WR_FACILITIES)`, and call user data is specified with `ioctl(X25_WR_USER_DATA)`. The D bit can be set with the `ioctl(X25_SEND_TYPE)`. These `ioctl()` calls are described in the following chapters.

If the client and server are using protocol IDs for address matching, prior to issuing the `connect()` call, the client must specify the server's protocol ID with the `ioctl(X25_WR_USER_DATA)` call (see "The ioctl(X25_WR_USER_DATA) Call" on page 84). The example below describes how to use `ioctl(X25_WR_USER_DATA)` to specify a remote protocol ID.

Example (client specifying the protocol ID)

The example below shows the system calls that a client must execute if the protocol ID is to be specified in the CALL REQUEST packet.

```
/* put the protocol ID in the call-user data field */
struct x25_userdata userdata;
...
userdata.x25_cud_len = 1; /* one byte for PID */
userdata.x25_cu_data[0] = 0x05; /*PID is 0x05 */
result = ioctl(s, X25_WR_USER_DATA, &userdata);
...
result = connect (s, &peeraddr, sizeof(struct x25addrstr));
```

Refer to the `connect(2)` entry in your `man` pages for more information.

Controlling Call Acceptance

The server process can control the acceptance of incoming CALL REQUESTs. How call acceptance operates is controlled through the use of `ioctl(X25_CALL_ACPT_APPROVAL)` and `ioctl(X25_SEND_CALL_ACEPT)` calls.

The steps required for controlling call acceptance are shown in the following table.

Table 5 **Controlling Call Acceptance when Establishing an SVC**

Server Events	X.25 Events	Client Events
1. `socket()`	No Event	No Event
2. `bind()`	No Event	No Event
3. `ioctl (X25_CALL_ACPT_APPROVAL)`	No Event	No Event
4. `listen()`	No Event	No Event
5. `accept() blocks`	No Event	No Event
6. No Event	No Event	`socket()`
7. No Event	CALL REQUEST packet transmitted	`con-nect() blocks`

Table 5 **Controlling Call Acceptance when Establishing an SVC**

8. No Event	CALL INDICATION packet received	No Event
9. `accept()` unblocks	No Event	No Event
10. `ioctl(X25_SEND_CALL_ACEPT)`	CALL ACCEPTED packet transmitted	No Event
11. No Event	CALL CONNECTED packet received	`con-nect()` unblocks

The `ioctl(X25_CALL_ACPT_APPROVAL)` call is used to instruct X.25 not to automatically send the CALL ACCEPTED packet. For a listen socket, the `ioctl(X25_CALL_ACPT_APPROVAL)` call should be implemented before the `listen()` call. If the `ioctl()` call is issued after the `listen()` call there is a risk that incoming calls might be automatically accepted during the brief delay that occurs between the `listen()` and `accept()` calls. The `accept()` still returns a socket descriptor connected to the incoming SVC, but no packets are sent.

note Any CALL INDICATION packet received after a `listen()` and before a `ioctl(X25_CALL_ACPT_APPROVAL)` is automatically accepted. Use the `ioctl(X25_NEXT_MSG_STAT)` to detect if the call has been automatically connected.

When the `ioctl(X25_SEND_CALL_ACEPT)` call is issued on an SVC socket it causes a CALL ACCEPTED packet to be transmitted.

The two `ioctl()` call acceptance calls are described below.

The ioctl (X25_CALL_ACPT_APPROVAL)

The `ioctl(X25_CALL_ACPT_APPROVAL)` call allows applications to screen incoming calls. When the call is issued for a given `listen()` socket, a new `accept()` socket is still created whenever a valid call comes in, but no data can be sent or received on the new socket until an `ioctl(X25_SEND_CALL_ACPT)` call is issued on the new socket. This feature must be set if:

- the process is going to control use of the D bit,
- facilities are specified,
- or user data in the CALL ACCEPTED packet.

If the application does not want to accept the call, the circuit can be cleared with the `close()` call. The `ioctl(X25_SEND_CALL_ACPT)` call is described below.

Once the `ioctl(X25_CALL_ACPT_APPROVAL)` is enabled, it cannot be turned off, unless the listen socket is closed and the `socket()`, `bind()`, and `listen()` calls are repeated.

Syntax for ioctl (X25_CALL_ACPT_APPROVAL)

The `ioctl(X25_CALL_ACPT_APPROVAL)` call and its parameters are described below.

```
#include <x25/x25ioctls.h>
int err;
int sd;
err = ioctl(sd, X25_CALL_ACPT_APPROVAL, 0);
```

`sd`	A socket descriptor for a listen socket that has no `accept()` pending on it.
`X25_CALL_ACPT _APPROVAL`	The definition for the request.
`0`	A dummy variable used because `ioctl(X25_CALL_ACPT_APPROVAL)` does not use any arguments.

The ioctl (X25_SEND_CALL_ACEPT) Call

The `ioctl(X25_SEND_CALL_ACEPT)` call causes X.25 (level 3) to send a CALL ACCEPTED packet. The call is executed on a socket descriptor returned from an `accept()` call. The listen socket on which the `accept()` call was issued must have previously had an `ioctl(X25_CALL_APPROVAL)` issued on it. The `ioctl(X25_SEND_CALL_ACEPT)` causes a CALL ACCEPTED packet to be sent on the SVC.

If the application determines that the call should not be accepted, the call can be rejected with a CLEAR packet by calling `close()` or `shutdown()` on the socket descriptor.

The application may specify D bit usage, facilities, and call user data to be placed in the CALL ACCEPTED packet. The `ioctl(X25_SEND_TYPE)`, `ioctl(X25_WR_USER_DATA)`, and `ioctl(X25_WR_FACILITIES)` calls control these functions. They must be issued prior to accepting the call with the `ioctl(X25_SEND_CALL_ACEPT)`.

The `ioctl(X25_SEND_TYPE)` is described in chapter 4, the `ioctl(X25_WR_USER_DATA)` is described in chapter 5, and the `ioctl(X25_WR_FACILITIES)` is described in chapter 6.

Syntax for ioctl(X25_SEND_CALL_ACEPT)

The `ioctl(X25_SEND_CALL_ACEPT)` call and its parameters are described below.

```
#include <x25/x25ioctls.h>
int err;
int sd;
err = ioctl(sd, X25_SEND_CALL_ACEPT, 0);
```

`sd`	A socket descriptor for an SVC socket. Is returned during an `accept()` call.
`X25_SEND_CALL _ACEPT`	The definition for the request.
`0`	A dummy variable used because `X25_SEND_CALL_ACEPT` does not use any arguments.

Terminating a Connection

When data communications activity over an SVC is completed, the connection should be terminated to free network memory and other resources. This also reduces communications costs because most PDNs charge for transmission and connection time.

An SVC is terminated when a CLEAR REQUEST packet is transmitted. A CLEAR packet can be transmitted to both ends of an SVC by the network provider, or it can be transmitted by one of the processes using the SVC. An X.25 application can transmit a CLEAR REQUEST packet by issuing a `close()` call on the last open socket descriptor for the socket associated with that SVC, or by issuing a `shutdown()` call on the socket associated with that SVC.

When a process terminates, all open socket descriptors are closed automatically.

Closing a Socket Descriptor

The `close()` system call closes a socket descriptor. If a `close()` is issued on the last open socket descriptor for an SVC socket, a CLEAR REQUEST packet is transmitted on the SVC.

If a process is no longer using a socket descriptor, it should be closed using the `close()` system call. This technique prevents the SVC from remaining connected after the last process has completed using it.

One of the design strategies described earlier suggested that a server process monitor a listen socket and then fork a child process to handle SVCs as they arrive. The server should close the socket descriptor for the newly-created socket immediately after spawning the child process. This allows the child process to close its socket descriptor and transmit a CLEAR REQUEST packet when needed. If the server did not close its socket descriptor, no CLEAR packet would be transmitted and the SVC would remain connected.

Syntax for close()

The following is the syntax for the `close()` system call and its parameters:

```
int err;
int sd;
err = close(sd);
```

`sd`	A socket descriptor for a listen socket or an SVC socket. If there are other open socket descriptors for the socket, the socket is no longer usable by the process issuing the call. If there are no other open socket descriptors for the socket, the socket is destroyed and may not be used again by any process. If `sd` is the last open socket descriptor and a descriptor for an SVC socket, a CLEAR REQUEST packet is transmitted on the SVC.
`err`	Upon successful completion, `err` is set to 0. Otherwise, a value of –1 is returned, and `errno` is set to indicate the error. Even when a nonzero value is returned, `sd` will not be usable.

The state of the socket and the state of the circuit are independent of one another. When a socket is closed, it vanishes and cannot be accessed again. When a circuit is cleared and the socket is not closed, it can no longer transmit data. However, you can read any unread out-of-band data that arrived on the inbound `recv()` queues before the clear. Any unread normal data cannot be read. See chapter 5 for more on receiving out-of-band data.

When the last `close` call is executed on a socket descriptor, any data that has not yet been sent or received is lost.

The best way to end a session without losing data is summarized below:

1. With the `send()` call, the sending side sends an “I am finished” message—the message content is defined by the application designer.
2. The receiving side reads this “I am finished” message with the `recv()` call.
3. The receiving side closes the virtual circuit with a `close()` call.
4. The sending side receives notification of the `close()` with a `recv(OOB)` call.
5. The sending side frees its socket resources by issuing a `close()` call.

You can specify data and facilities information in the CLEAR packet sent by the final `close()`. See the “RESET and CLEAR Packets” in chapter 5 for details.

For syntax and details on `close()`, refer to the `close(2)` entry in your `man` pages.

Note Closing a socket immediately after sending data can result in data loss.

Shutting Down a Socket

When your program finishes reading or writing on a particular socket connection, it can call `shutdown(2)` to bring down a part of the connection. Unlike `close()`, `shutdown()` affects the entire socket and all other socket descriptors. `shutdown()` causes all or part of an SVC to be disabled regardless of how many other socket descriptors are open on the socket.

Syntax for shutdown()

The following section describes the syntax for `shutdown()` and its parameters:

```
int err;
int sd, how;
err = shutdown(sd, how);
```

`sd` A socket descriptor for a listen socket or an SVC socket. The call affects the entire socket whether or not other socket descriptors are open on the socket.

`how` Describes the type of shutdown. Can be set to one of three possible values:

- `0`—disables data reception on the socket, but the connection is not cleared. If data is received on the connection, it is lost. The socket descriptor may still be used to read any unread data and transmit data.
- `1`—clears the connection, any unread data cannot be read using the socket descriptor.
- `2`—disables reception and transmission. A CLEAR REQUEST packet is sent on the SVC. Any unread data can be read.

On a listen socket all `how` values have the same effect—all requests in the listen queue are cleared and any requests received after the `shutdown()` are cleared.

`err`	Upon successful completion, `err` is set to 0. Otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

The `shutdown()` and `close()` procedures differ in that `shutdown()` takes effect regardless of the number of open socket descriptors, while `close()` takes effect only when the last process with an open socket descriptor makes the call.

Note The `shutdown()` call does not free internal system resources associated with the socket. To free this buffer space, you must issue a `close()` call.

Refer to the `shutdown(2)` entry in your `man` pages for more information.

4

Sending and Receiving Data

Overview

This chapter describes how applications can send and receive data on an SVC socket. The topics discussed include:

- Sending and Receiving Data
- Controlling the MDTF, D, and Q bits
- Using Nonblocking I/O
- Getting Next Message Status
- Setting the Buffer Thresholds and Sizes

This chapter assumes that the SVC has been established using the techniques described in chapter 3. Once the connection has been established, the distinctions between the client and server processes break down. The peer processes on either side of an SVC can transmit and receive data freely and equally.

Data transmission requirements

When an SVC has been connected, data can be sent and received over the SVC. The sending and receiving of data is identical for both sides of a connection regardless of whether the role of client or server was played when the connection was made.

An application process can use `send()` and `write()` system calls to transmit data, and use `recv()` and `read()` system calls to receive data, on an SVC. Using `read()` and `write()` has the advantage of being part of the UNIX standard I/O package. They are also simple to use. Using `send()` and `recv()` offers greater flexibility and control of data transmission. They are the only calls that can be used for data in INTERRUPT packets.

The M bit versus the MDTF bit

Data is transmitted and received as messages or message fragments. A message contains one or more DATA packets. All packets are separated by an M bit ("More" bit), except for the last packet.

If the MDTF bit is not used with the `ioctl(X25_SEND_TYPE)` system call, all of the packets which have been sent by a single `send()` or `write()` system call are sent as a single message. That is, X.25 divides the send or write buffer into packets and sends them in order with the M bit set in all but the last packet.

If the MDTF bit is specified with the `ioctl(X25_SEND_TYPE)`system call, longer messages can be sent with a single `send()` or `write()` system call.

Unless the `ioctl(X25_SET_FRAGMENT_SIZE)` system call is used with the MDTF bit, messages must be received with a single `recv()` or `read()` system call. If the `ioctl(X25_SET_FRAGMENT_SIZE)` system call is used, the message may be read as a series of message fragments, which allows extremely long messages to be read.

The `send()`, `recv()`, `read()`, and `write()` calls can be mixed within a program.

Sending Data

The `send(2)` and `write()` calls can be used anytime after a connection has been established. The caller is blocked until the specified number of bytes have been queued to be sent, unless you are using nonblocking I/O. Nonblocking I/O is described later in this chapter.

If the number of bytes to be transmitted is greater than the packet size permits, the data will be transmitted in a series of packets with the M bit set automatically. If you need to send a message which is longer than a single `send(2)` or `write(2)` buffer, you can control the use of the M bit with `ioctl(X25_SEND_TYPE)`.

The syntax for `write()` is fully described in your HP-UX `man` pages.

Syntax for send()

The syntax for the `send()` system call and its parameters are described below.

```
#include <sys/types.h>
#include <sys/socket.h>
int count;

int sd;
char *msg;
int len, flags;

count = send(sd, msg, len, flags);
```

`sd`	A socket descriptor for a connected SVC socket.
`msg`	A pointer to the buffer containing the data to be transmitted over the SVC. Although this is a character buffer, X.25 has no requirement that the data actually be ASCII characters. This is entirely up to the application.
`len`	The length of the `msg` buffer in bytes.
`flags`	Indicates the type of data and packet in which to send the data. If 0 is specified, then X.25 sends a DATA packet containing the data. If the out-of-band flag is set (OOB_MSG), then X.25 sends an INTERRUPT packet containing the data. For details on sending interrupt data, refer to chapter 5.
`count`	Is either –1 or the number of bytes actually sent. If count is –1, then `errno` contains the error code. If `errno` returned `EINTR`, a signal was received.

If a **SIGURG** signal is received during a `send()` call, the state of the VC may have changed. Design your program to check the state of the VC before attempting to `send()` data again after a SIGURG signal has arrived.

Refer to the `send(2)` entry in your HP-UX `man` pages for more on this.

Receiving Data

The `recv(2)` and `read(2)` calls can be issued any time after a connection has been established. The caller is blocked until there is a message available for reception, unless nonblocking I/O is in use. Nonblocking I/O is described later in this chapter.

If X.25 receives a series of packets with the M bit set, the packets must be read with a single system call, or as a series of message fragments. If the application is reading message fragments the `ioctl(X25_SET_FRAGMENT_SIZE)` must be set.

The syntax for `read()` is fully described in your HP-UX `man` pages.

Syntax for recv()

The syntax for the `recv()` system call and its parameters are described below.

```
#include <sys/types.h>
#include <sys/socket.h>
int count;

int sd;
char *buf;
int len, flags;

count = recv(sd, buf, len, flags);
```

`sd`	A socket descriptor for a connected SVC socket.
`buf`	A pointer to the buffer which will receive the data message. Although this is a character buffer, X.25 has no requirement that the data actually be ASCII characters. This is entirely up to the application.
`len`	The length of `buf` in bytes. If the message or message fragment available to be read is larger than the value specified in `len`, the remainder of the message or message fragment will be discarded. Once the data has been read

it is discarded. The `MSG_PEEK` allows a message to be received without any data being discarded. The number of bytes to be read can be obtained with `ioctl(X25_NEXT_MSG_STAT)`, described later in this chapter.

`flags`	Indicates the type of data to be read, and whether or not to discard the message after reception. If 0 is specified, then X.25 returns data transmitted in normal DATA packets. If the peek flag `MSG_PEEK` is set, the data is copied into the buffer, but not discarded afterwards. If the out-of-band flag is set `(OOB_MSG)`, then the call never blocks and X.25 returns data from the out-of-band queue. If `OOB_MSG` is set and no out-of-band data is available, `recv()` returns 0. For details on receiving interrupt data, refer to chapter 5.
`count`	Either –1 or the number of bytes actually copied into `buf`. If count is –1, then `errno` contains the error code. If `errno` returned `EINTR`, a signal was received. Correct the cause of the error before attempting to `recv()` data again. If the out-of-band flag is set (`OOB_MSG`), then the call never blocks and X.25 returns data form the out-of-band queue.

Unless nonblocking I/O is being used, `recv()` blocks until a complete X.25 message arrives, a signal arrives (for example, SIGURG), or the VC is terminated. Nonblocking I/O is described later in this chapter.

Controlling the MDTF, D, and Q bits

The MDTF, D, and Q bits are used to indicate special usage of DATA packets. X.25 allows applications to control the use of these bits with `ioctl(X25_SEND_TYPE)`.

When the M bit (more bit) is set in a DATA packet it indicates that the message requires one or more additional packets before it is completed. The M bit can be used automatically or under application control: all packets used to transmit a single `send()` or `write()` buffer are linked with the M bit.

When the D bit (Delivery Confirmation bit) is set in a DATA packet it requires that confirmation be sent upon its arrival at the remote DTE. To use the D bit in data transmission, the D bit must be set at connection time in the CALL REQUEST or CALL ACCEPTED packet.

The Q bit (Qualifier bit) is used to indicate a PAD CONTROL packet and is used when the remote DTE is a Packet Assembler/Disassembler (PAD). PAD CONTROL packets can control the operation of the PAD. PAD

CONTROL packets received indicate success or failure in the control operations sent. For a complete description of how to control a PAD and the meaning of the response packets, consult the CCITT X.29 and X.3 Recommendations.

The use of this bit is undefined if the remote DTE is not a PAD or emulating a PAD. Therefore, the application may use it for its own purposes when PAD CONTROL packets are not used on the SVC.

Syntax for ioctl(X25_SEND_TYPE)

The syntax for the `ioctl(X25_SEND_TYPE)` system call and its parameters are described below.

```
#include <x25/x25ioctls.h>
#include <x25/x25.h>
int err;
int sd, type;
err = ioctl(sd, X25_SEND_TYPE, &type);
```

`sd` A socket descriptor for an SVC socket. The socket need not be connected to an SVC when the `ioctl()` is issued.

`X25_SEND_TYPE` indicates the type of `ioctl()` being performed.

`type` Indicates which bits are being set with this `ioctl()`. HP supplies three predefined values which indicate the position of these bits within type: `X25_MDTF_BIT, X25_Q_BIT, and X25_D_BIT`. These values represent the bits' positions and not their actual placement; that is, they must be used in a shift operation.

For example, the following expression returns an integer with the D bit set: 1 << `X25_D_BIT.` The M, D, and Q bits can be set in the same message, but the use of the M and D bits in the same packet are subject to CCITT specifications. Typically, the D bit is not set in the same packet as the M bit.

Once a MDTF, D, or Q bit has been turned on with the `ioctl(X25_SEND_TYPE)`, it remains on until it is explicitly turned off with a subsequent call.

Using the MDTF Bit

The MDTF bit can be set automatically or controlled by the program. It is set automatically when you issue a `send()` or `write()` call and when you specify more data than a single packet can hold. The MDTF bit is automatically set to link all of the packets transmitted with a single system call.

To control the use of the MDTF bit use the `ioctl(X25_SEND_TYPE)` call with the `X25_MDTF_BIT` set (set to 1). All subsequent `send()` or `write()` calls will be treated as fragments of a large message. That is, all of the packets used to send the data have their MDTF bits set.

To clear the M bit on a long message use the `ioctl(X25_SEND_TYPE)` with the `X25_MDTF_BIT` call cleared (set to 0). This must be done before the program issues the last `send()` or `write()` call in the message series. The final packet of a long message does not have the M bit set.

Setting the D Bit in CALL REQUEST Packets

Use the `ioctl(X25_SEND_TYPE)`call to set the D bit in a CALL REQUEST packet. Once this is done issue the `connect()` call to transmit the CALL REQUEST packet.

The D bit is turned of by issuing the `ioctl(X25_SEND_TYPE)` call after the `connect()` call. If this is not done, the next message will have the D bit set.

Setting the D Bit in CALL ACCEPTED Packets

To set the D Bit in **CALL ACCEPTED** packets:

- First set the listen socket so that transmission of the CALL ACCEPTED packet is controlled by the application. Use the `ioctl(X25_CALL_ACPT_APPROVAL)` call.
- When the `accept()` call returns, issue `ioctl(X25_SEND_TYPE)` on the SVC socket to set the D bit for the transmission of the CALL ACCEPTED packet.
- Finally, send the CALL ACCEPTED packet with the `ioctl(X25_SEND_CALL_ACEPT)` call.

Turn off the D bit with the `ioctl(X25_SEND_TYPE)` call after the `ioctl(SEND_CALL_ACEPT)` call; otherwise, the next message you send will have the D bit set.

Setting the D Bit in a Data Message

To use the D bit in a DATA packet, the D bit must have been set at connection time in either the CALL REQUEST packet (on the calling side) or the CALL ACCEPTED packet (on the called side).

Issue the `ioctl(X25_SEND_TYPE)` call to set the D bit immediately prior to issuing a `send()` or `write()` call. In blocking mode, the process is blocked until confirmation is received. If the connection has been established to use the D bit, the application can set the D bit for any data message being transmitted. A data message may be a single packet or a set of packets with the M bit set on all but the last packet.

The D bit is set on every data message transmitted until it is cleared with a second `ioctl(X25_SEND_TYPE)` call.

Detecting D Bit Arrival and Confirmation

If you have not set the D bit during connection establishment, you will not be able to determine if the D bit is set on an incoming message with the `ioctl(X25_NEXT_MSG_STAT)` described later in this chapter. When a data packet arrives with the D bit set, X.25 Level 3 will acknowledge the arrival of the packet automatically. The X.25/300 subsystem acknowledges the D bit immediately upon reception of the packet. The X.25/800 subsystem acknowledges the D bit when the packet is received, without `MSG_PEEK`, by the application.

If the socket is in blocking mode, the process sending data blocks until D bit confirmation arrives. If the socket is in nonblocking mode, the X.25/800 subsystem sends an out-of-band event (`OOB_VC_DBIT_CONF`) to the process. The arrival of a D bit confirmation can also be detected with the `ioctl(X25_NEXT_MSG_STAT)`.

If the peer process is not using the X.25/800 subsystem, the D bit does not imply that the process has read the data. It implies only that the data has been received by the remote interface. If an end-to-end or disk-to-disk data confirmation protocol is required, it must be developed between the application processes.

Refer to the `socket_x25(7)` entry in your HP-UX `man` pages for more on `ioctl(X25_SEND_TYPE)`.

Using Nonblocking I/O

Sockets are created in blocking I/O mode by default. You can specify that a socket be put in nonblocking mode with the `ioctl(FIOSNBIO)` call.

Syntax for ioctl(FIOSNBIO)

The syntax for the `ioctl(FIOSNBIO)` system call and its parameters are described below.

```
#include <x25/x25ioctls.h>
int err;
int sd, arg;
err = ioctl(sd, FIOSNBIO, &arg);
```

`sd`	A socket descriptor for an SVC socket.
`FIOSNBIO`	Controls whether the socket is set to blocking or nonblocking I/O mode. This is specified by the value in `arg`.
`arg`	Specifies the socket's operating mode. If the value in `arg` is 0, the socket is in blocking mode. If the value in `arg` is 1, the socket is in nonblocking mode. Sockets are in blocking mode by default.
`err`	Upon successful completion, `err` is set to 0. Otherwise, a value of –1 is returned, and `errno` is set to indicate the error.

Nonblocking Behavior of System Calls

The behavior of many system calls changes when the socket is in nonblocking mode. The system calls of primary importance to X.25 programmers are the `accept()` and `connect()`, `send()` and `recv()`, `read()` and `write()`, and `ioctl()`. These differences are described below.

- `accept()` returns immediately. If no connection requests are present in the listen queue, `accept()` returns –1 and the EWOULDBLOCK error is contained in `errno`. If a connection request is present in the listen queue, `accept()` returns a socket descriptor for the SVC.
- `connect()` returns –1 immediately and the EINPROGRESS error is contained

in `errno`. The socket may be polled with the `select()` call; the socket will select writable when connection establishment is complete. At that point data can be retrieved from the CALL ACCEPTED packet with `ioctl(X25_RD_USER_DATA)`, and data can be sent and received with `send()` and `recv()` calls.

- `recv()` and `read()` returns immediately. If there is a complete message to be received in the X.25 buffer space, the data is returned immediately. If no data is available to be received, `recv()` and `read()` return the value –1 and the EWOULDBLOCK error is contained in `errno.`
- `send()` and `write()` returns immediately. If there is X.25 buffer space available, for the `send()` or `write()` buffer, the data is transferred into the X.25 buffer space and sent in the order it was received. If there is no available buffer space for the data to be transmitted, `send()` and `write()` return the value –1 and the EWOULDBLOCK error is placed in `errno`. In this case the call must be reissued at a later time.
- Series 800 only: `send()` and `write()` with the D Bit Set returns immediately. When the D bit is set, the data is transmitted as described above. When the D bit confirmation arrives the X.25 subsystem sends SIGURG signal to the process and the event `OOB_VC_DBIT_CONF` is added to the out-of-band queue. While waiting for D bit confirmation, the process must not send any more data until confirmation is received.
- `send(MSG_OOB)` returns immediately. If there is X.25 out-of-band buffer space available, the data is transferred into it and sent. If there is no available out-of-band buffer space for the data to be transmitted, `send()` returns the value –1 and the EWOULDBLOCK error is placed in `errno.` The call must be reissued at a later time. The use of `send()` to transmit out-of-band data is fully described in chapter 5.

When the INTERRUPT CONFIRMATION packet arrives, the subsystem sends a SIGURG signal to the process with `OOB_VC_INTERRUPT_CONF` in the out-of-band queue. The process must not send a second INTERRUPT packet until interrupt confirmation has been received or a reset indication has been received.

- `ioctl()` returns immediately. Most `ioctl()` values used by X.25 are not directly related to the transmission and reception of packets. The `ioctl(X25_RESET_VC)` and `ioctl(X25_SEND_CALL_ACEPT)`cause the transmission of packets. Both of these `ioctl()` types cause a packet to be sent. The `ioctl(X25_SEND_CALL_ACEPT)` never blocks because all transmit buffers are empty when it is issued. The socket is ready for transmission and reception.

When issued in nonblocking mode, the `ioctl(X25_RESET_VC)` returns 0 and a RESET REQUEST packet is sent on the VC. When a RESET CONFIRMATION packet is received, X.25 sends a SIGURG signal and a `OOB_VC_RESET_CONF` event is added to the out-of-band queue. No data of any sort may be sent by the process until the confirmation has been received.

Getting Next Message Status

You can use the `ioctl(X25_NEXT_MSG_STAT)` system call to obtain information about the next available message. `ioctl(X25_NEXT_MSG_STAT)` is almost always used when the `ioctl(X25_SET_FRAGMENT_SIZE)` is used. `ioctl(X25_NEXT_MSG_STAT)` returns the following information:

- Size of the next message or next message fragment
- Status of the MDTF, D or Q bits on the next message
- If call user data is available on the next message
- If clear data is available on the next message
- If a connection is established on the VC

Syntax for ioctl(X25_NEXT_MSG_STAT)

The syntax for the `ioctl(X25_NEXT_MSG_STAT)` system call and its parameters are described below.

```
#include <x25/x25ioctls.h>
#include <x25/x25str.h>
/* struct x25_msg_stat {
 *    int x25_msg_size
 *    int x25_msg_flags;
 * } x25_msg_stat;
 */
int err;

int sd;
struct x25_msg_stat status;
err = ioctl(sd, X25_NEXT_MSG_STAT, &status);
```

`sd`	A socket descriptor for an SVC socket.
`X25_NEXT_MSG_STAT`	Indicates that the status of the socket is being obtained.
`status`	Indicates the current status of the next message. If `status.x25_msg_size` is 0, there is no message in the queue; otherwise, it indicates the size of the next message or the next fragment of a

message to be read. This is useful to ensure that there is enough buffer space for the next read system call. The `status.x25_msg_flags` field indicates whether the D or Q bits were set in the next message to be read. The `status.x25_msg_flags` field also indicates if the next fragment is the last fragment in the message.

The M bit was set in the last packet of the last message fragment read (this means that this is a continuation of the previous message fragment). The position of these bits in the field are indicated by the `X25_MDTF_BIT`, `X25_D_BIT`, and `X25_Q_BIT` definitions.

`err`	Upon successful completion, `err` is set to 0. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

Setting Buffer Thresholds and Sizes

X.25 allows programmers to fine-tune socket behavior by specifying certain characteristics such as when a socket (is writable) allows information to be written to its buffers. This is accomplished by setting the threshold values that control the size of the three socket buffers: outbound message, send, and receive.

Setting the Write Buffer Threshold

The write buffer threshold is used to determine if there is enough buffer space available to send another message without blocking the socket. If the buffer space available is greater than or equal to the write threshold, the socket will indicate writable when a `select()` call is issued. The `ioctl(X25_WR_WTHRESHOLD)` call is used to set the write threshold value.

The value specified with `ioctl(X25_WR_THRESHOLD)` effects `send()`, `write()`, and `select()`. Whenever a `send()` or a `write()` call are issued, the amount of space remaining in the send socket buffer is checked; the amount of data in the outbound queue is subtracted from the size of the send socket's buffer. This check is performed in terms of network memory units and so is subject to round-off error. If this remaining space is insufficient to accept another message of the write threshold size, the free space size for the send socket is forced to zero, and remains that way until enough data is moved to the X.25 interface. If another `send()` or `write()` call is issued during this time, the call is blocked (for nonblocking I/O, EWOULDBLOCK is returned).

The `ioctl(X25_WR_WTHRESHOLD)` call and its parameters are described below.

Syntax for ioctl (X25_WR_WTHRESHOLD)

The syntax for the `ioctl(X25_WR_WTHRESHOLD)` system call and its parameters are described below.

```
#include <x25/x25ioctls.h>
#include <x25/x25str.h>
```

```
int err;
int sd, thresh;
err = ioctl(sd, X25_WR_WTHRESHOLD, &thresh);
```

`sd`	A socket descriptor for an SVC socket.
`X25_WR_WTHRESHOLD`	Indicates that the write threshold is being changed.
`thresh`	Indicates the new value for the write threshold.
`err`	Upon successful completion, `err` is set to 0. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

Setting the Read Message Fragment Size

X.25 assumes that all VCs will read whole messages with a single `read()` or `recv()` system call. This is usually the most efficient use of the VC. A message is a set of packets that all have their M bits set to 1 (except the last packet).

Connections over most VCs do not use extremely long messages, and the maximum size of a `read()` or `recv()` buffer is usually sufficiently large. However, if at any time during the connection the application anticipates that the VC will receive messages longer than the maximum `read()` or `recv()` buffer size, it must set the read message fragment size to a value greater than 0.

The `read()` and `recv()` calls return a whole number of packets, even when a message fragment is being read. This may cause the message fragment being read to be slightly longer or shorter than the fragment size specified in the `ioctl(X25_SET_FRAGMENT_SIZE)` system call. The `ioctl(X25_NEXT_MSG_STAT)` call is used to indicate the necessary buffer size in all instances.

When reading a long message requiring several `read()` or `recv()` calls, you must use the `ioctl(X25_NEXT_MSG_STAT)` call to detect the end of the message. When this call returns the `x25_msg_flags` value with the `X25_MDTF_BIT` set to 0, the next `read()` or `recv()` system call will return the end of the message.

To avoid data collision problems the `ioctl(X25_SET_FRAGMENT_SIZE)` call should be issued before the `connect()` call or between the `accept()` and the `ioctl(X25_SEND_CALL_ACEPT)` calls.

The `ioctl(X25_SET_FRAGMENT_SIZE)` call and its parameters are described below.

Syntax for ioctl(X25_SET_FRAGMENT_SIZE)

The syntax for the `ioctl(X25_SET_FRAGMENT_SIZE)` system call and its parameters are described below.

```
#include <x25/x25ioctls.h>
#include <x25/x25str.h>
int err;
int sd, size;
err = ioctl(sd, X25_SET_FRAGMENT_SIZE, &size);
```

`sd`	A socket descriptor for a VC socket.
`X25_SET_FRAGMENT_SIZE`	Indicates that the inbound message fragment size is being changed.
`size`	Indicates the new value for the read fragment size of messages. The range is from 0 to 32,767 where 0 indicates that all messages must be read with a single system call.
`err`	Upon successful completion, `err` is set to 0. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

Changing the Size of Socket Buffers

You can set the message sizes for a socket using the `setsockopt(2)` system call. It is important for the sockets to be able to handle the largest possible message that will be sent or received over the network. The default send and receive buffer size is 4096 bytes.

If your interface receives a message larger than the receive socket buffer size, the data will be discarded, the circuit will be reset, and you will receive an `OOB_VC_MSG_TOO_BIG` out-of-band event.

You can increase the size of a socket's `send()` or `recv()` buffer at any time, but they can only be reduced before a connection is established.

Syntax for setsockopt()

The syntax for the `setsockopt()` system call and its parameters are described below.

```
#include <sys/types.h>
#include <sys/socket.h>
int err;
int sd, level, optname, optval, optlen;
optlen = sizeof(int);
err = setsockopt(sd, level, optname,(char*) &optval,
optlen);
```

`sd`	A socket descriptor for an SVC socket.
`level`	Indicates the level at which the socket takes effect. The definition `SOL_SOCKET` should be specified.
`optname`	Indicates the type of option to be modified. This can be `SO_SNDBUF` to change the size of the send buffer or `SO_RCVBUF` to change the size of the receive buffer.
`optval`	Indicates the new send or receive buffer sizes where the maximum size is 58,254.
`optlen`	Indicates the length in bytes of the `optval` parameter; that is, `sizeof(int)`.
`err`	Upon successful completion, `err` is set to 0. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

Increasing the send socket buffet size allows a user to send more data before the user's application blocks, waiting for more buffer space. If more than one message cannot be sent without the user waiting for a reply, the programmer may want to increase the send buffer size to allow enough room to send multiple messages.

NOTE Increasing the buffer size to send larger portions of data before the application blocks **may** increase throughput, but the best method of tuning performance is to experiment with various buffer sizes.

Refer to the `setsockopt(2)` entry in your HP-UX `man` pages for more information.

5

Receiving and Transmitting Out-of-Band Information

Overview

This chapter describes how to send and receive out-of-band events. Many of the out-of-band events are associated with the arrival or transmission of a particular type of packet. However, in some cases, out-of-band events are independent of a particular packet type. The topics covered in this chapter include:

- Receiving Out-of-Band Events
- Building a Signal Handler
- Transmitting Out-of-Band Events

Receiving Out-of-Band Events

Out-of-band events, such as RESET INDICATION, INTERRUPT, and CLEAR INDICATION packets, occur during typical X.25 VC operation.

When an out-of-band event occurs, X.25 indicates the event by sending a SIGURG signal, and places a description of the event in the out-of-band queue.

WARNING **Applications using X.25 programmatic access should be designed to receive the SIGURG signal and process the information associated with the event. If the process has not attached a signal handler to receive this signal, the signal is ignored. Failure to respond to some out-of-band events (a RESET packet for example) may result in the VC being cleared by the network provider.**

Out-of-band events are placed in the out-of-band queue regardless of whether a signal handler has been installed. While it is possible to program applications to periodically examine the out-of-band queue to obtain any out-of-band events that have occurred, it is not recommended.

Signal Reception

Signal handling under HP-UX is controlled by the `signal(2)`, `sigsetmask(2)`, and `sigvector(2)` system calls. These system calls are described in your *HP-UX man pages*. The `sigvector()` system call allows you to specify a signal handler (signal catcher) to process signals when they arrive.

The following example shows how a signal handler may be installed to receive the SIGURG signal:

```
struct sigvec vec;
int onurg();
int pid, s;

/*
** arrange for the onurg() signal handler to be called
when SIGURG is received:
*/
vec.sv_handler = onurg;
vec.sv_mask; = 0
vec.sv_onstack = 0;
if (sigvector(SIGURG, &vec,  0) < 0) {
  perror("sigvector(SIGURG)");
}
```

In addition to installing the signal handler, you must also call `ioctl(SIOCSPGRP)` to ensure that the SIGURG signal is delivered upon receipt of the out-of-bound data as shown in the code example below. Refer to the `socket(7)` man page for more information about the `ioctl(SIOCSPGRP)` call.

```
setsigskt(s)
int s;
{
int pid;
/* enables the current process to receive SIGURG
 * when the socket has urgent data;
 */
pid = getpid();
/* Note that specifying the process id in the next ioctl()
 * means that only this process shall receive the SIGURG
 * signal. If (-1)*getpid() is used instead, the entire
 * process group (including
 * parents and children) will receive the SIGURG signal.
 */
if (ioctl(s, SIOCSPGRP, (char *) &pid) < 0) {
  perror ("ioctl(SIOCSPGRP)");
  }
```

Once installed, the signal handler is called whenever the specified signal arrives. While a signal handler is executing, additional signals of the same type are blocked from arrival. All signals sent by a socket should be handled by the process which is controlling the VC. This ensures process continuity.

Building a Signal Handler

A signal handler is a routine that is called when a signal arrives. The signal handler executes in the process address space and all global data items are available to the signal handler. The process is halted while the signal handler executes. When the signal handler returns, the process resumes execution at the point where it was halted.

There are four steps that a signal handler should perform each time it is executed:

1 Obtain the cause for the signal being sent.

To obtain the out-of-band event that caused the signal handler to be executed in the first place, use the `recv()` system call with the `MSG_OOB` flag set. `recv(MSG_OOB)` returns a buffer. The first byte in the buffer contains the number of bytes in the event (range: 3 to 34 bytes). The second byte contains the event code (described below) and the rest of the buffer contains the event data, if any.

The `recv(MSG_OOB)` call is nonblocking. If the `recv()` call returns 0, then no out-of-band events were queued. If the call returns a negative value, an error occurred. A value greater than 0 indicates the size of the buffer being returned.

2 Process the event which caused the signal to be sent.

After the cause of the event has been received, the signal handler should process the event. Usually out-of-band events have an effect on the state of the VC. For example, the arrival of a CLEAR INDICATION packets makes the VC unusable. This information should be made known to the main program. Typically, this is done with globally-defined state variables. The state variables can be tested by the main program, or examined only after a system call returns with an error. Many of the state changes of a VC are made known to the process through `errno` when a system call is executed.

The actual strategy used to pass information to the main program is up to the application designer.

Possible out-of-band events, and the appropriate actions for each, are described below.

3 Obtain any out-of-band events which may have arrived while the signal handler was executing.

No more than one signal of the same type can be blocked while a signal handler is operating. If a second signal arrives, the first is lost. To ensure there is no loss of data, once the signal handler is executed, it should obtain all of the data in the out-of-band queue. That is, issue `recv(MSG_OOB)` calls until a 0 is returned.

4 Return execution control to the main process.

Normal operation of the program is halted while the signal handler is executing. When it returns, processing resumes at the point at which it was interrupted.

`onurg()` is a routine that handles out-of-band events in the client program.

Example of an X.25 Signal Handler

NOTE For the purpose of simplicity, this example assumes only one socket; if you have more than one socket you can have the handler poll each socket in turn to see if OOB information has arrived.

```
#define MAX_EVENT_SIZE 34
/* Define maximum OOB message size: 32 + 1 byte for the
* packet type + 1 byte for the total event size */
/* Definitions for Out-of-Band events (OOB_INTERRUPT,
 * OOB_VC_CLEAR, OOB_VC_RESET and others) are stored in
 * x25.h
 */
onurg(skt)
int skt
{
    int error, s, n, buflen;
    unsigned buf[MAX_EVENT_SIZE];

    while (1)
    {
        buflen = MAX_EVENT_SIZE;
        if ((n = recv(skt, buf, buflen, MSG_OOB)) < 0)
        {
            perror("recv MSG_OOB")
            break;
        }
        else  if ( n == 0 ) break;
        else
        switch (buf[1])
           {
           case OOB_INTERRUPT:
             printf("INTERRUPT Packet Received\n");
             for (i = 2; i < n; i++)
             printf("%d “,buf[i]);
             printf(”\n");
             break;
           case OOB_VC_RESET:
             printf("RESET Packet Received\n");
             printf("Cause-code:%d\n",buf[2]);
             if (n >= 4) printf("Diagnostic code:
%d\n",buf[3])
             if (n >= 5) printf("Reason: %d\n",buf[4])
             break;
           case OOB_VC_CLEAR:
             printf("CLEAR Packet Received\n");
             printf("Cause-code:%d\n",buf[2]);
             if (n >= 4) printf("Diagnostic code:
%d\n",buf[3])
             if (n >= 5) printf("Reason: %d\n",buf[4])
             break;
           case OOB_VC_RESET_CONF:
          printf("RESET CONFIRMATION Packet Received\n");
             break;
           case OOB_VC_INTERRUPT_CONF:
             printf("INTERUPT CONFIRMATION Packet
Received\n");
             break;
           case OOB_VC_DBIT_CONF:
          printf("D-Bit Confirmation Packet Received\n");
             break;
           case OOB_VC_MSG_TOO_BIG:
             printf("Message Larger Than Inbound Buffer
Received\n");
             break;
           case OOB_VC_L2_DOWN:
             printf("X.25 Level 2 is Down\n");
             break;
           }
    }
} /* onurg */
```

WARNING

If the out-of-band data is not read quickly, the out-of-band data queue could overflow. If the queue overflows, subsequent out-of-band events are discarded.

The Out-of-Band Events

There are eight out-of-band (OOB) events which a signal handler can receive. The out-of-band values are defined in the `x25.h` include (program header) file. The out-of-band values are:

- OOB_INTERRUPT
- OOB_VC_CLEAR
- OOB_VC_DBIT_CONF
- OOB_VC_INTERRUPT_CONF
- OOB_VC_L2DOWN
- OOB_VC_MSG_TOO_BIG
- OOB_VC_RESET
- OOB_VC_RESET_CONF

The out-of-band events are described below.

OOB_INTERRUPT

An INTERRUPT packet was received. The confirmation is sent by the subsystem when reading this event, if the `MSG_PEEK` flag was not set.

The buffer received may contain from 3 to 34 bytes of data. The first 2 bytes of the buffer are its length and the event code respectively. The remainder of the buffer contains the interrupt data. The use of the interrupt data is application-dependent.

NOTE The maximum number of bytes of interrupt data depends on the version of X.25 recommendations being used by the network. The 1980 X.25 recommendations permit 1 byte of interrupt data while the 1984 X.25 recommendations permit up to 32 bytes of interrupt data.

OOB_VC_CLEAR

A CLEAR INDICATION packet was received on the SVC: the SVC can no longer send or receive data, and is closed. The `OOB_VC_CLEAR` is always the last event in the out-of-band queue.

You can use the `recv(MSG_OOB)` call to read out-of-band data that arrived before the CLEAR INDICATION packet. You can also use the `ioctl(X25_RD_USER_DATA)` (described below) to read cleared user data, if any exists. In addition, a CLEAR INDICATION packet may contain a facilities field, that can be examined with the `ioctl(X25_RD_FACILITIES)` call (see “The ioctl(X25_RD_FACILITIES) Call” on page 93).

You can get information on the state of the interface with the `ioctl(X25_GET_IFSTATE)` call.

The first byte contains the length of the buffer, and the second byte contains the clear indication. The third byte `buf[2]` contains the cause code. The fifth byte `byte[4]` always contains 0 (zero).

Table 6

Value	Reason
0	CLEAR INDICATION packet sent by network provider, subsystem, or X.25 device driver.

OOB_VC_DBIT_CONF

The D bit confirmation was received on a socket in nonblocking mode. This event can only be received on sockets in nonblocking mode, because a `send()` or `write()` with the D bit set blocks until confirmation is received. For more on D bit usage see “Controlling the MDTF, D, and Q bits” on page 56. The first byte of the buffer contains the length and the second byte contains the event code.

OOB_VC_INTERRUPT_CONF

Confirmation to a previously-sent INTERRUPT packet was received on a socket in nonblocking mode. This event can only be received on sockets in nonblocking mode, because `send(OOB_MSG)` blocks until interrupt confirmation is received.

The first 2 bytes of `buf[]` contain the buffer length and the event code respectively.

OOB_VC_L2DOWN

X.25 Level 2 is down. This event is returned only on a socket accessing a permanent virtual circuit (PVC); on an SVC, a clear indication is sent. Recover by issuing a `close()` on the socket, create another socket with the `socket()` system call, and binding the new socket to a PVC with the `ioctl(X25_SET_PVC)` call. Level 2 will probably not come up immediately. Level 2 usually goes down because of high noise on the line for a sustained period of time (see “Using Permanent Virtual Circuits” on page 106 for a description of PVC usage).

The first 2 bytes of `buf[]` contain the buffer length and the event code respectively.

OOB_VC_MSG_TOO_BIG

This event usually means that a message larger than the inbound buffer size was received. The data was therefore discarded and the VC was reset. This event occurs when the receiving side’s inbound buffer size is set to a value too small to receive the message. In this case you must increase this value with the `setsockopt()` system call to ensure that the buffer size is sufficient to receive any message which may arrive, or use the `ioctl(X25_SET_FRAGMENT_SIZE)` to enable the reception of message fragments.

If a `shutdown(0)` is issued on the socket and a DATA packet is received, a `OOB_VC_MSG_TOO_BIG` event will be delivered to the process and a CLEAR packet will be sent on the SVC.

The first 2 bytes of `buf[]` contain the buffer length and the event code respectively.

OOB_VC_RESET

A RESET INDICATION packet was received on the VC. Reading this event, without the `MSG_PEEK` flag set, causes a RESET CONFIRMATION packet to be sent, unblocking the sending process. A RESET can also be sent by the X.25 network provider, in which case, both ends of the VC will receive RESET INDICATION packets.

When a RESET INDICATION packet is received, the out-of-band queue is destroyed, and the RESET INDICATION data is placed in the out-of-band queue as its only event. All data that is not sent or received (normal and out-of-band data) is discarded. You must read the RESET INDICATION data before you can send further data on that connection.

The first 2 bytes of `buf[]` contain the buffer length and the event code respectively. The third byte `buf[2]` contains the cause code. The fourth byte `buf[3]` contains the diagnostic code. If the diagnostic code was not present in the packet, 0 is specified. The fifth byte `buf[4]` contains the reason the RESET INDICATION packet was sent. A 0 value indicates that the RESET INDICATION packet was sent by the network provider. Other values may be specified in the future.

After you have read the out-of-band data, you can proceed sending normal data. The connection is then ready to send and receive more data. Application programmers must provide a recovery mechanism to handle the loss of data that can occur due to a RESET INDICATION.

OOB_VC_RESET_CONF

Confirmation to a previously-sent `ioctl(X25_RESET_VC)` was received on a socket in nonblocking mode. This event can only be received on sockets in nonblocking mode, because in blocking mode the `ioctl(X25_RESET_VC)` blocks until confirmation is received.

The first 2 bytes of `buf[]` contain the buffer length and the event code respectively.

Transmitting Out-of-Band Events

There are three types of out-of-band events that can be sent by the X.25 application: CLEAR packets, RESET packets, and INTERRUPT packets. These packets can be sent at any time as the needs of the program dictate. While chapter 3 discussed the use of the `close()` system call for the transmission of CLEAR packets, this chapter covers the use of the cause and diagnostic codes, as well as the facilities and user data fields.

Clearing a Switched Virtual Circuit

An SVC can be cleared when one of the processes that use it (X.25 or the network provider) issues a CLEAR REQUEST packet. A CLEAR REQUEST packet can be sent by the X.25 application (with a `close()` call) otherwise it will be sent if the process terminates without executing a `close()` call.

Once a CLEAR REQUEST packet is issued, data cannot be sent or received over the connection because the connection is destroyed. The X.25 application can control the contents of certain data fields in the CLEAR REQUEST packet. These fields and the `ioctl()` calls which control them are:

- cause code—`ioctl(X25_WR_CAUSE_DIAG)`
- diagnostic code—`ioctl(X25_WR_CAUSE_DIAG)`
- facilities—`ioctl(X25_WR_FACILITIES)`
- user data—`ioctl(X25_WR_USER_DATA)`

All of these calls must be issued prior to the `close()` call. The `ioctl(X25_WR_CAUSE_DIAG)` and `ioctl(X25_WR_USER_DATA)` calls are described below. The `ioctl(X25_WR_FACILITIES)` call is described in chapter 6.

The ioctl(X25_WR_CAUSE_DIAG) Call

The cause code that is sent depends upon the cause code specified with the `ioctl(X25_WR_CAUSE_DIAG)`call and the network type specified during X.25 interface configuration. The table below summarizes cause code settings.

Table 7 **Setting Cause Codes**

Network Type	Cause Specified	Cause Sent
DTE 84	any	as specified
DCE (1980 or 1984)	any	as specified
DTE 80	any	0

The `ioctl(X25_WR_CAUSE_DIAG)` call must be issued prior to a RESET or CLEAR packet in order to set cause and diagnostic codes.

NOTE: When using this call over a PDN (Public Data Network) avoid using the same cause codes as those already employed by the network. This will facilitate eventual troubleshooting by eliminating ambiguity regarding the originator of RESET packets.

Syntax for ioctl(X25_WR_CAUSE_DIAG)

The `ioctl(X25_WR_CAUSE_DIAG)` call and its parameters are described below.

```
#include <x25/x25ioctls.h>
#include <x25/x25str.h>
/* struct x25_cause_diag {
 *     u_char x25_cd_loc_rem;
 *     u_char x25_cd_cause;
 *     u_char x25_cd_diag;
 *     }
 */
int err;
int sd;
struct x25_cause_diag diag;
err = ioctl(sd, X25_WR_CAUSE_DIAG, &diag);
```

`sd`	A socket descriptor for an SVC socket.
`X25_WR_CAUSE_DIAG`	The definition for the request.
`diag`	Indicates the cause and diagnostic codes to be sent.
`err`	Upon successful completion, `err` is set to 0. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

The ioctl(X25_WR_USER_DATA) Call

The `ioctl(X25_WR_USER_DATA)` call is used to write data to the user data fields in CLEAR REQUEST, CALL REQUEST, and CALL ACCEPTED packets. Call acceptance must be in effect before the call can be used on a CALL ACCEPTED packet. You can use this call in any of the these three situations:

- Before issuing a `connect()` call (to write call user data to the CALL REQUEST packet).
- Before issuing an `ioctl(X25_SEND_CALL_ACCEPT)`call when call-accept approval is in effect on the `listen()` socket (to write call user data to the CALL ACCEPTED packet).
- Before issuing a `close()` call (to write clear user data to the CLEAR REQUEST packet, when the fast select facility must be in effect), or `shutdown()` with the how parameter set to 1 or 2.

NOTE The call user data field in CALL REQUEST and CALL ACCEPTED packets can be up to 16 bytes unless the fast select facility is specified. In which case it may contain up to 128 bytes. When the fast select facility has been specified, the clear user data field of a CLEAR REQUEST packet may contain 128 bytes of data.

The `ioctl(X25_RD_USER_DATA)` call cannot be used to read data previously written with the `ioctl(X25_WR_USER_DATA)` call.

Syntax for ioctl(X25_WR_USER_DATA)

The `ioctl(X25_WR_USER_DATA)` call and its parameters are described below.

```
#include <x25/x25ioctls.h>
```

```
#include <x25/x25str.h>
/* define X25_MAX_CU_LEN 126
 * struct x25_userdata {
 *     u_char x25_cud_len;
 *     u_char x25_cud_data[X25_MAX_CU_LEN];
 *     }
 */
int err;
int sd;
struct x25_userdata udata;
err = ioctl(sd, X25_WR_USER_DATA, &udata);
```

`sd`	A socket descriptor for an SVC socket.
`X25_WR_USER_DATA`	The definition for the request.
`udata`	Contains the length and the data for the user data field (`x25_cud_data`). The maximum length is 126 bytes. If you want to send more than this, the `ioctl()` call must be called again. Usually the maximum length of the call user data field is 16 bytes. If the fast select facility is specified, the maximum length of the call user data field is 128 bytes.
`err`	Upon successful completion, `err` is set to 0. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

WARNING **Avoid using 0xCC in the first byte of the call user data field. If this is not possible, the interface must not be used for X.25/IP. If the first byte of an incoming call's call user data field contains 0xCC, and it arrives over an X.25 interface which has been assigned an IP address (see the x25init command in your *X.25/9000 User's Guide*), then the X.25 subsystem will assume the call is meant for the X.25/IP interface, and not the programmatic access interface.**

The ioctl(X25_RD_USER_DATA) Call

The `ioctl(X25_RD_USER_DATA)` call can read any data in the clear user data field of a CLEAR INDICATION packet or the call user data field of an incoming CALL INDICATION or CALL CONNECTED packet. This call is best used with the `ioctl(X25_NEXT_MSG_STAT)` call. The `ioctl(X25_NEXT_MSG_STAT)` call can determine whether call user data is available in the next message, and whether or not clear user data is available.

There may not be any user data available or there may only be clear or call user data available. Also both clear and call user data may be available. Use `ioctl(X25_NEXT_MESG_STAT)` to determine if both are available. If both call user data and clear user data are available, you should issue an `ioctl(X25_RD_USER_DATA)` call sequence to read the call user data first, until `ioctl(X25_NEXT_MSG_STAT)` says there is no longer call user data available, then issue an additional `ioctl(X25_RD_USER_DATA)` call sequence to read the clear user data.

The `ioctl(X25_RD_USER_DATA)` call can read a maximum of 126 bytes of call or clear user data (one byte must contain the length) per `ioctl()` call. If more than 126 bytes of call or clear user data is received, the `ioctl(X25_RD_USER_DATA)` must be called, until returned buffer length is 0.

Syntax for ioctl(X25_RD_USER_DATA)

The syntax for the `ioctl(X25_RD_USER_DATA)` call and its parameters are described below.

```
#include <x25/x25ioctls.h>
#include <x25/x25str.h>
/* struct x25_userdata {
 *     u_char x25_cud_len;
 *     u_char x25_cud_data[X25_MAX_CU_LEN];
 */    }
int err;
int sd;
struct x25_userdata udata;
err = ioctl(sd, X25_RD_USER_DATA, &udata);
```

`sd`	A socket descriptor for an SVC socket over which a CLEAR INDICATION packet, CALL INDICATION packet, or CALL CONNECTED packet has been received.
`X25_RD_USER_DATA`	The definition for the request.
`udata`	Contains the length of and data for the user data field, where the maximum length is 126 bytes. If there is more than 126 bytes of call or clear user data, the `ioctl()` must be called, until `udata.x25_cud_len` is 0.
`err`	Upon successful completion, `err` is set to 0. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

The code example below shows how to use the `ioctl(X25_RD_USER_DATA)` call to obtain the data from a CALL or CLEAR packet.

```
struct x25_userdata userdata;
struct x25_msg_stat msgstat;
unsigned char call_udata[128], clear_udata[128];
int i, call_ndata = 0,clear_ndata = 0, error = 0;
while ( !error )
    {
    error = ioctl(s, X25_NEXT_MSG_STAT, &msgstat);
    if (error< 0 ) {
       perror("ioctl(X25_NEXT_MSG_STAT) failed");
       exit(1);
       }
   if (msgstat.x25_msg_flags & (1 << X25_CA_DATA_AVAIL))
        {
        error = ioctl(s, X25_RD_USER_DATA, &userdata);
        if (error != 0)
            {
            perror("X25_RD_USER_DATA returned error");
            break;
            }
        if (userdata.x25_cud_len == 0) break;
            for (i = 0; i < userdata.x25_cud_len; i++)
                call_udata[call_ndata++] =
userdata.x25_cu_data[i];
        }else break;
    }
/*
 * If error = 0 at this point all of the call data is now
in “call_udata”.
 * “call_ndata” gives the number of bytes of data.
 */
while ( !error )
    {
    error = ioctl(s, X25_NEXT_MSG_STAT, &msgstat);
    if (error != 0 )
        {
        perror("ioctl(X25_NEXT_MSG_STAT) failed");
        exit(1);
        }
   if (msgstat.x25_msg_flags & (1 << X25_CL_DATA_AVAIL))
        {
        error = ioctl(s, X25_RD_USER_DATA, &userdata);
        if (error != 0)
            {
            perror("X25_RD_USER_DATA returned error");
            break;
            }
        if (userdata.x25_cud_len == 0) break;
            for (i = 0; i < userdata.x25_cud_len; i++)
                clear_udata[clear_ndata++] =
userdata.x25_cu_data[i];
        }else break;
    }
/*
 * If error = 0 at this point all of the clear data is now
in “clear_udata”.
 * “clear_ndata” gives the number of bytes of data.
 */
```

Resetting a Virtual Circuit

This section describes how to send RESET packets. A RESET packet clears all data on a connection, including any inbound data queued but not yet read, while maintaining the virtual circuit. The user application must be prepared to handle RESET packets; the handling of RESET packets at the application level is not defined by X.25. RESET packets can be generated by the X.25 subsystem, by the application processes on either side of the VC, or by the network provider.

If the user application wants to send a RESET packet, the application can set a cause code and diagnostic code with the `ioctl(X25_WR_CAUSE_DIAG)` call, then issue a RESET packet with the `ioctl(X25_RESET_VC)`. The `ioctl(X25_WR_CAUSE_DIAG)` call is described above under “Clearing a Switched Virtual Circuit”.

The ioctl(X25_RESET_VC) Call

The `ioctl(X25_RESET_VC)` call sends a RESET REQUEST packet over the virtual circuit. A VC cannot be reset before a connection is established or after a connection has been cleared.

When a user application resets a circuit, and blocking I/O is in effect, the application is blocked until a RESET CONFIRMATION packet is received.

The X.25 subsystem sends a RESET CONFIRMATION packet when the application reads the RESET out-of-band event without the MSG_PEEK flag set.

If nonblocking I/O is in effect, the `ioctl(X25_RESET_VC)` call does not block the caller. When the RESET CONFIRMATION packet arrives and the socket is in the nonblocking state, an out-of-band event `(OOB_RESET_CONF)` is placed in the socket's out-of-band queue. Also, `SIGURG` is sent to the process, which must have a signal handler installed for the `SIGURG` signal and enabled the socket to transmit `SIGURG` signals.

Syntax for ioctl(X25_RESET_VC)

The `ioctl(X25_RESET_VC)` and its parameters are described below.

```
int err;
int sd;
err = ioctl(sd, X25_RESET_VC, 0);
```

`sd`	A socket descriptor for an SVC socket.
`X25_RESET_VC`	The definition for the request which cause a RESET packet to be sent on the VC.
`0`	A dummy parameter for the system call.
`err`	Upon successful completion, `err` is set to 0. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

WARNING **All X.25 applications should be designed to handle out-of-band information. Failure to do so can cause SVCs to be cleared and PVCs to become unusable until your application process is killed.**

Sending Interrupts on a VC

An INTERRUPT packet can be sent at any time by either side of a VC after call connection has been established.

X.25 supports sending up to 32 bytes of interrupt data in an INTERRUPT packet.

To send an INTERRUPT packet, issue the `send()` system call with the `MSG_OOB` flag set. If blocking I/O is being used, the application process remains blocked until the INTERRUPT CONFIRMATION packet is received. If nonblocking I/O is being used, the confirmation is received as an out-of-band event (`OOB_VC_INTERRUPT_CONF`). See chapter 4 for a description of how to send interrupt data.

NOTE CCITT 1980 X.25 networks and equipment only support one byte of interrupt data. If your X.25 interface is configured in the `x25init(1M)` network type file as 1980 version, and you attempt to send more than 1 byte of interrupt data, an EMSGSIZE error will be returned. If the network provider will not support more than 1 byte of interrupt data, and you try to send more, the circuit will be reset. The network type file is described in your *X.25/9000 User's Guide*.

6

Extended Features

Overview

This section discusses the extended features available through X.25. The topics include:

- Using Facilities
- Using Fast Select
- Using Permanent Virtual Circuits
- Reestablishing Terminated Connections
- Obtaining Programmatic Diagnostics and Status

Using Facilities

X.25 permits the negotiation of Facilities that can be implemented when calls are connected or cleared.

Facilities are specified in the following packet types:

- CALL REQUEST/CALL INDICATION,
- CALL ACCEPTED/CALL CONNECTED,
- CLEAR REQUEST/CLEAR INDICATION,
- CLEAR CONFIRMATION.

X.25 includes the default Facility settings specified at configuration time for CALL REQUEST packets. The packet's Facility field is read and written with the `ioctl: X25_RD_FACILITIES` and the `X25_WR_FACILITIES` calls, respectively.

Consult your system administrator for the X.25 configuration constraints, and the documentation supplied by your network provider for more information about the Facilities permitted on the network.

The ioctl(X25_RD_FACILITIES) Call

The `ioctl(X25_RD_FACILITIES)` call returns the contents of the Facilities field contained in the last inbound packet containing a Facilities field. New inbound Facilities data overwrites preceding data. Inbound Facilities data can only be read once and storage resources are freed after the data is read. If the process tries to read inbound Facilities data that has already been read, or if no Facilities data is available, the `ioctl()` call returns a data length of 0.

The `ioctl(X25_RD_FACILITIES)` call:

- only reads Facilities data from inbound packets
- can read Facilities data from a CLEAR INDICATION packet (useful with Fast Select described below)

A process cannot use the `ioctl(X25_RD_FACILITIES)` call to read Facilities data that it has just written with the `ioctl(X25_WR_FACILITIES)` call.

Facilities may be added, deleted, rearranged or modified by the X.25 subsystem, the packet switching equipment, or the network provider.

Flow control negotiation, throughput class negotiation, and fast select/reverse charging are subject to X.25 configuration constraints. Some network providers also place further constraints on the Facilities that may be used.

The Facilities supported for `ioctl(X25_RD_FACILITIES)` are those described in the CCITT X.25 Recommendation (1988). These include:

- **Flow Control Parameter Negotiation**—(Packet size and Window size). When a flow control parameter negotiation facility is received and after the Packet Layer finishes the negotiation, the values are such that no further negotiation can take place. The facilities are not available with the ioctl(X25_RD_FACILITIES), for example 0x430302 with a configured window size of 2 2 is not reported by the ioctl, since the final values are 2 and 2 which is the Standard default.
- **Closed User Group Selection**—When either a closed user group selection basic or extended format, or a closed user group with outgoing access basic or extended format facility is received, it is always reported as a Closed user group selection extended format.
- **Transit Delay Selection**—There is no way to know if the Transit delay selection is received.
- **End-to-end Transit Delay Selection**—The cumulative end-to-end transit delay as well as the maximum acceptable end-to-end transit delay is forwarded. The requested end-to-end transit delay is not available and replaced with the unknown transit delay (0xffff).

Syntax for ioctl(X25_RD_FACILITIES)

The `ioctl(X25_RD_FACILITIES)` **and its parameters are described below.**

```
#include <x25/x25ioctls.h>
#include <x25/x25str.h>
/* define X25_MAX_FACILITIES_LEN 109
 * struct x25_facilities {
 *    u_char x25_fac_len; /* length of facilities field */
 *     u_char x25_fac[X25_MAX_FACILITIES_LEN]
 *     /* facilities field, exactly as in packet */
 *     }
*/
int err;
int sd;
struct x25_facilities fac_data;
err = ioctl(sd, X25_RD_FACILITIES, &fac_data);
```

`sd`	A socket descriptor for an SVC socket that has just received a CALL INDICATION, CALL CONNECTED, or CLEAR INDICATION packet.
`X25_RD_FACILITIES`	The definition for the request.
`fac_data`	Indicates the Facilities from the packet received. The `x25_fac[]` field contains the Facilities in the format that is required by the X.25 recommendations.

NOTE If the `ioctl(X25_RD_FACILITIES)`call is issued before the connection is initiated, the `ioctl()` system call will fail, –1 is returned and errno contains an indication of the cause of the error.

The ioctl (X25_WR_FACILITIES) Call

When the `ioctl(X25_WR_FACILITIES)` call is issued it writes the contents of the Facilities field for the next transmitted CALL REQUEST and CALL ACCEPTED packets. For a call to be accepted on the server side of a connection with the `ioctl()` call, call acceptance approval must be in effect. On the client side of the connection, you must issue the `ioctl(X25_WR_FACILITIES)` call prior to the `connect()` call.

When you use the `ioctl(X25_WR_FACILITIES)`call, you:

- can only write Facilities data for the next outbound CALL REQUEST, CALL ACCEPTED or CLEAR REQUEST packet,
- override any inbound Facilities data that has not already been read,
- must specify all of the default settings in order to modify any of the default Facilities.

If no Facilities are specified with the `ioctl(X25_WR_FACILITIES)` call, the default configuration is set.

The `ioctl(X25_WR_FACILITIES)` call will overwrite any inbound Facilities data that have not already been read.

It is possible to modify Facilities such as negotiated parameters. Facilities may be added, deleted, or rearranged by the network provider or packet switching equipment.

The Facilities supported for `ioctl(X25_WR_FACILITIES)`, are those specified in the CCITT and X.25 (1988) recommendation. These include:

- **Flow Control Parameter Negotiation**—If your configuration specifies that flow control negotiation is ON and Flow Control Facilities are not requested with the `ioctl(X25_WR_FACILITIES)` call, your default values for Packet Size and Window Size Facilities are inserted in the packet.
- **Throughput class negotiation**—not inserted in the packet unless it is requested, and there is no default configured.
- **Fast Select / Reverse Charging**—not inserted in the call or call accepted packet if 0x0100 is requested (neither fast select or reverse charging is requested).
- **Charging information requesting service**—if 0x0400 is requested, no charging information is inserted in the call or call accepted packet.
- **RPOA Selection extended format**—If only one DNIC is specified, the `ioctl(X25_WR_FACILITIES)` returns an error (errno `EINVAL=22`). Use the basic format instead. The extended format is only available if more than one DNIC is specified.
- **Transit Delay Selection**—this Facility is used to request a specific end-to-end transit delay (a Packet Layer Protocol function). If present, the value is based on the actual end-to-end transit time (minus the system processing time).
- **End-to-End Transit Delay Selection**—this Facility overrides the Transit Delay Selection parameter (above) with a constant corresponding to the system's data processing time. The requested end-to-end transit delay and maximum acceptable end-to-end transit delay values are sent without modification to packets. If this is the only parameter provided and the Transit Delay Selection parameter is not requested, no information is inserted in the packet.
- **Expedited Data Negotiation**—not inserted in the packet if requested after a 0x0b00 marker.
- **Network Facilities**—these facilities must have a Facility marker just before them.

Network Facilities

Network Facilities are the facilities that need a Facility marker just before them. The only legal Facility markers are those described in the 1988 CCITT X.25 Recommendations and in the 1987 ISO 8208 Standard.

Legal Facilities markers are:

- 0x000f —for CCITT specified DTE Facilities used to support the OSI Network service.
- 0x0000 —for non-X.25 Facilities supported by the local network for an **internal** network call, or for non-X.25 Facilities supported by the network of the calling DTE for an **internetwork** call.
- 0x00ff— for non-X.25 Facilities supported by the network of the called DTE for an **internetwork** call.

NOTE Network Facilities entries must be provided in the above order with the X.25 Facilities first. The length of the Network Facility entry is limited to 32 Bytes. Illegal Facility codes return the EINVAL(22) error condition.

Syntax for ioctl(X25_WR_FACILITIES)

The `ioctl(X25_WR_FACILITIES)` **call and its parameters are described below.**

```
#include <x25/x25ioctls.h>
#include <x25/x25str.h>
/* define X25_MAX_FACILITIES_LEN 109
 * struct x25_facilities {
 *     u_char x25_fac_len;
 *     u_char x25_fac[X25_MAX_FACILITIES_LEN];
 *     }
 */
int err;
int sd;
struct x25_facilities fac_data;
err = ioctl(sd, X25_WR_FACILITIES, &fac_data);
```

`sd`	A socket descriptor for an SVC socket that has not yet been connected.
`X25_WR_ FACILITIES`	The definition for the request.

`fac_data`	Indicates the Facilities to be included in the CALL REQUEST, CALL ACCEPTED, or CLEAR REQUEST packet. The `x25_fac[]` field contains the Facilities in the exact format that is required by the X.25 recommendations.

Using Fast Select

The use of fast select in X.25 is described in *X.25: The PSN Connection*. This section describes how to access this Facility through X.25.

The fast select Facility allows processes to specify up to 128 bytes of call user data in the CALL REQUEST, CALL ACCEPTED, and CLEAR REQUEST packets. Usually, only 16 bytes are permitted in the CALL REQUEST packet and no user data in the CALL ACCEPTED and CLEAR REQUEST packets. This Facility is called “fast” because it allows programmers to specify user data in the same packet which is requesting or acknowledging a connection request. Usually, the CALL REQUEST packet contains all of the information the server side needs to service the request, and it terminates the connection and places all the information that the client needs regarding the results of its request in the CLEAR REQUEST packet. The circuit is never fully set up, but is then quickly torn down, hence the term “fast select”.

The `ioctl()` calls used for fast select are listed below:

- `X25_WR_USER_DATA` (for writing extended call user data).
- `X25_RD_USER_DATA` (for reading extended call user data).
- `X25_WR_FACILITIES` (for writing Facilities).
- `X25_RD_FACILITIES` (for reading Facilities).

A summary of the steps needed to enable fast select on the calling and receiving side of a connection appears in the next two subsections.

Fast Select on the Calling Side

To implement fast select on the calling side, issue the `ioctl(X25_WR_FACILITIES)` call with the fast select Facilities code defined in CCITT X.25 recommendations. Specify the call user data for the packet with the `ioctl(X25_WR_USER_DATA)`. These calls must be executed before issuing the `connect()` call.

If the calling side is using protocol IDs for address matching, the protocol ID must be specified in the call user data field before any other call user data is added with the `ioctl(X25_WR_USER_DATA)` call. These data may be specified in separate `ioctl()` calls, or combined into one, but the protocol

ID must come first. Remember that the protocol ID length is not defined by the CCITT. Programmers of the two processes must agree on the number of bits of the call user data field to use for the protocol ID before connection establishment begins.

When the Facilities and call user data fields have been specified for the fast select call, the calling side issues a connection request with `connect()` and waits for the response from the remote server.

Fast Select on the Called Side

The called side may respond in three ways:

- If not accepting fast select calls, the called side will return a CLEAR REQUEST packet with no data.
- If accepting fast select calls, the called side may execute an `ioctl(X25_WR_USER_DATA)` call to send data back to the calling side, then return a CLEAR REQUEST packet with data by executing a `close()` call. The called side can also return a call accept (with data) by executing `ioctl(x25_SEND_CALL ACCEPT)`. The called side may return clear-user data or call user data to the calling side only if the called side is controlling call acceptance with `ioctl(X25_CALL_ACPT_APPROVAL)`.
- If fast select was indicated in the Facilities field with no restrictions on response, the called side may simply accept the call, and read the call user data with the `ioctl(X25_RD_USER_DATA)`. The connection is then considered an ordinary virtual circuit; that is, fast select is no longer in effect.

The maximum length for the user data field in the CLEAR REQUEST packet is 128 bytes. The server may execute the `ioctl(X25_WR_USER_DATA)` more than once to write the entire user data field. If more than 128 bytes must be transmitted, the called side (server) of the connection should respond with a CALL ACCEPTED packet and transmit the entire response with normal DATA packets, or the called side must simply clear the connection completely by calling `close()`.

Details regarding fast select are defined in the CCITT X.25 Recommendations, and X.25 configuration information is presented in your *X.25/9000 User's Guide.*

Fast Select Operation Summary

A server can respond to a fast select call in any of three ways:

- The call can be cleared immediately with no data
- The call can be cleared with user data specified
- A normal SVC can be established with or without user data specified in the call accept packet

Which of these alternatives are available depends on the design of the client and server processes.

This section is written with the assumption that interfaces for both the client and the server are configured to permit fast select, and the network provider allows fast select.

The actions to be taken are shown in the three scenarios below. The initial steps for sending and receiving fast select is the same for all scenarios. These initial steps are shown in Table 8. Depending on the design of the application, the server may take any of three actions that are described later.

Initial Steps for Fast Select

The initial steps performed by the client and server when configured to permit fast select are described below and shown in Table 8 on the following page.

1 The server creates an AF_CCITT listen socket.

2 The server binds an `x25addrstr` structure containing the addressing information.

3 The server establishes a listen queue.

4 The server prepares the call to control call acceptance.

5 The server listens for a CALL REQUEST packet.

6 The client creates an AF_CCITT socket.

7 The client specifies the fast select Facility for the CALL REQUEST packet.

8 The client specifies the call user data for the CALL REQUEST packet.

9 The client sends the CALL REQUEST packet.

10 The server's `accept()` call unblocks when it receives the CALL INDICATION packet.

11 The server reads and parses the Facilities field.

12 The server reads and interprets the user data field.

Table 8 **Steps for Fast Select**

Server Events	X.25 Events	Client Events
1. `socket()`	No Event	No Event
2. `bind()`	No Event	No Event
3. `ioctl(X25_CALL_ACPT_APPROVAL)`	No Event	No Event
4. `listen()`	No Event	No Event
5. `accept()` blocks	No Event	No Event
6. No Event	No Event	`socket()`
7. No Event	No Event	`ioctl(X25_WR_FACILITIES)`
8. No Event	No Event	`ioctl(X25_WR_USER_DATA)`
9. No Event	CALL REQUEST packet transmitted	`connect()` blocks
10. `accept()` unblocks	CALL INDICATION packet received	No Event
11. `ioctl(X25_RD_FACILITIES)`	No Event	No Event
12. `ioctl(X25_RD_USER_DATA)`	No Event	No Event

After the above steps have been done, the server has the information it needs to process the fast select. Three actions are possible. They are discussed individually in the next three subsections.

Clearing a Fast Select Call Immediately

If the server detects that it cannot handle the fast select call for any reason, it can clear the circuit immediately. Steps for this are described below and in Table 9. Steps shown in Table 8 are assumed to have already occurred.

1 The server transmits the CLEAR REQUEST packet.

2 The client receives the CLEAR INDICATION packet and takes appropriate actions.

Table 9 **Clearing a Fast Select Call Immediately**

Server Events	X.25 Events	*Client Events*
1. `close()`	CLEAR REQUEST packet transmit-ted	No Event
2. No Event	CLEAR INDICATION packed received	`connect()` returns -1 (`ECONNREFUSED`)

Transmitting a CLEAR Packet with User Data

The server can respond to a fast select call by sending a CLEAR REQUEST packet that contains a user data field. Steps for this are described below and shown in Table 10. Steps shown in Table 8 are assumed to have already occurred.

1 The server writes the user data field with information appropriate to respond to the fast select call.

2 The server transmits the CLEAR REQUEST packet.

3 The client receives the CLEAR INDICATION packet and takes appropriate actions.

Table 10 **Sending Clear User Data for a Fast Select**

Server Events	X.25 Events	Client Events
1. `ioctl(X25_WR_USER_DATA)`	No Event	No Event
2. `close()`	CLEAR REQUEST packet transmitted	No Event
3. No Event	CLEAR INDICATION packed received	`connect()` returns -1 (`ECONNREFUSED`)

Establishing an SVC in Response to a Fast Select Call

The server can establish an SVC in response to a fast select call. Steps for this are described below and in Table 11. Steps shown in Table 8 are assumed to have already occurred. The fast select Facility must be set for no *restrictions* on *response* for this option to be used.

1 The server optionally writes the user data field with information appropriate to indicate the reason for its response to the fast select call.

2 The server transmits the CALL ACCEPTED packet.

3 The client receives the CALL CONNECTED packet.

4 Both the server and client have the option to write user data and carry on normal SVC data transmission with DATA packets.

5 The connection is closed by the client or server (see “Terminating a Connection” on page 47).

Table 11 **Fast Select Connection Establishment**

Server Events	X.25 Events	Client Events
1. `ioctl(X25_WR_USER_DATA)`	No Event	No Event
2. `ioctl(X25_SEND_CALL_ACPT)`	CALL ACCEPTED packet transmitted	No Event

Table 11 **Fast Select Connection Establishment**

3. No Event	CALL INDICATION packet received	`connect()` unblocks (returns 0)
4. `read()` and `write()`	DATA packets transmitted and received	`read()` and `write()`
5. `close()`	CLEAR REQUEST/ INDICATION packet transmit-ted and received	`close()`

Using Permanent Virtual Circuits

Permanent virtual circuits (PVCs) must be acquired from the network provider. They are set-up during X.25 configuration. Refer to your *X.25/9000 User's Guide* for details on configuring PVCs.

A PVC is similar to a "leased line" in that it is always connected and terminates at a single destination. By contrast, an SVC can terminate at different destinations, depending upon the call. A PVC may be used by only one socket at a time.

Although a PVC can be RESET, so all data on the connection is discarded, it cannot be cleared like an SVC. A PVC is connected as long as X.25 Level 3 is active.

A socket is bound to a PVC with the `ioctl(X25_SETUP_PVC)` call. Unlike the case of an SVC set-up, the `listen()`, `connect()`, and `accept()` system calls are not used to establish a connection over a PVC. The `close()` call removes the binding between a socket and the circuit of a PVC and also sends a RESET REQUEST packet out on the circuit. Data is cleared from the circuit, but the circuit is not destroyed.

The `ioctl()` calls that are associated with sending and receiving CALL REQUEST and CALL ACCEPTED packets, and the fields within them, such as `ioctl(X25_RD_USER_DATA)`, `ioctl(X25_WR_USER_DATA)`, `ioctl(X25_WR_FACILITIES)`, and `ioctl(X25_RD_FACILITIES)`have no effect on a PVC.

Preparing a PVC for Use

A PVC is connected from the moment the programmatic access interface is active. However, until a socket is bound to the PVC, X.25 discards all packets received over the PVC and a RESET REQUEST packet is sent. There is no implicit client/server relationship in establishing a PVC connection. Actions taken on either end of a PVC are essentially the same.

The `ioctl(X25_SETUP_PVC)` call binds a socket to a PVC. The application first creates the socket, then calls the `ioctl(X25_SETUP_PVC)`. The logical channel identifier (lci) of the PVC must be known before the `ioctl(X25_SETUP_PVC)` is issued. This can be obtained from the system administrator.

If the `ioctl(X25_SETUP_PVC)` call is successful, the socket is bound to the PVC as a dedicated socket. However, there is no assurance that there is any process running at the other end of the PVC. Programs should send a message to say that they are alive and then wait for a reply from the remote end of the PVC. Once a process is operational at the remote end of the PVC, it can be used, like a SVC, to send and receive data and out-of-band information.

The primary difference between PVCs and SVCs is that a CLEAR REQUEST packet cannot be sent or received on a PVC. A remote peer process may abort or die without notice. X.25 sends a RESET REQUEST packet on the PVC if the socket is closed for any reason, for example, the local process dies. However, RESET REQUEST packets can be received on a PVC for other reasons. Refer to the discussion of `OOB_VC_L2DOWN` in chapter 5 for how a PVC recovers when Level 2 fails.

Syntax for ioctl(X25_SETUP_PVC)

The `ioctl(X25_SETUP_PVC)` and its parameters are described below.

```
#include <x25/x25ioctls.h>
#include <x25/x25str.h>
/* define X25_MAX_IFNAMELEN 12
 * struct x25_setup_pvc_str {
 *     char ifname[X25_MAX_IFNAMELEN+1];
 *     int lci;


*     }
 */
int err;
int sd;
struct x25_setup_pvc_str pvc_str;
err = ioctl(sd, X25_SETUP_PVC, &pvc_str);
```

`sd`	Is a socket descriptor for an AF_CCITT socket that has not yet been bound to an address.

`X25_SETUP_PVC`	Is the definition for the request.
`pvc_str`	Indicates the interface and logical channel indicator for the PVC to which the socket shall be connected.
`err`	If the call succeeds, 0 is returned. If the call fails, –1 is returned and errno contains an indication of the cause of the error. `ENETUNREACH` indicates Level 2 is down. `ENODEV` indicates the named interface does not exist. `ENETDOWN` indicates the interface is down because it has been shutdown, suffered either a power fail without being reinitialized or incurred a hard error. `EINVAL` indicates the virtual circuit is illegal or not a PVC. `EBUSY` indicates that the lci is in use.

Reestablishing Terminated Connections

Maintaining a connection over an X.25 network may not suit all applications. For example, a client may send a request to a server that may require processing for an extended period of time before a reply can be sent. It would not be economical to maintain the connection until data processing has been completed. A more cost-effective method is to clear the connection, then reestablish it from the server side when the server finishes processing the data.

Usually, in a CALL REQUEST packet, only the X.121 address of the interface being used is written into the calling address field. The `ioctl(X25_WR_CALLING_SUBADDR)` writes a subaddress as a suffix to the calling address field in the CALL REQUEST packet. The server can read this data and save it until the connection is reestablished.

If the server is using the X.25 interface, the `getpeername()` system call will return the subaddress information formatted in an `x25addrstr` structure. The server then stores the information (`x25addrstr` structure) until the server reestablishes the connection.

When protocol IDs are used, the reestablishment proceeds as discussed above. However, the server must know the protocol ID to which it will connect. The protocol ID can be stored in the user data field of the CALL REQUEST packet. This must be designed into the application on both side of the connection. There is no standard specifying the way to transfer the protocol ID of the caller through X.25.

Syntax for ioctl(X25_WR_CALLING_SUBADDR)

The `ioctl(X25_WR_CALLING_SUBADDR)` and its parameters are described below.

```
#include <x25/x25ioctls.h>
#include <x25/x25str.h>
/* define X25_MAX_HOSTADDR 15
 * define X25_MAXHP1 X25_MAX_HOSTADDR+1
 * define X25_MAX_PIDLEN 8
 * struct x25addrstr {
 *     u_char x25hostlen;
 *     u_char x25pidlen;
 *     u_char x25pid[X25_MAX_PID_LEN];
 *     u_char x25_host[X25_MAXHP1]p
 *     }
 */
int err;
int sd;
struct x25addrstr subaddr;
err = ioctl(sd, X25_WR_CALLING_SUBADDR, &subaddr);
```

`sd`	Is a socket descriptor for an SVC socket which has not yet been connected.
`X25_WR_CALLING_SUBADDR`	Is the definition for the request.
`subaddr`	Indicates the subaddress to be added to the calling address field in the CALL REQUEST packet. Only the `x25hostlen` and `x25_host` from the `x25addrstr` structure are used. The `x25hostlen` field contains only the subaddress to be added to the calling address field. The rest of the address is supplied by X.25.

Obtaining programmatic diagnostics and status

Diagnostics and status information on applications that use X.25 programmatic access can be obtained via the following features:

- Error codes and log messages.
- The `ioctl(X25_RD_CTI)` call (to get the circuit table index entry associated with a particular socket).
- The `ioctl(X25_RD_LCI)` call (to get the logical channel identifier associated with a particular virtual circuit).
- The `ioctl(X25_RD_HOSTADR)` call (to get the X.121 address of the interface).
- The `ioctl(X25_GET_IFSTATE)` call (to get the condition/state of the interface).
- Remote and local address information (`getpeername(2)` and `getsockname(2)`)

Each feature is described below.

Error codes and log messages

The most commonly-used programmatic diagnostic is the value returned by errno. Possible errno values returned for each call are listed in the `man` pages for this command.

In addition to the error codes returned, X.25 maintains a log which describes the activities of the interface and all active virtual circuits. For details on logging, refer to your X.25/9000 User’s Guide.

The ioctl (X25_RD_CTI) Call

The `ioctl(X25_RD_CTI)` call returns the circuit table index (cti) entry associated with a particular socket. This identifies one connection regardless of the interface that is used. The circuit table index entry is a sub-identification number (sid) which is also logged in `strace` output (see your `man` pages for `strace`). Once you know the cti entry for a socket, you can use that information when examining the X.25 log file for information concerning the VC. The cti is also useful for network logging. `EINVAL` is returned if the circuit is not connected or if the socket is no longer bound to a circuit. For details on logging, see your *X.25/9000 User's Guide*.

Syntax for ioctl(X25_RD_CTI)

```
#include <x25/x25ioctls.h>
#include <x25/x25str.h>
int err;
int sd, cti;
err = ioctl(sd, X25_RD_CTI, &cti);
```

`sd`	Is a socket descriptor for a connected SVC socket.
`X25_RD_CTI`	Is the definition for the request.
`cti`	Contains the circuit table index for the socket.

A programming example is shown below.

```
int error, s, cti;
/*
 * Assume at this point that s is a socket
 */
error = ioctl (s, X25_RD_CTI, &cti);
if (error 0) {
   perror("X25_RD_CTI");
   exit(1);
   }
```

The ioctl(X25_RD_LCI) Call

The **`ioctl(X25_RD_LCI)`** call returns the logical channel identifier (lci) associated with a particular virtual circuit. Once you know the lci entry for a circuit, you can use that information when examining statistics for a virtual circuit, with network logging and the corresponding data from a protocol-analyzer trace. The lci is the value written into the header of most CCITT X.25 packets.

Syntax for ioctl(X25_RD_LCI)

```
#include <x25/x25ioctls.h>
#include <x25/x25str.h>
int err;
int sd, lci;
err = ioctl(sd, X25_RD_LCI, &lci);
```

	`sd`
	Is a socket descriptor for a connected SVC socket.
`X25_RD_LCI`	Is the definition for the request.
`lci`	Contains the logical channel indicator for the socket.
`err`	Is set to 0 if the call was successful; otherwise, contains –1 and errno contains the cause of the error. If `EINVAL` is returned, the circuit is not connected or the socket is no longer bound to a circuit.

A programming example is shown below.

```
int error, s, lci;
/*
 * Assume at this point that s is a socket
 */
error = ioctl (s, X25_RD_LCI, &lci);
if (error < 0) {
perror("X25_RD_LCI");
exit(1);
}
/* Use the lci as the logical circuit indicator. */
```

The ioctl(X25_RD_HOSTADR) call

The `ioctl(X25_RD_HOSTADR)` call returns the X.121 address of the interface. This is the X.121 address of the interface (NOT the address inserted in the call packet)

Syntax for ioctl(X25_RD_HOSTADR)

The `ioctl(X25_RD_HOSTADR)` **and its parameters are described below.**

```
#include <x25/x25ioctls.h>
#include <x25/x25addrstr.h>
int error;
int sd;
struct x25addrstr addr;
error = ioctl(sd, X25_RD_HOSTADR, &addr);
```

`sd`	Is a socket descriptor for an X.25 socket. Note that this can be a completely new socket, a connected socket or a connected socket that has been shutdown
`X25_RD_HOSTADR`	Is the definition for the request.
`addr`	Indicates the X.25 interface name. `x25_family` must be set to AF_CCITT. `x25ifname` must be set to a name string (with "\0" at the end). The address is returned in `x25hostl()` and `x25hostlen`. `x25ifname` can be an empty string, but this is only advised when there is only one interface on the system; the interface address returned is unpredictable when there are several interfaces.
`error`	Is set to 0 if the call was successful; otherwise, contains –1 and errno contains the cause of the error.

A programming example is shown below.

```
/* Get X.25 parameters used for binding an address */
result = ioctl( call_soc, X25_RD_HOSTADR, &locl_addr);
if (result == -1) {
perror("ioctl(X25_RD_HOSTADR) call");
exit(1);
}
```

The ioctl(X25_GET_IFSTATE) call

The `ioctl(X25_GET_IFSTATE)` call returns the interface state.

Syntax for ioctl(X25_GET_IFSTATE)

```
#include <x25/x25ioctls.h>
#include <x25/x25str.h>
#include <x25/x25.h>
/*
 * Define data structure for X25_GET_IFSTATE
 */
/* struct x25_state_str {
 * char ifname[X25_MAX_IFNAMELEN+1];/* Name of X.25
interface to use */
 * int ifstate; /* Status of the X.25 interface */
 * };
 */
int error;
int sd;
struct x25_state_str state_str;
```

```
error = ioctl(sd, X25_GET_IFSTATE, &state_str);
```

`sd`	Is a socket descriptor for an X.25 socket. Note that this can be a completely new socket, a connected socket or a connected socket that has been shutdown.
`X25_GET_IFSTATE`	Is the definition for the request.
`state_str`	Contains the state of the interface.

The state of the interface is returned in `ifstate` and can take the following values:

IFSTATE_UNINIT	The interface has been stopped
IFSTATE_INIT	Level 3 is up (R1)
IFSTATE_L2DOWN	Level 3 is down (not R1)

`error`	Is set to 0 if the call was successful; otherwise, contains –1 and errno contains the cause of the error.

A programming example is shown below.

```
/* Get if state */
result = ioctl( call_soc, X25_GET_IFSTATE, &if_state);
if (result == -1) {
perror("ioctl(X25_GET_IFSTATE) call");
exit(1);
}
```

Obtaining Status Information Programmatically

X.25 uses two system calls that return information regarding the status of a circuit. They are `getpeername()` to obtain the address of the remote DTE, and `getsockname()` to obtain the address of the local DTE.

Using `getpeername()` after a call has been established, a process can obtain the caller's address from the calling address field of the CALL ACCEPTED/INDICATION packet.

Using `getsockname()` after a call has been established, a process can obtain the called address from the CALL ACCEPTED/INDICATION packet. `getsockname()` can be used to obtain the actual called address when wildcard addressing is being used.

Syntax for getpeername()

The `getpeername()` and its parameters are described below.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <x25/x25addrstr.h>

int error;
int sd, addrlen;
struct x25addrstr addr;
addrlen = sizeof(struct x25addrstr);
error = getpeername(sd, &addr, addrlen);
```

`sd`	Is the socket descriptor for the socket whose peer address will be obtained.
`addr`	Upon successful completion, this `x25addrstr` structure will contain the X.121 address of the remote DTE. This information is useful when using wildcard addressing and when reestablishing a terminated connection.
`addrlen`	Upon successful completion, this integer will contain the length in bytes of the `x25addrstr` structure pointed to by `addr`. Before calling `getpeername()`, this field must be initialized with the size of the `x25addrstr` structure.
`error`	If the call successfully completes, `error` contains a 0; otherwise, –1 is returned, and errno contains the cause of the error.

Refer to the `getpeername(2)` entry in `man` pages for more information on this command.

Syntax for getsockname()

The `getsockname()` and its parameters are described below.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <x25/x25addrstr.h>

int error;
int sd, addrlen;
struct x25addrstr addr;
addrlen = sizeof(struct x25addrstr);
error = getsockname(sd, &addr, &addrlen);
```

`sd`	Is the socket descriptor for the socket whose addressing information is being obtained.
`addr`	Upon successful completion, this `x25addrstr` structure will contain the X.121 address specified by the remote process to make the connection. This information is useful when using wildcard addressing.
`addrlen`	Upon successful completion, this integer will contain the length in bytes of the `x25addrstr` structure pointed to by `addr`. Before calling `getpeername()`, this field must be initialized with the size of the `x25addrstr` structure.
`error`	If the call successfully completes, error contains a 0; otherwise, –1 is returned, and errno contains the cause of the error.

Refer to the `getsockname(2)` entry in your `man` pages for more information.

A

X.25 Packet Formats

X25 Packet Formats

This appendix shows some of the packet formats specified in the CCITT X.25 Recommendations. It also describes the X.25 actions required to transmit the packet, and the X.25 events that occur when the packet is received. The fields of each packet are identified and the X.25 actions required to read and modify these fields are described.

Introduction

The packets described here are listed below:

- CALL REQUEST/INDICATION packet
- CALL ACCEPTED/CONNECTED packet
- CLEAR REQUEST/INDICATION packet
- DATA packet
- INTERRUPT packet
- INTERRUPT CONFIRMATION packet
- RESET REQUEST/INDICATION packet
- RESET CONFIRMATION packet

This appendix does not describe the use and range of possible values for the packets and their fields. It is here to describe how the X.25 applications can access the features defined in the X.25 recommendations.

Several packets are not accessible to X.25 applications. These packets are listed below:

- DIAGNOSTIC packet
- REJ (retransmission) packet
- REGISTRATION packet
- REGISTRATION CONFIRMATION packet
- CLEAR CONFIRMATION packet
- RECEIVE READY packet
- RECEIVE NOT READY packet

X.25 Level 3 handles these packets according to the X.25 recommendations without the need for any intervention by the user application. X.25 does not support REGISTRATION packets.

CALL REQUEST/INDICATION Packet

A client process sends a CALL REQUEST packet to establish an SVC with a remote host. This packet is transmitted when a `connect()` system call is issued. When the X.25 subsystem receives a CALL INDICATION packet and locates a socket whose bind address matches the specified called address, it allows the associated listen socket to unblock an `accept()` system call or select readable.

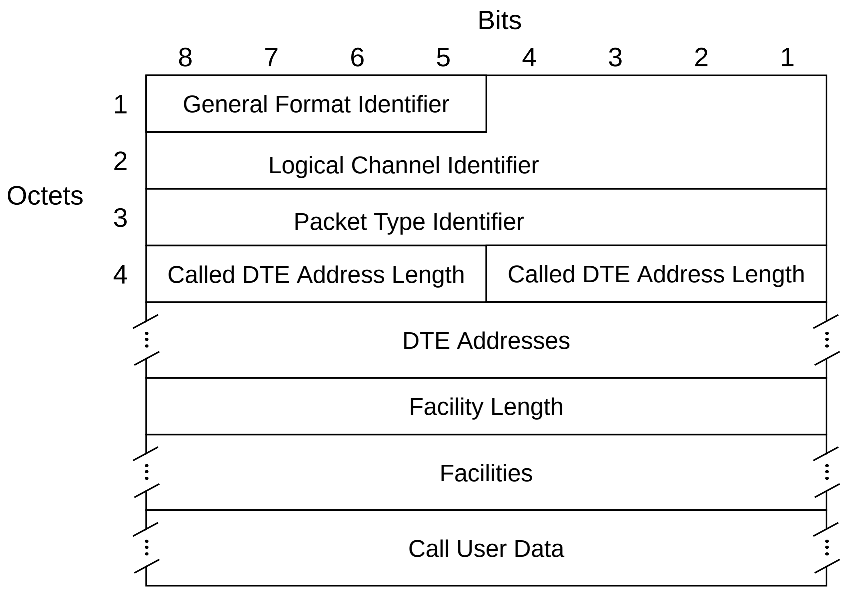
Bits
8
7
6
5
4
3
2
1
Octets
1
2
3
4
General Format Identifier
Logical Channel Identifier
Packet Type Identifier
Called DTE Address Length
Called DTE Address Length
DTE Addresses
Facility Length
Facilities
Call User Data

Figure 1 CALL REQUEST Packet

The access to the fields in the CALL REQUEST/INDICATION packet is described below:

General Format Identifier — Indicates D bit and packet sequence count. This field cannot be read by an X.25 application. The D bit cannot be read or written by an X.25/300 application. The packet sequence count (X.25 supports only modulo 8) is set at configuration time and cannot be read or written by the application. However, the configuration file can be read by the application.

To read this field on an incoming packet:

This field cannot be read by the application.

To write this field on an outgoing packet:

X.25 700/800 applications use the `ioctl(X25_SEND_TYPE)` to set the D bit.

Logical Channel Identifier

Associates a logical channel number with the SVC. This field is controlled by the X.25 subsystem. It can be read with the `ioctl(X25_RD_LCI)` after the call has been established.

Packet Type Identifier

Indicates the kind of packet. It cannot be read or written directly, but through system calls, the transmission and arrival of packets can be effected or detected.

To read this field on an incoming packet:
Use the `listen()` and `accept()` system calls to detect the arrival of a CALL REQUEST packet.

To write this field on an outgoing packet:
Use the `connect()` system call to transmit a CALL REQUEST packet.

Calling DTE Address (Length)

Indicates the source of the packet.

To read this field on an incoming packet:
These fields can be obtained through the `accept()` system call, which returns an `x25addrstr` structure containing the calling DTE address of the incoming CALL REQUEST packet.

To write this field on an outgoing packet:
The subsystem computes this field. If the local machine has multiple X.25 interfaces connected to it, the client can select which interface (each with a unique X.121 address) to use by specifying the `x25ifname` field in the `x25addrstr` structure. The address is the X.121 configuration packet address, plus the subaddress specified with `ioctl(X25_WR_CALLING_SUBADDR)` call.

Called DTE Address (Length)

Indicates the destination of the packet. This field is accessed through the `x25addrstr` structure.

To read this field on an incoming packet:
These fields can be read using the `getsockname()` system call after the connection has been established.

To write this field on an outgoing packet:
The subsystem computes this field from the `x25addrstr` supplied in the `connect()` system call.

Facilities (Length)	**Optional.** Indicates the facilities used for this SVC.

To read this fields on an incoming packet:
These fields can be read with the `ioctl(X25_RD_FACILITIES)`.

To write this field on an outgoing packet:
These fields can be written with the `ioctl(X25_WR_FACILITIES)`.

Call User Data	**Optional.** Contains a protocol ID and/or user defined data.

To read this field on an incoming packet:
The entire user data field including the protocol ID can be read with the `ioctl(X25_RD_USER_DATA)`.

To write this field on an outgoing packet:
The entire user data field including the protocol ID can be written with the `ioctl(X25_WR_USER_DATA)`.

CALL ACCEPTED/CONNECTED Packet

A CALL ACCEPTED/CONNECTED packet establishes an SVC with a remote host. A CALL ACCEPTED packet is sent in response to a CALL INDICATION packet. Typically, the X.25 subsystem sends a CALL ACCEPTED packet immediately in response to a CALL INDICATION packet when an accept system call is issued on a listen socket. However, if the `ioctl(X25_CALL_ACPT_APPROVAL)` has been issued on the listen socket, X.25 waits for the `ioctl(X25_SEND_CALL_ACEPT)` to be issued on the socket descriptor returned in the `accept()` system call.

The access to the fields in the CALL ACCEPTED/CONNECTED packet is described below:

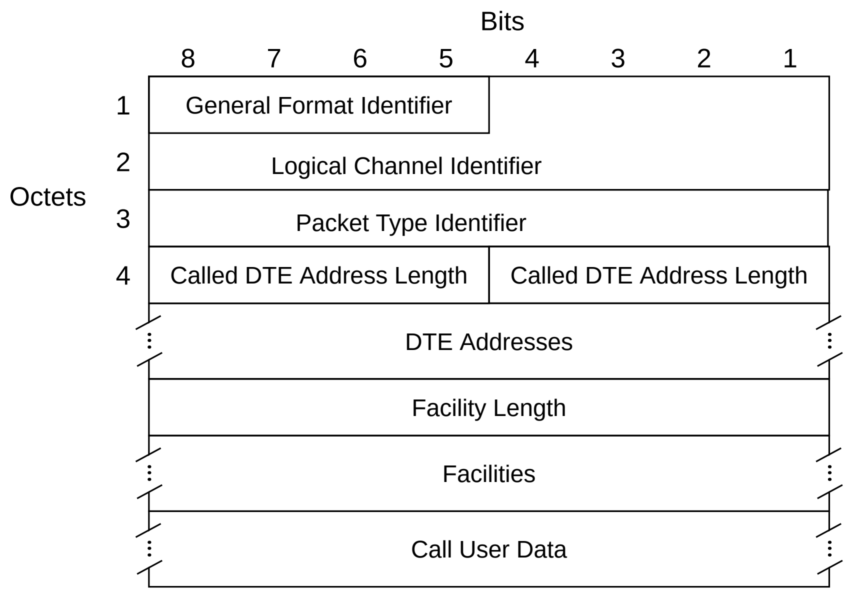
Bits
8
7
6
5
4
3
2
1
Octets
1
2
3
4
General Format Identifier
Logical Channel Identifier
Packet Type Identifier
Called DTE Address Length
Called DTE Address Length
DTE Addresses
Facility Length
Facilities
Call User Data

Figure 2 **CALL ACCEPTED Packet**

General Format Identifier	Indicates D bit usage and packet sequence count. This field cannot be read by an X.25 application. The D bit cannot be read or written by an X.25/300 application. The packet sequence count (modulo 8) is set at configuration time and cannot be read or written by the application. However, the configuration file can be read by the application. **To read this field on an incoming packet:** This field cannot be read by the application. **To write this field on an outgoing packet:** X.25/800 applications use the `ioctl(X25_SEND_TYPE)` to set the D bit.
Logical Channel Identifier	Associates a logical channel number with the SVC. This field is controlled by the X.25 subsystem. It can be read with the `ioctl(X25_RD_LCI)` after the call has been established.
Packet Type Identifier	Indicates the kind of packet. It cannot be read or written directly, but through system calls, the transmission and arrival can be effected or detected. **To read this field on an incoming packet:** The application can detect the arrival of this packet when the `connect()` system call unblocks or selects writable. **To write this field on an outgoing packet:** The subsystem automatically sends this packet when an `accept()` system call has been issued, unless the application is controlling call acceptance. In that case this packet is sent only after an `ioctl(X25_SEND_CALL_ACCEPT)` is issued on a socket that had `ioctl(X25_ACPT_APPROVAL)` issued on the listen socket.
Calling DTE Address (Length)	Indicates the source of the packet. **To read this field on an incoming packet:** These fields cannot be read by the application. **To write this field on an outgoing packet:** The X.25 subsystem computes this field.
Called DTE Address (Length)	Indicates the destination of the packet. They are accessed through the `x25addrstr` structure. **To read this field on an incoming packet:** These fields cannot be read by the application.

To write this field on an outgoing packet:
The X.25 subsystem computes this field from the information supplied in the CALL REQUEST packet.

Facilities (Length) — **Optional.** Indicates the facilities used for this SVC.

To read this field on an incoming packet:
These fields can be read with the `ioctl(X25_RD_FACILITIES)`.

To write this field on an outgoing packet:
Use the `ioctl(X25_WR_FACILITIES)` prior to issuing the `ioctl(X25_SEND_CALL_ACEPT)`. The `ioctl(X25_ACPT_APPROVAL)` must first be issued on the listen socket. Applications cannot control the access to the facilities field, if the call accept packet is generated using the `accept()` system call.

Call User Data — **Optional.** Contains application defined data.

To read this field on an incoming packet:
These fields can be read with the `ioctl(X25_RD_USER_DATA)`.

To write this field on an outgoing packet:
The application must be controlling call acceptance. Use the `ioctl(X25_WR_USER_DATA)` prior to issuing the `ioctl(X25_SEND_CALL_ACEPT)` the `ioctl(X25_ACPT_APPROVAL)` must first be issued on the listen socket. Applications cannot control the access to the call user data field, if the call accept packet is generated using the `accept()` system call.

CLEAR REQUEST/INDICATION Packet

A client process sends a CLEAR REQUEST/INDICATION packet to terminate an SVC with a remote host. The subsystem sends a CLEAR REQUEST/INDICATION packet when a close or `shutdown()` system call is issued.

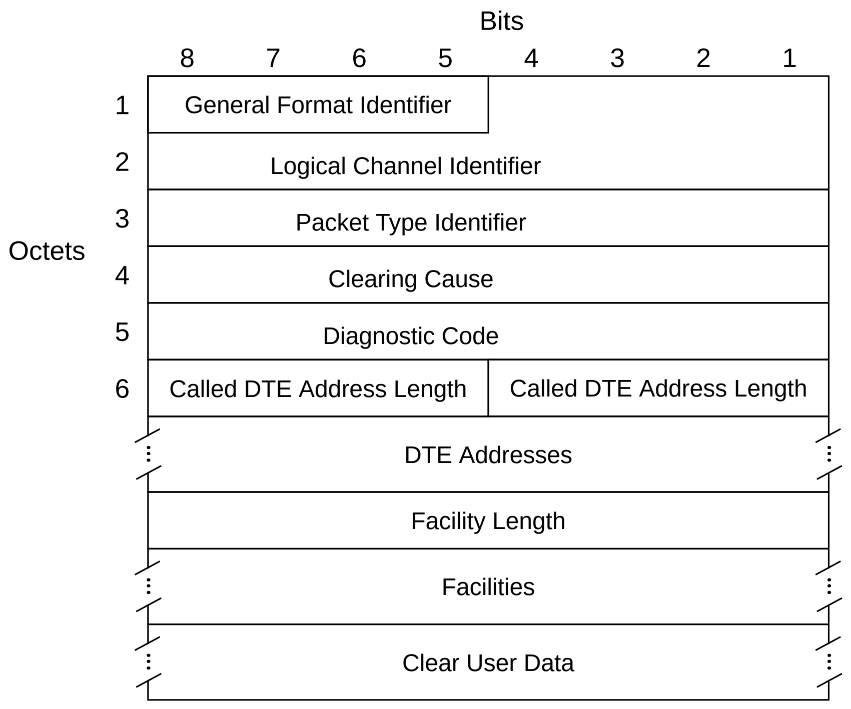

Figure 3 **CLEAR REQUEST Packet**

The access to the fields in the CLEAR REQUEST/INDICATION packet is described below:

General Format Identifier	Indicates the packet sequence count (modulo 8 or modulo 128). This field cannot be read or written by an X.25 application.
Logical Channel Identifier	Controlled by the subsystem. It can be read with the `ioctl(X25_RD_LCI)`after the call has been established.

Packet Type Identifier	Indicates the kind of packet. It cannot be read or written directly, but through system calls, the transmission and arrival can be effected. **To read this field on an incoming packet:** `recv(MSG_OOB)` returns a buffer the second byte of which (`buf[1]`) will contain `OOB_VC_CLEAR` if a CLEAR INDICATION packet was received. **To write this field on an outgoing packet:** Use the close or `shutdown()` system call to transmit a CLEAR REQUEST packet.
Clearing Cause	Indicates the cause code for the clear request packet. **To read this field on an incoming packet:** `recv(MSG_OOB)` returns a buffer the third byte of which (`buf[2]`) will contain the clearing cause field. **To write this field on an outgoing packet:** Use the `ioctl(X25_WR_CAUSE_DIAG)`.
Diagnostic Code	Contains the diagnostic code for the CLEAR REQUEST packet. **To read this field on an incoming packet:** `recv(MSG_OOB)` returns a buffer the fourth byte of which (`buf[4]`) will contain the clearing diagnostic field. **To write this field on an outgoing packet:** Use the `ioctl(X25_WR_CAUSE_DIAG)`.
Calling DTE Address (Length)	Indicates the source of the packet. **To read this field on an incoming packet:** These fields cannot be read by the application. **To write this field on an outgoing packet:** The X.25 subsystem computes this field. It cannot be directly written.
Called DTE Address (Length)	Indicates the destination of the packet. **To read this field on an incoming packet:** These fields cannot be read by the application. **To write this field on an outgoing packet:** The X.25 subsystem computes this field.
Facilities (Length)	**Optional**. Indicates the facilities used for this SVC.

To read this field on an incoming packet:
These fields can be read with the `ioctl(X25_RD_FACILITIES)`.

To write this field on an outgoing packet:
These fields can be written with the `ioctl(X25_WR_FACILITIES)`.

Call User Data

Optional. Contains application-defined data.

To read this field on an incoming packet:
Use the `ioctl(X25_RD_USER_DATA)` to read the data.

To write this field on an outgoing packet:
Use the `ioctl(X25_WR_USER_DATA)` to write this field.

DATA Packet

A client process sends a DATA packet to terminate an SVC with a remote server. The subsystem sends a DATA packet when a `send()` or `write()` system call is issued.

Figure 4

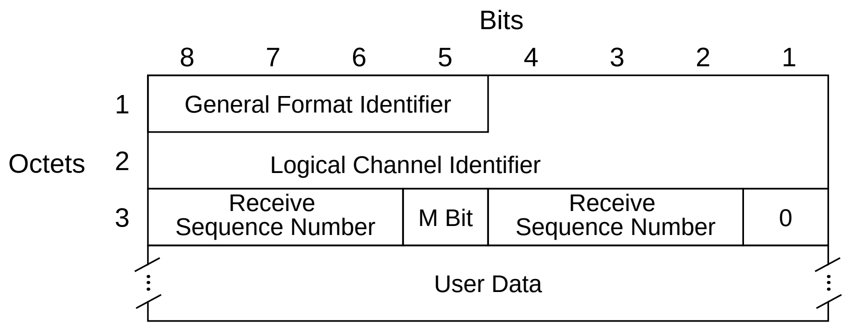

Figure 5 **DATA Packet (Modulo 8)**

The access to the fields in the DATA packet is described below:

General Format Identifier

Contains the D bit and the packet sequence count (modulo 8). The packet sequence window size cannot be read or written by an X.25 application.

To read this field on an incoming packet:
This field cannot be directly read by the application. The arrival of a DATA packet with the D bit set can be detected with the `ioctl(X25_NEXT_MSG_STAT)` system call. The X.25 subsystem automatically responds to an incoming DATA packet with the D bit set when data is read by the application (D bit confirmation indicates that the data has been read by the remote X.25 application.)

To write this field on an outgoing packet:
Use the `ioctl(X25_SEND_TYPE)` before issuing a `send()` or `write()` system call.

Logical Channel Identifier

Contains the logical channel identifier. It is controlled by the X.25 subsystem. It can be read with the `ioctl(X25_RD_LCI)` after the call has been established.

Receive Sequence Number	Indicates the sequence number of the last packet received. It is controlled by the subsystem and cannot be read nor written by the application.
M Bit	Indicates that the packet is part of a group of packets (a message). It is set whenever the buffer specified in a `send()` or `write()` system call is larger than the maximum packet size, or when the `X25_MDTF_BIT` is set in the `ioctl(X25_SEND_TYPE)` system call. Packets with the M bit set are part of a larger message. The entire message must be read by at a single `recv()` or `read()` system call, or as a series of fragments when the `ioctl(X25_SET_FRAGMENT_SIZE)` system call is used. This field cannot be directly read or written by an application.
Send Sequence Number	Indicates the sequence number of this packet. It is controlled by the sub-system and cannot be read nor written by the application.
User Data	Contains the data being transmitted in this packet.

To read this field on an incoming packet:
Use the `recv()` or `read()` system calls to read this field.

To write this field on an outgoing packet:
Use the `send()` or `write()` system calls to write this field.

INTERRUPT Packet

A application process sends an INTERRUPT packet to transmit out-of-band data to the remote process.

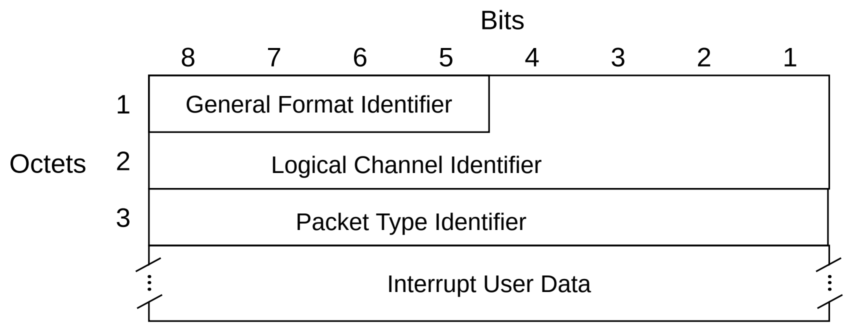

Figure 6 INTERRUPT Packet

The access to the fields in the INTERRUPT packet is described below:

General Format Identifier — Indicates the packet sequence count (modulo 8). This field cannot be read or written by an X.25 application.

Logical Channel Identifier — Associates a logical channel number with the SVC. This field is controlled by the subsystem. It can be read with the `ioctl(X25_RD_LCI)`.

Packet Type Identifier — Indicates the kind of packet. It cannot be read or written directly, but through system calls, the transmission and arrival can be effected.

To read this field on an incoming packet:
`recv(MSG_OOB)` returns a buffer the second byte of which (`buf[1]`) contains an indication that an INTERRUPT packet has arrived (OOB_INTERRUPT).

To write this field on an outgoing packet:
Use the `send(MSG_OOB)` system call to transmit an INTERRUPT packet.

Interrupt User Data — Contains data associated with the INTERRUPT packet.

To read this field on an incoming packet:
When the SIGURG signal handler is called, issue a `recv()` system call with the `MSG_OOB` flag set.

To write this field on an outgoing packet:
Use the `send()` system call with the `MSG_OOB` flag set to transmit an INTERRUPT packet and specify the interrupt user data field.

INTERRUPT CONFIRMATION Packet

This packet is transmitted automatically by the X.25/300 subsystem when an INTERRUPT packet is received. The X.25/800 subsystem transmits this packet when the application process reads the interrupt data (issues a `recv(MSG_OOB)` system call).

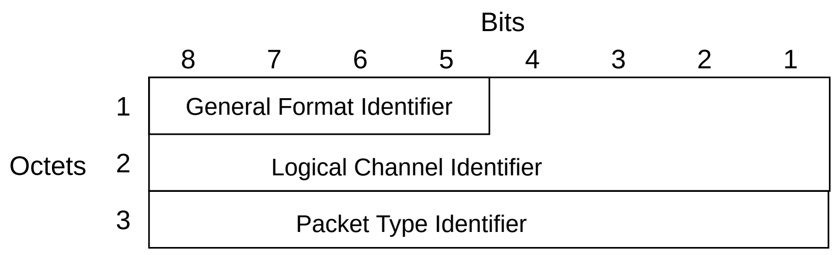

Figure 7 INTERRUPT CONFIRMATION Packet

The access to the fields in the INTERRUPT CONFIRMATION packet is described below:

General Format Identifier	Indicates the packet sequence count (modulo 8 or modulo 128). This field cannot be read or written by an X.25 application.
Logical Channel Identifier	Contains the logical channel identifier. It is controlled by the subsystem. It can be read with the `ioctl(X25_RD_LCI)` after the call has been established.
Packet Type Identifier	Indicates the kind of packet. X.25/300 automatically transmits this packet in response to an INTERRUPT packet, and transmission cannot be controlled by the application.

To read this field on an incoming packet:
In blocking mode, the `send(MSG_OOB)` system call blocks until confirmation is received. In nonblocking mode, `recv(MSG_OOB)` returns a buffer the second byte of which (`buf[1]`) indicates that an INTERRUPT CONFIRMATION packet has arrived (`OOB_VC_INTERRUPT_CONF`).

To write this field on an outgoing packet:
The X.25/800 subsystem transmits an INTERRUPT CONFIRMATION packet when the corresponding INTERRUPT packet's interrupt user data field is read with a `recv(MSG_OOB)` system call.

RESET REQUEST/INDICATION Packet

A process sends a RESET REQUEST packet to reset a VC. The subsystem sends a RESET REQUEST when a `close()` or `shutdown()` system call is issued on a PVC.

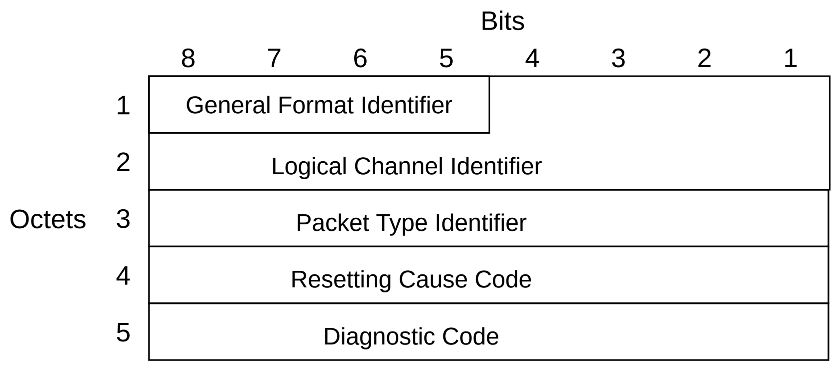

Figure 8 RESET REQUEST Packet

The access to the fields in the RESET REQUEST/INDICATION packet is described below:

General Format Identifier	Indicates the packet sequence count (modulo 8 or modulo 128). This field cannot be read or written by an X.25 application.
Logical Channel Identifier	Contains the logical channel identifier. It is controlled by the X.25 subsystem. It can be read with the `ioctl(X25_RD_LCI)` after the call has been established.
Packet Type Identifier	Indicates the kind of packet. It cannot be read or written directly, but through system calls, the transmission and arrival can be effected.

To read this field on an incoming packet:
`recv(MSG_OOB)` returns a buffer the second byte of which (`buf[1]`) contains an indication that a RESET INDICATION packet has arrived (`OOB_VC_RESET`).

To write this field on an outgoing packet:
Use the `ioctl(X25_RESET_VC)` to transmit a reset packet.

Resetting Cause Code	Indicates the cause code for the RESET REQUEST/INDICATION packet. **To read this field on an incoming packet:** `recv(MSG_OOB)` returns a buffer the second byte of which (`buf[2]`) contains the resetting cause field. **To write this field on an outgoing packet:** Use the `ioctl(X25_WR_CAUSE_DIAG)` before the `ioctl(X25_RESET_VC)` is issued.
Diagnostic Code	Contains the diagnostic code for the RESET REQUEST/INDICATION packet. **To read this field on an incoming packet:** `recv(MSG_OOB)` returns a buffer the second byte of which (`buf[3]`) contains the resetting diagnostic code field. **To write this field on an outgoing packet:** Use the `ioctl(X25_WR_CAUSE_DIAG)` before issuing the `ioctl(X25_RESET_VC)`.

RESET CONFIRMATION Packet

A RESET CONFIRMATION packet is automatically generated by the subsystem when the application process reads the reset data (issues a `recv(MSG_OOB)` system call).

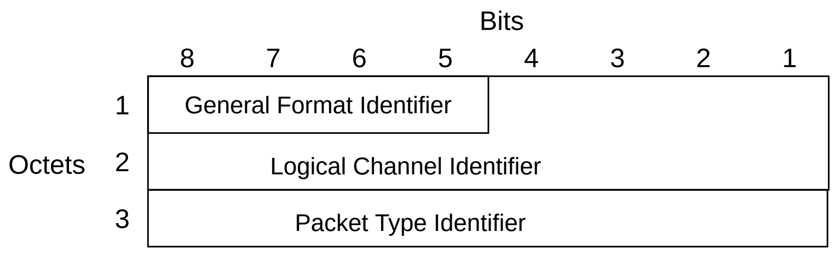

Figure 9 RESET CONFIRMATION Packet

The access to the fields in the RESET CONFIRMATION packet is described below:

General Format Identifier	Indicates the packet sequence count (modulo 8 or modulo 128). This field cannot be read or written by an X.25 application.
Logical Channel Identifier	Contains the logical channel identifier. It is controlled by the subsystem. It can be read with the `ioctl(X25_RD_LCI)` after the call has been established.
Packet Type Identifier	Indicates the kind of packet. X.25/300 automatically transmits this packet in response to an RESET packet, and transmission cannot be controlled by the application.

To read this field on an incoming packet:
In blocking mode, the `ioctl(X25_RESET_VC)`system call blocks until confirmation is received. In nonblocking mode, `recv(MSG_OOB)` returns a buffer the second byte of which (`buf[1]`) indicates that an RESET CONFIRMATION packet has arrived (`OOB_VC_RESET_CONF`).

To write this field on an outgoing packet:
The X.25/800 subsystem transmits an RESET CONFIRMATION packet when the corresponding RESET packet's reset user data field is read with a `recv(MSG_OOB)` system call.

B

Program Examples

Example Programs

Several pairs of example programs are shipped with the X.25 product. These programs are in the `/usr/netdemo/x25 directory`, and are as follows:

Program name	Purpose
client.c/server.c	Send/receive one message of data, no Out-of-Band signal handling. For Switched Virtual Circuits only.
client2.c/server2.c	Send/receive one message of data, then send/receive one Interrupt packet, with Out-of-Band signal handling. For Switched Virtual Circuits only.
clientpvc.c/serverpvc.c	Send/receive 1 message of data, then send/receive 1 Interrupt packet, then send a Reset packet with Out-of-Band signal handling. For Permanent Virtual Circuits only.
client2pvc.c/ server2pvc.c	Identical to clientpvc.c/serverpvc.c, except that there is a long loop on sending/receiving data messages. This difference allows the testing of the Out-of-Band Level 2 Down event. For Permanent Virtual Circuits only.
infoserv_client.c/ infoserv.c	Similar to the programs above, but the main difference is that the server "forks" the listener's process before receiving the incoming calls; this allows faster processing at connection time. The other main difference is that several parameters can be configured, such as number of messages, message size, number of processes to fork etc.

About this book

Reader comments

Comments on this book can be sent to the following address

The Manager
Network Usability Group
Enterprise Networking and Security Division
Hewlett-Packard France
5, Avenue Raymond Chanas - Eybens
38053 GRENOBLE Cedex 09
France

You can also send comments to the following e-mail address

doc_comments@grenoble.hp.com

Please quote the Customer order number, the HP Manufacturing part number and the edition number in all correspondence.

Printing history

The printing date changes when a new edition is printed. The book's part number changes when major changes are made.

Edition	Edition number	Print date	Customer order number	HP manufacturing part number	Comments
Edition 2	E1095	Oct. 1995	36960-90050	36960-90056	Also available on CD-ROM
Edition 3	E0596	May 1996	36960-90050	36960-90058	Also available on CD-ROM

Reader comments

Index

Index

Index

Index

HEWLETT PACKARD

Printed in - E05/96

Order Number
36960-90050

Manufacturing Part Number
36960-90058