X25API Programming Manual

Author: Nenad Corbic

1. Introduction

The new X25API driver has been completely redesigned to take advantage of Linux socket architecture. Due to unsecured nature of standard Linux sockets, Sangoma has developed a secure streaming socket which guarantees packet delivery during high traffic.

2. Installation

X25API and Wanpipe Socket drivers, are included in the latest WANPIPE package, (refer to ftp site below). Please refer to WANPIPE user manual, wanpipeForLinux.pdf, for wanpipe package installation, kernel compilation and startup. (/usr/local/wanrouter/doc directory). The most current WANPIPE(tm) package is located on Sangoma ftp site: ftp.sangom.com in /linux/current_wanpipe/ directory.

Details about structures, commands and return codes can be found in the manual entitled "X.25 SUPPORT SANGOMA CARDS"; February 2000. (/usr/local/wanrouter/doc/x25.pdf)

3. Configuring X25API

Once the wanpipe package is installed, proceed to create a /etc/wanpipe#.conf configuration file, which will be used to startup and configure wanpipe drivers/protocols.

Instead of manually configuring the wanpipe configuration file, one can use a GUI configuration utility, called 'wancfg'. It resides in /usr/local/wanrouter/wancfg directory. The wancfg application contains all the help files needed to successfully create a wanpipe configuration file. In case you need more information refer to WANPIPE user manual, wanpipeForLinux.pdf (located in /usr/local/wanrouter/doc directory).

Important Note:

When configuring a driver for API use, make sure that API option is selected, under the interface section.

```
[interfaces]
 wp1_svc = wanpipe1, ,API, x25 svc chan
```

4. Starting up X25API

Once the wanpipe configuration file is created, start up the x25api driver using `“wanrouter start”` command.

Check /var/log/messages file for configuration errors and link status. Before attempting to send data, make sure the “link connected” message is present in the message file. If the X25/HDLC protocol layer is not connected, no communication is possible.
For further debugging and line testing, use the xpipemon debugged and refer to the WANPIPE user manual, wanpipeForLinux.pdf in /usr/local/wanrouter/doc directory.

5. Programming X25API

X25API programming model is closely modeled to TCP/IP socket programming. Functions such as, sock(), bind(), listen(), accept() and connect() are used to communicate to the X25API driver.

The /usr/local/wanrouter/api/x25 directory contains the samples code for server and client implementations. We suggest that you edit these files and use them as a starting point in your x25api application development.

a. socket() function

```
int socket(int domain, int type, int protocol);
```

The socket function call creates a new sock and returns
the sock file descriptor to the user. This must be done before any other BSD IPC system call is executed. The returned file descriptor is used for the subsequent system calls used to establish an SVC or PVC connection.

```
Domain:   AF_WANPIPE
Type:     SOCK_RAW
Protocol: 0
```

```
Ex:   int sock = socket(AF_WANPIPE, SOCK_RAW, 0);
```

Please refer to the sample programs server.c and client.c in
/usr/local/wanrouter/api/x25 directory.

b. bind() function

```
int bind ( int sock fh, struct sockaddr *addr, int *addrlen);
```

After a socket has been created, sock must be bound to a network device. The bind() system call binds the newly created sock to the appropriate network device. Only after the sock has been bound, can communication between the sock and the driver take place.

During driver startup, a network device is created for each LCN; defined in WANPIPE configuration file.

Bind function uses a `wan_sockaddr_ll` address structure to bind a socket to the appropriate network device. Thus, the wan_sockaddr_ll structure must be filled in and passed by reference to the bind system call.

```
struct wan_sockaddr_ll
{
      unsigned short    sll_family;
      unsigned short    sll_protocol;
      int               sll_ifindex;
      unsigned short    sll_hatype;
      unsigned char     sll_pkttype;
      unsigned char     sll_halen;
      unsigned char     sll_addr[8];
      unsigned char     sll_device[14];
      unsigned char     sll_card[14];
};
```

i. Server Process: SVC

```
struct wan_sockaddr_ll addr;

addr.sll_family = AF_WANPIPE;
addr.sll_protocol = htons(X25_PROT);
strcpy(addr.sll_device, “svc_listen”);
strcpy(addr.sll_card, “wanpipe1");
```

`svc_listen:`	Represents a virtual network interface name, which is used to bind a sock to a listening queue for a particular wanpipe card.
`wanpipe#:`	Wanpipe card name. It is used to bind a sock to a correct WANPIPE card. This prevents problems in multiple card systems. # = 1,2,3 ... 16 : Card number

ii. Client Process: SVC

```
struct wan_sockaddr_ll addr;

addr.sll_family = AF_WANPIPE;
addr.sll_protocol = htons(X25_PROT);
```

```
strcpy(addr.sll_device, “svc_connect”);
strcpy(addr.sll_card, “wanpipe1");
```

`svc_listen:` Represents a virtual network interface name, which is used to bind a sock to a next available network device on a particular wanpipe card.

`wanpipe#:` Wanpipe card name. It is used to bind a sock to a correct WANPIPE card. This prevents problems in multiple card systems.
# = 1,2,3 ... 16 : Card number

Please refer to the sample files, server.c and client.c in /usr/local/wanrouter/api/x25 directory.

c. listen() system call

```
int listen ( int sockfh, int backlog);
```

The listen() system call prepares a socket to receive CALL INDICATION packets. All CALL_INDICATION packets are put into this queue for a particular wanpipe card. The server cannot receive a connection request until it has executed a listen() system call.

After a socket has been set into a listening mode, it cannot be used to transmit or receive data, it can only be used to accept incoming calls.

Refer to the /usr/local/wanrouter/api/x25/server.c example code.

d. accept() system call

```
int accept ( int sock fh, struct sockaddr *addr, int *addr_len);
```

Before accept() system call can be executed, the socket be in listening mode, otherwise an error will be generated.

The accept() system call returns a socket descriptor for a socket associated with an SVC connection. The accept() system call doesn’t establish the x25 connection, instead it’s up to the user to analyze call data and respond accordingly: accept or reject the call.

The accept() call blocks the socket until a CALL REQUEST packet arrives.

The addr and addrlen fields in accept() system call are optional, they can be set to NULL. Up on a successful CALL INDICATION, the addr structure will contain the network device the call came in on. This information is not very useful thus, it is recommended that the accept() call is used with addr and addrlen fileds set to

NULL.

Ex: int sock1 = accept(sock, NULL, NULL);

where sock variable is a file descriptor of the listeing socket.

To establish a connection, the user application must execute an ACCEPT command through an ioctl call

Ex: int ioctl(SIOC_WANPIPE_ACCEPT_CALL ,&accept_data);

Please refer to the IOCTL section of the manual.

e. connect() system call

```
int connect ( int sock fh, struct sockaddr *addr, int *addrlen);
```

Once a socket was bound to a "svc_connect" virtual network device (refer to bind() system call), client application may request the X.25 connection using connect() system call.

Before connect() system call, the user must supply, call data information through an IOCTL call.

```
int ioctl (sock,SIOC_WANPIPE_SET_CALL_DATA,&data);
```

The "data" structure is as follows: `"x25api_t data";`

```
typedef struct {
      unsigned char  qdm      PACKED;     /* Q/D/M bits */
      unsigned char  cause    PACKED;     /* cause field */
      unsigned char  diagn    PACKED;     /* diagnostics */
      unsigned char  pktType   PACKED;
      unsigned short length    PACKED;
      unsigned char  result   PACKED;
      unsigned short lcn      PACKED;
      char reserved[7]  PACKED;
}x25api_hdr_t;


typedef struct {
      x25api_hdr_t hdr  PACKED;
      char data[X25_MAX_DATA] PACKED;
}x25api_t;
```

Therefore, before the ioctl() call is executed, fill in the call data information in the above structure. Please refer to the sample client application in /usr/local/wanrouter/api/x25 directory.

```
x25api_t api_data;
memset(&api_data,0,sizeof(x25api_t));
sprintf(api_data.data, "-d1234 -s2345 -f3232 -uC21010");
api_data.hdr.length = strlen(api_data.data);
```

Once the call data has been properly set, we can request x25 link establishment using the connect() system call.

```
int connect(sock, NULL, NULL);
```

On a successful call establishment, the addr structure will contain the network device on which the connection took place. This information is not very useful; thus, it is recommended that addr and addrlen variables are set to NULL.

f. Socket ioctl() system calls

int ioctl (int sock_fh, int x25api_cmd, unsigned long data);

x25api ioctl commands are as follows:

`SIOC_WANPIPE_SET_CALL_DATA :`

Set the call information data into a socket mailbox. This mailbox is used to send the command down to the driver.
This command must be executed, before a client requests connection establishment using connect() system call.

`SIOC_WANPIPE_GET_CALL_DATA:`

Get the incoming call data for the socket mailbox. Once accept() system call returns a new socket descriptor, the above commands must be executed to retrieve the incoming call information. Then it is up the the user to accept or clear the pending call.

`SIOC_WANPIPE_ACCEPT_CALL:`

Accept the incoming call. Once the incoming call data has been approved, the above command will establish the call. Note, that data filed is optional for this command, thus, data filed can be set to zero.

`SIOC_WANPIPE_CLEAR_CALL:`

The user can clear the call at any time using the above command. The data filed is optional; Thus, it can be set to zero in most cases.

`SIOC_WANPIPE_RESET:`

The user can send a reset any time once the connection is established by executing the above command. The data filed is optional; Thus, it can be set to zero in most cases.

`SIOC_WANPIPE_SET_NONBLOCK:`

This option will set the socket into a non blocking mode. The socket can be set for non-blocking only in conjunction with connect() system call. In which case connect() call will not block until the connection is confirmed. Using this option, a single process can place multiple calls without waiting for connection establishment.

`SIOC_WANPIPE_CHECK_TX:`

This option will return the number of bytes pending for transmission. One can use this ioctl call to check whether there is data waiting in the sock transmit queue, before the socket is closed. If data acknowledgement is not used, this ioctl must be implemented so that is not lost on close() system call. Note: This command is usually used in PVC systems.

`SIOC_WANPIPE_SOCK_STATE:`

This commad will return 0 if socket is in CONNECTED state, otherwise the return code will be 1. This command can be used after an OOB message to test whether the link is still up.

Data structure for ioctl() system calls is, `x25api_t`:

```
typedef struct {
      unsigned char  qdm      PACKED; /* Q/D/M bits */
      unsigned char  cause    PACKED; /* cause field */
      unsigned char  diagn    PACKED; /* diagnostics */
      unsigned char  pktType  PACKED; /* OOB packet type */
      unsigned short length   PACKED; /* data length */
      unsigned char  result   PACKED; /* command result */
      unsigned short lcn      PACKED; /* bound lcn number */
      char reserved[7]  PACKED;
}x25api_hdr_t;


typedef struct {
      x25api_hdr_t hdr  PACKED;
      char data[X25_MAX_DATA] PACKED;
}x25api_t;
```

i. QDM Bits

QDM Bits are bit mapped into one byte of data.
This filed is optional and should be set to zero in most cases.

Q: Is a bit which marks a data packet as a special kind of packet.
Eg: Async PAD use the Q bit packets to negotiate PAD parameters after call setup.

D: Is used for confirmation of delivery of important packets.

M: Used to fragment the messages which are larger than the maximum packet size.

ii. Cause and Diagnostic Fields

Cause and diagnostic fields are used to relay information to the remote user in regards to a asynchronous event. These fields are optional and should be set to zero in most cases.

Eg: When clear call is issued due to invalid call data, cause and diagnostic fields should be set appropriately. Thus, when the remote side receives an asynchronous message it will know the reason for it.

iii. Packet Type Field

Packet type field should be used as read_only information. Upon receiving an OOB asynchronous message, the packet type field will indicate which asynchronous event occurred. Thus, based on the event, the state of the link can be established.

iv. Length Field

Length field indicates the size of the data buffer associated with the command or the OOB event. Any time data buffer is used, the length field must be set the data buffer length.

v. Result Field

Result field should be used as read_only information. Result of every command executed will be stored in this field. This field should be used for statistic purposes only.

vi. LCN Field

This field should be used as read_only information. Upon successful link establishment the LCN field will indicate the link number currently used.

g. select() system call

```
int select(int sock_fh, fd_set *read, fd_set *write, fd_set *oob,
        struct timeval *timeout)
```

Select() system call blocks and polls single or multiple sockets for receiving data, writing data or receiving OOB data. Select is the key factor in Sangoma secure socket architecture. It must be used to guarantee that no data will be lost. The select method is tied to the socket flow control code, which will block if transmit buffers become full.

Please refer to the x25api example code.

h. send() system call

int send (int sockfh, void *buf, int len, unsigned int flags)

Send() system call, can only be used when the connection is established, otherwise an error will be returned. If send is successful the return code will indicate the number of bytes transmitted. On error the return code is set to -1.

Every packet transmitted must contain a 16 byte Header (x25api_hdr_t, refer to the above data structure in section 5.f). The header data will not be passed out the port. It should be used to convey special information to the driver. For example, setting QDM bits. Details can be found in "X.25 SUPPORT SANGOMA CARDS".

`flags` : set to zero.

i. recv() system call

int recv (int sockfh, void *buf, int len, unsigned int flags);

Recv() system call, receives packets from the sock, if receive is successful the recv() will return number of bytes read. On error the return code is set to -1.

Every receive packet comes with 16 byte (`x25api_hdr_t` refer to section 5.f), application should remove the header before using the data. Furthermore, the header will contain special bits which were transmitted by the remote switch, for example QDM bits. Details can be found in "X.25 SUPPORT SANGOMA CARDS".

`flags` : set to zero, for regular data.
set to MSG_OOB, for reading synchronous message

Note: The select() method will indicate that OOB message is waiting, the recv() system call should be used with flags set to MSG_OOB. Once the OOB message is received, use the `packet type` to determine the asynchronous event.

6. Sending and Receiving Data

C Once the connection is established, the socket is ready to receive and transmit data.

C Regardless of the application, the driver will start sending data up the socket as soon as it arrives.

C It is up the application to wait for the data and receive packets from the socket. Once the packet is received by the user, the packet is flushed out of the socket; Thus, only one process should read a single socket a time.

C If the driver fills up the socket, receive interrupt will be turned off, which will block all the other channels from receiving data. If this occurs, after X number of seconds, the HDLC protocol will generate a protocol violation followed by the restart request, which will bring down all active channels. Therefore, make sure that the application is waiting to receive data all the time. Note, socket has a 64KB buffer and driver has 10 packet local buffer to prevent this from ever happening.

C In order to provide secure packet delivery to and from the application, the select() system call must be implemented. It will indicate when a data packet or OOB packet is pending, or when the socket is ready to transmit.

C The select must be setup to poll OOB events, otherwise the send() system call will fail when trying to send on a disconnected socket: in case of restart.

C Every transmitted packet must contain a 16 byte header which is to be used to convey important information to the driver : ex: QDM bits. Please refer to section 5.f, structure x25api_hdr_t. Details can be found in "X.25 SUPPORT SANGOMA CARDS".

C Every received packet has a 16 byte header which contains important information such as QDM bits. Please refer to section 5.f, structure x25api_hdr_t.. Details can be found in "X.25 SUPPORT SANGOMA CARDS".

7. X25API Programming Strategies

Most of the x25api application fall into ether server or client programming style.

C The **server** application usually waits for the incoming calls. On incoming call it ether spawns off children to handle the pending connection, or handles the connection itself after which it returns to listen state. The later option is a more robust solution, especially in Linux multi-tasking environment. Under heavy-load conditions no incoming calls will timeout, since most of the work is distributed to the children.

C On the other hand, **client** application usually requests the x25api connection. Upon successful connection, the client handles the call after which the session is terminated, or another connection is requested.

C The two programming model implementations can be found in /usr/local/wanrouter/api/x25 directory: `server.c and client.c`

C We suggest that these programs be used as the starting point of the x25api application development. The user is free to modify change or use the code in any way which will provide the fastest time to market and ease of use.

C The program xpipemon allows you to monitor the X.25 statistics, debug line problems and it will help you debug your programs. If you use the trace feature, you can see both your own data and the incoming data.