X25API/MP−API Programming Manual

Date: April 9, 2003

Author: Nenad Corbic

Introduction

Wanpipe X25 API solution consists of two separate products:
Original X25API and New Multi−Protocol Stack.

Original X25API

> The X25API is the standard wanpipe product that implements x25 protocol in firmware. The wanpipe API driver provides socket API support to the on−board x25 protocol allowing users to build custom applications.

Multi−Protocol Stack−X25API

> The MP−API is a new multi−protocol kernel stack developed on top of wanpipe HDLC driver. The multi−protocol kernel stack includes full Frame Relay, X25/LAPB and DSP protocols implemented as kernel modules. The MP−API architecture was designed to be very scalable and flexible. Using the MP Stack architecture multiple protocols can be stacked together or used independently.
>
> The X25API Stack would look like the following:
>
> > API−>X25−>LAPB−>Frame Relay−>HDLC (adapter)
> > API−>X25−>LAPB−>HDLC (adapter)
>
> When using MP−API architecture the sangoma adapter is running RAW HDLC protocol, thus acting as dumb card. Furthermore, all adapter ports both PRI and SEC can now be used to carry independent protocols. (This was a big limitation in the original design.)

From the users perspective, the API interface is identical to both X25 API models except for few subtle differences. The choice between to API models is made on creation of the API socket.

To speed up API development detailed sample API code has been supplied with the WANPIPE drivers. We suggest that these programs be used as the starting point of the x25api/MP−API 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.

Sample Code Location:

X25API : /etc/wanpipe/api/x25
MP−API: /etc/wanpipe/api/mpapi/x25

Wanpipe API Interface

The Wanpipe API layer, 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.

Application Programming Interface (API), is used to send and receive custom raw, non−IP, packets to and from the card. Since data communicated is not in IP format, the network interface is configured without the IP address. This effectively removes the kernel routing table entry and unhooks the IP routing stack from the Wanpipe driver. Non−IP communication is achieved using the RAW socket calls to the driver. As the name implies, packets are transferred without any modification.

Most API applications are very sensitive to packet loss, which presented a problem in using Linux Raw sockets, which silently dropped packets during high traffic. Dropping packets is ok for TCP/IP but not for X25: protocol based on zero drop policy.

Consequently, the secure socket solution was created; custom socket that guaranteed delivery in both upstream and downstream. The Wanpipe socket is based on the Linux raw socket, developed by Alan Cox and others. Following rules were needed to guarantee delivery:

- Check for adequate buffer space before sending data.
- If send buffers are full the user is blocked from sending until buffers are freed up.
- If the receive sock buffers are full, driver will enable protocol flow control to slow down traffic until receive buffers are freed up.

Wanpipe Installation

Both X25API, Multi−Protocol API and Wanpipe Socket API drivers, are included in the latest WANPIPE package, (refer to ftp site below).

Please refer to WANPIPE user manual: **WanpipeInstallation.pdf,**
It contains Wanpipe package installation, compilation and startup.

Location:
Web: `ftp.sangoma.com/linux/current_wanpipe/doc`.
Local: `/usr/share/doc/wanpipe/`

To enable the Multi−Protocol API stack one must select the **MP−API** option from the ./Setup install, driver compilation protocol options.
Please refer to the WanpipeInstallation.pdf or README−1.install

Wanpipe X25API/MP−API Configuration

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

Configuration utility: `/usr/sbin/wancfg`

Instead of manually configuring the wanpipe configuration file, use a GUI configuration utility, '/usr/sbin/wancfg'. 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 configuration manual:
WanpipeConfiguration.pdf or REAMDE−2.config

Location:
Web: `ftp.sangoma.com /linux/current_wanpipe/doc`
Local: `/usr/share/doc/wanpipe/`

X25API:

Protocol: X25 Protocol

Interface Operaton: API

MPAPI:

Protocol: Cisco HDLC or MP Frame Relay

The carrier protocol for Mulit−Protocol API is always HDLC. Thus, the board level code is always Raw HDLC.

When connecting to a real X25 line select the Cisco HDLC protocol which will implement the RAW HDLC firmware layer.

When connecting to a Frame Relay line select MP Frame Relay Protocol (note that MP Frame Relay protocol is implemented in the kernel and it uses the HDLC firmware)

Interface Operation Mode: ANNEXG

The ANNEXG operation mode indicates that the MP−API stack will be running over this carrier protocol. Once ANNEXG option is selected further X25/LAPB protocol options will be enabled.

Proceed to configure X25/LAPB link layers.

Wanpipe X25API Operation

Once the wanpipe configuration file is created, start up the x25api/MP−API driver using "wanrouter start" command.

- `/usr/sbin/wanrouter start wanpipe1`
- 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/MP−API/HDLC protocol layer is not connected, no communication is possible.

For further debugging and line testing, use the wanpipemon debugged and refer to the WANPIPE operational manual: **WanpipeOperation.pdf or README−3.operation.**

Location:
Web: `ftp.sangoma.com /linux/current_wanpipe/doc`
Local: `/usr/share/doc/Wanpipe/`

Wanpipe API Architecture

The X25API/MP−API interface uses sockets to communicate to each x25 logical channel (svc or pvc). Standard TCP/IP calls must be used to setup and establish connection to the remote server/client.

Socket Handling

- Once the connection is established, each socket acts as a data pump transmitting data to and from the driver. Since the socket buffering is limited (64KB), the user application is responsible to pull data out of the sockets as fast as possible.

- If the single channel, on the x25 link, fails to send data up to the socket due to socket being full, the driver will implement x25 flow control. Which will block the remote side from transmitting until the user application has pulled packets out of the full socket.

- Thus, on a multi channel x25 link, if a single channel is ignored, and the socket for that channel becomes full, the x25 flow control will block all incoming that for that particular link. In other words, **a single channel can block ALL active channels on an x25 link**.

    - To avoid this from happening, a select() system call must be setup to wait on all active sockets. So all events from all active channels, regardless of the user application states, are handled.

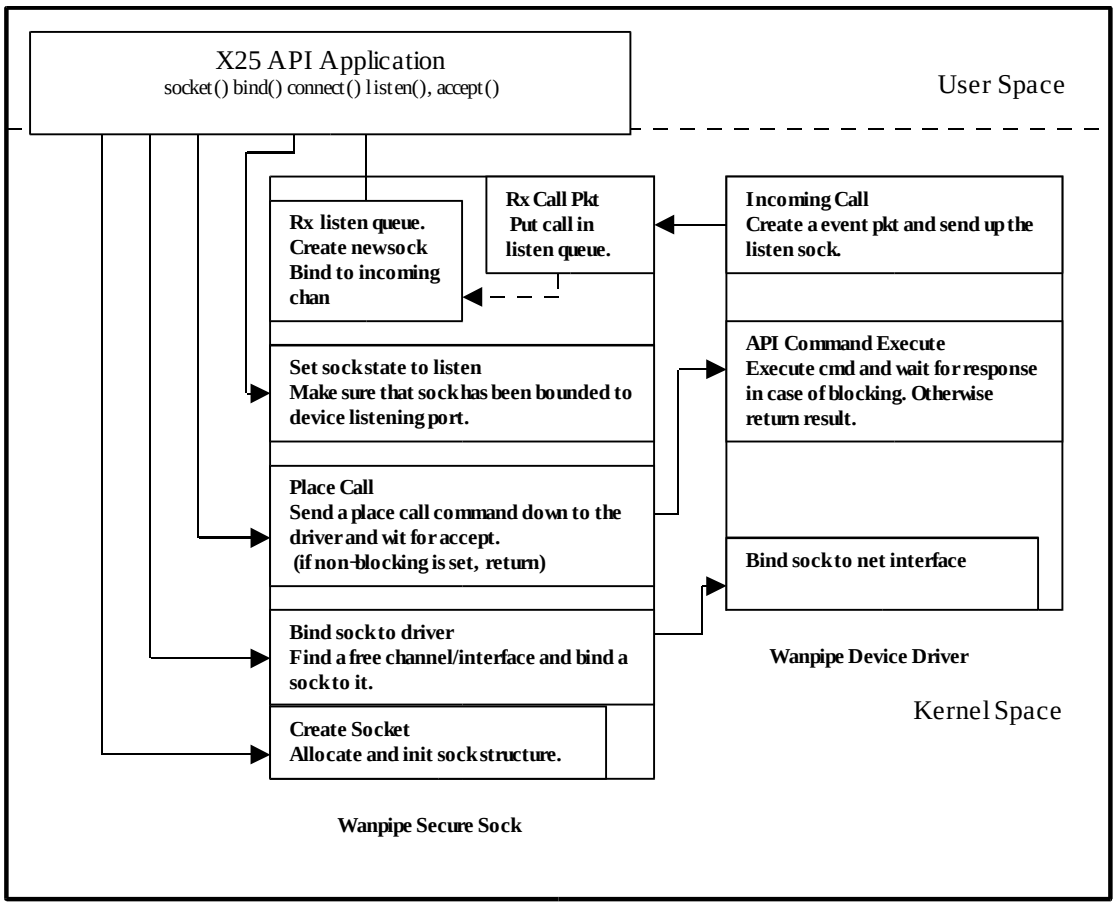
Event handling

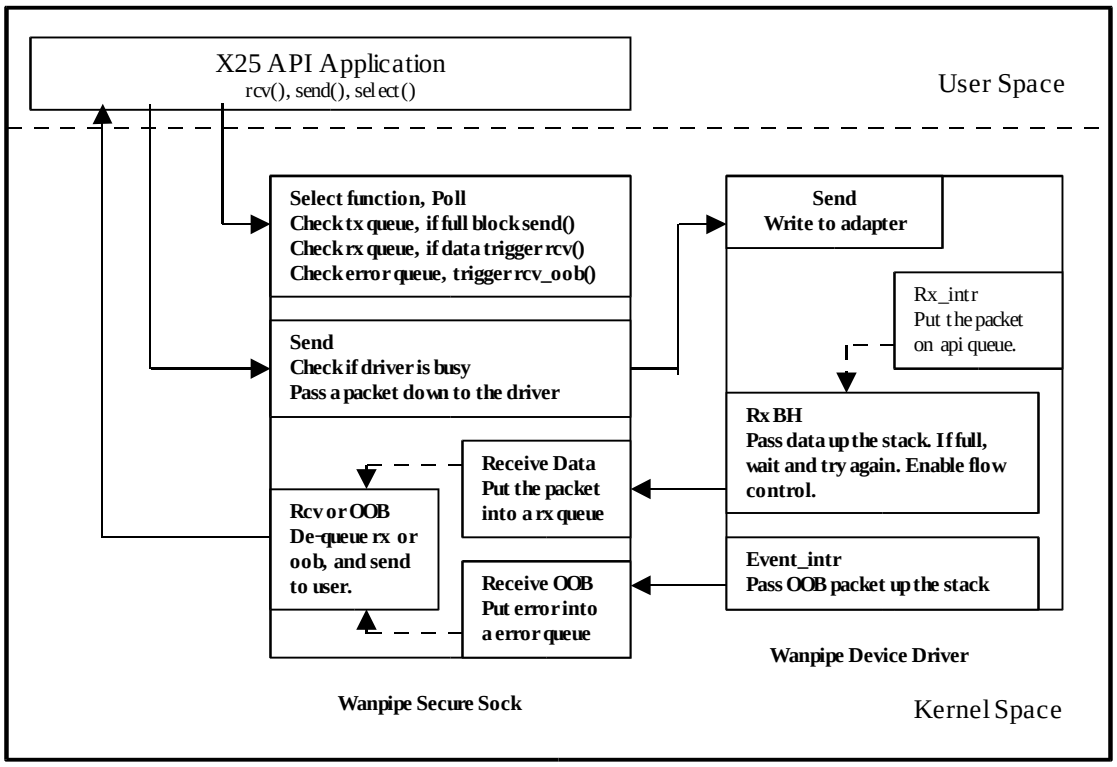
- Once the communication is established, the application uses the socket to transmit and receive data to and from the remote connection.

- The channel will stay connected until the out of band (oob) event is received, or the user initiates disconnect (clear call or socket close).

- The OOB event must be decoded, and handled by the application (i,e, application should handle its states accordingly). Not all OOB events will bring the connection down (Reset and Interrupt packets). The user should check the state of the socket before closing it.

- It is very important to pull off any remaining RX data out of the socket, after the OOB event has occurred, before closing the socket. Otherwise, the close() system call will delete all socket queues as well as the socket itself.


X25 API Application
socket() bind() connect() listen(), accept()
User Space
Rx listen queue.
Create newsock
Bind to incoming
chan
Rx Call Pkt
Put call in
listen queue.
Incoming Call
Create a event pkt and send up the
listen sock.
Set sockstate to listen
Make sure that sock has been bounded to
device listening port.
API Command Execute
Execute cmd and wait for response
in case of blocking. Otherwise
return result.
Place Call
Send a place call command down to the
driver and wit for accept.
(if non-blocking is set, return)
Bind sock to net interface
Bind sock to driver
Find a free channel/interface and bind a
sock to it.
Wanpipe Device Driver
Kernel Space
Create Socket
Allocate and init sock structure.
Wanpipe Secure Sock


X25 API Application
rcv(), send(), select()
User Space
Select function, Poll
Check tx queue, if full block send()
Check rx queue, if data trigger rcv()
Check error queue, trigger rcv_oob()
Send
Check if driver is busy
Pass a packet down to the driver
Rcv or OOB
De-queue rx or oob, and send to user.
Receive Data
Put the packet into a rx queue
Receive OOB
Put error into a error queue
Wanpipe Secure Sock
Send
Write to adapter
Rx_intr
Put the packet on api queue.
Rx BH
Pass data up the stack. If full, wait and try again. Enable flow control.
Event_intr
Pass OOB packet up the stack
Wanpipe Device Driver
Kernel Space

X25API/MP−API Client Programming Model

The x25api client is responsible for initiating outgoing calls to the server. Once the call is accepted, the client proceeds to send and receive data packets over the x25 network. The communication will continue until the client initiates a disconnect or an out−of−band (oob) event occurs (eg. Clear call). The example code `svc_client.c` should be used to kick start the x25api client development.

- Create a socket
  `select() system call`

- Bind socket to a free svc network interface
  `bind () system call`
  `device name "svc_connect"`

- Write call data information into the socket.
  (−d<called address> −s<calling addr> −f<facils> −u<user data>)
  `ioctl(SIOC_WANPIPE_SET_CALL_DATA);`

- Initiate outgoing call to the server
  `connect() system call`
  Block method: waits for the remote response (not available for MP−API)
  Non−Block method: returns as soon as call is placed.

- Initialize the select() statement and block for receive, transmit and oob.
  `select() system call`

- Proceed to send and receive data

- Once an oob connection is received, or the connection is terminated, the socket MUST be closed.

X25API/MP−API Server Programming Model

The x25api server is responsible for accepting and handling incoming connections. Once the server accepts a call, a child process should be spawned to handle the data tx/rx and oob events for that particular channel. The server process must remain in listening mode waiting for incoming call requests. The child will continue to exist until the connection is up. Once the OOB message is received, the child must handle the OOB message, close the socket and die.

The child can be created using a fork(), clone() or pthread. The fork() system call is adequate for small number of channels (1−50), however its very expensive and is not very scalable for application running 100+ simultaneous connections. The pthread architecture is the fastest and most scalable method of implementing a server. The x25api sample codes contain three implementations of the x25api server, located in /etc/wanpipe/api/x25 or /etc/wanpipe/api/mpapi/x25 directory.

server_v1.c : fork() based server
servier_v2.c : single process server, that doesn't spawn children
pthread/server.c : pthead based server that uses threads to spawn children
(*not implemented for MP−API*).

- Create a socket
  `select() system call`
- Bind a socket to a x25 link listening port
  ```
  bind () system call
  device name "svc_listen"
  ```
- Set the socket in listening mode
  `listen() system call`

- Setup select system call to wait for any incoming call requests on the listen socket.
  `select(listen_sk_id,&readwait,NULL,NULL,NULL)`

- When an incoming call arrives use the accept() system call to create a new socket, which will be used to tx/rx data to and from the remote connection.
  `accept() system call`
- The server must send an accept to the remote clinet in order to establish a connection. This can be done by the server or by the child which will be handling that connection. It is suggested that the child sends the accept call, so that the server can go back to listening faster.
- Once the call is established the server should tx/rx data until an OOB event occurs or disconnect transmitted (clear call).

X25API System Calls

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

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.

X25API:

Domain: AF_WANPIPE
Type: SOCK_RAW
Protocol: 0

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

MP−API:

Domain: AF_WANPIPE
Type: SOCK_RAW
Protocol: AF_MP_WANPIPE

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

Please refer to the sample programs server_v1.c and svc_client.c in /etc/wanpipe/api/x25 or /etc/wanpipe/api/mpapi/x25 directory.

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];
};
```

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 conflicts in multiple card systems. # = 1,2,3 ... 16 : Card number

MPAPI:

```
addr.sll_family = AF_MP_WANPIPE;
addr.sll_protocol = htons(ETH_P_X25);
```

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_connect:` 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

MPAPI:

```
addr.sll_family = AF_MP_WANPIPE;
addr.sll_protocol = htons(ETH_P_X25);
```

Please refer to the sample files, server_v1.c and svc_client.c in /etc/wanpipe/api/x25 or /etc/wanpipe/api/mpapi/x25 directory.

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 server_v1.c example code.

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.

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 /etc/wanpipe/api/x25 or /etc/wanpipe/api/mpapi/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.

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. The data filed is optional; It can be set to zero in most cases.

SIOC_WANPIPE_INTERRUPT:
The user can send an interrupt packet any time once the connection is established.

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.
(Only to be used with X25API. MPAPI only support non−blocking system calls except select())

SIOC_WANPIPE_SOCK_STATE:
This commad will return 0 if socket is in CONNECTED state, return 1 if socket is in DISCONNECTED state, otherwise the return code will be 2. This command can be used after an OOB message to test whether the link is still up.

SIOC_WANPIPE_CHECK_TX:
This commad will return 0 if all tx packets have been send out the port. This comand should be used before executing a CLEAR CALL or closing the socket, to make sure that all data transmitted has left the card. Otherwise data loss could occur.

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.

Select 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.

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.
- If send fails it means that the driver is busy: try again. Note: it's not an error condition. If error return code is not EBUSY then there is a user or driver error.
- 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.

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.

- flags : set to 0, 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.

Sending and Receiving Data

- Once the connection is established, the socket is ready to receive and transmit data.
- Regardless of the application state, the driver will start sending data up the socket as soon as data arrives.
- It is up the application to wait for the data and receive packets from the socket.
- Once a 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.
- 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.
- 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.
- 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.
- 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 appendix, structure x25api_hdr_t.
- 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..

X25API Programming Strategies

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

- The **server** application usually waits for the incoming calls. On incoming call it ether spawns off children to handle the pending connection (`server_v1.c`), or handles the connection itself after which it returns to listen state (`server_v2.c`). 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.

- The **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. (`svc_clinet.c`)

The two programming model implementations can be found in /etc/wanpipe/api/x25 or /etc/wanpipe/api/mpapi/x25 directory: server_v1.c and svc_client.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.

The program **wanpipemon** 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.

APPENDIX

X25API TX/RX Header and IOCTL Command Data structure for the X25 API system calls: **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; /* 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 Byte

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

Q: **bit 2**
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: **bit 1**
Is used for confirmation of delivery of important packets.

M: **bit 0**
Used to fragment the messages which are larger than the maximum packet size.

IMPORTANT

TX: Use this byte to send fragmented data via Mbit option
RX: Use this byte to determine if a fragment was received or a whole packet.
IOCLT: When **clearing call**: user must ensure that transmitted data was successfully received by the remote end. By setting **QDM = 0x80**, the clear call will fail, if the x25 protocol has not transmitted all tx packets. Thus, insuring that all packets were sent successfully.
(Only supported by X25API not MP−API)

MP−API uses the SIOC_WANPIPE_CHECK_TX ioctl() instead.

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.