PPP/UDP Virtual Peripheral Technical Note

20 February 1999

Christopher Waters

Celsius Research Ltd

Introduction

This technical note describes how to implement the lower levels of a TCP/IP networking stack on a Scenix SX 8-bit micro-controller. TCP/IP is usually implemented on 32-bit micro-processors with memory measured in megabytes and is almost unheard of for 8-bit micro-controllers. Carefully structuring of the code to avoid packet buffering makes Internetworking possible in smaller and cheaper devices than ever before.

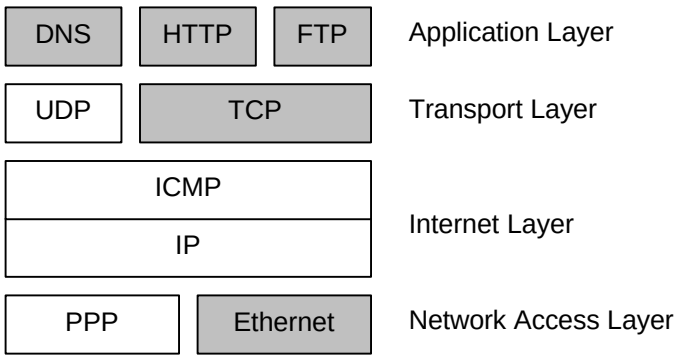
The phenomenal growth in Internet usage has seen a corresponding increase in the number of hardware devices capable of communicating using TCP/IP. The ubiquity of TCP/IP in turn drives the desire for network stacks in smaller and cheaper devices. The applications of a TCP/IP capable micro-controller are vast, in effect any sensory or control device could be communicated with from any desktop in the world.

In this technical note the source code for a subset of a TCP/IP stack is described. This stack includes PPP, IP, ICMP and UDP. Together these protocols are enough to enable an SX to connect to TCP/IP router and by extension, to the rest of the Internet.

The technical note is divided into three sections. The first gives an overview of TCP/IP and the particular protocols that the software implements. The second section describes how to use the software in an application. In the third section the arrangement of the source code and how it works are described along with schematics for a demonstration circuit board.

The TCP/IP Stack

The collection of protocols for transport of data over the Internet is commonly known as TCP/IP. In fact it is the Internet Protocol (IP) which is the fundamental building block. Transmission Control Protocol (TCP) is an optional higher level protocol, which just happens to be the most commonly used. The next diagram shows the commonly used protocols in the Internet stack. This technical note is concerned with the protocols without the gray fill.



Typical Internet Protocol Stack

At the very top layer are application protocols that we are familiar with using as part of web browsers, chat programs and telnet clients. The next layer down, the transport layer, provides two methods of data delivery across the Internet. Reliable, connection-based delivery is provided by TCP. Unreliable, connection-less delivery is provided by the User Datagram Protocol (UDP). The Internet layer provides addressing, quality of service and other routing options. The Internet Control Message Protocol (ICMP) which is tightly integrated with IP is a service for sending messages in response to error conditions. Once facility provided by ICMP is echo which is used by the Ping program.

At the network access layer the most common protocol is ethernet which is used for local area networks. The Point-to-Point Protocol (PPP) is used to encapsulate IP (and other protocols) over serial links. It is the most commonly used protocol for dial-up links, such as when you call your ISP with a modem.

The next sections provide more detail on the protocols used in this technical note, starting with the network layer and moving up the stack.

The Point-to-Point Protocol (PPP)

Overview

PPP provides a mechanism for encapsulating multiple protocols over point-to-point links. Usually PPP is used over serial links such as RS-232 or telephone lines (with the use of a modem). The bulk of PPP is defined in two Requests for Comments (RFC) documents, RFC1661 and RFC1662, published by the Internet Engineering Taskforce (IETF). RFC1661 describes the option negotiation mechanism while RFC1662 defines a method for using PPP with HDLC framing. RFC1662 also describes a method for data transparency and a frame check sequence (FCS) for detecting transmission errors.

PPP is a very general protocol and can be used for almost any protocol, although it is almost always used for encapsulating TCP/IP over dial-up links.

PPP works between two end-points called `peers'. There is no distinction between the end-points, such as client and server. As far as PPP is concerned both are equivalent and it is not important which end-point initiates the connection.

A typical PPP session proceeds as follows:

1. The connection is initiated by one end-point requesting configuration.

2. Both end-points simultaneously negotiate the link parameters using the Link Control Protocol (LCP).
3. A network connection is opened by the initiating end-point using a Network Control Protocol (NCP).
4. Data packets are transferred between the end-points.
5. The connection is closed.

Packet Format

The format of a PPP frame is shown below:

PPP frame format

The frame format may be changed if header compression is negotiated during link configuration. The PPP-VP does not allow header compression to be negotiated thus all frames have the format shown above.

Flag, Address, Control

Every frame begins and ends with a flag sequence ($7E). The address and control fields are described by ISO 4335-1979 HDLC. For PPP they are the constants $FF and $03.

Protocol

The protocol field is one or two bytes (in fact all the protocol numbers used for PPP are two bytes). This field indicates the protocol contained in the frame and thus how it should be interpreted. The protocol numbers relevant for this document are:

$0021	Internet Protocol
$8021	Internet Protocol Control Protocol (IPCP)
$C021	Link Control Protocol (LCP)
$C023	Password Authentication Protocol (PAP)
$C223	Challenge Handshake Authentication Protocol (CHAP)

Information

The content of the information field depends on the link state. The information field contains the negotiation options or IP packets. The maximum length of the field can be negotiated, but defaults to 1500 bytes.

Frame Check Sequence (FCS)

The FCS field holds a 16-bit CCITT-CRC to check for errors in transmission of the frame. The FCS is computed over the entire frame between the flag sequences and without the application of transparency (i.e. over the raw frame). An algorithm for computing the FCS using a lookup table is given in RFC1662. To conserve ROM space the PPP-VP uses a novel byte-at-a-time algorithm.

Transparency

Before a frame is transmitted extra escape characters are added to ensure that any data with the same value as the flag sequence or other control characters used by the link won't cause confusion. The transparency algorithm works on every character in the frame, including the FCS but, excluding the start and stop flag sequences. The control escape sequence is defined as $7D. Any instance of the control escape or the flag sequence in the frame are prefixed with the control escape character before being transmitted. Also, any bytes with a value less than $20 are xored with $20 and prefixed with a control escape. This is to ensure that control characters in the data can be distinguished from control characters used for tasks such as hand-shaking (XON/XOFF).

As an example, the frame

```
7E FF 03 C0 21 02 01 00 00 45 0A 7E
```

becomes

```
7E FF 7D 23 CO 21 7D 22 7D 21 7D 20 7D 20 45 7D 2A 7E
```

with transparency applied.

The control characters to which transparency is applied can be negotiated with LCP. The PPP-VP uses only the default transparency described above and will not allow any other configuration to be negotiated.

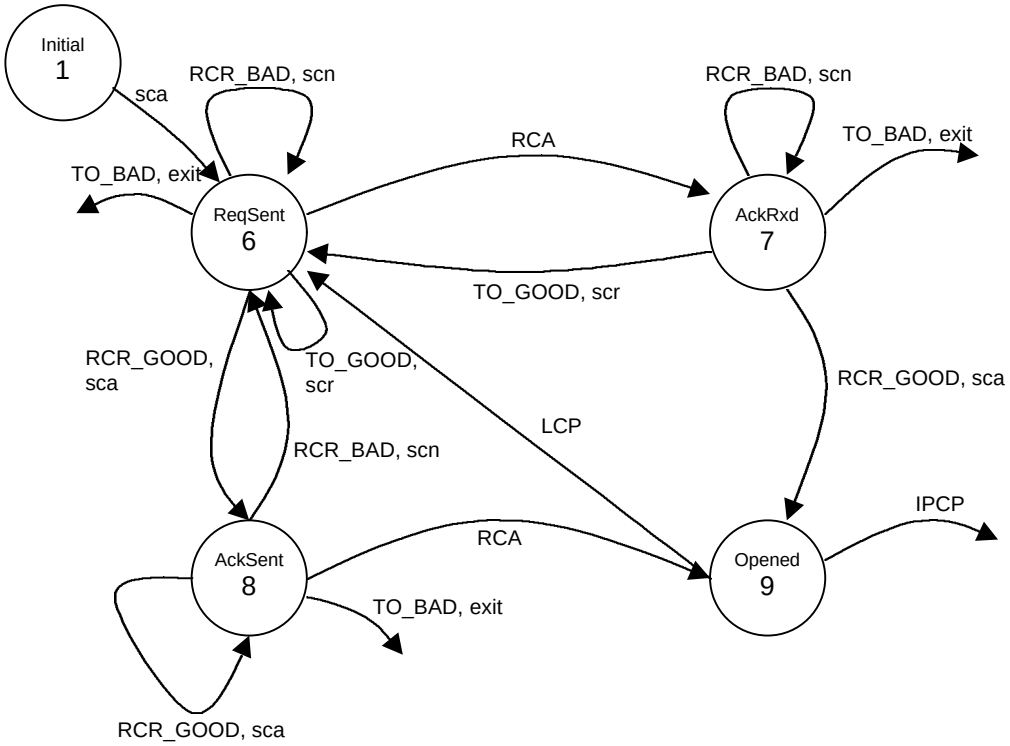
Option Negotiation

Several parameters of the link can be negotiated during initiation of a connection. These parameters include such things as the maximum frame size, the control characters to be escaped, authentication parameters, link quality, transparency characters and header compression. To keep code size to a minimum the PPP-VP insists that the peer accept default settings for all parameters. In this case negotiation is simplified, but still necessary for confirming that both end-points can support the default settings. Since PPP implementations must support the default configuration in order to be compliant this will not prevent the PPP-VP from communicating with any other PPP implementation. The negotiation protocol is called Link Control Protocol (LCP).

Negotiation State Machine

The diagram below shows the option negotiation state machine used in the PPP-VP. This is a substantial simplification of the state machine in RFC1661 but will still inter-operate with any other PPP implementation. There are four states in the state machine. The initial state is state 1. (The state numbering is not sequential so that it matches with the state transition table in the RFC.) Each transition has associated with it a condition and an action. The condition (shown in upper case) must be true for the transition to be followed. The action (in lower case) is executed when the transition is taken.

Conditions		**Actions**	
RCR_GOOD	Received an acceptable configure request	scr	Send a configure request
RCR_BAD	Received an unacceptable configure request	sca	Send a configure acknowledge
TO_GOOD	Timer expired but counter > 0	scn	Send a configure reject
TO_BAD	Timer expired by counter = 0	exit	Exit the state machine
RCA	Received a configure acknowledge		
LCP	Doing LCP negotiation		
IPCP	Doing IPCP negotiation		



PPP option negotiation state machine.

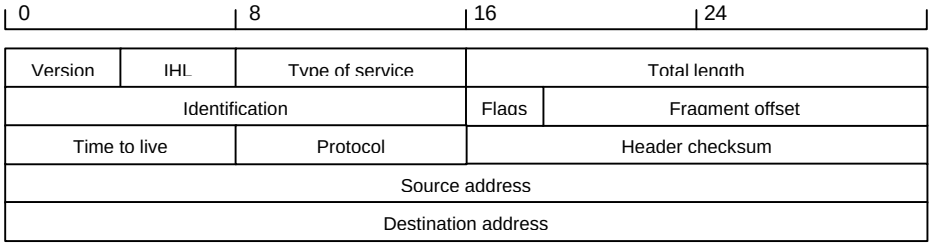
Once the options have been negotiated for the link the end-points must then negotiate a compatible network protocol (in this case the desired protocol is IP). The network protocol negotiation protocol is called Internet Protocol Control Protocol (IPCP). IPCP uses the same negotiation mechanism as LCP so to save code space they are implemented in the same state machine. Once LCP negotiation is finished the machine switches to IPCP negotiation and goes back to the ReqSent state. Only when the Opened state is reached during IPCP negotiation is the link ready for IP packets.

Internet Protocol (IP)

For the purposes of this technical note IP is used merely for adding addressing information to the packets being transferred. Other IP options, such as fragmentation and quality of service are ignored.

Packet Format

Each IP packet consists of a header, followed by zero or more data bytes. The header looks like the following:



Internet protocol header

The important fields for the PPP-VP are the total length, protocol, header checksum, source address and destination address. As its name suggests the length field contains the total length of the IP packet, including the header. The length of the data can be computed using the header length (IHL) which is the

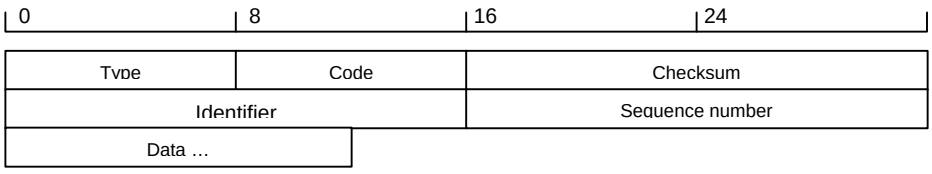
number of 32 bit words in the header. The header can be longer than five words if there are any options attached. The PPP-VP will not accept IP packets with options. The protocol field indicates the type of data contained in the rest of the packet. The protocols of interest to us are ICMP (1) and UDP (17). The IP checksum is computed over the header only. It is the ones complement of the ones complement sum of the 16 bit words in the header. The source and destination addresses are standard 32 bit Internet addresses which are usually written in the A.B.C.D notation. All multi-byte words in the IP header (and in fact all TCP/IP protocols) are in network-byte-order which is big-endian (most significant byte first). For more information about IP see Comer[1].

Internet Control Message Protocol (ICMP)

ICMP is not considered as a separate layer in the TCP/IP stack, but rather an extension to IP. For the PPP-VP ICMP will be used to provide a response to the standard ping tool. Strictly speaking every IP implementation must implement all ICMP messages. However, correct operation is still possible without them so the PPP-VP only implements the echo-request and echo-reply packets.

For every echo-request that is received an echo-reply will be generated.

Packet Format



ICMP echo-request and echo-reply packet format.

The type field indicates whether the packet is a request (8) or a reply (0). The checksum is computed over the ICMP packet only. The identifier and sequence number are used by the sender to match up requests and replies. An echo request can contain data which is simply copied to the reply.

Here is an example echo reply packet:

```
45 00 00 54 00 1A 00 00
0F 01 38 74 C0 A8 01 01
82 D8 2E 9A 00 00 3D DA
BC 52 1A 00 3E DB BF 36
F8 C5 03 00 08 09 0A 0B
0C 0D 0E 0F 10 11 12 13
14 15 16 17 18 19 1A 1B
1C 1D 1E 1F 20 21 22 23
24 25 26 27 28 29 2A 2B
2C 2D 2E 2F 30 31 32 33
34 35 36 37
```

This packet is from 192.168.1.1 (C0 A8 01 01) to 130.216.46.154 (82 D8 2E 9A). It contains 56 bytes of data.

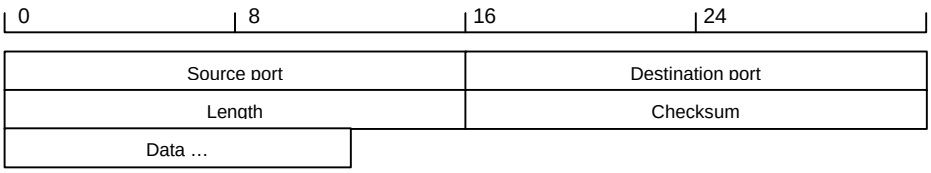
The User Datagram Protocol (UDP)

UDP is an unreliable, connectionless transport mechanism. The word unreliable shouldn't be taken to mean that UDP often loses packets. It is more an indicator of the fact that there is no acknowledgment for packets and so delivery is not *guaranteed*. UDP is used for common Internet services such as the Domain Name Service (DNS). It is suitable for short transaction client/server applications or where a station transmits period information. For instance a weather monitoring station might periodically transmit the temperature and humidity. If a packet is lost there is no problem because another will be transmitted shortly anyway.

UDP provides a finer addressing scheme that Internet address through the use of port numbers. A packet comes from a *source port* and is delivered to a *destination port*.

Packet Format

The format of a UDP packet is as follows:



UDP packet format

Port numbers are specified as 16 bit numbers. The length includes the UDP header and data only. The checksum is computed over the UDP header and data. If the checksum is zero then it is ignored. Since the SX doesn't buffer packets, computing a checksum over the packet contents is impossible when the checksum must be transmitted in the packet header.

Here is an example UDP packet containing the data "ABC":

```
45 00 00 1F 00 02 00 00
0F 11 38 B1 C0 A8 01 01
82 D8 2E 9A 04 01 04 00
00 0B 00 00 61 62 63
```

This packet is from port 1025 on machine 192.168.1.1 to port 1024 on mahcine 130.216.46.154.

Using PPP/UDP in an Application

One of the biggest considerations when designing packet oriented communications software is whether or not to buffer the packets (this is separate from buffering performed by the UART). With only 136 bytes of RAM available on the SX, buffering even a single ICMP packet would require half of the available memory. Communication is full-duplex meaning that both transmit and receive buffers would be necessary. Hence even implementing ICMP would require more RAM than the SX has. Since the buffer size would be limited this would also place a constraint on the maximum packet size the software could accommodate. The advantage of using a buffer is that computing checksums becomes much easier and there are no problems with regenerating data if a packet must be retransmitted. The software described in this technical note doesn't use packet buffering. Instead it processes both received and transmitted data streams a byte at a time.

API

The source code is divided into a large number of subroutines. Those routines that are likely to be used by application code form the network stack API and are described below. All of these routines should be called with a `call` instruction. They all return with a `retp` instruction. You should assume that every API call will change the bank and won't preserve the W register. Some routines expect a parameter in the W register and some will use W or the zero flag to indicate a result. Since the routines are deeply nested care should be taken that application code that is also deeply nested doesn't overflow the call stack.

Physical Layer

The physical layer handles communication with the VP UART.

`PhyTxByte`	Add a byte to the UART's transmit queue. PPP transparency is added at this point and the byte is accumulated into the FCS.
`PhyTxByteNoFCS`	Add a byte to the transmit queue. Transparency is added but the FCS is not calculated. This is for transmitting the FCS and flag bytes.
`PhyRxTest`	Test to see if there are any bytes waiting in the receive queue. This routine does not block if there are no bytes waiting. The zero flag is set if at least one byte is available, otherwise it is cleared.
`PhyRxByte`	Return a byte from the receive queue. If there are no bytes waiting then the routine blocks until a byte is received. Any transparency bytes are removed and the receive FCS is accumulated.
`ModemConnect`	Required when connecting to Windows using Dial-up Networking (DUP). DUP needs to think that it is talking to a modem. This routine pretends to be accept the standard AT command set. In reply to any command from the peer starting with AT it replies with OK. In response to any command starting with ATDT it replies with CONNECT and returns. Once `ModemConnect` has returned DUP thinks that it is talking with the remote host and not its local modem.

PPP

The PPP layer handles the PPP option negotiation protocols.

`PPPOpen`	Negotiate and open a PPP connection with the peer. If a connection is successfully opened then the `linkUp` bit in the `PPPFlags` register will be set.
`PPPRxData`	Once the PP link is up `PPPRxData` is used by the IP routines to receive IP packets. This routine accepts a PPP frame header. If the frame contains LCP or IPCP data then it is handled internally. If the frame

	contains IP data then the zero flag is set and the routine returns. At this point the next byte in the receive queue will be the first byte of the IP header.
PPPClose	Close the open PPP connection. A terminate-request packet is sent to the peer and the link state machine is reset. The routine does not wait for the terminate-ack packet from the peer so the peer may require a timeout period before it is ready for another connection.
PPPClosePacket	Send the trailing bytes of a PPP frame. Every PPP frame ends with 16 bits of FCS and a flag character. This routine sends the FCS and flag character.
PPPCheckFCS	Check that the FCS for a received packet is valid. The FCS must be the next two bytes to be received. The zero flag is set if the frame is valid and cleared otherwise.

IP

At the IP layer the IP header is decoded, the received protocol determined and ICMP echo requests are handled.

IPReceivePacket	Process incoming packets. This routine should be called periodically by the application to ensure that incoming data is processing. If there is no data in the receive queue then the routine returns immediately. If there is data waiting then control is passed to `PPPRxData` to receive the PPP frame header. If the incoming data was not an IP data packet then it is processed by `PPPRxData` and `IPReceivePacket` returns. If the incoming data is an IP packet then the complete IP header is read into memory and the protocol decoded. If the packet is an ICMP echo request then an echo reply is generated and sent. Otherwise the routine returns setting values in `IPFlags` to indicate how the packet should be processed. If it is a UDP packet then `IPFlags.UDPPacket` will be set. If it is a TCP packet then `IPFlags.TCPPacket` will be set.
IPStartPacket	Transmit an IP packet header. The information about the packet is read from the following registers which must be set before the call: `IPLength` Length of the data to be sent. `IPProtocol` Protocol to be sent. `IPDestAddress1` `IPDestAddress2` `IPDestAddress3` `IPDestAddress4` Destination address of the packet. The length field should be the length of the packet data and does not include the IP header.
IPRxClosePacket	Consume the rest of an incoming packet that is not processed by the application.
IPTxData	This is an alias for `PhyTxByte`.
IPRxData	This is an alias for `PhyRxByte`.

UDP

These are utility routines to make it easier to send and receive UDP packets.

UDPStartPacket	Start sending a UDP packet. First `IPStartPacket` is called to send the IP header. Then the UDP header is transmitted. The variables

	UDPSrcPorth, UDPSrcPortl, UDPDestPorth, and UDPDestPortl must be set to indicate the ports to use. IPLength should be set to the length of the data to be sent in the packet (not including the UDP header).
UDPRxHeader	Decode a UDP header. Once IPReceivePacket has indicated that a UDP packet is being received UDPRxHeader should be called to receive the UDP header. The port fields in the header are saved into their converse registers. I.e. the source port is saved in UDPDestPort and the destination port is saved in UDPSrcPort. This is to make it easier to reply to a packet.

Writing Application Code

The following example illustrates how the API could be used in an application. This application uses UDP to read and write file registers on the SX. It is described in more detail in a later section.

If we are communicating with Windows Dial-up Networking then pretend to be a modem.

```
IF WIN95 = 1
   call  @ModemConnect           ; Pretend we are a modem.
ENDIF
```

Try and open the PPP link. If the link is successfully opened then PPPFlags.linkUp will be set.

```
      call  @PPPOpen
      sb    PPPFlags.linkUp     ; Is the link up?
      jmp   :done               ; No.
```

Now loop indefinitely receiving and processing IP packets.

```
   :loop    call  @IPReceivePacket
```

See if the received packet is UDP.

```
      bank  IPVars
      snb   IPFlags.UDPPacket
      jmp   :UDPRx
```

If it isn't UDP then we aren't interested and the next line will consume it.

```
      call  @IPClosePacket    ; Consume the rest of the packet.
      jmp   :loop             ; Loop forever.
```

The next code fragment interprets the received UDP packets.

```
   :UDPRx
      call  @UDPRxHeader            ; Receive the UDP header.
      bank  UDPVars
      cse   UDPSrcPorth,#(DemoPort&$ff00)>>8    ; Check the port.
      jmp   :gobble
      cse   UDPSrcPortl,#DemoPort&$00ff
      jmp   :gobble
```

Generally, different UDP based applications are differentiated by the port number they use. For this demonstration port 280 has been chosen arbitrarily. Each incoming packet is checked to see if it is for port 280. If it is not for this port it is discarded. The next code reads the first byte from the packet data and decides what to do.

```
      ; OK the packet is for the right port.
      call  @IPRxData
```

```
   mov   Scratch0,w
   cje   Scratch0,#DemoMemDump,:memdump ; Is the command a dump?
   cje   Scratch0,#DemoMemSet,:set      ; Is the command a set?
   cje   Scratch0,#DemoMemGet,:get      ; Is the command a get?
   cje   Scratch0,#DemoHello,:hello     ; Is the command a hello?
```

Discard an unwanted packet.

```
:gobble
   call  @IPRxClosePacket
   jmp   :loop
```

To handle a get request we read the address from the received packet and then reply with a packet containing the byte at that address.

```
:get
   bank  IPVars
   mov   IPLength,#1
   mov   IPDestAddress1,IPSrcAddress1  ; Copy the address of the
   mov   IPDestAddress2,IPSrcAddress2  ; sender.
   mov   IPDestAddress3,IPSrcAddress3
   mov   IPDestAddress4,IPSrcAddress4
```

Start by setting up the IP packet header variables. The return packet will contain only one data byte and will have the same destination address as the source of the packet just received. There is no need to set the protocol because it will be the same as the received packet.

Start transmitting the reply UDP packet. This sends the PPP frame header, IP packet header and UDP header.

```
   call  @UDPStartPacket   ; Start the reply packet.
```

We read the address from the received packet and use indirect addressing to get the required register. This value is then transmitted as the data of the reply packet.

```
   call  @IPRxData   ; Read the address.
   mov   FSR,w       ; Use indirect addressing.
   mov   w,IND       ; Load the byte.
   call  @IPTxData   ; Transmit it.
```

The packet is closed by sending the PPP FCS and flag sequence. We are now ready to process the next incoming packet so return to the loop.

```
   call  @PPPClosePacket ; Finish the packet.
   jmp   :loop
```

The code for dumping and setting memory continues in the same way.

Caveats and Limits

There are a few limitations imposed by the IP stack. These are a natural consequence of the limited resources available on an 8-bit micro-controller.

- Packets are limited to 256 bytes. This is because only the LSB of the packet length is used. Longer packets could be accommodated by using a second byte to store the length.
- Closing a PPP connection may require forcing the peer to timeout (about 9 seconds). This is because the SX doesn’t send a terminate-acknowledge packet.
- Fragmented IP packets are not handled at all. However fragmentation is unlikely to occur if packet sizes are limited to 256 bytes.

- The UDP checksum is not calculated. The CRC type FCS provided by the PPP layer is much better than the simple, summed IP checksum anyway. Calculating the UDP checksum would be difficult without buffering packets since it is computed over the data but transmitted in the header. UDP allows the checksum to be ignored if the checksum field is set to zero.
- Echo requests are the only ICMP packets handled.

Configuring the PPP Peer

To connect to the SX the peer must be configured with some suitable PPP settings. The next two sections describe how to do this for Windows 95/NT and Linux.

Windows 95/NT

Dial-up Networking and TCP/IP must be installed. In the network control panel install the Dial-up Adapter and make sure it is bound to TCP/IP only. Create a new modem in the Modems control panel. Avoid the 'Detect my modem…' wizard and choose it yourself. It should be a standard 19,200 baud modem connected to the same COM port the SX will be connected to. In the System control panel for the COM port, under the advanced settings, make sure that the FIFO is turned OFF.

Now create a new connection in the Dial-up Networking window. It should use the new modem you just created. It doesn't matter what the telephone number is. You need to specify the IP address for the local machine in the same subnet as the SX. For example, by default the code comes with the IP address 192.168.10.1 for the SX so set your machine to 192.168.10.2. Any options for IP compression, header compression, software compression, CHAP or PAP should be OFF. Use default gateway on local network should be ON. There is no need to set DNS addresses.

Now the connection is ready to go. Reset the SX then open up the connection document and click 'Connect'. A PPP connection will be negotiated. Once the link is up try typing `ping 192.168.10.1` into a DOS window. You should see a reply from the SX with the round trip time. If Dial-up Networking reports that a PPP connection has been opened but `ping` doesn't work then your routing table is probably not set up correctly.

You may wish to read the following documents from the Microsoft Knowledge Base (at support.microsoft.com) if you have any problems getting a PPP connection:

- How to enable PPP logging under Windows NT: Q115929
- How to interpret the ppplog.txt file: Q156435

Two common problems are: not disabling the UART FIFOs on the PC and using the wrong type of cable. The FIFO must be disabled because even when the SX drops the CTS line the PC continues to transmit the data in the FIFO, which is enough to over-run the SX's receive buffer.

The serial cable should be straight through (not null modem) and must have the rx, tx, rts, cts and gnd pins connected.

Linux

The SX has been tested under version 2.0.30 of Linux with pppd 2.3.4.

Create a script to start the PPP daemon which looks something like the following:

```
# Start a PPP server running on this machine.
/usr/sbin/pppd debug kdebug 7 19200 /dev/cua0 passive \
               persist crtscts 192.168.10.2:
```

You must be root to run the PPP daemon. Reset the SX and run the script; a PPP connection will be negotiated. To see if the PPP layer is up run `/sbin/ifconfig`. It should list the network interface PPP0 which is the PPP connection to the SX. As with Windows, `ping` can be used to check the connection.

If there are any problems, inspecting /var/log/debug and /var/log/messages can indicate where the PPP negotiation failed.

Demo Application

As a demonstration of the PPP/UDP code a simple application has been developed. The SX will respond to requests to read and write the file registers. The commands are encapsulated in UDP packets and so can be sent from any computer connected to the Internet (provided intervening firewalls don't block the packets). Since there is no standard application for sending UDP packets a short 'C' program is shown in the appendices which will communicate with the SX.

Every packet received on port 280 of the SX will be treated as a command. The first byte of the UDP packet determines what action will be taken:

- If it is $10 then a reply packet will be generated containing the full SX register set.
- If it is $20 then the next byte will be read as an address and the third byte in the packet will be written to that address. In this way a file register can be modified remotely.
- If it is $30 then the next byte is read as an address. The register at that address is returned in a reply packet.

Here is an interaction with the SX using this demo program:

```
C:\Demo>sxdemo -d 192.168.10.1
Dumping the SX's register file
Connecting to local port 1024
Using port 280 on the SX
Sending command
Starting receive
$ 0:  8
$ 1:  8
$ 2:  0
$ 3:  0
$ 4:  0
$ 5:  0
$ 6:  0
$ 7:  0
$10: 14 02 00 18 00 11 7D 00
$11: 09 11 00 01 00 41 30 00
$12: 00 C0 00 00 00 D0 86 00
$13: 0E A8 00 04 00 09 27 00
$14: 0B 01 00 C8 00 10 7E 00
$15: 00 02 00 01 00 00 67 00
$16: 00 DC 00 20 00 18 70 00
$17: 03 27 00 D9 00 7E 5E 00
$18: 00 BD 00 4C 00 00 67 00
$19: 00 02 00 00 00 D5 70 00
$1A: E4 C0 00 00 00 D2 5E 00
$1B: EA A8 00 00 00 03 67 00
$1C: 5E 01 00 00 00 D9 70 00
$1D: FE 02 00 00 00 DA 5E 00
$1E: 01 1D 00 00 00 07 5E 00
$1F: 02 00 00 00 00 00 7E 00

C:\Demo>sxdemo -g 144 192.168.10.1
Getting the register at address 90
Connecting to local port 1024
```

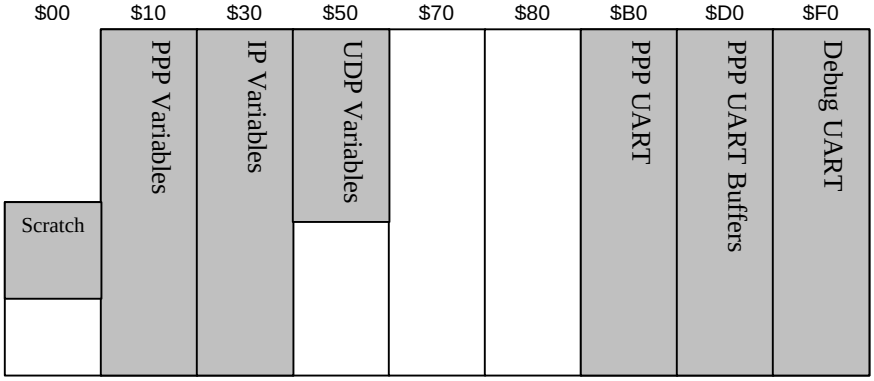
```
Using port 280 on the SX
Sending command
Starting receive
Register value: 00

C:\Demo>sxdemo -s 144 12 192.168.10.1
Setting the register at address 90 to 0C
Connecting to local port 1024
Using port 280 on the SX
Sending command
Set command sent

C:\Demo>sxdemo -g 144 192.168.10.1
Getting the register at address 90
Connecting to local port 1024
Using port 280 on the SX
Sending command
Starting receive
Register value: 0C
```

Source Code Description

The code uses a software UART VP to implement the physical communications layer. A second UART VP can optionally be used for transmitting debug information. The existence of the debug UART is controlled by the define `DEBUG` which is set or unset at the top of the source code. A second define (`WIN95`) is used to indicate whether the PPP-VP will be talking to a Windows 95 PC (which requires modem AT command set emulation).

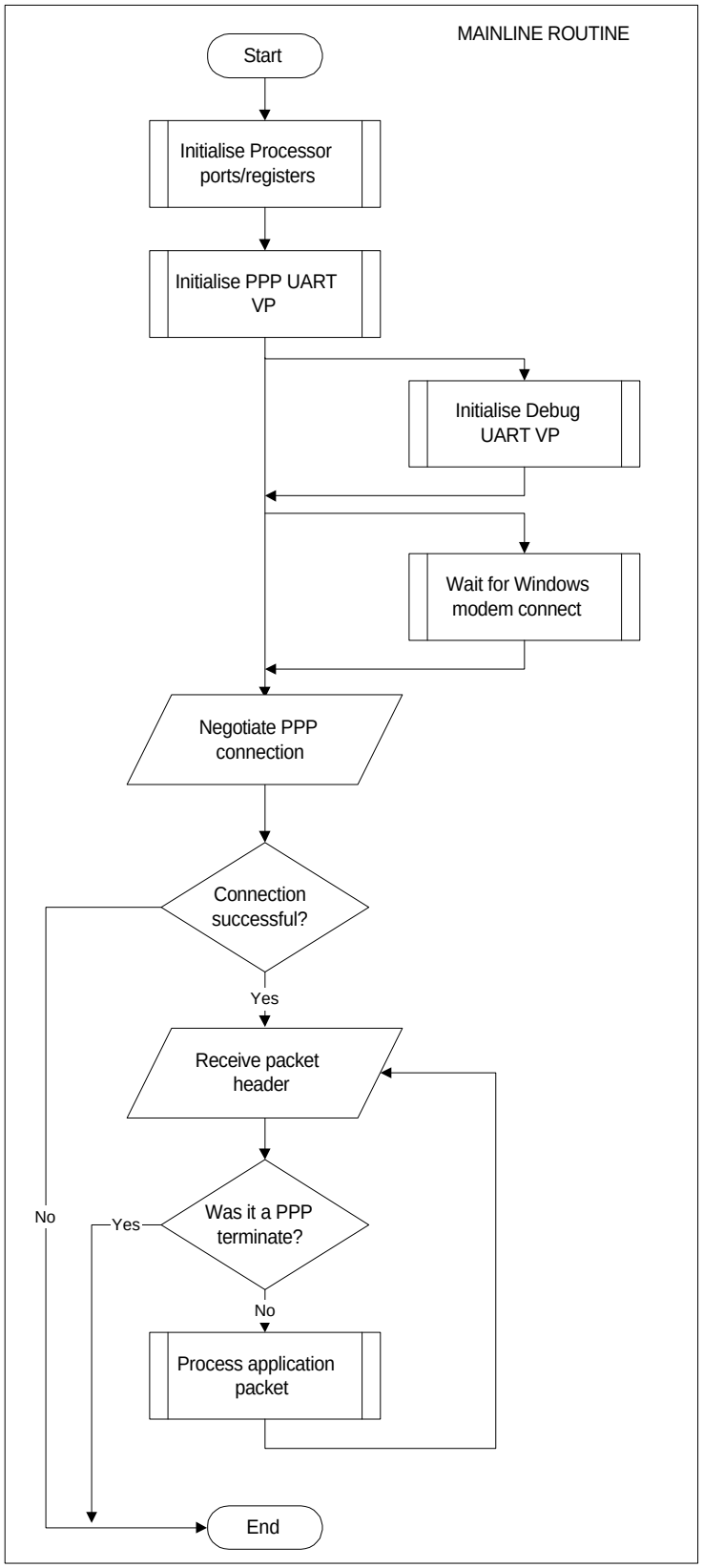
The interrupt service routine is used to run the two UART VPs. The rest of the network stack runs in the mainline code. It is up to the application code to initiate a PPP connection and then to ask for an incoming packet to be received and processed.

The following diagram shows how the file registers are used by the VP. At least 32 bytes of banked registers as well as half the global registers are available to the application code.



File register usage

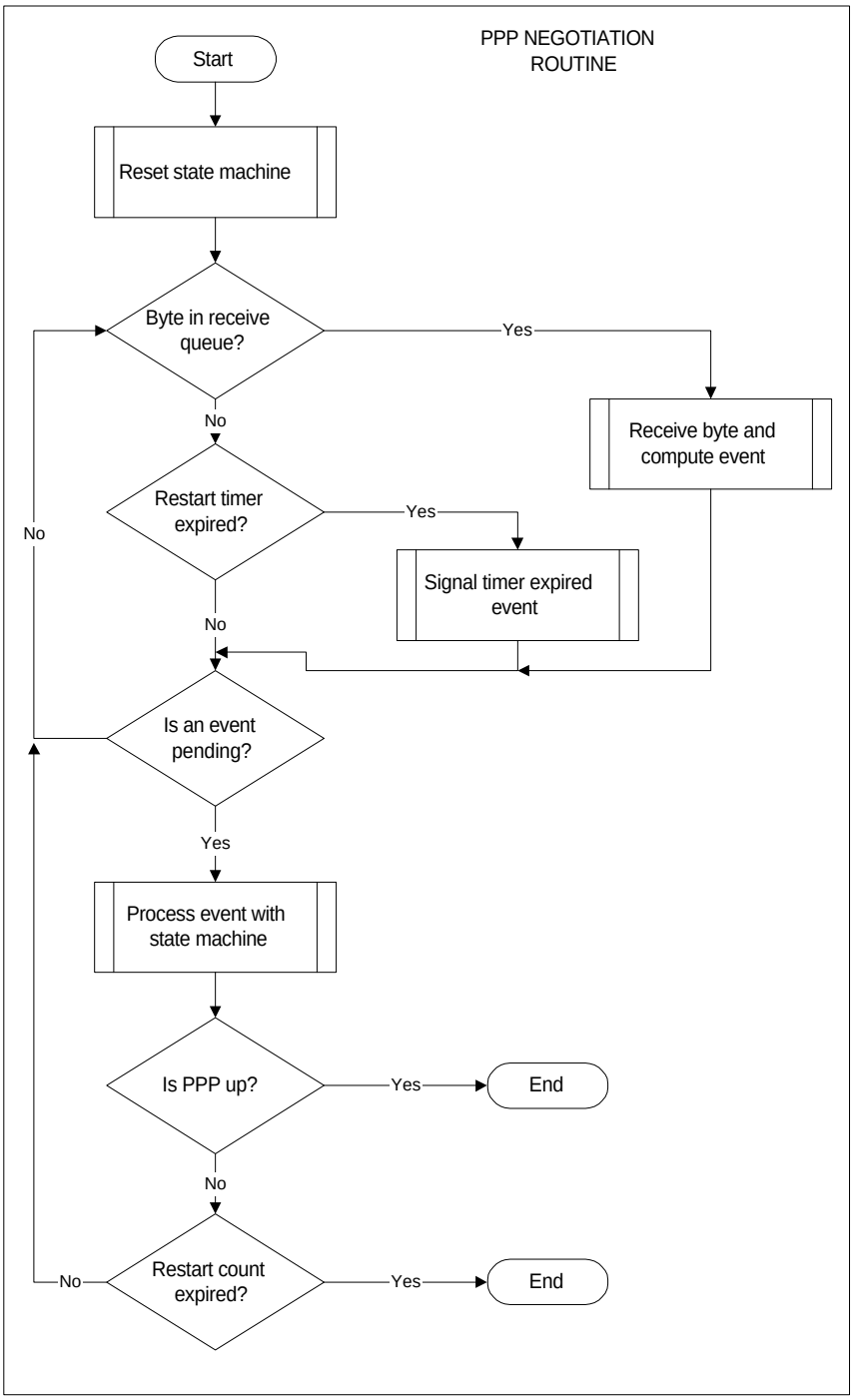
The flowcharts on the following pages show the mainline routine, and the PPP negotiation.



The mainline routine is entered after processor reset. It initialiazes the registers used by the UARTs before the application code takes over.

The application code will then initiate a PPP connection. If the connection is successful the main loop will start.

Each iteration the loop processes one incoming IP packet.



The PPP negotiation routine waits for a byte to be received. Each byte is processed by the `PPPReceive` routine. When enough of a frame has been received to determine the frame type an event is returned to be processed by the PPP state machine.

Negotiation ends when the PPP link is up, too many timeouts have been received or acceptable options cannot be negotiated with the peer.

The Hardware

Two UART virtual peripherals are used by the PPP/UDP code. The first is for the PPP physical layer and the second is to provide a trace of debugging information.

To negotiate PPP settings with a PC it is necessary to use hardware flow control on the PPP UART. Since PC has much larger buffers than the SX it is likely that the SX might lose data. RTS/CTS flow control is used.

The debugging UART is transmit only and so doesn't need flow control.

On this demonstration board port C of the SX is available to connect peripherals. Port B is used by the UARTs and port A controls the status LEDs. The five LED indicate what the board is doing:

PWR Power

ERR An error has occured

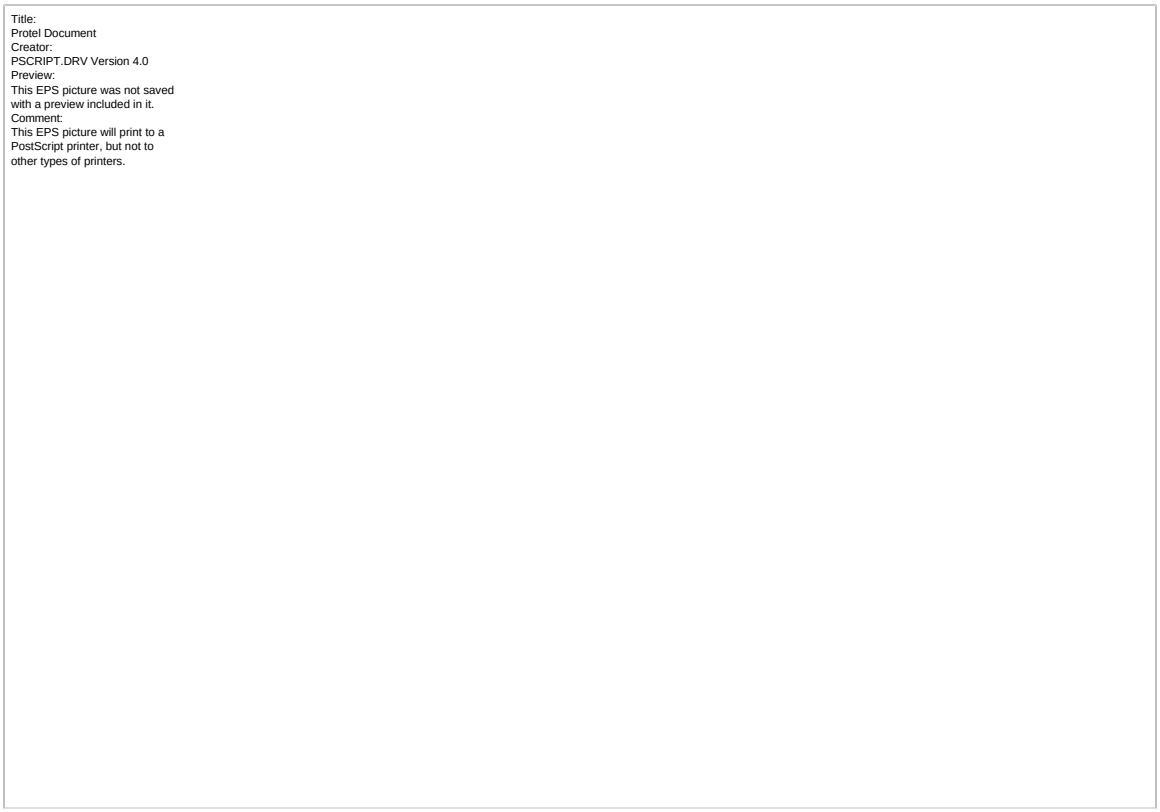
TR Data is being transmitted or received by the PPP UART

UP The PPP link is up

NEG PPP Negotiation is in progress

The power input on the demonstration board requires 9V DC or 7V AC.

Circuit Diagram



Appendix: UDP 'C' Code

```
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <mem.h>

#define WIN32

#ifdef WIN32
#include <winsock.h>
#else
#include <sys/socket.h>
#include <netinet/in.h>
#endif

#define BUF_LEN       1024

#define DUMP_COMMAND  0x10
#define SET_COMMAND   0x20
#define GET_COMMAND   0x30
#define HELLO_COMMAND 0x40

extern int errno;

void print_help(char *prog) {
  printf("%s [-hedsg] ip - Send and receive UDP packets to an SX microcontroller.\n", prog);
  printf("   ip             IP address of the SX in x.x.x.x notation\n");
  printf("   -h             This help message.\n");
  printf("   -e             Get the SX's hello message.\n");
  printf("   -d             Dump the SX's register file.\n");
  printf("   -s addr value  Set the register at the address to a certain value.\n");
  printf("   -g addr        Get the value at the given address.\n");
  printf("   For the -s and -g commands the values must be given in decimal\n" );
}

unsigned int decodeAddress( char *a ) {
  char *dot;
  unsigned int addr = 0;

  dot = strchr(a,'.');
  if (!dot )
    return 0;
  *dot = '\0';
  addr = atoi(a)<<24;

  a = dot + 1;
  dot = strchr(a,'.');
  if (!dot )
    return 0;
  *dot = '\0';
  addr |= atoi(a)<<16;

  a = dot + 1;
  dot = strchr(a,'.');
  if (!dot )
    return 0;
  *dot = '\0';
  addr |= atoi(a)<<8;

  a = dot + 1;
  addr |= atoi(a);

  return addr;
}

int main( int argc, char *argv[] ) {
  int err, i, j;
  int clientLen, lenReceived, commandLen;
  unsigned char buffer[BUF_LEN];
  int sock;
  struct sockaddr_in addr, clientAddr;
  unsigned char commandBuf[3];
  char command;
  unsigned char address, data;
  unsigned int ipAddr;

#ifdef WIN32
```

```
  WSADATA lpWSAData;
#endif


  /* Decode the command line options. */
  if ( argc > 1 ) {
    if ( !strcmp( argv[1], "-d" ) ) {
      command = 'd';
      commandBuf[0] = DUMP_COMMAND;
      commandLen = 1;
      printf("Dumping the SX's register file\n");
    }
    else if ( !strcmp( argv[1], "-e" ) ) {
      command = 'e';
      commandBuf[0] = HELLO_COMMAND;
      commandLen = 1;
      printf("Reqesting hello message\n");
    }
    else if ( !strcmp( argv[1], "-s" ) && argc == 4 ) {
      command = 's';
      address = atoi( argv[2] );
      data = atoi( argv[3] );
      commandBuf[0] = SET_COMMAND;
      commandBuf[1] = address;
      commandBuf[2] = data;
      commandLen = 3;
     printf("Setting the register at address %2.2X to %2.2X\n", address, data);
    }
    else if ( !strcmp( argv[1], "-g" ) && argc == 3 ) {
      command = 'g';
      address = atoi( argv[2] );
      commandBuf[0] = GET_COMMAND;
      commandBuf[1] = address;
      commandLen = 2;
      printf("Getting the register at address %2.2X\n", address);
    }
    else {
      print_help(argv[0]);
      return 1;
    }
  }
  else {
    print_help(argv[0]);
    return 1;
  }

#ifdef WIN32

  /* Windows requires that winsock be initialized. */
  err = WSAStartup (0x0101, &lpWSAData);
  if ( err != 0 ) {
    printf("Cannot open WinSock\n");
    return 1;
  }
#endif


  /* Get the IP address of the destination. */
  ipAddr = decodeAddress( argv[argc-1] );
  if ( ipAddr == 0 ) {
    printf("Invalid IP address\n");
    return 1;
  }
  printf("IP address of SX: %d.%d.%d.%d\n", (ipAddr&0xff000000)>>24,
         (ipAddr&0x00ff0000)>>16, (ipAddr&0x0000ff00)>>8, ipAddr&0x000000ff );

  clientLen = sizeof( clientAddr );

  sock = socket(AF_INET, SOCK_DGRAM, 0);
  if ( sock < 0 ) {
    perror("socket");
    return 1;
  }

  memset( (char*) &addr, 0, sizeof( addr ) );
  memset( (char*) &clientAddr, 0, sizeof( clientAddr ) );

  addr.sin_family = AF_INET;
  addr.sin_port = htons(1024);
  addr.sin_addr.s_addr = INADDR_ANY;
```

```
  printf("Connecting to local port %d\n", ntohs(addr.sin_port) );

  err = bind( sock, (struct sockaddr*) &addr, sizeof(addr) );
  if ( err == -1 ) {
    perror("bind");
    return 1;
  }

  clientAddr.sin_family = AF_INET;
  clientAddr.sin_port = htons(280);
  printf("Using port %d on the SX\n", ntohs(clientAddr.sin_port) );
  clientAddr.sin_addr.s_addr = htonl( ipAddr );

  printf("Sending command\n" );

  if (sendto( sock, &commandBuf, commandLen, 0, (struct sockaddr*)&clientAddr,
          sizeof( clientAddr )) == -1 ) {
    perror("sendto");
    return 1;
  }

  if ( command == 's' ) {
    printf("Set command sent\n");
    return 0;
  }

  printf("Starting receive\n");

  lenReceived = recvfrom( sock, &buffer, BUF_LEN, 0, (struct sockaddr*)&clientAddr, &clientLen);

  if ( command == 'e' ) {
     printf("Hello message from the SX: %s\n", &buffer );
  }
  else if ( command == 'd' ) {
    if ( lenReceived != 192 ) {
      printf("No enough data received. Expected 192 bytes.\n" );
       return 1;
    }

    for ( i = 0; i < 8; i++ )
      printf("$%2X: %2X\n", i, buffer[i] );

    for ( i = 0; i < 16; i++ ) {
      printf("$%2.2X: ", i + 16 );
      for ( j = 0; j < 8; j++ )
     printf("%2.2X ", buffer[8 + i + (j * 24)] );
      printf("\n");
    }
  }
  else if ( command == 'g' ) {
    if ( lenReceived != 1 ) {
      printf("No enough data received. Expected 1 byte.\n" );
      return 1;
    }

    printf("Register value: %2.2X\n", buffer[0]);

  }
  return 0;
}
```

References

[1] Douglas E. Comer, “Internetworking with TCP/IP”, Prentice-Hall, 1995.