TCP Virtual Peripheral Implementation

Application Note 27
Christopher Waters
September 1999

1.0 Introduction

This application note describes an implementation of the Transmission Control Protocol (TCP) for the Scenix SX microcontroller. TCP is a transport layer protocol from the TCP/IP network stack. It provides reliable, flow controlled, connection based host-to-host data transfer across the Internet. TCP is used as the transport layer for many of the most common Internet protocols, including HTTP (world-wide-web), SMTP and POP3 (email), Telnet (host command line interface) and FTP (file transfer).

The next section describes the TCP protocol and special features of the Scenix SX implementation. This is followed by a section describing the TCP API which is used to write application layer protocols which use the TCP transport services.

The TCP Virtual Peripheral™ uses the other Scenix TCP/IP Virtual Peripheral modules described in application note AN23 (UDP/PPP Virtual Peripheral Implementation). AN23 also describes how to configure a Windows or Linux PC to access the stack over PPP. It should be read in conjunction with this document to understand the complete Virtual Peripheral source code.

2.0 The TCP/IP Stack

For a general overview of the TCP/IP protocol stack, see application note AN23. This application note focuses on TCP protocol shown in Figure 2-1.

The transmission control protocol (TCP) provides a data delivery service. It differs from UDP in that it is connection-oriented and reliable.

2.1 The Transmission Control Protocol (TCP)

TCP is described in the IETF Request for Comments RFC793. It is a host-to-host transport layer protocol for the reliable transmission of data over the Internet.

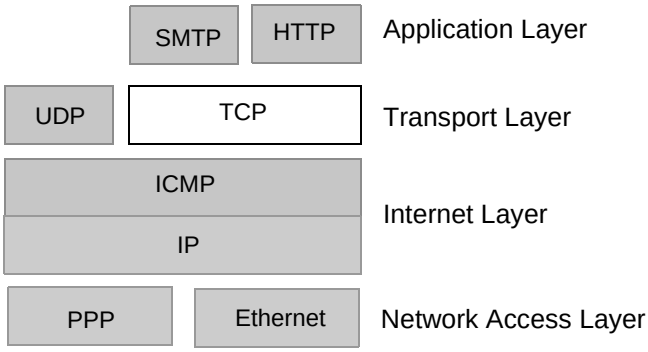

Figure 2-1. The Internet Protocol Stack

2.2 Packet Format Overview

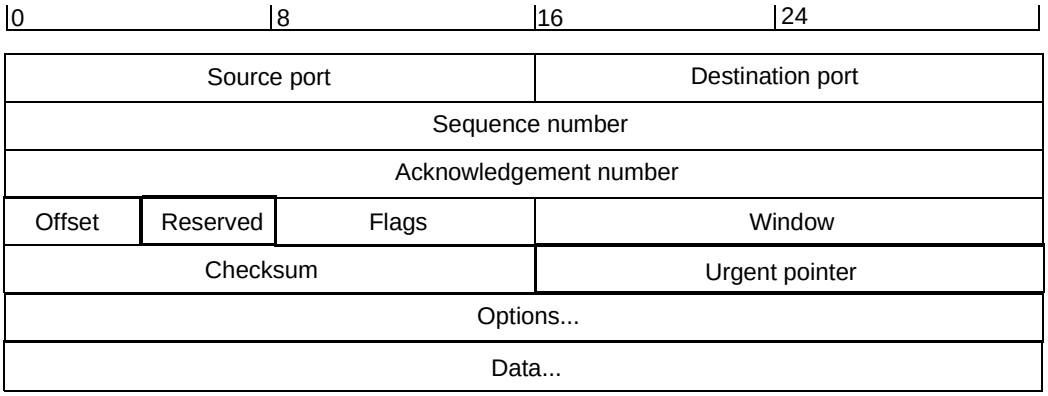

Figure 2-2. TCP Protocol Header

The source port is the conceptual network port this segment originated from. The destination port is the port the segment is meant for. A unique TCP connection is determined by the combination of source and destination IP addresses and source and destination ports. The numbering of the destination port is usually used to determine what application the TCP connection relates to. For example, SMTP is assigned port number 25 and HTTP is assigned port 80. So, by convention, an HTTP server will listen on port 80 for incoming connections.

The sequence number and acknowledgment number are used by either end of the connection to achieve reliable data transfer. The sequence number is the number of the first byte (called octets in the TCP RC) contained within this segment. The acknowledgment number optionally contains the next sequence number the sender expects to receive from the remote host.

The data offset is the number of 32-bit words contained in the TCP header. It can be used to skip over any options in the header.

The control bits determine the semantics of the packet:

- URG – This packet contains urgent data and the urgent pointer field is valid.
- ACK – This packet is an acknowledgment and the acknowledgment field is valid.
- PSH – The data in the packet should be processed immediately, rather than waiting for a full buffer or an even number of bytes.
- RST – An error has occurred and the connection should be reset.
- SYN – The packet requests sequence number synchronization.
- FIN – The connection is closing.

More than one control bit can be set at a time indication multiple conditions.

The window indicates to the remote the number of bytes the sender is willing to accept. By changing the window size hosts can perform flow control and minimize dropped packets due to buffer overruns.

The checksum field contains a checksum over the TCP header, the TCP data and a pseudo header made up of fields from the IP header. Computing the checksum is particularly problematic on a memory limited device and a later sections covers this in more detail.

The urgent pointer and options are not used by the SX implementation. TCP has the ability to send out-of-band data, usually cancel commands to halt the flow of data.

Following the header is the packet data, if any. The data is not padded and can be an odd number of bytes. The length of the data is determined by the length field in the IP header. No translation, such as byte-stuffing for transparency, is performed on the data. This is left to the network access layer protocol.

2.3 TCP Checksum

The checksum field in the TCP header contains the ones complement of the 16-bit ones complement sum of the TCP packet with an attached pseudo header. The pseudo header consists of the source and destination address, protocol type and length from the IP header. The checksum is computed over the packet data, but transmitted in the header, which poses a problem for a device which doesn't buffer a packet before transmit. There are two ways around this problem. The first is to transmit an arbitrary checksum in the header. As the packet data is transmitted the real checksum is computed and two extra bytes are appended to the data so that the checksum in the header is correct. The problem with this technique is that there is no universal way to tell the recipient of the packet to ignore the last two bytes of the packet. These bytes will interfere with most existing TCP based application protocols.

The alternative, which is the method used in the SX, is to code the application in such a way that it can generate the packet data more than once. The application is thus asked to generate the packet once and the checksum is computed. This, correct, checksum is transmitted in the header. Then the application data is computed again and this time transmitted in the packet. Obviously this technique imposes some constraints on applications. There

are two possible solutions where applications can't generate the same data more than once:

- Use spare RAM to buffer packet data before transmit.
- Design a new application level protocol and then used the byte-stuffing checksum technique.

2.4 Sequence Numbers

Each unique byte transferred over a TCP connection is uniquely numbered. These are the sequence numbers. When a connection is initiated each host generates an (effectively random) initial sequence number ISN). Each subsequent byte is numbered relative to the ISN. Each TCP maintains three variables for managing the connection's sequence numbers:

- SND.UNA – The oldest unacknowledged sequence number.
- SND.NXT – The next sequence number to be sent.
- RCV.NXT – The next sequence number to expect from the remote host.

When there is no outstanding data to be sent then SND.UNA = SND.NXT.

2.5 Three Way Handshaking

In order for the two ends of the connection to know each other's ISN an initial exchange of ISNs, called a three way handshake (because three packets are exchanged) is used. The three way handshake is used to initiate the connection, and once it has completed data can be reliably exchanged. The handshake works as follows:

1. A → B	SYN	My ISN is X.
2. B → A	ACK,SYN	My ISN is Y, I acknowledge that your ISN is X.
3. A → A	ACK	I acknowledge that your ISN is Y.

The ISNs are exchanged in the sequence number field.

3.0 Application interface to the TCP Virtual Peripheral

The TCP protocol on the SX is designed so that it does not buffer any packet data. Once the application has informed the TCP Virtual Peripheral it wants to initiate a connection, or that it is willing to accept connections, then the flow of control is passed to the TCP Virtual Peripheral. The TCP layer will subsequently call application routines to request data, or indicate that data has been received. Any new application must provide the nine event API routines.

3.1 TCP Interface

These routines are used internally by the TCP Virtual Peripheral. For some applications it might be desirable to call them directly.

TCPPassiveOpen

Do a TCP passive open. The SX will then accept connections on the port specified in `TCPLocalPorth` and `TCPLocalPortl`. This routine should be used to start a connection if the SX is to act as the server.

TCPActiveOpen

Do a TCP active open. The SX will initiate a connection with a remote server. Both the local and remote port numbers and IP addresses should already have been set.

TCPClose

Force the open TCP connection to close.

TCPTransmit

This routine should be called periodically in the main loop of the program to allow time to transmit TCP packets.

TCPProcessPacket

This routine should be called when the `IPReceive-Packet` routine indicates that a TCP packet has been received.

4.0 Application Programming Interface

The API is event driven. This is arrangement allows the byte-at-a-time processing, but does require a different programming style to be adopted when writing application level protocols. Rather than calling sub-routines to send and receive data, the application writer implements sub-routines which are called by the TCP when data has been received or can be transmitted. To create a new application the following sub-routines must be implemented:

AppInit

Called to allow the application initialize variables and possibly do a passive open.

AppBytesToSend

Called by the TCP/IP stack to see if the application has any data to transmit. The application should return with the number of bytes it wishes to send in the 'w' register.

AppBytesAvailable

Called by the TCP/IP stack when a segment is being received. 'w' contains the number of bytes of data that have been received. This routine is a warning to the application that its `AppRxData` routine is about to be called 'w' times.

AppTxData

Called once for each byte to be sent in a segment. The byte to be transmitted should be returned in 'w'. The application must maintain a counter that is incremented once for each call to `AppTxData`.

AppRxData

Called once for each received byte. Until the complete segment has been received the TCP/IP stack cannot be sure whether the segment has been corrupted in transmit. Therefore the application should not make any irreversible changes based on the incoming data, until `AppPacketOK` is called.

AppAck

Indication to the application that the last segment transmitted has been acknowledged. This indicates that the segment won't need to be transmitted again.

AppNak

Indication to the application that the last segment has not been acknowledged. Subsequent calls to AppTxData should return the same segment data again because the segment will be transmitted again.

AppPacketOK

The last segment received was not corrupt. At this point the application can use the segment data.

AppPacketBad

The last segment received was corrupt. The application should expect to receive the segment again.

4.1 Using the API

The best way to understand how this API works is with an example event sequence. The first example is of transmitting a short (3 bytes of data) segment. The sequence of sub-routine calls might look like the following:

```
AppBytesToSend (returns w=0)
AppBytesToSend (returns w=0)
…
AppBytesToSend (returns w=3)
AppTxData (returns w=43, first data byte)
AppTxData (returns w=8, second data byte)
AppTxData (returns w=56, third data byte)
AppNak
AppTxData (returns w=43, first data byte)
AppTxData (returns w=8, second data byte)
AppTxData (returns w=56, third data byte)
AppAck (or AppNak if segment not acknowledged by remote host)
```

Whenever the TCP is idle it calls the API routine AppBytesToSend to see if the application has an data to send. As long as the application returns 0 no further action is taken. When the application returns non-zero then the TCP knows it has a segment to transmit. So that the TCP checksum can be calculated the application is asked to generate the segment data. This is done by calling AppTxData for each byte in the segment. At the end of the checksum calculation AppNak is called to fool the application into thinking that the segment was not acknowledged by the remote host and that it needs to be transmitted again. AppTxData is then called again for each byte in the segment. If the segment is acknowledged AppAck is called to let the application know it can move on to the next segment. If the segment is not acknowledged then AppNak tells the application to generate the data again and the process repeats until the segment is successfully transmitted.

Receiving a segment will consist of a sequence of events like the following:

```
AppBytesAvailable (w contains number of bytes in segment)
AppRxData (w contains byte received)
AppRxData (w contains byte received)
AppRxData (w contains byte received)
AppRxData (w contains byte received)
AppPacketOK
```

When a TCP segment header is received AppBytesAvailable is called to inform the application that w bytes are about to be received. This is to allow an opportunity to do any pre-receive preparation. AppRxData is called once for each byte of data in the segment. Once the segment is completely received the network access layer CRC is evaluated and AppPacketOK or AppPacketBad are called to indicate whether the packet passed the CRC. If the CRC failed then the application should ignore any data received in the segment. The TCP will inform the remote host to transmit the packet again.

For examples of how the API should be used see applications notes AN26 (SMTP Virtual Peripheral Implementation) and AN25 (HTTP Virtual Peripheral Implementation).

4.2 Source Code Description

The lower layers of the TCP/IP stack are described in application note 23. The TCP protocol can be enabled or disabled in the TCP/IP source code using the defines at the top of the file.

TCP uses two banks of SRAM. One bank holds state information and fields extracted from each packet as it is processed. The second bank holds the transmission control block (TCB) which is the data structure that describes an open TCP connection. Extending the Virtual Peripheral to handle multiple simultaneous connections would require storing and managing multiple TCBs.

4.3 Debugging

The source code contains many debugging statements which can be used to monitor what the TCP/IP stack is doing. The debugging statements use the debug serial port to send special codes to a special program running on a PC. This program, StatusMonitor.exe can decode the debugging bytes and display them as descriptive messages. Debugging messages consist of either one or two bytes. Single byte messages indicate that a point in the code has been reached. In two byte messages the first byte is the message type and second byte is extra information which is decoded by StatusMonitor. The complete listing of codes and their meanings is in the file PacketDef.cpp.

Lit#: SXL-AN27-01

