Web Server & Send Email Client Implementation with Ethernet as the Physical Layer

Application Note 37
Wing Poon
Deon Roelofse
October 2000

1.0 Introduction

This Application Note describes the implementation of a TCP/IP networking stack on the SX52BD communications controller. Ubicom has created an evaluation kit for demonstrating the Ubicom Internet connectivity networking stack. The kit contains a demonstration/evaluation board, the SX source code, and documentation on how to use/customize the stack using the built-in Application Programming Interface (API).

For information on setting up and running the Ethernet SX-Stack demo board, please refer to the Ethernet SX-Stack Board User's Guide.

2.0 Quick Tutorial on Internet Networking

The transfer of data between remote computers or more commonly from applications on computers to applications on remote computers over the internet is done with the use of communication protocols. A suite of network communication protocols have been developed by the internet community and there are forever new additions to this list. Communication protocols are sets of rules that governs how data are transferred as well as how it should be interpreted. Most network communication protocols are defined in documents called RFCs (Request For Comments), for example, RFC793 defines the TCP protocol.

2.1 Network Communications Terminology

Network protocols are defined and developed by a huge number of different programmers and organizations worldwide. There are a lot of conflicting terms in which the workings of networking communications are described. Sometimes two different terms may be used to refer to the same thing. Due to the enormous amount of confusion that is sometimes caused, a short list of terms used in this application note and the specific meaning thereof is in order:

communications protocol - a set of rules of how communications pertaining to a specific protocol should be handled and interpreted.

network protocols - used to refer to a collection of communications protocols used to perform network communications. Commonly called a network device's TCP/IP stack.

network device - used to refer to any electrical device connected to a network we are talking about, able to communicate with other network devices through the use of the network protocols. May be a PC or the Ethernet SX-Stack demo board.

NIC device driver - a distinct piece of software consisting of function calls and variables that is used by network protocols to control the Ethernet controller on a network device.

local - used to refer to communications at the demo board or network device and software under discussion such as a local port or a local server.

remote - used to refer to a physically removed network device or the communications thereof. Always used in the sense that communications are between a local and a remote device. May also be used to distinguish a remote port or a remote server from the local one.

server - used to refer to a software application on a network device such as a PC that offers network services such as SMTP.

client - used to refer to a software application on a network device such as a PC that uses the services of a server.

host - used to refer to a network device offering services to an application such as an email program. For our purposes the terms host and server means exactly the same thing. Could also just refer to an application program on either side of a communications channel, thus the terms local and remote host.

API - Application Programmer's Interface. A set of software functions and variables an application programmer may use to utilize network communications via the network protocols. The TCP API is used to transfer data over a TCP connection.

connection - used to refer to a logical channel or path for communications between two physically removed hosts or applications. A connection is always in a certain state. A connection may be closed or open or in some other state. An open connection is also said to be established. Data can only be transferred between hosts when a connection is open or established.

octet - used to refer to a 8-bit quantity, commonly substituted for “byte”. Some processors have 32-bit bytes and therefore the term octet is used in RFCs to avoid confusion. When this application note uses the term byte, reference is made to 8-bit quantities as used by the Ubicom controller.

data - the bytes a user application or network protocol sends or receives over a connection. It is analogous to the payload of a missile. All the other material is considered to be part of the necessities in delivering the payload. A message can also be called the data.

header - the first number of bytes ahead of the data bytes in a datagram, a segment, or a message. The header contains information described in the RFC of the applicable protocol. The data contained in a segment conveys no information described in the applicable protocol’s RFC. If the protocol was used as a carrier for another protocol, the data may be interpreted according to the RFC of the other protocol eg. SMTP data carried by the TCP protocol.

segment - used to refer to a logical unit of data. Almost universally used to refer to TCP. A TCP segment consists of a header and data. The TCP data may be a HTTP message or SMTP data.

datagram - used to refer to a complete block of data. An internet datagram consists of an IP header and IP data.

packet - used to refer to a block of data physically transmitted over a network. An internet datagram is handed to a network module to transmit as one packet or a series of packets, depending on whether the physical network can send packets as large as the datagram is. If the network cannot, it fragments a datagram into a series of packets which are sent over the network. The packets are reorganized at the receiving network node to obtain the original datagram.

frame - used exclusively to refer to a packet transmitted on an Ethernet network. Commonly used when discussing Ethernet communications when only referring to the Ethernet protocol.

socket - used to indicate the unique logical address of a remote or local TCP connection’s application. A socket contains a TCP port number and an IP address.

2.2 TCP/IP

The term “TCP/IP” is commonly used whenever the topic of Internet communications are discussed. Indeed, “TCP/IP” here refers to a whole suite of networking protocols, with the core building blocks being the IP and TCP protocols. The point to keep in mind here is that TCP/IP can be used to imply one of two things – the first, and more common interpretation, is that it is a collection of all standard networking protocols used for communicating on the Internet, and the second, and less common interpretation, that of the TCP protocol and IP protocol exclusively.

2.3 Packet-Based vs. Stream-Based

Fundamentally, at its core, the Internet is a packet-based network. Thus everything that flows through it has to ultimately be split into discrete packets, of data and headers, which may vary in size. Often, application programmers prefer to deal with stream-based data transfer mechanisms, which is an essentially open-ended form of communication in that the amount of data that can be transmitted is non-finite. Stream-based mechanisms often have a push-mechanism too, which is a way to ‘hurry’ some section of data along to its destination. The UDP protocol is an example of a packet-based protocol. TCP is an example of a stream-based protocol. Knowing this will help you determine which type of network transport layer is suitable for your application

2.4 Ethernet

Ethernet is a shared-bus multiple-access with collision-detection communication scheme. Ethernet is defined broadly enough that it supports several physical media types. The media type used in this implementation is 10BaseT, commonly known as twisted-pair.

With Ethernet, it is important to understand the difference between a logical node address and a physical node address. A physical node address in Ethernet is a guaranteed unique 48-bit number that is assigned to every Ethernet terminal interface manufactured. A logical node address is the address that networking protocols use when directing packets. It allows a many-to-one mapping of physical addresses to a logical address. This essentially means that Ethernet controllers can carry packets between different physical networks and still deliver them to only one final destination. The Internet world uses “IP Address” for its logical addressing. This is a 32-bit number (commonly expressed as “w.x.y.z”).

IP addresses are used to segment the internet into big groups of LANs each containing a lot of network devices. A unique IP address for each computer on the internet would be impossible because of the huge volumes and difficulty to control assignments. Therefore, you may encounter two PCs on different physical LANs that have the same IP address. The moment communication is required from LAN to LAN over the internet, the use is made of gateway or router PCs that will act on behalf of a network device connected on a LAN.

Ethernet is a best-attempt delivery network. In other words, the network will try its best to deliver a packet, once sent, to its destination. It is not a guaranteed delivery network, which means that additional software is required to ensure a reliable packet delivery or stream transport service. Furthermore, Ethernet does not guarantee in-order delivery of packets, once sent.

Ethernet is a packet-switched network. The term ‘frame’ is used to refer to a packet of data in Ethernet. An Ethernet frame must be at least 64 bytes in length, and no more than 1518 bytes.

To ensure data integrity (i.e. to provide error-checking, but not error-correction), Ethernet frames are constructed with a 32-bit CRC. Ethernet interfaces, when receiving Ethernet frames, are required to check the CRC field and discard (usually) the frame should the CRC not match.

2.5 Ethernet SX-Stack

One can think of TCP/IP software as being built up of four levels of abstraction as shown in Figure 2-1.

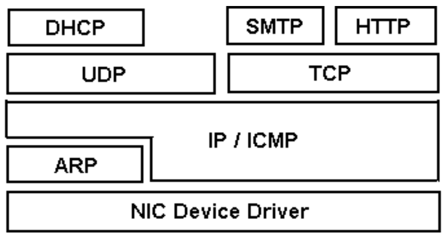


Figure 2-1. Ethernet SX-Stack

At the bottom, the *Physical* layer, is the software that is specific to the physical media being used to transport the IP packets (NIC Device Driver).

Above this is the *Internet* layer, which implements the protocols to enable packets to be routed from one node to another on the Internet, as well as communications test and diagnostic services (ARP, IP. ICMP).

The next layer is the *Transport* protocols layer. This layer is responsible for end-to-end communication between application programs. The protocols implemented in this layer allow multiple connections to be established by multiple application programs. These protocols can regulate the flow of data, ensure data integrity, establish and close-down connections (UDP, TCP).

Up to this point, all the layers discussed are generally implemented by the Operating System running on the host. This is so that software developers need not reinvent the wheel when they need to communicate information across a network. Along these lines, the stack provides for the same infrastructure for thin, resource-scarce, embedded devices.

Finally, capping the stack, is the *Applications* layer. Software at this level does not need to worry about the mechanics of how information is transported from one machine on the Internet to another. At the same time, applications are guaranteed that whatever information they want to communicate will be transported over LANs and WANs, including the globe-spanning Internet. Application layer software may implement standard services (e.g. DNS, FTP, SMTP, HTTP, etc.), or they may be proprietary customized applications whose audience is restricted to the same applications running on other hosts.

2.6 ARP

The Address Resolution Protocol (ARP) allows the dynamic mapping of logical addresses to physical addresses.

As discussed in section 2.3, Ethernet physical interface supports a unique 48-bit address. However, for efficient routing of packets, the IP networks use a 32-bit logical address. This is a hierarchical addressing scheme as it allows individual nodes to be part of a subnet, which in turn can be subsumed into a larger subnet. The Ethernet specification is independent of the logical addressing scheme used. The delivery of all Ethernet frames is specified exclusively by Ethernet physical addresses.

ARP, hence, is the bridge that maps IP addresses to physical addresses. It provides the mechanism by which a node can transmit an IP packet to a destination, whose IP address is all that is known, over an Ethernet network.

2.7 IP Datagrams

A datagram is a packet of actual data combined with a packet header. *All* Internet communication travels in the form of IP datagrams. They are the common mail pouches of the Internet world. Hence it is important to understand the role of IP, which is described next.

Figure 2-2 shows a few equivalent IP datagrams or internet datagrams.

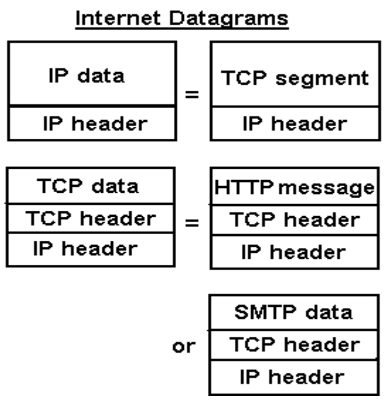


Figure 2-2. IP Datagrams

2.8 IP/ICMP

IP stands for Internet Protocol. It is the common denominator of the entire TCP/IP suite of protocols. The IP protocol assumes an unreliable, best-effort, connectionless packet delivery system, which is exactly what Ethernet provides.

Let us describe each of the above terms in more detail.

By *unreliable*, it is meant that a sent packet is not guaranteed to reach its destination, even if the destination is reachable. There are several reasons why a packet may not reach its destination:

1. It was corrupted en-route and discarded at some point,
2. The destination node is not connected to the network.
3. The network routing tables were incorrect thus the network does not know how to deliver the packet.
4. Network congestion has forced a router somewhere en-route to drop the packet.

Best-effort, means that the network always tries its best, given what it knows, to deliver the packet.

Connectionless, implies that there is no handshaking involved between the sender and the recipient(s) prior to the packet being sent. Senders are at liberty to send whenever they wish, subject of course to their getting time on the Ethernet, which is a shared-bus network.

P*acket*, implies that all data that traverses an IP network have to be of finite size, and accompanied by a header. IP packets are allowed to be fragmented by the network as the packet traverses the network. However, the SX-Stack does not accept fragmented packets, since it does not do packet reconstruction.

2.9 DHCP

Dynamic Host Configuration Protocol (DHCP) allows the use of the client-server paradigm for host machines to dynamically bootstrap, as well as configure, themselves when placed in a networked environment. Often used for its ability to dynamically assign scarce IP addresses to networked devices upon startup, DHCP is however not restricted to that, as it has the capability to configure server address, router address and any number of vendor-specific configuration parameters.

For DHCP to work, there must be both a client (such as the iSX) as well as a server(s). The client does not need to know the IP address of the server, as it will discover it for itself. The server(s) will offer an IP address to the client on request. Some servers allocate limited-time leases on IP addresses, which means the client will need to periodically renew its lease on its IP address.

2.10 UDP

The User Datagram Protocol (UDP) allows applications to send packets of data across the network to each other. It can also be used to broadcast data to multiple nodes. UDP is a *unreliable*, *connectionless* delivery service (see section 2.7).

UDP packets may be lost in the network and may arrive out of order with other sent UDP packets. However, UDP packets are checked for data integrity; so if they are received, it is safe to assume the data contained within is good.

To send a UDP packet to a destination application, the source application must know both the IP address of the destination host, as well as the port number of the destination application.

2.11 TCP

The Transmission Control Protocol (TCP) has become the protocol of choice for many applications because it allows for *connection-based*, *reliable* transport of data across an unreliable network. It does this at the cost of throughput, latency and bandwidth utilization.

Like UDP, TCP achieves multiplexing through the use of source and destination ports. A TCP ‘connection’ is defined to be a communication channel established between two ‘end-points’. An ‘end-point’ is defined to be a unique combination of both an IP address and a port number.

Before a single byte of data can be transmitted using TCP, a connection has to be set up between end-points. This is typically done using a three-way handshake involving the transmission of three TCP segments. The TCP protocol takes care of data sequencing, re-transmissions and data-checking. It is important to understand that HTTP messages and SMTP data are sent between applications on remote network devices by using TCP as the transport service. HTTP messages and SMTP data sent using TCP are nothing more than the data bytes in a TCP segment.

It is important to understand that TCP is based on a client-server paradigm. This does not mean that one transmits exclusively while the other receives only. The ‘client’ is the application which *initiates* the connection with the ‘server’. To achieve that, the TCP protocol allows server applications to do a *passive-open*, which means listening on that port and accepting connections to it, while allowing client applications to do an *active-open*, which initiates the connection. This current Ubicom TCP/IP stack software supports two TCP connections. Either connection may be established through a *passive-open* or an *active-open.*

2.12 HTTP

The Hypertext Transfer Protocol is used to transfer images and text to web browsers. A web browser contacts a web server such as implemented on the Ethernet demo board. The web browser requests resources or submits data by using a certain defined method of information transfer. To retrieve resources from the web server the method is “get”. To submit data to a program on a web server the method is “post” or “put”. In the case of the “get” method, the web server will send the requested resource such as a html file via the open TCP connection to the web browser. The “post” method is used to submit data to a program on a web server such as found on input fields on some web pages. The data can be used to control devices or more commonly, to generate html files on the fly to send back to the web browser such as found in internet “search engines” sending back the results of a search.

2.13 SMTP

The Simple Mail Transfer Protocol is used to send email to a SMTP server. The sending program is called the send client. A send client sends email to a recipient via a remote SMTP server who receives the email message. A recipient will have to retrieve email from a mail server via the Post Office Protocol. The SMTP protocol is a command response type protocol and uses TCP as its data carrier or transport service.

3.0 The Ethernet Demo Board

3.0.1 Hardware Description

The Ethernet demo board can be connected to a 10/100BaseT Ethernet connection. To describe the board's Ethernet capability, it simply consists of two controllers and a transformer, T1. The Ethernet controller, U5, handles Ethernet communication over the physical media while the Ubicom controller, U2, controls the setup and commanding of the Ethernet controller. The Ubicom controller runs the only program code named the Ubicom Ethernet SX-Stack software on the board. The stack software consists of a driver to interface to the Ethernet controller and of various network protocols used to communicate to remote network devices over the Ethernet. All other components or sections of circuitry are either supporting the basic Ethernet components or implement additional functionality on the board. A more detailed description of the hardware on the demo board follows:

1. The Ubicom controller, U2, X1

The Ubicom controller operates at a frequency of 50MHz implemented with an external resonator, X1. The controller executes one instruction per clock cycle, thus one every 20ns. The controller has 4096 of electrically erasable FLASH program memory and 262 bytes of SRAM data memory.

2. The Ethernet physical layer controller, U5, T1, X2

The board is to be connected to a 10/100BaseT Ethernet hub or connection through J2. The Ethernet electrical signalling and control interface to the Ubicom controller is implemented by the Ethernet Controller, U5. The Ethernet controller interfaces to the Ubicom controller through ports B and C on the Ubicom controller. Port C is set up as a bidirectional databus to transfer 8-bit data to and from the Ethernet controller. Port B is used as address lines. SA0-4, read and write signals complete the interface to the Ethernet controller. Since the Realtek Ethernet controller was developed for use on ISA bus PC type plug-in cards, the addressing and control interface to the Ethernet controller is according to the ISA bus standard.

3. Memory for storing web pages, U3

The web server on the board is implemented by storing resources consisting of html files and graphics data on the serial EEPROM memory, U3. The Ubicom controller will get a request for a certain resource from a remote web browser on the Ethernet and will retrieve the applicable html file or graphics content from the serial memory via a serial I^2C bus implemented on pins RA5 and RA6 of the Ubicom controller. The data is then packaged into TCP segments and sent over the Ethernet to the requesting web-browser.

4. RS232 port for downloading new web content to the memory, U1, JP1

The RS232 port on the board is used to download new web content to the serial memory. U3 is a voltage level converter IC that is used to change voltage levels from TLL to RS232 and vice versa when communicating with a RS232 port such as the serial COM port on a PC.

5. The +5V power supply, U4

The power supply on the board is implemented with a voltage regulator, U4 and associated components. It supplies a steady +5VDC to the components on the board.

6. A user expansion IO port, JP4

The user port may be used for interfacing to a daughter board and each IO pin may be set up to be either an output or an input pin.

7. A Java Virtual Machine port, JP5, JP6

Contact sales@Ubicom.com for further information.

8. A Sigma Delta Analog to Digital Conversion circuit, R1,7,10, C7

The Sigma Delta conversion technique is commonly used to implement low cost analog to digital converters. The circuit in this case basically works by alternating the voltage between high and low on pin RE7 of the Ubicom controller. This either charges or discharges the capacitor C7. By sensing the capacitor's voltage level on pin RE6 of the Ubicom controller through utilizing the inherent voltage trip level of the input stage on this pin, a comparator is implemented. The output of the comparator is used to increment a software result counter whenever the comparator swings into the low state. Another software counter is also incremented on each swing of the comparator to count the number of "conversions" up to a certain number after which the value in the result counter is valid. A new conversion starts by clearing the counters. The voltage level on C7 is affected by an external influence which is the voltage divider R11 and R1 feeding the capacitor through R10. The resistance of thermistor R1 changes with temperature and will thus influence the initial voltage level from which C7 is charged or discharged. This means that the time it takes for the comparator to swing over from the one to the other state is either increased or decreased and the result counter will be counting higher or lower according to the temperature of the thermistor. This effectively makes an analog to digital converter of which the result counter value is a representation of the temperature of R1. The demo board software only displays the value of the result counter on a dynamic web page in the web server demo and does not attempt to translate this value to a real temperature value.

9. Ubicom ISP port for programming the Ubicom controller and debugging code, JP2

Program and debug tools currently from Parallax and Advanced TransData can be plugged into JP2. A single jumper connector, JP3 is also included to isolate the resonator, X1, from the ISP port during programming and debugging.

4.0 Demo Programs

4.1 The Web Server

The web server is implemented in software by exclusively using the TCP API which is explained in 6.0 “Using the TCP API”. The Ubicom Ethernet stack software supports two simultaneous TCP connections to the Ethernet SX-Stack demo board. A TCP connection to a remote server can be either initiated from the demo board or can be established from the remote server initiating the connection. A web server is essentially a passive device that waits for requests from remote web-browsers. A remote browser initiates a TCP connection to the demo board. Once the demo board receives a request to establish a TCP connection, it creates a socket that consists of the remote TCP port number and the remote IP address. The TCP connection manager function on the TCP/IP stack software of the demo board uses this socket to recognize subsequent incoming datagrams from the remote server running the web-browser and directs the TCP data from those packets to the application software using the TCP connection bound to that socket. This means that another application using the other TCP connection may be receiving data from a whole different remote server and only the data destined for it will be delivered to it.

The remote web-browser requests resources from the demo board web server by asking for it in the very first text line in the HTTP message. The HTTP message is the payload of the TCP segment meaning that the data section of the TCP segment is the whole HTTP message. The web server supports two types of requests, also referred to as methods:

1. get - the get method request text line looks something like this:

GET /index.htm

The resource stored in the EEPROM memory under index.htm will then be retrieved from the memory, packaged into TCP segments and sent back to the requesting web-browser by using the information in the socket bound to the TCP connection for the web server.

2. post - the post method request text line looks something like this:

POST /postctrl.htm

Normally, a html file associated with the post method will contain text input forms and buttons that may be clicked to send control values to a web server. This means that before a post of text or control values can take place, a web-browser will first get the html file to display the page with the input controls. A HTTP message containing the post method line will be sent to the web server once the user presses a submit button on the web page with the controls. A variable that is known to the web server and is used as a control variable to control something like a LED on the board will be sent along with the post request. Web-browsers use different ways of doing a post request to a web server. Some like Internet Explorer will send only one HTTP message containing the post text line and the variables. Web-browsers like Netscape Navigator sends two seperate messages, each in a seperate TCP segment. One contains the post method text line and the other contains the variables. The Web server demo on the demo board distinguishes between posts from these two types of browsers and will still find the control variables in the first or following HTTP message.

Once the web server sends a resource, may it be a html file’s text or other graphics data, the web-browser will close the open TCP connection and the TCP connection manager on the demo board will delete the socket that was bound to that open TCP connection. Subsequent requests from a web server will result in a new socket being created. The reason why a open TCP connection should be closed immediately after a successful data transfer is to conserve memory that is taken up by storing sockets as well as to prevent spurious packets on the internet to cause false data transfers.

The resources on the serial EEPROM memory is stored as pure text or data and is retrieved by a hash value that is stored in a lookup table in the memory ahead of the actual content. Essentially, when the data was stored on the EEPROM, every resource’s data-bytes were added together and clipped to a one byte value which is a handle or hash that identifies the resource. The web server application searches for the resource name in the get or post text line of the HTML message and adds the byte values of the resource name together and this value corresponds to the handle or hash value which is stored in the serial memory when the resources were transferred to the memory via the RS232 serial port. The lookup table simply contains pointers into the memory indicating the starting addresses and number of bytes of each resource.

4.1.1 Dynamic Web Pages

The serial EEPROM memory contains three dynamic web pages:

1. temperature.htm

2. resources.htm

3. postctrl.htm

A dynamic web page contains content that changes according to some external value or parameter. A simple example is a web page that shows the temperature of a sensor on the board. The temperature may be 50 degrees now but may be 60 when the page is retrieved again. The Ethernet SX-Stack demo board implements dynamic web pages in a very easy to understand manner. Dynamic web pages are stored in the serial EEPROM with a special sequence of data bytes or characters embedded at the exact location where the dynamic content needs to be inserted. Look at the temperature.htm file below:

```
<HTML>

<HEAD>

<TITLE> Board Temperature Monitor </TITLE>

</HEAD>

<BODY BGCOLOR=#404040 text=#80FF00>
```

```
<H2> Real-time Board Temperature Monitor </H2>
<br>
<br>
<br>
<center>Temperature is now: <font size=+2 color=#FF99FF> ðñò </font></center>
<P>
<font color=#FFFFFF>Hit 'F5' or click the 'Refresh' button on your browser to see the current temperature</font>
</BODY>
</HTML>
```

The three characters indicated with bold text are the "magic key" that is used to indicate the location of the dynamic content in this web page. In this case a value from an analog to digital converter needs to be inserted here when the text of this htm file or resource is being packaged into a TCP segment ready to be sent to the requesting web-browser. The web server software will look for these three characters while retrieving the data bytes from the serial memory and will replace them with the value from the analog to digital converter. It is important to note that a magic key needs to be non-printable characters that could never be mistaken for real text repeated elsewhere in the htm file.

The resources.htm file below also contains dynamic content which is the value of a page visited counter. The magic key is the same as for the temperature.htm page. The web server software distinguishes which dynamic content belongs to which dynamic webpages htm file by also looking at the hash value of the requested resource. In this case the requested resource is /resources.htm

```
<HTML>
<HEAD>
<TITLE> iSX Resource Usage </TITLE>
</HEAD>
<BODY BGCOLOR=#404040 text=#FF8000>
<H2>Resource Usage:</H2>
<CENTER>
<TABLE border=2 bgcolor=#000000 width="50%">
<TR>
<TD width="30%">ROM</TD>
<TD align="center">2527</TD>
</TR>
<TR>
<TD width="30%">RAM</TD>
<TD align="center">98</TD>
</TR>
</TABLE>
</CENTER>
<P>
<P>
This pages has been accessed <font size=+2 color=#FF99FF> ðñò </font> times.
</BODY>
</HTML>
```

Finally, the postctrl.htm page is shown below. Only one of the characters for the magic key is used here to indicate to a requesting web browser which graphic gif picture to get once the LED has been controlled using the http post method. In this case the web server software will always replace the magic key character with either a 'n' or a 'f' after checking the status of the LED. If the LED is switched on via the http post method, it will send back this page with a 'n' giving ledon.gif. The web page will thus display a bright LED picture indicating that the http post method request to switch the LED on was successful.

```
<HTML>
<HEAD><TITLE>Ubicom control via HTML POST</TITLE></HEAD>
<BODY BGCOLOR=#404040 text=#80FF00>
<H2> Remote control demo </H2>
<P>
<FONT color=#FFFFFF>Control messages can be sent to the SX via the HTTP POST method. In this way devices such as air conditioners can be remotely switched on and off from a standard web-browser.
<BR><BR>Select the LED on or off radio button and click on Submit to control the LED on the demo board.<BR><BR>
LED<BR>
<left><img SRC="images/ledoð.gif">
<form action="http://10.1.1.20/postctrl.htm" method="POST">
<input type="radio" name="led" value="1"> On<br>
<input type="radio" name="led" value="0" CHECKED> Off<br>
<input type="radio" name="led" value="t"> Toggle<br><br>
<input type="submit" value="Submit">
</form>
</FONT>
</BODY>
</HTML>
```

4.2 The Send Email Client

The Send Email client is also implemented by using the TCP API functions listed in 6.0 “Using the TCP API”. The web server uses TCP connection 2 while connection 1 is used by the email client. The email client differs from the web server in the sense that it is not passively waiting for a request to establish a TCP connection with a remote server in order to transfer data in TCP segments. The email client initiates the opening of a TCP connection with a remote server that has a listening SMTP server. Listening refers to a half socket created on the remote server that contains only the remote SMTP server’s TCP port number which is universally assigned as 25. Once a TCP connection is established between the demo board’s email client and the remote SMTP server, the SMTP server will fill in the socket with the demo board’s IP address and TCP port number. On the demo board the socket is already fully defined before the email client attempts to establish a TCP connection with the remote SMTP server. The socket on the demo board contains the IP address of the remote SMTP server and the port number of the local TCP connection. As before, incoming TCP data from the remote SMTP server will be delivered to the send email client application which is using TCP connection 1.

To send an email the user presses a button on the demo board. This sets the SMTP send state machine into motion. The SMTP protocol is a command-response type protocol. The sender, in this case the demo board, sends a SMTP command in a TCP segment to the remote SMTP server which will respond with a response code and the applicable data bytes in a TCP segment. Even though the email client on the demo board is the commander during the data transfers, the first SMTP data is sent from the remote SMTP server to the email client directly after a TCP connection has been established. This very first SMTP data serves as an identifier which contains something like the following text:

220 Eserv/2.92 ESMTP Server Ready.

A response code is always sent as the first three characters. In this case 220 means “service ready”. The email client will now identify itself by sending a SMTP command like this:

HELO sx

The SMTP server may respond with an error code or if ready and recognizing “sx” as a user with an email account on the SMTP server, will respond with:

250 Hello 10.1.1.20

The 250 response code means “Requested mail action ok or completed”.

To send an email the client will now send a packet containing:

MAIL FROM: <sx>

The sender’s email address name is inserted between braces.

If the sender is valid on the SMTP server (you need an email account on a SMTP server to be able to send email) it will respond with:

250 OK

The send email client will now send:

RCPT TO: <joe@demo.sx>

This indicates the recipient of the mail. The SMTP server will accept this command if joe has an email account there or if it can relay the email to another SMTP server that will deliver the email to joe.

The SMTP server will respond with:

250 OK

This indicates to the email client that the SMTP server will accept the mail.

The email client will now send:

DATA

to indicate that the email data is to be sent.

The SMTP server can now respond with a text string which contains formatting instructions or whatever may be needed for a custom implementation:

354 send the mail data, end with .

This indicates to the send email client it can send the mail data but must end it with a definite <CRLF>.<CRLF> since TCP packets are usually padded with extra data bytes that may be confused for mail data.

The email client will now send the mail data as follows in a TCP segment:

From: sx

To: Joe

Subject: Button Pressed!

Button SW2 Pressed!

The SMTP server may indicate a syntax error or some other error code if the mail format is wrong or will respond with the following if the mail data is acceptable:

250 OK message accepted for delivery

The send email client will now indicate to the SMTP server it has nothing more to do with:

QUIT

The SMTP server will respond with the following message if no other errors are encountered:

221 Service closing transmission channel

Now, the SMTP server will initiate the closing of the TCP connection established between the demo board and itself. This change in the command / response roles in the SMTP protocol is probably to give a large degree of confidence to the email send client program that the SMTP server was indeed alive during the whole mail transfer up to the very end when the transmission channel was closed.

5.0 The TCP/IP Stack Software

The TCP/IP stack software consists of three API's which a programmer may use:

1. The TCP API

2. The UDP API

3. The SMTP API

Since there is only one Ethernet controller on the board, U5, the board is set up to have one IP address and one Ethernet address (also referred to as the MAC address). Every Ethernet controller IC has a unique MAC address since electrical signalling is done by these devices.

The following variables are to be set up before the stack can be used:

1. myIP3-0 - the board's IP address

Typically we could have:

myIP3 = 10

myIP2 = 1

myIP1 = 1

myIP0 = 20

giving the demo board an IP address of 10.1.1.20

2. SX_ETH_ADDR0-5 - the Ethernet controller, U5's MAC address

This is the 48-bit Ethernet address of Media Access Controller address which will be used by the Ethernet controller when sending or listening for Ethernet frames. The demo board uses a MAC address that is to be used only temporary. To change this you need to assign a new MAC address.

3. INT_PERIOD

This is a value from 0 to 255 that is subtracted from a free-running hardware counter that causes an interrupt when it overflows. An interrupt routine is called in response to the overflowing counter and currently the stack uses the interrupt routine to advance various software counters as well as to flash a LED and read the value of the temperature sensor. The value in INT_PERIOD is subtracted from the counter's value when the return from interrupt instruction is executed to end the interrupt routine. This ensures a known interrupt frequency with no jitter. The frequency at which the interrupt routine is called can easily be determined by dividing the Ubicom controller, U2's clock frequency by the number in INT_PERIOD. Since the hardware timer advances once every cycle the calculation is a simple divide.

It is important to note that the various state machines used by the stack (TCP/UDP/SMTP) execute as fast as possible. They are not implemented with virtual peripheral type code such as other Ubicom virtual peripheral software. Virtual peripheral software always run inside an interrupt routine and are thus executed periodically. The stack's state machines do not run inside an interrupt routine. They run in mainline loop code such as the following code will explain:

```
main:

; state machines code

jmp :main
```

6.0 Using the TCP API

The TCP API is a collection of functions which the TCP state machine calls and variables that are used by the state machine. To use the API one must think of how a typical interrupt routine in the C programming language is used by a programmer. A C programmer that wants to execute code inside an interrupt routine function, let's say to advance a counter or to check if a switch is pressed, doesn't need to call the interrupt routine. The interrupt routine is automatically called in response to an event or it may be periodically called in response to a hardware timer's value. The TCP state machine calls certain TCP API functions when events occur such as incoming data that is available. It also calls other TCP API functions periodically like to check if an application such as the send email client wants to open a TCP connection. As a programmer using the TCP API you can never call an API function directly but would rather simply insert your code in the space where the function is called. Your code will be executed once the applicable event occurs or it will be executed periodically. Before we look at an example of how to use the TCP API, note the following:

1. There are two TCP connections available on this TCP/IP stack. Two applications cannot share a TCP connection since a connection is defined by a socket that describes only one remote TCP connection.

2. TCP connection 1 can only be used to do an active open. TCP connection 2 can only be used to do a passive open.

3. Your application software may either open a TCP connection to a remote server referred to as an "active open" by initiating the connection, or your software may do a "passive open" by waiting for a remote server to initiate the connection. The web server is a case of a "passive open" and the send email client uses an "active open".

4. For an active open you need to specify the IP address of the remote server you wish to establish a connection with and the TCP port number of the application you wish to connect with.

5. For a passive open you need to specify the local TCP port your application will use. The TCP connection manager will "listen" on this port for incoming TCP segments from remote servers wishing to establish a connection with your application software.

A short discussion on how two TCP connections are handled by the TCP API is in order now. To prevent an overrun of the receive buffer of the Ethernet controller possible resulting in dropped frames, the main loop of the TCP/IP stack software gives precedence to the reception of internet datagrams as opposed to the transmission of internet datagrams. This means that the software will continue handling incoming internet datagrams of whatever protocols until there are no more datagrams in the receive buffer of the Ethernet controller. Once the receive buffer is empty it will call the necessary functions to check if an application wants to transmit something. In the case of the TCP state machine, your application can only transmit TCP data once a TCP connection is established or open. Incoming TCP data is delivered to the correct application with the use of a bit or flag called TCP_SOCK. The TCP state machine sets this flag when an incoming internet datagram is for the TCP application using TCP connection 2 and clears it when the incoming datagram is for the application using connection 1. At the top of every TCP API function (except TCPApp1Init and TCPApp2Init), there is a piece of code that checks this flag and will jump to a label under either the space where TCP connection 1 application code goes or where TCP connection 2 application code goes. If your TCP application used TCPApp1Init you must insert your code in the TCP API functions under the label indicated for TCP connection 1 and vice versa.

Now for the transmission of TCP data. Each TCP connection has the same priority to transmit and therefore each pass through the TCP API functions called by the TCP state machine is alternated between TCP connection 1 and 2. At the top of every TCP API function (except TCPApp1Init and TCPApp2Init), there is a piece of code that checks a flag called TCP_TXSEMA and will jump to a label under either the space where TCP connection 1 application code goes or where TCP connection 2 application code goes. If your TCP application used TCPApp1Init you must insert your code in the TCP API functions under the label indicated for TCP connection 1 and vice versa. The TCP_TXSEMA flag is alternated inside the main loop code of the TCP/IP stack software.

6.1 Establishing a TCP connection

A description of the logical sequence of steps you will need to take to establish a TCP connection is as follows:

1. You will need to have set up the stack variables such as described under 5.0. They are the board's IP address and the MAC address for the Ethernet controller.

2. There are two TCP API functions available, one for each TCP connection. They are:

TCPApp1Init - Only to be used for doing an active open.

TCPApp2Init - Only to be used for doing a passive open.

These functions are called continuously as long as a TCP connection state is closed. In these two functions you will insert "trigger code" that will start a passive open or an active open in response to some trigger. You will also define the TCP sockets here. An example of trigger code may be to continuously check whether a switch is pressed and once pressed, have an active open be done to send an email. Passive open code will more than often be just the definition of the TCP socket since the trigger to open the TCP connection will be from the remote TCP. An example of doing a passive open in TCPApp2Init is shown below:

```
;
*****************************************************************
**************
_TCPApp2Init
; Called repeatedly as long as TCP connection2 state is
closed
; [TCP API Function]
; INPUT:  none
; OUTPUT: none
;
*****************************************************************
**************
_bank TCB2_BANK
mov tcb2LocalPortLSB, #HTTP_PORT_LSB
mov tcb2LocalPortMSB, #HTTP_PORT_MSB

bank HTTP_BANK
clr httpParseState
clr httpURIHash

; indicate tcp2 connection
setb flags2.TCP_SOCK
jmp @TCPAppPassiveOpen
retp
```

Looking at the code excerpt we see the local TCP port that we will be listening on being set up by tcb2LocalPortLSB and tcb2LocalPortMSB. In this case a value declared elsewhere of 80 is used which is the worldwide standard TCP port number a web server listens on for requests from web browsers. Two http variables are cleared which is needed to initialize the http protocol. The last section of the code is where a flag or bit is set in the flags2 variable to indicate to the TCP state machine we are using TCP connection 2. Finally a jump is made to a function called TCPAppPassiveOpen. This function will start the TCP state machine by putting it into the “Listening” state. Note that since the TCP connection state is not closed now any more but in “listen”, TCPApp2Init will not be called again from the main loop. Only when another TCP API function named TCPAppClose is called, will the TCP connection state be reset to closed and will TCPAppPassiveOpen be called again. When we use trigger code however, we would jump over the call to TCPAppPassiveOpen and will only allow this function to be called once the trigger is made eg. a “web server enable” push button on a board.

Now we will look at how to use the TCPApp1Init API function to do an active open.

```
;
*****************************************************************
**************
_TCPApp1Init
; Called repeatedly as long as TCP connection1 state is
closed
; [TCP API Function]
; INPUT:  none
; OUTPUT: none
;
*****************************************************************
**************
; trigger code
test switch
sz
retp ; exit, switch not pressed

_bank SMTP_BANK
mov w, #SMTP_CONNECT ; start SMTP state machine
to connect
mov smtpState, w

; set up local & remote ports for tcp1 connection
_bank TCB1_BANK
mov tcb1LocalPortLSB, #100
mov tcb1LocalPortMSB, #100
mov tcb1RemotePortLSB, #SMTP_PORT_LSB
mov tcb1RemotePortMSB, #SMTP_PORT_MSB

; fill in remote IP to connect with in tcp socket
_bank TCPSOCKET_BANK
mov sock1RemoteIP3,#SMTP_SERVER_IP3
mov sock1RemoteIP2,#SMTP_SERVER_IP2
mov sock1RemoteIP1,#SMTP_SERVER_IP1
mov sock1RemoteIP0,#SMTP_SERVER_IP0

; indicate tcp1 connection
clrb flags2.TCP_SOCK
jmp @TCPAppActiveOpen; open a tcp connection on
socket1
retp
```

The trigger code at the top of the function checks the state of the switch variable and will keep exiting the function until the switch is pressed. The SMTP code inserted here is used to start the SMTP state machine once the switch is pressed. Note the setup of the TCP socket for this connection. The local TCP port to be used by our email application is set up with tcb1LocalPortLSB and tcb1LocalPortMSB. The remote TCP port our email client must contact is set up by tcb1RemotePortLSB and tcb1RemotePortMSB. In this case a value of 25 defined elsewhere is used which is the worldwide standard port number SMTP servers use to listen for incoming connections. Finally, the remote SMTP server's IP address we wish to contact is set by sock1RemoteIP3-0. Note the clearing of the flags2 bit to indicate the use of TCP connection 1 to the TCP state machine and then the final jump to a function named TCPAppActiveOpen which will put the TCP state machine into motion to connect to the remote SMTP server. Also note that again, TCPApp1Init will not be called again from the main loop since the TCP connection state for connection 1 is now transitioning from closed to other states. Only when the connection is closed will TCPApp1Init be called again and the trigger code will again block the opening of a TCP connection until the switch is pressed again.

Note that only once a TCP connection is established can actual data be transferred. During the opening process, not a single byte of TCP data can be transmitted or received by your application.

6.2 Handling incoming TCP data and sending TCP data

Once the TCP connection is open there may be one of two situations:

1. TCP data may be received from the remote server we are connected to.

2. We may send TCP data to the remote server we are connected to.

It is important to realize again now that there is a difference between a TCP segment and TCP data. A TCP segment contains a header and data. On incoming TCP segments, the TCP state machine strips the header from the TCP segment and only delivers the data to your application. On outgoing TCP segments, your application only hands the data to the TCP state machine and once done, the state machine will append a header to the data and send the TCP segment.

An incoming TCP segment will cause a receive event so that the TCP state machine will call the following TCP API functions in the following order:

1. TCPAppRxBytes - This function is called only once after a TCP segment was received. A variable called tcpAppRxBytesMSB, LSB contains the value of the number of data bytes the TCP segment contains. This value may be checked against expected bytes or to prepare a counter.

2. TCPAppRxData - This function will be called as many times as the value in tcpAppRxBytesMSB, LSB represents. Note that the data bytes are passed here through the working register. It is here where you could parse or store the incoming data bytes.

3. TCPAppRxDone - This function is called only once after the last call to TCPAppRxData. This is handy since you do not need to set up a counter in TCPAppRxData to know when the last byte will be delivered. Your application may use this function as an acknowledgement that data has been successfully received. Also, if you need to make a decision after the reception of data you will insert your code here.

As previously said, the main loop of the TCP/IP stack software gives precedence to the reception of internet datagrams as opposed to the transmission of internet datagrams. This means that the software will continue handling incoming internet datagrams of whatever protocol until there are no more in the receive buffer of the Ethernet controller. Once the receive buffer is empty it will call the necessary functions to check if an application wants to transmit something. In the case of the TCP state machine, your application can only transmit data once a TCP connection is established or open. Once the connection is open, the TCP state machine will call the following TCP API functions in the following order:

1. TCPAppTxBytes - This function is called only once before the rest of the TCP API functions shown below. In this function your application must indicate how many data bytes it wants to send. This is done by loading the variables named tcpXUnAckMSB, LSB with a value indicating the number of bytes to be sent. Because there are two TCP connections, each internet datagram containing a TCP segment is stored in a separate transmit buffer of the Ethernet controller. This is needed so that TCP segments may be re-sent if they were not acknowledged by the remote TCP on the remote server we are connected with. The 'X' in the tcpXUnAck variable needs to be replaced with 1 or 2 depending on which TCP connection you are using.

2. TCPAppTxData - This function will be called as many times as the value in tcpXUnAckMSB, LSB represents. Each time the function exits it loads the data byte in the working register into the TCP segment being constructed. Here you pass the data that you wish to transmit to the TCP state machine.

3. TCPAppTxDone - This function is called only once after the last call to TCPAppTxData. Your application can use this as an acknowledgement that the data has been sent. Also, if you need to make a decision after the sending of data you would insert your code here. Note that this function is not called to indicate that the data has been successfully delivered to the remote server. It doesn't even mean that it was acknowledged yet by the remote TCP. The outgoing internet datagram containing your data will be loaded into the transmit buffer of the Ethernet controller and will be re-transmitted every approximately every 10 seconds until an acknowledge is received from the remote TCP. There is however is timeout implemented on the stack which will cause the TCP connection to close and reset should there be no reply from a remote TCP in about 30 seconds after the first transmission was made.

7.0 Using the UDP API

Since UDP is a connectionless protocol, UDP packets are sent in a fire and forget fashion as opposed to TCP which waits for an acknowledgment that a TCP segment has indeed been received by a remote host. Note that this acknowledgement does not mean that the remote application received the data in the TCP segment. The TCP/IP stack uses protocol state machines only for connection-oriented protocols. The state machine would typically start out in a closed state and transition through various states to an open state once a connection is established. It could stay in a specific state until acknowledges are received or until connections are open or closed. Outgoing UDP packets are constructed and sent immediately without waiting for a response from the destination. Incoming UDP packets are handled by function calls to UDP API functions in which you can insert your application code.

To use the UDP API you may commonly want to do the following:

1. Send data to a remote server

2. Receive data from a remote host

7.1 Listening for UDP packets

A UDP API function named UDPAppInit is called once and never again when the TCP/IP stack software starts executing. You have to specify the local UDP port number which to listen on for incoming UDP packets. The function is shown below:

```
; ******************************************************************************
UDPAppInit
; Application UDP Initialization code (Example)
; This function is called automatically once by the stack During startup
; [UDP API Function]
; INPUT:  none
; OUTPUT: none
; ******************************************************************************
_bank UDP_BANK
mov udpRxDestPortMSB, #UDP_RX_DEST_MSB
mov udpRxDestPortLSB, #UDP_RX_DEST_LSB
retp
```

7.2 Handling incoming UDP data and sending UDP data

Whenever a UDP packet is received on the listening port specified in UDPAppInit, the TCP/IP stack software calls the following functions:

1. UDPAppProcPktIn - This function is called only once when a UDP packet is received. The stack also receives UDP packets that were sent in broadcast mode, but those packets are handled by other functions of the stack that are not part of the UDP API. Therefore, UDPAppProcPktIn is not called by the stack when a broadcast UDP packet was received. Broadcast UDP packets are commonly used by the DHCP protocol on this stack.

As opposed to the TCP state machine that calls TCPAppRxData as many times as there are data bytes in the received segment to pass the data to you application, you must call NICReadAgain which will each time it is called, retrieve a data byte from the received UDP packet. The databyte is delivered in the working register. The following variables also contains the value of the number of databytes in the received UDP packet:udpRxDataLenMSB, LSB. The remote UDP port from which the packet came is indicated in the value of a variable called udpRxSrcPortMSB, LSB.

You can respond immediately to a received UDP packet by calling UDPStartPktOut in UDPAppProcPktIn. Before calling UDPStartPktOut, you will have to set up the local UDP port number from which you will be sending the packet as well as the remote UDP port number you will be sending the packet to. Since you are immediately responding to a UDP packet, the remote UDP port number will be available in variable udpRxSrcPortMSB, LSB. The local UDP port is set up by udpTxSrcPortMSB, LSB and the remote UDP port by udpTxDestPortMSB, LSB. You will also have to set up a variable called udpTxDataLenMSB, LSB which indicates the number of data bytes you will be transmitting in the UDP packet.

After the call to UDPStartPktOut you can load the databytes into the UDP packet by loading each databytes into the working register and calling NICWriteAgain after each load. Once all the databytes are loaded you must call UDPEndPktOut to end and send the UDP packet. The sequence described above is shown below:

```
; ******************************************************************************
```

```
UDPAppProcPktIn
; Application Incoming UDP packet handler (Example)
; This function is called whenever an application (matches udpRxDestPortxSB)
; packet is received. The appplication can call NICReadAgain() to extract
; sequentially extract each byte of the <data> field in the UDP packet.
; [UDP API Function]
; INPUT: {udpRxDataLenMSB,udpRxDataLenLSB} = number of bytes in UDP <data>
; {udpRxSrcPortMSB,udpRxSrcPortLSB} = UDP <source_port>
; OUTPUT: none
;
*****************************************************************
**************
call NICReadAgain_7 ; retrieve first databyte from UDP packet
and w, #%01000000
xor ra, w; toggle I/O pins
_bank UDP_BANK
clr udpTxSrcPortMSB ; set up local UDP port
clr udpTxSrcPortLSB

mov udpTxDestPortMSB, udpRxSrcPortMSB ; set up remote UDP port (reply to received UDP packet)
mov udpTxDestPortLSB, udpRxSrcPortLSB

mov udpTxDataLenMSB, #0 ; set up the number of databytes to be loaded into the UDP packet
mov udpTxDataLenLSB, #2

call UDPStartPktOut

mov w, ra; send new port state, (set up databyte no1)
call NICWriteAgain_7 ; write databyte no1 into UDP packet
mov w, #$00; one-byte padding, (set up databyte no2)
call NICWriteAgain_7 ; write databyte no2 into UDP packet

call UDPEndPktOut
retp
```

You may also use the above code statements in a custom function called from the main loop if you need to periodically transmit UDP packets or transmit a packet in response to a local hardware trigger such as a switch press. You will however need to set up a variable called remoteIP3-0 with the remote IP address of the remote server you are sending the UDP packet to. RemoteIP3-0 is also used by the other protocols on the stack and in the above code example the remoteIP3-0 variable is already set up with the correct IP address of the remote server since we are replying immediately to an incoming UDP packet. The variable was loaded by the TCP/IP stack software when the UDP packet was received.

8.0 Using the SMTP API

The SMTP API does not consist of function calls as does the TCP API and UDP API. SMTP uses TCP as its carrier or transport protocol to send and receive SMTP data. The SMTP API thus consists only of pre-defined locations in program memory where you can insert SMTP data. SMTP data commonly consists of sender and recipient email addresses, a domain name and email text. The locations in the source file where SMTP data is located are indicated with assembler labels looking similar to the TCP API function definitions. When we say that the SMTP API "function" is "called" we will be meaning that the SMTP data located in the SMTP API function will be read by the SMTP state machine. The SMTP state machine can also be called a part of the overall application code called the send email client.

8.1 Using TCP as the carrier for SMTP data

Before the SMTP state machine can read the data stored in the SMTP API locations, there has to be an established TCP connection between the demo board and a remote SMTP server. SMTP servers listen on TCP port number 25 for incoming connections. You will therefore have to insert "trigger code" in TCPApp1Init to establish a TCP connection with a remote SMTP server. TCPApp2Init cannot be used since it can only be used to do a passive open of a TCP connection. Refer to 6.0 "Using the TCP API" for example code on using TCPApp1Init to establish a TCP connection.

8.2 Starting the SMTP state machine

The SMTP state machine is implemented in the send email client application code. The application code is inserted in the TCP API functions since SMTP uses TCP as its data carrier. This means that incoming SMTP data will be delivered to the SMTP state machine in the receive TCP API functions and outgoing SMTP data will be loaded from the TCP API functions related to the transmission of TCP segments. The SMTP state machine is started by loading a variable called smtpState with a value pre-defined as SMTP_CONNECT. This must be done in TCPApp1Init since this function will not be called again once the TCP connection is established. The SMTP state machine does not actually start to do anything until the TCP connection is open. Once the connection is open, the send email client will connect with the remote SMTP server to send an email.

8.3 Sending email

Email is commonly sent to SMTP servers listening on TCP port 25. This is a universally well-known port number and is setup by setting SMTP_PORT_MSB and SMTP_PORT_LSB to a value of 25. This may be changed if using a private SMTP server listening at a different port.

The following SMTP API "functions" will be "called" by the SMTP state machine once a TCP connection is open and the state machine was started:

1. _SenderDomainName - The SMTP state machine reads the databytes stored here as program words and the send email client uses this domain name to identify itself to the remote SMTP server. In the code shown below, the domain name is sx. If you created an email account for the demo board called board@acme.com, the domain name will be acme.

```
;
******************************************************************************
_senderDomainName
; Contains the sender's domain name.
; [SMTP API function]
; INPUT:  none
; OUTPUT: none
;
******************************************************************************
SMTPTEXT_HELO = $
dw 'HELO '
; insert the Domain name here
dw 'sx'
dw CR,LF
SMTPTEXT_HELO_END = $
```

2. _mailFrom - The SMTP state machine reads the databytes stored here as program words and the send email client uses this as the name of the sender. In the code below, the sender's name is sx. If the board's email address was created as board@acme.com, the sender's name would be board.

```
;
******************************************************************************
_mailFrom
; Contains the sender's name.
; [SMTP API function]
; INPUT:  none
; OUTPUT: none
;
******************************************************************************
SMTPTEXT_MAIL=$
dw'MAIL FROM: '
; insert the sender's name here
dw'<sx>'
dwCR,LF
SMTPTEXT_MAIL_END=$
```

3. _mailTo - The SMTP state machine reads the databytes stored here as program words and uses this as the email address of the recipient. In the code shown below, the recipient’s email address is joe@demo.sx

```
; ******************************************************************************
_mailTo
; Contains the recipient's email address.
; [SMTP API function]
; INPUT:  none
; OUTPUT: none
; ******************************************************************************
SMTPTEXT_RCPT=$
dw 'RCPT TO: '
; insert the recipient's email address here
dw '<joe@demo.sx>'
dw CR,LF
SMTPTEXT_RCPT_END=$
```

4. _mailData - The SMTP state machine reads the databytes stored here as program words and uses this as the email message text that is sent to the recipient. Standard fields interpreted by email programs such as Outlook Express are commonly included in the email message. Some of these fields are “From:, To: and Subject”.

```
; ******************************************************************************
_mailData
; Contains the mail message Data.
; [SMTP API function]
; INPUT:  none
; OUTPUT: none
; ******************************************************************************
SMTPTEXT_TEXT=$
dw 'From: SX'
dw CR,LF
; insert To: field here
dw 'To: Joe'
dw CR,LF
; insert Subject: field here
dw 'Subject: Button Pressed!'
dw CR,LF,CR,LF
; insert body of email message here
dw 'Button SW2 pressed'
dw CR,LF,'.',CR, LF
SMTPTEXT_TEXT_END=$
```

9.0 Application Programming Interface (API)

9.1 UDP Functions

9.1.1 UDPAppInit()

Application UDP Initialization code. This function is called automatically once by the stack during startup.

INPUT: none

OUTPUT: none

9.1.2 UDPAppProcPktIn()

Application Incoming UDP packet handler. This function is called whenever an application (matches udpRxDestPortxSB) packet is received. The application can call NICReadAgain() to extract sequentially extract each byte of the <data> field in the UDP packet.

INPUT: {udpRxDataLenMSB,udpRxDataLenLSB} = number of bytes in UDP <data>

{udpRxSrcPortMSB,udpRxSrcPortLSB} = UDP <source_port>

OUTPUT: none

9.1.3 UDPStartPktOut()

Starts an outgoing UDP packet by constructing an IP and UDP packet header.

INPUT: {remoteIP0-3} = destination IP addr for UDP pkt

{udpTxSrcPortMSB,udpTxSrcPortLSB} = UDP Source Port

{udpTxDestPortMSB,udpTxDestPortLSB} = UDP Destination Port

{udpTxDataLenMSB,udpTxDataLenLSB} = UDP Data Length (just data)

OUTPUT: none

9.1.4 UDPEndPktOut()

Wraps up and transmits the UDP packet.

INPUT: none

OUTPUT: none

9.1.5 NICReadAgain()

Call this function to extract, one byte at a time, the data encapsulated in the UDP packet This function should be called within UDPAppProcPktIn().

INPUT: none

OUTPUT: w = byte read

9.1.6 NICWriteAgain()

Call this function to write, on byte at a time, the data to be sent in a UDP packet.This function should be called after calling UDPStartPktOut(), and before calling UDPEndPktOut().

INPUT: w = byte to be written

OUTPUT: none

9.2 UDP Variables

9.2.1 remoteIP[3:0]

Destination IP address of outgoing packet, as well as, Source IP address of incoming packet.

9.2.2 myIP[3:0]

Source IP address of outgoing packet, as well as, filter for Destination IP address of incoming packets. This is usually set to the IP address assigned to the SX.

9.2.3 UDPRxSrcPortMSB, UDPRxSrcPortLSB

Source UDP Port number of incoming packet.

9.2.4 UDPRxDestPortMSB, UDPRxDestPortLSB

Filter for Destination Port number of incoming UDP packets.

9.2.5 UDPRxDataLenMSB, UDPRxDataLenLSB

Length, in bytes, of the data field of incoming UDP packet.

9.2.6 UDPTxSrcPortMSB, UDPTxSrcPortLSB

Source UDP Port number of outgoing packet.

9.2.7 UDPTxDestPortMSB, UDPTxDestPortLSB

Destination UDP Port number of outgoing packet.

9.2.8 UDPTxDataLenMSB, UDPTxDataLenLSB

Length, in bytes, of the data field of incoming UDP packet.

9.3 TCP Functions

9.3.1 TCPApp1Init()

TCP application no. 1 initialization code. Called repeatedly as long as TCP connection no. 1 state is closed.

INPUT: none

OUTPUT: none

9.3.2 TCPApp2Init()

TCP application no. 2 initialization code. Called repeatedly as long as TCP connection no. 2 state is closed.

INPUT: none

OUTPUT: none

9.3.3 TCPAppTxBytes()

Called before transmitting a TCP packet to see if the application has any data it wishes to send. The application cannot send more than TCP_SEG_SIZE (1400) bytes at one go.

INPUT: none

OUTPUT: {tcp1UnAckMSB,tcp1UnAckLSB} = number of bytes to transmit on tcp connection no.1 or

{tcp2UnAckMSB,tcp2UnAckLSB} = number of bytes to transmit on tcp connection no.2

9.3.4 TCPAppRxBytes()

Indicator to the application that a packet has been received and that TCPAppRxByte is about to be called as many times as they are bytes of data.

INPUT: {tcpAppRxBytesMSB,tcpAppRxBytesLSB} = number of received data bytes

OUTPUT: none

9.3.5 TCPAppTxData()

This routine is called once for each byte the application has says it wishes to transmit.

INPUT: none

OUTPUT: w = data byte to transmit

9.3.6 TCPAppRxData()

Called once for each byte received in a packet.

INPUT: w = received data byte

OUTPUT: none

9.3.7 TCPAppTxDone()

This is called following the last call to TCPAppTxData(). It signifies the transmitted data has successfully reached the remote host.

INPUT: none

OUTPUT: none

9.3.8 TCPAppRxDone()

This is called following the last call to TCPAppRxData(). It signifies the end of the received packet.

INPUT: none

OUTPUT: none

9.3.9 TCPAppPassiveOpen()

Do a passive open. For example, listen for connections on a given port.

INPUT: {tcb1LocalPortMSB, tcb1LocalPortLSB} = TCP port to listen on for TCP connection no. 1 or

{tcb2LocalPortMSB, tcb2LocalPortLSB} = TCP port to listen on for TCP connection no. 2

OUTPUT: none

9.3.10 TCPAppActiveOpen()

For example, initiate a connect to a remote TCP.

INPUT: {tcb1LocalPortMSB, tcb1LocalPortLSB} = TCP port to listen on for TCP connection no. 1 or

{tcb2LocalPortMSB, tcb2LocalPortLSB} = TCP port to listen on for TCP connection no. 2

{tcb1RemotePortMSB, tcb1RemotePortLSB} = remote TCP port to establish connection with on TCP connection no. 1 or

{tcb2RemotePortMSB, tcb2RemotePortLSB} = remote TCP port to establish connection with on TCP connection no. 2

{sock1RemoteIP3-0} = remote IP address for TCP connection no. 1

{sock2RemoteIP3-0} = remote IP address for TCP connection no. 2

OUTPUT: none

9.3.11 TCPAppClose()

Force the current connection to close.

INPUT: none

OUTPUT: none

9.4 Variables

9.4.1 sock1RemoteIP[3:0], sock2RemoteIP[3:0]

IP address of remote end-point for which a TCP connection has been, or is to be, established.

9.4.2 myIP[3:0]

IP address of local end-point. This is usually set to the IP address assigned to the SX.

9.4.3 tcb1LocalPortMSB, tcb1LocalPortLSB, tcb2LocalPortMSB, tcb2LocalPortLSB

Local TCP port number for each connection. A TCP connection 'end-point' is specified by the unique pair comprising of the node's IP address, as well as a socket, or 'port' number.

9.4.4 tcb1RemotePortMSB, tcb1RemotePortLSB, tcb2RemotePortMSB, tcb2RemotePortLSB

Remote TCP port number for each connection. A TCP connection 'end-point' is specified by the unique pair comprising of the node's IP address, as well as a socket, or 'port' number.

9.5 SMTP functions

9.5.1 _senderDomainName

Contains the send email client's domain name. Example: If the Ethernet SX-Stack board's email address is eSX@acme.com, the sender's domain name is acme.com.

9.5.2 _mailFrom

Contains the sender's name. Example: If the EthernetSX-Stack board's email address is eSX@acme.com, the sender's name is eSX.

9.5.3 _mailTo

Contains the recipient's email address. Example: If the Ethernet SX-Stack board sends an email to joe@demo.sx, the recipient's email address is joe@demo.sx

9.5.4 _mailData

Contains the data of the email message.

9.6 Variables

9.6.1 SMTP_SERVER_IP[3:0]

IP address of the SMTP server to be contacted when sending email. Example: If the SMTP server's IP address is 168.75.232.39, SMTP_SERVER_IP3=168, SMTP_SERVER_IP2=75, SMTP_SERVER_IP1=232, SMTP_SERVER_IP0=39

9.6.2 smtpState

Contains the current state of the SMTP state machine. Must be set equal to SMTP_CONNECT to start the state machine to send an email.

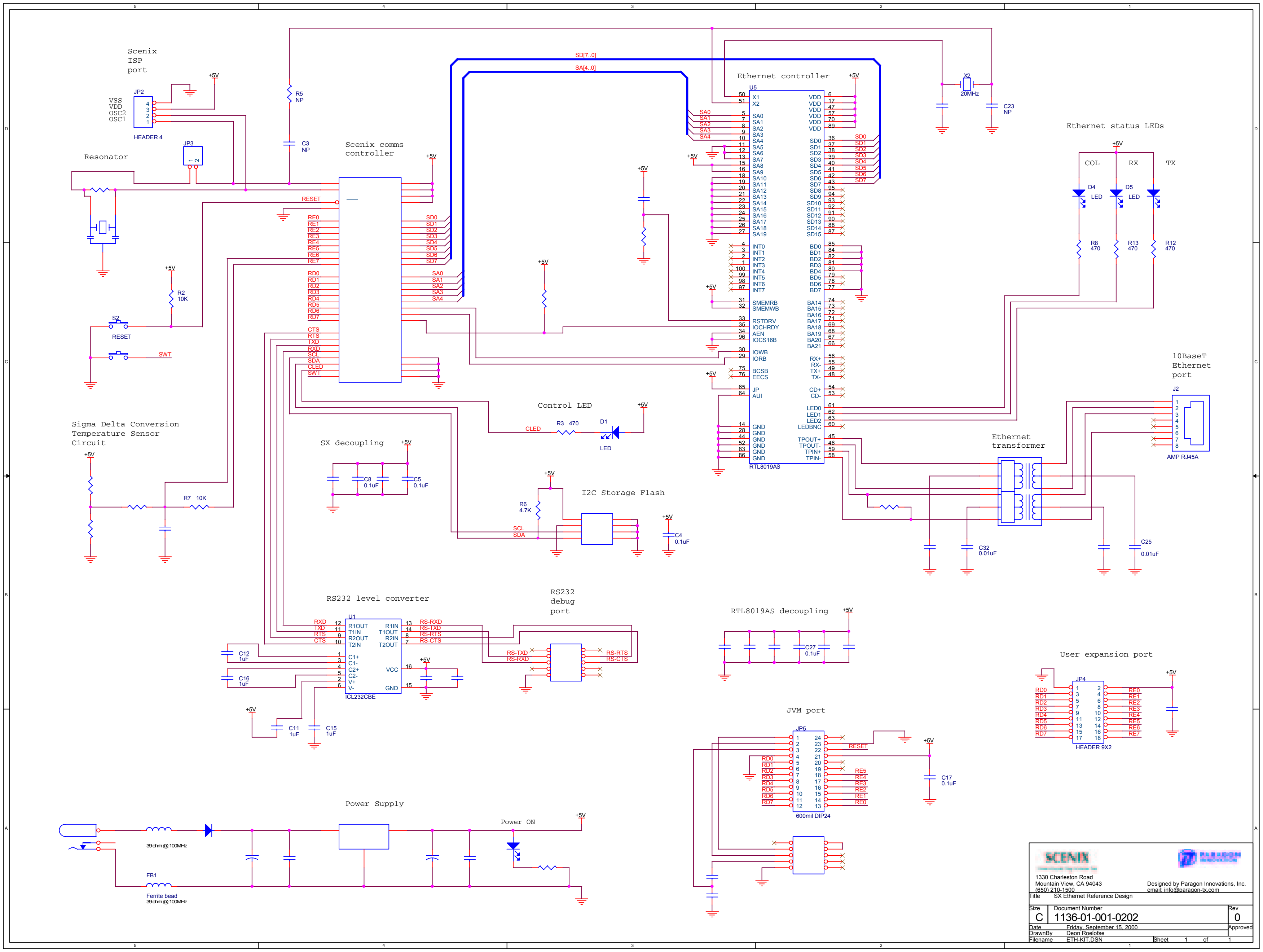
9.6.3 flags3.SMTP_OK

A flag that indicates if the SMTP state machine finished successfully (email accepted by remote SMTP server). Equals 1 if successful or 0 if error encountered.

Appendix A: Ethernet SX-Stack Demo Board Schematic

Following is the description of the key elements of the schematic:

- The frequency of X1, the crystal/resonator, can be changed by the user, since the TCP/IP protocol stack is non-real time and operates almost exclusively in the mainline context. Of course, certain restrictions apply. For example, the receive buffer on the NIC should not be allowed to overflow, or packets will be lost.
- R5 and C3 can be loaded in lieu of X1 for cost-reduction (will reduce SX speed to 20MIPS).
- The supplied code will only work with SX52 Revision 2.x silicon (the production-release version).
- Whilst the supplied reference design demonstration code works with the prescribed I/O pinouts, in general, the I/O pinouts can be changed with little modification to the code.
- Some components in the reference design are only used for demonstration purposes, please see the Bill-of-Materials section for a list of the core components.
- Connectors J4, J5, and J6 are for future expansion.
- U3 is a 32KB I^2C Serial EEPROM, and is used for storing web data only. Thus, users who do not need HTTP need not have this component in their design.
- U1 (and associated components) and J1 are used for reprogramming the Serial EEPROM (U3).
- R1, R7, R10, R11 and C7 implement a temperature sensor, which will provide real-time data to be displayed on a dynamic web-page for demonstration purposes.
- D1, R3 and S1 are for demonstration purposes only.
- The reference design hardcodes the 48-bit Ethernet physical address in the SX, which is FLASH-reprogrammable. The user can also store the physical address in a separate configuration device by attaching a 9346 Serial 1k-bit EEPROM to BD[5:7] on U5. Refer to Sect. 6.3 of the RTL8019AS datasheet for more information.
- JP2 is the In-System-Programming (ISP) header. The SX can be (re)programmed using just the OSC1and OSC2 pins.
- For more information on U5, visit http://www.realtek.com.tw/cn . For information on ordering or pricing on U5, visit http://www.realtek.com.tw/cn/contact/service.htm .
- For more information on T1, visit http://www.bothhandusa.com/datasheets/filters/filters.htm .
- For information on ordering or pricing on X1, visit http://www.murata.com/murata/murata.nsf/pages/sales/#salesreps.


Scenix ISP port
JP2
VSS
VDD
OSC2
OSC1
HEADER 4
JP3
Resonator
R5 NP
C3 NP
Scenix comms controller
RESET
RE0 RE1 RE2 RE3 RE4 RE5 RE6 RE7
RD0 RD1 RD2 RD3 RD4 RD5 RD6 RD7
CTS RTS TXD RXD SCL SDA CLED SWT
SD0 SD1 SD2 SD3 SD4 SD5 SD6 SD7
SA0 SA1 SA2 SA3 SA4
SD[7..0]
SA[4..0]
Ethernet controller
U5
X1 X2
VDD
SA0 SA1 SA2 SA3 SA4 SA5 SA6 SA7 SA8 SA9 SA10 SA11 SA12 SA13 SA14 SA15 SA16 SA17 SA18 SA19
INT0 INT1 INT2 INT3 INT4 INT5 INT6 INT7
SMEMRB SMEMWB
RSTDRV IOCHRDY AEN IOCS16B
IOWB IORB
BCSB EECS
JP AUI
GND
SD0 SD1 SD2 SD3 SD4 SD5 SD6 SD7 SD8 SD9 SD10 SD11 SD12 SD13 SD14 SD15
BD0 BD1 BD2 BD3 BD4 BD5 BD6 BD7
BA14 BA15 BA16 BA17 BA18 BA19 BA20 BA21
RX+ RX- TX+ TX-
CD+ CD-
LED0 LED1 LED2 LEDBNC
TPOUT+ TPOUT- TPIN+ TPIN-
RTL8019AS
X2 20MHz
C23 NP
Ethernet status LEDs
COL RX TX
D4 LED
D5 LED
R8 470
R13 470
R12 470
10BaseT Ethernet port
J2
AMP RJ45A
Ethernet transformer
C32 0.01uF
C25 0.01uF
S2 RESET
SWT
R2 10K
Control LED
CLED
R3 470
D1 LED
Sigma Delta Conversion Temperature Sensor Circuit
R7 10K
SX decoupling
C8 0.1uF
C5 0.1uF
I2C Storage Flash
R6 4.7K
SCL SDA
C4 0.1uF
RS232 level converter
U1
R1OUT T1IN R2OUT T2IN
R1IN T1OUT R2IN T2OUT
C1+ C1- C2+ C2- V+ V-
VCC GND
ICL232CBE
RXD TXD RTS CTS
RS-RXD RS-TXD RS-RTS RS-CTS
C12 1uF
C16 1uF
C11 1uF
C15 1uF
RS232 debug port
RTL8019AS decoupling
C27 0.1uF
User expansion port
JP4
HEADER 9X2
JVM port
JP5
600mil DIP24
C17 0.1uF
Power Supply
39ohm @ 100MHz
FB1
Ferrite bead 39ohm @ 100MHz
Power ON
+5V
SCENIX
1330 Charleston Road
Mountain View, CA 94043
(650) 210-1500
Designed by Paragon Innovations, Inc.
email: info@paragon-tx.com
Title SX Ethernet Reference Design
Size C
Document Number 1136-01-001-0202
Rev 0
Date Friday, September 15, 2000
DrawnBy Deon Roelofse
Filename ETH-KIT.DSN
Sheet 1 of 1
Approved

Appendix B: Bill of Material

The reference designators highlighted in bold are components either used only for demonstration purposes, or whose use is optional.

No	Qty	Ref	Val	Description	Part no
1	2	C1,2	47uF 25V	Electrolytic capacitor	Panasonic: ECE-V1EA470UP
2	5	**C10-12,15,16**	1uF 16V	Ceramic capacitor SMT0805	Panasonic: ECJ-2VF1C105Z
3	1	C13	0.1uF 50V	Ceramic capacitor SMT0805	Panasonic: ECJ-2YB1H104K
4	4	C**3**,25,26,31,32	0.01uF	Ceramic capacitor SMT1206	Panasonic: ECU-V1H103KBM
5	19	C4-9,14,**17-22,24**,27,29,30	0.1uF 16V	Ceramic capacitor SMT0603	Panasonic: ECJ-1VB1C104K
6	1	**D1**	-	Orange LED	Panasonic: LNJ808R8ERA
7	1	D2	1A 50V	Rectifier diode MELF	Diodes Inc: DL4001
8	2	D**3**,4	-	Red LED	Panasonic: LNJ208R8ARA
9	1	D5	-	Green LED	Panasonic: LNJ308G8TRA
10	1	D6	-	Amber LED	Panasonic: LNJ408K8ZRA
11	2	FB1,2	39 ohm	Ferrite bead SMT0805	Panasonic: EXC-ML20A390U
12	1	J1	-	2.0mm Power jack SMT	Cui Stack: PJ-002A
13	1	J2	-	Modular connector	AMP: 520426-4
14	2	**JP1,6**	-	10pin dual row male header	Digikey: S2012-05-ND
15	1	JP2	-	4pin single row male header	Digikey: S1012-04-ND
16	1	JP3	-	2pin single row male header	Digikey: S1012-02-ND
17	1	**JP4**	-	18pin dual row male header	Digikey: S2012-09-ND
18	1	**R1**	5K	NTC thermistor SMT0805	Thermometrics: NC0805R502T10
19	1	R16	200 ohm 5%	Carbon film resistor SMT0603	Panasonic: ERJ-3GSYJ201V
20	5	R2,4,7,10,14	10K 5%	Carbon film resistor SMT0603	Panasonic: ERJ-3GSYJ103V
21	5	R3,**5**,8,9,12,13	470 ohm 5%	Carbon film resistor SMT0603	Panasonic: ERJ-3GSYJ471V
22	3	R6,11,15	4.7K 5%	Carbon film resistor SMT0603	Panasonic: ERJ-3GSYJ472V
23	2	S**1**,2	-	Tact switch	Omron: B3S-1002
24	1	T1	-	Filter transformer	Bothhand: FB2022
25	1	**U1**	-	5V RS232 transceiver SOIC16	Intersil: ICL232CBE
26	1	U2	-	Comms controller PQFP52	Ubicom: SX52BD/PQ
27	1	U3	32Kx8	Serial EEPROM SOIC8	Microchip: 24LC256-I/SM
28	1	U4	5V 1A	Voltage regulator D^2PAK	ST: L7805CD2T
29	1	U5	-	Ethernet controller QFP100-14	Realtek: RTL8019AS
30	1	X1	50MHz	Ceramic resonator	Murata: CSTCV50.00MXJ0H3
31	1	X2	20MHz	Crystal CSM-7	ECS: ECS-200-20-5P

Appendix C: References

1. AN23: PPP/UDP Virtual Peripheral Implementation [Ubicom]
2. AN27: TCP Virtual Peripheral Implementation [Ubicom]
3. AN25: HTTP Virtual Peripheral Implementation [Ubicom]
4. RTL8019 Datasheet [Realtek]
5. RTL8019AS Datasheet [Realtek]
6. DP83905 Datasheet [National Semiconductor]
7. Internetworking with TCP/IP Volume I, 3rd Edition [Douglas E. Comer]
8. RFC1533: DHCP Options
9. RFC1541: DHCP
10. RFC791: IP
11. RFC792: ICMP
12. RFC793: TCP
13. RFC951: BOOTP
14. RFC768: UDP

Lit #: AN37-04

Sales and Tech Support Contact Information

For the latest contact and support information on SX devices, please visit the Ubicom website at www.ubicom.com. The site contains technical literature, local sales contacts, tech support and many other features.

1330 Charleston Road
Mountain View, CA 94043
Contact: Sales@ubicom.com
http://www.ubicom.com
Tel.: (650) 210-1500
Fax: (650) 210-8715