Embedded ATMEL HTTP Server

A Design Project Report

Presented to the Engineering Division of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Masters of Engineering (Electrical)

by

Tzeming Tan, Jeremy

Project Advisor: Dr. Bruce R. Land

Degree date: May 2004
Abstract

Master of Electrical and Computer Engineering Program
Cornell University
Design Project Report

**Project Title:** Embedded ATMEL HTTP Server

**Author:** Tzeming Tan, Jeremy

**Abstract:** The objective of this project was to design and build an embedded HTTP server using a microcontroller chip. The webserver required the implementation of the interface with Ethernet as well as several internet protocols such as TCP/IP and ARP. This embedded web server is able to serve small, static web pages as well as perform certain useful laboratory lab functions such as displaying the current temperature read by the microcontroller from a thermometer, on the webpage. While the capabilities of the embedded webserver are no where near that of a regular server computer, its small size and relatively low cost makes it more practical for some applications. The web server was built, tested to work, and a temperature reporting feature added to it.

Report Approved by

Project Advisor: _____________________________ Date: ________
Executive Summary

The internet is a versatile, convenient and efficient means of communication in the 21st century. Protocols such as TCP/IP, UDP, DHCP and ICMP form the backbone of internet communications a large bulk of which consists of Hyper Text Transfer Protocol (HTTP) traffic for the World Wide Web. A HTTP or web server is a server process running at a web site which sends out web pages in response to HTTP requests from remote browsers. While high performance 32 bit desktop computers are used for serving websites, much smaller and cheaper 8 or 16 bit microcontrollers, though not as powerful in terms of processing power, can do the job as well. This report details the workings of the embedded web server built for the project.

The AT-Mega32, both being versatile and adequate in terms of capability was chosen for this project. Building a HTTP server involved implementing several protocols, namely, UDP, TCP/IP, DHCP and ARP. ICMP was also implemented for testing. The chip was run on a STK500 development board. A Realtek RTL8019s Ethernet controller chip was used to interface the microcontroller with Ethernet. A RJ45 Ethernet jack was used to connect the Ethernet controller to a router.

The web server was implemented with no problems and worked. The server was able to send a DHCP request for an IP address from a router and served the required webpage on the browser when the IP address of the web server was entered. While the TCP stack is not fully RFC compliant, it is adequate for the purposes of this project. The webpage itself was stored in the flash memory of the AT-Mega32 but future improvements could include adding an external EEPROM to support larger web pages.
Table of Contents

ABSTRACT ............................................................................................................................................... II
EXECUTIVE SUMMARY ...................................................................................................................III
TABLE OF CONTENTS .................................................................................................................... IV
I) INTRODUCTION ........................................................................................................................... 1
   I.1) MOTIVATION .......................................................................................................................... 1
II) DESIGN PROBLEM AND SYSTEM OF REQUIREMENTS ......................................................... 1
   II.1) DESIGN PROBLEM ......................................................................................................... 1
   II.2) SYSTEM OF REQUIREMENTS .......................................................................................... 2
III) DESIGN AND IMPLEMENTATION ......................................................................................... 4
   III.1) HARDWARE .................................................................................................................. 4
   III.2) INTERNET PROTOCOLS .............................................................................................. 4
         ARP (Address Resolution Protocol) .................................................................................. 5
         IP (Internet Protocol) ........................................................................................................ 5
         ICMP (Internet Control Message Protocol) ..................................................................... 6
         UDP (User Datagram Protocol) ...................................................................................... 7
         DHCP (Dynamic Host Configuration Protocol) ................................................................. 7
         TCP (Transmission Control Protocol) ............................................................................. 8
         Checksum ....................................................................................................................... 10
         HTTP (Hyper Text Transfer Protocol) ............................................................................ 11
   III.3) EMBEDDED CODE ......................................................................................................... 12
   III.4) TEMPERATURE REPORTING ...................................................................................... 15
   III.5) STORING THE WEBPAGE ............................................................................................ 15
IV) TEST RESULTS ......................................................................................................................... 15
V) CONCLUSION .......................................................................................................................... 16
VI) ACKNOWLEDGEMENTS ...................................................................................................... 17
VII) REFERENCES ....................................................................................................................... 17
APPENDIX A: CODE ...................................................................................................................... 18
APPENDIX B: PACKETWACKER SCHEMATICS ................................................................. 58
APPENDIX C: WEBPAGE ............................................................................................................. 59
I) Introduction

I.1) Motivation

With the rapid advancement of the x86 processors in the recent years, 8 and 16 bit microcontrollers have become rather obsolete. However, their relatively simple architecture and cheap price make them ideal for simple functions in systems that do not require the higher computing power of the more expensive 32 bit chips.

Even so, microcontrollers can also sometimes be used to perform tasks usually relegated to 32 bit processors. Internet protocols such as TCP/IP have already been successfully ported to small 8-bit microcontrollers and thus with this capability, microcontrollers have the potential function as embedded web servers for simple web pages which can be adapted for lab applications.

The initial motivation for this project was to create either a temperature reporting web server or systems control web server which allows the user to control certain systems via the internet.

II) Design Problem and System of Requirements

II.1) Design Problem

The goal of this project was to design and implement TCP/IP as well as other internet protocols on an ATMEL Mega 32 chip so that the chip will be able to function as a simple RFC compliant web server.

The Requests for Comments (RFC) document series is a set of technical and organizational notes about the Internet (originally the ARPANET), beginning in 1969. It is important that the protocols are implemented as close to RFC specifications as possible so that the web server can be safely connected to the internet. The web server will also have to comply with HTTP standards so that the data it sends to the browser will enable a webpage to be displayed.

Since the Mega 32 is limited in terms of processing power and memory space, the implementation has to be efficient and small enough to fit into the on-
chip memory. Thus, even though the maximum allowable packet size on the internet is more than 65,000 bytes, the web server can only send and receive packets of size 700 bytes since it has only 2,000 bytes of SRAM. This limitation is easily solved by simply sending more packets. HTTP requests are usually less than 300 bytes long and therefore it is within the limitations.

After the webserver was designed, the project was taken a step further by connecting the Mega 32 to a LM34 thermometer and reporting the temperature on the webpage. This is just one example of a useful application for the embedded web server.

II.2) System of Requirements
Since the web server will be referenced to by its IP address in the browser, it will be connected either to a DHCP (Dynamic Host Configuration Protocol) enabled router or directly to the internet so a DHCP implementation is required for the web server to obtain its IP address.

There were already existing implementations of UDP/IP for the ATMEL microcontrollers and also free embedded TCP/IP (Adam Dunkel’s uIP) source code for microcontrollers in general. However, these were either too complex or too simple to be used for implementing the web server. Therefore, a major part of this project was dedicated to creating a new TCP/IP stack along with ICMP, UDP, ARP and DHCP specifically for the purposes of serving web pages. The web server should be able to fulfill the following requirements:

- Send and receive Ethernet packets
- Differentiate between and respond to ARP and IP packets
- Request and receive an IP address from a router (DHCP)
- Respond to a ping (ICMP)
- Send and receive TCP and UDP packets
- Perform the appropriate checksums and acknowledgements for TCP
- Have enough TCP functionality to serve webpages.
- Have enough versatility such that another user can change and modify the webpage or add webpages.
• Have a very small code footprint and requirement for RAM so that it can fit onto an ATMEL Mega 32 chip.

This preliminary set of requirements will be discussed in greater detail in the sections to come.
III) Design and Implementation

III.1) Hardware

The 8-bit ATMEL Mega32 was chosen for this project since it has a sizable amount of SRAM (2kb) and Flash (32kb) and is one of the more current microcontrollers in the market. It also had an inexpensive price tag and came with comprehensive documentation and software support. The development board used in this project was the STK500.

There are several ways to connect the microcontroller to the internet, two of which include using an Ethernet controller and using the SLIP interface for a serial connection. The latter was shown in previous projects to be extremely lossy and unreliable therefore the Ethernet controller method was chosen for this project. The Realtek RTL8019s Ethernet controller chip was chosen since it is compatible with the ATMEL microcontrollers and a Packet Wacker module from EDTP which consisted of a RJ45 jack and the RTL8019s was used for this project. The schematics for the Packet Wacker module are shown in Appendix B. The Ethernet controller works by receiving only packets destined to its MAC address (which is defined by the microcontroller) and sending it to the microcontroller. For the sending of packets, it stores the data in the buffer and employs the use of collision detection to determine when to send.

The temperature reporting function was implemented using a LM34 temperature sensor along with a LMC7111 amplifier. The temperature sensor produces 10mV/°F which is passed through the amplifier and then to the on board ADC at PORTA of the STK500. The microcontroller then reads in the voltage and displays the current temperature on the webpage.

III.2) Internet Protocols

Information is transmitted in packets of binary code on the internet. The code is grouped into octets (bytes) and the bytes are grouped into packets of data. Several internet protocols are required such that the receiver can interpret the data correctly. The following are brief descriptions of internet protocols were implemented for the web server.
**ARP (Address Resolution Protocol)**

ARP is used to translate IP addresses to link addresses (MAC) and hide these addresses from the upper layers. This protocol maps the IP address to a corresponding MAC address. In general, an ARP module is broadcast into the network containing the IP address. If a machine recognizes its IP address in the ARP request, it will return an ARP reply to the inquiring machine containing its MAC address. In essence, a broadcast ARP packet asks "who belongs to this IP address" and the reply from the corresponding machine is "I do and here is my MAC address". The MAC address of the host machine must be known in order to send it Ethernet packets and thus ARP is needed in this project.

The ARP packet structure is shown below with the corresponding number of bytes for each field:

<table>
<thead>
<tr>
<th>Field</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination Address</td>
<td>6</td>
</tr>
<tr>
<td>Source Address</td>
<td>6</td>
</tr>
<tr>
<td>Ethertype</td>
<td>2</td>
</tr>
<tr>
<td>Hardware type</td>
<td>2</td>
</tr>
<tr>
<td>Protocol type</td>
<td>2</td>
</tr>
<tr>
<td>Hardware length</td>
<td>1</td>
</tr>
<tr>
<td>Protocol length</td>
<td>1</td>
</tr>
<tr>
<td>Op code</td>
<td>2</td>
</tr>
<tr>
<td>Sending hardware address (MAC)</td>
<td>6</td>
</tr>
<tr>
<td>Sending protocol address</td>
<td>4</td>
</tr>
<tr>
<td>Target hardware address</td>
<td>6</td>
</tr>
<tr>
<td>Target protocol address</td>
<td>4</td>
</tr>
</tbody>
</table>

**IP (Internet Protocol)**

The IP protocol is a network layer protocol, which permits the exchange of traffic between two host computers. Each computer is assigned an IP address so that the networks can know which computer the packet is addressed to and which
computer the packet is from. Protocols such as TCP, UDP and ICMP are encapsulated into IP packets. The IP packet structure is shown in the table below:

<table>
<thead>
<tr>
<th>Field</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>4</td>
</tr>
<tr>
<td>Header Length</td>
<td>4</td>
</tr>
<tr>
<td>Type of Service</td>
<td>8</td>
</tr>
<tr>
<td>Total Length</td>
<td>16</td>
</tr>
<tr>
<td>Identifier</td>
<td>16</td>
</tr>
<tr>
<td>Flags</td>
<td>3</td>
</tr>
<tr>
<td>Fragment Offset</td>
<td>13</td>
</tr>
<tr>
<td>Time to Live</td>
<td>8</td>
</tr>
<tr>
<td>Protocol</td>
<td>8</td>
</tr>
<tr>
<td>Header Checksum</td>
<td>16</td>
</tr>
<tr>
<td>Source Address</td>
<td>32</td>
</tr>
<tr>
<td>Destination Address</td>
<td>32</td>
</tr>
<tr>
<td>Options and Padding</td>
<td>Variable</td>
</tr>
<tr>
<td>Data</td>
<td>Variable</td>
</tr>
</tbody>
</table>

Since IP is a best effort, connectionless protocol, the tasks of error checking, reliability and flow control are given to upper layers such as TCP. The protocol number field indicates the type of upper layer service required by the data packet.

**ICMP (Internet Control Message Protocol)**

This protocol is used for pinging and for reporting errors in the network. The pinging computer sends an ICMP packet to the destination computer which then echos the packet back to the pinging computer. This protocol is used also to provide for some administrative and status messages such as response time. This protocol was implemented on the webserver mainly for testing purposes. The ICMP packet consists of the IP header and the first 64 bits of the original data. ICMP has a protocol number of 1 in the IP Protocol ID field.
**UDP (User Datagram Protocol)**

UDP is a connectionless protocol used for sending data. It has very limited checksum and does not have end to end accountability of traffic. It is only used when the full TCP services are not needed. For the webserver in this project, it is used for sending and receiving DHCP messages. The port fields are used to identify and direct the datagrams to the proper upper layer application. UDP has a protocol number of 17 in the IP Protocol ID field. The header structure of UDP is given below:

<table>
<thead>
<tr>
<th>Field</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>16</td>
</tr>
<tr>
<td>Destination Port</td>
<td>16</td>
</tr>
<tr>
<td>Length</td>
<td>16</td>
</tr>
<tr>
<td>Checksum</td>
<td>16</td>
</tr>
<tr>
<td>Data</td>
<td>Variable</td>
</tr>
</tbody>
</table>

**DHCP (Dynamic Host Configuration Protocol)**

DHCP is a BOOTP based protocol for the transfer of configuration information to hosts in a TCP/IP based network. UDP is used in this case for transmitting the DHCP packet. DHCP is mainly used for obtaining an IP address from a designated DHCP server such as a router. Dynamic IP allocation (which means that the server leases the IP address) is used for the embedded webserver. DHCP has the longest header among all the protocols which is given in figure 4 on the next page.
<table>
<thead>
<tr>
<th>Field</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>8</td>
</tr>
<tr>
<td>Htype</td>
<td>8</td>
</tr>
<tr>
<td>Hlen</td>
<td>8</td>
</tr>
<tr>
<td>Hops8</td>
<td>8</td>
</tr>
<tr>
<td>Xid</td>
<td>32</td>
</tr>
<tr>
<td>Secs</td>
<td>16</td>
</tr>
<tr>
<td>Flags</td>
<td>16</td>
</tr>
<tr>
<td>Ciaddr</td>
<td>32</td>
</tr>
<tr>
<td>Yiaddr</td>
<td>32</td>
</tr>
<tr>
<td>Siaddr</td>
<td>32</td>
</tr>
<tr>
<td>Giaddr</td>
<td>32</td>
</tr>
<tr>
<td>Chaddr</td>
<td>128</td>
</tr>
<tr>
<td>Sname</td>
<td>512</td>
</tr>
<tr>
<td>Magic cookie</td>
<td>4</td>
</tr>
<tr>
<td>Options</td>
<td>variable</td>
</tr>
</tbody>
</table>

TCP (Transmission Control Protocol)

TCP traffic accounts for more than 90% of the internet traffic. It is an interactive connection protocol which deals primarily with end to end reliability, the flow of data in the internet, as well as error checking, retransmission and sequencing. As with UDP, socket calls are used to determine the type of service required and in this case, the well known port 80 is used to indicate a HTTP request. HTTP traffic is sent via TCP and therefore it is the most essential protocol for this project. Functions such as the 3-way handshake synchronization, TCP close connection, checksum, data retransmission and data sequencing were implemented in this project. Many of the other complex protocol functions such as traffic management and multiple connectivity (being able to maintain multiple connections simultaneously) were not implemented since they were redundant for the purposes of this project. TCP has a protocol number of 6 in the IP Protocol ID field. The header fields of TCP are shown in figure 5 on the following page.
The first step in establishing a TCP connection is a 3-way handshake which is shown below:

1) Client sends a SYN request (SYN flag = 1)
2) Host replies with a SYN and an ACK (SYN, ACK =1)
3) Client sends an ACK (ACK=1)
4) Connection is established
After establishing a connection, the host proceeds to send the client data. The steps in which the data is sent is shown below:

![Figure 7: Data transfer example](image)

The host sends the data to the client with a starting sequence number. The client responds by replying with an acknowledgement number which is the sum of the number of bytes in the data received and the sequence number. If the client does not receive all of the data sent, the Host TCP will resend the lost bytes starting from the client’s acknowledgement number (e.g. if the ACK in the above diagram is 402 instead of 502, the Host TCP will resend the last 100 bytes of the initial data). If an ACK is not received after a certain amount of time, the host will resend the original data and continue to do so until an ACK is received. There are several ways to inform the client that all the data has been sent and the method used in this webserver was simply to set the FIN flag when sending the last packet.

**Checksum**

The checksum operations for IP, ICMP, TCP and UDP use the same algorithm. This algorithm follows the following steps:

1) Set checksum field to 0
2) Calculate 16-bit 1s complement sum of the header which is treated as a sequence of 16 bit words
3) Store this sum in the checksum field
4) At the receiver, calculate 16-bit 1s complement of the header
5) Receiver’s checksum is all 1s if the data has not been corrupted
The receiver essentially checks if the 16-bit 1s complement of the header is the same as that of the checksum field.

**HTTP (Hyper Text Transfer Protocol)**

This is the basic protocol used to code web pages. The code is text-based, which makes it relatively easy to send in TCP packets. Below is the HTTP code of the webpage served by the webserver:

```html
<html>
<head>
<title>MENG PROJECT</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>
<body bgcolor="#FFFFFF">
<h1><strong><font color="#0000FF" face="Arial, Helvetica, sans-serif">WELCOME TO ATMEL WEBSERVER</font></strong></h1>
<p>&nbsp;</p>
<p><strong><font color="#0000FF" face="Arial, Helvetica, sans-serif">This webserver is running on a mega32 and using an EDTP packet wacker</font></strong></p>
<p>&nbsp;</p>
<p><strong><font color="#0000FF" face="Arial, Helvetica, sans-serif">This project was done by Tzeming Tan</font></strong></p>
<p>&nbsp;</p>
<p>ATMEL embedded webserver</p>
<p>TCP/IP is the communications protocol most widely used for accessing the internet today. The objective of this project is to introduce this protocol to the ATMEL Mega 32 Microcontroller chip such that it can run as a simple webserver which can then be adapted for useful lab-based applications.</p>
<p>Although the Mega 32 and the development board will only be able to run limited web functions, it is a relatively inexpensive device compared to high power web servers. Therefore it can still have many applications for example, connecting the microcontroller to household appliances will allow the user to turn them on and off anywhere using the internet. 

The current temperature is now "<font color=#FF0000>" temp "</font>" degrees F.
</p>
<p>&nbsp;</p>
</body>
</html>
```

Fig8. HTTP code for index.htm

Each character of this HTTP code is stored as its hexadecimal equivalent in a byte of flash memory of the microcontroller. The TCP function retrieves the data when a HTTP Get request is received and sends it in packets using the `strncpy()` function. The receiver knows that the data contains HTTP since the source port is the internet port 80 and converts the hexadecimal numbers back into characters after which the appropriate webpage is displayed on the browser. One problem with programming the HTTP code in C was that the double quotation symbol was used to indicate a string and there was no way to include it in a string.
III.3) Embedded Code

Please refer to Appendix A for the C code of the webserver. The first thing the microcontroller does after initialization is to execute the DHCP function call. Once the IP address has been assigned to the webserver, it will enter a while loop where it waits for the Ethernet controller to signal that a packet has been received. Once a packet has been successfully received, the webserver then decides which protocol to execute in response to the received packet. Due to memory constraints, the same SRAM buffer is used both for sending and receiving packets and therefore, received or sent packets are discarded once they are processed. A detailed description of the important functions in the code is given below:

- `void init_RTL8019AS(void)`
  This function initializes the Ethernet controller as well as the PORTs of the microcontroller.

- `void get_packet(void)`
  This function retrieves a packet from the Ethernet controller buffer and decides which next layer protocol (ARP, IP) to execute based on the type of packet received. If an IP packet is received, the function first performs an IP checksum after which it looks at the Type of Service Field to determine if the packet is UDP, TCP or ICMP and sends it to the appropriate upper layer protocol.

- `void setipaddrs(void)`
  This function sets the addresses and fields of the IP header and is called by protocols which are to be encapsulated in IP. It also performs a checksum of the entire IP datagram and places it in the checksum field of the header.
• **void arp(void)**
  This routine responds to an address query by supplying the requesting computer with the MAC address of the webserver.

• **void udp(void)**
  This function executes the User Datagram Protocol used to assemble and process the DHCP packets. It also performs a checksum on the received packet to ensure that there is no data corruption.

• **void icmp(void)**
  If an ICMP packet is received, the webserver responds by simply switching the destination and source fields and echoing the packet back to the sender. This function is called primarily when the webserver is pinged for diagnostic purposes.

• **void dhcp(void)**
  This function is used to execute the Dynamic Host Configuration Protocol which obtains an IP address from a router using UDP. At this stage, the IP address of the webserver is set to 255.255.255.255 so as to receive broadcasts. This function is a state machine which goes through the following stages:
  1) send a DHCP discover packet
  2) wait for a DHCP offer
  3) responds with a DHCP acknowledgement after which it obtains its IP address assigned by the router.
  The webserver will wait 7 seconds for the DHCP offer before resending the DHCP discover packet.
• **void tcp(void)**
  
  Upon receiving a SYN request, the TCP function will perform the 3-way handshake illustrated above. This routine also performs checksums on received packets to ensure correctness before proceeding. Data sequencing and retransmission functions were also implemented in this routine. However, since the implementation of TCP did not include multiple connectivity, the webserver can only connect to one client at a time. This means that once another SYN request is received, the connection with the previous client is lost. However, the webserver was also implemented in such a way that it is able to serve different clients and allow for browser refreshes. Once the connection is established, the webserver will wait for a HTTP GET packet (which is also in TCP) and send the HTTP code in packets via TCP to the client. The TCP 4-step close function was also implemented although it is redundant in the webserver implementation.

• **void http_server(void)**

  This function determines if the received TCP packet is a HTTP request and packs the data to be sent into the outgoing TCP datagram. In addition, this routine works with the TCP function to split the data into packets and send them in sequence.

• **void pack_html(unsigned int page, unsigned int x, unsigned int y)**

  This is the routine used to split the HTTP data into smaller portions of 500 bytes and packing them into the outgoing TCP/IP packet. The page variable indicates the webpage to be sent and the integers x and y are used for indexing the characters in the HTTP code. These indexes are stored so that on the next pass, the routine knows where to continue from where the previous packet left off.
III.4) Temperature Reporting

Port A was left free so that the onboard ADC could be used to interface with the LM34 temperature sensor for temperature reporting.

- void get_temp(void)

The temperature reporting is done by using PORT A as an ADC which will compare the voltage from the amplifier after the LM34 with Aref and send it to the microcontroller. This voltage will then be converted to its Fahrenheit equivalent using the following equation:

\[
\text{Temp} = \frac{\text{Voltage} \times \text{Aref}}{(256 \times 0.02)}
\]

which will then be displayed on the webpage using the sprintf function which converts the variable into a string.

III.5) Storing the Webpage

The webpage was stored in the flash memory of the microcontroller. Due to time constraints, only one webpage, index.htm was stored, however, the rest of the code was written such that the webserver will be able to support multiple pages and even picture files encoded in hex if so desired. The HTTP GET request usually has the requested filename right after that and if the field is empty, it is assumed that the browser is requesting the file index.htm. Therefore, if more webpages are to be added, the http_server() function has to be modified to check which file is requested.

IV) Test Results

A packet capture program, Ethereal was used to view the packets sent out by the webserver. This program was highly essential to checking and debugging the webserver. Before the DHCP implementation was done, testing was performed by connecting the webserver directly to the Ethernet port of the computer via a cross cable and using Ethereal to check that the DHCP implementation was correct. Once DHCP was implemented, I was able to connect the webserver to a router and from there, debug using Ethereal. The webserver was mostly tested on a Microsoft router but it was also tested to work seamlessly with a Linksys router. Since the DHCP implementation was as close
to the RFC specifications as possible, the webserver should be able to work with any DHCP enabled router. The browser displayed the correct webpage (shown in Appendix C) when the IP address (in this case 192.168.2.185) of the webserver was keyed in and the temperature reporting function was also working. The TCP data retransmission protocol was also tested by setting the received ACK to always be a certain number less than the expected ACK and the webserver was able to resend this number of “lost bits” in the next packets. In addition, the total amount of flash memory used by this project was less than 50% of the available flash memory of the microcontroller which implies that there are 16 kilobytes of memory available for storing additional webpages and images. This project was demonstrated successfully to Dr. Land and fulfilled the requirements of building an embedded webserver with lab applications.

V) Conclusion

This project required detailed and extensive knowledge about the workings of computer networks as well as internet protocols. It also required some expertise in C programming. Having no prior knowledge of the former and mediocre experience in the latter, I was fortunate to have access to many reference books on the internet and source code. The fact that the RFCs were open source and easily available on the internet was also a great help in this project. The initial phase of the project was simply to familiarize myself with the internet protocols, computer networks, as well as programming in C. Despite several setbacks encountered early in the project such as failed attempts to adapt Adam Dunkel’s uIP open source code for the Mega32, I was able to start writing my own code, using an open source barebones RTL8019s driver code from EDTP as a reference. Debugging the project without having an actual internet interface was very frustrating since it was impossible to know what the microcontroller was doing with the packets. Fortunately, the decision to purchase the Ethernet controller from EDTP was made early and thus, I had enough time to figure out how to connect the hardware interface. Connecting and debugging the hardware took a considerable amount of time since the RTL8019s had very
little documentation. Once that was done, I could use Ethereal to check and debug the packets that were being assembled in the microcontroller.

Future improvements to the webserver could include a full implementation of TCP as well as more webpages and perhaps even picture and audio files. External flash memory could also be added to the webserver for added storage space. There might also be other innovative applications for this embedded webservers that could be implemented.

Through this project, I have gained immense knowledge and familiarity with internet protocols such as TCP/IP since I had to actually write code that executes the protocol. This project has also given me a glimpse on the workings of computer networks although that aspect of the webserver was almost wholly handled by the Ethernet controller.

VI) Acknowledgements

This project has been made possible with the support and guidance of Dr. Bruce R. Land. In addition to providing valuable information and help in C programming and the Mega 32, Dr. Land also helped to solder the individual components of the EDTP packetwacker together.

VII) References

1) Internet Architecture: An Introduction to IP Protocols, Uyless Black
2) TCP/IP LEAN Web Servers for Embedded Systems, Jeremy Bentham
3) Ethernet: the definitive guide, Charles E. Spurgeon.
4) http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/ethernet.htm#xtocid4
5) http://www.faqs.org/rfcs/rfc2132.html
6) http://www.avrfreaks.net/
7) http://www.embedded-creations.com/projects/uipAVR.html
8) http://www.edtp.com/
Appendix A: Code

/********************************************
This program was produced by the
CodeWizardAVR V1.23.7a Evaluation
Automatic Program Generator
© Copyright 1998-2002 HP InfoTech s.r.l.
http://www.hpinfotech.ro
e-mail:office@hpinfotech.ro

Project : AVRWEBSERVER
Version : 0.93 beta
Date    : 3/4/2004
Author  : Jeremy
Company :
Comments:

Chip type           : ATmega32
Program type        : Application
Clock frequency     : 16.000000 MHz
Memory model        : Small
Internal SRAM size  : 2048
External SRAM size  : 0
Data Stack size     : 512
********************************************/

/ * VERSION INFO
1) Added Checksum for incoming TCP packets (fixed)
2) Added TCP data sending function (only sends 1 packet at a time Window
   functionality should be done)
3) Increased packet size from 96 - 300 (since we can have 576 max packet
   length)
4) Added HTTP functions and HTTP sample
5) Added DHCP functionality
6) Need to tweak the TCP_close() functionality
7) Changed the rst pin to port B so that PORTC is totally free
7) Note: The webbrowsers use the RST function whenever it is closed. Don't
   think will need the TCP_close()
*****TESTING*****
6) DHCP WORKING!
7) IMCP working!!
7) HTTP up!!
8) TCP resend lost data working (tested)
8) (fixed)didn't do the setting of packets properly only set the 1st byte must do all bytes
*/

//******************************************************************
//* PORT MAP
//******************************************************************
// PORT C = rtldata - data bus RCTL8019 and AVR
// 0   SD0
// 1   SD1
// 2   SD2
// 3   SD3
// 4   SD4
// 5   SD5
// 6   SD6
// 7   SD7
// PORT B
// 0   SA0
// 1   SA1
// 2   SA2
// 3   SA3
// 4   SA4
// 5
// 6
// 7   make this the rst_pin
// PORT A
// temperature sensor port

// PORT D
// 0   RXD
// 1   TXD
// 2   INT0 --> for EEPROM only
// 3   EESK
// 4   EEDI
// 5   EEDO
// 6   ior_pin
// 7   iow_pin  */

#include <mega32.h>
#include <string.h>
#include <stdio.h>
#include <delay.h>
#include <stdlib.h>

#define ISO_G 0x47
#define ISO_E 0x45
#define ISO_T 0x54
#define ISO_slash 0x2f
#define ISO_c 0x63
#define ISO_g 0x67
#define ISO_i 0x69
#define ISO_space 0x20
#define ISO_nl 0x0a
#define ISO_cr 0x0d
#define ISO_a 0x61
#define ISO_t 0x74
#define ISO_hash 0x23
#define ISO_period 0x2e

// define the connection structure for a single TCP socket (multiple connections)

char flash *req_page[100];
unsigned int page_size;
flash char flash *index[71] = {
"HTTP/1.1 200 OK\r\n","Server: My MEng Project\r\n","Content-Type: text/html\r\n",
"<html>\r\n","<head>\r\n","<title>ECE MEng Project Cornell University 2003~4 Done by Jeremy</title>\r\n","<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">\r\n","<head>\r\n","<body bgcolor="#FFFFFF">\r\n","<h1><strong><font color="#0000FF" face="Arial, Helvetica, sans-serif">WELCOME TO ATMEL WEBSERVER</font></strong></h1>\r\n","<p>&nbsp;</p>\r\n","<p><strong><font color="#0000FF" face="Arial, Helvetica, sans-serif">This webserver is running entirely on a mega32 and using an EDTP packet wacker</font></strong></p>\r\n","<p><strong><font color="#0000FF" face="Arial, Helvetica, sans-serif">This project was done by Tzeming Tan, Jeremy supervised by Dr. Bruce R. Land.</font></strong></p>\r\n","<p>Cornell University</p>\r\n","<p>ATMEL embedded webserver</p>\r\n","<p>TCP/IP is the communications protocol most widely used for accessing the internet</p>\r\n","<p>today. The objective of this project is to introduce this protocol to the ATMEL</p>\r\n","<p>Mega 32 Microcontroller chip such that it can run as a simple webserver which</p>\r\n","}
can then be adapted for useful lab-based applications.

Although the Mega 32 and the development board will only be able to run limited web functions, it is a relatively inexpensive device compared to high power web servers. Therefore it can still have many applications for example, connecting the microcontroller to a thermometer which will display the current temperature on a browser. The current temperature is now: % degrees F.

The current temperature is now: % degrees F.

Link to ECE 476 website: http://instruct1.cit.cornell.edu/courses/ee476/

unsigned int size_index = 71;
unsigned int http_state = 0;
unsigned int sendflag = 0;
unsigned int pageendflag = 0;
char temperature = 0;
char temp[5];
float voltage; //scaled input voltage
unsigned int Ain;

FUNCTION PROTOTYPES

void http_server(void);
void tcp(void);
void tcp_close(void);
void assemble_ack(void);
void write_rtl(unsigned char regaddr, unsigned char regdata);
void read_rtl(unsigned char regaddr);
void get_packet(void);
void setipaddrs(void);
void cksum(void);
void echo_packet(void);
// x is the index number, y is the character number
void pack_html(unsigned int page, unsigned int x, unsigned int y);
#define INDEX 0
unsigned int dex,pos =0;
unsigned int rollback,counter = 0;
// end of pack_html function
void send_tcp_packet(void);
void arp(void);
void icmp(void);
void udp(void);
void udp_send(void);

// DHCP FUNCTIONS
void dhcp(void);
void dhcp_setip(void);

// Temperature function
void gettemp(void);

//******************************************************************
//* IP ADDRESS DEFINITION
//* This is the Ethernet Module IP address.
//* You may change this to any valid address.
//******************************************************************

unsigned char client[4];
unsigned char serverid[4];

//******************************************************************
//* HARDWARE (MAC) ADDRESS DEFINITION
//* This is the Ethernet Module hardware address.
//* You may change this to any valid address.
//******************************************************************

char MYMAC[6] = { 'J','e','s','t','e','r' };

//******************************************************************
//* Receive Ring Buffer Header Layout
//* This is the 4-byte header that resides infront of the
//* data packet in the receive buffer.
//******************************************************************

unsigned char pageheader[4];
#define enetpacketstatus 0x00
#define nextblock_ptr 0x01
#define enetpacketLenL 0x02
#define enetpacketLenH 0x03

//******************************************************************
//* Ethernet Header Layout
//******************************************************************

unsigned char packet[700]; //700 bytes of packet space
#define enetpacketDest0 0x00 //destination mac address
#define enetpacketDest1 0x01
#define enetpacketDest2 0x02
#define enetpacketDest3 0x03
#define enetpacketDest4 0x04
#define enetpacketDest5 0x05
#define enetpacketSrc0 0x06 //source mac address
#define enetpacketSrc1 0x07
#define enetpacketSrc2 0x08
#define enetpacketSrc3  0x09
#define enetpacketSrc4  0x0A
#define enetpacketSrc5  0x0B
#define enetpacketType0  0x0C //type/length field
#define enetpacketType1  0x0D
#define enetpacketData   0x0E //IP data area begins here

//******************************************************************
//* ARP Layout
//******************************************************************
#define arp_hwtype      0x0E
#define arp_prtype      0x10
#define arp_hwlen      0x12
#define arp_prlen      0x13
#define arp_op          0x14
#define arp_shaddr      0x16   //arp source mac address
#define arp_sipaddr      0x1C   //arp source ip address
#define arp_thaddr      0x20   //arp target mac address
#define arp_tipaddr      0x26   //arp target ip address

//******************************************************************
//* IP Header Layout
//******************************************************************
#define ip_vers_len      0x0E //IP version and header length
#define ip_tos           0x0F //IP type of service
#define ip_pktlen        0x10 //packet length
#define ip_id            0x12 //datagram id
#define ip_frag_offset    0x14 //fragment offset
#define ip_ttl           0x16 //time to live
#define ip_proto         0x17 //protocol (ICMP=1, TCP=6, UDP=11)
#define ip_hdr_cksum      0x18 //header checksum
#define ip_srcaddr       0x1A //IP address of source
#define ip_destaddr      0x1E //IP address of destination
#define ip_data           0x22 //IP data area

//******************************************************************
//* TCP Header Layout
//******************************************************************
#define TCP_srcport       0x22 //TCP source port
#define TCP_destport       0x24 //TCP destination port
#define TCP_seqnum        0x26 //sequence number
#define TCP_acknum        0x2A //acknowledgement number
#define TCP_hdrflags      0x2E //4-bit header len(DATA OFFSET) and flags
#define TCP_window        0x30 //window size
#define TCP_cksum         0x32 //TCP checksum
#define TCP_urgentptr     0x34 //urgent pointer
#define TCP_data               0x36  //option/data

//**************************************************************************
//* TCP Flags
//*   IN flags represent incoming bits
//*   OUT flags represent outgoing bits  576 octets(8 x bit) max datalength
//**************************************************************************

#define  FIN_IN               (packet[TCP_hdrflags+1] & 0x01)
#define  SYN_IN               (packet[TCP_hdrflags+1] & 0x02)
#define  RST_IN               (packet[TCP_hdrflags+1] & 0x04)
#define  PSH_IN               (packet[TCP_hdrflags+1] & 0x08)
#define  ACK_IN               (packet[TCP_hdrflags+1] & 0x10)
#define  URG_IN               (packet[TCP_hdrflags+1] & 0x20)
#define  FIN_OUT              packet[TCP_hdrflags+1] |= 0x01  //00000001
#define  NO_FIN        packet[TCP_hdrflags+1] &= 0x62  //00111110
#define  SYN_OUT              packet[TCP_hdrflags+1] |= 0x02  //00000010
#define  NO_SYN        packet[TCP_hdrflags+1] &= 0x61  //00111101
#define  RST_OUT              packet[TCP_hdrflags+1] |= 0x04  //00000100
#define  PSH_OUT              packet[TCP_hdrflags+1] |= 0x08  //00001000
#define  ACK_OUT              packet[TCP_hdrflags+1] |= 0x10  //00010000
#define  NO_ACK        packet[TCP_hdrflags+1] &= 0x47  //00101111
#define  URG_OUT              packet[TCP_hdrflags+1] |= 0x20  //00100000

//**************************************************************************
//*   Port Definitions
//*   This address is used by TCP for HTTP server function.
//*   This can be changed to any valid port number as long as
//*   you modify your code to recognize the new port number.
//**************************************************************************

#define  MY_PORT_ADDRESS      0x50  // 80 DECIMAL for internet

//**************************************************************************
//* IP Protocol Types
//**************************************************************************

#define  PROT_ICMP     0x01
#define  PROT_TCP     0x06
#define  PROT_UDP     0x11

//**************************************************************************
//*   ICMP Header
//**************************************************************************

#define  ICMP_type      ip_data
#define  ICMP_code      ICMP_type+1
#define  ICMP_cksum      ICMP_code+1
#define  ICMP_id       ICMP_cksum+2
#define  ICMP_seqnum      ICMP_id+2
#define  ICMP_data               ICMP_seqnum+2

//**************************************************************************
//*   UDP Header and DHCP headers
//**************************************************************************
#define UDP_srcport      ip_data
#define UDP_destport         UDP_srcport+2
#define UDP_len       UDP_destport+2
#define UDP_cksum      UDP_len+2
#define UDP_data      UDP_cksum+2
#define DHCP_op       UDP_cksum+2
#define DHCP_htype         DHCP_op+1
#define DHCP_hlen         DHCP_htype+1
#define DHCP_hops         DHCP_hlen+1
#define DHCP_xid         DHCP_hops+1
#define DHCP_secs         DHCP_xid+4
#define DHCP_flags         DHCP_secs+2
#define DHCP_ciaddr         DHCP_flags+2
#define DHCP_yiaddr         DHCP_ciaddr+4
#define DHCP_siaddr         DHCP_yiaddr+4
#define DHCP_giaddr         DHCP_siaddr+4
#define DHCP_chaddr         DHCP_giaddr+4
#define DHCP_sname         DHCP_chaddr+16
#define DHCP_file         DHCP_sname+64
#define DHCP_options         DHCP_file+128

//  DHCP states
#define DHCP_DIS       0
#define DHCP_OFF    1
#define DHCP_ACK    2
unsigned int dhcpstate = DHCP_DIS;

/***************************************************************************/
/* REALTEK CONTROL REGISTER OFFSETS */
/***************************************************************************/
#define CR    0x00
#define PSTART  0x01
#define PAR0       0x01    // Page 1
#define CR9346     0x01    // Page 3
#define PSTOP  0x02
#define BNRY  0x03
#define TSR   0x04
#define TPSR  0x04
#define TBCR0  0x05
#define NCR   0x05
#define TBCR1  0x06
#define ISR   0x07   // Page 1
#define RSAR0  0x08
#define CRDA0  0x08
#define RSAR1  0x09
```c
#define CRDAL 0x09
#define RBCR0 0x0A
#define RBCR1 0x0B
#define RSR 0x0C
#define RCR 0x0C
#define TCR 0x0D
#define CNTR0 0x0D
#define DCR 0x0E
#define CNTR1 0x0E
#define IMR 0x0F
#define CNTR2 0x0F
#define RDMAPORT 0x10
#define RSTPORT 0x18

//****************************************************************************
//* RTL8019AS INITIAL REGISTER VALUES
//****************************************************************************
#define rcrval 0x04
#define tcrval 0x00
#define dcrval 0x58 // was 0x48
#define imrval 0x11 // PRX and OVW interrupt enabled
#define txstart 0x40
#define rxstart 0x46
#define rxstop 0x60

//****************************************************************************
//* RTL8019AS DATA/ADDRESS PIN DEFINITIONS
//****************************************************************************
#define rtladdr PORTB
#define rtldata PORTC
#define tortl DDRC = 0xFF
#define fromrtl DDRC = 0x00

//****************************************************************************
//* RTL8019AS 9346 EEPROM PIN DEFINITIONS
//****************************************************************************
#define EESK 0x08 //PORTD3 00001000
#define EEDI 0x10 //PORTD4 00010000
#define EEDO 0x20 //PORTD5 00100000

//****************************************************************************
//* RTL8019AS PIN DEFINITIONS
//****************************************************************************
#define ior_pin 0x40 //PORTD6 01000000
#define iow_pin 0x80 //PORTD7 10000000
#define rst_pin 0x80 //PORTB7 10000000
#define INTO_pin 0x04 //PORTD2 00000100

//****************************************************************************
//* RTL8019AS ISR REGISTER DEFINITIONS
//****************************************************************************
```
#define  RST         0x80 //1000000
#define  RDC         0x40 //0100000
#define  OVW         0x10 //0001000
#define  PRX         0x01 //0000001

 AVR RAM Definitions

unsigned char aux_data[400];           //tcp received data area (200 char)
unsigned char req_ip[4];
int waitcount = 800;
unsigned char *addr,flags,last_line;
unsigned char byte_read,data_H,data_L;
unsigned char resend;
unsigned int i,t,txlen,rxlen,chksum16,hdrlen,tcpplen,tcpdatalen_in,dhcpoptlen;
unsigned int tcpdatalen_out,ISN,portaddr,ip_packet_len;
unsigned long
ic_chksum,hdr_chksum,my_seqnum,prev_seqnum,client_seqnum,incoming_ack,
expected_ack;

 Flags

#define synflag 0x01 //00000001
#define finflag 0x02 //00000010
#define synflag_bit flags & synflag
#define finflag_bit flags & finflag

 RTL8019AS PIN MACROS

#define set_ior_pin iorwport |= ior_pin
#define clr_ior_pin iorwport &= ~ior_pin
#define set_iow_pin iorwport |= iow_pin
#define clr_iow_pin iorwport &= ~iow_pin
#define set_rst_pin resetport |= rst_pin
#define clr_rst_pin resetport &= ~rst_pin
#define clr_EEDO eeprom &= ~EEDO
#define set_EEDO eeprom |= EEDO
#define clr_synflag flags &= ~synflag
#define set_synflag flags |= synflag
#define clr_finflag flags &= ~finflag
#define set_finflag flags |= finflag
#define set_packet32(d,s) packet[d] = make8(s,3); \n    packet[d+1] = make8(s,2); \n    packet[d+2] = make8(s,1); \n    packet[d+3]= make8(s,0);
// converts decimal into words (8bit)
#define make8(var,offset) (var >> (offset *8)) & 0xFF
// joins two 8bit binary into a 16bit binary and converts it to a decimal
#define make16(varhigh,varlow) ((varhigh & 0xFF)* 0x100) + (varlow & 
    0xFF)
// joins 4 8 bit numbers to form a 32 bit number
#define make32(var1,var2,var3,var4) \n    ((unsigned long)var1<<24)+((unsigned long)var2<<16)+ \n    ((unsigned long)var3<<8)+((unsigned long)var4)

//******************************************************************
// timer interrupt
//******************************************************************
interrupt [TIM0_COMP] void t0_cmp(void)
{
    waitcount--;
    if (waitcount < 0)
    {
        waitcount = 9000;
    }
}
//******************************************************************
// Application Code
//******************************************************************
int main()
{
    // Getty Server
    http_server()
    
    return 0;
}
http_state = 1;
//get the sample
   //The sleep statement lowers digital noise
   //and starts the A/D conversion
#asm
   sleep
#endasm
gettemp();
// send the http
// check which file client wants
// set the dataptr to the file
if(sendflag == 0 && pageendflag == 0)
{
    if(rollback)
    {
        // start from beginning again
        dex=0;
        pos=0;
    }
    sendflag = 1;
    pack_html(INDEX,dex,pos);
    counter = counter+tcpdatalen_out;
    if(pageendflag == 1)
        set_finflag;
    send_tcp_packet();
    rollback=0;
}
    // the send operation has been completed
else if(pageendflag == 1)
{
    pageendflag = 0;
    dex=0;
    pos=0;
    counter = 0;
    rollback = 0;
    http_state = 0;
}

//******************************************************************
//*   Get Temperature
//*
//*
//*
//******************************************************************
void gettemp()
{
    voltage = (float)Ain;
    voltage = (voltage/256)*2.6 ; //(fraction of full scale)*Aref
    voltage = voltage/0.02;
    ftoa(voltage,3,temp);
}

interrupt [ADC_INT] void adc_done(void)
{
    Ain = ADCH;
}

//****************************************************************************
//* Perform ARP Response
//* This routine supplies a requesting computer with the
//* Ethernet module's MAC (hardware) address.
//****************************************************************************
void arp()
{
    //start the NIC
    write_rtl(CR,0x22);

    //load beginning page for transmit buffer
    write_rtl(TPSR,txstart);

    //set start address for remote DMA operation
    write_rtl(RSAR0,0x00);
    write_rtl(RSAR1,0x40);

    //clear the Interrupts
    write_rtl(ISR,0xFF);

    //load data byte count for remote DMA
    write_rtl(RBCR0,0x3C);
    write_rtl(RBCR1,0x00);

    //do remote write operation
    write_rtl(CR,0x12);

    //write destination MAC address
    for(i=0;i<6;++i)
        write_rtl(RDMAPORT,packet[enetpacketSrc0+i]);

    //write source MAC address
    for(i=0;i<6;++i)
        write_rtl(RDMAPORT,MYMAC[i]);
// write typelen hwtype prtype hwlen prlen op:
addr = &packet[enetpacketType0];
packet[arp_op+1] = 0x02;
for(i=0;i<10;++i)
    write_rtl(RDMAPORT,*addr++);

// write ethernet module MAC address
for(i=0;i<6;++i)
    write_rtl(RDMAPORT,MYMAC[i]);

// write ethernet module IP address
for(i=0;i<4;++i)
    write_rtl(RDMAPORT,MYIP[i]);

// write remote MAC address
for(i=0;i<6;++i)
    write_rtl(RDMAPORT,packet[enetpacketSrc0+i]);

// write remote IP address
for(i=0;i<4;++i)
    write_rtl(RDMAPORT,packet[arp_sipaddr+i]);

// write some pad characters to fill out the packet to
// the minimum length
for(i=0;i<0x12;++i)
    write_rtl(RDMAPORT,0x00);

// make sure the DMA operation has successfully completed
byte_read = 0;
while(!(byte_read & RDC))
    read_rtl(ISR);

// load number of bytes to be transmitted
write_rtl(TBCR0,0x3C);
write_rtl(TBCR1,0x00);

// send the contents of the transmit buffer onto the network
write_rtl(CR,0x24);
}

//******************************************************************
// Perform ICMP Function
// This routine responds to a ping.
//******************************************************************
void icmp()
{
    // set echo reply
packet[ICMP_type]=0x00;
packet[ICMP_code]=0x00;

//clear the ICMP checksum
packet[ICMP_cksum]=0x00;
packet[ICMP_cksum+1]=0x00;

//setup the IP header
setipaddrs();

//calculate the ICMP checksum
hdr_chksum = 0;
hdrlen = (make16(packet[ip_pktlen],packet[ip_pktlen+1])) - \n((packet[ip_vers_len] & 0x0F) * 4);
addr = &packet[ICMP_type];
cksum();
chksum16= ~(hdr_chksum + ((hdr_chksum & 0xFFFF0000) >> 16));
packet[ICMP_cksum] = make8(chksum16,1);
packet[ICMP_cksum+1] = make8(chksum16,0);

//send the ICMP packet along on its way
echo_packet();
}
//******************************************************************
//*    UDP Function    (To be used with DHCP)
//*    UDP_srcport = 0, destination is either 67 or 68 IP is
//*    0000000 and 255.255.255.255.255
//******************************************************************
void udp()
{
    //use port 68 for DHCP
    if(packet[UDP_destport] == 0x00 && packet[UDP_destport+1] == 0x44)
    {
        ic_chksum = make16(packet[UDP_cksum],packet[UDP_cksum+1]);
        //calculate the UDP checksum
        packet[UDP_cksum] = 0x00;
        packet[UDP_cksum+1] = 0x00;

        hdr_chksum = 0;
        hdrlen = 0x08;
        addr = &packet[ip_srcaddr];
        cksum();
        hdr_chksum = hdr_chksum + packet[ip_proto];
        hdrlen = 0x02;
        addr = &packet[UDP_len];
        cksum();
    }
hdrlen = make16(packet[UDP_len], packet[UDP_len+1]);
addr = &packet[UDP_srcport];
cksum();
chksum16= ~ (hdr_chksum + ((hdr_chksum & 0xFFFF0000) >> 16));
// perform checksum
if (chksum16 == ic_chksum)
    dhcp();

}

void udp_send()
{
    ip_packet_len = 20 + make16(packet[UDP_len], packet[UDP_len+1]);

    packet[ip_pktlen] = make8(ip_packet_len, 1);
    packet[ip_pktlen+1] = make8(ip_packet_len, 0);
    packet[ip_proto] = PROT_UDP;

    // calculate the IP header checksum
    packet[ip_hdr_cksum] = 0x00;
    packet[ip_hdr_cksum+1] = 0x00;
    hdr_chksum = 0;
    chksum16 = 0;
    hdrlen = (packet[ip_vers_len] & 0x0F) * 4;
    addr = &packet[ip_vers_len];
    cksum();
    chksum16 = ~ (hdr_chksum + ((hdr_chksum & 0xFFFF0000) >> 16));
    packet[ip_hdr_cksum] = make8(chksum16, 1);
    packet[ip_hdr_cksum+1] = make8(chksum16, 0);

    // set the source port to 68(client)
    packet[UDP_srcport] = 0x00;
    packet[UDP_srcport+1] = 0x44;

    // set the destination port to 67(server)
    packet[UDP_destport] = 0x00;
    packet[UDP_destport+1] = 0x43;

    // calculate the UDP checksum
    packet[UDP_cksum] = 0x00;
    packet[UDP_cksum+1] = 0x00;

    hdr_chksum = 0;
    hdrlen = 0x08;
    addr = &packet[ip_srcaddr];
cksum();
hdr_chksum = hdr_chksum + packet[ip_proto];
hdrlen = 0x02;
addr = &packet[UDP_len];
cksum();
hdrlen = make16(packet[UDP_len],packet[UDP_len+1]);
addr = &packet[UDP_srcport];
cksum();
chksum16 = ~(hdr_chksum + ((hdr_chksum & 0xFFFF0000) >> 16));
packet[UDP_cksum] = make8(chksum16,1);
packet[UDP_cksum+1] = make8(chksum16,0);

txlen = ip_packet_len + 14;
  // transmit length
  if(txlen < 60)
    txlen = 60;
  data_L = make8(txlen,0);
  data_H = make8(txlen,1);
  write_rtl(CR,0x22);
read_rtl(CR);
  while( byte_read & 0x04 )
    read_rtl(CR);
  write_rtl(TPSR,txstart);
  write_rtl(RSAR0,0x00);
  write_rtl(RSAR1,0x40);
  write_rtl(ISR,0xFF);
  write_rtl(RBCR0,data_L);
  write_rtl(RBCR1,data_H);
  write_rtl(CR,0x12);
  // the actual send operation
  for(i=0;i<txlen;++i)
    write_rtl(RDMAPORT,packet[enetpacketDest0+i]);
  byte_read = 0;
  while(!(byte_read & RDC))
    read_rtl(ISR);
  write_rtl(TBCR0,data_L);
  write_rtl(TBCR1,data_H);
  write_rtl(CR,0x24);

}

void dhcp_setip()
{
  //build the IP header
  //destination ip = 255.255.255.255
  packet[ip_destaddr]=0xFF;
  packet[ip_destaddr+1]=0xFF;
}
packet[ip_destaddr+2]=0xFF;
packet[ip_destaddr+3]=0xFF;
//source IP = 0.0.0.0
packet[ip_srcaddr]=0;
packet[ip_srcaddr+1]=0;
packet[ip_srcaddr+2]=0;
packet[ip_srcaddr+3]=0;
//you don't know the destination MAC
packet[enetpacketDest0]=255;
packet[enetpacketDest1]=255;
packet[enetpacketDest2]=255;
packet[enetpacketDest3]=255;
packet[enetpacketDest4]=255;
packet[enetpacketDest5]=255;
//make ethernet module mac address the source address
packet[enetpacketSrc0]=MYMAC[0];
packet[enetpacketSrc1]=MYMAC[1];
packet[enetpacketSrc2]=MYMAC[2];
packet[enetpacketSrc3]=MYMAC[3];
packet[enetpacketSrc4]=MYMAC[4];
packet[enetpacketSrc5]=MYMAC[5];
//calculate IP packet length done by the respective protocols
packet[enetpacketType0] = 0x08;
packet[enetpacketType1] = 0x00;
//set IP header length to 20 bytes
packet[ip_vers_len] = 0x45;
// 1st step in getting an IP address
}

//*****************************************************************
// DHCP for obtaining IP from the router port 67~68 using UDP
//*****************************************************************
void dhcp()
{

    if(dhcpstate == DHCP_DIS)
    {

        // listen to broadcast
        for(i=0;i<4;i++)
            MYIP[i] = 255;
packet[DHCP_op] = 1;
packet[DHCP_htype] = 1;
packet[DHCP_hlen] = 6;
packet[DHCP_hops] = 0;
packet[DHCP_xid] = make8(0x31257A1D,3);
packet[DHCP_xid+1] = make8(0x31257A1D,2);
packet[DHCP_xid+2] = make8(0x31257A1D,1);
packet[DHCP_xid+3] = make8(0x31257A1D,0);
for(i=DHCP_secs;i<DHCP_chaddr;i++)
    packet[i] = 0;
for(i=0;i<6;i++)
    packet[DHCP_chaddr+i] = MYMAC[i];
for(i=0;i<10;i++)
    packet[DHCP_chaddr+6+i] = 0;
for(i=0;i<192;i++)
    packet[DHCP_sname+i]=0;
// magic cookie
packet[DHCP_options] = 99;
packet[DHCP_options+1] = 130;
packet[DHCP_options+2] = 83;
packet[DHCP_options+3] = 99;
// message type
packet[DHCP_options+4] = 53;
packet[DHCP_options+5] = 1;
// DHCP_DISCOVER
packet[DHCP_options+6] = 1;
// Client identifier
packet[DHCP_options+7] = 61;
packet[DHCP_options+8] = 7;
packet[DHCP_options+9] = 1;
for(i=0;i<6;i++)
    packet[DHCP_options+10+i] = MYMAC[i];
// END OPTIONS
packet[DHCP_options+16] = 255;
// length of UDP datagram = 8 bytes; length of DHCP data = 236 bytes + options
dhcpoptlen = 17;
packet[UDP_len]= make8(244+dhcpoptlen,1);
packet[UDP_len+1]= make8(244+dhcpoptlen,0);
dhcp_setup();
udp_send();
for(i=0;i<4;i++)
    MYIP[i]=255;
DHCP_wait = 1;
// wait for DHCP offer
dhcpstate = DHCP_OFF;
}
// if we have an offer from the server
if(dhcpstate == DHCP_OFF) // && packet[ip_srcaddr] &&
{
    // check transaction id and message type

if(DHCP_wait == 2)\((\text{make32}(\text{packet}[\text{DHCP}\_\text{xid}], \text{packet}[\text{DHCP}\_\text{xid}+1], \text{packet}[\text{DHCP}\_\text{xid}+2], \text{packet}[\text{DHCP}\_\text{xid}+3])) == 0x31257A1D)\&\& (\text{packet}[\text{DHCP}\_\text{options}+4] == 53)\&\& (\text{packet}[\text{DHCP}\_\text{options}+5] == 1)\&\& (\text{packet}[\text{DHCP}\_\text{options}+6] == 2))\)\}
{
  if(DHCP_wait == 1)
  for(i=0;i<4;i++)
  {
    req_ip[i] = packet[DHCP\_yiaddr+i];
    serverid[i] = packet[ip_srcaddr+i];
  }

  // stop resending discover
  DHCP_wait=2;
}

// listen to broadcast
for(i=0;i<4;i++)
  MYIP[i] = 255;
// assemble DHCP\_req
packet[DHCP\_op] = 1;
packet[DHCP\_htype] = 1;
packet[DHCP\_hlen] = 6;
packet[DHCP\_hops] = 0;
packet[DHCP\_xid] = make8(0x31257A1D,3);
packet[DHCP\_xid+1] = make8(0x31257A1D,2);
packet[DHCP\_xid+2] = make8(0x31257A1D,1);
packet[DHCP\_xid+3] = make8(0x31257A1D,0);
for(i=DHCP\_secs;i<DHCP\_yiaddr;i++)
  packet[i] = 0;
for(i=DHCP\_siaddr;i<DHCP\_chaddr;i++)
  packet[i] = 0;
for(i=0;i<6;i++)
  packet[DHCP\_chaddr+i] = MYMAC[i];
for(i=0;i<10;i++)
  packet[DHCP\_chaddr+6+i] = 0;
for(i=0;i<192;i++)
  packet[DHCP\_sname+i]=0;
// magic cookie
packet[DHCP\_options] = 99;
packet[DHCP\_options+1] = 130;
packet[DHCP\_options+2] = 83;
packet[DHCP\_options+3] = 99;
// message type
packet[DHCP\_options+4] = 53;
packet[DHCP\_options+5] = 1;
// DHCP\_REQUEST
packet[DHCP\_options+6] = 3;
// Client identifier
packet[DHCP_options+7] = 61;
packet[DHCP_options+8] = 7;
packet[DHCP_options+9] = 1;
for(i=0;i<6;i++)
    packet[DHCP_options+10+i] = MYMAC[i];
// Requested IP address
packet[DHCP_options+16] = 50;
packet[DHCP_options+17] = 4;
for(i=0;i<4;i++)
{
    packet[DHCP_options+18+i] = req_ip[i];
}
for(i=0;i<4;i++)
    packet[DHCP_yiaddr+i]=0;
// server ID
packet[DHCP_options+22] = 54;
packet[DHCP_options+23] = 4;
for(i=0;i<4;i++)
{
    packet[DHCP_options+24+i] = serverid[i];
}
// END OPTIONS
packet[DHCP_options+28] = 255;
// length of UDP datagram = 8bytes; length of DHCP data = 236 bytes+ options
dhcppoptlen = 29;
packet[UDP_len]= make8(244+dhcppoptlen,1);
packet[UDP_len+1]= make8(244+dhcppoptlen,0);
// make a DHCP request
dhcp_setip();
udp_send();
// wait for DHCP ACK
dhcpstate = DHCP_ACK;
}
if((dhcpstate == DHCP_ACK) && (packet[ip_srcaddr] == serverid[0]) &&
    (packet[ip_srcaddr+1] == serverid[1]) && (packet[ip_srcaddr+2] == serverid[2])
    && (packet[ip_srcaddr+3] == serverid[3]))
{
    // check if message type is an ack
if((make32(packet[DHCP_xid],packet[DHCP_xid+1],packet[DHCP_xid+2],
packet[DHCP_xid+3]) == 0x31257A1D) && (packet[DHCP_options+4] ==
53) && (packet[DHCP_options+5] == 1) && (packet[DHCP_options+6] == 5)) {
    DHCP_wait = 0;
    //take the IP address
    for(i=0;i<4;i++)
        MYIP[i] = packet[DHCP_yiaddr+i];
}

//******************************************************************
//* TCP Function
//* This function uses TCP protocol to interface with the browser
//* using well known port 80. The application function is called with
//* every incoming character.
//******************************************************************
void tcp()
{
    //assemble the destination port address (my) from the incoming packet
    portaddr = make16(packet[TCP_destport],packet[TCP_destport+1]);
    //calculate the length of the data coming in with the packet
    //incoming tcp header length
tcplen = ip_packet_len - ((packet[ip_vers_len] & 0x0F) * 4);
    //incoming data length =
tcpdatalen_in = (make16(packet[ip_pktlen],packet[ip_pktlen+1]))- \
    ((packet[ip_vers_len] & 0x0F)* 4)-((packet[TCP_hdrflags] & 0xF0) >> 4) * 4);

    // convert the entire packet into a checksum
    // checksum of entire datagram
    ic_chksum = make16(packet[TCP_cksum],packet[TCP_cksum+1]);
    packet[TCP_cksum] = 0x00;
    packet[TCP_cksum+1] = 0x00;
    hdr_chksum = 0;
    hdrhlen = 0x08;
    addr = &packet[ip_srcaddr];
    cksum();
    hdr_chksum = hdr_chksum + packet[ip_proto];
    hdr_chksum = hdr_chksum + tcplen;
    hdrhlen = tcplen;
    addr = &packet[TCP_srcport];
    cksum();
    checksum16= ~(hdr_chksum + ((hdr_chksum & 0xFFFF0000) >> 16));
if((chksum16 == icchksum)&&(portaddr==MY_PORT_ADDRESS))
{
    // The webserver can only connect to one client at a time.
    
    /* ---------------------3-way handshake---------------------------*/
    //this code segment processes the incoming SYN from the client
    //and sends back the initial sequence number (ISN) and
    acknowledges
    //the incoming SYN packet (step 1 and 2 of 3 way handshake)
    if(SYN_IN && portaddr == MY_PORT_ADDRESS)
    {
        dex=0;
pos=0;
tcpdatalen_in = 0x01;
tcpdatalen_out = 0;
set_synflag;
client[0] = packet[ip_srcaddr];
client[1] = packet[ip_srcaddr+1];
client[2] = packet[ip_srcaddr+2];
client[3] = packet[ip_srcaddr+3];
    // build IP header switch the dest and src IPs
setipaddrs();
    // set the header field to 24 bytes(MSS options)
    // packet[TCP_hdrflags] = (0x6 << 4) & 0xF0;
    // set the ports
data_L = packet[TCP_srcport];
packet[TCP_srcport] = packet[TCP_destport];
packet[TCP_destport] = data_L;

data_L = packet[TCP_srcport+1];
packet[TCP_srcport+1] = packet[TCP_destport+1];
packet[TCP_destport+1] = data_L;
    // ack = SEQ_IN + 1
assemble_ack();
    // if the seqnum overflows (>16bits)
if(++ISN == 0x0000 || ++ISN == 0xFFFF)
    my_seqnum = 0x1234FFFF;
//expected acknowledgement
expected_ack = my_seqnum+1;
    set_packet32(TCP_seqnum,my_seqnum);
packet[TCP_hdrflags+1] = 0x00;
SYN_OUT;
ACK_OUT;

packet[TCP_cksum] = 0x00;
packet[TCP_cksum+1] = 0x00;

hdr_chksum = 0;
hdrlen = 0x08;
addr = &packet[ip_srcaddr];
cksum();
hdr_chksum = hdr_chksum + packet[ip_proto];
tcplen = make16(packet[ip_pktlen], packet[ip_pktlen+1]) -
((packet[ip_vers_len] & 0x0F) * 4);
hdr_chksum = hdr_chksum + tcplen;
hdrlen = tcplen;
addr = &packet[TCP_srcport];
cksum();
chksum16 = ~hdr_chksum + ((hdr_chksum &
0xFFFF0000) >> 16));

// write the checksum into the packet
packet[TCP_cksum] = make8(chksum16, 1);
packet[TCP_cksum+1] = make8(chksum16, 0);
// send the packet with the same data it came with
echo_packet();

// if we are waiting for an ack or waiting for data from the client we
are connected to
{
    //If an ACK is received
    if(ACK_IN)
    {
        //assemble the acknowledgment number from the
incoming packet
        incoming_ack
        = make32(packet[TCP_acknum], packet[TCP_acknum+1], \
                packet[TCP_acknum+2], packet[TCP_acknum+3]);
        if(incoming_ack==expected_ack)

{
    my_seqnum = incoming_ack;
    //if it is the result of a close operations
    
    // if the client is the one who initiated the close
    
    if(closeflag==2)
        closeflag = 0;
    else if(closeflag==1)
        closeflag = 2;

    if(synflag_bit)
    {
        clr_synflag;
        // next step is to wait for a "get" request
    }
    
    if(tcpdatalen_in)
    {
        // if the packet is more than we can handle, we
        // and then ack the 200 bytes so that the client
        can resend the excluded data
        
        if(tcpdatalen_in > 400)
            tcpdatalen_in = 400;
        ackflag=1;
        http_server();
        // wait for ack
    }
    else
    {
        if(sendflag == 1)
        {
            sendflag = 0;
            ackflag=1;
            //send next batch of data
            http_server();
        }
    }
}
else if(incoming_ack<expected_ack)
{
    my_seqnum = expected_ack - (expected_ack - incoming_ack);
    sendflag = 0;
    ackflag=1;
    pageendflag = 0;
}
rollback = 1;
counter = counter - (expected_ack - incoming_ack);
//resend data
http_server();
}
}
if(FIN_IN)
{
    ackflag = 1;
send_tcp_packet();
    if(closeflag == 0)
    {
        closeflag = 1;
tcp_close();
    }
    else if(closeflag == 2)
    {
        closeflag = 0;
    }
}
}
/* ---------------TCP CLOSE CONNECTION FUNCTION-------------------- */
void tcp_close()
{
    set_finflag;
tcpdatalen_out=0;
send_tcp_packet();
closeflag=closeflag+1;
}

//******************************************************************
//* Assemble the Acknowledgment
//* This function assembles the acknowledgment to send to
//* to the client by adding the received data count to the
//* client's incoming sequence number.
//******************************************************************
void assemble_ack()
{
client_seqnum = make32(packet[TCP_seqnum], packet[TCP_seqnum+1],
packet[TCP_seqnum+2], packet[TCP_seqnum+3]);
client_seqnum = client_seqnum + tcpdatalen_in;
set_packet32(TCP_acknum, client_seqnum);
}

//******************************************************************
//* Send TCP Packet
//*   This routine assembles and sends a complete TCP/IP packet.
//*   40 bytes of IP and TCP header data is assumed.(no options)
//******************************************************************
void send_tcp_packet()
{
    //count IP and TCP header bytes.. Total = 40 bytes
    if(tcpdatalen_out == 0)
    {
        tcpdatalen_out = 14;
        for(i=0;i<14;i++)
            packet[TCP_data+i]=0;
        expected_ack=my_seqnum+1;
    }
    else
    {
        expected_ack=my_seqnum+tcpdatalen_out;
    }
    ip_packet_len = 40 + tcpdatalen_out;
    packet[ip_pktlen] = make8(ip_packet_len,1);
    packet[ip_pktlen+1] = make8(ip_packet_len,0);
    packet[ip_proto] = PROT_TCP;
    setipaddrs();
    data_L = packet[TCP_srcport];
    packet[TCP_srcport] = packet[TCP_destport];
    packet[TCP_destport] = data_L;
    data_L = packet[TCP_srcport+1];
    packet[TCP_srcport+1] = packet[TCP_destport+1];
    packet[TCP_destport+1] = data_L;
    assemble_ack();
    set_packet32(TCP_seqnum, my_seqnum);

    packet[TCP_hdrflags+1] = 0x00;
    if(ackflag ==1)
        ACK_OUT;
    else
        NO_ACK;
    ackflag=0;
    if(flags & finflag)
FIN_OUT;
clr_finflag;
}

packet[TCP_cksum] = 0x00;
packet[TCP_cksum+1] = 0x00;

hdr_chksum = 0;
hdrlen = 0x08;
addr = &packet[ip_srcaddr];
cksum();
hdr_chksum = hdr_chksum + packet[ip_proto];
tcplen = ip_packet_len - ((packet[ip_vers_len] & 0x0F) * 4);
hdr_chksum = hdr_chksum + tcplen;
hdrlen = tcplen;
addr = &packet[TCP_srcport];
cksum();
chksum16 = ~(hdr_chksum + ((hdr_chksum & 0xFFFF0000) >> 16));
packet[TCP_cksum] = make8(chksum16, 1);
packet[TCP_cksum+1] = make8(chksum16, 0);

txlen = ip_packet_len + 14;
if(txlen < 60)
  txlen = 60;
data_L = make8(txlen, 0);
data_H = make8(txlen, 1);
write_rtl(CR, 0x22);
read_rtl(CR);
while(byte_read & 0x04)
  read_rtl(CR);
write_rtl(TPSR, txstart);
write_rtl(RSAR0, 0x00);
write_rtl(RSAR1, 0x40);
write_rtl(ISR, 0xFF);
write_rtl(RBCR0, data_L);
write_rtl(RBCR1, data_H);
write_rtl(CR, 0x12);

for(i=0;i<txlen;++i)
  write_rtl(RDMAPORT, packet[enetpacketDest0+i]);

byte_read = 0;
while(!(byte_read & RDC))
  read_rtl(ISR);

write_rtl(TBCR0, data_L);
write_rtl(TBCR1, data_H);
write_rtl(CR, 0x24);
}

// for sending the html
void pack_html(unsigned int page, unsigned int x, unsigned int y)
{
    if(page == INDEX)
    {
        page_size = size_index;
        // get the required page
        for(i=0; i< size_index; i++)
            req_page[i]=index[i];
    }

tcpdatalen_out=0;
i=0;
t=0;
while(x<page_size)
{

    while(*(req_page[x]+y)!=0x00 && i<500)
    {
        strncpyf(&packet[TCP_data+i], req_page[x]+y, 1);
        y=y+1;
        if(packet[TCP_data+i] == 0x27)
        {
            packet[TCP_data+i] = 0x22;
            x=x+1;
            y=0;
        }
        if(packet[TCP_data+i] == 0x25)
        {
            // sprintf(temp,"%d", temperature);
            // if there is enough space to send the temperature
            if((i + 5)<500)
            {
                while(t<5)
                {
                    packet[TCP_data+i]=temp[t];
                    temp[t] = 0;
                    i=i+1;
                    t=t+1;
                    tcpdatalen_out=tcpdatalen_out+1;
                }
                i=i-1;
                x=x+1;
                y=0;
            }
        }
    }
}

}
tcpdatalen_out=tcpdatalen_out-1;
}
else
{
  i=500; // exit the loop
  tcpdatalen_out=tcpdatalen_out-1;
}

if(rollback && counter>=rollback)
{
  rollback=rollback+1;
}
else
{
  i=i+1;
  tcpdatalen_out=tcpdatalen_out+1;
}

if(i<500)
{
  x=x+1;
  dex = x;
  y=0 ;
}
// max size of packet reached
else
{
  // save for sending next packet
  dex = x;
  pos = y;
  // get out of loop
  x=page_size+1;
}
if(dex >= page_size)
  pageendflag = 1;

//******************************************************************
//* Write to NIC Control Register
//******************************************************************

void write_rtl(unsigned char regaddr, unsigned char regdata)
{
  // write the regaddr into PORTB
  rtladdr = regaddr;
tortl;
// write the data into PORTC
rtl_data = regdata;
#if
  nop
#endif
// toggle write pin
clr_iow_pin;
#if
  nop
  nop
  nop
#endif
set_iow_pin;
#if
  nop
#endif
// set data port back to input
fromrtl;
PORTC = 0xFF;
}

//******************************************************************
//* Read From NIC Control Register
//******************************************************************

void read_rtl(unsigned char regaddr)
{
  fromrtl;
  PORTC = 0xFF;
  rtladdr = regaddr;
  // assert read
clr_ior_pin;
#if
  nop
#endif
#if
  nop
  nop
  nop
#endif
byte_read = PINC;
set_ior_pin;
#if
  nop
#endif
}
/** HANDLE RECEIVE_RING_BUFFER_OVERRUN **/

Handle Receive Ring Buffer Overrun

No packets are recovered

void overrun()
{
    read_rtl(CR);
    data_L = byte_read;
    write_rtl(CR,0x21);
    delay_ms(2);
    write_rtl(RBCR0,0x00);
    write_rtl(RBCR1,0x00);
    if(!(data_L & 0x04))
        resend = 0;
    else if(data_L & 0x04)
    {
        read_rtl(ISR);
        data_L = byte_read;
        if((data_L & 0x02) || (data_L & 0x08))
            resend = 0;
        else
            resend = 1;
    }
    write_rtl(TCR,0x02);
    write_rtl(CR,0x22);
    write_rtl(BNRY,rxstart);
    write_rtl(CR,0x62);
    write_rtl(CURR,rxstart);
    write_rtl(CR,0x22);
    write_rtl(ISR,0x10);
    write_rtl(TCR,tcrval);
}

Echo Packet Function

This routine does not modify the incoming packet size and thus echoes the original packet structure.

void echo_packet()
{
    write_rtl(CR,0x22);
    write_rtl(TPSR,txstart);
    write_rtl(RSAR0,0x00);
    write_rtl(RSAR1,0x40);
    write_rtl(ISR,0xFF);
    write_rtl(RBCR0,pageheader[enetpacketLenL] - 4 );
}
write_rtl(RBCR1, pageheader[enetpacketLenH]);
write_rtl(CR, 0x12);

txlen = make16(pageheader[enetpacketLenH], pageheader[enetpacketLenL]) - 4;
for(i=0;i<txlen;++i)
    write_rtl(RDMAPORT, packet[enetpacketDest0+i]);

byte_read = 0;
while(!(byte_read & RDC))
    read_rtl(ISR);

write_rtl(TBCR0, pageheader[enetpacketLenL] - 4);
write_rtl(TBCR1, pageheader[enetpacketLenH]);
write_rtl(CR, 0x24);

//******************************************************************
//*       Get A Packet From the Ring
//*   This routine removes a data packet from the receive buffer
//*   ring.
//******************************************************************
void get_packet()
{
    //execute Send Packet command to retrieve the packet
    write_rtl(CR, 0x1A);
    for(i=0;i<4;++i)
    {
        read_rtl(RDMAPORT);
        pageheader[i] = byte_read;
    }

    rxlen = make16(pageheader[enetpacketLenH], pageheader[enetpacketLenL]);
    for(i=0;i<rxlen;++i)
    {
        read_rtl(RDMAPORT);
        //dump any bytes that will overrun the receive buffer(which is probably > 1kbyte)
        if(i < 700)
            packet[i] = byte_read;
    }
    while(!(byte_read & RDC))
        read_rtl(ISR);

    write_rtl(ISR, 0xFF);
//process an ARP packet
if(packet[enetpacketType0] == 0x08 && packet[enetpacketType1] == 0x06)
{
    if(packet[arp_hwtype+1] == 0x01 &&
        packet[arp_prtype] == 0x08 && packet[arp_prtype+1] == 0x00 &&
        packet[arp_hwlen] == 0x06 && packet[arp_prlen] == 0x04 &&
        packet[arp_op+1] == 0x01 &&
        MYIP[0] == packet[arp_tipaddr] &&
        MYIP[3] == packet[arp_tipaddr+3])
    {
        arp();
    }
}

//process an IP packet
else if(packet[enetpacketType0] == 0x08 && packet[enetpacketType1] == 0x00
    && packet[ip_destaddr] == MYIP[0]
    && packet[ip_destaddr+1] == MYIP[1]
    && packet[ip_destaddr+3] == MYIP[3])
{
    //do a checksum of the ipheader
    ic_chksum = make16(packet[ip_hdr_cksum],packet[ip_hdr_cksum+1]);
    packet[ip_hdr_cksum]=0x00;
    packet[ip_hdr_cksum+1]=0x00;
    hdr_chksum =0;
    hdrlen = (packet[ip_vers_len] & 0x0F) * 4;
    addr = &packet[ip_vers_len];
    cksum();
    chksum16 = ~(hdr_chksum + ((hdr_chksum & 0xFFFF0000) >> 16));
    if(chksum16 == ic_chksum)
    {
        packet[ip_hdr_cksum]=make8(ic_chksum,1);
        packet[ip_hdr_cksum+1]=make8(ic_chksum,0);
        //Find the IP packet length
        ip_packet_len = make16(packet[ip_pktlen],packet[ip_pktlen+1]);
        //response to packet here
        if(packet[ip_proto] == PROT_ICMP)
            icmp();
        else if(packet[ip_proto] == PROT_UDP)
            udp();
        else if(packet[ip_proto] == PROT_TCP)
            tcp();
    }
}
void setipaddrs()
{
    packet[enetpacketType0] = 0x08;
    packet[enetpacketType1] = 0x00;
    /*client[0] = packet[ip_srcaddr];
     client[1] = packet[ip_srcaddr+1];
     client[2] = packet[ip_srcaddr+2];
     client[3] = packet[ip_srcaddr+3];
     */
    //move IP source address to destination address
    packet[ip_destaddr]=client[0];
    packet[ip_destaddr+1]=client[1];
    packet[ip_destaddr+2]=client[2];
    packet[ip_destaddr+3]=client[3]; /*
     */
    //move IP source address to destination address
    packet[ip_destaddr]=packet[ip_srcaddr];
    packet[ip_destaddr+1]=packet[ip_srcaddr+1];
    packet[ip_destaddr+2]=packet[ip_srcaddr+2];
    packet[ip_destaddr+3]=packet[ip_srcaddr+3];
    //make ethernet module IP address source address
    packet[ip_srcaddr]=MYIP[0];
    packet[ip_srcaddr+1]=MYIP[1];
    packet[ip_srcaddr+2]=MYIP[2];
    packet[ip_srcaddr+3]=MYIP[3];
    //move hardware source address to destinatin address
    packet[enetpacketDest0]=packet[enetpacketSrc0];
    packet[enetpacketDest1]=packet[enetpacketSrc1];
    packet[enetpacketDest2]=packet[enetpacketSrc2];
    packet[enetpacketDest3]=packet[enetpacketSrc3];
    packet[enetpacketDest4]=packet[enetpacketSrc4];
    packet[enetpacketDest5]=packet[enetpacketSrc5];
    //make ethernet module mac address the source address
    packet[enetpacketSrc0]=MYMAC[0];
    packet[enetpacketSrc1]=MYMAC[1];
    packet[enetpacketSrc2]=MYMAC[2];
    packet[enetpacketSrc3]=MYMAC[3];
    packet[enetpacketSrc4]=MYMAC[4];
    packet[enetpacketSrc5]=MYMAC[5];
//set IP header length to 20 bytes
packet[ip_vers_len] = 0x45;
//calculate IP packet length done by the respective protocols
//calculate the IP header checksum
packet[ip_hdr_cksum]=0x00;
packet[ip_hdr_cksum+1]=0x00;
hdr_chksum =0;
hdrlen = (packet[ip_vers_len] & 0x0F) * 4;
addr = &packet[ip_vers_len];
cksum();
chksum16= ~(hdr_chksum + ((hdr_chksum & 0xFFFF0000) >> 16));
packet[ip_hdr_cksum] = make8(chksum16,1);
packet[ip_hdr_cksum+1] = make8(chksum16,0);
}
//******************************************************************
//* CHECKSUM CALCULATION ROUTINE
//* just add 16 bits numbers to hdrcksum until you reach the end of hdrlen
//******************************************************************
void cksum()
{
    while(hdrlen > 1)
    {
        // top 8 bits pointed to
data_H=*addr++;
        // next 8 bits pointed to
data_L=*addr++;
        // converting the 2 bits together into a 16bit number
        chksum16=make16(data_H,data_L);
        // adding the 16bit number to itself (where is the 1s complement?!?)
        hdr_chksum = hdr_chksum + chksum16;
        // move along the header
        hdrlen -=2;
    }
    // when hdrlen = 1 (ie only 8 bits left)
    if(hdrlen > 0)
    {
        data_H=*addr;
        data_L=0x00;
        chksum16=make16(data_H,data_L);
        hdr_chksum = hdr_chksum + chksum16;
    }
}
//******************************************************************
//* Initialize the RTL8019AS
//******************************************************************
void init_RTL8019AS()
{
    fromrtl;                           // PORTC data lines = input
    PORTC = 0xFF;
    DDRB = 0xFF;
    rtladdr = 0x00;                    // clear address lines
    DDRA=0x00;  // PORT A is an input
    //DDRA = 0xFF;
    DDRD = 0xE0;          // setup IOW, IOR,
    EEPROM,RXD,TXD,CTS
    PORTD = 0x1F;        // enable pullups on input pins
    clr_EEDO;
    set_iow_pin;         // disable IOW
    set_ior_pin;         // disable IOR
    set_rst_pin;         // put NIC in reset
    delay_ms(2);         // delay at least 1.6ms
    clr_rst_pin;         // disable reset line
    read_rtl(RSTPORT);    // read contents of reset port
    write_rtl(RSTPORT,byte_read);  // do soft reset
    delay_ms(20);         // give it time
    read_rtl(ISR);        // check for good soft reset

    if(!(byte_read & RST))
    {

        //for(i=0;i<sizeof(msg_initfail)-1;++i)
        // { }
        //    delay_ms1(1);
        //    lcd_send_byte(1,msg_initfail[i]);
        // }
    }
    write_rtl(CR,0x21);       // stop the NIC, abort DMA, page 0
    delay_ms(2);               // make sure nothing is coming in or going out
    write_rtl(DCR,dcrval);    // 0x58
    write_rtl(RBCR0,0x00);
    write_rtl(RBCR1,0x00);
    write_rtl(RCR,0x04);
    write_rtl(TPSR,txstart);
    write_rtl(TCR,0x02);
    write_rtl(PSTART,rxstart);
    write_rtl(BNRY,rxstart);
    write_rtl(PSTOP,rxstop);
}
write_rtl(CR,0x61);
delay_ms(2);
write_rtl(CURR,rxstart);
for(i=0;i<6;++i)
    write_rtl(PAR0+i,MYMACH[i]);

write_rtl(CR,0x21);
write_rtl(DCR,dcrval);
write_rtl(CR,0x22);
write_rtl(ISR,0xFF);
write_rtl(IMR,imrval);
write_rtl(TCR,tcrval);
write_rtl(CR,0x22);
}
//******************************************************************
//* MAIN MAIN MAIN MAIN MAIN MAIN MAIN MAIN MAIN MAIN MAIN MAIN
MAIN
//******************************************************************
void main(void)
{
    init_RTL8019AS();
    //setup timer 0
    TIMSK = 2;
    OCR0 = 200;
    TCCR0 = 0b00001011;
    ADMUX = 0b11100000;      //internal 2.56voltage ref with ext cap at AREF pin
    //enable ADC and set prescaler to 1/64*16MHz=125,000
    //and set int enable
    ADCSR = 0x80 + 0x07 + 0x08;
    MCUCR = 0b10010000; //enable sleep and choose ADC mode
    #asm
        sei
    #endasm
    clr_synflag;
    clr_finflag;
delay_ms(5000); // wait for boot up (5 seconds)

    //ob-mstory an ip address
dhcp();
//******************************************************************
//* Look for a packet in the receive buffer ring
//******************************************************************
    while(1)
{  //start the NIC
  write_rtl(CR,0x22);
  write_rtl(ISR,0x7F);

  //wait for a good packet
  read_rtl(ISR);
  while(!(byte_read & 1))
  {
    //PORTA.0=1;
    // resend previous data
    if(waitcount == 0)
    {
      if(DHCP_wait==1)
      {
        dhcpstate = DHCP_DIS;
        dhcp();
      }
      if(DHCP_wait==2)
      {
        dhcpstate = DHCP_OFF;
        dhcp();
      }
    }
    read_rtl(ISR);
  }
  //PORTA.0=0;

  //read the interrupt status register
  read_rtl(ISR);

  //if the receive buffer has been overrun
  if(byte_read & OVW)
  overrun();

  //if the receive buffer holds a good packet
  if(byte_read & PRX)
  get_packet();
  //make sure the receive buffer ring is empty
  //if BNRY = CURR, the buffer is empty
  read_rtl(BNRY);
  data_L = byte_read;
  write_rtl(CR,0x62);
  read_rtl(CURR);
  data_H = byte_read;
write_rtl(CR,0x22);
//buffer is not empty.. get next packet
if(data_L != data_H)
    get_packet();

//reset the interrupt bits
write_rtl(ISR,0xFF);
Appendix B: Packetwacker Schematics
Appendix C: Webpage

This webpage is running entirely on a mega32 and using an EDTP packet wacker.

This project was done by Tzeming Tan, Jeremy supervised by Dr. Bruce R. Land.

Cornell University

ATMEL embedded webservice

TCP/IP is the communications protocol most widely used for accessing the internet today. The objective of this project is to introduce this protocol to the ATMEL Mega 32 Microcontroller chip such that it can run as a simple webserver which can then be adapted for useful lab-based applications.

Although the Mega 32 and the development board will only be able to run limited web functions, it is a relatively inexpensive device compared to high power web servers. Therefore it can still have many applications for example, connecting the microcontroller to a thermometer which will display the current temperature on a browser.

The current temperature is now 29.49 degrees F.

link to ECE 476 website here