AVR460: Embedded Web Server

Introduction

Intelligent homes will be connected to the Internet and require a microcontroller to communicate with the other network devices. The AVR® embedded web server can simplify the design process for embedded web server applications. The AVR embedded web server design includes a complete web server with TCP/IP support and Ethernet interface. It also includes support for sending mail, and software for automatic configuration of the web server in the network. The AVR web server reference design includes complete source code written in C-language. A comprehensive designers guide describes all web server components and gives embedded systems designers a quick start to embedded web servers.

System Description

The AVR embedded web server reference design is designed for integration in digital equipment. The AVR embedded web server can be plugged into any Ethernet interface and communicate with a standard web browser. Figure 2 illustrates some situations where the web server can be used.

As shown in Figure 2, various consumer electronics can be controlled from a computer connected to the Internet. The web page is the "Control Center" for the AVR embedded web server.

**Figure 1.** AVR Embedded Web Server
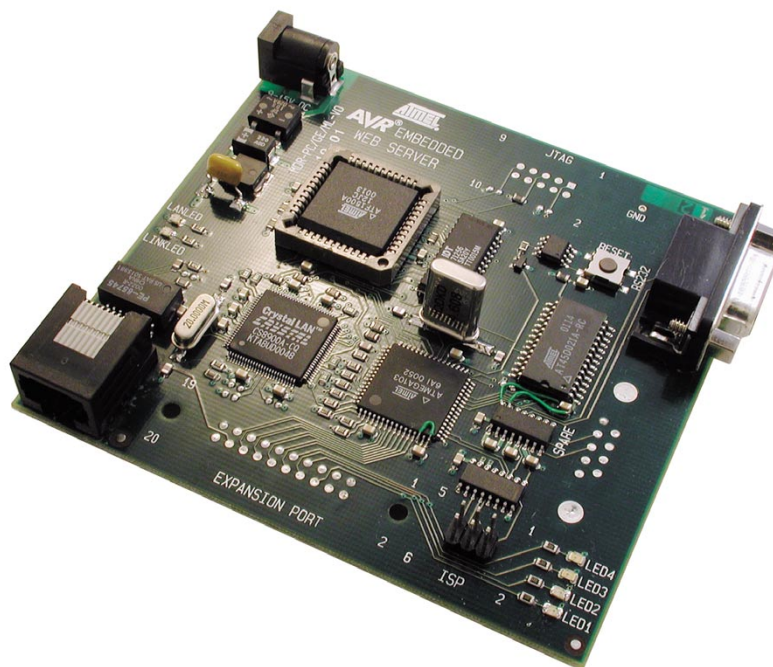
Suppose the AVR embedded web server is embedded in several units in a house. Every server is connected to the network. A computer located at home as on Figure 2 controls all devices and can receive requests from other computers on the Internet. The web server is identified by its unique IP address and can be controlled remotely from anywhere in the world as long as the authorization is in order.

Figure 2. Monitoring Home Equipment from the Office
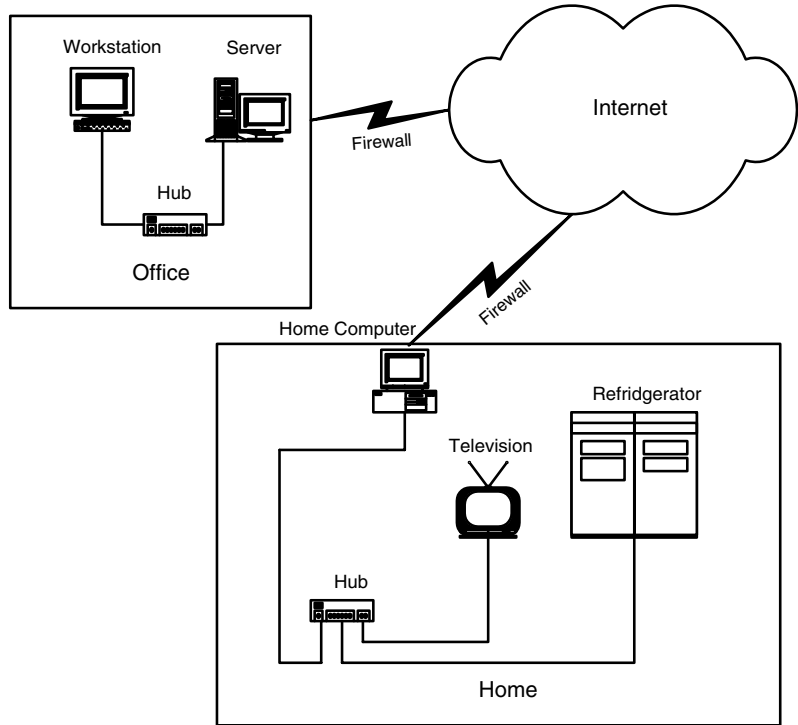
TCP/IP Protocol Suite

The TCP/IP protocol suite allows computers of all sizes, running different operating systems, to communicate with each other. It forms the basis for what is called the worldwide Internet, a Wide Area Network (WAN) of several million computers.

TCP/IP Suite Layers

The TCP/IP protocol suite is a combination of different protocols at various layers. TCP/IP is normally considered to be a 4-layer system as shown in Figure 3.

Figure 3. Four Layers of TCP/IP Protocol Suite

Application	Telnet, FTP, HTTP
Transport	TCP, UDP
Network	IP, ICMP
Link	Device Driver and Interface Card

Application Layer
The Application layer handles the details of a particular application. Common TCP/IP applications include:

• Telnet for remote login

• Browser support for displaying web pages

• File transfer applications

• E-mail applications

The three lower layers do not know anything about the specific application and only take care of communications details.

Transport Layer
TCP is responsible for a reliable flow of data between two hosts. Typically, TCP divides data passed to it from the application into appropriately sized chunks for the network layer below, acknowledging received packets that are sent and retransmits lost packets. Since this reliable, flow of data is provided by the Transport Layer, the Application Layer above can ignore these details.

UDP is a much simpler service to the Application Layer. It sends packets of data called datagrams from one host to the other, but with no guarantee that the datagrams reach the other end. Desired reliability must be added by the Application Layer.

Network Layer
This layer is sometimes called the Internet Layer. It handles the movements of packets around the network. Routing of packets, for example, takes place here. IP (Internet Protocol) and ICMP (Internet Control Message Protocol) provides the Network Layer in the TCP/IP Protocol Suite.

Link Layer
Data-link or Network Interface Layer is another common name of this layer. The Link Layer normally includes the device driver in the operating system and the corresponding network interface (card) in the computer. Together they handle all the hardware details of physically interfacing with the cable.

Figure 4 shows an example that includes two hosts on a Local Area Network (LAN) such as Ethernet, using HTTP.

Figure 4. Example with Protocols Involved
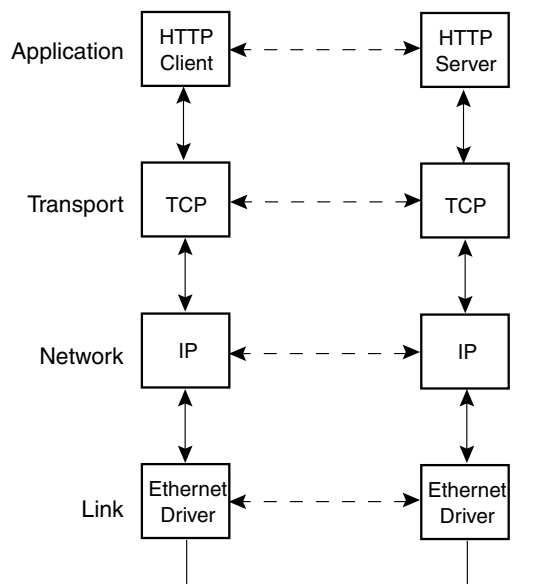
One side represents the client, and the other the server. The server provides some type of service to clients, in this case, access to web pages on the server host. Each layer has one or more protocols for communicating with its peer at the same layer. One protocol, for example, allows the two TCP layers to communicate, and another protocol lets the two IP layers communicate.

The Application Layer is normally a user-process while the lower three layers are usually implemented in the kernel (the operating system).

Port Numbers

Different applications can use TCP or UDP at any time. The Transport layer protocols store an identifier in the headers they generate to identify the application. Both TCP and UDP use 16-bit port numbers to identify applications. TCP and UDP store the source port number and the destination port number in those respective headers.

Servers are normally known by their well-known port number. Every TCP/IP implementation with a FTP server provides that service on TCP port 21. Every Telnet server is on TCP port 23. Services provided by any implementation of TCP/IP have well-known port numbers between 1 and 1023. The well-known ports are managed by the Internet Assigned Numbers Authority (IANA).

A client usually does not care what port number it uses on its end. All it needs to be certain of is that whatever port number it uses, it must be unique on its host. Client port numbers are called ephemeral ports (i.e., short lived). This is because a client typically exists only as long as the user running the client needs its service, while servers typically run as long as the host is up. Most TCP/IP implementations allocate ephemeral port numbers between 1024 and 5000. The port numbers above 5000 are intended for other services (those that are not well known across the Internet).

The combination of an IP address and a port number is called a socket.

Encapsulation

When an application sends data using TCP, the data is sent down the protocol stack, through each layer, until it is sent as a stream of bits across the network. Each layer adds information to the data by prepending headers and adding trailers to the data it receives. Figure 5 shows this process.

Figure 5. Encapsulation of Data as It Goes Down the Protocol Stack
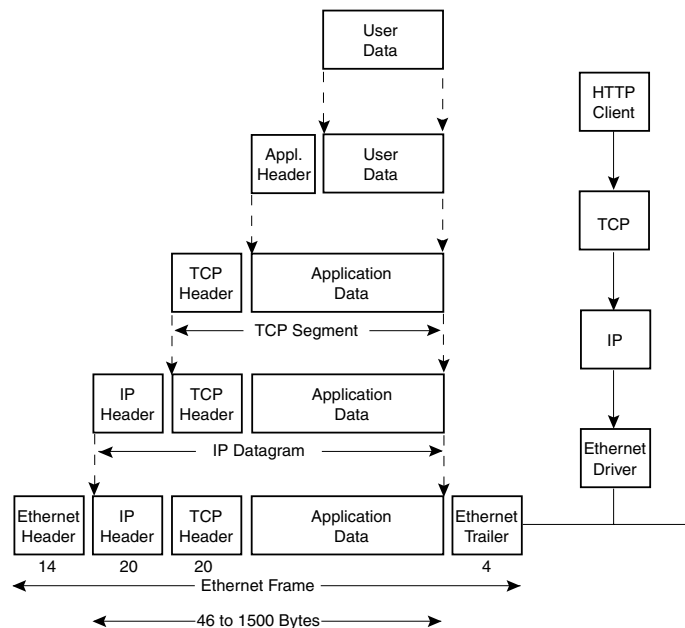
Some abbreviations:

- TCP segment: The unit of data that TCP sends to IP.
- IP datagram: The unit of data that IP sends to the network interface.
- Frame: The stream of bits that flows across the Ethernet.

IP (Internet Protocol) adds an identifier to the IP header it generates to indicate which layer the data belongs to. IP handles this by storing an 8-bit value in its header called the protocol field. Similarly, many different applications can be using TCP or UDP at any time. The Transport Layer protocol stores an identifier in the header they generate to identify the application. Both TCP and UDP use 16-bit port numbers to identify applications. The TCP and UDP store the source port number and the destination port number in their respective headers. The network interface sends and receives frames on behalf of IP, ARP, RARP. There must be some form of identification in the Ethernet header indicating which network layer protocol generates the data. To handle this, there is a 16-bit frame type field in the Ethernet header.

Internet Addresses

Every interface on the internet has a unique Internet Address (IP Address). The addresses are 32-bit numbers. Figure 6 shows the structure and the five different classes of Internet addresses.

Figure 6. The Five Different Classes of Internet Addresses
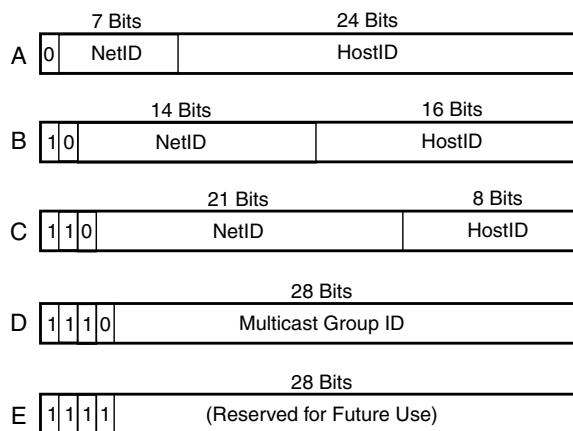
The 32-bit addresses are normally written as four decimal numbers, one for each byte of the address. See Table 1. This notation is called dotted-decimal. Since every interface on internet must have a unique IP address, there must be one central authority for allocating these addresses for networks connected to the worldwide Internet. Internet Network Information Center (NIC) is the responsible authority.

Table 1. Ranges of Different Classes of IP Addresses

Class	Range
A	0.0.0.0 to 127.255.255.255
B	128.0.0.0 to 191.255.255.255
C	192.0.0.0 to 223.255.255.255
D	224.0.0.0 to 239.255.255.255
E	240.0.0.0 to 255.255.255.255

There are three types of IP addresses: Unicast (destined for a single host), Broadcast (destined for all hosts on a given network), and Multicast (destined to a multicast group).

IP: Internet Protocol

All TCP, UDP and ICMP data get transmitted as IP datagrams. IP provides an unreliable, connectionless datagram delivery service. There is no guarantee that an IP datagram successfully gets to its destination. When something goes wrong, IP has a simple error handling algorithm: Throw away datagram and try to send an ICMP message back to the source. Any required reliability must be provided by the upper layers.

Connectionless means that IP does not maintain any state information about successive datagrams. Each datagram is handled independently from all other datagrams. This means that IP datagrams can get delivered out of order.

IP Header

Figure 7 shows the format of an IP datagram. The normal size of the IP header is 20 bytes, unless options are present.

The most significant bit is numbered 0 at the left, and the least significant bit of a 32-bit value is numbered 31 on the right. The 4 bytes in the 32-bit value are transmitted in the order: bits 0 - 7, then bits 8 - 15 and so on. This is called Big-Endian byte ordering, which is the byte ordering required for all binary integers in the TCP/IP headers as they traverse a network. This is called the network byte order.

The fields in the IP header are described below:

Version

The current protocol version is 4, so IP is sometimes called IPv4.

Figure 7. IP Header Fields
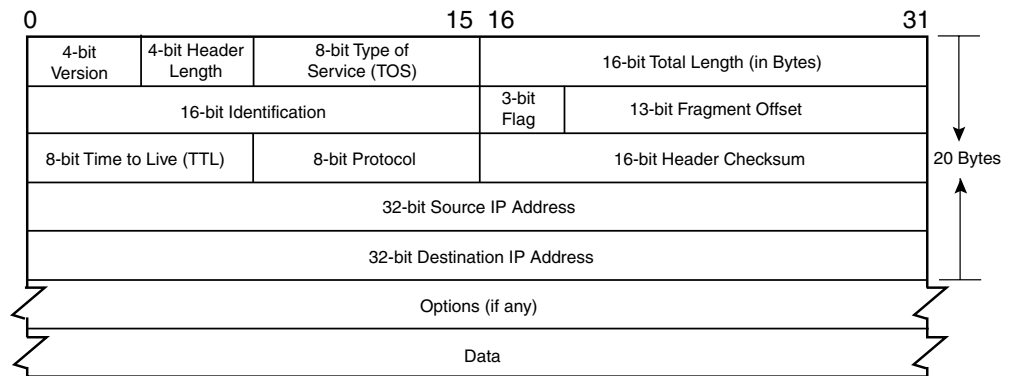
Header Length

The header length is the number of 32-bit words (4 bytes per word) in the header, including any options. This limits the header length to 60 bytes.

Type-of-service

The field, (TOS) is composed of a 32-bit precedence field (which is currently ignored), 4 TOS bits, and an unused bit that must be 0. The 4 TOS bits are: minimize delay, maximize throughput, maximize reliability and minimize monetary cost. Only 1 of these 4 bits can be turned on. If all 4 bits are 0, it implies normal service.

Total Length

This is the total length of the IP datagram in bytes. Using this field and the header length field, we know where the data portion of the IP datagram starts and its length. Since this is a 16-bit field, the maximum size of an IP datagram is 65535 bytes.

Identification

This field uniquely identifies each datagram sent by a host. It normally increments by one each time a datagram is sent.

Time-to-live	Time-to-live sets an upper limit on the number of routers through which datagram can pass. It limits the lifetime of the datagram. It is initialized by the sender to some value (often 32 or 64) and decrements by one in every router that handles the datagram. When this field reaches 0, the datagram is discarded, and the sender is notified with an ICMP message. This prevents the packet from getting caught in eternal routing loops.
Protocol	Protocol is used by IP to demultiplex incoming datagrams. It identifies which protocol gave the data for IP to send.
Header Checksum	The Header checksum is calculated over the IP header only. It does not cover any data that follows the header. ICMP, UDP and TCP all have a checksum in their own headers to cover their own header and data.
	Every IP datagram contains the source IP address and the destination IP address. These are 32-bit values.
The Option Field	This is a variable-length list of optional information for the datagrams. The options are rarely used and not all hosts and routers support the option.

UDP: User Datagram Protocol

UDP provides a way for applications to send encapsulated raw IP datagrams and send them without having to establish a connection. The protocol is transaction oriented. Delivery and duplicate protection is not guaranteed.

UDP Header

The UDP header is 8 bytes. The field UDP length is the length in bytes of the user datagram including both header and data. UDP checksum is calculated and stored by the sender and then verified by the receiver. The procedure is the same as is used in TCP.

Figure 8. The UDP Header

0	31
Source Port	Destination Port
UDP Length	UDP Checksum

TCP: Transmission Control Protocol

TCP and UDP use the same network layer (IP), but TCP provides a totally different service to the Application Layer. TCP provides a connection-oriented, reliable, byte stream service. The term connection-oriented means the two applications using TCP (normally considered a client and a server) must establish a TCP connection with each other before they can exchange data.

TCP provides reliability by doing the following:

- The application data is broken into what TCP considers the best sized chunks to send. This is totally different from UDP, where each write by the application generate a UDP datagram of that size. The unit of information passed by TCP to IP is called a segment.

- When TCP sends a segment it maintains a timer, waiting for the other end to acknowledge reception of the segment. If an acknowledge is not received in time, the segment is retransmitted.

- When TCP receives data from the other end of the connection, it sends an acknowledgment. This acknowledgment is not sent immediately, but normally delayed a fraction of a second.

- TCP maintains a checksum on its data header page. This is an end-to-end checksum whose purpose is to detect any modifications of the data in transit. If a segment arrives with an invalid checksum, TCP discards it and does not acknowledge receiving it. This expects the sender to time out and retransmit.

- Since TCP segments are transmitted as IP datagrams, and since IP datagrams can arrive out of order, TCP segments can also arrive out of order. A receiving TCP resequences the data if necessary, passing the received data in the correct order to the application.

- Since IP datagrams can get duplicated, a receiving TCP must discard duplicate data.

- TCP also provides flow control. Each end of a TCP connection has a finite amount of buffer space. A receiving TCP only allows the other end to send as much data as the receiver has buffer for. This prevents a fast host from taking all the buffer on a slower host.

A stream of 8-bit bytes is exchanged across the TCP connection between the two applications. There are no record markers automatically inserted by TCP. This is what is called a byte stream service. If the application on one end writes 10 bytes, followed by a write of 20 bytes, followed by a write of 50 bytes, the application at the other end may read 80 bytes in four reads of 20 bytes at a time. One end puts a stream of bytes into TCP and the same, identical stream of bytes appears at the other end. However, TCP does not interpret the contents of the bytes at all. TCP has no idea if the data bytes being exchanged are binary data, ASCII characters, or any other form. The interpretation of this byte stream is up to the applications on each end of the connection.

TCP Header

TCP data is encapsulated in an IP datagram, as shown in Figure 9.

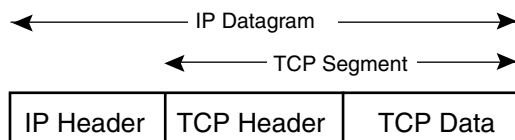**Figure 9.** Encapslation of TCP Data in a IP Datagram

Figure 10 shows the format of the TCP header. Its normal size is 20 bytes, unless options are present.

Each TCP segment contains the source and destination port number to identify the sending and receiving application. These two values, along with the source and destination IP addresses in the IP header uniquely identify each connection.

The combination of an IP address and a port number is called a socket. TCP is a full-duplex service, this means that data can be flowing in both directions, independent of each other. Therefore, each end of the connection must maintain a sequence number of data flowing in each direction.

Sequence Number

This identifies the byte stream of data from the sending TCP to the receiving TCP that the first byte of data in this segment represents. TCP numbers each byte with a sequence number. The sequence number is a 32-bit unsigned number that wraps back around to 0 after reaching 2^{32}-1.

Acknowledge Number

This contains the next sequence number that the sender of the acknowledgment expects to receive. This is therefore the sequence number plus 1 of the last successfully received byte of data. This field is only valid if the ACK flag is set. The 32-bit acknowledgment number field is always part of the header, as is the ACK flag. This field is always set and the ACK FLAG is on.

Header Length

This gives the length of the header in 32-bit words. This is required because the length of the options field is variable. With a 4-bit field, TCP is limited to a 60-byte header. Without options, however, the normal size is 20 bytes.

Figure 10. TCP Header
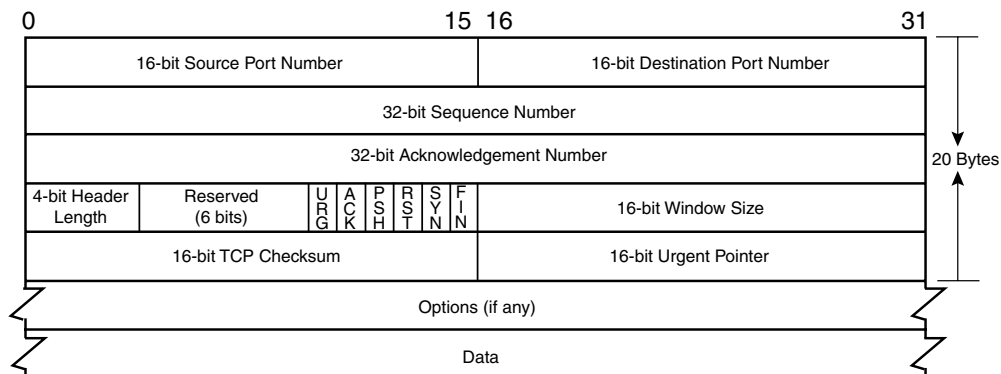
Flag Bits

There are six flag bits in the TCP header – one or more of them can be turned on at the same time:

1. URG – The urgent pointer is valid.
2. ACK – The acknowledgment number is valid.
3. PSH – The receiver should pass data to the application as soon as possible.
4. RST – Reset the connection.
5. SYN – Synchronize sequence numbers to initiate a connection.
6. FIN – the sender is finished sending data.

Window Size	TCP's flow control is provided by each end advertising a window's size. This is the number of bytes, starting with the one specified by the acknowledgment number field, that the receiver is willing to accept. This is a 16-bit field, limiting the window to 65535 bytes.
Checksum	Checksum covers the TCP segment: the TCP header and the TCP data. This is a mandatory field and must be calculated and stored by the sender and then verified by the receiver.
Urgent Pointer	The urgent pointer is valid only if the URG flag is set. This pointer is a positive offset that must be added to the sequence number field of the segment to yield the segment number of the last byte of urgent data. TCP's urgent mode is a way for the sender to transmit emergency data to the other end.
Checksum	The most common option field is the MSS (maximum segment size). Each end of a connection normally specifies this option on the first segment exchanged (the one sent with the SYN flag set to establish the connection). It specifies the maximum size segment that the sender wants to receive.
Data	The data portion of the TCP header is optional. A header without data is used to acknowledge received data if there is no data to be transmitted in that direction. There are also some cases dealing with time-outs when a segment can be sent without any data.

Ethernet Encapsulation

The term Ethernet generally refers to a standard published in 1982 by Digital Equipment Corporation, Intel Corporation, and Xerox Corporation. It is the predominate form of local area network technology used with TCP/IP today. It uses an access method called CSMA/CD (Carrier Sense, Multiple Access with Collision Detection). It operates at 10/100 Mbits/s and uses 48-bit addresses. Two encapsulations are used, described by RFC 1042 and RFC 894. RFC 894 encapsulation is most commonly used. Figure 11 shows the RFC 894 encapsulation.

The frame format uses 48-bit (6-byte) destination and source addresses. The ARP and RARP protocols map between the 32-bit IP addresses and the 48-bit hardware addresses.

Figure 11. Ethernet Encapsulation (RFC 894)

Preamble	Dest. Adr.	Source Adr.	Type	Data	CRC
8	6	6	2	45-1500	4

Destination Address	Specifies the 48-bit destination address.
Source Address	Specifies the 48-bit source address.
Ethernet Type	Identifies the type of data that follows.
CRC	Cyclic redundancy checksum used to detects errors in the rest of the frame.
Minimum Size	For an Ethernet frame it is 46 bytes. To handle this, pad bytes are inserted to assure that the frame is sufficiently large.

ARP and RARP – (Reverse) Address Resolution Protocol

Address resolution provides a mapping between the two different forms of addresses: 32-bit IP addresses and whatever type of address the data link layer uses. ARP is specified in RFC 826.

ARP provides a dynamic mapping from an IP address to the corresponding hardware address. The term dynamic since it happens automatically and is normally not a concern of either the application user or the system administrator.

RARP is used by many diskless systems to obtain their IP address when bootstrapped. The RARP packet format is nearly identical to the ARP packet.

ARP cache maintenance on each host is essential to efficient operation of ARP. The cache maintains the recent mappings from Internet addresses to hardware addresses. The normal expiration time of an entry in the cache is 20 minutes from the time the cache was created.

ARP Packet Format

Figure 12 shows the format of an ARP request and reply packet, when used on an Ethernet to resolve an IP address.

Source and Destination Ethernet Addresses

The special Ethernet destination address of all 48 bits set (0xFFFFFFFFFFFF) means the broadcast address. All Ethernet interfaces on the LAN receive these frames.

Ethernet Frame Type

This specifies the type of data that follows. For an ARP request or an ARP reply, this field is 0x0806.

The adjectives hardware and protocol are used to describe the fields in the ARP packets. For example, an ARP request asks for the hardware address corresponding to a protocol address.

Hard Type

This specifies the type of hardware address. Its value is 1 for an Ethernet.

Prot Type

Prot Type specifies the type of protocol address being mapped. Its value is 0x0800 for an IP address.

Hard Size and Prot Size

Hard Size and Prot Size specify the sizes (in bytes) of the hardware address and the protocol addresses. For an ARP request or reply for an IP address on an Ethernet they are 6 and 4, respectively.

Op

Op specifies whether the operation is an ARP request (a value of 1), ARP reply (2), RARP request (3), RARP reply (4). This field is required since the frame type field is the same for an ARP request and an ARP reply.

Sender Hardware Address

Ethernet address in this example, the sender's protocol address (an IP address), the target hardware address, and the target protocol address.

For an ARP request, all the fields are filled in except the target hardware address. When a system receives an ARP request to it, it fills in its hardware address, swaps the two sender addresses with the two target addresses, sets the OP field to 2 and sends the reply.

Dynamic Host Configuration Protocol (DHCP)

DHCP provides configuration parameters to Internet hosts. Hosts can send a DHCP-packet while booting, DHCP-servers (if present) reply to this message and supplies the host with parameters necessary to complete the reboot.

Figure 12. Format of ARP Request or Reply Packet When Used on an Ethernet.

6	6	2	2	2	1	1	2	6	4	6	4
Ethernet Dest. Addr	Ethernet Source Addr	Frame Type	Hard Type	Prot Type			Op	Sender Ethernet Addr	Sender IP Addr	Target Ethernet Addr	Target IP Addr

← Ethernet Header → ← 28 Byte ARP Request/Reply →

Hard Size ← → Prot Size

Figure 13. DHCP Header

0 7	8 15	16 23	24 31
Opcode (1=request, 0=reply)	Hardware Type (1=Ethernet)	Hardware Addr. Length (6 for Ethernet)	Hop Count
Transaction ID			
Number of Seconds		(Unused)	
Client IP Address			
Your IP Address			
Server IP Address			
Gateway IP Address			
Client Hardware Address (16 Bytes)			
Server Hostname (64 Bytes)			
Boot Filename (128 Bytes)			
Vendor-specific Information (64 Bytes)			

300 Bytes

Parameters provided by DHCP include IP address, gateway's IP address, DHCP-server's IP address, server hostname and lease time. This gives a dynamic and easy maintenance of the network and IP addresses on the network for the network administrator and automatic configuration for the network clients. The IP address is leased for a given period, typically 90 days, but can be released if the host does not need it anymore. When half the lease time is used, the host should ask for a new lease time. If the lease time is not renewed before the lease time expires, the host must give up its IP address, and wait until a new IP address is provided.

Hypertext Transfer Protocol (HTTP)

The HTTP protocol is a protocol that allows a (web) client to request files or other resources from a server. Various types of requests can be sent by the client. The most basic are the "GET" request and the "POST" request which are used to fetch and post data, respectively. The server processes the request, returns a header containing a status code and either a file or an HTML document attached after the header. Finally, the server closes the connection.

HTTP Message

An HTTP message consists of requests from client to server and responses from server to client. The message format is similar in many ways to that used by Internet Mail and the Multipurpose Internet Mail Extension (MIME) as defined in RFC 822 and RFC 1521.

HTTP Request Message

A request message consists of a request-line followed by some header-lines specifying the request. Example (POST) request is shown in Table 2.

Table 2. Example POST Request

Request-line	POST/ewsScript.cgi HTTP/1.0
Request-headers	Referer: http://ews/ewsScript.cgi?action=add&form=resource
	User-Agent: Mozilla/4.0 (compatible; MSIE 5.01; Windows NT® 5.0)
Entity-headers	Accept-encoding: gzip, deflate
	Content-type: application/x-www-form-urlencoded
	Content-length: 113
CRLF	Carriage Return, Line Feed
Entity-body	action=add&form=resource

HTTP Response Message

The response messages consists of a response-line followed by header-lines and the entity body. The entity body is separated from the headers by a null line. Example (GET) response is shown in Table 3.

Table 3. Example GET Response

Status-line	HTTP/1.0 200 OK
Response-headers	Server: Atmel AVR EWS
Entity-headers	Content-type: image/gif
	Content-length: 1340
CRLF	Carriage Return, Line Feed
Entity-body	<file contents comes here>

HTTP Methods

The first word in the request line is the name of the method to be executed. Since the method is specified in this way, HTTP can be expanded to cover the needs of future object-oriented applications. The common methods are listed below in Table 4.

Table 4. The Commonly Built-in HTTP Request Methods

Method	Description
GET	Request to read a web page or whatever information is identified by the Request-URI.
POST	Append to a named resource (e.g. a Web page), or provide a block of data to a data-handling process at the server.
HEAD	Request to read a web page's header.
PUT	Request to store a web page.
DELETE	Remove a web page.
LINK	Connects two existing resources.
UNLINK	Breaks an existing connection between two resources.

Status Codes

Every request gets a response starting with a status line. The status line consists of the protocol version followed by a numeric status code and its associated textual phrase. The common HTTP/1.0 status codes are listed below in Table 5.

Table 5. Common Status Codes

Numeric Code	Description
200	OK
201	Created
202	Accepted
204	No Content
301	Moved Permanently
302	Moved Temporarily
304	Not Modified
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable

Simple Mail Transfer Protocol (SMTP)

SMTP, as defined in RFC 821, is designed to transfer mail over a reliable ordered data stream channel. When sending mail over the Internet, TCP is used for transport.

SMTP Mail Transfer

The sender establishes a connection with the receiver on port 25, and identifies itself with the HELO command. The sender then transmits the source and destination mail addresses, using the MAIL– and the RCPT–command. If the receiver can accept the mail it replies with a 250 reply to each of these commands. The sender then transmits a DATA command, if the receiver replies with 354, the sender may start transmitting the message. The end of the message is indicated by <CRLF>.<CRLF>. After successful reception the receiver replies with a 250, the sender may then send a QUIT command to close the connection.

The receiver may reply with various error codes during all of the steps described above.

Table 6. Most Commonly Used SMTP Commands

Command	Description	Example
HELO	Used to identify the sender-SMTP to the receiver-SMTP	HELO EWS
MAIL	Used to specify the senders mail address	MAIL FROM: <avr@atmel.com>
RCPT	Used to specify the receivers mail address	RCPT TO: <itanium@intel.com>
DATA	Used by the sender to initiate the transfer of the message	DATA
QUIT	Tells the receiver to close the channel	QUIT

# Diagnostic Programs: PING

Typical programs include ping and traceroute. They can both be run from Unix® or MS-DOS.

The Ping program was written by Mike Muss and tests whether another host is reachable. The program sends an ICMP echo request message to a host, expecting an ICMP echo reply to be returned. Ping measures the round-trip time to the host, giving some indications of how far away the host is.

The Ping program that sends the echo request is called the client and the host being pinged the server. Most TCP/IP implementations support the Ping server directly in the kernel– the server is not a user process.

Figure 14 shows the ICMP echo request.

Figure 14. Format of ICMP Message for Echo Request and Echo Reply
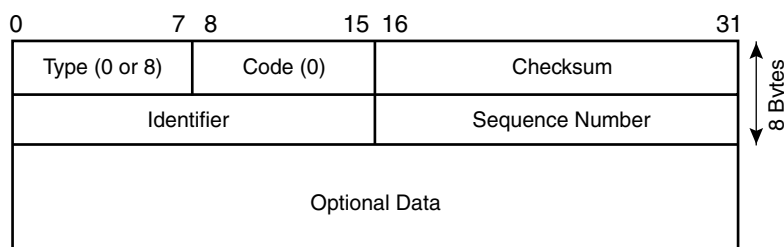
As with other ICMP query messages, the server must echo the identifier and sequence number fields. Also, any optional data sent by the client must be echoed. The sequence number starts at 0 and increments every time a new echo request is sent. Ping prints the sequence number of each returned packet, allowing us to see if packets are missing, reordered, or duplicated. IP is the best effort datagram delivery service, so any of these three conditions can occur.

When the ICMP echo reply is returned, the sequence number is printed, followed by the TTL, and the round-trip time is calculated.

Hardware

The AVR embedded web server reference design is designed to be flexible for future development. The web server primarily communicates through an Ethernet connection. But it is also able to communicate with SLIP and modem connection using the built–in UART. A CPLD is included in the reference design to allow memory mapping of other devices to the system. With only SRAM and the Ethernet controller connected to the data bus the CPLD can be omitted.

Figure 15 shows how the microcontroller communicates through Ethernet and serial communication.

Figure 15. Overview of Data Flow between Components
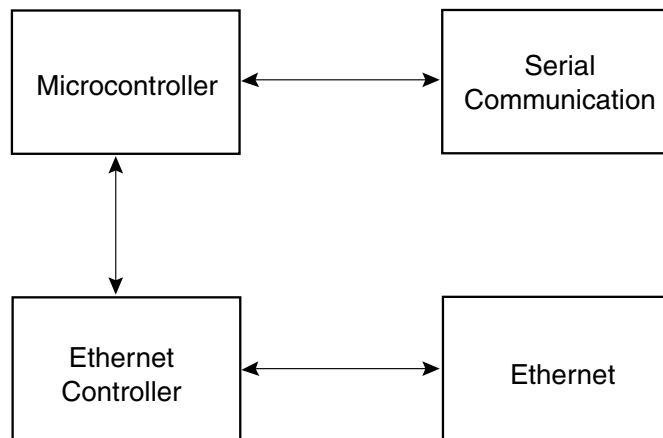
The hardware design is flexible, making it usable for many applications. It is designed to be used with Ethernet, SLIP or PPP connections, either through a LAN, another computer or through a dial-up connection. It is also possible to connect other external peripherals to the system. The on-board Flash memory can be expanded without hardware changes.

Memory

The web server includes enough memory to develop large applications on top of the web server protocols. 32K bytes of external SRAM is used for buffering data. A 2-Mbit external DataFlash® is used for storing web pages to allow a large amount of pages to be stored. The SRAM is connected to the address bus and data bus. The Serial Peripheral Interface (SPI) is used for communication with the DataFlash.

Ethernet Controller

The Ethernet controller was originally a 16-bit ISA device, but can also be controlled in 8-bit mode. The Ethernet controller is configured as an 8-bit device.

The Ethernet controller features a 4K bytes of internal memory which is accessed through the I/O registers or directly through memory mapping of the entire memory.

Default operation on the Ethernet controller is I/O mode and address 0300h. Since only address lines A0 - A12 are connected (need only 4 K bytes of address space), the I/O registers are mapped to address 8300h - 830Fh (I/O mode when the address lines into the PLD have the following configuration: bit 15 is high and bit 14 is low). By configuring the Ethernet controller through the I/O registers, the address can be changed and memory mode can be enabled. Memory mode operations can be mapped into address locations C000h - D000h.

Limitations

The web server has no obvious limitations. The microcontroller SRAM size is limited to 32 Kb, but this should be enough for development of the web server with a range of applications on top of the TCP/IP stack. The ATmega103 microcontroller provides 128K bytes of internal programmable Flash memory which is sufficient for a large range of applications. The web server runs at a speed of 4.608 MHz which makes the microcontroller able to handle incoming requests, and performance to handle other applications.

Communication and Add-on Cards

In addition to the Ethernet controller, the card has one UART making it possible to run the link through a SLIP/PPP connection or sending debugging information to a computer.

There is also a 20-pin connector for an additional card. This may be a card measuring the temperature or a card with relays to do a special task such as opening and closing a window. The connector has ground and power pins to supply the add-on card with power, a 20 MHz clock signal and eight input and eight output pins. The input pins are connected to the analog to digital converter on the microcontroller so they can be used either as digital I/O pins or analog inputs for measuring analog signals with the internal 10-bit A/D converter.

AVR Software

The software running on the AVR embedded web server follows the same layered structure as used in the TCP/IP protocol suite. Every layer acts independently from each other. An Ethernet controller driver controls the Ethernet interface. The Address Resolution Protocol (ARP) translates IP addresses to Ethernet MAC addresses (and vice versa) The Internet Protocol (IP) delivers packets to Transmission Control Protocol (TCP), UDP, and Internet Control Message Protocol (ICMP), the ICMP answers to PING requests and TCP/UDP delivers data to the applications. The applications can communicate with the transport layer through buffers with data and variables with control information. This section explains how the TCP/IP protocol suite is built up in our approach.

Figure 16. Protocol Stack
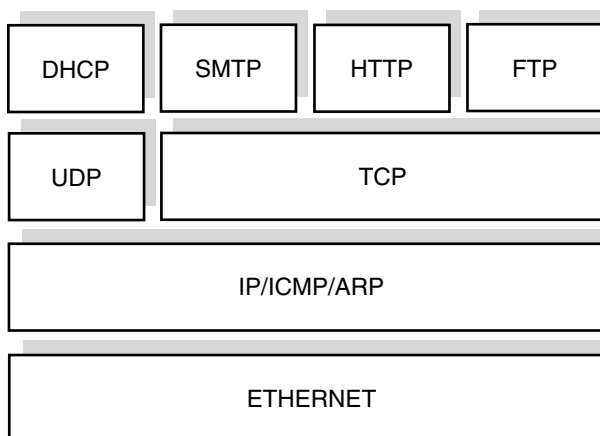
Link Layer

The Ethernet controller is configured to generate an interrupt every time a packet addressed directly to the Ethernet address arrives or when a broadcast arrives. When an interrupt occurs, the microcontroller reads the whole Ethernet frame into memory.

Ethernet Driver

A buffer of 1514 bytes, which is the maximum frame size on Ethernet, is reserved for this frame. Once the frame is transferred to the microcontroller, the Ethernet header is checked in order to ensure not receiving a misplaced frame. If the Ethernet address seen by the receiver is either a broadcast (all binary 1's) or addressed directly to specific Ethernet device, the frame is sent to the next layer or protocol according to the field, protocol type, in the Ethernet header.

Network Layer

The network layer controls the communication between hosts on the Ethernet. There is no form of transmission control to ensure that IP datagrams arrive to the host or that all IP datagrams from another host is received. This makes the layer rather easy to make. The ICMP sends messages between hosts and is only used to answer PING requests from a host. The IP handles communication for the overlaying Transport Layer.

**ARP:
Address Resolution
Protocol**

If the type in the Ethernet header is 0x0806, the frame is sent to the ARP which calculates the checksum. The checksum computed must always equal to zero. If the checksum is correct, the microcontroller checks to see if this is a request for the IP address or a response to a request sent by the microcontroller. If it is a request, a response is returned with the Ethernet address filled into the sender's Ethernet address.

A request for an Ethernet address can also be initiated by TCP or UDP. TCP or UDP checks to see if the Ethernet address for the user is available and if it's not, a request is sent. When the response to the request returns, ARP fills the Ethernet address into an Ethernet/IP address translation table. This table contains the last ten Ethernet addresses communicated with the web server.

**IP:
Internet Protocol**

An IP datagram contains a TCP Segment, UDP datagram, ICMP message or another transport control protocol segment. The IP offers a connectionless and unreliable delivery and is in this web server only used to check the datagram for errors and direct the data to the correct protocol.

When an IP datagram arrives the checksum is controlled and if the checksum is correct, the protocol field in the header is controlled. Figure 7 shows the protocol types and corresponding message types for IP protocols. Protocols not shown in Figure 7 are discarded.

Table 7. IP Protocol and Message Types

Protocol	Message type
0x01	ICMP
0x06	TCP
0x11	UDP

When IP gets a message or segment from either ICMP, TCP, or UDP, IP makes a new header for the datagram and computes a new checksum before the datagram is sent to the link layer.

**DHCP:
Dynamic Host
Configuration Protocol**

The implementation is based on the specifications in RFC2131. In the current implementation DHCP is only used for obtaining an IP address. Additional configuration parameters from the DHCP-server are ignored.

Interface

To start the DHCP-client dhcp() must be called once. A request for an IP address will then be made and eventually the web server will get an IP address for a given lease time and enter the state BOUND. To maintain renewing of the lease time the function dhcp() must be called regularly by the web server. Expiration of the lease time is monitored by a counter. Accumulation of the counter is performed by the function checkDHCP(). This function must be called within the timer interrupt function of timer0.

Initialization

DHCP is used for initial configuration before the TCP/IP software has been configured. In its communication with the DHCP server, it will have to send and receive UDP packets. A requirement is, therefore, that the IP layer must be able to forward packets to the UDP layer even before it has been configured with an IP address. To accomplish this the IP layer check if the flag dhcpConf is set. In that case IP forward any packet, regardless of addresses, right up to the DHCP The flag is reset when DHCP reach the state BOUND.

Lease Time

The lease time is the time quantum for which the DHCP server can grant the offered IP address. When lease time is received from the server it is converted from seconds to minutes to reduce the size of the stored variable. The counter that monitors the lease time, *dhcpStatus.min* increments once every minute. This is done in *checkDHCP*() which is called on every timer0 interrupt. The constant DHCP_MINUTE should be scaled to match the clock frequency used.

DhcpStatus.min is limited to 45 days (at 4.6 MHz clock frequency). Because DhcpStatus.min must be able to count to T1, which is 0.5 • lease time, this means that the longest lease time that can be handled is 90 days. Lease times extending 90 days will simply be interpreted as 90 days.

**Other Modifications and
Limitations**

The DHCP client stores the obtained IP address in EEPROM. After a reboot the client reads this IP address and sends a request to the DHCP server with the old IP address in the requested IP address field.

If there is no answer to a discover or request message, the messages will be resent after a short period of waiting. The function *wait* (unsigned char time) is called to poll the UDP port for incoming packets. The input parameter *time* is the multiple of 3,64 seconds (dependent of clock frequency) before the wait function generates a time-out.

The client simply accept the first incoming offer in response to a discover message without trying to choose among servers.

Decline and release messages are not implemented. This means that the client has no way to release a leased IP address.

**ICMP:
Internet Control Message
Protocol**

The ICMP is used only to answer PING requests from a client. All other messages are discarded. When an ICMP message is received, the checksum is controlled and if the message is an echo request, the microcontroller changes the request to a response, calculates a new checksum and returns the message to the client.

Transport Layer

On the transport layer there are two major protocols which offer two different kinds of service; TCP which is a reliable delivery service and UDP which offers an unreliable service. TCP also offers flow control for retransmission of segments and acknowledgement of received segments.
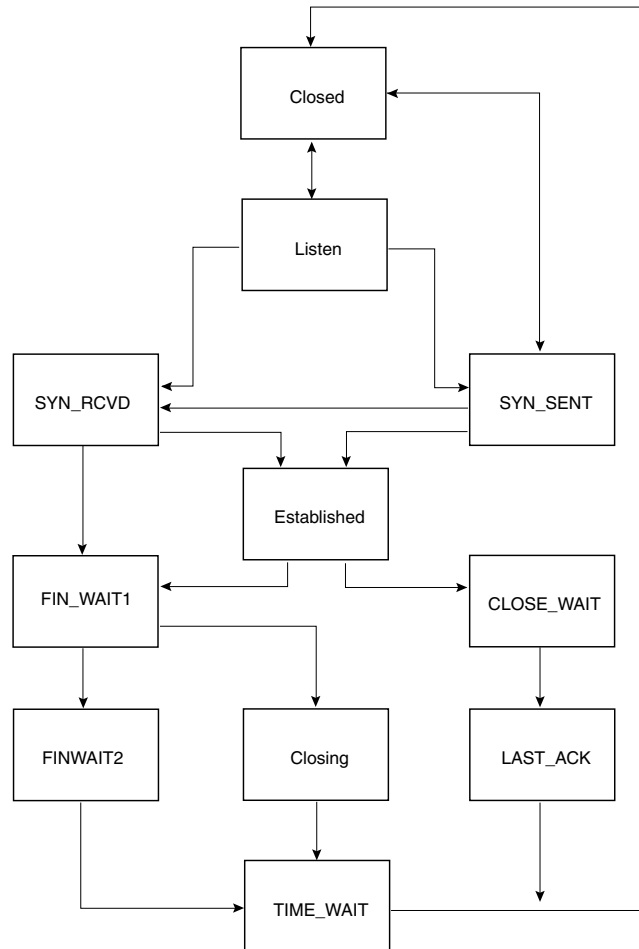
Transmission Control Protocol

The TCP is the most complex of the protocols in the TCP/IP protocol stack. TCP makes a connection oriented and reliable delivery of data and controls the flow between the two hosts. Before a connection is established, the two hosts perform a handshake to make sure both hosts are ready for transmission. When a connection is closed, both hosts must confirm the closing before the connection is shut down. A sliding window makes sure the hosts don't send segments that the other host is not able to receive. Typically, the size of the sliding window is in the range 512 - 8192 bytes.

TCP State Machine

A state machine controls the connection. Figure 17 shows this state machine according to the standard described in RFC 793.

Figure 17. TCP State Machine According to the Standard Described in RFC 793.

The implementation in the AVR embedded web server follows the standard given in RFC 793 (Transmission Control Protocol) and RFC 1122 (Requirements for Internet Hosts) except for some special cases:

- Urgent and Precedence. The use of urgent pointers and precedence is considered unnecessary for our implementation and is not implemented at all. Such options are ignored.
- Retransmissions. Algorithms for calculating retransmission times, such as Jacobson slow start algorithm and Karn's algorithm, are omitted.
- Application Layer – TCP interface. RFC 1122 requires TCP implementations to support an error reporting mechanism. This mechanism and other mechanisms for communication between TCP and application are omitted.

LISTEN

In the LISTEN state TCP waits for either an application to initiate a transfer or another host to request a connection. When a TCP segment arrives in LISTEN state, TCP first checks if the RST(reset) flag is set. If the flag is set, the segment is discarded. Second TCP looks for an acknowledge, which should not happen here, and sends a RST back to the user if ACK is set. Next it looks for a SYN. If the SYN flag is set, and there are room for another socket, a Transmission Control Block (TCB) and a socket is created, next state is SYN_RCVD and a SYN/ACK is returned.

SYN_SENT

The state machine enters this state when the server application initiates a transfer and a SYN is sent. When a segment arrives in this state, TCP checks for an ACK. If the ACK flag is set and the sequence number acknowledged is less or equal the sequence number sent, a RST is sent (if the RST flag is not set in the segment). Then the segments are dropped and the TCP connection deleted. If the acknowledged sequence number is correct and SYN is set, an ACK is returned and next state is ESTABLISHED. If the ACK flag is not set but SYN is set, return a SYN/ACK to host and next state is SYN_RCVD. In other cases, the segment is dropped.

SYN_RCVD

This state occurs if a SYN is received while in either LISTEN state or SYN_SENT state. First check if the sequence number is acceptable[1]. If the sequence number is not acceptable, send a reply with an ACK and drop the segment. If the sequence number is acceptable, check for RST. If the RST flag was set, delete the TCB and drop the segment. Then check for the SYN flag. If the SYN flag was set, reply with RST, drop the segment and delete the TCB. If the ACK flag is set, the next state is ESTABLISHED. If the ACK flag was not set, drop the frame.

Note: 1. Acceptable means that the segment received is within the window.

Established

Established state is the main state in the TCP state machine. This is the state where most of the data transfer appears. Like we did in the SYN_RCVD state, we first check to see if the sequence number is acceptable and if the RST or SYN flag is set. If one of these flags are set, a RST reply is sent and the TCB is deleted. Next check for an ACK. If the ACK flag is set, update the window and delete elements acknowledged from retransmission queue. The next thing to do is to process the data in the segment. This means to copy the contents to the receive buffer and update window size. If the FIN flag is set, enter CLOSE_WAIT and acknowledge the FIN. If the application wants to close the connection, send a FIN and enter FIN_WAIT1.

FIN_WAIT1

This state does all processing in the same manner as the ESTABLISHED state. The only exception is, when all segments sent are acknowledged, the next state is FIN_WAIT2.

FIN_WAIT2

Like FIN_WAIT1, this state processes all segments in the same way as ESTABLISHED. The only exception is when the FIN flag is received, an acknowledge is returned, the next state is TIME_WAIT.

CLOSE_WAIT

While in this state, the only segments arriving should be acknowledges to segments sent. The other host is done sending data, so when there is no more data to send from the server, a FIN should be sent and connection closed. First check to see if the sequence number is acceptable and if the RST or SYN flag is set. If one of these flags are set, a RST reply is sent and the TCB is deleted. Next check for an ACK. If the ACK flag is set, update the window and delete elements acknowledged from retransmission queue. When the application wants to close the connection, a FIN is sent and next state is LAST_ACK.

LAST_ACK

The only segments that should arrive here are acknowledges of either previous data segments or our FIN. If this is not an acknowledge of our FIN, drop the table. Else delete the TCB and next state is LISTEN.

TIME_WAIT

After the server has closed the connection, it must still take care so the other host receives the last segments from the retransmission buffer. Nothing is done in this state except for the normal retransmission handling. After the time-out the TCB is deleted.

Sockets and Windows

When a connection is established, a Transmission Control Block (TCB) is created. This block contains information about the connection and a pointer to the input and output buffers. The buffer size is determined by the constant TCP_WIN_SIZE and the maximum number of TCBs are determined by the constant TCP_MAX_SOCKETS. When the connection is established, and when an acknowledge is received, the window size is updated. This means that segments with sequence number above acknowledged sequence number plus window size should not be sent or received. If a segment arrives out of order, the segment is processed and data stored in the buffer at the correct location.

Applications

Several applications may be implemented in the AVR embedded web server. The main limitation is memory usage and performance. Running several applications at once means lower performance. Applications implemented in AVR embedded Ethernet are HTTPD, FTPD and mail client. HTTPD is a HTTP server to provide web pages to a standard web browser, FTPD is a file transfer protocol server that enables user to remote update the contents of the server from anywhere in the network. SMTP is a mail client used to send mail to specified users using a specified mail server.

File Transfer Protocol Daemon

The file transfer protocol is implemented as described in RFC 959, but with several exceptions. According to RFC 959 the minimum implementation of FTP should support the following commands: USER, QUIT, PORT, TYPE, MODE, STRU, RETR, STOR and NOOP. MODE, STRU and TYPE should be implemented for the supported values. Since MODE and STRU can only have one value in our implementation, these commands are not implemented and "500 Command not understood." is returned. TYPE supports ASCII and BINARY types, which are the one needed for transmissions of text and binary files. The rest of the commands in the list are implemented in addition to PASS for sending password, LIST and NLST for directory listings.

Another limitation of FTP is that only one user is allowed at a time. If a new connection is established to FTP while there already is one, The client receive a message that the maximum number of connections are exceeded, and the connection is closed.

FTP uses the well known port 21 for control connections. When the client or user wants to transmit data, a data connection has to be opened. This connection uses port 21.

The FTP daemon is realized as a state machine with the following states:

- **UNLOCKED** – FTP is free and ready for a new connection. When a connection is established, the server sends a welcome message to the client and requests a user name. After the message is sent, state USER is entered.
- **USER** – The only command accepted in this state is USER giving the user name of the user. If a USER command is received, a request for the password is sent and FTP enters PASS state.
- **PASS** – The only command accepted in this state is PASS giving the password of the user. If the PASS command is received and the password is correct, FTP enters IDLE state and sends an acknowledge to the client telling him the user is logged in. If the password is incorrect, the connection is closed and FTP enters UNLOCKED.
- **IDLE** – This state is where the rest of the commands are used. When a command requiring a data connection is received, FTP enters the DATA state. If one of the other commands is received, FTP does the right action and continues in IDLE state.
- **DATA** – When FTP is in DATA state, no command processing is done. FTP feeds TCP with data if the client has asked for data and stores incoming data to a file if the client wants to store data..

Table 8. FTP Commands

Command	Usage	Description
USER	USER <username>	Used during login. Server returns request for password for the user and enters the PASS state.
PASS	PASS <password>	Used during login. If the password is correct for the user, the user is logged in.
TYPE	TYPE <type>	Tells the server which type the file being transmitted is. Possible types are I (binary) and A (ASCII). In AVR embedded web server there is no difference between the two types.
PORT	PORT 192,168,1,2,4,231	Port tells the server which IP address and port number should be used when opening a data connection. The first four numbers denotes the IP address and the two last numbers denotes the port number. In this example the port number is 4 • 256 + 231 = 1255.
RETR	RETR <filename>	Used when the client wants to download a file. The server opens a connection to the IP address and port number given by the PORT command and transmits the file using the type given by TYPE.
STOR	STOR <filename>	Used when the client wants to upload a file. The server opens a connection to the IP address given by PORT and retrieves the file sent by the client.
LIST	LIST	Retrieve the long directory listing with attributes, file size and file names. The data connections are opened in the same way used when sending a file (RETR). After the last file the amount of free space is sent.
NLST	NLST	Retrieve the short directory listing with only the file names.
QUIT	QUIT	Used by the client when the connection is to be closed. The server sends 'Goodbye' to the client and closes the connection.

Flash File System

The file system on the AVR embedded web server is designed for storing web pages and pictures together with configuration files. These files normally do not need frequent updates. This means that efficient reading is most important for an overall good performance. Another important issue in a Flash file system design is to keep write cycles distributed over the Flash since each page in the DataFlash is guaranteed for only 10,000 write cycles. A file system with a File Allocation Table (FAT) in one part of the Flash is unwanted since this would lead to a non-uniform distribution of write cycles.

The file system supports traditional MS-DOS file names with eight plus three characters and is case sensitive. The maximum number of files is limited only by the number of pages and available space. The number of concurrently open files is limited by *FILE_MAX_OPEN_FILES* defined in **file.h**. Only one file can be open for writing at a time.

Linear File System

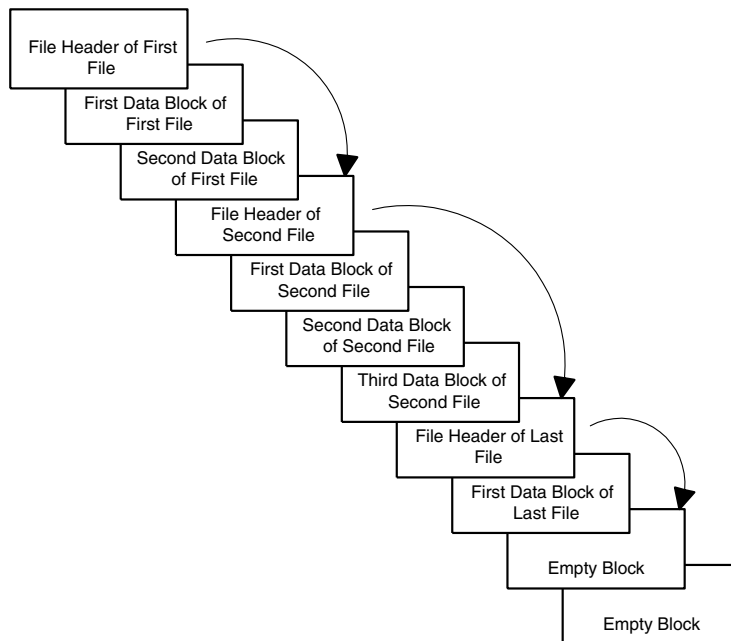**Figure 18.** Block Organization in File System

Figure 18 shows how blocks are organized in the DataFlash. The block size is equal to the page size. Information about the file system is written to EEPROM in a media status table. This table is changed every time a file is written or deleted. When a file is deleted, the page number to the file header is written in a table. Pages containing a deleted file is not freed for use immediately, but only deallocated. When the number of deallocated files exceeds a predefined number, or there are not any free pages left, deallocated pages are reclaimed and free for use. Reclaiming is done by sorting the list of deleted files in increasing page number order and calculating the offset for each file after the deleted files. All pages are moved according to the offset. The result is that all free pages is allocated after the last file and can be used again.

Figure 19. Media Status Table

Media Status Table	
Status	Free_space
Free_space	
De_allocated_space	
Last_page	
Deallocated_ptrs (16 Filepointers)	

status: BUFFER_READY, BUFFER_READ, BUFFER_WRITE, BUFFER_RECLAIM.
free_space: Number of unused blocks. (Not including deallocated blocks)
deallocated_space: Number of deallocated blocks. After reclaim process: free_space += deallocated_space.
deallocated_ptrs: Pointers to deleted files. During reclaim process these files are freed.

Figure 20. File Header and Block Header

File Header

File_id	
File_name	
File_ext	
File_ext	
File_size	
Num_pages	

file_id: Unique identification for each file.
file_name: 8 bytes long filename
file_ext: 3 bytes long file extension.
file_size: file size in bytes.
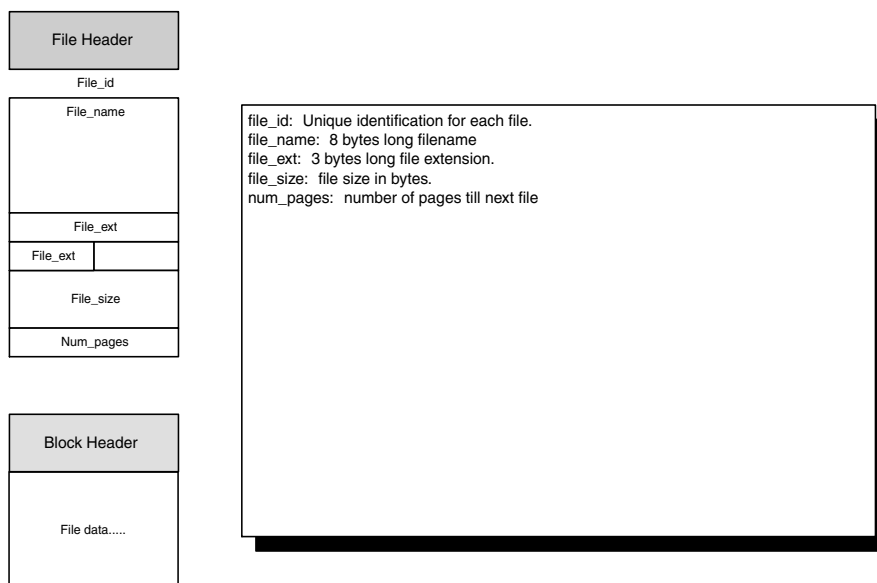num_pages: number of pages till next file

Block Header

File data.....

Figure 19 and Figure 20 show how the media status table and file header is formatted. With this solution a page is only written when a file is written or when deallocated pages are reclaimed. Normally reclaiming occurs only after files equal the DataFlash size has been written. This means that write cycles are almost uniformly distributed over the DataFlash.

Access Time and Throughput Since this file system is primarily constructed for frequent reading and occasional writing, write speed is not prioritized. Therefore only read accesses are discussed here.

To access a given file, the file manager starts a linear search starting with the first file. One read access is needed for each file. The average number of accesses are therefore given by (N is the number of files):

$$n = \frac{N}{2} \qquad (1)$$

The number of pages one file consists of equals:

$$n_p = 1 + \left\lceil \frac{\text{size}}{\text{blocksize}} \right\rceil = 1 + \left\lceil \frac{\text{size}}{264} \right\rceil \qquad (2)$$

which means the number of blocks needed plus the file header. The SPI uses one fourth of the frequency of the microcontroller core and for each SPI clock cycle one bit is read. This gives a access time of approximately 1.86 ms for each page. The average read time for one file is therefore:

$$T = (n + n_p)t_p = \left(\frac{N}{2} + 1 + \left\lceil \frac{\text{size}}{264} \right\rceil \right)(1.86)\text{ms} \qquad (3)$$

The average throughput when reading files with an average size of 10 kB and an average of 30 files is:

$$T = \left(\frac{30}{2} + 1 + \left\lceil \frac{10000}{264} \right\rceil \right)1.86 \text{ ms} = (15 + 1 + 38)1.86 \text{ ms} = 100 \text{ ms} \qquad (4)$$

$$\text{Throughput} = \frac{10 kB}{100 ms} \approx 0.1 \text{ MB/s}$$

The maximum throughputs, with one file which fill the entire DataFlash is 0,14 MB/s.

Writing to the file system will be close to the same efficiency as reading, but if a write triggers the reclaim process, more time is needed.

Ethernet/TCP/IP/ Applications

Every receive event is triggered by an interrupt from the Ethernet controller. This interrupt has the highest priority, so all other activities are stopped immediately. Responses to the packet received are sent within this interrupt.

Data sent from the Transmission Control Protocol are sent periodically with a timer interrupt. Every time the timer interrupt occurs, a counter is incremented. This counter controls when a packet has to be retransmitted.

Applications are running all the time when there is no transmission of packets. This means that when there are several applications, each application has to be activated in a round robin manner. Time and memory sharing has to be taken care of by the programmer.

Limitations

Since the software for this web server has been optimized with regard to both size and speed, there are some limitations to the web server. In the lower layer protocols (Ethernet - IP), only the functionality required to keep the protocols able to respond to normal headers is implemented. TCP is simplified, but almost fully implemented.

Security

Special precautions must be taken when equipment is connected to an insecure network. This becomes even more important if the equipment performs critical operations or contains sensitive information. The AVR embedded web server does not take care of these problems even if the server is capable of controlling critical systems. There are several ways to prevent unauthorized users to access the AVR embedded web server. Some of these methods are discussed in this section.

Limiting Access

The easiest way to prevent other people from accessing applications running on top of TCP is only to listen for connection from one specific client. This can be achieved by giving an IP address when the server starts listening to a port.

    TCPpopen (21, 0xc0a80102)

The example above shows how to make TCP listen for a connection on port 21 (FTP) made by the host with IP address 0xc0a80102 (192.168.1.2). Other requests on this port are rejected. It is also possible to specify several IP addresses that have access to the server. This is done by calling TCPpopen several times.

Even if this method seems secure enough for normal applications, the reader must keep in mind that the connection is not encrypted, so everyone can "listen" to the connection. It is also possible to steal the IP address and log on to the web server from another IP location. When the web server IP address is configured by a DHCP server the IP address will change without notifying the user.

Encryption

A more dynamic approach is to encrypt the communication with the server. Instead of sending data directly to the application on top of the TCP/IP stack, it is sent via an encryption/decryption algorithm. This means that other users are not able to "listen" to the connection and that other users, which do not have the correct key, can not communicate with the server. SSL is an example of such a service. There are also several other implementations available.

Denial of Service (DOS)

Since this server is not intended to serve many users and does not take into account that thousands of connections can be requested at the same time, a hostile user can hang the server by exceeding the maximum load of the server. As long as there are hostile users connected to the same server as the AVR embedded web server; it is not possible to avoid this problem. The best way is to limit the number of users connected to the same network as the server.

Even if it is not possible to be a hundred percent secured, several things can be done to prevent hostile activity. One is to use a firewall between the AVR embedded web server and external users. A firewall can stop other users to enter the network while having the necessary features available for the right users. In combination with some kind of encryption the server should be secure enough for normal applications.

Configuration

The file, server.ini, located in the DataFlash, configures the embedded web server. This file can be written using either serial communication (ymodem) or FTP. Since FTP can only be used when the AVR Embedded Web Server is accessible through Ethernet, serial communication will be the right choice for initial configuration or if the server is not functioning correctly due to a corrupt configuration. Since serial communication uses ymodem, any communication software supporting ymodem can be used. For easy configuration and uploading of server.ini, "ATMEL AVR Embedded Web Server Terminal", which is a graphical front-end, could be used.

Startup

During startup *config*() is called. It will open the server.ini file and read the following options:

- MAC-address
- DHCP enable/disable
- Static IP, if DHCP is disabled

If *config*() is unable to read all the required options, (for example, if the file does not exist) 0 is returned. *DefaultConfig*() is called and the default values specified in config.h will be used.

server.ini

The configuration file should be formatted as follows:

[Header1]

name1=value1

name2=value2

[Header2]

name3=value3

When *config*() is called the following fields are read from server.ini.

Table 9. Settings in server.ini

| Label	Description	Example
MAC0	The 16 MSBs of the MAC-address, should be written in hex.	MAC0=0001
MAC1	Bit 17-32 of the MAC-address, should be written in hex.	MAC1=0A0B
MAC2	The 16 LSBs of the MAC-address, should be written in hex.	MAC2=0C0D
DHCP	DHCP enable/disable (1/0)	DHCP=0
IP0	The 16 MSBs of IP address (if DHCP is disabled), should be written in hex.	IP0=0ABF
IP1	The 16 LSBs of IP address (if DHCP is disabled), should be written in hex.	IP1=FF9D

The fields should be placed under the header [System].

Server.ini can also be used to configure applications; a header for each application should be made. Values can be accessed by using the getOption() function in config.c. Currently only SMTP and FTP use server.ini for configuration. Please read the SMTP and FTP documentation for further details.

Protocol Dependencies

Figure 21 shows the protocol dependencies for the modules in the AVR Embedded Web server. The protocols are described below:

Figure 21. Protocol Dependencies
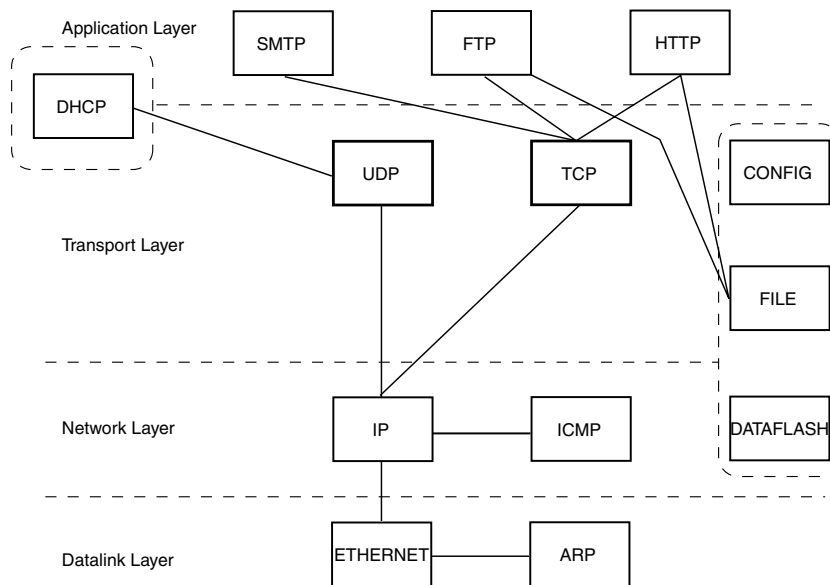
**httpd.c
(Hypertext Transfer Protocol Daemon)**

HTTPD needs TCP and a file-system/DataFlash to operate.

**ftpd.c
File Transfer Protocol Daemon)**

FTPD needs TCP and a file-system/DataFlash to operate.

**smtp.c
(Simple Mail Transfer Protocol)**

SMTP needs a running TCP implementation to operate.

**dhcp.c
(Dynamic Host Configuration Protocol)**

DHCP needs IP and UDP to operate. In the initialization phase DHCP requires that the IP and UDP protocol forward any IP packets delivered before the IP address is configured. The flag dhcpConf is used to signal when DHCP is in initialization phase.

tcp.c (Transmission Control Protocol)

TCP needs IP to operate.

**udp.c
(User Datagram
Protocol)**

UDP needs IP to operate.

**icmp.c
(Internet Control
Message Protocol)**

ICMP needs IP to operate.

**ip.c
(Internet Protocol)**

IP needs ETHERNET to operate.

**arp.c
(Address Resolution
Protocol)**

ARP needs ETHERNET to operate

**ethernet.c (Ethernet
Controller Driver)**

ETHERNET depends only on the hardware Ethernet controller.

**config.c (Automatic
Configuration Program)**

CONFIG needs file-system/DataFlash to read the configuration file. If the configuration file is unavailable it can return standard values.

**file.c
(File System)**

FILE needs DATAFLASH to work.

**dataflash.c (DataFlash
Interface)**

DATAFLASH is only dependent on the hardware DataFlash.

**main.c
(Main Loop)**

Main Loop

MAIN must initialize Ethernet, DataFlash, file-system, TCP, DHCP and HTTPD if these protocols are to be used.

DHCP, FTPD and HTTPD require repeatedly polling to operate.

Figure 22. Schematic
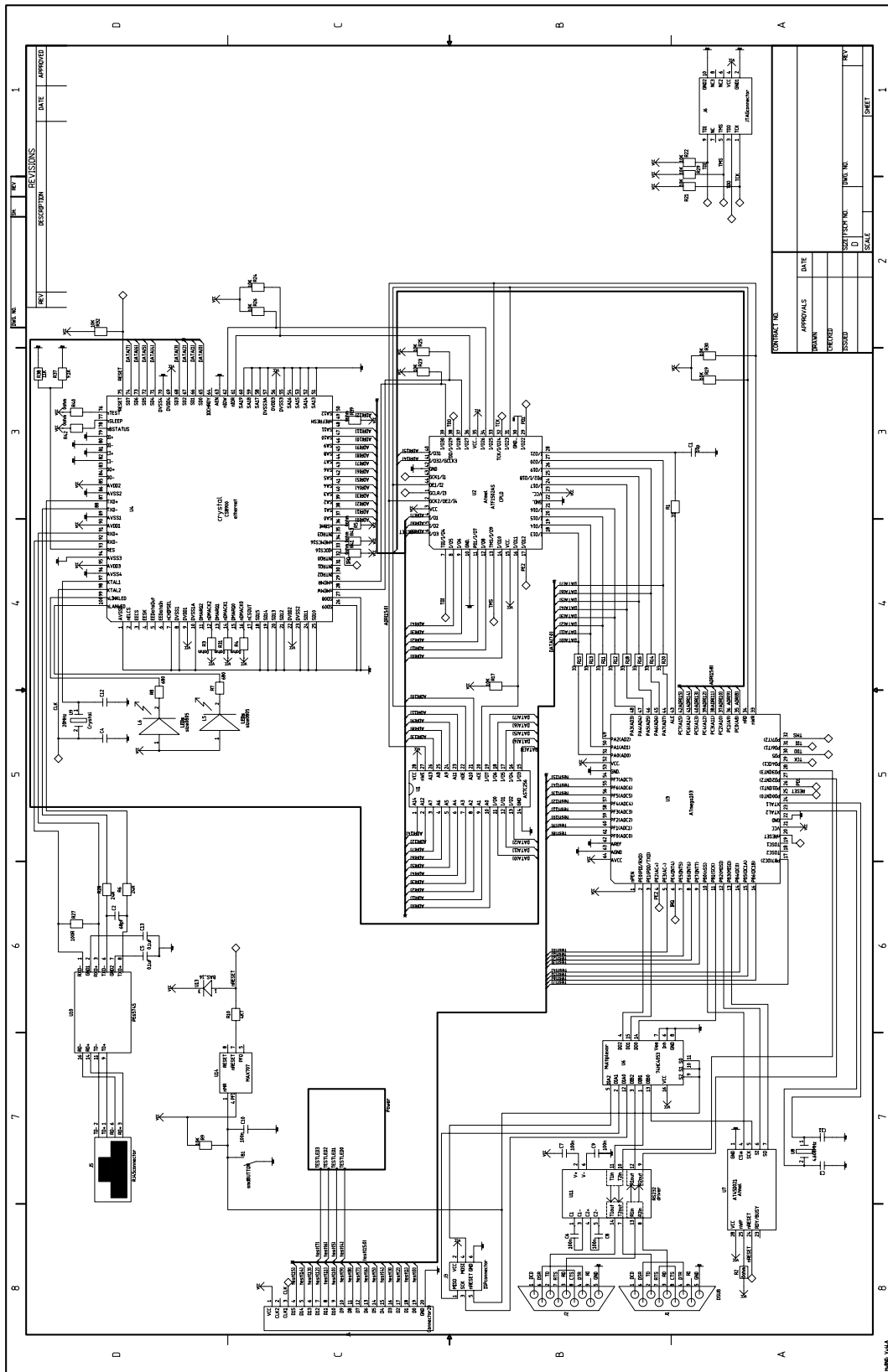
Pin Description

Table 10. Atmel ATmega103

Pin Number	Pin Name	Description
1	PEN	Programming Enable for low-voltage serial programming mode. Tied high.
2	PE0(PDI/RXD)	Program Data In for ISP and receive for UART1. Connected to multiplexer, which switches between the two devices.
3	PE1(PDO/TXD)	Program Data Out for ISP and transmit for UART1. Connected to multiplexer, which switches between the two devices.
4	PE2	Connected to CPLD for future expansions.
5, 7, 8, 9, 14, 15, 16, 17	PE3, PE5, PE6, PE7, PB4, PB5, PB6, PB7	Output[0:7]. Connected to 10x2 connector. Can be used to connect another card to the web server. These pins may also be used as input.
6	PE4(INT4)	Interrupt request line to Ethernet controller.
10	PB0($\overline{\text{SS}}$)	Slave Select for Serial Peripheral Interface. Connected to DataFlash.
11	PB1(SCK)	Serial clock input for ISP and serial clock output for SPI. Connected to multiplexer, which switches between the two devices.
12	PB2(MOSI)	Master Output for SPI. Connected to DataFlash.
13	PB3(MISO)	Master Input for SPI. Connected to DataFlash.
18	TOSC2	Not Connected
19	TOSC1	Not Connected
20	RESET	Reset signal for microcontroller. Connected to MAX 707 reset circuit.
23	XTAL2	Output crystal oscillator.
24	XTAL1	Input crystal oscillator.
25	PD0(INT0)	Reset for Ethernet controller.
26	PD1(INT1)	Connected to CPLD for future expansions.
27	PD2(INT2)	UART2 receive input.
28	PD3(INT3)	UART2 transmit output.
29	PD4(IC1)	TCK. JTAG clock. Connected to CPLD for programming the CPLD.
30	PD5	TDO. JTAG. Connected to CPLD for programming the CPLD.
31	PD6(T1)	TDI. JTAG. Connected to CPLD for programming the CPLD.
32	PD7(T2)	TMS. JTAG. Connected to CPLD for programming the CPLD.
33	WR	External SRAM write signal. Connected to CPLD.
34	RD	External SRAM read signal. Connected to CPLD.
35 - 42	PC[0:7]	Address 8 up to 15.

Table 10. Atmel ATmega103 (Continued)

Pin Number	Pin Name	Description
43	ALE	Address Latch Enable. Connected to CPLD.
44 - 51	PA[7:0]	Address 7 down to 0 and Data 7 down to 0. Connected to CPLD, SRAM and Ethernet controller.
54 - 61	PF[7:0]	Input[0:7]. Connected to 10x2 connector. Can be used to connect another card to the web server. This port also features an ADC (Analog to Digital Converter). Can only be used as input.
62	AREF	Analog reference voltage.

Table 11. Crystal CS8900

Pin Number	Pin Name	Description
2, 3, 4, 5, 6, 17	ELCS, EECS, EESK, EEDataOut, EEDataIn, CSOUT	EEPROM and Boot PROM interface. Not connected.
7	CHIPSEL	Used together with external leachable address bus decode logic. Tied low.
11, 13, 15	DMARQ[2:0]	DMA request. Not connected.
12, 14, 16	DMACK[2:0]	DMA Acknowledge. Active low. Tied high.
18, 19, 20, 21, 24, 25, 26, 27	SD[15:8]	Data[15:8]. Not used. Tied low.
28	MEMW	Memory mode write. Connected to CPLD.
29	MEMR	Memory mode read. Connected to CPLD.
30, 31, 35	INTRQ[3:1]	Interrupt request line 3 down to 1. Not connected.
32	INTRQ0	Interrupt request line 0. Connected to microcontroller.
33	IOCS16	I/O Chip Select 16-bit. Output generated by CS8900 when it recognizes an address on the ISA bus that corresponds to its assign I/O space.
34	MEMCS16	Memory Chip Select 16-bit. Output generated by CS8900 when it recognizes an address on the ISA bus that corresponds to its assign memory space.
36	SBHE	System Bus High Enable. Active-low input indicating data on SD[15:8]. Connected to CPLD.
37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 50, 51, 52, 53, 54, 58, 59, 60	SA[0:19]	Address[0:15]. Only address[0:12] connected to address bus. Rest of pins are tied low.
49	REFRESH	Active-low input indicating that a DRAM refresh cycle is in progress. Tied high.
61	IOW	I/O mode write. Connected to CPLD.
62	IOR	I/O mode read. Connected to CPLD.

Table 11. Crystal CS8900 (Continued)

Pin Number	Pin Name	Description
63	AEN	Address Enable. Active-high input indicating that the system DMA controller has control of the ISA bus. Connected to CPLD.
64	IOCHRDY	I/O Channel Ready. Extends read cycles to the CS8900. Not connected.
65, 66, 67, 68, 71, 72, 73, 74	SD[0:7]	Data[0:7]. Connected to Databus.
76	TEST	Test select. Tied high.
77	SLEEP	Hardware sleep. Tied high.
78	BSTATUS	Not connected.
79, 80, 81, 82, 83, 84	AUI	Attachment Unit Interface. Not connected.
87, 88	TXD+, TXD-	10BASE-T Transmit. Differential output. Connected to isolation transformer.
91, 92	RXD+, RXD-	10BASE-T Receive. Differential input. Connected to isolation transformer.
93	RES	Reference Resistor. Connected to a 4.99 ohm resistor connected to ground.
97	XTAL1	Crystal Oscillator input.
98	XTAL2	Crystal Oscillator output.
99	LINKLED	Active-low when CS8900 detects valid link pulses. Connected to LED.
100	LANLED	Active-low when CS8900 detects LAN activity. Connected to LED.

Table 12. Atmel ATF1502AS

Pin Number	Pin Name	Description
1	GCLR	Global Clear. Connected to global reset signal. Resets the state of flipflops.
2	WR	Write input from microcontroller. Used to generate write signals and chip enable signals.
4, 5, 6, 8, 9, 11, 12, 14	ADR[7:0]	Address[7:0] from address latch. Connected to address bus.
7	TDI	Test Data Input. Connected to microcontroller and JTAG.
13	TMS	Connected to microcontroller and JTAG.
17	I/O	Connected to PE2 on microcontroller for future expansions.
18, 19, 20, 21, 24, 25, 26, 27	ADR/Data[0:7]	Address and data[0:7]. Address to address latch. Connected to data bus.
28	ALE	Address Latch Enable.

Table 12. Atmel ATF1502AS (Continued)

Pin Number	Pin Name	Description
29	I/O	Connected to PD1 on microcontroller for future expansions.
32	TCK	Connected to microcontroller and JTAG.
34	IOW	I/O write. Connected to $\overline{\text{IOW}}$ on Ethernet controller.
36	IOR	I/O read. Connected to $\overline{\text{IOR}}$ on Ethernet controller.
37	MEMW	Memory write. Connected to $\overline{\text{MEMW}}$ on Ethernet controller.
38	TDO	Test Data Output. Connected to microcontroller and JTAG.
39	MEMR	Memory read. Connected to $\overline{\text{MEMR}}$ on Ethernet controller.
40, 41	ADR[15:14]	Address[15:14]. Used for address decoding.
43	GCK	Global clock. Connected to 20 MHz crystal.
44	RD	Read input from microcontroller. Used to generate read signals and chip enable signals.

Components

Table 13. Components

Ref.des	Type	Size/Description	Package
U1	AS7C256	SRAM	SOJ-28
U2	ATF1502AS	CPLD	PLCC-44
U3	ATmega103	Microcontroller	TQFP-64
U4	CS8900	Ethernet Controller	TQFP-100
U5	L78M05CDT	Voltage Regulator	TO252
U6	74HC4053	Multiplexer	SO-16
U7	AT45D021	DataFlash	SOIC_28
XC2	Crystal	4.608 MHz	HC49/4H
XC1	Crystal	20 MHz	HC49/4H
T10	PE65745	Isolation Transformer	SMT4
U11	MAX202CSE	RS232 Driver	SO16N
D7	DF10S	Rectifier	
D8	BAS_16	Protection Diode	SOT23
U14	MAX707CSA	Reset Circuit	SO-8
J1	DSUB9-2	RS232	
J2	DSUB9-1	RS232	
J3	ISP	ISP Connector	6 Pins
J4	HPpod	2x10 Connector	20 Pins
J5	RJ45	RJ45 Connector	8/8
J6	JTAG	JTAG Connector	10 Pins
J7	Power	Power Connector	
R1	Resistor 0805	33 Ohms	0805
R2	Resistor 0805	0 Ohms	0805
R3	Resistor 0805	0 Ohms	0805
R4	Resistor 0805	0 Ohms	0805
R5	Resistor 0805	0 Ohms	0805
R6	Resistor 0805	24 Ohms	0805
R7	Resistor 0805	680 Ohms	0805
R8	Resistor 0805	680 Ohms	0805
R9	Resistor 0805	10K Ohms	0805
R10	Resistor 0805	4.7K Ohms	0805
R11	Resistor 0805	33 Ohms	0805
R12	Resistor 0805	33 Ohms	0805
R13	Resistor 0805	33 Ohms	0805
R14	Resistor 0805	33 Ohms	0805

Table 13. Components (Continued)

Ref.des	Type	Size/Description	Package
R15	Resistor 0805	33 Ohms	0805
R16	Resistor 0805	33 Ohms	0805
R17	Resistor 0805	10K Ohms	0805
R18	Resistor 0805	33 Ohms	0805
R19	Resistor 0805	10K Ohms	0805
R20	Resistor 0805	33 Ohms	0805
R21	Resistor 0805	10K Ohms	0805
R22	Resistor 0805	10K Ohms	0805
R23	Resistor 0805	10K Ohms	0805
R24	Resistor 0805	10K Ohms	0805
R25	Resistor 0805	10K Ohms	0805
R26	Resistor 0805	10K Ohms	0805
R27	Resistor 0805	100 Ohms	0805
R28	Resistor 0805	24 Ohms	0805
R29	Resistor 0805	10K Ohms	0805
R30	Resistor 0805	10K Ohms	0805
R31	Resistor 0805	0 Ohms	0805
R32	Resistor 0805	10K Ohms	0805
R33	Resistor 0805	680 Ohms	0805
R34	Resistor 0805	680 Ohms	0805
R35	Resistor 0805	680 Ohms	0805
R36	Resistor 0805	680 Ohms	0805
R37	Resistor 0805	9.1K Ohms	0805
R38	Resistor 0805	11K Ohms	0805
R39	Resistor 0805	0 Ohms	0805
R40	Resistor 0805	0 Ohms	0805
R41	Resistor 0805	0 Ohms	0805
R42	Resistor 0805	0 Ohms	0805
R43	Resistor 0805	0 Ohms	0805
C1	Capacitor 0805	10 pF	0805
C2	Capacitor 0805	68 pF	0805
C3	Capacitor 0805	18 pF	0805
C4	Capacitor 0805	18 pF	0805
C5	Capacitor 0805	100 nF	0805
C6	Capacitor 0805	100 nF	0805
C7	Capacitor 0805	100 nF	0805

Table 13. Components (Continued)

Ref.des	Type	Size/Description	Package
C8	Capacitor 0805	100 nF	0805
C9	Capacitor 0805	100 nF	0805
C10	Capacitor 0805	100 nF	0805
C11	Capacitor 0805	18 pF	0805
C12	Capacitor 0805	18 pF	0805
C13	Capacitor 0805	100 nF	0805
C14	Capacitor 0805	100 nF	0805
C15	Capacitor 0805	100 nF	0805
C16	Capacitor 0805	100 nF	0805
C17	Capacitor 0805	100 nF	0805
C18	Capacitor 0805	100 nF	0805
C19	Capacitor 0805	100 nF	0805
C20	Capacitor 0805	100 nF	0805
C21	Capacitor 0805	100 nF	0805
C22	Capacitor 0805	100 nF	0805
C23	Capacitor 0805	100 nF	0805
C24	Capacitor 0805	100 nF	0805
C25	Capacitor 0805	100 nF	0805
C26	Capacitor 0805	100 nF	0805
C27	Capacitor 0805	100 nF	0805
C28	Capacitor 0805	100 nF	0805
C29	Capacitor 0805	100 nF	0805
C30	Capacitor 0805	100 nF	0805
C31	Tantalum Capacitor 7343	220 µF	7343
C32	Tantalum Capacitor 3216	470 nF	3216
L1	LED 0805	Red	0805
L2	LED 0805	Red	0805
L3	LED 0805	Red	0805
L4	LED 0805	Red	0805
L5	LED 0805	Yellow	0805
L6	LED 0805	Yellow	0805
B1	Button	SKHUAD	SKHUAD

Appendix 1: C-code Reference

Ethernet

initEthernet

Name: initEthernet - initialize the Ethernet controller.

Usage: #include "ethernet.h"
void initEthernet(void);

Prototype In: ethernet.h

Description: initEthernet initializes the Ethernet controller. initEthernet configures the Ethernet controller with interrupt on packets with individual address or broadcasts, maps the memory of the Ethernet controller into the microcontroller's memory and gives the Ethernet controller a unique address.

receiveEvent

Name: receiveEvent - reads a frame from the Ethernet controller.

Usage: #include "ethernet.h"
void receiveEvent(void);

Prototype In: ethernet.h

Description: receiveEvent should be called everytime a packet is received on the Ethernet controller (at interrupt 4). receiveEvent reads the packet and gives the control to either ARP or IP depending on the protocol number in the header field.

sendFrame

Name: sendFrame - write the frame to the Ethernet controller for sending.

Usage: #include "ethernet.h"
char sendFrame(unsigned int *length*, unsigned int *type*, unsigned int *ip0*, unsigned int *ip1*);

Prototype In: ethernet.h

Description: sendFrame sends *length* bytes located in the global frame buffer. *type* is used to denote the protocol, which sent the frame. *ip0* and *ip1* are used to find the Ethernet address of the recipient. If the Ethernet address is not present in the ARP table, the frame is discarded and an ARP request is sent to the IP address. The sender is responsible to retransmit the frame upon failure.

Return Value

sendFrame returns 1 upon success and 0 upon failure.

getMAC

Name: getMAC - check if Ethernet address is present in the ARP table.

Usage: #include "ethernet.h"
char getMAC(unsigned int *ip0,* unsigned int *ip1*);

Prototype In: ethernet.h

Description: getMAC checks if the Ethernet (Media Access Control) address to the given IP address is present in the ARP table. If the address is not present, an ARP request is sent.

Return Value: If the Ethernet address is present in the ARP table, 1 is returned, else 0.

dhcpMAC

Name: dhcpMAC - store the Ethernet address of the dhcp server permanently.

Usage: #include "ethernet.h"
void dhcpMAC(unsigned int *ip0*, unsigned int *ip1*);

Prototype In: ethernet.h

Description: dhcpMAC is called the first time a dhcp offer is received. The Ethernet address present in the frame is stored permanently for later use.

ARP

receiveARP

Name: receiveARP - handles an ARP request.

Usage: #include "arp.h"
void receiveARP(void);

Prototype In: arp.h

Description: receiveARP is called by the link layer (Ethernet, slip) when the frame received is an ARP request. receiveARP change the request to a reply and calls sendFrame to send the frame.

sendARP

Name: sendARP - send an ARP request.

Usage: #include "arp.h"
void sendARP (unsigned int *ip0*, unsigned int *ip1*);

Prototype In: arp.h

Description: sendARP makes an ARP request requesting the IP address and calls sendFrame to send the frame

IP

transmitIP

Name: transmitIP - fills in IP header and sends the frame to link layer.

Usage: #include "ip.h"
void transmitIP(int length, unsigned int ip0, unsigned int ip1, char type);

Prototype In: ip.h

Description: transmitIP is called from transmit layer when a frame needs to be sent. transmitIP makes an IP header and calls the link layer to send the frame.

receiveIP

Name: receiveIP - receive an IP packet.

Usage: #include "ip.h"
void receiveIP(unsigned int length);

Prototype In: ip.h

Description: receiveIP must be called from the link layer (Ethernet) when an IP packet is received. receiveIP checks the IP header and calls ICMP, TCP or UDP.

ICMP

receiveICMP

Name: receiveICMP - receive an ICMP packet.

Usage: #include "icmp.h"

void receiveICMP(unsigned int *frame, int length, unsigned int ip0, unsigned int ip1);

Prototype In: icmp.h

Description: receiveICMP must be called by IP when an ICMP packet is received. receiveICMP checks the ICMP header fields and responds to requests.

UDP

receiveUDP

Name: receiveUDP - receive an UDP datagram from IP.

Usage: Void receiveUDP(unsigned int UDPStart, unsigned int UDPLength, unsigned int hisIP0, unsigned int hisIP1, unsigned int myIP0, unsigned int myIP1).

Prototype In: udp.h

Description: receiveUDP is called from the IP layer, informing the UDP layer that data is available in the global buffer frame. The data is copied from the frame to a buffer in the UDP layer.

UDPStart is the position in the frame where the UDP datagram starts.

UDPLength is the length of the UDP datagram.

hisIP0 is the 16 MSBs of the sender's IP address.

hisIP1 is the 16 LSBs of the sender's IP address.

myIP0 is the 16 MSBs of the receiver's IP address, as it appears in the received IP-header.

myIP1 is the 16 LSBs of the receiver's IP address, as it appears in the received IP-header.

The reason for passing the receiver's IP address to the UDP layer is when calculating the checksum, the receiver's IP address must be known. During DHCP configuration of the client, the client hasn't received an IP, and needs the IP address from the IP-header to calculate the checksum correctly.

sendUDP

Name: sendUDP - send a UDP-datagram.

Usage: Unsigned char sendUDP(unsigned char * data, unsigned int dataLength, unsigned int hisIP0, unsigned int hisIP1, unsigned int myPort, unsigned int hisPort).

Prototype In: udp.h

Description: SendUDP sends an UDP datagram.

data - a pointer to the data to be sent.

dataLength - the length of the data.

hisIP0 - the 16 MSBs of the destination IP address.

hisIP1 - the 16 LSBs of the destination IP address.

myPort - the sender's port.

hisPort - the destination port.

Return Value: If the datagram has been sent, a 1 is returned; if unsuccessful, 0 is returned. If the specified destination is not in the ARP- table, an ARP request is sent and 0 is returned; the datagram must then be retransmitted later.

readUDP

Name: readUDP - poll the UDP buffer.

Usage: Unsigned char readUDP(unsigned int port, UDPB * appBuffer).

Prototype In: udp.h

Description: Called by an application to check for new data. If new data is found it is copied into the application buffer.

AppBuffer is a pointer to the buffer in which the data should be copied.

Return Value: Returns 1 if new data has been found and copied to the buffer; returns 0 otherwise.

TCP

TCPpopen

Name: TCPpopen - open a port for listening

Usage: #include "tcp.h"
char TCPpopen (unsigned int *port*, unsigned long *ip*);

Prototype In: tcp.h

Description: TCPpopen opens a port and starts listening for connections. *port* denotes the port number to listen to and *ip* denotes the IP, which is allowed to use this port. If *ip* is 0, there are no limitations on the IP address.

Return Value: On success TCPpopen returns 1 and on failure 0.

TCPaopen

Name: TCPaopen - actively open a connection on the given port.

Usage: #include "tcp.h"
SOCKET *TCPaopen (unsigned int *port*, unsigned int *ip0*, unsigned int *ip1*, unsigned int *myport*);

Prototype In: tcp.h

Description: TCPaopen actively opens a connection to the given IP and *port*, from *MYport*. If *MYport* is 0, a portnumber is assigned by TCPaopen. Once TCPaopen has completed, data can be sent using TCPsend.

Return Value: TCPaopen returns a pointer to the newly opened socket on success and a null pointer on failure.

TCPlistenPort

Name: TCPlistenPort - check to see if a port is listened to.

Usage: #include "tcp.h"
char TCPlistenPort(unsigned int *port*, unsigned long *ip*);

Prototype In: tcp.h

Description: TCPlistenPort reports if there is a socket on the given *port*. *port* denotes the port number and *ip* denotes the IP address.

Return Value: If the port is listened to, TCPlistenPort returns 1, else 0.

TCPstop

Name: TCPstop - stop listening to a port.

Usage: #include "tcp.h"
void TCPstop(unsigned int *port*, unsigned long *ip*);

Prototype In: tcp.h

Description: TCPstop stops listening to a given *port*. *port* denotes the port and *ip* denotes the IP address (see TCPpopen).

TCPfindSockets

Name: TCPfindSockets - find connections on a given port.

Usage: #include "tcp.h"
SOCKET *TCPfindSockets(unsigned int *port*);

Prototype In: tcp.h

Description: TCPfindSockets finds all connections on a given *port*. If there is more than one connection, a linked list of sockets is returned. The next *socket* in the list is found at *socket->next*.

Return Value: TCPfindSockets returns a pointer to the first *socket* on the *port* if there is existing connections, else a null pointer is returned.

TCPget

Name: TCPget - read incoming data from TCP buffer.

Usage: #include "tcp.h"
int TCPget(SOCKET *socket*, int *maxSize*, char *buffer*);

Prototype In: tcp.h

Description: TCPget reads data from the TCP buffer. The pointer to current position is moved and buffer space is released for new data. *socket* denotes the socket to read from, *maxSize* denotes the maximum size to read and *buffer* is a pointer to the buffer where the data should be stored.

Return Value: TCPget returns the number of bytes read from TCP buffer. If there is no new data in the buffer, 0 is returned.

TCPread

Name: TCPread - read incoming data from TCP buffer without moving pointer.

Usage: #include "tcp.h"
int TCPread(SOCKET *socket*, int *maxSize*, char *buffer*, char *reset*);

Prototype In: tcp.h

Description: TCPread reads data from the TCP buffer. The pointer to current position is not moved. Instead a temporary pointer shows where to read from the next time TCPread is called. TCPread should be called with a positive value as *reset* the first time it is called. A positive *reset* value sets the temporarily pointer to the same position as the current position pointer. (see TCPget)

Return Value: TCPread returns the number of bytes read.

TCPreadln

Name: TCPreadln - read one line from the TCP buffer.

Usage: #include "tcp.h"
int TCPreadln(SOCKET *socket, int maxSize, char *buffer, char reset);

Prototype In: tcp.h

Description: TCPreadln works the same way as TCPread except that TCPreadln reads one line at a time. If the line is too long for the *buffer*, only the *maxSize* bytes of the TCP buffer is returned. Next time TCPreadln is called, the rest of the line is read.

Return Value: TCPreadln returns the number of bytes read.

TCPsend

Name: TCPsend - write data in TCP buffer for sending.

Usage: #include "tcp.h"
int TCPsend(SOCKET *socket, int size, char *buffer);

Prototype In: tcp.h

Description: TCPsend write *size* bytes from *buffer* to TCP buffer for sending on *socket*.

Return Value: TCPsend returns the number of bytes sent on success. On failure 0 is returned.

TCPsend_P

Name: TCPsend_P - write data from program space to TCP buffer for sending.

Usage: #include "tcp.h"
int TCPsend_P(SOCKET *socket, int size, char flash *buffer);

Prototype In: tcp.h

Description: TCPsend_P writes *size* bytes from program memory buffer to TCP buffer for sending on *socket*.

Return Value: TCPsend_P returns the number of bytes sent on success. On failure, 0 is returned.

TCPclose

Name: TCPclose - close a connection.

Usage: #include "tcp.h"
void TCPclose(SOCKET *socket);

Prototype In: tcp.h

Description: TCPclose closes the connection on *socket*. If the TCP state is established, TCP waits until all data is sent and enters fin_wait1. If the TCP state is close_wait, TCP waits until all data are sent and enters last_ack state. When all data are sent and both hosts have closed the connection, *socket* is deleted and prepared for a new connection.

TCPsize

Name: TCPsize - return number of unread bytes in TCP buffer.

Usage: #include "tcp.h"
int TCPsize(SOCKET *socket);

Prototype In: tcp.h

Description: TCPsize returns the amount of unread data in TCP buffer. After a TCPget call, the size is reduced by the size TCPget returned.

Return Value: TCPsize returns the number of bytes in TCP buffer, if the socket does not exist or there are no data in TCP buffer, 0 is returned.

TCPbufferSpace

Name: TCPbufferSpace - return free space in TCP buffer.

Usage: #include "tcp.h"
int TCPbufferSpace(SOCKET *socket);

Prototype In: tcp.h

Description: TCPbufferSpace returns amount of free space in TCP buffer. TCPbuffer-Space could be called before TCPsend to ensure that there is room for the data written by TCPsend.

Return Value: TCPbufferSpace returns number of bytes free in TCP buffer.

TCPabort

Name: TCPabort - abort a connection.

Usage: #include "tcp.h"
void TCPabort(SOCKET *socket);

Prototype In: tcp.h

Description: TCPabort aborts the connection on *socket*. A reset is sent to the other host before the *socket* is deleted.

TCPinit

Name: TCPinit - initialize TCP.

Usage: #include "tcp.h"
void TCPinit(void);

Prototype In: tcp.h

Description: TCPinit must be called before TCP can start.

checkTCP

Name: checkTCP - check the connections and send unsent frames.

Usage: #include "tcp.h"
void checkTCP(void);

Prototype In: tcp.h

Description: checkTCP checks all sockets to see if there is some data to be sent or if a connection should be closed. checkTCP also checks the retransmission buffer to see if there are any frames that should be sent or deleted from the retransmission buffer.

receiveTCP

Name: receiveTCP - receive a TCP frame.

Usage: #include "tcp.h"
void receiveTCP(int *length*, unsigned int *ip0*, unsigned int *ip1*);

Prototype In: tcp.h

Description: receiveTCP must be called by IP every time a TCP frame is received. receiveTCP responds to TCP control data and acknowledges received data.

FTP daemon

ftpd

Name: ftpd - file transfer protocol daemon.

Usage: void ftpd (void)

Prototype In: ftpd.h

Description: ftpd should be a part of the main loop. Everytime ftpd is called, ftpd checks if there is a FTP connection and controls that connection. ftpd also takes care of the FTP data connection.

DHCP

DHCP

Name: DHCP - configure and maintain the DHCP client.

Usage: void DHCP(void);

Prototype In: dhcp.h

Description: The initial call to DHCP configures the server with an IP address and a lease time for that address. When configuration is done DHCP must be called repeatedly to monitor the DHCP finite state machine. The state machine is responsible for renewal of lease time when time is due.

checkDHCP

Name: checkDHCP - DHCP timer function.

Usage: void checkDHCP(void);

Prototype In: dhcp.h

Description: checkDHCP must be called within a timer interrupt every time a timer overflow occurs, between constant time intervals. To compensate for the frequency of which checkDHCP is called, DHCP_MINUTE must be adjusted accordingly.

HTTP

httpdInit

Name: httpdInit - initialize the HTTP server.

Usage: void httpdInit(void)

Prototype In: httpd.h

Description: httpdInit must be called to initialize the HTTP server.

httpd

Name: httpd - HTTP daemon

Usage: void httpd(void)

Prototype In: httpd.h

Description: The HTTP daemon listens to TCP port 80 and processes incoming GET and POST requests. To run the daemon httpd must be polled in the main program.

SMTP

sendMail

Name: sendMail - send a mail.

Usage: unsigned char sendMail(unsigned char * *subject*, unsigned char * *message*)

Description: Sends a mail to the mail address specified in the server.ini file, using the mail server specified in the server.ini file.

subject is a pointer to the subject, the string must end with a '\0'

message is a pointer to the message, the string must end with a '\0'

Note that maximum length of the two strings is dependent upon the chosen buffer size in the sendMail function.

Return Value: sendMail returns 1 if the mail has been successfully sent, 0 otherwise

DataFlash

read_page

Name: read_page - read a page from DataFlash.

Usage: #include "dataflash.h"
char read_page(int *pageNr*, char **buffer*, int *length*);

Prototype In: dataflash.h

Description: read_page reads *length* bytes from *pageNr* in DataFlash and writes the data into *buffer*. *pageNr* must be less than the number of pages in DataFlash.

Return Value: read_page returns 1 upon success.

write_page

Name: write_page - write buffer to page in DataFlash.

Usage: #include "dataflash.h"
char write_page (int *pageNr*, char **buffer*);

Prototype In: dataflash.h

Description: write_page writes 264 bytes from *buffer* to *pageNr* in DataFlash. *pageNr* must be less than the number of pages in DataFlash.

copy_page

Name: copy_page - copy a page in DataFlash to another page in DataFlash.

Usage: #include "dataflash.h"
char copy_page (int *toPage*, int *fromPage*);

Prototype In: dataflash.h

Description: copy_page copy the contents of *fromPage* to *toPage*.

Return Value: copy_page returns 1 on success.

EEput

Name: EEput - write one byte to EEPROM.

Usage: #include "dataflash.h"
void EEput(int *uiAddress*, char *cValue*);

Prototype In: dataflash.h

Description: EEput performs a secure write to EEPROM. *cValue* is written to *uiAddress*.

EEget

Name: EEget - read one byte from EEPROM.

Usage: #include "dataflash.h"
char EEget (int *uiAddress*);

Prototype In: dataflash.h

Description: EEget preforms a secure read from EEPROM.

Return Value: EEget returns the byte read.

File System

fopen

Name: fopen - open a stream.

Usage: #include "file.h"
FILE *fopen(char *filename*, char *type*);

Prototype In: file.h

Description: fopen opens the file named by filename and associates a *stream* with it. fopen returns a pointer to be used to identify the stream in subsequent operations.

The type string used in each of these calls is one of the following values:

r   Open for reading only.

w   Create for writing.

Return Value: On successful completion, fopen returns the newly opened *stream*. In the event of error, fopen returns NULL.

fclose

Name: fclose - close a stream.

Usage: #include "file.h"
char fclose(FILE *stream*);

Prototype In: file.h

Description: fclose closes the named *stream*. Buffers associated with the *stream* are flushed upon closing.

Return Value: fclose returns 1 on success and 0 on failure.

fget

Name: fget - get character from *stream.*

Usage: #include "file.h"
char fget(FILE *stream*);

Prototype In: file.h

Description: On success fget returns the character read from the stream. On end-of-file or error, 0 is returned.

fput

Name: fput - put a character to a stream.

Usage: #include "file.h"
void fput(FILE *stream*, const char *ch*);

Prototype In: file.h

Description: On success fput puts a character to the *stream*. If the file system is unable to write the character to the DataFlash, no indication of this is returned to the user.

fdelete

Name: <u>fdelete</u> - delete a file.

Usage: #include "file.h"
char <u>fdelete</u>(char *_filename_);

Prototype In: file.h

Description: <u>fdelete</u> deletes the file pointed to by _filename_.

Return Value: <u>fdelete</u> returns 1 upon success. If the file could not be deleted, 0 is returned.

format

Name: <u>format</u> - format the DataFlash.

Usage: #include "file.h"
void <u>format</u> (void);

Prototype In: file.h

Description: <u>format</u> overwrites the media status table containing information about the files in the file system, and makes all the space in the file system available for new files. Old files are not physically deleted, but the references to the files are removed.

read

Name: <u>read</u> - reads from file.

Usage: #include "file.h"
int <u>read</u> (FILE *_stream_, char *_buffer_, int _size_);

Prototype In: file.h

Description: <u>read</u> attempts to read _size_ bytes from the file. read uses <u>fget</u> to read one byte from the file several times to _buffer_.

Return Value: <u>read</u> returns the number of bytes read. If <u>read</u> is not able to read from the file because the file is not open or the end-of-file is reached, 0 is returned.

write

Name: <u>write</u> - write to file.

Usage: #include "file.h"
int <u>write</u>(FILE *_stream_, char *_buffer_, int _size_);

Prototype In: file.h

Description: <u>write</u> attempts to write _size_ bytes from _buffer_ to the file. write uses <u>fput</u> to write one byte at a time from _buffer_.

Return Value: <u>write</u> returns the number of bytes successfully written. If <u>write</u> is not able to write to the file, 0 is returned.

readln

Name: <u>readln</u> - read one line from file.

Usage: #include "file.h"
int <u>readln</u>(FILE *_stream_, char *_buffer_, int _size_);

Prototype In: file.h

Description: <u>readln</u> reads one line from the file. If the line exceeds _size_, only the first _size_ bytes of the stream is read. The pointer is moved and next time <u>readln</u> is called, the next _size_ bytes of the same line is read.

Return Value: <u>readln</u> returns the number of bytes read. If <u>readln</u> is not able to read from the file because the file is not open or the end-of-file is reached, 0 is returned.

feof

Name: feof - check if end-of-file is reached.

Usage: #include "file.h"
char feof(FILE *stream);

Prototype In: file.h

Description: feof check if the end-of-file (EOF) is reached.

Return Value: feof returns 1 if EOF is reached, else 0.

finit

Name: finit - initialize the file system.

Usage: #include "file.h"
char finit(void);

Prototype In: file.h

Description: finit initializes the file system and must be called before the file system is used.

Return Value: finit returns 1 one successful initialization, else 0.

findFile

Name: findFile - find a file on the DataFlash.

Usage: #include "file.h"
int findFile(char *filename, char *fileext);

Prototype In: file.h

Description: findFile finds the file specified by *filename* and *fileext*. findFile can be used to check if a file is present before opening the file for reading.

Return Value: findFile returns the page number where the file header for the file is located on success and FNULL (0xffff) on failure.

opendir

Name: opendir - open the directory for reading.

Usage: #include "file.h"
DIR *opendir(void);

Prototype In: file.h

Description: opendir opens the directory listing for reading. Only one directory can be open at a time.

Return Value: fopendir returns a pointer to the directory stream on success. If the opendir is not able to open the directory for reading, NULL is returned.

readdir

Name: readdir - output one line of the directory listing.

Usage: #include "file.h"
char *readdir(DIR *_dir);

Prototype In: file.h

Description: readdir reads one filename each time it is called. To read a whole directory, readdir must be called several times until the whole directory listing is read.

Return Value: readdir returns a pointer to a null terminated string containing a filename and CRLF. When the whole listing is read, a null pointer is returned.

readwdir

Name: readwdir - output one line of the directory listing.

Usage: ##include "file.h"
char *readwdir(DIR *_dir);

Prototype In: file.h

Description: readwdir reads one filename and the information about that file. After all the files are read, the amount of free space is returned. See readdir.

Return Value: readwdir returns a pointer to a null terminated string containing a filename and CRLF. After the whole listing is read, a pointer to a null terminated string containing "Free space: xxxx" where x is the size of free space. If readwdir is called another time, a null pointer is returned.

rewinddir

Name: rewinddir - reset the pointer to directory listing.

Usage: #include "file.h"
void rewinddir(DIR *_dir);

Prototype In: file.h

Description: rewinddir reset the pointer to the listing so that next time readdir or readwdir is called, the first file in the listing is returned.

closedir

Name: closedir - closes the directory stream.

Usage: #include "file.h"
char closedir(DIR *_dir);

Prototype In: file.h

Description: closedir - close the directory stream.

Return Value: closedir returns 1 on success and 0 on failure.

Bibliography

[EME99]   Gary Desrosiers, 99. Embedded Ethernet. http://www.embeddedether-net.com/ [Accessed Feb 2000].

[TCP86]   Geoffrey H. Cooper. IMAGEN Corporation. http://www.csonline.net/bpad-dock/tinytcp/default.htm [Accessed Feb 2000].

[RFC1945] Berners-Lee, Fielding, T. R., Irvine, H. UC, Frystyk. Request for Comments: 1945, May 1996. http://www.faqs.org/rfcs/rfc1945.html [Accessed Mars 2000].

[RCF793]   Postel, Jon., Information Sciences Institute. University of Southern California., Request For Comments:793, September 1981. http://www.faqs.org/rfcs/std/std7.html [Accessed Mars 2000].

[RCF791]   Postel, Jon., Information Sciences Institute. University of Southern California., Network Working Group, Request For Comments: 791, September 1981. http://www.faqs.org/rfcs/rfc791.html [Accessed Mars 2000].

[RFC826]   Plummer, David C., Network Working Group. Request For Comments: 826, November 1982 http://www.faqs.org/rfcs/rfc826.html [Accessed Mars 2000].

[RFC894]   Hornig, Charles., Network Working Group.,Symbolics Cambridge Research Center, Network Working Group, Request For Comments: 894, April 1984 http://www.faqs.org/rfcs/rfc894.html [Accessed Mars 2000].

[RCF792]   Postel, Jon., Information Sciences Institute. University of Southern California., Network Working Group, Request For Comments: 792, September 1981 http://www.faqs.org/rfcs/rfc792.html [Accessed Mars 2000].

[WST94]   Richard Stevens, W., TCP/IP Illustrated, Volume 1, The Protocols, USA, Addison-Wesley,. 1994

[IAR99]   IAR Systems, AT90S C Compiler, Programming guide, September 1996

[AVR99]   Atmel Corporation., AVR, 8-bit RISC Microcontrollers, Data Book, USA August 1999.

[INT98]   Intel Application Note, AP-686, Flash File System Selection Guide, 1998