BSc Computer and Network Technology

Network Connectivity for Embedded Systems

May 2003

Stuart Haslam
stuart@bozon.net

Abstract

Few people could have failed to notice two important trends over recent years; the miniaturisation of electronic devices and the continued proliferation of the Internet. It is a natural step to integrate the two, fuelling the development of small, Internet enabled devices. Whilst the oft-cited idea of an Internet Toaster may be somewhat whimsical, Internet connectivity for home appliances is rapidly becoming a reality.

This report details the techniques involved in reducing the resource requirements of a network stack implementation, thus allowing it to be embedded within a small device. In order to provide an understanding of the network requirements, relevant protocols are presented and explained. Parts of an Internet Protocol stack have been developed, and a working device produced.

Keywords: TCP/IP, embedded device, PICmicro®.

CONTENTS

1 - Introduction 1
1.1 - Project Aims 2
1.2 - Report Structure. 3

2 - Background 4
2.1 - The Layered Model. 4
2.1.1 - Physical Media. 5
2.1.2 - Link Layer 6
2.1.3 - Network Layer 7
2.1.4 - Transport Layer 8
2.1.5 - Application Layer 9
2.2 - The PICmicro® Microcontoller 10
2.3 - Existing TCP/IP Implementations 11

3 - Development Tools 13
3.1 - The C Development Environment 13
3.2 - The In-Circuit Debugger 14
3.3 - The Development Board 15
3.4 - The PICmicro® Programmer 15
3.5 - Support and testing applications 16

4 - Implementation 17
4.1 - Memory Allocation. 18
4.2 - SLIP: Serial Line Internet Protocol 19
4.3 - IP: Internet Protocol. 20
4.3.1 - IP Reception. 21
4.3.2 - IP Transmission. 22
4.4 - ICMP: Internet Control Message Protocol. 24
4.3.1 - Ping. 24
4.5 - UDP: User Datagram Protocol. 25
4.5.1 - UDP Reception. 26
4.5.2 - UDP Transmission. 26
4.5.3 - TFTP: Trivial File Transfer Protocol. 27

CONTENTS

4.6 - TCP: Transmission Control Protocol 28
4.6.1 - TCP Limitations. 31
4.6.2 - TCP Reception. 33
4.6.3 - TCP Transmission. 35
4.6.4 - HTTP: Hyper-Text Transfer Protocol. 36
4.7 - The Gateway Model. 38
4.8 - Hardware Production. 39

5 - Discussion 40
5.1 – The Complexities of TCP. 40
5.2 – Evaluation 41
5.2.1 – Summary of Features 41
5.2.2 – Compatibility. 41
5.2.3 – Performance. 41
5.2.4 – Limitations and Resource Requirements 42
5.3 - Future Work. 42

6 – Conclusion 43

References

Bibliography

DIAGRAMS

1.1 - A protocol bridging device example 2
2.1 - Layers of the TCP/IP stack. 4
2.2 - Encapsulation of a datagram 5
3.3 - SLIP encapsulation. 7
3.1 - The WIZ-C development environment 13
3.2 - In-Circuit Debugger and Development Board 14
3.3 - PICmicro® programmer hardware 15
3.4 - PICmicro® programmer software 15
3.5 - Setting of configuration parameters 15
4.1 - Protocol placement within the stack 17
4.2 - Internet Protocol header format 21
4.3 - User Datagram Protocol header format 25
4.4 - Successful TFTP transfer 27
4.5 - Failed TFTP commands 27
4.6 - The effect of multiple TCP segment data and packet loss 29
4.7 - The effect of single TCP segment data and packet loss 30
4.8 - Transmission Control Protocol header format 33
4.9 - TCP connection, transfer, and termination 34
4.10 - A static HTML document 36
4.11 - Dynamic data via HTTP 36
4.12 - Internet connection through a NAT gateway 38
4.13 - Minimal hardware support 39

Introduction

A concept recently introduced to the world of information technology is that of Pervasive Computing [16]. The term pertains to the embedding of intelligent devices in all aspects of everyday life. The global demand for access to information at any time, from anywhere, has seen network connectivity added to everyday devices such as mobile phones, set-top boxes and personal digital assistants (PDAs). To provide a fully distributed service, each of these devices requires connection to a ubiquitous network infrastructure. Such a network already exists, in the form of the Transmission Control Protocol/Internet Protocol (TCP/IP) internetwork, more commonly referred to as the Internet.

The Internet has become an integral feature of modern computing, carrying the vast majority of todays data traffic. The only network similar in size to the Internet is the circuit-switched telephone network, which could soon see a reduction in usage due to migration of voice traffic to IP networks [2]. Experts estimate over half of large companies currently utilise some form of IP-based telephony, and this figure is expected to rise rapidly [3]. The desire for functionality such as this is fuelling further development of embedded network solutions. It is estimated that by the year 2005, the number of embedded applications with the ability to connect to the Internet will be larger than the number of PCs by a factor of 100 or more [7].

As the market for network enabled devices expands manufacturers are looking for methods of allowing their product to connect to the Internet. But the design and realisation of an embedded Internet Protocol (IP) solution is no small task, and many manufacturers are not prepared to invest the money and time it would take to add this functionality. The ideal solution is an intermediate device acting as a bridge between their product, and the complicated protocols of the Internet.

1.1 - Project Aims

This project has been given the title "Network Connectivity for Embedded Systems", indicating there are two issues requiring clarification when defining project goals.

1. How is network connectivity defined?

To allow communication throughout heterogeneous networks, each device on the network must be able to speak the same language. The common language of the global Internet is defined as the Internet Protocol (IP), which must provide support for other members of the TCP/IP suite.

The generic term TCP/IP refers not only to TCP and IP, but a whole family of protocols, not all of which need be implemented in a particular network stack. The only constituent of the TCP/IP suite essential for networking a device is IP itself, though it is of little use without additional protocols to facilitate data exchange. The choice of protocol implementations is discussed in more depth in Chapter 2.

2. What is an embedded system?

An embedded system is one whose primary purpose is to control and monitor another device.

The device developed in this project is not intended to function as a single unit (though it has that ability), but rather to provide a service to another embedded device. Whether network connectivity is to be added to a product retrospectively, or designed into it from day one, a protocol bridging device can drastically reduce development time, as well as alleviating much of the resource requirements from the application device. The diagram below illustrates this relationship, although the bridging and application devices would usually be embedded within a single unit.

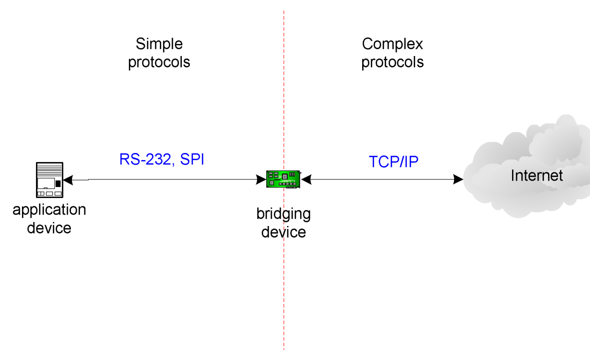


Figure 1.1 – A protocol bridging device example

1.2 - Report Structure

This report takes the reader through the development process, from the initial idea through to production and testing of a working device. The solution involves both hardware and software development, though the emphasis is strongly on software.

Chapter 2 gives details of the software and hardware technologies required to complete the project. Parts of the TCP/IP stack are presented and their relative complexities discussed. A brief background to PICmico® microcontrollers is given. Existing TCP/IP implementations are appraised.

Chapter 3 introduces various software and hardware development tools used throughout the project to produce a working example of theoretical concepts.

Chapter 4 details the software development process, paying particular attention to methods employed to reduce the resource requirements of protocol implementations. Each protocol is listed individually and its importance within the stack assessed.

Chapter 5 provides a discussion of issues raised before assessing project achievements. The compatibility, functionality and performance of the device are presented. Possible future work is listed.

Chapter 6 gives a concise view of the project.

BACKGROUND

2.1 - The Layered Model

In order to divide the tasks involved in implementing a network stack, networking protocols are usually developed in layers, with each layer carrying out a specific function.

In the early 1980s, the International Standards Organization (ISO) specified an Open Systems Interconnection (OSI) model. Its purpose was to provide a standard model for network stack implementations to follow, assisting developers to produce universally compatible software. The OSI model lists 7 layers within a network protocol stack; 1) Physical 2) Data Link 3) Network 4) Transport 5) Session 6) Presentation 7) Application. Since its inception, the OSI model has been widely dismissed as being unrealistic, though it is still widely used as a practical model for understanding and developing network protocols.

As the TCP/IP protocol suite pre-dates the OSI model it does not fit well within the seven layers. It is generally considered to be made up of four layers, with the upper layer (Application) being roughly equivalent to the upper three layers of the OSI model, and the lowest layer (Link) equivalent to the Data Link and Physical layers of the OSI model.

Layer	Function
Application	User processes
Transport	End-to-end delivery
Network	Local addressing and routing
Link	Interface with physical media

Figure 2.1 – Layers of the TCP/IP stack

Each layer provides a service to the layer above, while utilising the services of the lower layers. Every layer aims to provide an additional level of abstraction from its neighbouring layers; the link layer is not concerned about the data within a datagram, just as the application layer has no knowledge of network routing.

When transmitting data, each layer of the protocol stack adds its own header containing protocol specific information. This information is used by the corresponding protocols of the destination host (and intermediary hosts, depending on their function) to make decisions as to what to do with the packet.

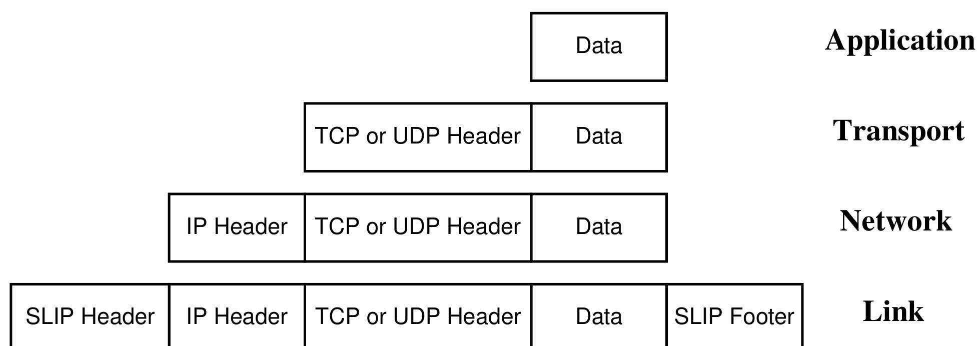


Figure 2.2 – Encapsulation of a datagram

The diagram above illustrates the transition of a packet within a network stack. On reception the packet travels from the bottom to the top, with each layer processing and stripping the corresponding headers before moving the data to the next layer.

2.1.1 – Physical Media

There are a number of possible media types for making a network connection; Ethernet, RS-232 serial line, Fiber Distributed Data Interface (FDDI), Token-ring. Each of these methods requires a link layer protocol to encapsulate data when sending, and strip the encapsulation when receiving. The link layer may also perform additional tasks such as address mapping, message queueing, and integrity checking. In the four layered model the physical media is generally thought to be part of the same layer as the protocol which supports it, and they are usually implemented very closely.

The focus of this project is primarily on the Network and Transport layers, so a simple media type and Link layer protocol are essential if fast progress is to be made. The most basic form of network connection between two computers is over a serial cable. In its simplest form this consists of only three wires; a ground connection, a wire to send, and a wire to receive [1].

The nature of a connection with a single serial cable allows a number of assumptions to be made about the properties of the network traffic it will convey, some of these assumptions can simplify other aspects of design.

A serial cable connects one host directly to another, so it can be assumed all traffic leaving one end of the connection is destined for the host at the other end - hence no form of physical addressing is required. Serial cables are usually used over relatively short distances, so the assumption is made that no data loss will occur and any data sent will arrive at its destination. For similar reasons, no integrity checking is performed on received data. Though it is possible to perform parity checking on RS-232 data, this is usually disabled, leaving integrity checking to upper layer protocols. RS-232 provides a full-duplex (asynchronous) connection, meaning both hosts on the network can send and receive at the same time and there is no need to wait for a transmit window before sending data. Finally, the RS-232 serial standard defines no data format, with bytes being sent in a continuous stream. With no definite beginning or end to a stream of data, there can be no limit to its length.

The simplicity of the RS-232 standard, and the availability of a serial port in most personal computers, makes it an obvious choice for basic network media.

2.1.2 - The Link Layer

The Link layer is the lowest layer of the Internet Protocol suite. Its purpose is to send and receive datagrams on behalf of the upper layers; to do this it must communicate directly with the physical media. The RS-232 standard alone does not fulfill the requirements of the link layer as it cannot deliver data to the IP layer in blocks (or datagrams). An intermediate protocol must be used to convert the continuous stream of data into datagrams, with a defined beginning and end. Fortunately, such a protocol has already been defined; Serial Line Internet Protocol (SLIP).

SLIP encapsulates data within a header and footer, escaping special characters within the data stream where necessary. There is no real maximum size for a SLIP packet, though it is recommended that an implementation be prepared to accept a maximum length of 1006 bytes, and should not send more than 1006 bytes in a single datagram [6]. This figure is referred to as the Maximum Transmission Unit (or MTU), and its value must be accessible to the upper layers.

The key feature of SLIP is its simplicity, most of the work required can be done on-the-fly generating very little overhead - both in terms of processor time and encapsulation.

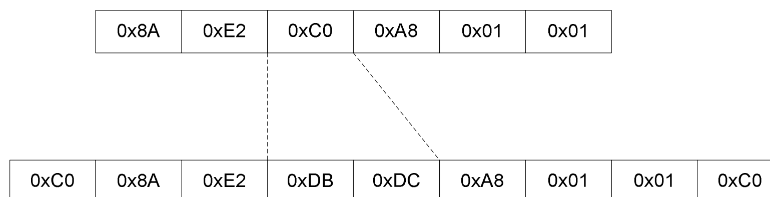


Figure 2.3 - SLIP encapsulation

Figure 2.3 shows a section of an IP header before and after encapsulation by the SLIP layer. The header and footer characters (0xC0) can be seen, as well as an escaped character within the byte stream. If an encapsulation character appears within the stream, it must be replaced by two special characters (similarly, if an escape character appears within the stream, this must also be replaced). The example deliberately includes a special character that requires escaping, but typical encapsultation overhead would be negligable.

2.1.3 - The Network Layer

The core component of the Network layer is the Internet Protocol. It is responsible for host-to-host delivery, requiring it to perform network addressing and routing. Each host on an IP network requires at least one uniquely identifiable address, and can optionally have several addresses. IP must make decisions about which packets to accept, which to route, and which to discard. The action taken is dependant on its configuration and standard rules defined in the RFC [24]. Due to the nature of the Internet end hosts are typically not required to perform routing, but the protocol implementation must have the ability to do so.

IP is also required to perform a process known as IP Fragmentation. As stated in the previous section, each Link layer protocol has a limit on how much data it can transmit within a single datagram, referred to as the MTU. If IP wishes to send a datagram larger than this limit, it must be split into smaller pieces.

When the IP layer receives a datagram to be transmitted it must check the size of the datagram (plus IP header) against the MTU supported by the Link layer of the interface the datagram is to be sent over. If the datagram exceeds this size it is split into a number of fragments, each with their own IP header, and they are transmitted separately.

Certain fields of the IP header are used to indicate if a datagram is fragmented, and the order they are to be reassembled. Intermediate routers treat each datagram as an individual packet, but the destination host must reassemble the fragments and pass the entire reconstructed datagram to the next layer.

IP itself has no method of reporting problems or generating diagnostic messages, for this purpose it employs the services of Internet Control Message Protocol (ICMP). Although ICMP messages are carried within an IP datagram, ICMP is usually considered to be an integral part of IP, therefore part of the Network layer. The messages carried by ICMP are usually an error indication or information about the state of the network, to be passed to another protocol, such as IP or TCP. ICMP messages from routers tend to be a report of better routes or network problems, while messages from end hosts are usually used for diagnostics or to report that a Transport layer port was unreachable.

2.1.4 - The Transport Layer

The purpose of the Transport layer is to provide a transparent communication service to the applications of the layer above. In the TCP/IP stack this service is provided by one of two protocols, the User Datagram Protocol (UDP [20]) or Transmission Control Protocol (TCP [29]). Although both of these protocols aim to provide the same basic service, they do so in entirely different ways.

UDP is an unreliable, connectionless service. It provides no guarantee the data will arrive at its intended destination or what state it will be in when it gets there. Although integrity checks are an optional part of the implementation in the form of checksums, this does not make it reliable. Any packet failing an integrity check is simply dropped without any notification to the Application layer, or attempt at retransmission.

Conversely, TCP is a reliable, connection-oriented service. It provides a reliable byte stream service to the Application layer by requiring caching of transmitted packets while waiting for them to be acknowledged by the recipient. If no acknowledgement is received the packet is assumed lost and retransmitted.

UDP is typically only used to provide a low latency, high throughput service where reliability is not essential. Approximatley 80% [13] of Internet traffic is sent via TCP.

The Transport layer is also responsible for linking incoming datagrams to applications, it does this by the use of port and address numbers. Each datagram arriving at this layer contains a source and destination port number, these numbers are used to identify the sending and receiving applications. These port numbers, along with the source and destination IP addresses identify each *connection*. This unique identifier (consisting of two port numbers and two IP addresses) is often referred to as a *socket*.

2.1.5 - The Application Layer

Whilst the lower layers are responsible for moving data around the network efficiently, the Application layer is where the data originates and usually also where it ends up. This covers all user interaction, such as the world wide web (HTTP), email (SMTP, POP) and file transfer (FTP, TFTP). The abstraction provided by the lower layers makes tasks such as checking email seem trivial, even though the work going on underneath is far from it.

In most operating systems, protocols below the Application layer are integral features of the operating system itself, implemented as part of the kernel, while Application layer tasks are typically performed by user processes. Whether the application acts as a server or a client, it has little or no knowledge of the networking process. It simply reads and writes to the Transport layer in much the same way as if it were a file on the local system.

One of the most common Application layer protocols is HTTP, carrying all WWW traffic. When combined with a markup language, such as HTML, HTTP is an excellent means of providing a user interface. Not only can information be easily presented to the user, but the user can transfer data back to the server with the use of HTML forms or hyperlinks. This two way communication is extremely useful in providing remote control and management facilities.

2.2 - The PICmicro® Microcontroller

A microcontroller provides all of the key components of a computer, on a single microchip. They are predominantly used in the production of low-cost high-volume devices, and can be found anywhere from a toaster to a BMW.

To many people a computer is the cream coloured box under the desk, and a processor is the thing inside the cream coloured box under the desk. But 32-bit processors such as those in personal computers (Intel Pentium 4, AMD Athlon) account for less than 10% of processors sold worldwide [9]. Over half of all processors sold are small 8-bit chips, most of which are destined for embedded devices. Despite this large share of the market, 8-bit processors account for less than 15% of the market value, because they are so cheap in comparison to the larger processors.

The falling price of electronic devices in line with their decreasing size and increasing capabilities, is acting as a catalyst for many new technologies. The PICmicro® used for this project is among the best specified in the Microchip range, yet it is available for around £3 in large quantities.

PIC18F452/252 Specification	
Maximum clock rate	40MHz
Data memory (RAM)	1536 bytes
Program Memory (ROM)	16384 words
Serial I/O capability	USART, SPI, I^2C

Clearly the resources available within this environment are many orders of magnitude smaller than those of a typical Internet server. While this limits the scope of software development, it provides enough support for countless possible applications.

2.3 - Existing TCP/IP Implementations

There are a number of existing TCP/IP implementations designed specifically with embedded devices in mind, they usually fall into one of two categories; those developed by students or enthusiats as a project, and those developed as a commercial product. The protocol stacks are aimed at target architectures from tiny 8-bit microcontrollers to 32-bit embedded processors, with data memory (RAM) requirements ranging from a few tens of bytes to several megabytes.

When developing a TCP/IP implementation for a system with very limited resources, it is often necessary to remove some functionality in order to conserve vital memory. Where these savings are made is usually dependant on the amount of memory available, and the requirements of the implementation. Less memory typically means more drastic (and potentially crippling) shortcuts are taken.

The most basic step taken to simplify a network protocol stack is to tailor it heavily to suit the intended use. By doing this, the parts of protocols not required by the intended application can be left out of the implementation. While this may seem like a sensible limitation, it can severely impair the portability and scalability of the software. These kind of basic limitations can be seen most obviously in the smallest TCP/IP implementations, such as the IPic [10]. The IPic has become well known for being able to serve web pages from a tiny microcontroller using only 256 bytes of program memory. Although there is no source code available for this project, it can only be assumed this is a highly tailored implementation. The IPic serves only static pages and it is thought to store pre-computed IP and TCP headers, including checksum fields which would require little adjustment prior to transmission.

Another common limitation is to remove the ability of the IP layer to handle fragmentation and reassembly. Storing one fragment while waiting for the rest is often not possible due to severe memory limitations. Adam Dunkels' uIP for 8- and 16-bit microcontrollers imposes this limitation [25], as does the PICmicro stack by Jeremy Bentham [26]. Most of the implementations which support IP fragmentation often require more data memory than can be offered by a small microcontroller, such as the KADAK KwikNet TCP/IP Stack requiring over 4Kb of data memory [27].

The protocol implementation which usually suffers the most within a resource limited system is TCP, as this usually makes up a large proportion of the code size. Many savings are made by simply leaving out some of the features of TCP, or severely limiting the scope of the features.

Limiting the number of simultaneous connections handled by TCP is can be a simple method of reducing data and program memory requirements. While the smaller implementations support only one connection, implementations with slighly more memory available limit the simultaneous connections to a pre-defined maximum. The Texas Instruments MP430 TCP/IP stack [21] uses much of its 2Kb of data memory to buffer an outgoing packet, the packet is then left in memory until it is acknowledged. Whilst allowing for retransmission if necessary, this method only permits one connection at a time. The Texas Intruments implementation also provides no support for IP fragmentation and does not perform any checksumming on incoming data. In other implementations, such as the Internet-on-chip™ from LiveDevices [28], a limit on simultaneous connections is only imposed by the amount of memory available.

Any commercial embedded TCP/IP implementations tend to be developed on systems with more resources available – by either using a more capable microcontroller, or external memory chips to extend the resources of a limited device. This means they can implement more features of the TCP/IP stack, without the need to take some of the drastic resource limiting measures taken by some other implementations. The more standards compliant the device is, the more marketable it becomes.

3.1 - The C Development Environment

As is the case for many network protocol implementations, all of the code for this project was written in C. While every effort was made to make the code platform independent, it was at times necessary to take advantage of many of the hardware features offered by the PICmicro® microcontroller. Additionally, standard x86 compatible compilers (such as GCC [12]) are unable to generate code targeted for the 16-bit PICmicro® platform.

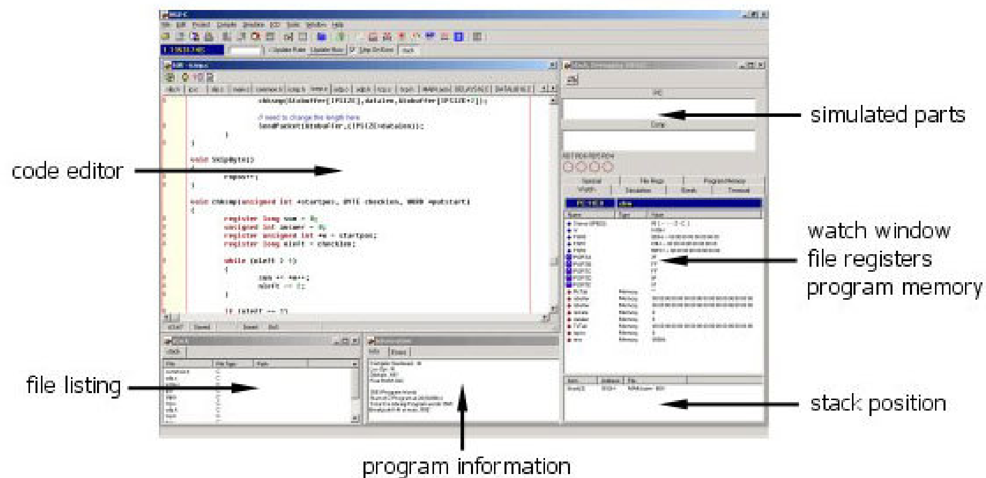


Figure 3.1 – The WIZ-C development environment

In light of this, a PICmicro® specific compiler and development environment from Forest Electronic Developments [15] was used. As well as a C compiler, the development environment provides simulation of a PICmicro® with all the usual debugging tools, such as breakpoints and watch variables. While the simulation was helpful in getting started (establishing a serial connection and developing the SLIP driver), it soon became necessary to interact within a real TCP/IP network environment.

Use of a higher level language undoubtedly allowed for faster development, although the compiler and development environment introduced some unexpected complications. Several bugs were found within the development environment and library routines, many of which stalled development considerably. To reduce time spent waiting for a fix, some of the more troublesome bugs were fixed and a patch sent to the product vendor. On several occasions it was necessary to employ non-standard procedures, often abandoning functions entirely due to their unpredictable behaviour. A good example of this is the `sprintf()` function, which currently does not work for the P18F4XX2 family of microcontrollers.

3.2 – The In-Circuit Debugger

The In-Circuit Debugger (ICD) allows for real time hardware monitoring, utilising the in-circuit debugging capability of FLASH microcontrollers. Breakpoints can be set at any instruction, the program can be single stepped, registers can be inspected and their values changed; all while the program is running on the hardware itself.

Setting a break point on a particular event call, such as an ICMP echo request, causes the ICD to halt the running of the program on the MCU itself, and update any registers or watch variables within the WIZ-C development environment. A combination of this method and the use of network analysis tools running on the machine issuing the ICMP request (or an intermediate host), allows for the quick verification of software running on the MCU in real time.

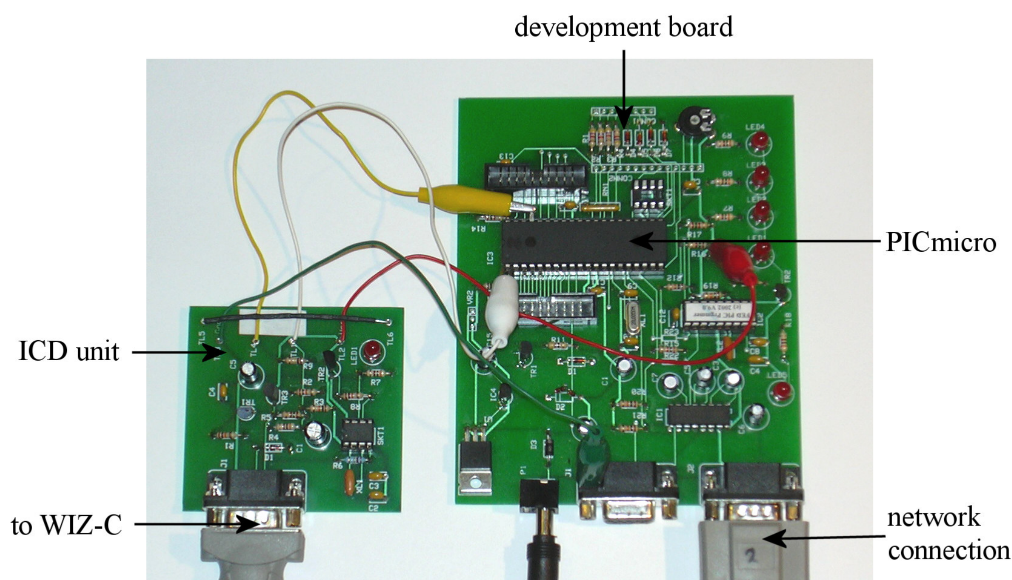


Figure 3.2 - In-Circuit Debugger and Development Board

The ICD has been an invaluable tool thoughout every stage of development, providing insight into software bugs when there seemed no logical explanation. A microcontroller can be a difficult device to work with if the only evidence of a bug in software is the fact it doesn’t work.

3.3 - The Development Board

As the impetus of this project is on software rather than hardware, a pre-built development board was used to expedite the software development process. The development board supports all 40 pin 16 and 18 series PICs (including the 18F452), and is shown in [Figure 3.2] with the ICD unit attached.

3.4 - The PICmicro® Programmer

This tool is required to load the assembled machine code onto the microcontroller. It is a hardware and software combination; the hardware connects to the PC via a standard serial port.

Figure 3.3 – PIC programmer, hardware

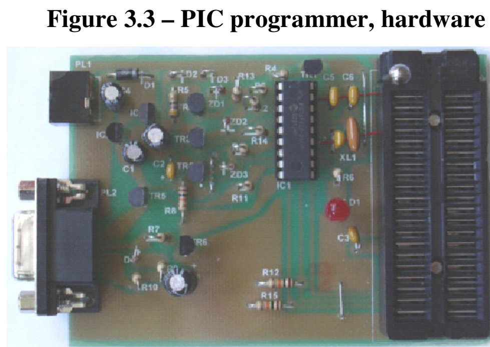
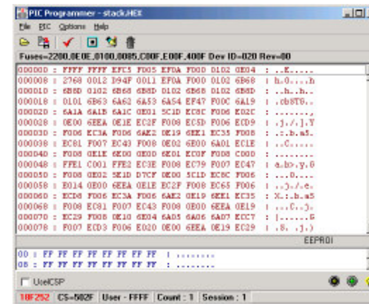


Figure 3.4 - PIC programmer, software



This tool provided its own complications. A PICmicro® contains a set of 'configuration fuses' which are used to control various parameters of the microcontroller operation, such as oscillator source and type. If the fuses are incorrectly set, the device will not run as expected. Unfortunately the programmer software does not provide any user interface for the setting of these fuses on the P18FXX2. The user must set them by calculating and inputting seven four digit hex numbers, so a simple Visual Basic program was written to calculate these values.

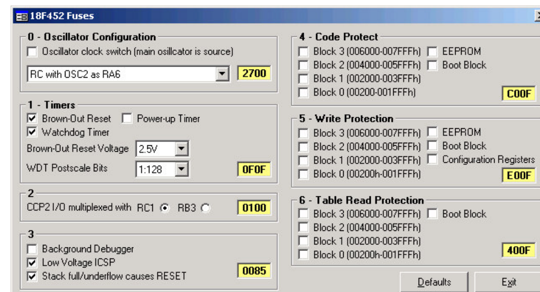


Figure 3.5 - Setting of configuration parameters

3.5 - Support and Testing Applications

Thoughout development the device was connected via the ICD to a PC running Windows and the WIZ-C software, and connected via the RS-232 interface to a FreeBSD 4.8 [11] host, enabling it to interact with a fully functional TCP/IP stack for testing and debugging purposes.

FreeBSD easily facilitates the establishment of a SLIP connection, and supports an excellent range of network analysis tools. Software development was a continuous cycle of programming and testing, the network analysis tools providing the means to review the functionality and compatibility of each protocol implementation as it was developed. It was often necessary to inspect the contents of received and transmitted packets at a bit level, however many network tools were unable to provide this functionality. In addition, the use of SLIP highlighted the inability of Ethereal (a popular network analysis tool) to handle this protocol.

Most of the network debugging was performed with the help of the ever-popular `tcpdump`, which provides detailed output as well as extremely useful functions such as checksum verification and TCP flag identification. An example output displaying an ICMP echo request and reply is shown below.

```
# tcpdump -X -vvv -n -i sl0

08:18:43.073989 192.168.1.1 > 192.168.1.2: icmp: echo request (ttl 64, id 61739, len
84)
0x0000   4500 0054 f12b 0000 4001 062a c0a8 0101        E..T.+..@..*....
0x0010   c0a8 0102 0800 3712 cc92 3400 53f8 a03e        ......7...4.S..>
0x0020   e020 0100 0809 0a0b 0c0d 0e0f 1011 1213        ................
0x0030   1415 1617 1819 1a1b 1c1d 1e1f 2021 2223        .............!"#
0x0040   2425 2627 2829 2a2b 2c2d 2e2f 3031 3233        $%&'()*+,-./0123

08:18:43.109111 192.168.1.2 > 192.168.1.1: icmp: echo reply (ttl 64, id 38408, len 84)
0x0000   4500 0054 9608 0000 4001 614d c0a8 0102        E..T....@.aM....
0x0010   c0a8 0101 0000 3f12 cc92 3400 53f8 a03e        ......?...4.S..>
0x0020   e020 0100 0809 0a0b 0c0d 0e0f 1011 1213        ................
0x0030   1415 1617 1819 1a1b 1c1d 1e1f 2021 2223        .............!"#
0x0040   2425 2627 2829 2a2b 2c2d 2e2f 3031 3233        $%&'()*+,-./0123
```

Once accustomed to the intricacies of the protocols, by studying tcpdump output such as above it is relatively easy to identify erroneous packet contents.

IMPLEMENTATION

Although by design the TCP/IP protocol stack is divided into distinct layers, this is not always the best way for it to be implemented. Having a distinct separation between layers often increases the program and data memory requirements, as the layers are unable to share a common memory space. Blurring the borders between protocol layers, while reducing resource requirements, may have a detrimental effect on the interoperability of the stack as a whole.

In terms of the TCP/IP stack, interoperability can be divided into two key areas; interoperability with other Internet hosts, and interoperability throughout the protocol layers. The division of tasks within the protocol stack will not have any adverse effect on the former, as external hosts are not concerned with the inner workings of any other host, but only with how it handles network traffic. but may cause difficulties in communicating with existing protocol implementations within the same system. However, in an embedded system such as a microcontroller, this will not be an issue.

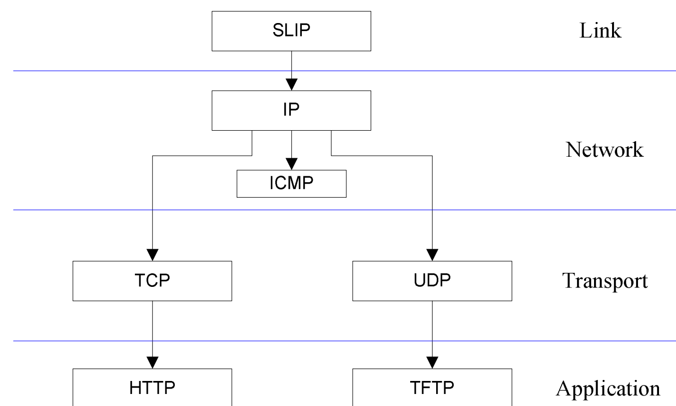


Figure 4.1 – Protocol placement within the stack

The software developed implements at least one protocol from each layer of the TCP/IP stack. Each protocol layer does its own processing before passing the rest of the data to the next protocol layer, which may be identified by a portion of data in the header, or by the destination address and port number combination.

4.1 - Memory Allocation

The scarcest resource within the microcontroller is data memory, so it must be used only where absolutely necessary. The C compiler used allows for both local and global variables. Local variables exist within a common memory space which is utilised by all functions, whereas global variables occupy the same memory space throughout the program cycle. Where data memory requirements are referred to, this is in terms of global memory unless otherwise specified.

Dedicating more data memory to one area of the software means it is not available to another area. By anticipating network traffic trends, decisions can be made about the relative importance of receive and transmit buffers.

4.1.1 - Receive Buffer

Protocol headers are essential in any received datagram for identification and processing by the upper layers. As we go further up the stack, more data is required to identify the characteristics of the packet.

For example, an HTTP request requires TCP and IP headers, as well as at least part of the application data. The IP header requires 20 bytes plus options (usually not present), the TCP header requires another 20 bytes plus options (typically between 0 and 24 bytes), and finally the application data, which is a variable length up to around 500 bytes. Storing all this data would use almost half of the global memory available within the device. With knowledge of the HTTP protocol it can be deduced that not all of the application data is essential to the operation of the device, therefore the memory allocated for it reduced.

In an HTTP/1.0 request (note that HTTP/1.1 is backwards compatible) the only essential data appears within the first few bytes of application data. By tailoring an application to suit a resource limited system, and by keeping filenames short, as little as 10 bytes of application data can be enough. So a minimum of around 80 bytes from an incoming packet must be processed. The receive buffer was allocated 128 bytes.

4.1.2 - Transmit Buffer

For reasons explained in detail when examining the requirements of the TCP implementation (Section 4.6), the transmit buffer has been allocated the majority of the available data memory.

4.2 - SLIP: Serial Line Internet Protocol

The SLIP layer works in tandem with the network interface, interacting with it via two circular buffers. When a character is received over the RS-232 serial connection an interrupt routine is triggered, and the character added to a receive buffer in data memory. The size of the buffer can be altered, depending on the available memory, processor frequency, and baud rate, but will typically remain relatively small.

The main program loop calls the `GetRxSize()` function to poll the size of the USART receive buffer, if it contains data, the SLIP driver is called upon to process the data with the `slip_recv()` function. The SLIP driver performs character replacement as defined in the SLIP RFC [6], and removes the SLIP encapsulation, before adding received data to a secondary receive buffer to be processed later by an upper layer.

The SLIP driver uses a simple state machine to keep track of the position within a packet, when an entire packet has been received a flag is set and the `ip_receive()` function called for processing.

A common memory space is used by the upper layers as a transmission buffer, to which data is added whenever necessary. The `slip_send()` function will then be passed a pointer to the start of the data, and the length of data to be sent. The buffer is then sent out, performing encapsulation and escaping special characters on-the-fly.

4.3 - IP: Internet Protocol

The primary function of the Internet Protocol is the addressing and routing of network traffic. These processes have been greatly simplified by making assumptions about the way in which the device will be connected to the network. The only currently supported Link layer protocol is SLIP, so many of these simplifications are in line with the support provided by the lower layer protocol and hardware beneath it.

The first limitation of the IP layer is that it can only handle a single network interface, therefore cannot perform any routing. This is partly because the hardware in its current state can only provide support for a single interface, support could be added at a later date with a few extra components. This would also require more complicated and memory intensive link layer software, with each interface requiring its own buffer areas.

Routing at the IP layer would not have required a great deal of program or data memory, but the hardware and lower layer support prohibited this functionality. This is not seen as a great disability, as an embedded device would usually not be required to route traffic within its role as an end host.

Another responsibility of the IP layer is the fragmenting and reassembly of datagrams. Fragment reassembly is a difficult task to perform on a system with limited memory, as it requires the fragments to be stored in memory until all the fragments arrive. Given that the reason datagrams become fragmented is because they are large (or have traversed a particularly slow network connection), this leads to high data memory requirements. This is complicated by the “best effort” nature of IP, and the fact any fragment may arrive late, out of order, or not at all.

On a target device with only 1536 bytes of data memory [18], practical fragment reassembly is an impossible task. This does not present as big a problem as it may at first seem. IP fragmentation and reassembly is known to be a burden on routers and end hosts and every effort is made to avoid it, ensuring it is a relatively rare occurence [14]. The requirement of every Internet host to handle packets of at least 576 bytes means up to around 536 bytes of application data can be sent over TCP (plus 20 bytes for an IP header and 20 bytes for a TCP header) before any fragmentation occurs. Many common applications are tailored to suit this figure, for example, web browsers will rarely send requests longer than 536 bytes.

Finally, the IP layer provides no support for IP options either in transmission or reception. IP options are rarely used, so this lack of support imposes no undesirable limitations.

4.3.1 - IP Reception

Upon reception of an IP datagram, a number of preliminary checks are performed which allow the IP layer to make a decision whether to discard the datagram or pass it on to the next layer for further processing (be that ICMP, UDP or TCP).

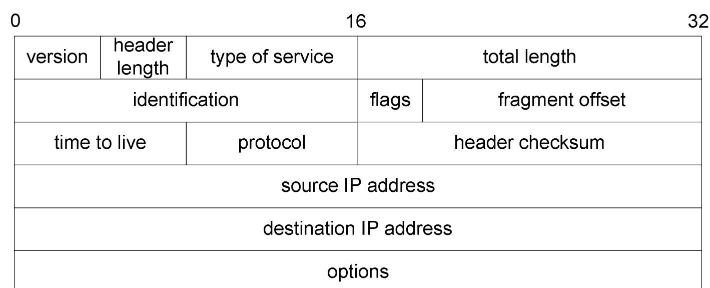


Figure 4.2 – Internet Protocol header format

Each portion of the IP header is processed as it is read from the receive buffer.

Version: this field indicates the IP version of the datagram. The device only supports the current version, version 4. The field is checked and the datagram silently discarded if the version is anything other than 4.

Header Length: the length of the IP header in 32-bit words. On reception this value is read and stored temporarily for use later in discarding any IP options.

Type of Service: traditionally used to prioritise one type of service over another. This field is rarely implemented and the device could provide no means of prioritisation, so the value is simply discarded.

Total Length: the length in bytes of the entire datagram, including the header and data. This value could be required by the upper layers, so is stored in a global variable where it can be read by any layer in the stack. In implementations with a strict boundary between protocol implementations, the length of a datagram would be retrieved with a function call to the IP

layer, but in this resource limited environment it is advantageous to give the upper layers direct access to information from other layers. Careful attention must be paid to the amount of information shared if maximum memory savings are to be made.

Identification, flags and fragment offset: these fields are used to indicate a fragmented packet. As this implementation of the IP layer does not support fragmentation, any fragmented datagrams will simply be discarded.

Time To Live: this is an 8-bit counter which is decremented each time the datagram passes through a router. It limits the amount of time a datagram can remain in transit, when the counter reaches zero, the datagram is discarded. As no routing will be done on the datagram, this value is simply discarded.

Protocol: this field is used to indicate which protocol layer passed IP the datagram to be sent, and therefore which layer it should be passed to on reception. If the value is not 1 (ICMP), 6 (TCP) or 17 (UDP) the datagram is discarded [19].

Header checksum: The calculated checksum over the entire IP header. This value need not be paid particular attention, but must be present in the header when the checksum is verified. The checksum is calculated using a standard internet checksum routine. If the integrity of the IP header is intact, the calculated value should be all 1s (0xFFFF). If this is not the case, the datagram is silently discarded.

Source and destination IP addresses: as the SLIP protocol does not support addressing, it is essential the IP layer knows the address of the SLIP interface. The device's IP address is defined at compile time and remains constant throughout operation. The stored address is matched with the destination address of every incoming IP datagram, and the datagram discarded if they do not match. No support is provided for broadcast or multicast addressing.

4.3.2 - IP Transmission

Transmission of an IP datagram is considerably simpler than reception. The processing is further simplified by assuming a number of properties remain constant, and merely placing the values into a buffer in the correct order.

Version: A value of 4 is always transmitted in this field, as that is the only supported version.

Header length: The transmitted value is always 5 (20 bytes), as no IP options are sent.

Type of Service: Unsupported, a value of zero is sent.

Total Length: the total length of the IP datagram must include the length of any Transport layer header and application data. This presents the first major problem, how can the total length of data be known before it is sent? If the data were not dynamic, the value could simply be stored with the static data and sent out when required. With the need for dynamic data, although there are some 'solutions' as mentioned earlier (Section 2.3), it was decided all outgoing data *must* be buffered in data memory prior to being sent. This also simplifies (and indeed, makes possible) checksum calculation across all data, which is required by TCP and optional in UDP.

Identification, flags and fragment offset: The Identification is a 16-bit counter, incremented each time a datagram is sent. As fragmentation is not supported, no flags are set and the fragment offset is always zero.

Time To Live: A constant value of 64 is transmitted.

Protocol: This value is derived from the layer calling the IP transmit function, so it will always indicate ICMP, TCP or UDP.

Header checksum: the standard internet checksum calculated over the 20 byte IP header. The header is stored in data memory with a zero value in the checksum field, the memory is read and checksummed and the result placed in the correct position within the buffer.

Finally, the `slip_send()` function is called to encapsulate and send the packet out.

4.4 - ICMP: Internet Control Message Protocol

The messages conveyed by ICMP can help diagnose network problems, but are not essential to the operation of any of the other protocols within the stack implementation. Therefore, the only incoming ICMP type interpretted by the implementation is an ICMP echo request (type 8), this is replied to with an ICMP echo reply (type 0). All other incoming ICMP types will be ignored, and no other ICMP messages are generated.

4.4.1 - IMCP Ping

An ICMP echo request and reply are more commonly known as a ping (the name of the program used to issue the request). It is used to test whether a host is reachable and the time it takes to do so. The request consists of an ICMP header and variable length data. The header contains the type and code of the ICMP message, a checksum covering the entire ICMP message (header and data), an identifier, and a sequence number. The host issuing an ICMP echo request is usually referred to as the client, and the host responding to the request is the server. As is the case with most other protocols, the device will only act as a server.

When an echo request is received, it must be replied to with an echo reply. The reply must have an identifier, sequence number and payload identical to that of the request. The identifier is used by the client machine to return the ICMP reply to the process issuing the request. the sequence number is incremented by one with each request and used to keep track of missing datagrams.

The only requirement of the ICMP layer in responding to an echo request is to change the message type to an echo reply, recalculate the checksum, and pass the datagram back to the IP layer for addressing and transmission. Thus, the ICMP implementation is small and consists of only 25 lines of code.

4.5 - UDP: User Datagram Protocol

The User Datagram Protocol is the simplest of all protocols within the stack, the UDP RFC [20] is a total of two pages.

source port number	destination port number
UDP length	UDP checksum

Figure 4.3 – UDP header format

The UDP header comprises of only 4 fields; the source and destination port numbers, which are used in conjunction with the IP addresses (values supplied by the IP layer) to form a *socket* to link and incoming packet to a specific application. The third field is the UDP length, which is the length of the entire UDP data and header. This value is somewhat redundant, as the UDP length can be extrapolated from the *total length* field of the IP header, but it is required nontheless.

The final field is the UDP checksum covering the UDP header and entire data area. Calculation of the checksum is optional, unlike the IP and ICMP checksums. Although the UDP checksum is not required, its calculation and tranmission is encouraged, as the checksum within the IP header only ensures the integrity of the IP header itself with no regard for the data within. If the checksum is not calculated, a value of zero must be sent in its place. If a UDP packet is received with a calculated checksum, the integrity of the packet is required to be checked.

The UDP checksum calculation uses the same algorithm as seem for IP and ICMP, but with some additional complications. As the checksum algorithm groups 8-bit bytes into 16-bit words, the number of bytes must be an even number. If the UDP length is not even, its data must be padded with a single zero byte before the checksum is calculated. The pad byte is not transmitted with the data.

The checksum calculation must also include a 12 byte pseudo-header containing the source and destination IP addresses, the protocol field from the IP header and the UDP length again. The reason for this additional check is to ensure the packet has reached the UDP layer in error, when it was destined for another address or protocol layer.

In order to keep memory usage to a minimum, the UDP layer works closely with any application layer protocol it services.

4.5.1 - UDP Reception

When a UDP packet is received, the source port number is stored for use in transmission and the destination port number is demultiplexed with port numbers for available applications. If a match is found, a pointer is set to the start of the UDP data and the relevant application is called. If the requested destination port is not bound to any application, the packet is silently discarded. The UDP length and checksum fields are simply discarded.

4.5.2 - UDP Transmission

When an application wishes to send a UDP packet, the data to be sent must already be in data memory. The position of the data within the memory space is defined by the UDP layer before passing control over to the application. The application simply adds data to the buffer then calls the `udp_send()` function with a parameter indicating the length of data to be sent. The UDP layer calls the `ip_header()` function to add the IP headers to the global transmit buffer, the UDP header can then be appended. A pointer keeps track of the current position within the transmit buffer, this position is available to all functions.

Source Port: this is typically an ephemeral port number, the `local_port()` number is used to supply incremental port numbers when required.

Destination Port: copied directly from the received UDP packet's source port, this is essential for the client to link the UDP packet with the client application.

UDP Length: a variable length, the application data size plus the UDP header size of 8 bytes.

UDP Checksum: If UDP checksum calculation is enabled, the checksum is calculated and transmitted, otherwise a value of zero is sent. Checksum calculation can be enabled at compile time.

Finally, the `slip_send()` function is called to send the packet out over the serial interface. The Link layer is called directly from the Transport layer to avoid passing or sharing of more information than absolutley necessary.

4.5.3 - TFTP: Trivial File Transfer Protocol

TFTP presents the first of the application layer protocols, and is one of the simplest protocols to implement at this level [22]. TFTP uses UDP to send datagrams, so any data transfer is inherently unreliable. The fact a TFTP client does not need to establish a connection with the server means the server need not maintain the state of the connection. Therefore, the implementation uses no data memory at all.

When a read request (RRQ) is received, the filename is read from the request string and, if the request matches a file on the server, the data is returned to the client.

The term file is used loosely to refer to any application data the server returns after the standard TFTP header. No filesystem has been implemented on the PICmicro®, so no 'files' exist on the system. This does not pose a problem when data to be returned is limited, but would need to be tackled if the device were to send (or receive) large amounts of data.

To provide a demonstration of the working UDP layer, TFTP server and dynamic data, IP packet statistics are returned when the requested filename is 'stats'.

As the device currently has no means of storing data other than program memory and data memory, the TFTP server will not accept write requests (WRQ), responding only with an "Access Violation" error. Additionally, read request for any filename other than 'stats' will result in a "File Not Found" error.

Figure 4.4 - Successful TFTP transfer

```
stuart@bsd ~> tftp 192.168.1.2
tftp> get stats
Received 64 bytes in 0.0 seconds
tftp> quit
stuart@bsd ~> cat stats
   IP Statistics

Received:       638
Transmitted:    518
Dropped:        0

stuart@bsd ~> []
```

Figure 4.5 - Failed TFTP commands

```
stuart@bsd ~> tftp 192.168.1.2
tftp> get abc
Error code 1: File Not Found.
tftp> put stats
Error code 2: Access Violation.
tftp> quit
stuart@bsd ~>
```

4.6 - TCP: Transmission Control Protocol

TCP is by far the most difficult and resource intensive protocol implementation within the TCP/IP suite. Unfortunately, the same features that allow TCP to provide a reliable service are the ones that make it a complex protocol to implement.

Providing a connection-based service involves both sides of the connection retaining some form of state information. In most implementations memory buffers are used for this purpose, each TCP session having its own memory buffer. Whenever a TCP segment is received or transmitted, the memory buffer relating to that TCP session is updated, to maintain a current record of the properties of the session. Each buffer must contain at least the following values:

Property	size (bytes)
Connection state	1
Remote IP address	4
Local IP address	4
Remote port number	2
Local port number	2
Last ACK number	4
Sequence number	4

While this memory buffer system can be used to maintain the state of many simultaneous connections, for it to be of any performance benefit each buffer must also retain a copy of the last segment to be sent, including any data sent within it. This will be required if the segment needs to be retransmitted. TCP utilises a retransmission system to compensate for any datagram which may have been lost or corrupted on route to the destination. If a transmitted packet has not been acknowledged after a certain amount of time, the packet is assumed to be lost, and retransmitted.

In most TCP implementations data memory (RAM) is used to store previously transmitted TCP segments which are awaiting acknowledgement. As there is barely enough data memory within the microcontroller to store a single segment before transmission, storing multiple segments is out of the question, thus making retransmission impossible.

The severe memory limitations of the microcontroller again mean shortcuts must be taken, and assumptions made about the environment the TCP protocol will operate in. By assuming all application layer services will act only as a server and not a client, as is the case with the HTTP server, huge simplifications can be made to the TCP implementation. The nature of a client-server relationship is that the client initiates an action and the server simply responds. If the TCP implementation is only ever going to respond to incoming segment, then no information about the connection need be saved, it can simply be copied from the incoming to outgoing segment.

While this solution removes much of the complexity of TCP, it greatly reduces functionality and introduces serveral additional complications. The lack of any state information about a connection is a problem when it comes to the transfer of large amounts of data (greater than the MTU of the Link layer). Consider the following HTTP request and server response:

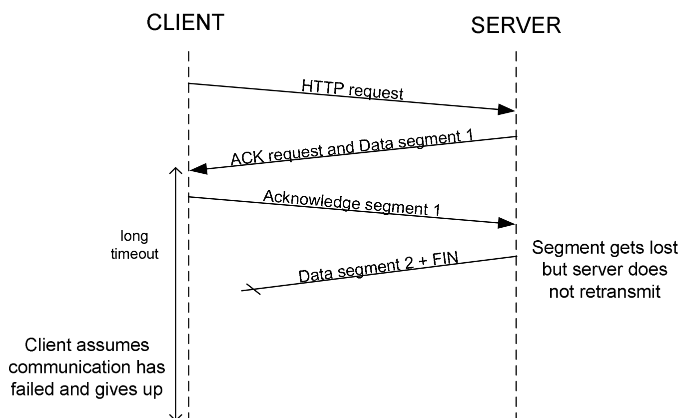


Figure 4.6 – The effect of multiple TCP segment data and packet loss

After the client sends a request and receives the acknowledgment, it then expects the server to send all the data before closing the connection. As the server now becomes the sender, it is required to handle lost packets using the timeout method explained earlier. Without implementing this in the sender, there is no way to deal with packet loss. The client simply assumes there is no data being sent, an idle TCP connection is perfectly legitimate and can stay that way for an indefinite period (or until the client's Keepalive timer expires, after 2 hours [14]).

To allow for a more reliable service without implementing retransmission on the server, any TCP segment received from the client must result in no more than one TCP segment in response. This method ensures every transmitted segment contains an acknowledgement of a segment from the client, so if it is lost in transit the client retransmits the previous segment along with any data it contained.

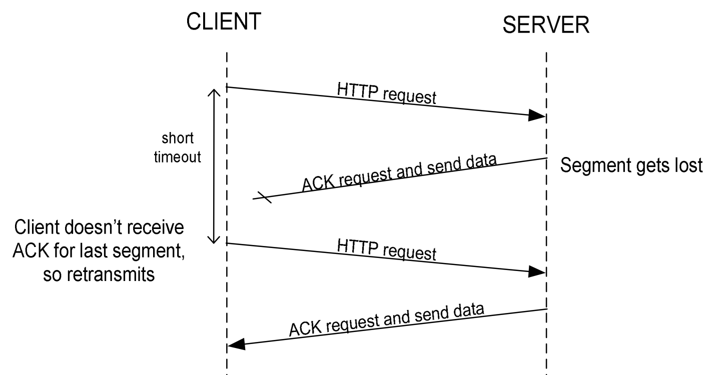


Figure 4.7 – The effect of single TCP segment data and packet loss

On reception of the retransmitted segment, the outgoing segment can be reconstructed without any knowledge of previous segments. If the purpose of the segment is to establish or terminate a connection, it is handled by the TCP layer alone. If it is a request to an application, it must be passed to the application layer to reproduce the data. The Application layers participation in retransmission is a necessary example of the blurring of borders between protocol implementations.

This method has a distinct disadvantage in that any data sent to the client must fit within a single TCP segment. It also clearly relies on the TCP implementation within the client performing retransmission, hence communication between two implementations which do not retransmit could well result in loss of data.

4.6.1 - TCP Limitations

Accepting these limitations imposed by the methods described above not only removes the need for storing state information, but allows for savings to be made in other aspects of the protocol which are now either redundant or impossible.

4.6.1.1 - Round-trip Time Measurement

TCP usually employs a complex algorithm to calculate the timeout period before assuming a segment has been lost and retransmitting it. Without any retransmission being performed, this is not required.

4.6.1.2 - Flow Control

This mechanism is used to ensure a sender does not overwhelm the receiver. For example, if the receiver has run out of buffer space to store incoming data, any segment received will simply be discarded.

The amount of buffer space available is included with every outgoing TCP segment, so each side of the connection knows how much data it can safely send without overflowing the buffer. This figure is referred to as the *window size*. A sending host must not have more unacknowledged data on the network than will fit within the receivers window size. If the sender stops receiving acknowledgements, it must stop sending segments.

As a host's receive buffer fills up, the window size is reduced, so the sender knows it must slow down transmission. When the data in the buffer is processed and removed, the window size can be increased again. Thus providing a means of ensuring neither side of the connection overwhelms the other, yet at the same time keeping throughput as high as possible.

A typical window size for modern operating systems is 64Kb. As the TCP implementation developed only ever sends one segment at a time, it would be almost impossible to overwhelm anything but the smallest embedded system. As a result, incoming window sizes are not taken into account, and a small window size is advertised.

4.6.1.3 - Congestion Avoidance

The congestion avoidance measures employed by TCP aim to prevent the overloading of routers between the two end points of a connection. It works on the assumption that the maximum rate at which TCP segments should be sent is the rate at which they are acknowledged by the other end of the connection, and that any discrepency between segment transmission rate and acknowledgement reception rate is a result of an intermediate router discarding packets.

Congestion avoidance introduces an additional transfer parameter known as the *congestion window*. The size of the congestion window is an indication of the maximum number of unacknowledged segments a sender may have. When a connection is first established a process called *slow start* is used to determine the maximum reliable congestion window size. The congestion window size is increased by one on each acknowledgement until packet loss begins to occur, at which point the sender must back off by reducing the congestion window size. A number of congestion avoidance techniques are then used to maintain a reliable and efficient congestion window size.

For much the same reason as specified for the Flow Control limitations, there is no need (or ability) to implement congestion avoidance on such a limited stack.

4.6.1.4 - TCP Options

The TCP RFC [27] only specifies one type of TCP option, the maximum segment size (MSS). This value is often specified when establishing a connection, and is an indication of the maximum amount a data a TCP segment can carry, excluding the header. To derive an MSS value the maximum transmission unit of the Link layer must be considered. In this implementation, SLIP uses an MTU of 1006. Note also that a TCP segment must be encapsulated within an IP and TCP header, so the following equation can be used:

```
MSS = MTU – (minimum size of IP header) – (minimum size of TCP header)
MSS = 1006 – 20 – 20
MSS = 966
```

Therefore, if data transfer is only possible within a single TCP segment, the maximum amount of data an application can send is 966 bytes. For a web server this would have to include all HTTP headers and the entire web page content. While this seems very restrictive, this is an embedded system displaying limited amounts of information.

4.6.2 - TCP Reception

Even in this greatly simplified implementation, a received TCP segment requires considerable processing in comparison to other protocols implemented thus far.

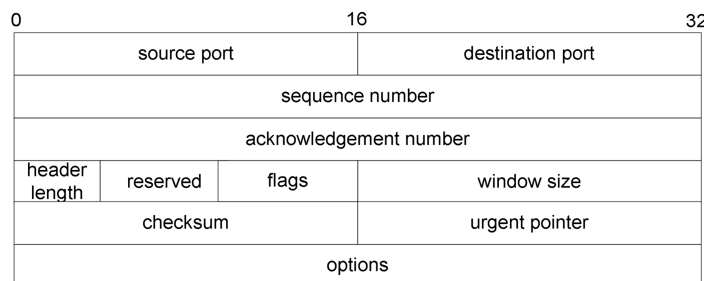


Figure 4.8 - The TCP header

As with other layers, the first action is to read the header values from the receive buffer. Most of the TCP header fields are required for processing, they are all stored except for checksum field and the urgent field, the TCP implementation provides no means of prioritising segments, so the urgent field is of no use.

The action taken by the TCP layer upon reception of a segment is dependant on six flag bits in the TCP header. Each segment can enable one or more of the flag bits to indicate the purpose of the segment. The meaning of each flag and how it is dealt with by the TCP implementation is listed below.

URG - The *urgent pointer* within the TCP header is valid. This is a method of transmitting urgent data. This functionality is not supported.

ACK - The sender is acknowledging a segment. Retransmission is not supported, so acknowledgements are ignored unless the segment has other flags set, in which case it will be processed accordingly.

FIN - The client wishes to close the connection. This is likely to be a response to a FIN sent out by the server, it is acknowledged with a value of the received sequence number plus one, for the FIN.

RST - The client wishes to reset the connection. No action is taken.

PSH - The sender wishes to pass data to the application layer. The destination port number is demultiplexed with application ports in use and the data passed to the corresponding application. The application services the request and adds data to the transmit buffer. The data is returned to the client in a single segment with an ACK flag to acknowledge the requesting segment, a PSH flag to indicate data should be passed to the application and a FIN flag to initiate closing the connection.

SYN - The client wishes to initiate a connection. If the destination port of the incoming packet is in use by an application and accepting connections the client must be sent a 32-bit Initial Sequence Number (ISN) and an acknowledgement of their segment. The value of the ISN is taken from a 32-bit global variable which is incremented when the processor is idle. The value of the acknowledgement should be equal to the received sequence number plus one, for the SYN. If the destination port is not in use by any application the connection is reset and the client sent an acknowledgement.

The ability to send acknowledgements along with application data and synchronisation segments allows for a one to one relationship between client and server, providing a reliable service by taking advantage of the clients retransmission procedures.

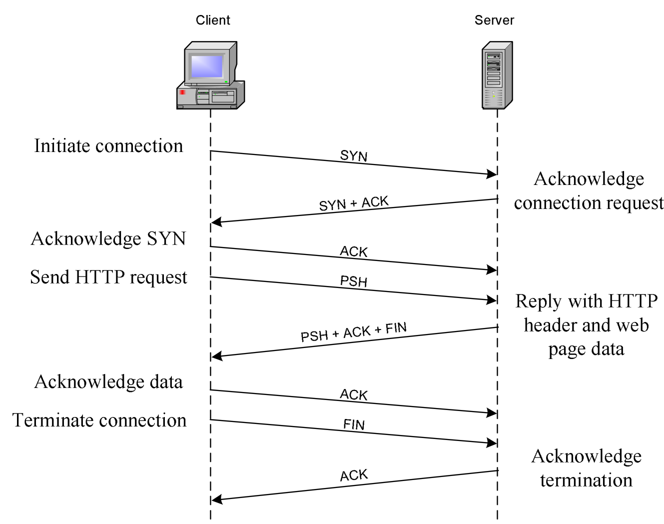


Figure 4.9 - TCP connection, transfer, and termination

The model above shows the entire sequence of segments involved in a client downloading a web page. Notice the redundant ACKs sent by the client.

4.6.3 - TCP Transmission

The most difficult obstacle to overcome when preparing a TCP segment for transmission is the checksum calculation. Unlike with UDP, the TCP checksum is not optional, it must be calculated over the TCP header and all of the data. With the header containing the checksum being sent before any data, this implies all the data must be known before the header is sent. While it is possible to store precalculated checksums, this does not allow for dynamic data. For this reason it is essential to buffer all outgoing TCP segments in data memory before they are transmitted so the memory can be read and the checksum calculated.

The Application layer is given access to the portion of data memory used for the transmit buffer, so it can simply add data to the buffer before returning control back to the TCP layer.

The TCP checksum is calculated in much the same way as with UDP, also requiring an even data length and a pseudo header containing the same values.

4.6.4 - HTTP: Hyper-Text Transfer Protocol

The HTTP server is implemented as a single C function, with a support function to produce web page content.

When the TCP layer receives a segment containing application data destined for the HTTP_PORT, it strips off any headers and calls the `www()` function. No data needs to be passed, as the www() function can simply read the HTTP request from the receive buffer. The first portion of the receive buffer is read and checked to be a GET request, with the server producing a 'Bad Request' response to any other form of request [23]. It is then a simple matter of reading the requested filename from the receive buffer and adding the corresponding data to the trasnmit buffer. The `tcp_header()`, `tcp_checksum()`, `ip_header()` and `slip_send()` functions are then called to add headers and send the packet.

As mentioned with regard to TFTP, there is no means for storing large quantities of data on the hardware device. Currently web pages are stored as static strings within the program memory, this allows for the easy insertion of dynamic values, but does not make the best use of available program memory.

Figure 4.10 - A static HTML document

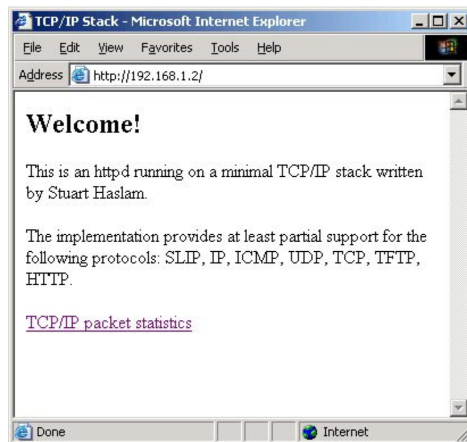


Figure 4.11 - Dynamic data via HTTP

TCP/IP Stack - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address http://192.168.1.2/1

TCP/IP Statistics

	Received	Transmitted	Dropped
Total	783	656	0
ICMP	484	484	0
UDP	74	37	0
TCP	225	135	0

index - clear stats

Done Internet

To provide a sample of static and dynamic web pages, the web server responds to three requests. A request for the index document ('/') results in a static introductory page providing a link to a page with dynamic content. The dynamic content is presented in a table to provide details of TCP/IP packet statistics, which are updated each time the page is requested.

A link from the statistics page allows the user to send a command to the web server, this is configured to reset the TCP/IP statistics returning all values in the table to zero. These values are simply used to provide an example, reading and writing to the I/O ports of the microcontroller could be done in much the same way. It would be a small step to add a thermometer to the device and return the current temperature via a web page, or display switches to control multiple LEDs.

The HTTP implementation is limited in that it does not support multiple requests per connection (HTTP/1.1), this is a common limitation for small web servers and presents no further problem. Support is only provided for an HTTP GET request, as other methods are not required.

4.7 – The Gateway Model

Accessing the Internet usually involves establishing a connection to an Internet Service Provider (ISP), all traffic destined for the Internet is then routed through the ISPs routers. This approach is often not ideal for connecting a number of embedded devices as each would require its own account, and a globally accessible IP address. Additionally, the connection process itself requires hardware (a modem) and protocols (PPP and DHCP) which would not otherwise be needed.

The solution is to provide a gateway machine on a local network, to which the embedded device connects. The gateway could itself be an embedded device, but would require substantially more resources than the client devices. During the development and testing of the protocol stack, the gateway machine was a PC running FreeBSD 4.8.

Network Address Translation (NAT) is implemented on the gateway, and facilitates communication between the private network and the Internet. This allows a node on the private network to access the Internet transparently, through the NAT gateway.

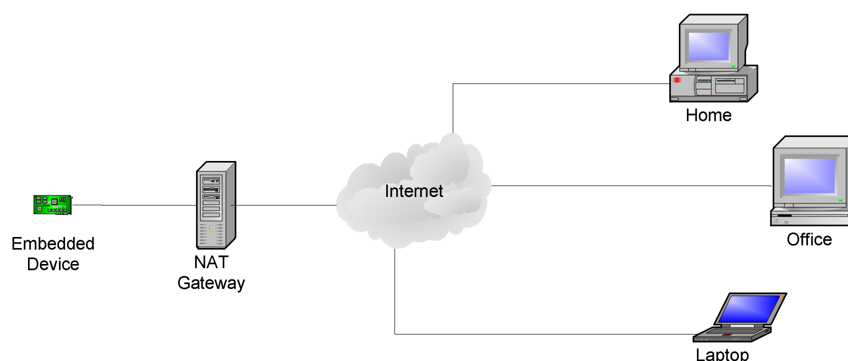


Figure 4.12 - Internet connection through a NAT gateway

This type of model is only required for systems with very limited resources. It is in effect relegating the embedded device to a lower class TCP/IP host, where all traffic to and from it is moderated by a more mature TCP/IP implementation.

Wherever possible each embedded device should have the ability to connect to any network as a so called “full citizen”, having the ability to perform any task required of it without the help of any other network host. Additional points of failure should be avoided if at all possible.

4.8 - Hardware Production

When the majority of software development was complete, it became clear many of the peripherals on the development board were not necessary to the operation of the final device. To reduce physical size and power consumption, a smaller circuit was designed and produced using a minimal number of components. Additionally, a PIC18F252 microcontroller was used in place of the PIC18F452 used during development. This MCU is functionally identical to the development version, but in a smaller package, having 28 pins rather than 40.

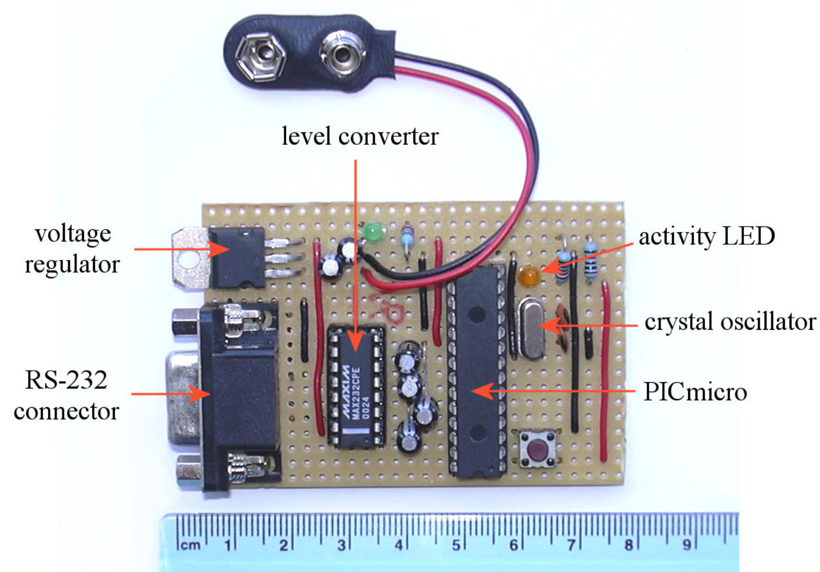


Figure 4.13 - Minimal hardware support

DISCUSSION

When setting out design goals for a network connected device it is important to remember there is more to the Internet than TCP and HTTP. If the aim of the device is to present information to a client in an easily accessible manner, then HTTP and TCP may well be the best tools for the job. But if the device is to be used for data acquisition, this may be cracking a nut with a sledge hammer.

Consider a remote weather station fitted with an embedded network device, which periodically (perhaps every minute) polls a number of sensors and sends the data to a central collation point for further analysis. Misplacing the odd sample will not make a large difference to the operation of the system, so the reliability (and complexity) of TCP is not required. Using the far simpler UDP can save not only memory and processing resources, but also Internet traffic. Establishing and tearing down a TCP connection requires a minimum of 5 packets, and a total of at least 200 bytes. While this may not sound a lot, reducing the traffic requirements of a device means the speed of the network connection can be reduced, which can reduce power requirements and extend battery life – do you want to travel 100 miles to replace a battery?

5.1 - The Complexities of TCP

Something which becomes apparent after spending some time working closely with TCP, is the maturity of the protocol. Many of the aspects of TCP that make it reliable were not present when the protocol was first developed, they evolved over decades to meet the changing requirements of modern networks. TCP as it stands today, is (or should be) an entirely indivisible protocol, every aspect of the protocol is there to meet a specific requirement and to omit any is to reduce the reliability of the protocol implementation.

5.2.1 - Summary of Features

The device currently offers the following features:

- Standard serial connection at 9600-57600 baud, no flow control.
- 5v operational voltage.
- A basic API.
- Link Layer support for SLIP.
- Network Layer support for IP and ICMP.
- Transport Layer support for UDP and TCP.
- Application Layer support for TFTP and HTTP.
- Low production cost (the prototype unit cost around £15 to produce).

5.2.2 - Compatibility

Despite the numerous compromises made within protocol implementations, the device has been proven to communicate well with every client operating system used during the evaluation. Tests have been performed using various client applications (all common web browsers and TFTP clients) running on different operating systems and TCP/IP stacks (Windows 98/XP/2000, FreeBSD 4.8, Linux kernel 2.4.18), and no combination was found to be incompatible.

It must be noted however, due to the characteristic unpredicability of the Internet, the tests performed in this limited time period cannot be deemed exhaustive. An important part of the development of a protocol implementation is undoubtedly the testing; without interoperability a protocol loses all worth.

5.2.3 - Performance

The performance of the device, in terms of data rate, was limited only by the baud rate of the serial connection to 57600bps. ICMP echo requests resulted in 100% success rate up to capacity of the connection; with a data size of 10 bytes, 100 requests per second could be serviced without any packet loss.

Informal testing shows the TCP and HTTP protocols can deal with 3 to 5 HTTP requests per second, dependant on the size of the requested document. Without more formal testing, it is difficult to say whether this is a limit imposed by the protocol stack, or the network connection.

5.2.4 - Limitations and Resource Requirements

Functionality within such a resource limited environment has been possible by implementing only the most essential parts of the TCP/IP stack. The main limitations of the TCP/IP stack are listed below:

- Ability to process only around 70 bytes of incoming application data
- No support for IP fragmentation or reassembly
- No checksums calculated for incoming TCP or UDP packets
- Ability to send only one segment worth of TCP data (about 950 bytes)

The entire protocol stack uses 250 bytes of global data memory for variable handling (storing port numbers, IP addresses, counters, etc.) and 1134 bytes for receive and transmit buffers, requiring a total of 1384 bytes of data memory. The code fits into about 5000 words of program memory, leaving more than half of the program memory available on the chip free for expansion.

5.3 - Future Work

The limited time scale and complexity of implementing a TCP/IP stack have left room for improvement within the writing and testing of software. While the protocol implementations are simplified, they fulfill (and in places surpass) the requirements stated in the design brief. Formal testing of the protocol stack in a variety of network environments would give a better idea of how well it functions under adverse conditions.

More work could be done to facilitate easy configuration and integration of additional application layer services, this would require further development of the application program interface. In addition to this, development of a basic scripting language in support of the web server would enable the production of dynamic web pages with minimal effort.

It is however doubtful whether pursuing the software development while maintaining the same hardware would bear much fruit. If development were to continue the next logical step would be to increase the available data memory thereby enabling the expansion of current protocol implementations with the aim being full compliance with protocol specifications.

CONCLUSION

This project has shown it is possible to implement parts of a TCP/IP stack within the limited resources offered by a PICmicro® microcontroller. The necessary steps taken to simplify such a complicated protocol suite have highlighted the important roles played by each aspect of the individual protocols in providing reliable and scalable network connectivity.

In order to maintain an efficient global network with millions (or billions) of interconnected nodes, there are strict rules to which every connected host must adhere. The Requirements of Internet Hosts RFC [5] implicitly specifies the responsibilities of any TCP/IP implementation wishing to connect to the Internet. Failure to comply with any one of the requirements could mean the implementation, when connected to the Internet, has an adverse affect on the reliability and efficiency of the network as a whole.

The future of the Internet would be bleak if it were allowed to be inundated with a myriad of non-compliant hosts. Any TCP implementation that does not *fully* meet the requirements of RFC1122 should not be permitted to connect to the Internet.

References

1 Bentham, Jeremy, 2002. TCP/IP Lean, Web Servers for Embedded Systems.

2 James F. Kurose and Keith W. Ross, 2003. Computer Networking, A Top-Down Approach Featuring the Internet.

3 Lange, Larry, 2002. VoIP Buzz Gets Louder.
URL: www.techweb.com/tech/network/20020726_networking

4 Wikipedia, The Free Encyclopedia. Embedded System, 2003.
URL: www.wikipedia.org/wiki/Embedded_system

5 Braden, Robert, 1989. Requirements for Internet Hosts – Communication Layers, RFC 1122.

6 Romkey, J., 1988. A Nonstandard For The Tranmission Of IP Datagrams Over Serial Lines: SLIP, RFC 1055.

7 Richey, Rodger and Humberd, Steve, 2000.Application Note AN731, Embedding PICmicro Microcontrollers in the Internet, Microchip Technology Inc.

8 International Organization for Standardization

9 World Semiconductor Trade Statistics, 2002.
URL: www.wsts.org

10 Shrikumar, H., 1999. IPic – A Match Head Sized Web Server.
URL: http://www-ccs.cs.umass.edu/~shri/iPic.html

11 The FreeBSD Project.
URL: www.freebsd.org

12 The GCC Team. GCC compiler documentation.
URL: http://gcc.gnu.org/

13 St Sauver. Joe, 15/06/2000. OWEN/NERO Bandwidth Audit.
URL: http://www-vms.uoregon.edu/~joe/bw2/owen/index.html

14 Stevens, W. Richard, 1998. TCP/IP Illustrated, Volume 1, The Protocols. Addison-Wesley Professional Computing Series

15 Abbott, Robin, 2003. WIZ-C, Visual Development for the PICmicro® MCU. Forest Electronic Developments.
URL: www.fored.co.uk

16 IBM. Pervasive Computing: IBM Wireless, Voice and Mobile Software Products.
URL: www.ibm.com/pvc/

17 Kate Marriot, 2003. Mobile Data Association, UK WAP Usage Figures.
URL: www.mda-mobiledata.org

18 Microchip Technology Inc., 2002. PICmicro® PIC18FXX2 product datasheet. Available from www.microchip.com

19 Postel, J., 1989, "Assigned Numbers", RFC 790, USC Information Sciences Institute.

20 Postel, J, 1980, "User Datagram Protocol", RFC 768, USC Information Sciences Institute.

21 Dannenberg, Andreas, November 2001. MSP430 Internet Connectivity. Texas Instruments Application Report SLAA137.

22 Sollins, Karen, July 1992. "The TFTP Protocol (Revision 2)", RFC 1350. Massachusetts Institute of Technology.

23 Berners-Lee, T., 1996. Hypertext Transfer Protocol – HTTP/1.0, RFC 1945.

24 Postel, J., Internet Protocol, 1981, RFC 791, USC Information Sciences Institute.

25 Dunkels, Adam, 2002. uIP - A Free Small TCP/IP Implementation for 8- and 16-bit Microcontrollers.
URL: www.dunkels.com/adam/

26 Bentham, Jeremy, 2002. PICmicro® TCP/IP Stack
URL: www.iosoft.co.uk

27 KADAK Products Ltd., 2002. KADAK KwikNet TCP/IP Stack.
URL: www.kadak.com

28 LiveDevices Inc, 2003. Internet-on-Chip TCP/IP connectivity solution.
URL: www.internetonchip.com

29 Postel, J, 1981, "Transmission Control Protocol – DARPA Internet Protocol Specification", RFC 793, DARPA.

Bibliography

1 Stevens, W. Richard, 1998. TCP/IP Illustrated, Volume 1, The Protocols. Addison-Wesley Professional Computing Series

2 Bentham, Jeremy, 2002. TCP/IP Lean, Web Servers for Embedded Systems.

3 James F. Kurose and Keith W. Ross, 2003. Computer Networking, A Top-Down Approach Featuring the Internet.

4 Kernighan, B. and Ritchie, D. The C Programming Language.

12 Richey, Rodger and Humberd, Steve, 2000.Application Note AN731, Embedding PICmicro Microcontrollers in the Internet, Microchip Technology Inc.

13 Abbott, Robin, December 2003. Learn to use C with FED. Forest Electronic Developments.
URL: www.fored.co.uk

7 Microchip Technology Inc., 2002. PICmicro® PIC18FXX2 product datasheet. Available from www.microchip.com

8 McRae, Andrew. C++ in an Embedded Environment,

9 Chuanxong, Guo and Shaoren, Zheng. Analysis and Evaluation of the TCP/IP Protocol Stack of Linux. Institute of Communications Engineering, China.

10 Dunkels, Adam, February 2001. Minimal TCP/IP implementation with proxy support. Swedish Institute of Computer Science.

11 Dunkels, Adam, Full TCP/IP for 8-bit architectures. Swedish Institute of Computer Science.

12 loyd, S. and Fall, K. 1999. Promoting the Use of End-to-End Congestion Control in the Internet.

13 Snell, Rodney. Web-Based Device Monitoring and Control. Intelligent Instrumentation Inc.

14 Thomas, Tracy. Internet-Accessible Home Appliances – How to Make eToast. US Software Corporation.

15 Herbert, Thomas. An Introduction to TCP/IP for Embedded Engineers, Session 404. Wind River Systems.

16 Terai, Tatuhiko, February 2003. Dynamic Resource Management Scheme for TCP Connections at Internet Servers. Osaka University.

17 Yates, Michael. TCP/IP over Ethernet for Hunter Watertech '386 Compact' Remote Telemetry Unit. The University of Newcastle, Australia.

18 Braden, Robert, 1989. Requirements for Internet Hosts – Communication Layers, RFC 1122.

19 Romkey, J., 1988. A Nonstandard For The Tranmission Of IP Datagrams Over Serial Lines: SLIP, RFC 1055.

20 Postel, J, 1981, "Transmission Control Protocol – DARPA Internet Protocol Specification", RFC 793, DARPA.

21 Berners-Lee, T., 1996. Hypertext Transfer Protocol – HTTP/1.0, RFC 1945.

22 Postel, J., Internet Protocol, 1981, RFC 791, USC Information Sciences Institute.

23 Postel, J., 1989, "Assigned Numbers", RFC 790, USC Information Sciences Institute.

24 Postel, J, 1980, "User Datagram Protocol", RFC 768, USC Information Sciences Institute.

25 Sollins, Karen, July 1992. "The TFTP Protocol (Revision 2)", RFC 1350. Massachusetts Institute of Technology.