Remote Control of Devices using an 8-bit Embedded XML & Dynamic Web-server in a SmartHouse Environment

A thesis presented in partial fulfilment of the requirements
for the degree of

Master of Engineering

in
Computer Systems Engineering

at Massey University, Albany, New Zealand

Albert Leon Kok

2006

TABLE OF CONTENTS

Acknowledgements

I would like to acknowledge Dr. Tom Moir for his guidance, support and supervision. Mr. Samir Bishay for his assistance in the hardware labs; Mr. Rafik Gadalla, Ms. Vaitheki Yoganathan and Ms. Grettle Lomiwes for their involvement and collaboration in the "SmartHouse" project currently under development at Massey University. I would also like to acknowledge Dr. Nilesh Rajbharti at Microchip for his work developing a TCP/IP stack for use on their PIC family of micro-controllers. To my parents, for everything else.

Thank you.

Abstract

This paper focuses on an Embedded System known as "TCP/IC" and its role in the "house of the future" – the SmartHouse.

Overall, the aim of the TCP/IC was to design a device which could interact with a user (or AI control system) and allow for the control various attached peripherals remotely.

Although such a device could well be used as a standalone device to aid in home-automation, this paper focuses on its use in a SmartHouse environment – one where a number of these devices are networked and controlled by a central AI.

The different technologies and protocols involved in the implementation of the TCP/IC, along with its two primary interfaces, namely HTML (used for user interaction) and XML (used for machine interaction) are also discussed.

The reader will also be introduced to Embedded Systems and the various design principles involved in the creation of quality Embedded Systems. Core-concepts of home-automation and its logical extension, the SmartHouse are also covered in detail.

Various additional interfaces (e.g. Web, XML, custom-formatted text) are also discussed and compared, as are the result of my work and some ideas for future implementations.

List of Tables & Figures

Chapter 4

Chapter 5

Chapter 6

List of Abbreviations

AI	Artificial Intelligence
ALU	Arithmetic and Logic Unit
ARP	Address Resolution Protocol
AUSART	Addressable Universal Asynchronous Receiver Transmitter
BIN	Binary (access file)
CCP	Capture/Compare/PWM
CGI	Common Gateway Interface
CMOS	Complementary Metal-Oxide Semiconductor
CPU	Central Processing Unit (including ALU)
DIP	Dual Inline Package
DOS	Disk Operating System
EEPROM	Electrically Erasable Programmable Read Only Memory
EPROM	Erasable Programmable Read Only Memory
FAT	File Allocation Table (referring to a File System)
FIFO	First-in, First-out (Abstract data type – e.g. a queue)
FSM	Finite State Machine
FTP	File Transfer Protocol
GSM	Global System for Mobile communication
GUI	Graphical User Interface
HTML	Hyper-Text Mark-up Language
HTTP	Hypertext Transfer Protocol
I/O	Input / Output
I^2C	2-wire Inter-Integrated Circuit
ICD	In-Circuit Debugger
ICMP	Internet Control Message Protocol
IP	Internet Protocol
IR	Infrared (communication)
IrDA	Infrared Data Association
ISA	Industry Standard Architecture (AT bus architecture)
ISO	International Standards Organisation
LAN	Local Area Network
LCD	Liquid Crystal Display
LED	Light Emitting Diode
LIFO	Last-in, First-out (Abstract data type – e.g. a stack)

LLC	Logical Link Control (layer)
MAC	Medium Access Controller
MIPS	Million Instructions per Second
MPFS	Microchip File System
MS	Microsoft
MSS	Maximum Segment Size (typically 576 or 1500 bytes)
MTU	Maximum Transmission Unit (typically 576 or 1500 bytes)
NIC	Network Interface Card
OS	Operating System
OSI	Open Systems Inter-connection
PC	Personal Computers (a.k.a Desktop Computer)
PCI	Peripheral Communication Interface
PVR	Personal Video Recorder
PWM	Pulse-Wave Modulation
RAM	Random Access Memory
RFC	Request For Comments (Internet Standards)
ROM	Read Only Memory
SBC	Single-Board Computer
SGML	Standardised General Mark-up Language
SLIP	Serial Line Internet Protocol
SMS	Short Message Service (relating to GSM/ Mobile Networks)
SOC	System-on-a-Chip
SPI	Serial Peripheral Interface
SSI	Server Side Includes
TCP	Transmission Control Protocol
TUI	Text User Interface
TV	Television
UDP	User Datagram Protocol
VCR	Video Cassette Recorder
W3C	World Wide Web Consortium (Internet Standards Committee)
WAN	Wide Area Network
WIFI	Wireless Fidelity (refers to RFC 802.11)
XML	eXtendable Mark-up Language

Chapter Summaries

Chapter 1 – Embedded Systems

This chapter introduces the reader to the Embedded Systems and show the significant role Embedded Systems play in our daily lives. By comparing Embedded Systems to the *Personal Computer* and detailing their design requirements, it shows that Embedded Systems have requirements that differ significantly in detail and in scope from the *Personal Computer*. It also includes examples of Embedded Systems.

Chapter 2 – Home Automation & The SmartHouse

This chapter introduces the concept of home-automation and the ideas it aims to incorporate. The social impact that home-automation has on society is also been briefly discussed.

The idea of a SmartHouse, a house where automation technology has been integrated with various AI techniques to help make life easier, safer and more enjoyable for its occupants, is introduced. The chapter concludes by introducing the SmartHouse project currently under development at Massey University along with the role the device outlined in this paper, the TCP/IC, plays in this project.

Chapter 3 – The TCP/IC

This chapter introduces the TCP/IC, an *Embedded System* used to remotely switch and monitor, in real-time, connected devices. It outlines the various design options available, ranging from serial and parallel based systems to Ethernet based systems.

It goes on to describe the various options made available by using an Ethernet based system and the advantages, and problems encountered when using this approach.

It also briefly introduces the concept of rapid prototyping and further details the rapid-prototyping board used in the construction of the TCP/IC prototype.

Chapter 4 – Protocols & Standards

This chapter formally introduces protocols and explains how protocols are used in network communication.

The OSI model for a protocol stack is then introduced along with each of the involved protocols, namely ARP, IP, ICMP, TCP, UDP and DHCP were implemented on the TCP/IC.

Chapter 5 – Interface Options and Implementation

This chapter introduces the various interface options available when dealing with remote devices along with the advantages each offer, focusing on the two interface options chosen for the TCP/IC, namely HTML and XML.

It describes what each interface offers to Embedded Systems and details how they were implemented on the TCP/IC.

Alternative interfaces are also discussed and compared with the two interfaces used by the TCP/IC. The chapter concludes by showing how although other, seemingly more efficient interfaces are available, in practice they may not be.

Chapter 6 – Results & Conclusion

This chapter discusses the results obtained when the TCP/IC along with its interfaces were developed and tested.

It goes on to discus ideas for possible future work, and concludes by summarising the ideas and their implications that are outlined in this paper.

1 CHAPTER 1 – EMBEDDED SYSTEMS

1.1 Introduction to Embedded Systems

The term *Embedded System* refers to a computer that is tightly integrated within a larger system. Microwave ovens, TV's, cars, elevators and aircraft are all examples of Embedded Systems, as they are all controlled by computers which don't necessarily have a screen, keyboard and hard-disk. Designers of Embedded Systems usually have to work within tight constraints of size, weight, power consumption, vibration, humidity, electrical, and above all else, cost and reliability.

For example:

> An *Embedded System* controls your car without your knowledge - an entire management system takes the input signal from the accelerator and provides output that controls the engine.

The key component of any modern *Embedded System* is undoubtedly its micro-controller. Namely, a CPU that incorporates an entire array of hardware into a tiny package to produce a device ready for connection to the outside world without relying heavily on external logic. Micro-controllers are often seen as the "ultimate" in computer miniaturisation because they incorporate all the necessary I/O interfaces of a complete computer system on a single piece of silicon – or "chip".

Regardless of the user interface, most Embedded Systems have an external interface for status monitoring and system diagnosis. Traditionally this has been in the form of a serial terminal, but the industry is starting to see the advantages of remote control and diagnosis offered by incorporating a network-aware interface.

1.2 A Brief History of Micro-controllers and Embedded Systems

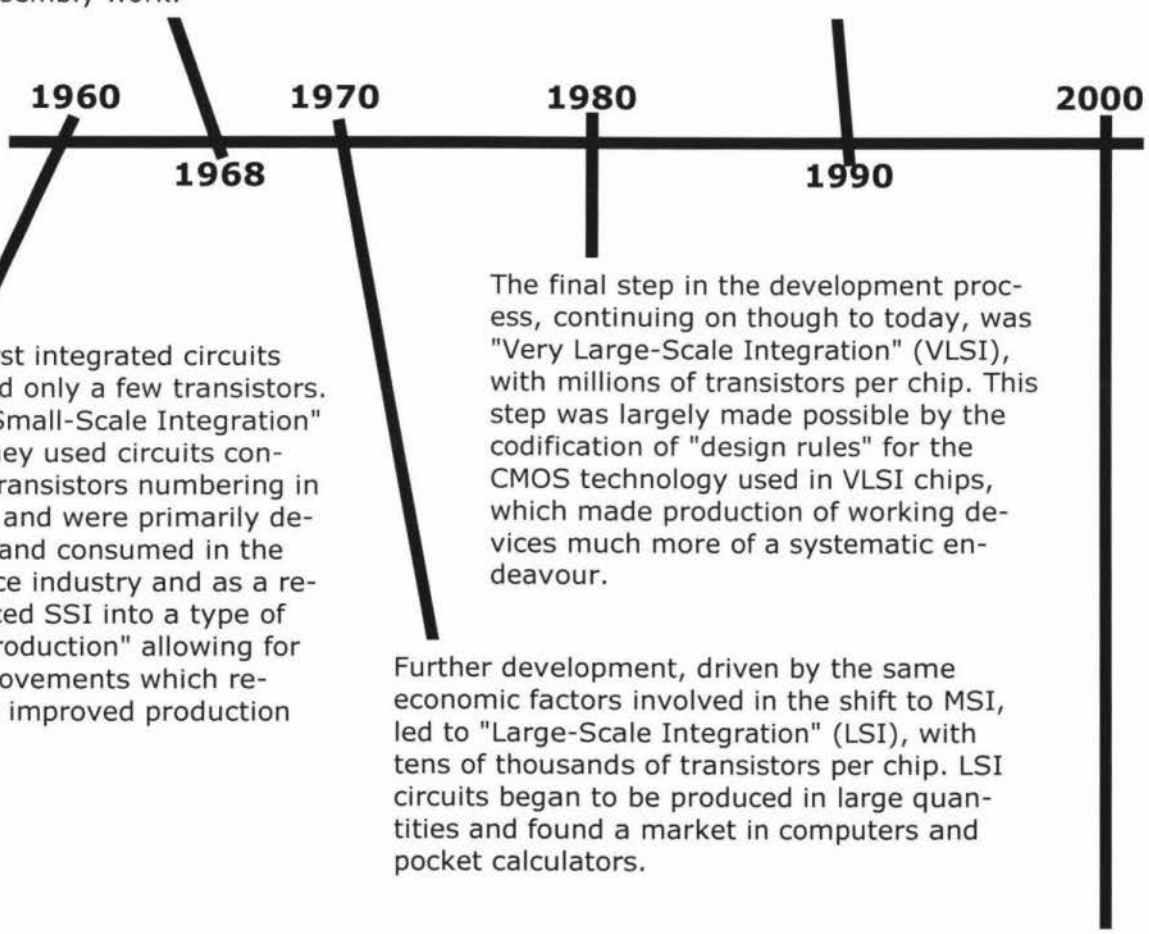
Early micro-controller applications tended to emphasise I/O and not computation [1]. A typical micro-controller would be used to interface to several, mostly analogue, I/O devices and to perform basic sensing operations such as environmental measurements and controller input.

Relatively simple transformations would be made on the data by the micro-controller, with most of the work done by the external analog devices themselves. The job of the early micro-controller was simply to control and sequence external devices to provide a higher level of overall functionality.

Today, however, much more of the system's work is done in the digital domain and micro-controllers, particularly, high-performance 8/16/32-bit micro-processors cheap enough to use in consumer products, have since allowed for an exponential growth in the market for Embedded Systems. Modem Embedded Systems may even include megabytes of code and run at relatively high (processor) speeds to meet tight performance deadlines.

In today's society, Embedded Systems are everywhere. In 2002, the *"New York Times"* estimated that the average American comes into contact with 100 micro-processors every day! [2]

1.3 Introductory Timeline of Micro-controllers



These first integrated circuits contained only a few transistors. Called "Small-Scale Integration" (SSI), they used circuits containing transistors numbering in the tens and were primarily developed and consumed in the Aerospace industry and as a result, forced SSI into a type of "mass production" allowing for the improvements which resulted in improved production costs.

This next step in the development of integrated circuits introduced devices which contained hundreds of transistors on each chip, called "Medium-Scale Integration" (MSI). They were economically attractive because, while they cost little more to produce than SSI devices, they allowed more complex systems to be produced using smaller circuit boards and less assembly work.

Further development, driven by the same economic factors involved in the shift to MSI, led to "Large-Scale Integration" (LSI), with tens of thousands of transistors per chip. LSI circuits began to be produced in large quantities and found a market in computers and pocket calculators.

The final step in the development process, continuing on though to today, was "Very Large-Scale Integration" (VLSI), with millions of transistors per chip. This step was largely made possible by the codification of "design rules" for the CMOS technology used in VLSI chips, which made production of working devices much more of a systematic endeavour.

The term ULSI that stands for "Ultra-Large Scale Integration" was proposed for chips of complexity more than 1 million of transistors. There is, however, no qualitative leap between VLSI and ULSI.

Known as System-on-Chip (SOC) design, components traditionally manufactured as separate chips to be wired together on a printed circuit board are designed to occupy a single chip that contains memory, microprocessor(s), peripheral interfaces, Input/Output logic control, data converters, and other components, together composing the whole electronic system.

1.4 Characteristics of Embedded Systems

The *Embedded System* design process is generally regarded as unique owing to the fact that it is a hardware-software co-design problem [3]. This means that both the hardware of the system, along with its software, must be designed together to ensure that the overall system functions both properly and is able to achieves its reliability goals.

The hardware-software co-design of Embedded Systems is performed at several different levels of abstraction. At each stage, critical architectural decisions are made using abstract hardware and software elements. As a result, Embedded System design requires intimate knowledge of the interactions between the hardware and software components even when "off-the-shelf"-type Embedded Systems (as opposed to ASICS) are used.

For example:

> The CPU and data/code memories are designed in hardware, while the process control is done within the embedded software.

In addition to the emphasis on interaction with the external world, Embedded Systems also provide functionality specific to their applications. Instead of executing spreadsheets, word processing and engineering analysis, Embedded Systems typically execute control laws, finite state machines, and signal processing

algorithms. They must often detect and react to faults in both the computing and surrounding electromechanical systems, and must manipulate application-specific user interface devices.

1.5 Embedded Systems vs. the Personal Computer

Although the PC architecture has been adapted for *Embedded System* operation and rugged single-board computers (SBC's), together with the necessary add-on cards to process real-world signals are available from a wide variety of suppliers, Embedded Systems usually have substantially different design constraints than the common "personal / desktop computer".

The most obvious difference between Embedded Systems and PCs is the way they interact with the user. PCs tend to focus on user-system interaction and usually do this in a "rich" way using, for example, a keyboard and monitor. This allows the user to develop a somewhat "natural" way of interacting with a PC. Conversely, Embedded Systems usually focus on a set of functions, often aimed at system-to-system communication, and so only require minimal user interaction.

Generally, Embedded Systems aim to integrate the various hardware components found in a PC to increase reliability and decrease both the size and cost of the final design.

For example:

Where a PC would commonly have a CPU, internal (Level 1/2 cache) and external (RAM) memory, external storage (HDD), a typical *Embedded System* would integrate these as "peripherals", essentially creating a "computer" consisting of one IC, also known as an SoC.

This one IC would contain a CPU, RAM, ROM/FLASH memory, various I/O interfaces such as a USART and connections (pin-outs) to use as a BUS for connecting sensors or any external logic that may be required.

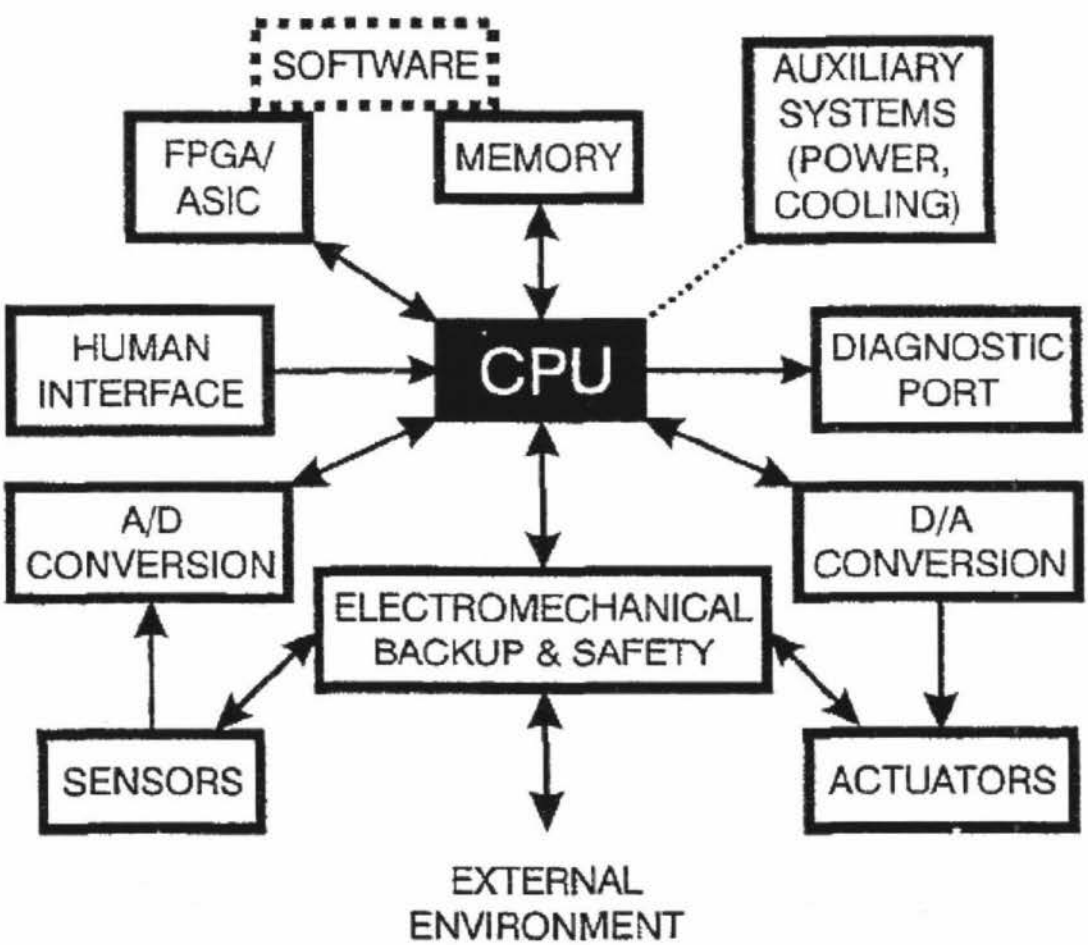


Figure 1.1: Embedded Systems Encompass many Resources

Furthermore, a PC usually creates an abstraction of the underlying hardware by running an Operating System. Programs make use of the abstracted hardware, while the operating system deals with the actual physical hardware [4]. This approach allows developers to write code (software) that is portable between different hardware platforms, but does not allow for the underlying hardware to be used directly, and thus not as efficiently as an embedded solution.

It should be noted, however, that in the last few years, operating systems have been developed specifically with Embedded Systems in mind. Examples include *eCos*, *vxWorks*, *uLinux*. These OS's differ from their PC siblings in that they all require in-depth knowledge of the hardware they are to run on and are usually only deployed to allow a real-time environment for software to run on.

1.6 Embedded System Design Requirements

As mentioned previously, Embedded Systems usually have tight constraints on the functionality they provide and the way they are to be implemented. Often they must guarantee real-time operation, conform to size and weight limits, budget power and cooling consumption, satisfy safety and reliability requirements, and meet tight cost targets [4]. Each of these constraints are covered briefly below.

1.6.1 Real-time (Reactive) Operation

Real-time systems are designed to process information in a predictable, timely manner, i.e. "the right answer late, is wrong". They must respond to I/O (or other control) correctly within strict time deadlines and the "correctness" on a computation result depends, in part, on the time at which it is delivered [5].

Most Embedded Systems have a significant reactive component, meaning that the software (or hardware) must primarily, react to external events. These events may be periodic, in which case the events are scheduled to guarantee performance or aperiodic, in which case the maximum event arrival rate (worst case performance) must be calculated in order to determine the time penalty involved. All *Mission Control* and *Signal Processing* applications depend on real-time systems to meet external stability requirements.

For example:

The ABS (Anti-Locking-Brakes) used in cars.

When the driver activates the brakes on a car, under certain conditions, the brakes apply too much pressure and the sudden deceleration cause the wheels to stop spinning, and the car begins to skid. By using an *Embedded System* to monitor the rate of deceleration, the correct amount of pressure needed to maintain a safe rate of deceleration can be calculated and applied. As time passes, the applied deceleration causes the car's momentum to decrease, thus a different amount of pressure is required to

maintain the same safe rate of deceleration. If the *Embedded System* takes too long to calculate the correct amount of deceleration to apply, the result will not be of any use.

1.6.2 Small size, low weight

Embedded Systems are usually designed to form part of a larger system, and so are usually located within some larger artefact. As a result, their form factor is usually dictated by the aesthetics and form factor of existing products, or commonly, have to fit among mechanical components. Note that in transportation and portable systems, weight may also be critical for fuel economy or human endurance.

1.6.3 Safety and reliability

Many systems have risks associated with failure. In *Mission Critical* applications such as AI craft flight control, personal injury or equipment damage could result from a failure of an *Embedded System*. Traditionally, multiple redundant systems have been implemented to ensure continued operation after an equipment failure.

Today, many Embedded Systems that could cause personal injury or equipment damage, cannot incorporate the added cost of redundancy required for traditional fault tolerance techniques. Instead, they deliver safe and reliable performance by limiting components (which could fail), developing hardware with

integrated redundancy and by thorough testing and simulation of the underlying hardware.

1.6.4 Harsh Environment

Many Embedded Systems do not operate in a controlled environment. Temperature extremes is often a major problem and additional problems can be caused for embedded computing by a need for protection from vibration, shock, lightning, power supply fluctuations, water, corrosion, fire, and general physical abuse.

For example:

> Certain Mission Critical Embedded Systems must function for a guaranteed, albeit brief, period of time even under non-survivable conditions, such as fire.

1.6.5 Cost Sensitivity

Even though Embedded Systems have very tight requirements, cost is almost always the largest issue of the requirement. The main reason for this is may be that an increase in the cost of the *Embedded System* relates to a fixed cost per unit.

For example:

> By adding an extra component to an *Embedded System*, the cost of each individual unit produced increases and so the "hardware" becomes a higher percentage of the **total** system cost.

1.7 *Examples of* Embedded Systems

An example of:	**Signal Processing**	**Mission Critical**	**Distributed**	**Small**
Computing speed	1 GFLOPS	10 - 100 MIPS	1-10 MIPS	100,000 IPS
I/O Transfer Rates	1 Gb/sec	10 Mb/sec	100 Kb/sec	1 Kb/sec
Memory Size	32 - 128 MB	16 - 32 MB	1 - 16 MB	1 KB
Units Sold	10 - 500	100 - 1000	100 - 10,000	1,000,000+
Development Cost	$20M - $100M	$10M - $50M	$1M - $10M	$100K - $1M
Lifetime	15 - 30 years	20 - 30 years	25 - 50 years	10 - 15 years
Environment	Vibration, Heat	Heat, Vibration, Lightning	Dirt, Fire	Over-voltage, Heat, Vibration
Cost Sensitivity	$1000	$100	$10	$0.05
Other Constraints	Size, weight, power	Size, weight	Size	Size, weight, power
Safety	—	Redundancy	Mechanical Safety	—
Maintenance	Frequent repairs	Aggressive fault detection/maintenance	Scheduled maintenance	"Never" breaks
Digital content	Digital except for signal I/O	~½ Digital	~½ Digital	Single digital chip; rest is analog/power
Certification authorities	Customer	Federal Government	Development team	Customer; Federal Government
Repair time goal	1-12 hours	30 minutes	4 min. - 12 hours	1-4 hours
Initial cycle time	3-5 years	4-10 years	2-4 years	0.1-4 years
Product variants	1-5	5-20	10-10,000	3-10
Engineering allocation method	Per-product budget	Per-product budget	Allocation from large pool	Demand-driven daily from small pool
Other possible examples in this category:	Radar/Sonar Video Medical imaging	Jet engines Manned spacecraft Nuclear power	High-rise elevators Trains/trams/subways Air conditioning	Automotive auxilliaries Consumer electronics "Smart" I/O

Table 1.2: Examples of Embedded Systems

1.8 Chapter Summary

This chapter has introduced the reader to the Embedded Systems and has shown the significant role Embedded Systems play in our daily lives.

By comparing Embedded Systems to the *Personal Computer*, detailing their design requirements and giving examples of common Embedded Systems, I have shown that most Embedded Systems have requirements that differ significantly in detail and in scope from the *Personal Computer*.

2 CHAPTER 2 – HOME AUTOMATION & THE SMARTHOUSE

2.1 Introduction to Home-Automation

The term home-automation can loosely be defined as something that allows remote or automatic control of anything around the home, for instance security, lighting, temperature and entertainment. A broader view would be that automation arises from the merging of computer and communications technologies.

It is expected that home-automation is to be achieved not with every household owning a "robot" [6], but by the various Embedded Systems already being incorporated in a wide range of domestic equipment - each of which communicate with the user as well as with one another. As was the case of factory-automation, it is expected that the introduction of home-automation will at first be gradual [6], starting with islands centred around connected devices, which will later merge to form a total information and control system.

The essential feature necessary for home-automation is communication - in that a means of establishing a common network inside a home is required before large-scale home-automation becomes a reality. Such a network would enable all connected devices, ranging from simple sensors, to more complete Embedded Systems, to PC's, all to be accessed for a wide range of

applications [7]. Various mediums already exist to form such a network, including infra-red, radio-wave, mains-wires and installed twisted-pair or coaxial cable.

It is widely believed that the range of possible applications for home-automation, are limited only by our understanding of our environment [8]. Systems already exist that assist in automating applications such as security, lighting, heating, cooking, washing appliances, audio and video systems, energy management, as well as a number of new applications such as health monitoring, home-publishing etc. [9]

Table 2.1: Examples of Home-Automation

Security (remote monitoring)	Intruder Alarm System
Heating and Environmental Control	Air-Conditioner
Energy Management	Under-floor Heating
Appliance Control	Oven
Entertainment	PVR and Audio Streaming
Home Shopping	Interactive catalogues / ordering
Information Services	Local Weather
Working from Home	Internet / Phone / Fax
Health Monitoring	Health check
Tele-metering	Gas, Water, Electricity

2.2 Social Impact of Home-Automation

As the field of home-automation matures, so too will the impact it has on people's everyday lives [10]. As it is a maturing technology, customers will have the choice of whether or not to buy home-automation products. This will likely have the effect that at first, the technology will spread into the homes of early-adopters, but later, into the homes of the masses, including those less favoured by society - such as the handicapped and elderly.

2.2.1 Housing and Lifestyle

A parallel between the social impact of home-automation and that experienced by the introduction of central heating can be drawn [5]. Before central heating became affordable, families tended to cluster together in a single room in the winter and watch TV. Today the trend is to use the rooms of a house for several purposes, for instance, a bedroom now often contains a television, perhaps even a gaming console.

This use of rooms for several activities can be seen to be encouraging the building of larger (room) homes [5]. Home-Automation can be expected to reinforce this trend by adding to the multi-use nature of rooms. Working from home and being able to control the home from work, may also change where we need to be at certain times, leading to a much more flexible lifestyle.

For example:

> The monitoring of personal health, which may require an online link (Internet) to a local medical centre, may change the use of some rooms. Electronic health-monitoring equipment may move into the bathroom, in the same way that it has already moved out of the lounge and into kitchens and bedrooms.

2.2.2 Disabled

Home-Automation is expected to bring with it the ability to control much of the home using a PC [9], perhaps even incorporated into an armchair. And while the ability to work and shop from home may only be a convenience for the majority of people, it would help improve the lifestyle of the disabled dramatically [10].

If the cost of such equipment is reduced by the adoption into the mass market, it would be significantly lower than the cost of specialised equipment developed for the disabled. Modifications of the equipment may be necessary, but as mass market systems will be standardised, the modification could come in the form of product "add-ons" or accessories.

2.2.3 Elderly

As mentioned, home-automation offers the ability to monitor health and signal emergency services in the case of emergencies. This one aspect alone would help elderly people to remain in their own homes for longer, rather than being institutionalised.

2.2.4 Safety

Besides the (remote) health monitoring and "panic" buttons that could be introduced to assist the elderly and disabled, it is also possible to sense for other, more common emergencies.

For example:

Detecting slips in bathrooms using infrared sensors or computer vision and respond appropriately.

Both fire and intruder alarms could connect to the *home-network*, allowing the "automated home" to automatically isolate, extinguish or alert the relevant authority.

2.3 The SmartHouse

Broadly, a SmartHouse can be defined as a house in which home-automation technology has been integrated with various AI techniques to help make life easier, safer and more enjoyable for its occupants. It should be able to track and respond to the needs and inferred desires of its occupants using computers, AI and automation technology. A SmartHouse is thought to encompass a range of technologies, including, but not limited to:

- Health and Safety
- Personal Mobility
- Quality of Life
- Smart Appliances
- Entertainment
- Security (Identification)
- Security (Household)
- Communications
- Building and Construction

2.3.1 The Massey SmartHouse Project

The *Massey University SmartHouse Project* consists of a group of people who have an interest in developing and improving home-automation technologies [11]. The group includes members of Massey University, as well as other academic institutions and industry partners located both within New Zealand and abroad.

The *Massey SmartHouse Project* was setup to showcase technology and research projects undertaken at Massey University by its students. The primary design consideration for the SmartHouse is making the life of its inhabitants easier and safer, thus allowing them to enjoy a higher quality of life.

It is also intended to showcase the potential benefits of integrating home-automation, health care and smart appliance technology into our daily lives.

The project aims to "build" / "outfit" a house with technology and appliances that help make life easier, safer and more enjoyable for the occupants.

In the past, similar projects have been developed [12], but little has been done on their integration into a cohesive unit. The main goal of the Massey SmartHouse research is to create a complete and integrated solution, one which can be customised, to give our elderly the independence, quality of life, and safety they require.

2.3.2 Contribution to the Massey SmartHouse

With *Massey's SmartHouse Project* nearing its first stage of completion, the device outlined in this paper, TCP/IC, offers the prospect of remote (LAN/WAN) switching of appliances - made easy by using the ubiquitous web browser.

Additionally, by using XML [13] (the global standard for interoperability), TCP/IC is able to tightly integrate with the SmartHouse AI, along with any other middle-ware.

2.3.3 Current Status of the Massey SmartHouse

In order to make it as easy as possible for the occupants of the smart house to interact with it and the various smart appliances, the medium of communication must be as natural as possible. The SmartHouse therefore required a speech recognition system, capable of analysing spoken language and extracting necessary instructions from it. This would allow users to interact easily with the house by simple talking naturally. In early 2006, this (distance) "Voice Recognition" system had been fully developed by Rafik Gadalla and a working prototype is being implemented.

In order to function efficiently and to avoid the duplication of much technology, the house requires a central computer and software package (or perhaps an AI) capable of using the information from the speech recognition system, feedback from smart appliances, and user location information and, based on this information,

making intelligent decisions on what to do, or relaying necessary information to individual intelligent devices throughout the house. Individual smart appliances and systems throughout the house would have on-board intelligence as required but they would be overseen by the house management system. As of 2005, a functional SmartHouse control system had been developed by Grettle Lomiwe.

For the SmartHouse to make "choices" and so interact with its occupants more effectively, a system had to be developed which would provide identification (of different occupants) as well as location information (positioning). An initial choice for this "positioning network" was Bluetooth, however, as of early 2006, although various tests have been carried out, no such system has been successfully implemented.

Some work has however, begun on a RFID occupant identification and positioning system. Currently this avenue of research is still in its infancy, however, with the recent industry adoption of RFID, it is expected that such a system will take over the role of the "Bluetooth Positioning" sub-system mentioned.

As noted, as of February 2006, the Massey SmartHouse project seems to be well underway and a building on campus is in the process of being ear-marked to create a "model-home" of the SmartHouse.

2.4 Chapter Summary

In this chapter I have introduced the concept of home-automation, the ideas it aims to incorporate and how home-automation is already taking place in our modern society. The social impact that home-automation will have on society has also been briefly discussed.

The idea of a SmartHouse, a house where automation technology has been integrated with various AI techniques to help make life easier, safer and more enjoyable for its occupants, was introduced and the chapter concluded by introducing the SmartHouse project currently under development at Massey University and the role the device outlined in this paper, the TCP/IC, plays in this project.

3 CHAPTER 3 – THE TCP/IC

3.1 Introduction and Brief History

Initially conceived to form part of the *Massey SmartHouse Project*, the idea behind creating the TCP/IC was to allow the SmartHouse AI to control various peripherals such as lamps, white-ware, appliances and entertainment devices in the home by signalling a set of "switching units".

During the design phase, both a serial-port (RS232) [14] and parallel-port (IEEE1284) [15] interface were trialled. A solution was designed that used a custom (copper-wire) network which connected all switchable devices to the SmartHouse AI server. The software on the server would then control the voltage levels of the serial / parallel port pins and the "network of devices" could be switched, although only in a basic manner.

The problem experienced with both the serial and parallel-port solutions was that a custom network had to be hard-wired into the SmartHouse – one which allowed all devices which were to be controlled, to be directly connected to the "server". This "network" can be more accurately defined as a point-to-point link that would be used exclusively for the switching of devices.

Not only would this be expensive to implement and place an unnecessary overhead on the SmartHouse "server" (ports), it

would also rely on the SmartHouse "server" – if and when the server experienced any downtime, the devices would simply switch back to their original state. This "network" would further be hindered by the length limitations imposed by serial / parallel cables and so would not be scalable [14, 15].

To break away from a serial/parallel connected approach, another solution was devised where each "room" in the SmartHouse would be outfitted with one, or more, embedded devices, which in-turn would be connected to the server. To steer clear of the custom "networks" encountered with the serial/ parallel solutions, these devices would connect to the ubiquitous Ethernet network already present in the SmartHouse.

This was a much more flexible solution, as it could easily be adapted to operate on a 801.11 wireless networks [16], as well as perhaps the Internet. An Ethernet solution, however, brought a whole new dimension to programming and interacting with the micro-controller. To allow the micro-controller to be network-aware a TCP/IP stack, small enough to run on the controllers' limited resources would have to be chosen or written, along with a multi-tasking scheme to allow the dynamic switching of all connected devices.

3.2 Design Options

Once Ethernet was chosen as the transport layer, another important decision had to be made: either use a micro-controller to interface with a commercial Ethernet interface, or use a micro-controller to interact with a NIC directly. And if that was the case, whether a commercially available or self-authored TCP/IP stack should be used.

3.2.1 Standalone Ethernet Interface

While doing research on SmartHouse's and various computer-controlled switching solutions, I came across a few commercially available Ethernet interfaces (e.g. SitePlayer [17]). Although these products vary in the features they support and levels of customisation, they all share the same design principle, namely that they handle all network communication.

Most of these "standalone" Ethernet interfaces are designed to communicate with a micro-controller by means of a serial connection – either via a USART or I2C connection. In essence, the micro-controller uses a set of "APIs" to interact with the network.

Although this approach seemed to solve the complexity problem introduced with an ISA solution (mentioned in next section), when suitable USART web-interface modules was located, it was found that their licensing cost would more than double the planned cost of each device.

For example:

SitePlayer Embedded Internet Server [17]
(http://www.siteplayer.com)

- Compact Size
- RJ-45 10BaseT Ethernet Interface
- Fully configurable RS-232 Serial Port
- Configured via a Web Browser
- Built in TCP/IP Stack
  (ARP, TCP, IP, UDP, ICMP Ping, HTTP, Daytime)

→ US$29.95 per unit (Last Checked: June 2005)
→ US$99.95 for development kit (Last Checked: June 2005)

3.2.2 Micro-controller / NIC based Approach

One alternative to using a commercial Ethernet interface is to directly bridge a "Network-Interface-Controller" (NIC) with a micro-controller – perhaps with an old ISA network card.

The idea being that the micro-controller would emulate the read/write-cycle of a 16-BIT (8 MHz) ISA bus and thereby be able to communicate with the Medium-Access-Controller (MAC) found on the ISA NIC [18].

When this was attempted, it was found that this configuration would tie up many (8) I/O pins and involve several more ICs to operate, increasing the component count of the proposed

embedded system, something that would have had a very big effect on the overall system cost.

Fortunately, another solution became apparent – instead of communicating with the MAC found on the ISA NIC by emulating the ISA bus, a micro-controller could be used to interact directly with a MAC/NIC.

If this was to be implemented, the micro-controller would have to be able to schedule tasks to the MAC as well as having enough input/output lines to facilitate connecting a MAC directly. This would still allow multiple inputs/outputs for any devices that it would be controlling (switching).

While software would have to be written to handle the MAC directly, this approach allowed for a much lower total system cost than the other two solutions outlined above. To facilitate development, a freely-licensed rapid-development board from "Microchip" [19], the developers of the PICmicro, was used to construct a prototype of such a system.

3.2.2.1 The TCP/IP Stack

By not employing a dedicated commercial Ethernet interface, network communication was no longer transparent to the systems design. That is, all protocols required for network communication (i.e. TCP stack) would have to be implemented in software, using only the limited RAM/ROM provided by the micro-controller.

Fortunately there are many commercial TCP/IP stacks available for a wide range of micro-controllers (Table 3.1). These stacks however would have to be licensed and per-unit royalties paid which would add a significant fixed cost onto the "final" price of every unit manufactured.

As an alternative, one of a few, non-commercial TCP/IP stacks that are publicly available (Table 3.1) could be chosen and then ported to the target hardware. Although this seemed like the most logical choice, a major problem was that the porting process *could* mean a complete re-write of the chosen stack.

Table 3.1: Commercial & Non-Commercial TCP/IP Stacks

Commercial TCP/IP Stacks

- CMX Systems (http://www.cmx.com/Tcpip.htm)
- InterNiche NicheStack (http://www.nohau.co.uk/products/interniche/nichestack.htm)
- Kadak KwikNet (http://www.kadak.com/tcp_ip/tcpip.htm)

Non-commercial TCP/IP Stacks

- Picnic (ISA Based) (http://members.vol.at/home.floery/electronix/picnic/)
- OpenTCP (http://www.opentcp.org/)

In the end, it was decided to attempt to author a custom TCP/IP stack, as the amount of work involved creating a new, lean, TCP/IP stack would be comparable to porting an existing open-source stack to the target hardware.

Additionally, it would allow me to familiarise and thus customise the stack for efficient use within the Massey SmartHouse.

3.3 Microchip's PIC-18F452

The PIC-18F452 micro-controller, manufactured by Microchip, is a powerful 10 MIPS (100 nanosecond instruction execution), CMOS FLASH-based 8-bit micro-controller [19].

It incorporates Microchip's powerful RISC PIC architecture and is compatible with the PIC-16C5X, PIC-12CXXX, PIC-16CXX and PIC-17CXX devices, thereby providing a migration path of software to higher levels of hardware integration.

Based on the Harvard bus architecture [20], it provides a 16-bit instruction bus as well as an 8-bit data bus. Furthermore, but employing a two-stage instruction pipeline, it allows all instructions to execute in a single cycle – except for program branches, which require only two cycles.

The PIC-18 family of micro-controllers provides a 'C' compiler optimized architecture/instruction set as well as a 'C' compiler friendly development environment. It also includes many integrated peripherals devices aimed at reducing external components to reduce cost, increase system reliability and reduce power consumption.

Features found on the PIC-18F452 are the 256 bytes of EEPROM, self-programming capability, ICD, two Capture/Compare/PWM functions and 8 channels of 10-bit Analog-to-Digital converter.

Refer to the datasheet provided as an appendix for further detail on the PIC-18 family of micro-controllers.

3.3.1 Block Diagram of the PIC-18F452

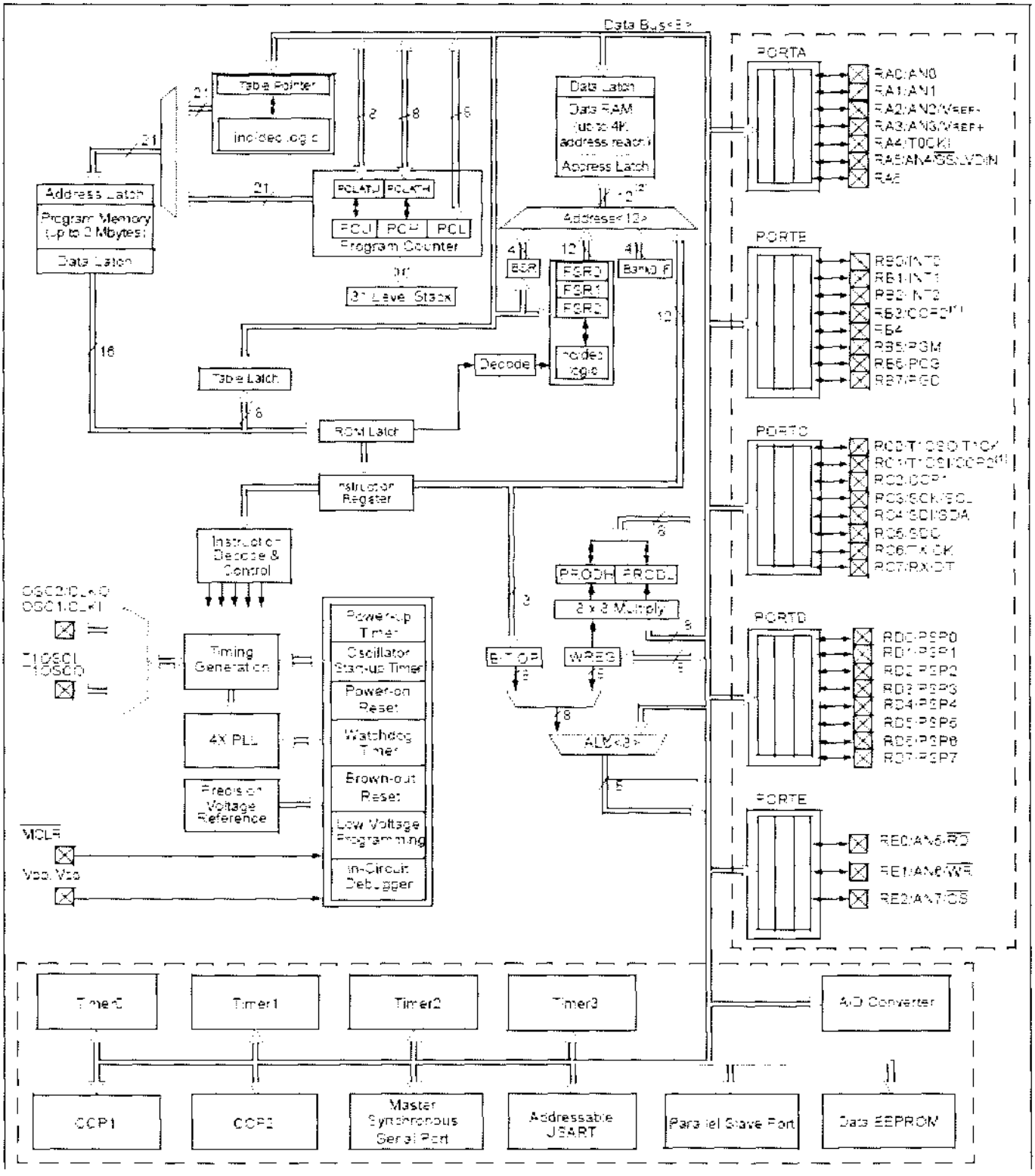


Figure 3.2 Block Diagram of the PIC-18F452

3.3.2 Overview of the PICDEM.net Demonstration Board

The PICDEM.net Demonstration Board has been created to allow developers to examine the possibilities of embedded connectivity solutions for Ethernet networks and the Internet.

Microchip [19]

The PICDEM.net demonstration board is used to experiment with Microchip's PIC family of micro-controllers in various TCP/IP solutions and supports any 40-pin DIP device that conforms to the standard pin out used by the PIC-16/18 device family (e.g. PIC-16F877).

Supporting both an Ethernet and an RS-232 interface natively, the rapid-prototyping board is also equipped with a 6-pin modular connector that interfaces directly with the Microchip In-Circuit Debugger (ICD). This feature allows me to easily debug, modify and re-program the onboard FLASH-based PIC micro-controller.

A bread-boarding area is also made available and several status indicators and user interface devices are provided, including a 16 x 2 LCD.

3.3.2.1 PICDEM.net Demonstration Board Components

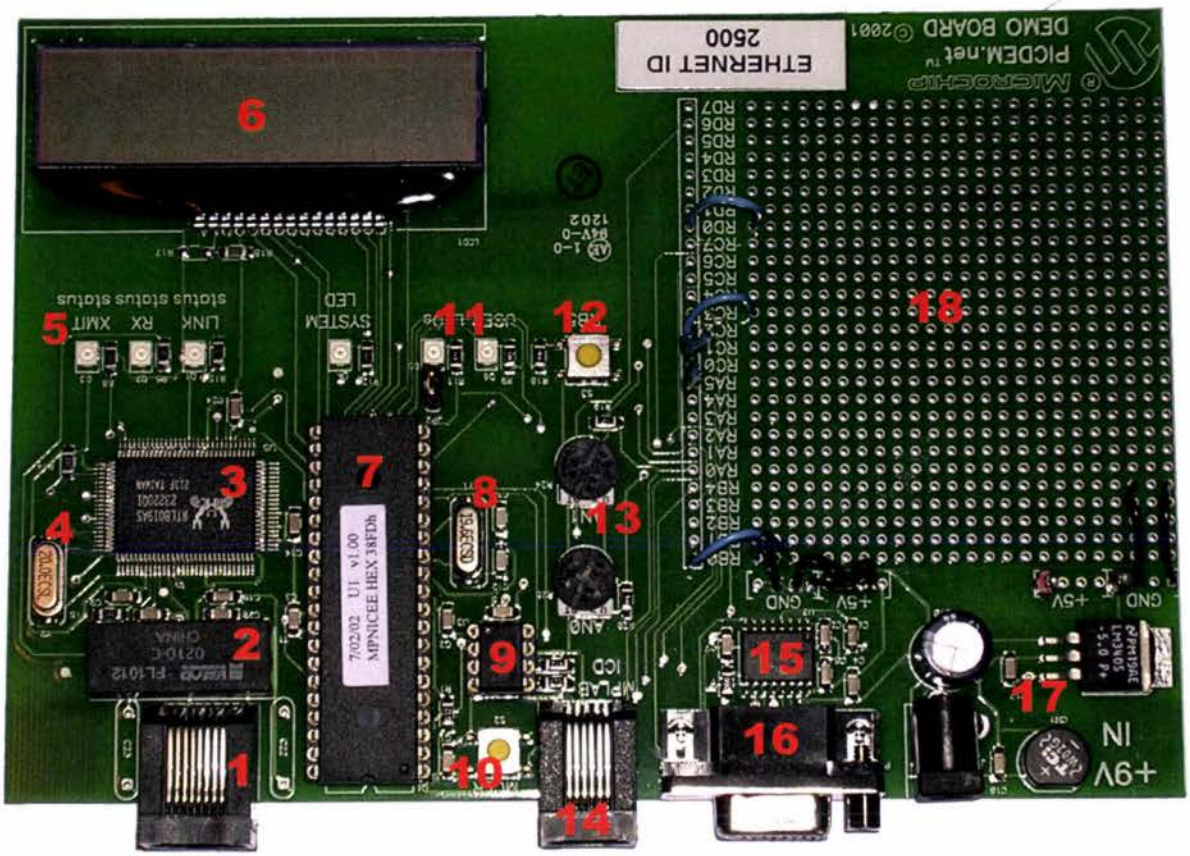


Figure 3.3: The PICDEM.net Board with labelled components

3.3.2.2 Description of numbered components:

1. **RJ-45 (10-Base T) MODULAR CONNECTOR**

   Connector provides a standard 10Base-T Ethernet interface.

2. **ETHERNET RELAY (FL1012)**

   Sets voltage levels for communication over Ethernet, comparable to a MAX-232. It also provides protection from stray voltages originating from the physical Ethernet connection.

3. **ETHERNET CONTROLLER**

   A Realtek RTL8019AS single chip Ethernet controller provides Ethernet network connectivity (10Base2/10BaseT/AUI).

4. **OSCILLATOR (20 MHz)**

   Used to generate the Ethernet controller's clock.

5. **STATUS LEDs**

   i. **LINK STATUS** – Shows that an Ethernet frame is present

   ii. **XMIT and RX** – Flashes OFF (inverted logic), when the board is transmitting or receiving a packet respectively.

6. **LCD DISPLAY**

   Two-line by 16-character dot matrix LCD display.

7. **MICROCONTROLLER (SOCKET)**

   A 40-pin DIP socket for Microchip PIC micro-controllers – 18F452 used.

8. **OSCILLATOR (19.6608 MHz)**

   Used to generate micro-controller's clock if set to use an external oscillator.

9. **EEPROM**

   A Microchip 24LC256 serial EEPROM provides 256 Kbits (32 Kbytes) of storage (programmable via a two-wire serial I2C interface.)

10. **RESET PUSH BUTTON**

    This switch is tied to the MCLR pin on the micro-controller, and is used to hardware-reset the board.

11. **USER-DEFINED LEDs**

    These two LEDs are driven by digital I/O pins of the controller, and can be used to simulate a digital output to an embedded device. They can be enabled or disabled by a jumper on the board as they correspond to PORTD:1 and PORTD:2.

12. **USER-DEFINED PUSH BUTTON**

This switch is connected to a digital I/O pin on the microcontroller, and can be used to simulate a digital input and corresponds to PORTC:3.

13. **USER-DEFINED POTENTIOMETERS**

Two 10 kOhm potentiometers are connected to analog I/O pins of the microcontroller (PORTB:1 and PORTB:2). These can be used to simulate analog inputs in an embedded application.

14. **RJ-11 (SIX-WIRE) MODULAR CONNECTOR**

This allows the demonstration board to be connected to Microchip MPLAB ICD systems for advanced micro-controller emulation, debugging and re-programming.

15. **MAX 232**

Sets correct voltage levels for communication over RS-232.

16. **RS-232 (DB9M) CONNECTOR**

Connector provides a standard RS-232 interface and allows the demonstration board to interface with a standard serial connection.

17. **ON-BOARD POWER**

An on-board, full-wave bridge rectifier allows for smoothing of the unregulated +9V DC power input. This on-board regulator provides 5V DC at up to 1A.

18. **PROTOTYPE AREA**

A 24x27 grid is provided to breadboard additional circuitry for development. Connections are provided for +5 VDC, ground, and four different I/O ports (RA<5:0>, RB<4:0>, RC<7:0> and RD<7:0>).

3.3.3 Connecting the PICDEM.net Board

There are two ways of interfacing the PICDEM.net board:

- SLIP
- ETHERNET

Ethernet was chosen as the preferred method for connecting the demonstration board as the final system would also use this interface. Because of the amount of network traffic analysers (e.g. Ethereal [21]) available, an Ethernet interface was also straightforward in terms of debugging and monitoring of data.

As the PICDEM.net acts much like any device connecting to Ethernet, the connection is straight forward. It can either be connected via a CROSSOVER cable directly to a single computer, or to an Ethernet hub with a standard PATCH cord.

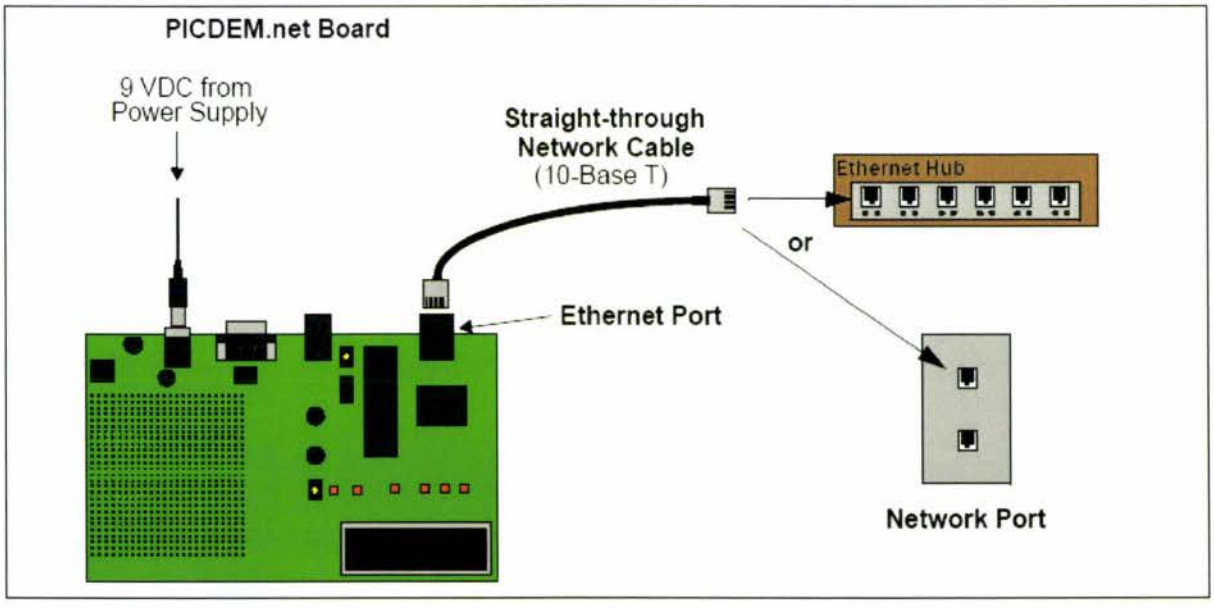


Figure 3.4: Connecting the PICDEM.net Development Board

3.3.4 A Brief note on SLIP

On microcontrollers, SLIP is still a common way of encapsulating IP packets due to its very small overhead. SLIP modifies a standard Internet datagram by appending a special "SLIP END" character to it, which allows datagrams to be distinguished as separate.

SLIP requires a port configuration of 8 data bits, no parity, and hardware flow control. SLIP does not provide error detection, being reliant on other higher-layer protocols for this.

3.4 The Demonstration Unit

To facilitate the demonstration of the prototype, a "Demonstration Box", of sorts was built. The enclosing itself was purchased from Dick Smith Electronics and then *adapted* to allow for additional connections – namely a RJ-45 plug and IEC power cable plug.

Figure 3.5.1

Figure 3.5.2

Inside the "box", an AC-DC transformer and a RJ-45 connection jack was fitted to allow the unit to run off a mains power supply.

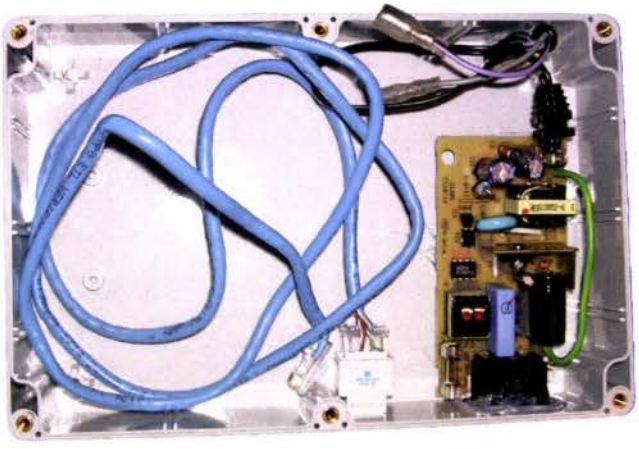


Figure 3.5.3

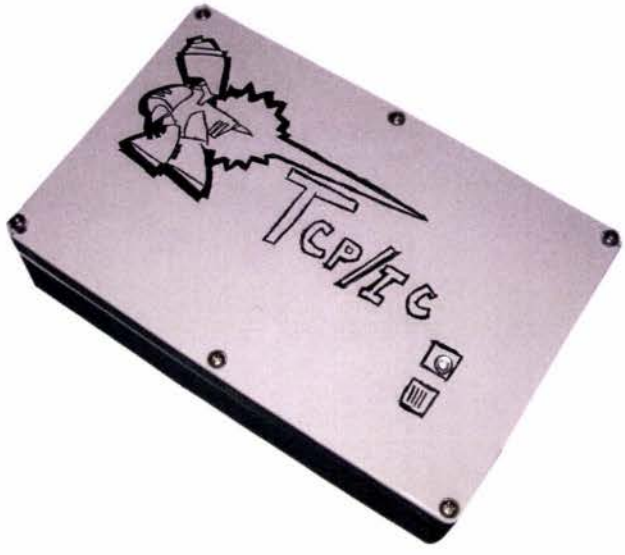


Figure 3.5.4

To demonstrate that the switching had been implemented successfully, as outputs, a LED and buzzer were also added. These components would provide an abstracted level of "feedback" from the device.

The LED was connected to the digital output pin RC0 (of PORTC) via a voltage divider circuit to provide only the necessary 3.3V required by the LED. A 330 Ohm resistor was also used to cap the current to 20 mA.

The buzzer was connected to pin RC1 (also of PORTC) and has an operating voltage from 3 – 15 V so no voltage divider was necessary for operation.

RC1 was chosen as the output for the buzzer as it is one of the two available C/C/P ports – thus allowed me to experiment with the PWM functionality of the micro-controller.

3.4.1 Additional Photo's

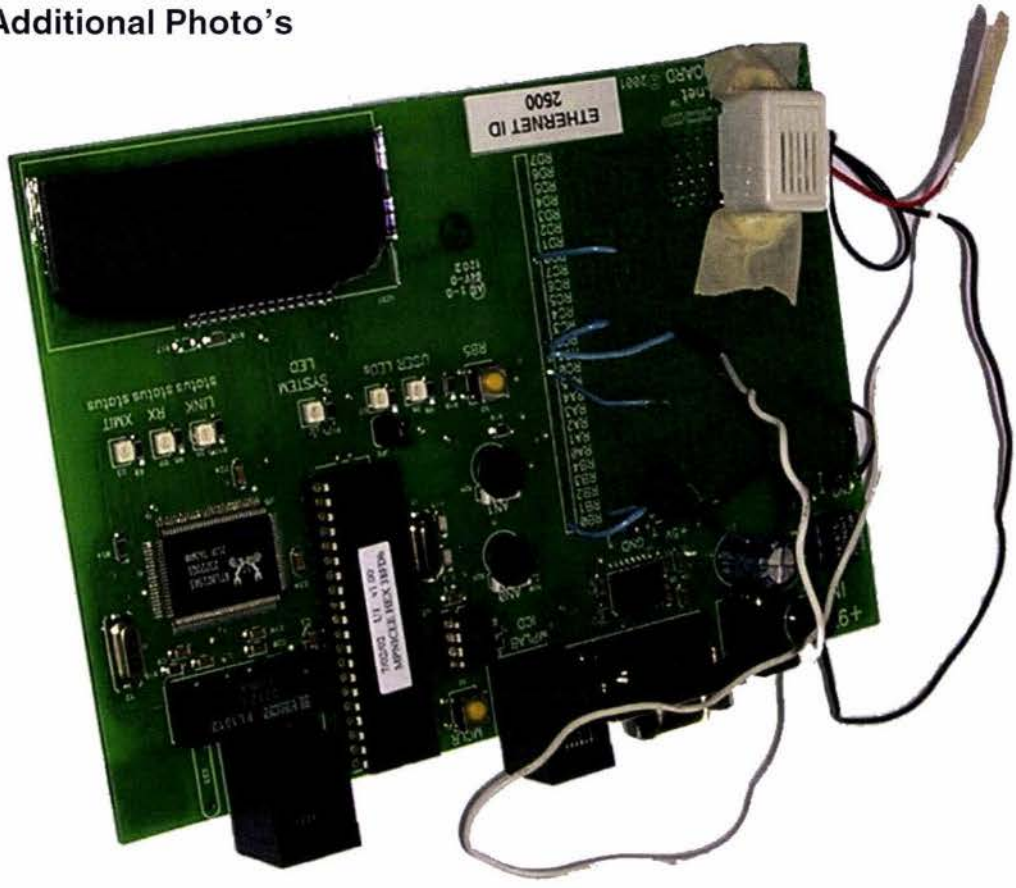


Figure 3.6: PicDem.net Circuit Board

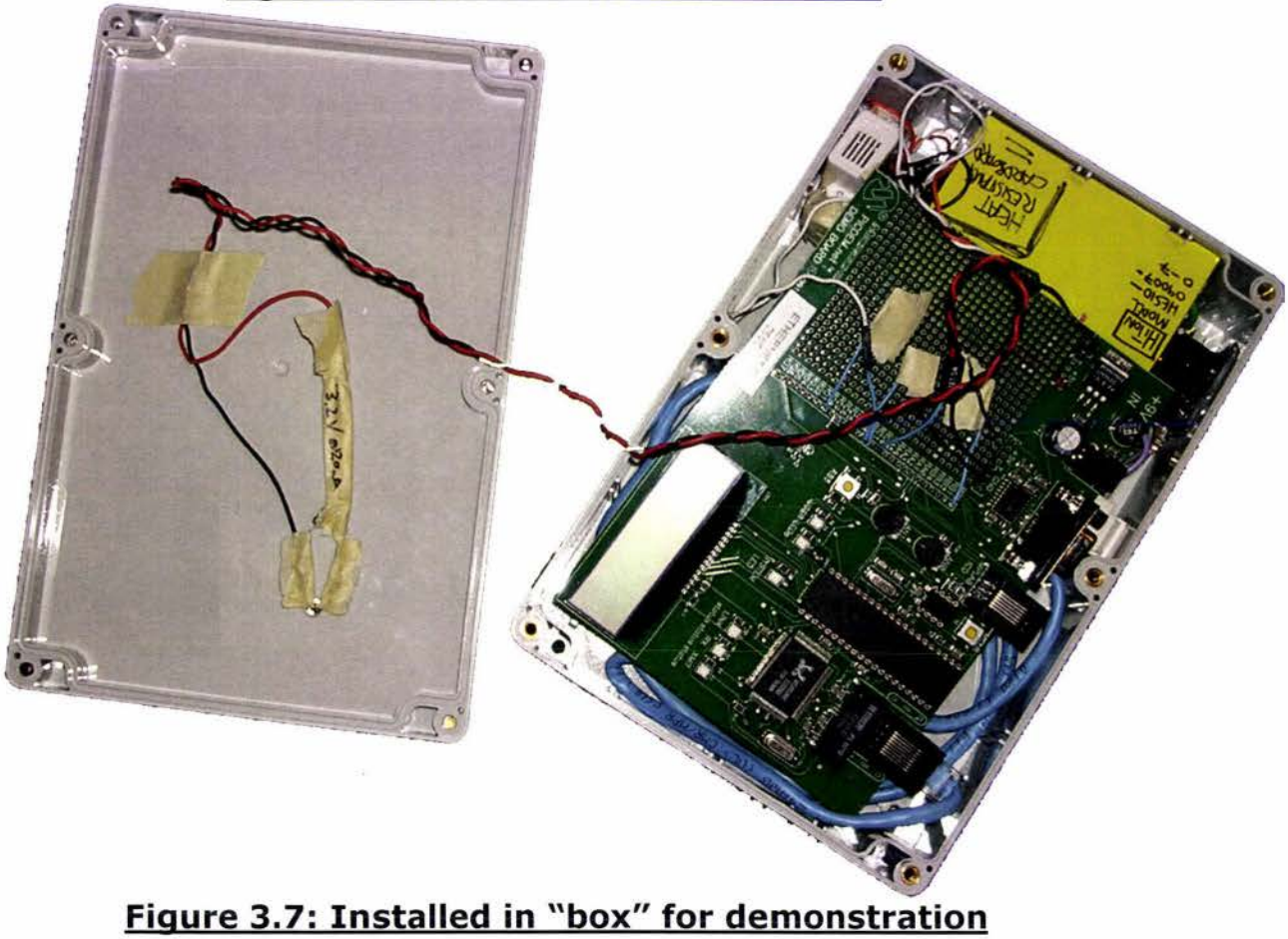


Figure 3.7: Installed in "box" for demonstration

3.5 Chapter Summary

This chapter introduced the TCP/IC, an *Embedded System* used to remotely switch and monitor, in real-time, connected devices. It outlined the various design options available, ranging from serial and parallel based systems to Ethernet based systems.

It went on to describe the various options made available by using an Ethernet based system and the advantages, along with the problems encountered by using this approach.

It also briefly introduced the concept of rapid prototyping and further detailed the rapid-prototyping board used in the construction of the TCP/IC along with the demonstration unit produced.

4 CHAPTER 4 – THE TCP/IP PROTOCOL SUITE

4.1 What is a Protocol?

For two devices to be able to communicate, they must speak the same "language". This "language" is known as a protocol. The name is derived from the framework employed by diplomats when attempting to communicate across cultural boundaries. Two computers may employ different processors, languages, and operating systems, but if they both use a common protocol, then they will be able to communicate. Examples of protocols include "TCP/IP", "AppleTalk" and "IPX/SPX".

Protocols, however, don't just enable communication, they also restrict it. By setting bounds on the protocol, neither party may stray outside the bounds without facing incomprehension or rejection. Therefore, a protocol doesn't just define how communication may occur, but also provides a framework for the information that is to be communicated [22].

The question of how any one protocol can encompass the variety of present-day computer communications then arises. The answer: it can't. We require a family of protocols, ranging from simpler, network-oriented tasks, to higher user-orientated tasks. Such a structure is often called a protocol stack, though this leads to confusion with the last in, first out (LIFO, push/pop) STACK data storage mechanism used by programmers.

In the context of a protocol, the term stack refers to the way protocol components are stacked on top of each other to give the designed functionality. If we want to transfer a file, we might take a standard file transfer protocol and stack it on top of a communications protocol. The communications protocol wouldn't understand about files – it simply moves blocks of data around. Conversely, the file transfer protocol wouldn't understand about networks – it simply converts files into blocks of data. Combine the two, and one has simple network file transfer capability.

4.2 OSI Protocol Standardisation

The International Standards Organisation (ISO) has developed a standard by which computers can communicate over a shared network [22].

The standard divides the functions of a protocol into a series of layers. This logical separation of layers makes reasoning about the behaviour of protocol stacks much easier, allowing the design of elaborate but highly reliable protocol stacks. Each layer performs services for only the next higher layer, and makes requests of only the next lower layer.

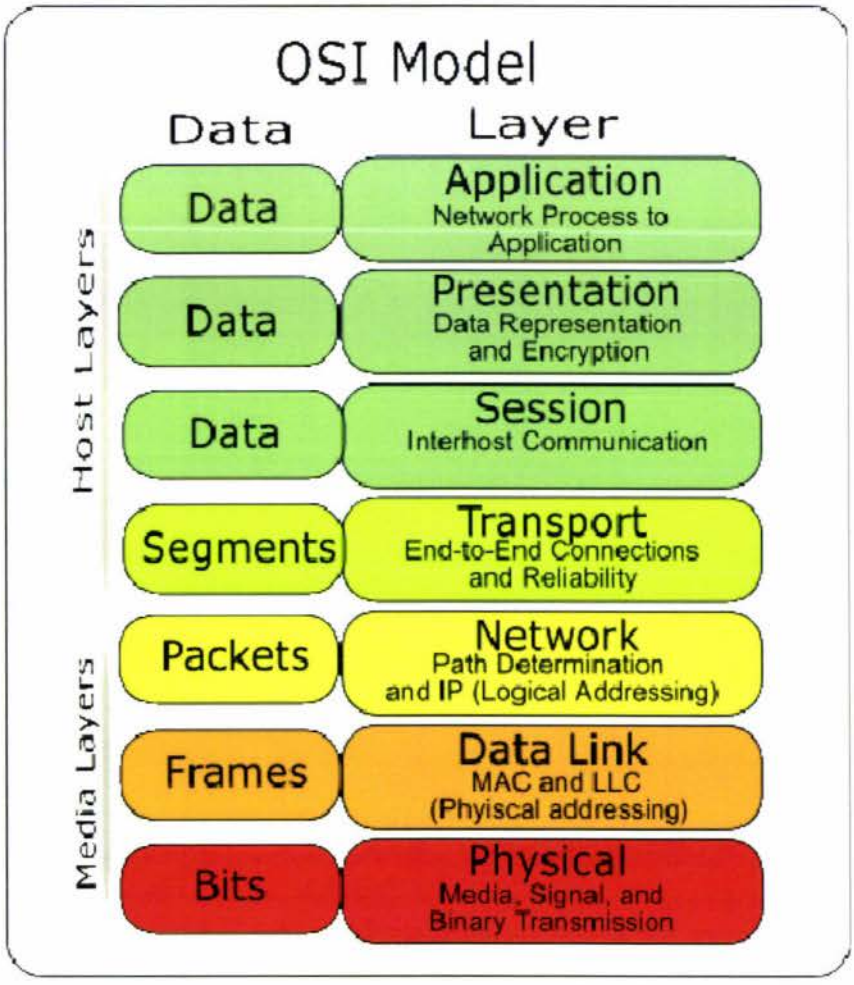


Figure 4.1: OSI Protocol Stack

An implementation of several OSI layers is often referred to as a *stack* – e.g. TCP/IP stack.

4.3 The Client-Server Model

When working with communication networks, it is useful to refer to one machine (the requester of the service) as a client and the other (the provider of the service) as the server [23]. In reality, most software is written in a way where every machine has the ability to become a client or a server.

4.3.1 Modal and Modeless Clients

Every time a client issues a command, it could go into some sort of command mode, so it knows the next communication it receives is going to be a response to that command. This mode would typically be stored in a state variable and is referred to as a modal client.

Modal techniques are frequently used in simple point-to-point serial links, but they should ideally not be used in networking, where it is impossible to anticipate what will happen next.

If one aims to keep the client as modeless (i.e. stateless) as possible, one must include more information in the data packet that is transmitted. Instead of sending pure data and storing the command mode internally, the transmitted packet must contain an indication that says "I am a command packet." The server's

response packet must then have a different indication that says, "I am a response packet."

By expressing the information externally, the client doesn't have to store it internally, and as a result, the client can deal with multiple out-of-order-packets. Troubleshooting along with debugging are also made significantly easier, as the client's intentions can be determined by examining the packets it has sent, rather than having to trace its internal data.

4.4 The TCP/IC Stack (TCP/IP Implementation)

Most TCP/IP implementations follow the OSI software architecture, as referred to previously, with the possible exception of a "fast path", where the most common transaction may be implemented as a single component encompassing aspects of several layers.

When working with the "OSI reference model", software is divided into multiple layers, where layers are stacked on top of each other

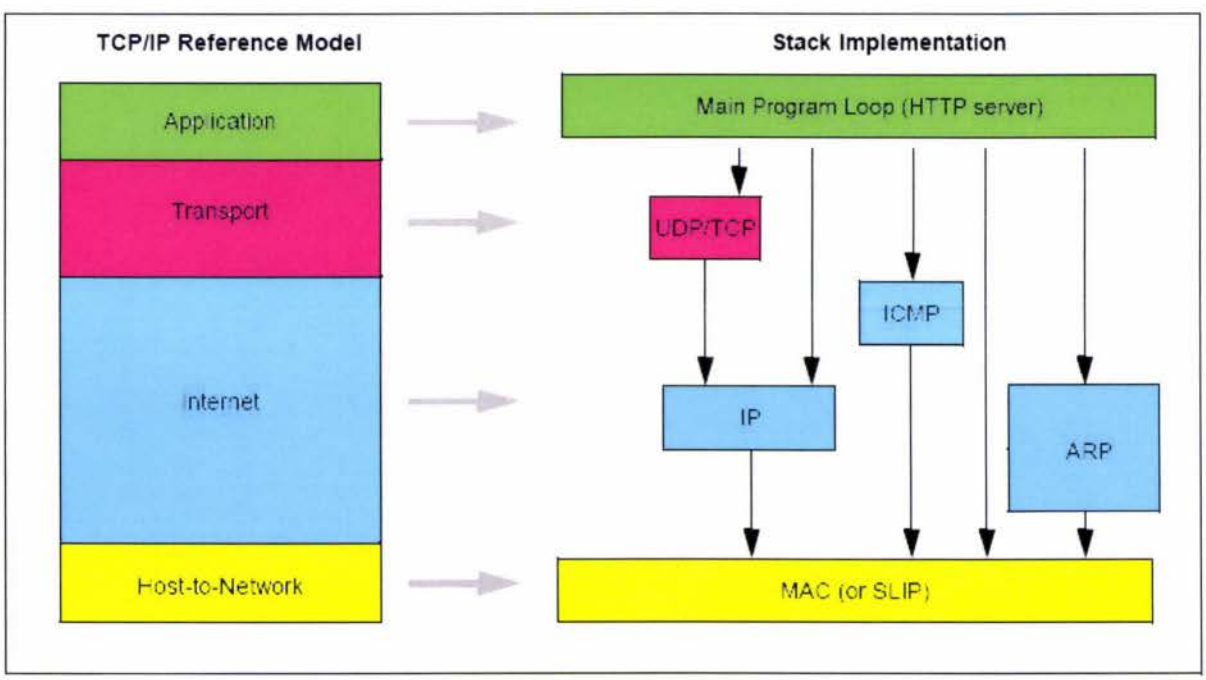


Figure 4.2: OSI Model vs. my Stack Implementation

The OSI specifications require that all of the TCP/IP layers are "live", that is they do not only act when a service is requested, but also when events like time-out or new packet arrival occurs. A system with plenty of data memory and program memory can easily incorporate these requirements as a multitasking operating system provides additional facilities/buffers and therefore make

implementation modular. But the task becomes more difficult when using a system with only an 8-bit micro-controller, a few hundred bytes of RAM and limited amount of program memory.

Furthermore, without access to a (pre-emptive-multitasking) operating system, special attention needs to be paid to make the stack independent of the main application. A TCP/IP Stack that is tightly integrated with its main application would have been easier to implement and may even be much more space efficient. Such a specialized stack however, encounters problems when applications are integrated with it.

Although not without its flaws, a co-operative multitasking environment was implemented to allow the TCP/IP stack running on the micro-controller to respond as a seemingly "live" stack.

The custom TCP/IP stack developed was adapted from the limited demonstration stack provided by Microchip (for the PICDem.net) and by the work done by Jeremy Bentham - author of "TCP/IP Lean".

As with both Microchip's and Jeremy Bentham's stack, my stack is also written in the 'C' programming language. The stack has been targeted for use with the Microchip C18 compiler, with some optimizations used to save program and data memory. However, with only relatively minor alteration, the stack should compile on any ANSI C compiler allowing it to be ported to other platforms.

It should also be noted that although the developed stack is functional, it does not implement all of the TCP/IP optimizations, etc. found in larger TCP/IP stacks. It would, however, be possible to add these features later, as a separate task/module, if required.

4.4.1 The Address Resolution Protocol (ARP)

The *Address Resolution Protocol* (ARP) is a protocol used by the IP network layer protocol to map IP network addresses to the hardware (MAC) addresses which are used by a data link protocol [24]. The protocol operates below the network layer as a part of the OSI link layer, and is used whenever IP is used over Ethernet.

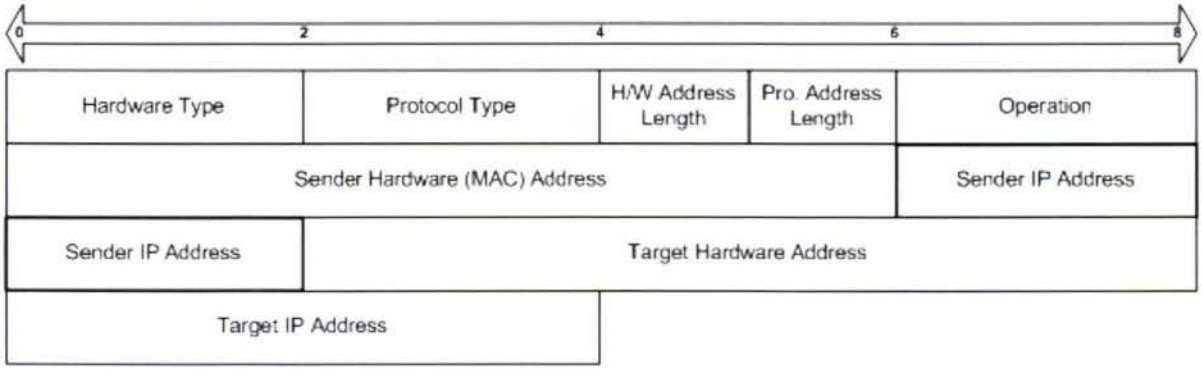


Figure 4.3: Structure of an ARP Packet

When an incoming packet destined for a host machine on a particular Local Area Network arrives at a gateway, the gateway asks the ARP program to find a physical host or MAC address that matches the IP address.

The ARP program looks in the ARP cache and, if it finds the address, provides it so that the packet can be converted to the right packet length and format and sent to the machine. If no entry is found for the IP address, an ARP request packet is broadcast to all the machines on the LAN to see if one machine knows that it has that IP address associated with it. A machine that recognizes

the IP address as its own, returns a reply. The ARP program then updates the ARP cache for future reference and sends the original packet to the MAC address that replied.

I chose to implement ARP as a finite state machine, responding to ARP requests from a remote host. It maintains a one-level cache of MAC addresses to store the ARP reply and then returns to a higher (stack) level when the appropriate calls are made. It does not implement a retry mechanism, so it has been left to the upper level modules or application to detect time-out conditions and respond accordingly.

Outlined below is the `struct` used to store the ARP packets. Refer to the code in the appendix for the complete listing.

```
/* ARP packet */
typedef struct _ARP_PACKET
{
    WORD        HardwareType;
    WORD        Protocol;
    BYTE        MACAddrLen;
    BYTE        ProtocolLen;
    WORD        Operation;
    MAC_ADDR    SenderMACAddr;
    IP_ADDR     SenderIPAddr;
    MAC_ADDR    TargetMACAddr;
    IP_ADDR     TargetIPAddr;
} ARP_PACKET;
```

4.4.2 The Internet Protocol (IP)

The Internet Protocol (IP) is the protocol by which data is sent from one computer to another over Ethernet networks [25]. Each computer on a network has at least one IP address that uniquely identifies it from all other computers on the same network.

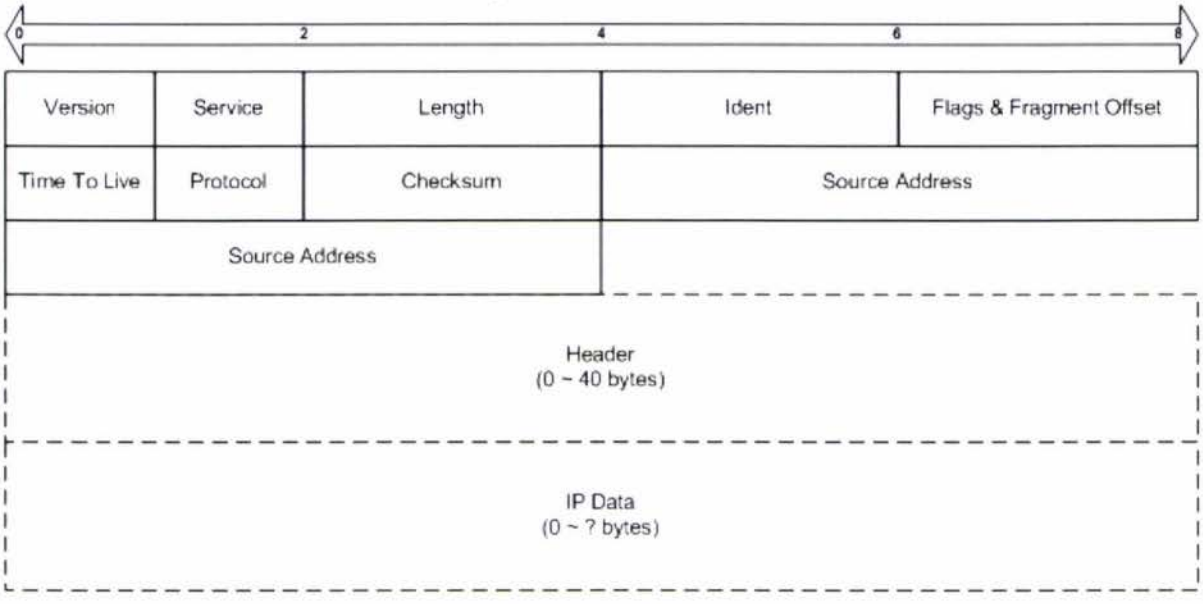


Figure 4.4: Structure of an IP

In my implementation, the IP layer is passive. That is, it does not respond to IP data packets directly. Instead, the higher level stack layers make use of IP primitives to fetch the entire IP packet, interpret it and then take the appropriate action.

The IP specification requires that the local host generate a unique packet identifier for each packet transmitted by it. This identifier allows a remote host to identify duplicate packets and discard them. In the developed stack, a 16-bit variable is maintained to track packet identifiers.

Outlined below is the `struct` used to store IP packets. Refer to the code in the appendix for the complete listing.

```
/* IP packet header definition */
typedef struct _IP_HEADER
{
    BYTE    VersionIHL;
    BYTE    TypeOfService;
    WORD    TotalLength;
    WORD    Identification;
    WORD    FragmentInfo;
    BYTE    TimeToLive;
    BYTE    Protocol;
    WORD    HeaderChecksum;
    IP_ADDR SourceAddress;
    IP_ADDR DestAddress;

} IP_HEADER;
```

4.4.3 The Internet Control Message Protocol (ICMP)

The *Internet Control Message Protocol* (ICMP) is a message control and error-reporting protocol used between a local and remote host that uses Internet Protocol (IP) datagrams [26]. It differs from IP application-level programs in that the messages are processed by the IP implementation "software" and are not directly apparent to the application user.

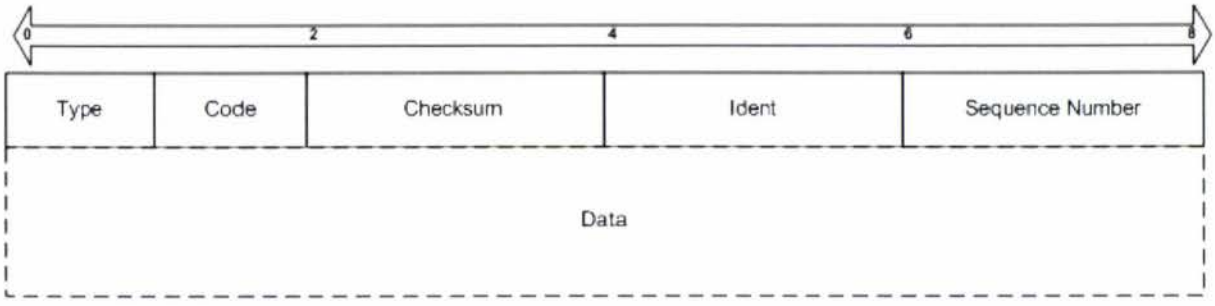


Figure 4.5: Structure of an ICMP Packet

In the developed stack, as was the case with IP, the ICMP layer is also implemented as a passive layer; it does not respond to ICMP data packets directly. Instead, as described with IP, the higher level layers make use of ICMP primitives to fetch the entire ICMP packet, interpret it and then take the appropriate action.

Normally, ICMP is used to send and receive a wide range of IP error or diagnostic messages. For this embedded application, ICMP was implemented to serve diagnostic purposes, specifically responding to ICMP "ping" packets, allowing a remote host to determine the presence of the device as well as the active state of its stack.

Additionally, the implemented stack will only respond to ICMP "ping" data packets of up to 32 bytes. Any packets larger than this are automatically discarded.

Although primarily implemented to save data memory, as it sets an upper-limit (maximum) on the packet size, this technique also helps eliminate the possibility for ICMP DOS (Denial-Of-Service) network attacks.

Furthermore, as the completed device will be accessible by other network devices, this serves to protect the SmartHouse network from common network exploits/ attacks.

Outlined below is the `struct` used to store ICMP packets. Refer to the code in the appendix for the complete listing.

```
/* ICMP packet definition */
typedef struct _ICMP_PACKET
{
    BYTE    Type;
    BYTE    Code;
    WORD    Checksum;
    WORD    Identifier;
    WORD    SequenceNumber;
    BYTE    Data[MAX_ICMP_DATA];
} ICMP_PACKET;
```

4.4.4 The Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP) is a protocol used along with the Internet Protocol (IP) to send data in the form of message units (packets) between computers over a network [27]. While IP takes care of handling the actual delivery of the data, TCP takes care of keeping track of the individual units of data (packets) that a message is divided into for efficient routing through the network.

TCP is known as a connection-oriented protocol as it establishes and maintains a logical connection between hosts until the entire message has been communicated.

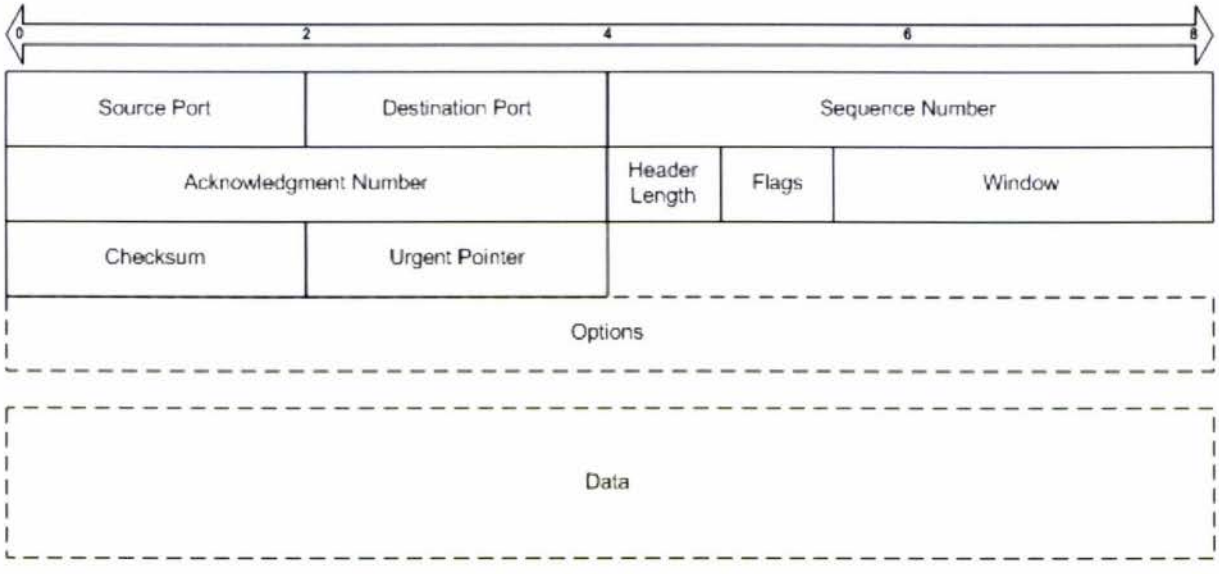


Figure 4.6: Structure of a TCP Segment

In the developed stack, TCP is implemented as an active layer. It fetches TCP packets and responds to the remote host according to the current state of the TCP state machine. To allow for this, the TCP module has been implemented as a co-operative task, thus

performing automatic operations without the knowledge of the main application.

It should be noted that unlike most modern TCP/IP implementations, all sockets in the stack share one common transmit buffer. This approach was chosen as it reduces RAM requirements and helps to eliminate a memory leak. It might, however, create a problem when a socket reserves the available transmit buffer and does not release it on time for another socket to use.

This potential problem was overcome by setting a fixed timeout period on the transmission time. The stack would simply discard the contents of the transmit-buffer (current packet) when the timer expired.

On the receiving end, there is also only one receive buffer. If a socket receives its data, the higher-layer owner of that socket must fetch and discard the receive buffer in one task time interval in order for the other sockets to receive their data.

The advantage of using a passive ARP and IP layer should now be apparent. When transmitting or receiving a particular packet "fails", the ARP or IP layer would not have responded themselves. A response would only have been sent if the packet was successfully handled by the appropriate higher-level layer.

Thus, if a remote host sends unsolicited/errored packets to the device, no acknowledgment packet (ACK) will be sent by the device as a higher-level layer has not handled the packet. This results in the "transmission error checking" effectively being done on the remote host end of the network as it will not receive any ACK if a problem occurs – meaning the remote host will have to re-transmit/re-request the effected packet. A negative to this approach is that, under certain error-prone conditions, network load would increase more than when using an "active" stack.

For example:

> When multiple hosts on an Ethernet are interconnected using a HUB (or 10BASET) and network load is already high, Ethernet collisions will occur and packets may take longer to transmit than the stack's timeout allows. The transmission will fail, but the remote host will detect this and broadcast a request for re-transmission. Upon receiving this request, re-transmission of the packet will be attempted.

As found in the Microchip demo stack, the TCP layer implements most of the TCP state machine states proposed by RFC793, including automatic retry and timed operations.

Outlined below is the `struct` used to store both the TCP segment header and TCP options. Refer to the code in the appendix for the complete listing.

```
/* TCP Header */
typedef struct _TCP_HEADER
{
    WORD    SourcePort;
    WORD    DestPort;
    DWORD   SeqNumber;
    DWORD   AckNumber;

    struct
    {
        unsigned int Reserved3      : 4;
        unsigned int Val            : 4;
    } DataOffset;

    struct
    {
        struct
        {
            unsigned int flagFIN    : 1;
            unsigned int flagSYN    : 1;
            unsigned int flagRST    : 1;
            unsigned int flagPSH    : 1;
            unsigned int flagACK    : 1;
            unsigned int flagURG    : 1;
            unsigned int Reserved2  : 2;
        } bits;
        BYTE byte;
    } Flags;

    WORD    Window;
    WORD    Checksum;
    WORD    UrgentPointer;

} TCP_HEADER;
```

```
/* TCP Options */
typedef struct _TCP_OPTIONS
{
    BYTE        Kind;
    BYTE        Length;
    WORD_VAL    MaxSegSize;
} TCP_OPTIONS;
```

4.4.5 The User Datagram Protocol (UDP)

The *User Datagram Protocol* (UDP) is a communications protocol that offers a limited amount of service when messages are exchanged between computers in a network that uses IP [28].

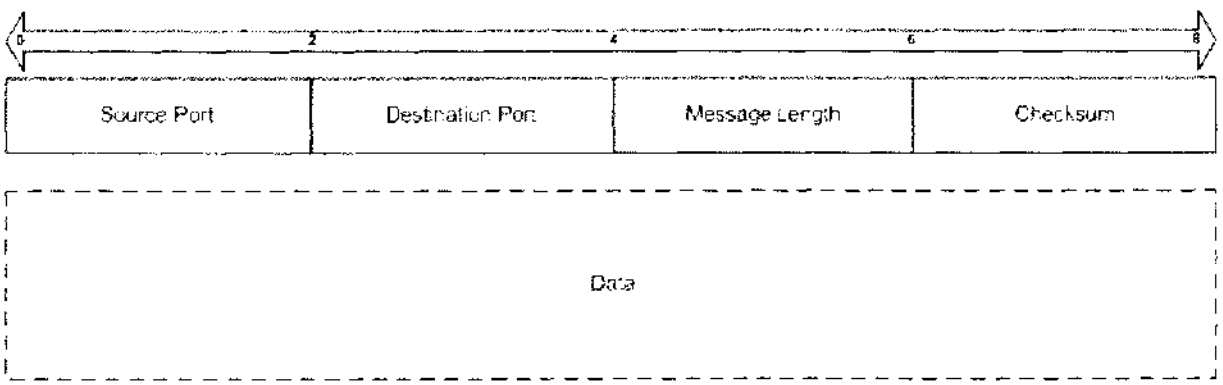


Figure 4.7: Structure of a UTP Datagram

Similar to TCP, UDP uses IP to get a datagram from one computer to another. Unlike TCP, however, UDP does not provide the service of dividing a message into packets and reassembling it at the other end. Specifically, UDP doesn't provide sequencing of the IP packets that the data arrives in. As a result, the application program that uses UDP must be able to make sure that the entire message has arrived and is in the correct order.

Although UTP has not been fully implemented in the developed stack, the DHCP "client" briefly discussed below, does make use of UTP packets and as such, UTP primitives have been implemented. A functional UTP layer could be written by simply modifying the TCP module and inserting it to run as another co-operative task.

Outlined below is the `struct` used to store UDP datagrams. Refer to the code in the appendix for the complete listing.

```
/* UDP Header */
typedef struct _UDP_HEADER
{
    UDP_PORT    SourcePort;
    UDP_PORT    DestinationPort;
    WORD        Length;
    WORD        Checksum;
} UDP_HEADER;
```

4.4.6 The Dynamic Host Configuration Protocol (DHCP)

The *Dynamic Host Configuration Protocol* (DHCP) is a communications protocol that allows network administrators to centrally manage and automate the assignment of unique *IP addresses* in a network [29].

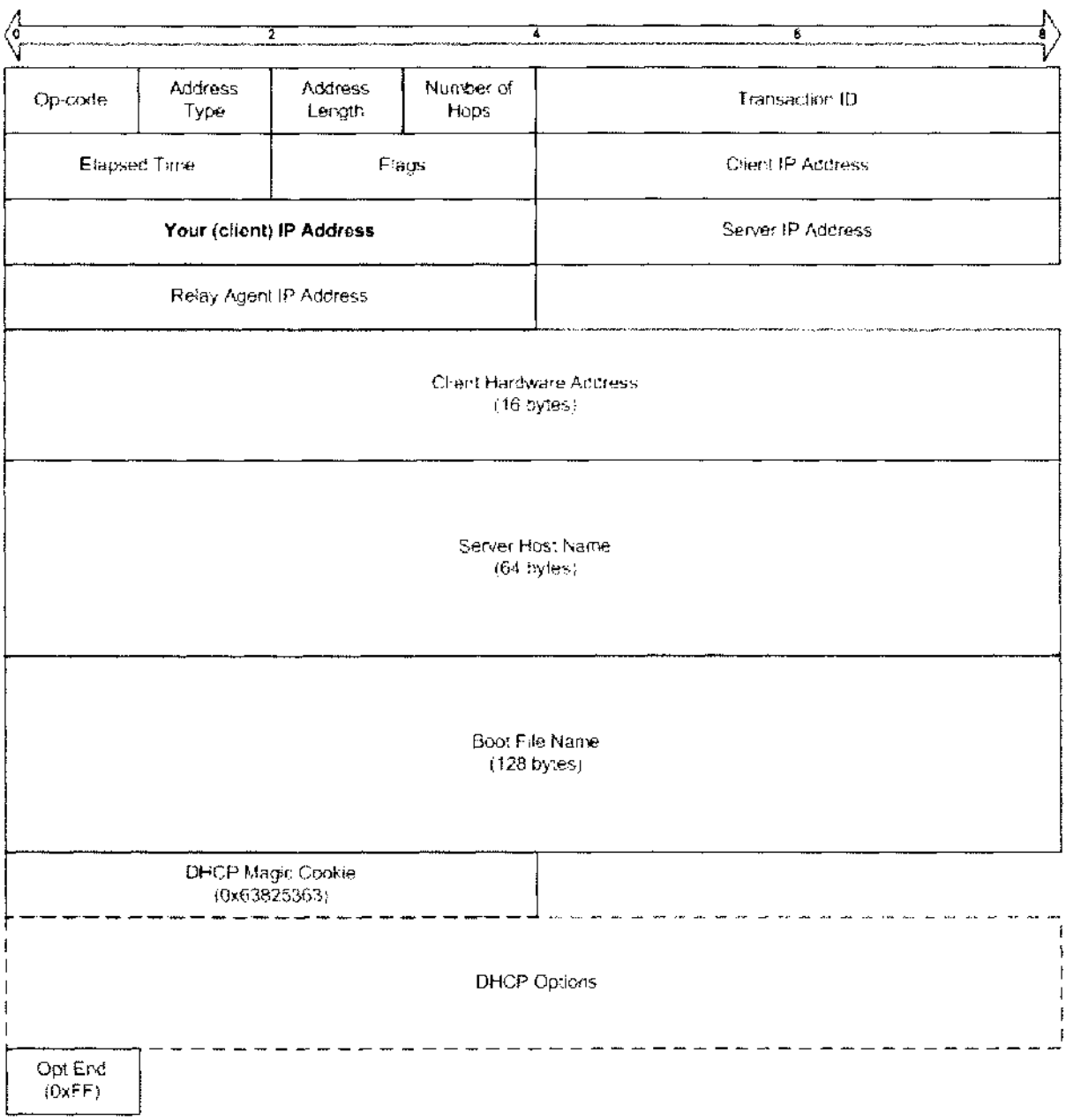


Figure 4.8: Structure of a DHCP Packet

At first, no DHCP implementation was included as a method known as *IP-Gleaning* was chosen for IP configuration. As the stack neared completion however, I chose to implement DHCP to ease installation of the TCP/IC in common networked environments, specifically the Massey SmartHouse.

Based primarily on the DHCP `struct` outlined in the Microchip PICDem.net documentation, DHCP has been implemented as an active layer that broadcasts DHCP requests and automatically receives and decodes DHCP responses, but one that ignores all other packets.

As DHCP requires a large amount or data memory, the implementation bypasses the co-operative multitasking environment by initialising immediately after the device is turned on, and components initialized, before the device is switched into its multitasking operating mode. It then polls, using the *`DHCPIsBound()`* function, for IP address information, only exiting and handing over control to the co-operative multi-tasking environment when an IP address has been received and configured.

Although this results in the TCP/IP stack not being operational before a DHCP address has been assigned, as the TCP/IP stack requires an IP address to function correctly, it has no effect on the operation of the unit other than the delay involved when receiving

an IP address using DCHP which is typically a delay of about 8-12ms.

By pushing the "push-button" on the demonstration board during start-up, the DHCP process can be cancelled and the device configured with a static-IP address before entering the multitasking operating mode.

4.4.7 IP-Gleaning (for IP Address Configuration)

As mentioned, at first, no DHCP implementation was included, as a method known as IP-Gleaning (not an Internet Protocol standard) was chosen for IP configuration. Later this was dropped in favour of the more ubiquitous DHCP.

IP Gleaning does not require any additional software modules. Instead, it uses the ARP and ICMP modules already implemented in the stack. When IP Gleaning is employed, the stack enters into a special "IP Configuration" mode upon RESET.

When in this mode, it accepted any ICMP (ping) message that is addressed to the specific MAC coded into each TCP/IC. It then uses the IP address found in the ICMP packet as its own IP address. Once a valid ICMP packet is received and IP settings configured, the stack exits from "configuration" and entered into normal co-operative mode like the DHCP implantation.

For example:

Assume that a node needs to be assigned an IP address of 192.168.1.99 and that the MAC address of this node is 0a-0b-0c-0d-0e-0f. After resetting the node, a host on the same network issues the commands:

```
arp -s 192.168.1.99 0a-0b-0c-0d-0e-0f
ping 192.168.1.99
```

An entry to the local machines ARP look-up table is added, thereby mapping a MAC address to an IP address. A "ping" packet is then sent out onto the network, allowing the device to retrieve and configure its IP address.

If successful, this should generate a standard series of ping responses from 192.168.1.99, indicating the node has successfully been assigned the IP address and that it has entered normal operation mode.

4.5 Chapter Summary

The chapter began by formally introducing protocols and explained how protocols are used in network communication – more specifically, how they don't just enable communication, but also restrict it.

The OSI model for a protocol stack was then introduced along with the client/server model and the modal techniques, frequently used in network communication.

The chapter went on to describe, in detail, how TCP/IP was implemented on the TCP/IC and how each of the involved protocols, namely ARP, IP, ICMP, TCP, UDP and DHCP were implemented.

5 CHAPTER 5 – CONTROL INTERFACES

When using a micro-controller as part of a larger system, for instance as part of a SmartHouse, both user/system and system/system interaction become important design aspects. There are many ways of interacting with a micro-controller [30], some are based on custom propriety solutions (e.g. X-10 [31]), while others are based on various open standards (e.g. IEEE [32], RFC [33]). During the development of the TCP/IC, three popular interfaces were researched, namely:

Web based Communication

This approach relies on a web-browser for all user and system interactions. An HTTP server of some kind runs on the embedded device, allowing any client with a web-browser to interact using standard HTML and a method providing device feedback (e.g. CGI, SSI, Pre-processor).

File based Communication

This approach uses a special plain-text, or binary file to relay user commands and active states. An example would be the use of a specially formatted plain-text file, such as pre-defined, custom formatted binary file or perhaps a (self-describing) XML file.

Protocol based Communication

This approach uses the underlying protocol for interaction. An example of this would be to use a specially formatted TCP or UDP packet, for instance an ICMP packet with a specific TTL, to relay commands and results.

The two sub-sections below each look at the two interfaces that have been implemented in the TCP/IC, namely Web-based and XML-based, in more detail. It also briefly covers alternative interface options and compares them to the two implanted interfaces.

5.1 Web Control Interface

Web browsers have become the standard user interface to a variety of applications. They can run on almost any platform, from PCs and workstations to PDAs, cell phones, and pagers, all while allowing end-users to access Web-enabled applications from any location.

When applied to Embedded Systems, Web technologies offer graphical user interfaces that are user friendly, inexpensive, cross-platform and network-ready. System designers are therefore using them to supplement or replace traditional proprietary command-line and graphical user interfaces [23]. Web interfaces also offer benefits to manufacturers, as well as to end-users [34].

Because Web-based interfaces are cross-platform and easier to develop than traditional proprietary interfaces, manufacturers enjoy a shorter time-to-market for new product features [35]. Moreover, the costs associated with product support, training, and documentation are lower and the Web interface can also be used for remote diagnostics of products in the field.

5.1.1 Web Technologies for Embedded Systems

As with the Web in general, Web pages from the *Embedded System* are transmitted to the clients Web-browser, which then implements the user interface. Some of these pages are unchanging (static) and can be stored in the *Embedded System's* internal or external program memory. In other cases, the *Embedded System* dynamically generates the pages to convey the current state of the device, or that of its sensors, etc. to the user. End-users can also use Web-browsers to send information to the *Embedded System* to configure or control the device.

Web-enabled devices use the HTTP protocol to transmit Web pages from the *Embedded System* to the Web-browser and to transmit HTML form data from the browser back to the device [34]. In order for these devices to be able to communicate via HTTP, they require a network interface such as Ethernet or SLIP, a TCP/IP stack, an embedded Web server and the Web pages (both static and dynamic) that make up the device-specific GUI.

The figure below shows how Web-enabled devices use the HTTP protocol to transmit Web pages from the *Embedded System* to the Web-browser, and to transmit HTML form data from the browser back to the device.

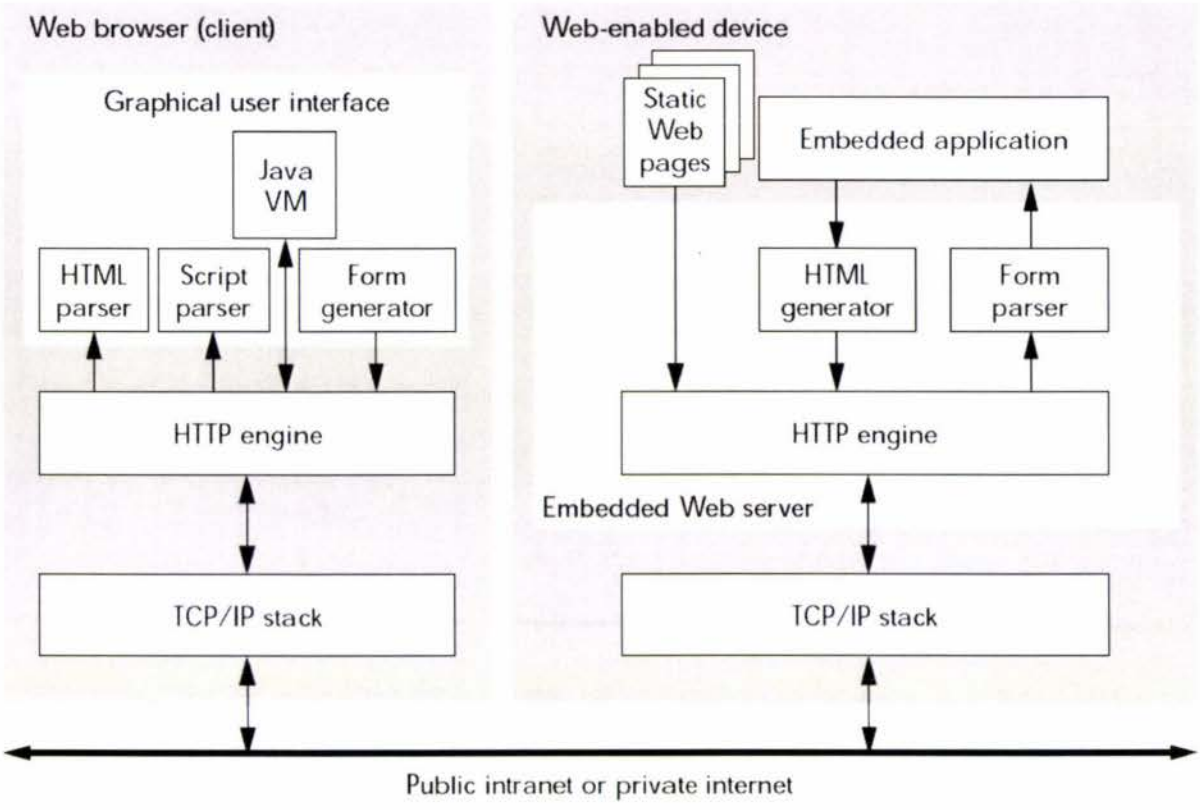


Figure 5.1: Relationship between Web-Browser and Embedded System

5.1.2 Design Issues for Embedded Systems

Traditional Web-servers are designed to serve static Web pages from high-end workstations with plentiful CPU and memory resources. Embedded Web-servers have different requirements for which traditional technologies are unsuitable. Outlined below are some of the considerations that have to be kept in-mind when designing an embedded Web-server [36].

5.1.2.1 Protocol Considerations

The core of the embedded Web-server is the HTTP engine. HTTP has evolved rapidly from the simple HTTP 0.9 used in the first experimental Web servers, to HTML 1.0 [37], followed by HTTP 1.1 [38].

While implementing a subset of HTTP 1.0 is fairly easy, one of the largest problems faced was that HTTP1.0, as is the case with HTTP1.1, lacks explicit cache control. Under certain conditions, Web-browsers and intermediate proxies may cache Web pages. For static Web-pages such as graphics (e.g. icons and logos), caching is desirable as it reduces the CPU load on the *Embedded System* by eliminating requests for redundant information.

Dynamically generated pages, however, must not be cached. Caching these pages would prevent the Web-browser from retrieving up-to-date information about the status of the system, thereby rendering the Web-interface, with regard to configuration and control, useless.

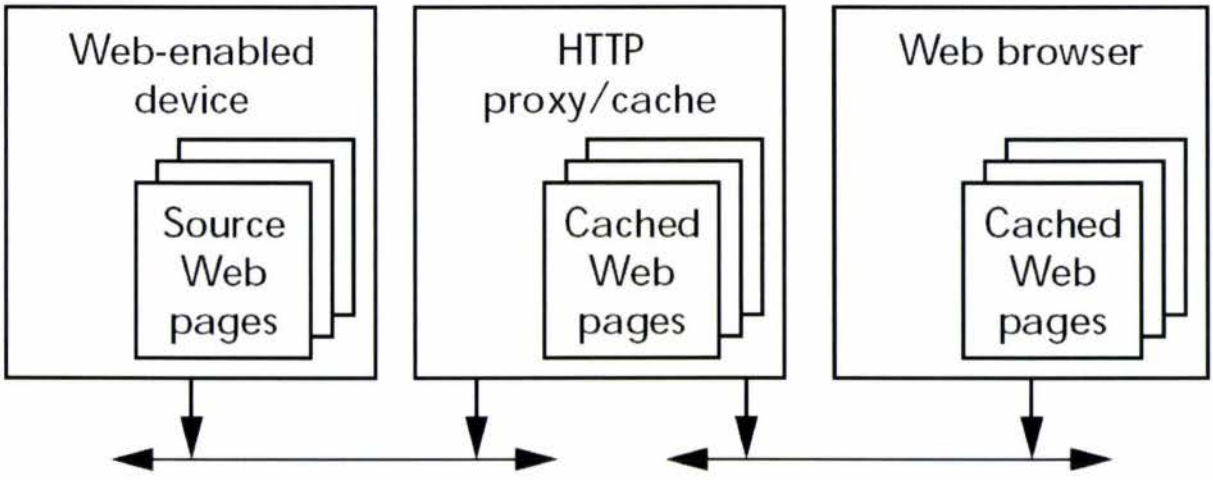


Figure 5.2: Intermediate Caching of Web pages.

Keeping this in mind during the design phase of the TCP/IC, a special system was implemented to ensure dynamic data was not cached.

5.1.2.2 Embedded Software Considerations

Another major consideration is noting that Embedded Systems have only limited CPU and memory resources available to them. In many cases, these limited resources are largely committed to mission-critical and real-time applications present on the system.

For example:

> A data communications router may require most of the CPU and memory resources of the *Embedded System* to store-and-forward packets. Unless the embedded Web-server is sensitive to the real-time requirements of mission-critical applications, it might drop packets (lose data). This would result in the router not performing its primary function!

In fact, in reality, the embedded Web-server is often the lowest priority service in the *Embedded System* with the result that end-users can end up waiting hundreds of milliseconds for as response from the Web-server – an eternity when compared with the low-latency requirements of many embedded applications.

As it currently stands, relatively few Embedded Systems have complete Operating Systems. With this in mind, for an embedded Web-server to remain portable between systems, the embedded Web-server should run independently of the Operating System service, such as multi-threading.

5.1.2.3 Embedded Applications Interface Considerations

As mentioned, embedded Web-server software must provide a mechanism for the embedded application to dynamically generate and serve Web pages to the browser, as well as process HTML FORM data, submitted by the browser.

5.1.2.4 Common Gateway Interface (CGI)

A possible solution to dynamically creating web pages is to model the CGI found in more traditional Web-servers [39]. When using this model, each URL is mapped to a CGI script that generates the Web page content.

In a typical *Embedded System*, the CGI "script" would actually be implemented by a function call (`call-back function`) to the software running on the *Embedded System*. The software would then send raw HTML, XML, or other types of data back to the browser by using the interface provided by the embedded Web-server.

While this approach is perhaps the easiest for the embedded Web-server developer, it is by far the most difficult for the GUI designer. CGI scripts, especially when employed using call-back functions, are tedious to write, test and to debug. Once written the "look and feel" of the Web page can only be determined when the script is actually executed, most commonly not by an emulator, but by the *Embedded System* itself. This involves building the binary image,

burning it into the device's flash-memory, and booting the device before the newly designed, or altered, Web pages can be viewed by a browser.

e.g.

The code below shows how a CGI call-back function implements state variables with regards to dynamic page content.

```
CGIprintf(ctx ,
        "<html>\n"
        "<head>\n"
        "<title>Room #1 Status</title>\n"
        "</head>\n"
        "<body>"\n"
        "The light is %s<p>\n"
        "<form method=post>\n"
        "Temperature: <input type=text name=temp
        value=%d>\n"
        "</form>\n"
        "</body>"\n"
        "</html>\n"
        ,status
        ,Temperature);
}
```

CGI solutions are easy to write, but tedious to test and to debug. They therefore slow down the time-to-market and are more difficult to maintain once deployed.

5.1.2.5 Server-side Scripting

Another solution to dynamic page creation is to use a method of server-side scripting. With this approach, Web pages are first developed and prototyped using conventional Web authoring tools and browsers. Next, proprietary mark-up tags that define server-side scripts, are inserted into the Web pages.

For example:
The code below shows the server-side script in a sample HTML page. These marked-up Web pages are then stored in the device by implementing a simple file system.

```
<html>
<head><title>Room #1 Status</title></head>
<body>
The light is $TEXT(status)
<form method=post>
Temperature: <input type=text name=temp
value=$TEXTINPUT(temp)>
</form></body>
</html>
```

When a marked-up Web page is served, the embedded Web-server first interprets and then executes the script to interface with the embedded application. For example, a proprietary scripting language could define an interface to invoke application functions used to generate the dynamic part of a Web-page.

Server-side scripts are easier to use than the raw interfaces of CGI based solutions. However, interpreting scripts at runtime in an

Embedded System may have a high impact on system performance.

Furthermore, a significant amount of memory is required to maintain a database for mapping script names into embedded software functions and variables.

Server-side scripts generally offer limited capabilities, resulting in larger and more complex embedded application code, while allowing for streamlined development, testing and debugging.

5.1.2.6 HTML-to-C Pre-processor

A third solution to creating dynamic pages makes use of a pre-processor tool that converts Web pages into C code. Like server-side scripting, this approach uses conventional Web authoring tools and browsers to support quick development and prototyping of Web pages.

The Web pages are then enhanced with proprietary mark-up tags that encapsulate fragments of C source code. These code fragments provide simple and efficient interfaces to the embedded application software.

e.g.

The code below shows the an HTML page containing encapsulated C source code.

```
<html>
     <head>
     <title>Room #1 Status</title>
     <C_HEAD C='#include "defs.h"'>
     </head>
     <body>
     The light is <C_STRING C='
     if (LightIsOn)
     {
          return "on";
     }
     else
     {
          return "off";
     }'>
     <form method=post C_NAME=temp>
     Temperature: <input type=text name=temp C_TYPE=
     DECIMAL_INT>
     </form>
     </body>
</html>
```

The pre-processor tool then compresses the Web pages, strips out the proprietary tags, parses HTML forms, and generates C code. The C code is then compiled and linked with the embedded Web-server and application software to produce a tightly integrated binary image.

The use of a pre-processor enables sophisticated dynamic Web-page capabilities by performing complex tasks up front and generates an efficient and tightly integrated representation of the Web pages and interfaces in the *Embedded System*.

The figure below shows how a pre-processor compresses the Web pages, strips out the proprietary tags, parses HTML forms, and generates C code. The C-compiler then compiles and links the code with the embedded Web server and application software to produce a tightly integrated executable image.

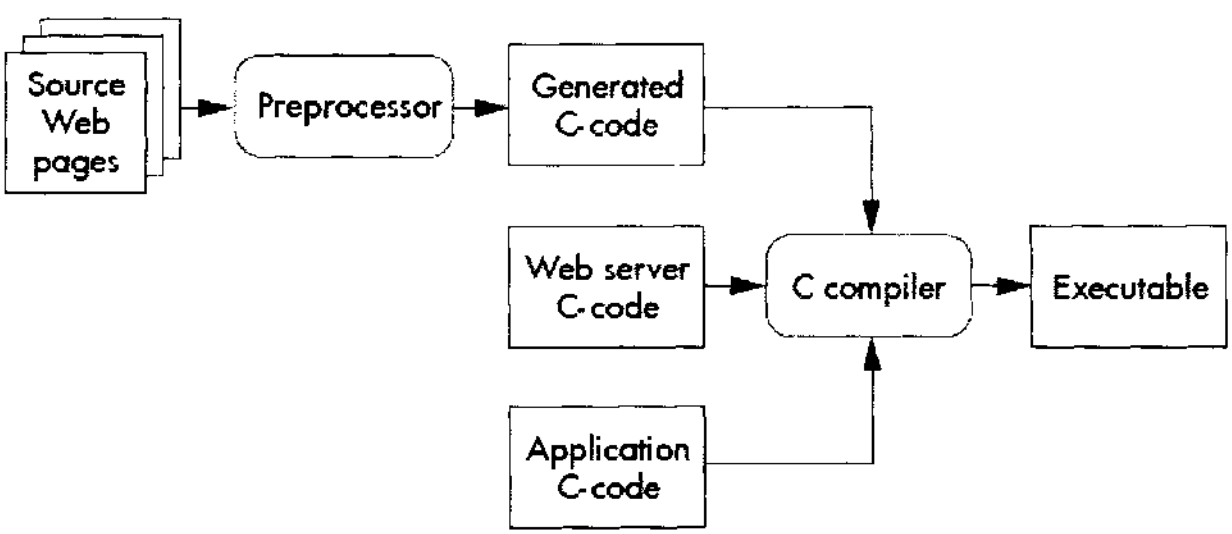


Figure 5.3: HTML-to-C Pre-processor

The HTML-to-C pre-processor offloads substantial Web-server processing from the *Embedded System*. This approach supports a small, efficient embedded Web server that increases system performance, but reduces the systems flexibility with regard to debugging and system maintenance.

5.1.3 TCP/IC Implantation: Embedded Web Server

Once I was able to create a TCP/IP stack that would accept and reply to TCP packets, it was a relatively simple task to implement a basic, static web-server as a module in the main embedded application.

The HTTP standard defines how clients and servers communicate with one another. When a Web-browser requests information form a Web-server, it is required to use a pre-defined request sting. Similarly, when the server replies, it too uses a pre-defined response.

```
For example:
A standard-compliant web-client (i.e. browser) would use
the following request to retrieve the file index.htm.

    GET /index.htm HTTP/1.0
    User-Agent: Mozilla/4.5 [en]
    Pragma: no-cache
    Host: 192.168.1.20
    Accept: image/gif, image/jpeg, image/png, */*
    Accept-Encoding: gzip
    Accept-Language: en
    Accept-Charset: iso-8859-1.*.utf-8
```

The modifications required to implement a functional Web-server on the TCP/IC were trivial. All that was required was to allow the TCP/IC to accept a series of formatted strings. These strings form the request made via HTTP (shown in above example.) The Web-server must then send an specially crafted HTTP response

(header), followed by the page -data as specified in the request string.

Although a significant portion of data memory would be required to accept the request in its entirety, much of this information can be discarded, particularly, the information about user compatibility as the Embedded System can't do much if the client cannot display the file-types (MIME-types) being sent.

This proves to be useful when we consider that the micro-controller has very limited RAM. Text manipulation is also difficult due to the strict type-casting encountered when working with a micro-controllers bank memory. As a result, the embedded Web-server only deals with the first line of text (as indicated above) and crafts only a minimal response. This response would be a string similar to the one shown below.

```
HTTP/1.0 200 OK
Content-type: text/html
```

The HTTP server running on the PIC is in essence the main application and is coded in a way to allow for dynamic content creation (discussed later). It is implemented as a co-operative task module as this allows it to co-exists with the TCP/IP stack.

It should be noted that the HTTP server implemented on the TCP/IC does not implement all HTTP functionality. It is a minimal server targeted for use on an *Embedded System*. One could, however, add additional features if required.

The main features incorporated by the TCP/IC's embedded HTTP server include:

- Support for multiple HTTP connections
- Contains a simple file system (MPFS)
- Supports the HTTP method "GET" (other methods can be added)
- Supports a modified CGI to invoke predefined functions
- Supports dynamic web page content generation

A limitation imposed by the implementation is that it is necessary to generate Web pages in advance and convert them into a compatible format for storage on the EEPROM. This is the main reason for utilizing a basic file system (MPFS). A PC-application is used to create a BIN file that is then "uploaded" into the EEPROM.

5.1.3.1 Complexity of Web Pages

If a Web page contains several graphics, the browser will attempt to fetch them all at the same time by opening multiple connections to the server [37]. Opening multiple connections can cause a problem when dealing with the TCP/IC as it is hampered in its

ability to handle multiple requests by the lack of RAM / buffer space.

If it is currently transmitting the response to one request when another arrives, it has no buffer space left, so it must discard the second request. Of course, the browser will retry the second request, so it will eventually succeed, but the Web page update time will be increased.

5.1.3.2 Dynamic Content Generation

Although it is impressive that such a small, low-powered 8-bit micro-controller can actually serve static Web pages, it is of no real use if the pages do not allow for dynamic content: real-time data that would not be available to a conventional disk-based system. Ideally, one should be able to interact with the miniature server: set its digital output lines and read from its various peripherals.

The TCP/IC's HTTP server has been designed to be able to dynamically alter pages and substitute real-time information, such as input/output status, by making use of certain call-back functions. This approach is has its roots in CGI but is implemented in a SSI manner.

For example:

The HTML/CGI call-back function

```
WORD HTTPGetVar(BYTE var, WORD ref, BYTE *val)
```

The use of a call-back function requires that the corresponding CGI file contains a text string such as "%xx", where the "%" character serves as a control code and "xx" represents a two-digit variable identifier. When the HTTP-server encounters such a text string, it removes the '%' character and "calls" the call-back function.

5.1.3.3 HTTP CGI

The HTTP server also implements a modified version of HTTP CGI. With this interface the HTTP client can invoke a function within HTTP and receive results in the form of a Web page. A remote client invokes a function by making use of the HTML GET method along with more than one parameter.

When a remote browser executes a GET method with more than one parameter, the HTTP server parses it and calls the main application with the actual method code along with its parameters.

For example:

```
WORD HTTPGetVar(BYTE var, WORD ref, BYTE *val)
void HTTPExecCmd(BYTE **argv, BYTE argc)
```

By creating the two call-back functions shows above, the HTTP server is still able to run as a co-operative task – as apposed to polling for data – thereby allowing the TCP/IP component so function as it should.

Refer to the code in the appendix for the actual code used in these two call-back functions.

5.2 XML

The eXtensible Markup Language (XML) is a simple, flexible, plain text based data format derived from SGML [40], an open standard for describing data from the W3C (World Wide Web Consortium) [41]. It is used to define data elements on a Web page and assist in business-to-business communication [42].

As with any SGML based language, XML uses a similar tag syntax as HTML. However, where HTML defines how elements are presented, XML defines the data which those elements contain. As a result, while HTML uses a set of predefined tags (i.e. HTML 4.0 [43]), XML tags are defined by the developer.

5.2.1 XML and HTML Tags

The following code-snippet is an example of XML and HTML tags showing how XML statements define data context and how HTML statements define presentation.

XML

```
<UID>130<UID>
<Location>Lounge</Location>
<Peripherals>
     <ID1 name="aircon">
          <Status>1</Status>
     </ID1>
     <ID2 name="lamp">
          <Status>0</Status>
     </ID2>
</Peripherals>
```

HTML

```
<font size="+2" color="red">
     <strong>Device 130 located in the "Lounge"
     </strong></font>
<br><ul><li>Air-conditioner is ON</li>
<li>Reading Lamp is OFF</li></ul>
```

5.2.2 XML and Embedded Systems

As Embedded Systems are becoming more and more Internet enabled they often expand on their core services and provide interesting new services. Often this involves interaction with the external environment and transmission of data to databases that store data for further processing and analysis.

If one tries to classify most Embedded Systems, currently one can distinguish between two broad types [42]:

- Isolated Embedded Systems that perform a narrow set of functions, and
- Distributed Embedded Systems that interact with an external environment to perform a wide range of functions.

When dealing with "Distributed" Embedded Systems, the need to communicate and share data in a structured way arises, along with a need for a suitable format for the exchange of data between different data sources.

Additionally, as the device will possibly be working with multiple data sources, a method for storing data, using clearly defined semantics, is also required [44].

As for storing information gathered from Embedded Systems, the semi-structured nature of XML data, allows for an easily extensible data store [42]. Something that is of considerable importance as the number and type of sensors may evolve.

5.2.3 TCP/IC XML Control Interface Implementation

XML was chosen for its ease of use, human-readable text-based format and wide scope of interoperability. More specifically, as TCP/IC was developed with the Massey SmartHouse project in mind, interoperability refers to the communication between the SmartHouse AI and the multiple TCP/IC devices. Working closely Ms. Grettle Lomiwes, responsible for the SmartHouse AI, allowed a XML Schema and a set of defined XML tags to be developed.

By programming the SmartHouse AI to call a specially crafted URL, an XML response can be generated. This XML data contains the ID and state of the peripherals connected to the particular TCP/IC device.

For example:

The XML listing shows the XML response of the TCP/IC

```
<TCPIC>
     <UID>130<UID>                    // Unique device ID
     <Location>Lounge</Location>      // Location
information
     <Peripherals>
          <ID1 name="aircon">         // "name" equals
                                      //  Short Name
               <Status>1</Status>     // 1 == ON
          </ID1>
          <ID2 name="lamp">
               <Status>0</Status>     // 0 == OFF
          </ID2>
     </Peripherals>
<TCPIC>
```

In the above example, the <UID> tag was used to identify the specific TCP/IC device the SmartHouse AI would be communicating with. To ease in installation, this was also the last octet of the IP address (i.e. xxx.xxx.xxx.UID / 192.168.88.130).

The <Location> tag was used to quickly identify the location of the unit. Initially, the SmartHouse AI incorporated this information in its relational database, but it was decided that to reduce strain on the database, the TCP/IC would also include the "redundant" information within its XML responses.

The <Peripherals> tag is where to the switchable outputs, along with their current state (on/off) is stored. The format of <IDx> was chosen for simplicity, as to toggle the output. The crafted URL required this ID number. A "quick-name" was also given to each output to further assist the SmartHouse AI and offload the database.

5.3 Alternative Interfaces

As mentioned at the beginning of this chapter, there are many ways to communicate with a network-enabled *Embedded System*. Although implemented with a high-degree of success, both methods employed by the TCP/IC are not as efficient as alternative methods, and as a result, can be misinterpreted as wasting both computing time and network bandwidth.

Two additional, perceivably more efficient interfaces are described below. Each has its own advantages and disadvantages which will be compared to the TCP/IC XML and HTML interfaces.

5.3.1 File Based (custom formatted plain-text file)

If we make use of the same example used in the XML section (refer to previous section), we can see that if the application controlling the device, in this case the SmartHouse AI, was made aware of the entire feature-set of the *Embedded System* at design time, a much smaller text file would need to be sent.

```
For example:
     UID&130          // UniqueID
     aircon&1         // Device an ON status
     lamp&0           // Device and OFF status
```

Using "&" as a delimiter, as long as the SmartHouse AI has facilities for specifying how the file should be formatted, the example above would relay the same information as XML, using significantly fewer bytes: 24 bytes vs. 195 bytes.

Although this equates to almost a 812% improvement over formatted XML, it is not self-descriptive and can only be used by an application which knows its format along with the details of the *Embedded System*.

This results in communication not being interoperable and understood only by the system's developers. When using this type of interface, the TCP/IP packet size would be around (20+20+24) 64 bytes.

5.3.2 Protocol based (custom TCP packet)

If make use of the same example, we can see that if the application controlling the device, the SmartHouse AI, was again made aware of the entire feature-set of the *Embedded System* at design time, and was able to use the OS's TCP/IP stack directly (e.g. .Unix, Linux, etc.), a custom TCP packet could be created that could control the unit.

```
For example:
( PACKET LENGTH set to 8 bytes )
     BYTE 0 = "1"    // UniqueID digit 1
     BYTE 1 = "3"    // UniqueID digit 2
     BYTE 2 = "0"    // UniqueID digit 3
     BYTE 3 = "1"    // Device 1 Status
     BYTE 4 = "0"    // Device 2 Status
     BYTE 5 = "&"    // Termination Character
```

If the SmartHouse AI has facilities for specifying how the TCP/IP packets should be formatted, the example above would relay the

same information as both the XML and pre-formatted text files, using significantly fewer bytes: 8 bytes vs. 24 bytes vs. 195 bytes.

This equates to a staggering 2437% improvement over XML and a 12% improvement over custom-formatted plain-text. Here, the TCP/IP packet size would (20+20+8) 48 bytes.

5.3.3 Interfaces Compared

Although both Web and XML based interfaces have been shown to be inefficient when compared with the two methods shown above, they do offer much more in terms of usability, scalability, interoperability, data integrity and even security.

It is also very important to note that as we are dealing with TCP/IP, the relative size of a packet comes into play. When packets are sent on a network, they incur the overhead of TCP/IP (i.e. the *TCP Header* and *IP Header*).

Assuming the largest TCP/IP packet allowable (MTU) is 576 bytes (defined by RFC), subtracting 20 bytes for the TCP header and 20 bytes for the IP header, we are left with 536 bytes for a payload. As shown above, the interfaces yield a packet size of:

```
64 bytes (64% overhead) : custom-formatted plain-text
48 bytes (83% overhead) : custom-formatted TCP packet
235 bytes (17% overhead): using XML
```

As the latency involved with packet-based, network communication can be considered to be almost equal when the packet MSS is smaller than the MTU. In the case of the TCP/IC, as with most implementations, this is 576 bytes. One can see that although using a custom-formatted plain-text file or a custom-formatted packet may be much more efficient in terms of storage, when we consider that a network is being used to transport the data. We may not see any improvement in network utilisation once we consider the latency involved.

5.4 Chapter Summary

This chapter introduced the various interface options available when dealing with remote devices, along with the advantages each offer. It went on to focus on the two interface options chosen for the TCP/IC, namely HTML and XML.

It described what Web-based interface offered to Embedded Systems and detailed how this was implemented on the TCP/IC. XML and its implementation was also introduced.

Alternative interfaces were then described and compared with the two interfaces used by the TCP/IC. The chapter concluded by showing how, although other, seemingly more efficient interfaces are available, in practice they may not be.

6 CHAPTER 6 – RESULTS & CONCLUSION

6.1 Results

By combing an embedded system (PIC18F452) with a custom TCP/IP stack, I was able to design a device which is able to allow numerous attached devices to be controlled remotely via the ever ubiquities web-browser.

Furthermore, by adding support for XML, the device (TCP/IC) is able to respond to commands issued to it not only by other devices, but also by a central control system such as the one found in the Massey SmatHouse.

As a final step, by using the PICDem.net rapid-prototyping development board, it was possible to create a prototype of the TCP/IC. This prototype, although significantly larger than the device designed to fit within the Massey SmartHouse, allows for the practical demonstration of the unit as well as the overall SmartHouse concept.

6.1.1 The Dynamic Web-Interface

The image below (*Figure 6.1*) shows the TCP/IC's dynamic web interface discussed in *Chapter 5*. It should, however, only be considered as a placeholder used for demonstration purposes. Once the Massey SmartHouse project is ready for implementation, this page will become obsolete as the functions of the TCP/IC will be accessed via XML calls by made by the SmartHouse AI.

The web page consists of three content areas: MAIN, CONTROL and STATUS. The page is made up of a number of tables, two of which contain inline frames to show how the careful design of the interface allows the GUI to be easily moulded for any number of applications.

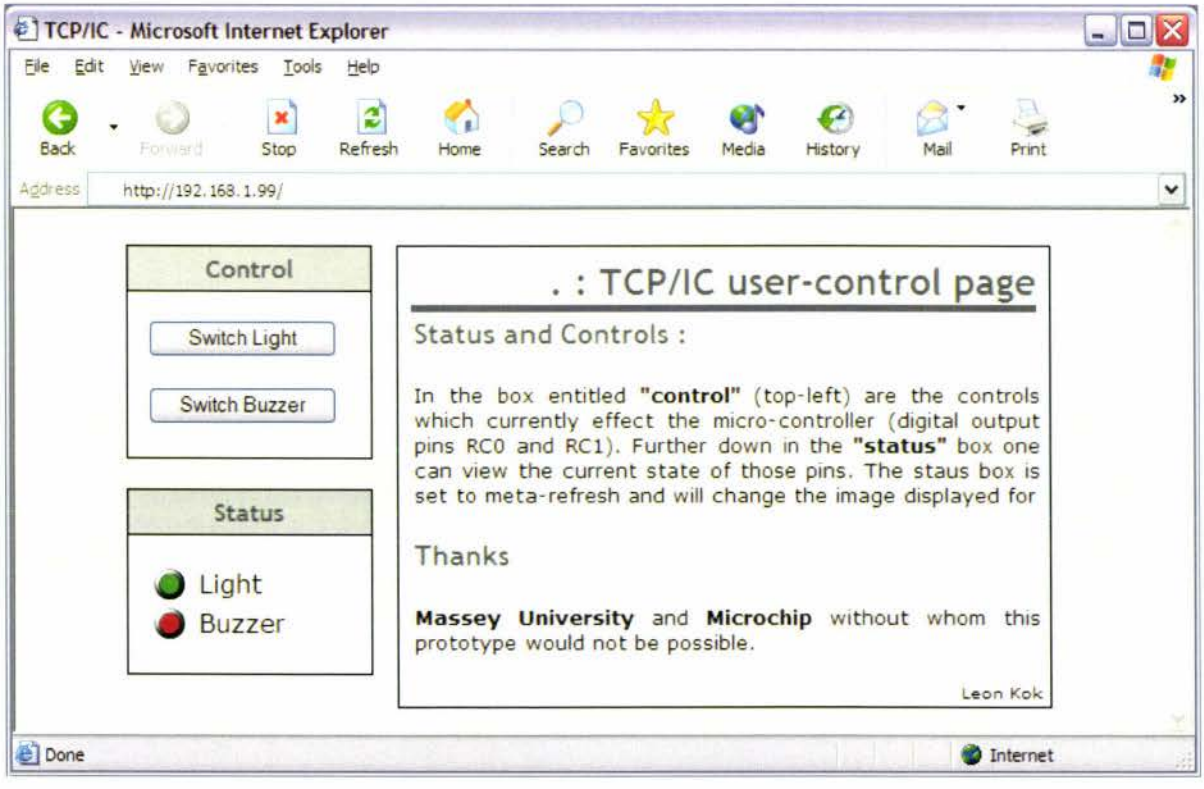


Figure 6.1: The TCP/IC HTTP Web-server

The Main Area

This is the primary content area and is currently being used for information relating to the demonstration of the TCP/IC – including some usage instructions. This part of the web page is static.

The Control Frame

This frame dynamically loads `commands.cgi`. It is a CGI page containing two BUTTON INPUTS that have been "mapped" to two of the TCP/ICs output ports. In this example, the buttons are used to turn on/off a LED and a buzzer.

The Status Frame

This frame loads `status.cgi`. It employs the variable substitution ("`%xx`") as discussed in *Chapter 5*. In the example, the ON/OFF status are shown by two different colour GIF files.

6.1.2 Additional file-types

As demonstrated by `status.cgi`, altering the HTTP MIME-type [45] allows the TCP/IC to serve any type of data. In accordance with the standard [13], when XML data is being sent, `text/xml` is used as the HTTP MIME type to allow the client to process it correctly.

I have also chosen to set aside some program memory for additional MIME-types. Added primarily to include rich multimedia content, it can also be used to transfer more abstract binary formats.

The figure below shows an example of a simple FLASH (`application/x-shockwave-flash`) animation served directly from the TCP/IC. It is intended to highlight the flexibility offered by using open standards such as HTTP.

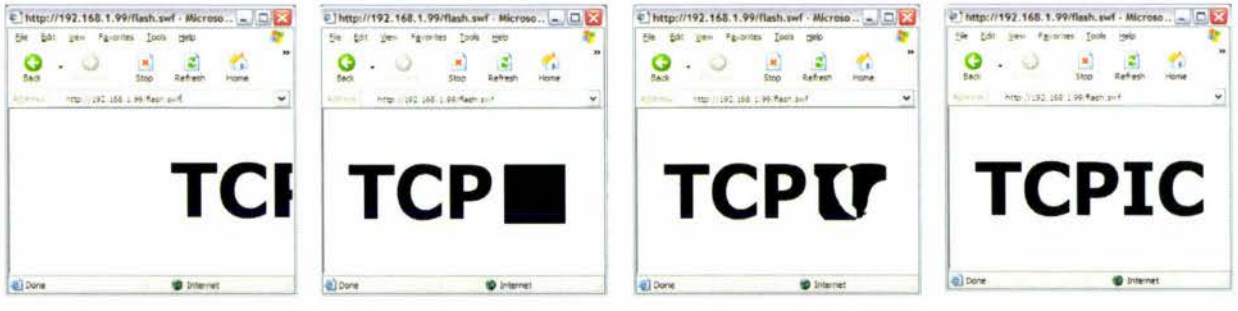


Figure 6.2: Macromedia FLASH Animation

6.2 Future Work

With the Massey SmartHouse project nearing its first stage of completion, the TCP/IC offers a way to implement the home-automation required by the project. As the SmartHouse moves closer to final completion, however, more and more practical ideas are being suggested and additional appliances added.

In its current state, although flexible, the TCP/IC may not be able to supply the needs of the next generation SmartHouse. To counter-act this, I suggest the following areas where future work could be carried out.

6.2.1 Efficient AC Switching

By modifying the way the TCP/IC interfaces with AC appliances, for example by adding thyristors to allow for full-wave power-control, future versions of the TCP/IC could more easily and efficiently manipulate AC appliances.

6.2.2 Wireless Ethernet

Although present in the SmartHouse, a wired Ethernet network may not always be available (e.g. Hazardous Environments). By adding a 802.11a/b/g bridge, or perhaps substituting the NIC with a 802.11a/b/g capable one, future versions of the TCP/IC would no longer rely on wired Ethernet and would allow a higher degree of portability.

6.2.3 Additional Interfaces

As the types of devices in the SmartHouse increase, new interfaces will be required to allow them to be integrated into the home-automation system. By using the available USART or perhaps an I2C connection, interfaces such as GSM/SMS, IrDA (Infrared) and Bluetooth could be added to provide this functionality.

6.2.4 Porting

The TCP/IC employs two main controllers: the PIC18F452 and the Realtek NIC. Future versions could be ported to a less expensive, feature-limited micro-controller such as the PIC16F877. Or perhaps it could be more tightly integrated into a FPGA architecture which would allow the NIC and the micro-controller to be combined. As the software written for the TCP/IC is almost exclusively written in ANSI C, a major code re-write would not be needed.

6.3 Conclusion

Most Embedded Systems have requirements that differ significantly in both detail and scope from the common desktop computer. Long life-cycles and cost sensitivity require more attention to optimization rather than simply maximizing the amount of computational throughput. Hardware/software co-design permits tradeoffs between hardware and software that are critical for more cost-effective Embedded Systems.

In this paper I have shown that the use of a rapid-prototype, based on a re-programmable device, can shorten the design time, while keeping the components used close to those used in the final product. The technology allows for real-time operation and this approach helps to fix errors early in the design cycle.

As a solution to home-automation, this paper describes the design and the implementation of a home-automation system, the TCP/IC, that utilizes the existing in-house Ethernet, to remotely monitor and control, in real-time, connected household appliances.

Based on the work of the *"Microchip TCP/IP stack"* along with the text "TCP/IP Lean", creating the TCP/IC has involved authoring a TCP/IP stack capable of running on the Microchip PIC-18 family of micro-controllers – despite the acute processing and memory limitations.

Although more complex than a serial interface, using Ethernet and TCP/IP; both XML and HTTP interfaces could be implemented thus ensuring the device would be able to be integrated into existing web-based automation systems.

7 REFERENCES

1. Wolf, W.H., *Hardware-software Co-design of* Embedded Systems. Proceedings of the IEEE, 1994. **82**(7): p. 967-989.

2. Wolf, W. and J. Madsen, Embedded Systems *Education for the Future.* Proceedings of the IEEE, 2000. **88**(1): p. 23-30.

3. Napper, S., *Embedded-system Design Plays Catch-up.* Computer, 1998. **31**(8): p. 120-118-19.

4. Cardelli, S., et al. *Rapid-prototyping of* Embedded Systems *via Reprogrammable Devices*. in *Rapid System Prototyping, 1996. Proceedings., Seventh IEEE International Workshop on*. 1996.

5. Koopman, P. *Embedded System Design Issues the Rest of the Story*. in *Computer Design: VLSI in Computers and Processors, 1996. ICCD '96. Proceedings., 1996 IEEE International Conference on*. 1996.

6. Ryan, J.L., *Home Automation.* Electronics & Communication Engineering Journal, 1989. **1**(4): p. 185-192.

7. Wacks, K., *Home Systems Standards: Achievements and Challenges.* Communications Magazine, IEEE, 2002. **40**(4): p. 152-159.

8. Badami, V.V. and N.W. Chbat, *Home Appliances Get Smart.* Spectrum, IEEE, 1998. **35**(8): p. 36-43.

9. Nunes, R. and J. Delgado. *An Architecture for a Home Automation System*. in *Electronics, Circuits and Systems, 1998 IEEE International Conference on*. 1998.

10. Guan, R., W. Pruehsner, and J.D. Enderle. *ST2000 Home Automation System for Persons with Disabilities*. in *Bioengineering Conference, 1999. Proceedings of the IEEE 25th Annual Northeast*. 1999.

11. *The Massey University SmartHouse Project*, (Retrieved June 25, 2005), [http://smarthouse.massey.ac.nz/].

12. Malinowski, A. *Networked Home Automation Projects*. in *Industrial Electronics Society, 2001. IECON '01. The 27th Annual Conference of the IEEE*. 2001.

13. *XML Media Types RFC*. 2001, (Retrieved June 25, 2005), [http://www.ietf.org/rfc/rfc3023.txt].

14. *The RS232 Standard - A tutorial with Signal Names and Definitions*, (Retrieved June 25, 2005), [http://www.camiresearch.com/Data_Com_Basics/RS232_standard.htm]

15. *Connectivity Knowledge Platform, IEEE 1284*, (Retrieved June 25, 2005), [http://ckp.made-it.com/ieee1284.html].

16. *IEEE 802.11 Wireless Local Area Networks*, (Retrieved June 25, 2005), [http://grouper.ieee.org/groups/802/11/].

17. *SitePlayer Embedded Ethernet Controller*, (Retrieved June 25, 2005), [http://www.siteplayer.com/].

18. *PICNIC: ISA Embedded Web-server*, (Retrieved June 25, 2005), [http://members.vol.at/home.floery/electronix/picnic/].

19. *Microchip PIC18F452 Home*, (Retrieved June 25, 2005), [http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1335&dDocName=en010296].

20. Stallings, W., *Reduced Instruction Set Computer Architecture.* Proceedings of the IEEE, 1988. **76**(1): p. 38-55.

21. *Ethereal: A Network Protocol Analyzer*, (Retrieved June 25, 2005), [http://www.ethereal.com/].

22. Bentham, J., *TCP/IP Lean, Second Edition*. 2002: CMP Media LLC.

23. Kuo, S., Z. Salcic, and U. Madawala. *A Real-time Hybrid Web-client Access Architecture for Home Automation*. in *Information, Communications and Signal Processing, 2003 and the Fourth Pacific Rim Conference on Multimedia. Proceedings of the 2003 Joint Conference of the Fourth International Conference on*. 2003.

24. *Ethernet Address Resolution Protocol RFC*. 1982, (Retrieved June 25, 2005), [http://www.ietf.org/rfc/rfc0826.txt].

25. *Internet Protocol RFC*. 1980, (Retrieved June 25, 2005), [http://www.ietf.org/rfc/rfc0760.txt].

26. *Internet Control Message Protocol RFC*. 1981, (Retrieved June 25, 2005), [http://www.ietf.org/rfc/rfc0792.txt].

27. *Transmission Control Protocol RFC*. 1981, (Retrieved June 25, 2005), [http://www.ietf.org/rfc/rfc0793.txt].

28. *User Datagram Protocol RFC*. 1980, (Retrieved June 25, 2005), [http://www.ietf.org/rfc/rfc0768.txt].

29. *Dynamic Host Configuration Protocol RFC*. 1997, (Retrieved June 25, 2005), [http://www.ietf.org/rfc/rfc2131.txt].

30. Tokunaga, E., et al. *A Framework for Connecting Home Computing Middleware*. in *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on*. 2002.

31. *X-10 Home Automation*, (Retrieved June 25, 2005), [http://www.x10.com]

32. *Institute of Electrical and Electronics Engineers (IEEE)*, (Retrieved June 25, 2005), [http://www.ieee.org/].

33. *The Internet Engineering Task Force*, (Retrieved June 25, 2005), [http://www.ietf.org/].

34. Corcoran, P.M. and J. Desbonnet, *Browser-style Interfaces to a Home Automation Network*. Consumer Electronics, IEEE Transactions on, 1997. **43**(4): p. 1063-1069.

35. Lloyd, B. and M. Susnik. *Web Embedded Field Devices*. in *Pulp and Paper Industry Technical Conference, 2002. Conference Record of the 2002 Annual*. 2002.

36. Eckel, C., G. Gaderer, and T. Sauter. *Implementation Requirements for Web-enabled Appliances - A Case Study*. in *Emerging Technologies and Factory Automation, 2003. Proceedings. ETFA '03. IEEE Conference*. 2003.

37. *Hypertext Transfer Protocol -- HTTP/1.0*. 1996, (Retrieved June 25, 2005), [http://www.ietf.org/rfc/rfc1945.txt].

38. *Hypertext Transfer Protocol -- HTTP/1.1*. 1999, (Retrieved June 25, 2005), [http://www.ietf.org/rfc/rfc2616.txt].

39. Corcoran, P.M., F. Papal, and A. Zoldi, *User Interface Technologies for Home Appliances and Networks.* Consumer Electronics, IEEE Transactions on, 1998. **44**(3): p. 679-685.

40. Pokorny, J. *XML functionally*. in *Database Engineering and Applications Symposium, 2000 International*. 2000.

41. *World Wide Web Consortium (W3C)*, (Retrieved June 25, 2005), [http://www.w3.org/].

42. Poon, P.M.S., T.S. Dillon, and E. Chang. *XML as a Basis for Interoperability in Real Time Distributed Systems*. in *Software Technologies for Future Embedded and Ubiquitous Systems, 2004. Proceedings. Second IEEE Workshop on*. 2004.

43. *HTML 4.01 Specification*. 1999, (Retrieved June 25, 2005), [http://www.w3.org/TR/REC-html40/].

44. Dissanaike, S., P. Wijkman, and M. Wijkman. *Utilizing XML-RPC or SOAP on an Embedded System*. in *Distributed Computing Systems Workshops, 2004. Proceedings. 24th International Conference on*. 2004.

45. *MIME Media Types*, (Retrieved June 25, 2005), [http://www.iana.org/assignments/media-types/].

ADDENDUM I

CGI Web Pages

STATUS.CGI

```
<html>
<head>
<meta http-equiv="Refresh" content="3">
<title>cgi-status-page</title>
<style type="text/css">
body{
      font-family: Verdana, "Lucida Sans", Arial, Geneva,
      Helvetica, Helv, "Myriad Web", Syntax, sans-serif;
      font-size: 80%;
}
</style>
</head>
<body>
<table width="100" border="0" cellpadding="2"cellspacing="1">
  <tr>
    <td><img src="status_%08.gif" width="20" height="20"></td>
    <td>Light</td>
  </tr>
  <tr>
    <td><img src="status_%09.gif" width="20" height="20"></td>
    <td>Buzzer</td>
  </tr>
</table></body></html>
```

COMMANDS.CGI

```
<html>
<head>
<title>cgi-commands-page</title>
</head>
<body>
<form method="get" action=0>
    <input type=submit name=8 value="  Switch Light  ">
    <br>
    <input type=submit name=9 value="Switch Buzzer">
</form>
</body>
</html>
```

ADDENDUM II

'C' Source Code

```
#define VAR_OUTPUT_RC0        (8)
#define VAR_OUTPUT_RC1        (9)


/*****************************************************************
 * Function:        WORD HTTPGetVar(BYTE var, WORD ref, BYTE* val)
 *
 *****************************************************************/

WORD HTTPGetVar(BYTE var, WORD ref, BYTE* val)
{
    switch(var)
    {
    case VAR_OUTPUT_RC0:
        if ( LATCbits.LATC0 )
            *val = '0';
        else
            *val = '1';
        break;

    case VAR_OUTPUT_RC1:
        if ( LATCbits.LATC1 )
            *val = '0';
        else
            *val = '1';
        break;
    }

    return HTTP_END_OF_VAR;
}
```

```
/*****************************************************************
 * Function:        void HTTPExecCmd(BYTE** argv, BYTE argc)
 *
 *                  Format of argv:
 *                  If HTTP action was : thank.htm?name=Joe&age=25
 *                      argv[0] => thank.htm
 *                      argv[1] => name
 *                      argv[2] => Joe
 *                      argv[3] => age
 *                      argv[4] => 25
 *****************************************************************/

ROM char COMMANDS_OK_PAGE[] = "COMMANDS.CGI";
ROM char CMD_UNKNOWN_PAGE[] = "INDEX.HTM";

void HTTPExecCmd(BYTE** argv, BYTE argc)
{
    BYTE command;
    BYTE var;

    command = argv[0][0] - '0';

    switch(command)
    {
    case CGI_CMD_DIGOUT:

        var = argv[1][0] - '0';

        switch(var)
        {
        case VAR_OUTPUT_RC0:
            // This is "RC0" - Toggle it
            LATCbits.LATC0 ^= 1;
            LATA3 ^= 1; // Also toggle other LED to indicate change !
            break;

        case VAR_OUTPUT_RC1:
            // This is "RC1" - Toggle it
            LATCbits.LATC1 ^= 1;
            LATA3 ^= 1; // Also toggle other LED to indicate change !
            break;
         }
         memcpypgm2ram(argv[0],
            (const ROM char*)COMMANDS_OK_PAGE, COMMANDS_OK_PAGE_LEN);
         break;

         default:
        memcpypgm2ram((unsigned char*)argv[0],
            (const ROM char*)CMD_UNKNOWN_PAGE, CMD_UNKNOWN_PAGE_LEN);
        break;
    }

}
```

```
001     /*********************************************************************
002      *                  ARP Module
003      *********************************************************************/
004     #include <string.h>
005
006     #include "stacktsk.h"
007     #include "helpers.h"
008     #include "arp.h"
009     #include "mac.h"
010
011     /*
012      * ARP Operation codes.
013      */
014     #define ARP_OPERATION_REQ       0x01
015     #define ARP_OPERATION_RESP      0x02
016
017     /*
018      * ETHERNET packet type as defined by IEEE 802.3
019      */
020     #define HW_ETHERNET             (0x0001)
021     #define ARP_IP                  (0x0800)
022
023
024     /*
025      * ARP packet
026      */
027     typedef struct _ARP_PACKET
028     {
029         WORD        HardwareType;
030         WORD        Protocol;
031         BYTE        MACAddrLen;
032         BYTE        ProtocolLen;
033         WORD        Operation;
034         MAC_ADDR    SenderMACAddr;
035         IP_ADDR     SenderIPAddr;
036         MAC_ADDR    TargetMACAddr;
037         IP_ADDR     TargetIPAddr;
038     } ARP_PACKET;
039
040     static void SwapARPPacket(ARP_PACKET *p);
041
042     /*********************************************************************
043      * Function:        BOOL ARPGet(NODE_INFO* remote, BYTE* opCode)
044      *
045      * PreCondition:    ARP packet is ready in MAC buffer.
046      *
047      * Input:           remote  - Remote node info
048      *                  opCode  - Buffer to hold ARP op code.
049      *
050      * Output:          TRUE if a valid ARP packet was received.
051      *                  FALSE otherwise.
052      *
053      *********************************************************************/
054     BOOL ARPGet(NODE_INFO *remote, BYTE *opCode)
055     {
056         ARP_PACKET packet;
057
058         MACGetArray((BYTE*)&packet, sizeof(packet));
059
060         MACDiscardRx();
061
062         SwapARPPacket(&packet);
063
064         if ( packet.HardwareType != HW_ETHERNET     ||
065              packet.MACAddrLen != sizeof(MAC_ADDR)  ||
066              packet.ProtocolLen != sizeof(IP_ADDR) )
067             return FALSE;
068
069         if ( packet.Operation == ARP_OPERATION_RESP )
070             *opCode = ARP_REPLY;
071         else if ( packet.Operation == ARP_OPERATION_REQ )
072             *opCode = ARP_REQUEST;
073         else
074         {
075             *opCode = ARP_UNKNOWN;
076             return FALSE;
077         }
078
079         if ( (packet.TargetIPAddr.v[0] == MY_IP_BYTE1) &&
080              (packet.TargetIPAddr.v[1] == MY_IP_BYTE2) &&
081              (packet.TargetIPAddr.v[2] == MY_IP_BYTE3) &&
082              (packet.TargetIPAddr.v[3] == MY_IP_BYTE4) )
083         {
084             remote->MACAddr     = packet.SenderMACAddr;
```

```
085             remote->IPAddr      = packet.SenderIPAddr;
086             return TRUE;
087         }
088         else
089             return FALSE;
090     }
091
092
093     /*****************************************************************
094      * Function:        void ARPPut(NODE_INFO* more, BYTE opCode)
095      *
096      * PreCondition:    MACIsTxReady() == TRUE
097      *
098      * Input:           remote  - Remote node info
099      *                  opCode  - ARP op code to send
100      *
101      * Output:          None
102      *
103      *****************************************************************/
104     void ARPPut(NODE_INFO *remote,
105                 BYTE opCode)
106     {
107         ARP_PACKET packet;
108
109         packet.HardwareType             = HW_ETHERNET;
110         packet.Protocol                 = ARP_IP;
111         packet.MACAddrLen               = sizeof(MAC_ADDR);
112         packet.ProtocolLen              = sizeof(IP_ADDR);
113
114         if ( opCode == ARP_REQUEST )
115         {
116             packet.Operation            = ARP_OPERATION_REQ;
117             packet.TargetMACAddr.v[0]   = 0xff;
118             packet.TargetMACAddr.v[1]   = 0xff;
119             packet.TargetMACAddr.v[2]   = 0xff;
120             packet.TargetMACAddr.v[3]   = 0xff;
121             packet.TargetMACAddr.v[4]   = 0xff;
122             packet.TargetMACAddr.v[5]   = 0xff;
123         }
124         else
125         {
126             packet.Operation            = ARP_OPERATION_RESP;
127             packet.TargetMACAddr        = remote->MACAddr;
128         }
129
130         packet.SenderMACAddr.v[0]       = MY_MAC_BYTE1;
131         packet.SenderMACAddr.v[1]       = MY_MAC_BYTE2;
132         packet.SenderMACAddr.v[2]       = MY_MAC_BYTE3;
133         packet.SenderMACAddr.v[3]       = MY_MAC_BYTE4;
134         packet.SenderMACAddr.v[4]       = MY_MAC_BYTE5;
135         packet.SenderMACAddr.v[5]       = MY_MAC_BYTE6;
136
137         packet.SenderIPAddr.v[0]        = MY_IP_BYTE1;
138         packet.SenderIPAddr.v[1]        = MY_IP_BYTE2;
139         packet.SenderIPAddr.v[2]        = MY_IP_BYTE3;
140         packet.SenderIPAddr.v[3]        = MY_IP_BYTE4;
141
142
143     /*
144          * Check to see if target is on same subnet, if not, find
145          * Gateway MAC.
146          * Once we get Gateway MAC, all access to remote host will
147          * go through Gateway.
148      */
149         if (((packet.SenderIPAddr.v[0] ^ remote->IPAddr.v[0]) &
150                                             MY_MASK_BYTE1) ||
151             ((packet.SenderIPAddr.v[1] ^ remote->IPAddr.v[1]) &
152                                             MY_MASK_BYTE2) ||
153             ((packet.SenderIPAddr.v[2] ^ remote->IPAddr.v[2]) &
154                                             MY_MASK_BYTE3) ||
155             ((packet.SenderIPAddr.v[3] ^ remote->IPAddr.v[3]) &
156                                             MY_MASK_BYTE4) )
157         {
158             packet.TargetIPAddr.v[0] = MY_GATE_BYTE1;
159             packet.TargetIPAddr.v[1] = MY_GATE_BYTE2;
160             packet.TargetIPAddr.v[2] = MY_GATE_BYTE3;
161             packet.TargetIPAddr.v[3] = MY_GATE_BYTE4;
162         }
163         else
164             packet.TargetIPAddr             = remote->IPAddr;
165
166         SwapARPPacket(&packet);
167
168         MACPutHeader(&packet.TargetMACAddr, MAC_ARP, sizeof(packet));
```

```
169
170         MACPutArray((BYTE*)&packet, sizeof(packet));
171
172         MACFlush();
173     }
174
175
176     /*****************************************************************
177      * Function:        static void SwapARPPacket(ARP_PACKET* p)
178      *
179      * PreCondition:    None
180      *
181      * Input:           p   - ARP packet to be swapped.
182      *
183      * Output:          None
184      *
185      *****************************************************************/
186     static void SwapARPPacket(ARP_PACKET *p)
187     {
188         p->HardwareType     = swaps(p->HardwareType);
189         p->Protocol         = swaps(p->Protocol);
190         p->Operation        = swaps(p->Operation);
191     }
192
```

```
001     /*****************************************************************************
002      *                  ICMP Module
003      *****************************************************************************/
004
005     #include <string.h>
006     #include "stacktsk.h"
007     #include "helpers.h"
008     #include "icmp.h"
009     #include "ip.h"
010     #include "mac.h"
011
012     #if !defined(STACK_USE_ICMP)
013     #error You have selected to not include ICMP.  Remove this file from \
014           your project to reduce code size.
015     #endif
016
017
018     #define MAX_ICMP_DATA       (32)
019
020     /*
021      * ICMP packet definition
022      */
023     typedef struct _ICMP_PACKET
024     {
025         BYTE    Type;
026         BYTE    Code;
027         WORD    Checksum;
028         WORD    Identifier;
029         WORD    SequenceNumber;
030         BYTE    Data[MAX_ICMP_DATA];
031     } ICMP_PACKET;
032     #define ICMP_HEADER_SIZE    (sizeof(ICMP_PACKET) - MAX_ICMP_DATA)
033
034     static void SwapICMPPacket(ICMP_PACKET* p);
035
036
037     /*****************************************************************************
038      * Function:        BOOL ICMPGet(ICMP_CODE *code,
039      *                               BYTE *data,
040      *                               BYTE *len,
041      *                               WORD *id,
042      *                               WORD *seq)
043      *
044      * PreCondition:    MAC buffer contains ICMP type packet.
045      *
046      * Input:           code    - Buffer to hold ICMP code value
047      *                  data    - Buffer to hold ICMP data
048      *                  len     - Buffer to hold ICMP data length
049      *                  id      - Buffer to hold ICMP id
050      *                  seq     - Buffer to hold ICMP seq
051      *
052      * Output:          TRUE if valid ICMP packet was received
053      *                  FALSE otherwise.
054      *
055      *****************************************************************************/
056     BOOL ICMPGet(ICMP_CODE *code,
057                  BYTE *data,
058                  BYTE *len,
059                  WORD *id,
060                  WORD *seq)
061     {
062         ICMP_PACKET packet;
063         WORD checksums[2];
064         WORD CalcChecksum;
065         WORD ReceivedChecksum;
066
067         MACGetArray((BYTE*)&packet, ICMP_HEADER_SIZE);
068
069         ReceivedChecksum = packet.Checksum;
070         packet.Checksum = 0;
071
072         checksums[0] = ~CalcIPChecksum((BYTE*)&packet, ICMP_HEADER_SIZE);
073
074         *len -= ICMP_HEADER_SIZE;
075
076         MACGetArray(data, *len);
077         checksums[1] = ~CalcIPChecksum(data, *len);
078
079         CalcChecksum = CalcIPChecksum((BYTE*)checksums, 2 * sizeof(WORD));
080
081         SwapICMPPacket(&packet);
082
083         *code = packet.Type;
084         *id = packet.Identifier;
```

```
085         *seq = packet.SequenceNumber;
086
087         return ( CalcChecksum == ReceivedChecksum );
088     }
089
090     /*********************************************************************
091      * Function:        void ICMPPut(NODE_INFO *remote,
092      *                               ICMP_CODE code,
093      *                               BYTE *data,
094      *                               BYTE len,
095      *                               WORD id,
096      *                               WORD seq)
097      *
098      * PreCondition:    ICMPIsTxReady() == TRUE
099      *
100      * Input:           remote      - Remote node info
101      *                  code        - ICMP_ECHO_REPLY or ICMP_ECHO_REQUEST
102      *                  data        - Data bytes
103      *                  len         - Number of bytes to send
104      *                  id          - ICMP identifier
105      *                  seq         - ICMP sequence number
106      *
107      * Output:          None
108      *
109      ********************************************************************/
110     void ICMPPut(NODE_INFO *remote,
111                  ICMP_CODE code,
112                  BYTE *data,
113                  BYTE len,
114                  WORD id,
115                  WORD seq)
116     {
117         ICMP_PACKET packet;
118
119         packet.Code             = 0;
120         packet.Type             = code;
121         packet.Checksum         = 0;
122         packet.Identifier       = id;
123         packet.SequenceNumber   = seq;
124
125         memcpy((void*)packet.Data, (void*)data, len);
126
127         SwapICMPPacket(&packet);
128
129         packet.Checksum         = CalcIPChecksum((BYTE*)&packet,
130                                                  (WORD)(ICMP_HEADER_SIZE + len));
131
132         IPPutHeader(remote,
133                     IP_PROT_ICMP,
134                     (WORD)(ICMP_HEADER_SIZE + len));
135
136         IPPutArray((BYTE*)&packet, (WORD)(ICMP_HEADER_SIZE + len));
137
138         MACFlush();
139     }
140
141     /*********************************************************************
142      * Function:        void SwapICMPPacket(ICMP_PACKET* p)
143      *
144      * PreCondition:    None
145      *
146      * Input:           p - ICMP packet header
147      *
148      * Output:          ICMP packet is swapped
149      *
150      ********************************************************************/
151     static void SwapICMPPacket(ICMP_PACKET* p)
152     {
153         p->Identifier           = swaps(p->Identifier);
154         p->SequenceNumber       = swaps(p->SequenceNumber);
155         p->Checksum             = swaps(p->Checksum);
156     }
157
```

```
001 /*********************************************************************
002  *                  PIC IP Module
003  *********************************************************************/
004
005 #include <string.h>
006
007 #include "stacktsk.h"
008 #include "helpers.h"
009 #include "mac.h"
010 #include "ip.h"
011
012 // This is left shifted by 4.  Actual value is 0x04.
013 #define IPv4                    (0x40)
014 #define IP_VERSION              IPv4
015
016 #define IP_IHL                  (0x05)
017
018 #define IP_SERVICE_NW_CTRL      (0x07)
019 #define IP_SERVICE_IN_CTRL      (0x06)
020 #define IP_SERVICE_ECP          (0x05)
021 #define IP_SERVICE_OVR          (0x04)
022 #define IP_SERVICE_FLASH        (0x03)
023 #define IP_SERVICE_IMM          (0x02)
024 #define IP_SERVICE_PRIOR        (0x01)
025 #define IP_SERVICE_ROUTINE      (0x00)
026
027 #define IP_SERVICE_N_DELAY      (0x00)
028 #define IP_SERCICE_L_DELAY      (0x08)
029 #define IP_SERVICE_N_THRPT      (0x00)
030 #define IP_SERVICE_H_THRPT      (0x10)
031 #define IP_SERVICE_N_RELIB      (0x00)
032 #define IP_SERVICE_H_RELIB      (0x20)
033
034 #define IP_SERVICE              (IP_SERVICE_ROUTINE | IP_SERVICE_N_DELAY)
035
036 #define MY_IP_TTL               (100)    /* Time-To-Live in Seconds */
037
038
039 static WORD _Identifier = 0;
040 static void SwapIPHeader(IP_HEADER* h);
041
042 /*********************************************************************
043  * Function:        BOOL IPGetHeader( IP_ADDR    *localIP,
044  *                                    NODE_INFO  *remote,
045  *                                    BYTE        *Protocol,
046  *                                    WORD        *len)
047  *
048  * PreCondition:    MACGetHeader() == TRUE
049  *
050  * Input:           localIP     - Local node IP Address as received
051  *                                in current IP header.
052  *                                If this information is not required
053  *                                caller may pass NULL value.
054  *                  remote      - Remote node info
055  *                  Protocol    - Current packet protocol
056  *                  len         - Current packet data length
057  *
058  * Output:          TRUE, if valid packet was received
059  *                        FALSE otherwise
060 *
061  ********************************************************************/
062 BOOL IPGetHeader(IP_ADDR *localIP,
063                  NODE_INFO *remote,
064                  BYTE *protocol,
065                  WORD *len)
066 {
067     WORD_VAL    ReceivedChecksum;
068     WORD_VAL    CalcChecksum;
069     WORD        checksums[2];
070     IP_HEADER   header;
071     BYTE        optionsLen;
072 #define MAX_OPTIONS_LEN     (20)            // As per RFC 791.
073     BYTE        options[MAX_OPTIONS_LEN];
074
075     // Read IP header.
076     MACGetArray((BYTE*)&header, sizeof(header));
077
078     // Make sure that this IPv4 packet.
079     if ( (header.VersionIHL & 0xf0) != IP_VERSION )
080     {
081         goto IPGetHeader_Discard;
082     }
083
084     optionsLen = ((header.VersionIHL & 0x0f) << 2) - sizeof(header);
```

```
085
086         if ( optionsLen > MAX_OPTIONS_LEN )
087         {
088             goto IPGetHeader_Discard;
089         }
090
091         if ( optionsLen > 0 )
092             MACGetArray(options, optionsLen);
093
094         // Save header checksum; clear it and recalculate it ourselves.
095         ReceivedChecksum.Val = header.HeaderChecksum;
096         header.HeaderChecksum = 0;
097
098         // Calculate checksum of header including options bytes.
099         checksums[0] = ~CalcIPChecksum((BYTE*)&header, sizeof(header));
100
101         // Calculate Options checksum too, if they are present.
102         if ( optionsLen > 0 )
103             checksums[1] = ~CalcIPChecksum((BYTE*)options, optionsLen);
104         else
105             checksums[1] = 0;
106
107         CalcChecksum.Val  = CalcIPChecksum((BYTE*)checksums,
108                                            2 * sizeof(WORD));
109         // Network to host conversion.
110         SwapIPHeader(&header);
111
112         // Make sure that checksum is correct and IP version is supported.
113         if ( ReceivedChecksum.Val != CalcChecksum.Val   ||
114              (header.VersionIHL & 0xf0) != IP_VERSION )
115
116         {
117             // Bad/Unknown packet. Discard it.
118             goto IPGetHeader_Discard;
119         }
120
121         if ( localIP )
122             localIP->Val    = header.DestAddress.Val;
123
124         remote->IPAddr.Val  = header.SourceAddress.Val;
125         *protocol           = header.Protocol;
126         *len                = header.TotalLength - optionsLen -
127                               sizeof(header);
128
129         return TRUE;
130
131     IPGetHeader_Discard:
132         MACDiscardRx();
133         return FALSE;
134
135     }
136
137     /*****************************************************************************
138      * Function:        WORD IPPutHeader(   IP_ADDR *Dest,
139      *                                      BYTE    Protocol,
140      *                                      WORD    Identifier,
141      *                                      WORD    DataLen)
142      *
143      * PreCondition:    IPIsTxReady() == TRUE
144      *
145      * Input:           Src         - Destination node address
146      *                  Protocol    - Current packet protocol
147      *                  Identifier  - Current packet identifier
148      *                  DataLen     - Current packet data length
149      *
150      * Output:          Handle to current packet - For use by
151      *                  IPSendByte() function.
152      *
153      *****************************************************************************/
154     WORD IPPutHeader(NODE_INFO *remote,
155                      BYTE protocol,
156                      WORD    len)
157     {
158         IP_HEADER   header;
159
160         header.VersionIHL       = IP_VERSION | IP_IHL;
161         header.TypeOfService    = IP_SERVICE;
162         header.TotalLength      = sizeof(header) + len;
163         header.Identification   = ++_Identifier;
164         header.FragmentInfo     = 0;
165         header.TimeToLive       = MY_IP_TTL;
166         header.Protocol         = protocol;
167         header.HeaderChecksum   = 0;
168         header.SourceAddress.v[0] = MY_IP_BYTE1;
```

```
169         header.SourceAddress.v[1] = MY_IP_BYTE2;
170         header.SourceAddress.v[2] = MY_IP_BYTE3;
171         header.SourceAddress.v[3] = MY_IP_BYTE4;
172
173         header.DestAddress.Val = remote->IPAddr.Val;
174         SwapIPHeader(&header);
175
176         header.HeaderChecksum    = CalcIPChecksum((BYTE*)&header,
177                                                   sizeof(header));
178
179         MACPutHeader(&remote->MACAddr, MAC_IP, (sizeof(header)+len));
180         MACPutArray((BYTE*)&header, sizeof(header));
181
182         return 0x0;
183
184     }
185
186     static void SwapIPHeader(IP_HEADER* h)
187     {
188         h->TotalLength      = swaps(h->TotalLength);
189         h->Identification   = swaps(h->Identification);
190         h->HeaderChecksum   = swaps(h->HeaderChecksum);
191     }
192
```

```
0001  /*****************************************************************************
0002   *                      TCP Module
0003   *****************************************************************************/
0004  #define THIS_IS_TCP
0005
0006  #include <string.h>
0007
0008  #include "stacktsk.h"
0009  #include "helpers.h"
0010  #include "ip.h"
0011  #include "mac.h"
0012  #include "tick.h"
0013  #include "tcp.h"
0014
0015
0016  #define MAX_TCP_DATA_LEN    (MAC_TX_BUFFER_SIZE - 54)
0017  #define TCP_START_TIMEOUT_VAL   ((TICK)TICK_SECOND * (TICK)60)
0018
0019  /*
0020   * TCP Flags as defined by rfc793
0021   */
0022  #define FIN     (0x01)
0023  #define SYN     (0x02)
0024  #define RST     (0x04)
0025  #define PSH     (0x08)
0026  #define ACK     (0x10)
0027  #define URG     (0x20)
0028
0029  /*
0030   * TCP Header def. as per rfc 793.
0031   */
0032
0033  typedef struct _TCP_HEADER
0034  {
0035      WORD    SourcePort;
0036      WORD    DestPort;
0037      DWORD   SeqNumber;
0038      DWORD   AckNumber;
0039
0040      struct
0041      {
0042          unsigned int Reserved3      : 4;
0043          unsigned int Val            : 4;
0044      } DataOffset;
0045
0046
0047      union
0048      {
0049          struct
0050          {
0051              unsigned int flagFIN    : 1;
0052              unsigned int flagSYN    : 1;
0053              unsigned int flagRST    : 1;
0054              unsigned int flagPSH    : 1;
0055              unsigned int flagACK    : 1;
0056              unsigned int flagURG    : 1;
0057              unsigned int Reserved2  : 2;
0058          } bits;
0059          BYTE byte;
0060      } Flags;
0061
0062      WORD    Window;
0063      WORD    Checksum;
0064      WORD    UrgentPointer;
0065
0066  } TCP_HEADER;
0067
0068  /*
0069   * TCP Options as defined by rfc 793
0070   */
0071  #define TCP_OPTIONS_END_OF_LIST     (0x00)
0072  #define TCP_OPTIONS_NO_OP           (0x01)
0073  #define TCP_OPTIONS_MAX_SEG_SIZE    (0x02)
0074  typedef struct _TCP_OPTIONS
0075  {
0076      BYTE        Kind;
0077      BYTE        Length;
0078      WORD_VAL    MaxSegSize;
0079  } TCP_OPTIONS;
0080
0081  #define SwapPseudoTCPHeader(h)  (h.TCPLength = swaps(h.TCPLength))
0082
0083  /*
0084   * Pseudo header as defined by rfc 793.
```

```
0085     */
0086    typedef struct _PSEUDO_HEADER
0087    {
0088        IP_ADDR SourceAddress;
0089        IP_ADDR DestAddress;
0090        BYTE Zero;
0091        BYTE Protocol;
0092        WORD TCPLength;
0093    } PSEUDO_HEADER;
0094
0095    /*
0096     * These are all sockets supported by this TCP.
0097     */
0098    SOCKET_INFO TCB[MAX_SOCKETS];
0099
0100    /*
0101     * Local temp port numbers.
0102     */
0103    static WORD _NextPort;
0104
0105    /*
0106     * Starting segment sequence number for each new connection.
0107     */
0108    static DWORD ISS;
0109    static  void    HandleTCPSeg(TCP_SOCKET s,
0110                                 NODE_INFO *remote,
0111                                 TCP_HEADER *h,
0112                                 WORD len);
0113
0114    static void TransmitTCP(NODE_INFO *remote,
0115                            TCP_PORT localPort,
0116                            TCP_PORT remotePort,
0117                            DWORD seq,
0118                            DWORD ack,
0119                            BYTE flags,
0120                            BUFFER buffer,
0121                            WORD len);
0122
0123    static  TCP_SOCKET FindMatchingSocket(TCP_HEADER *h,
0124                                          NODE_INFO *remote);
0125    static  void    SwapTCPHeader(TCP_HEADER* header);
0126    static WORD CalcTCPChecksum(WORD len);
0127    static void CloseSocket(SOCKET_INFO* ps);
0128
0129    #define SendTCP(remote, localPort, remotePort, seq, ack, flags)     \
0130            TransmitTCP(remote, localPort, remotePort, seq, ack, flags, \
0131                        INVALID_BUFFER, 0)
0132
0133    #define LOCAL_PORT_START_NUMBER (1024)
0134    #define LOCAL_PORT_END_NUMBER   (5000)
0135
0136    /*****************************************************************
0137     * Function:         void TCPInit(void)
0138     *
0139     * PreCondition:     None
0140     *
0141     * Input:            None
0142     *
0143     * Output:           TCP is initialized.
0144     *
0145     *****************************************************************/
0146    void TCPInit(void)
0147    {
0148        TCP_SOCKET s;
0149        SOCKET_INFO* ps;
0150
0151
0152        // Initialize all sockets.
0153        for ( s = 0; s < MAX_SOCKETS; s++ )
0154        {
0155            ps = &TCB[s];
0156
0157            ps->smState             = TCP_CLOSED;
0158            ps->Flags.bServer       = FALSE;
0159            ps->Flags.bIsPutReady   = TRUE;
0160            ps->Flags.bFirstRead    = TRUE;
0161            ps->Flags.bIsTxInProgress = FALSE;
0162            ps->Flags.bIsGetReady   = FALSE;
0163            ps->TxBuffer            = INVALID_BUFFER;
0164            ps->TimeOut             = TCP_START_TIMEOUT_VAL;
0165        }
0166
0167        _NextPort = LOCAL_PORT_START_NUMBER;
0168        ISS = 0;
```

```
0169    }
0170
0171    /*****************************************************************************
0172     * Function:        TCP_SOCKET TCPListen(TCP_PORT port)
0173     *
0174     * PreCondition:    TCPInit() is already called.
0175     *
0176     * Input:           port    - A TCP port to be opened.
0177     *
0178     * Output:          Given port is opened and returned on success
0179     *                  INVALID_SOCKET if no more sockets left.
0180     *
0181     *****************************************************************************/
0182    TCP_SOCKET TCPListen(TCP_PORT port)
0183    {
0184        TCP_SOCKET s;
0185        SOCKET_INFO* ps;
0186
0187        for ( s = 0; s < MAX_SOCKETS; s++ )
0188        {
0189            ps = &TCB[s];
0190
0191            if ( ps->smState == TCP_CLOSED )
0192            {
0193                ps->smState             = TCP_LISTEN;
0194                ps->localPort           = port;
0195                ps->remotePort          = 0;
0196                ps->remote.IPAddr.Val   = 0x00;
0197                ps->Flags.bServer       = TRUE;
0198                ps->Flags.bIsGetReady   = FALSE;
0199                ps->TxBuffer            = INVALID_BUFFER;
0200                ps->Flags.bIsPutReady   = TRUE;
0201
0202                return s;
0203            }
0204        }
0205        return INVALID_SOCKET;
0206    }
0207
0208
0209
0210    /*****************************************************************************
0211     * Function:        TCP_SOCKET TCPConnect(NODE_INFO* remote,
0212     *                                        TCP_PORT remotePort)
0213     *
0214     * PreCondition:    TCPInit() is already called.
0215     *
0216     * Input:           remote      - Remote node address info
0217     *                  remotePort  - remote port to be connected.
0218     *
0219     * Output:          A new socket is created, connection request is
0220     *                  sent and socket handle is returned.
0221     *
0222     *****************************************************************************/
0223    #ifdef STACK_CLIENT_MODE
0224    TCP_SOCKET TCPConnect(NODE_INFO *remote, TCP_PORT remotePort)
0225    {
0226        TCP_SOCKET s;
0227        SOCKET_INFO* ps;
0228        BOOL lbFound = FALSE;
0229
0230      /*
0231         * Find an available socket
0232         */
0233        for ( s = 0; s < MAX_SOCKETS; s++ )
0234        {
0235            ps = &TCB[s];
0236            if ( ps->smState == TCP_CLOSED )
0237            {
0238                lbFound = TRUE;
0239                break;
0240            }
0241        }
0242
0243      /*
0244         * If there is no socket available, return error.
0245         */
0246        if ( lbFound == FALSE )
0247            return INVALID_SOCKET;
0248
0249      /*
0250         * Each new socket that is opened by this node, gets
0251         * next sequential port number.
0252         */
```

```
0253        ps->localPort = ++_NextPort;
0254        if ( _NextPort > LOCAL_PORT_END_NUMBER )
0255            _NextPort = LOCAL_PORT_START_NUMBER;
0256
0257        ps->Flags.bServer = FALSE;
0258        ps->remotePort = remotePort;
0259
0260        ps->SND_SEQ = ++ISS;
0261        ps->SND_ACK = 0;
0262
0263        memcpy((BYTE*)&ps->remote, (BYTE*)remote, sizeof(ps->remote));
0264
0265        /*
0266         * Send SYN message.
0267         */
0268        SendTCP(&ps->remote,
0269                ps->localPort,
0270                ps->remotePort,
0271                ps->SND_SEQ,
0272                ps->SND_ACK,
0273                SYN);
0274
0275        ps->smState = TCP_SYN_SENT;
0276        ps->SND_SEQ++;
0277
0278        return s;
0279    }
0280    #endif
0281
0282
0283
0284    /*****************************************************************
0285     * Function:        BOOL TCPIsConnected(TCP_SOCKET s)
0286     *
0287     * PreCondition:    TCPInit() is already called.
0288     *
0289     * Input:           s       - Socket to be checked for connection.
0290     *
0291     * Output:          TRUE    if given socket is connected
0292     *                  FALSE   if given socket is not connected.
0293     *
0294     *****************************************************************/
0295    BOOL TCPIsConnected(TCP_SOCKET s)
0296    {
0297        return ( TCB[s].smState == TCP_EST );
0298    }
0299
0300    /*****************************************************************
0301     * Function:        void TCPDisconnect(TCP_SOCKET s)
0302     *
0303     * PreCondition:    TCPInit() is already called     AND
0304     *                  TCPIsPutReady(s) == TRUE
0305     *
0306     * Input:           s       - Socket to be disconnected.
0307     *
0308     * Output:          A disconnect request is sent for given socket.
0309     *
0310     *****************************************************************/
0311    void TCPDisconnect(TCP_SOCKET s)
0312    {
0313        SOCKET_INFO *ps;
0314
0315        ps = &TCB[s];
0316
0317        if ( ps->smState != TCP_EST )
0318        {
0319            CloseSocket(ps);
0320            return;
0321        }
0322
0323        TCPDiscard(s);
0324
0325        /*
0326         * Send FIN message.
0327         */
0328        SendTCP(&ps->remote,
0329                ps->localPort,
0330                ps->remotePort,
0331                ps->SND_SEQ,
0332                ps->SND_ACK,
0333                FIN | ACK);
0334
0335            ps->SND_SEQ++;
0336
```

```
0337        ps->smState = TCP_FIN_WAIT_1;
0338
0339        return;
0340    }
0341
0342    /*****************************************************************
0343     * Function:        BOOL TCPFlush(TCP_SOCKET s)
0344     *
0345     * PreCondition:    TCPInit() is already called.
0346     *
0347     * Input:           s       - Socket whose data is to be transmitted.
0348     *
0349     * Output:          All and any data associated with this socket
0350     *                  is marked as ready for transmission.
0351     *
0352     *****************************************************************/
0353    WORD tempTxCount;
0354    DWORD tempSND_SEQ;
0355    BOOL TCPFlush(TCP_SOCKET s)
0356    {
0357        SOCKET_INFO *ps;
0358
0359        ps = &TCB[s];
0360
0361        // Make sure that this already a TxBuffer assigned to this
0362        // socket.
0363        if ( ps->TxBuffer == INVALID_BUFFER )
0364            return FALSE;
0365
0366        if ( ps->Flags.bIsPutReady == FALSE )
0367            return FALSE;
0368
0369        TransmitTCP(&ps->remote,
0370                    ps->localPort,
0371                    ps->remotePort,
0372                    ps->SND_SEQ,
0373                    ps->SND_ACK,
0374                    ACK,
0375                    ps->TxBuffer,
0376                    ps->TxCount);
0377        tempSND_SEQ = ps->SND_SEQ;
0378        tempTxCount = ps->TxCount;
0379        ps->SND_SEQ += (DWORD)ps->TxCount;
0380        tempSND_SEQ = ps->SND_SEQ;
0381        ps->Flags.bIsPutReady       = FALSE;
0382        ps->Flags.bIsTxInProgress   = FALSE;
0383
0384    #ifdef TCP_NO_WAIT_FOR_ACK
0385        MACDiscardTx(ps->TxBuffer);
0386        ps->TxBuffer                = INVALID_BUFFER;
0387        ps->Flags.bIsPutReady       = TRUE;
0388    #endif
0389
0390        return TRUE;
0391    }
0392
0393
0394
0395    /*****************************************************************
0396     * Function:        BOOL TCPIsPutReady(TCP_SOCKET s)
0397     *
0398     * PreCondition:    TCPInit() is already called.
0399     *
0400     * Input:           s       - socket to test
0401     *
0402     * Output:          TRUE if socket 's' is free to transmit
0403     *                  FALSE if socket 's' is not free to transmit.
0404     *
0405     *****************************************************************/
0406    BOOL TCPIsPutReady(TCP_SOCKET s)
0407    {
0408        if ( TCB[s].TxBuffer == INVALID_BUFFER )
0409            return IPIsTxReady();
0410        else
0411            return TCB[s].Flags.bIsPutReady;
0412    }
0413
0414    /*****************************************************************
0415     * Function:        BOOL TCPPut(TCP_SOCKET s, BYTE byte)
0416     *
0417     * PreCondition:    TCPIsPutReady() == TRUE
0418     *
0419     * Input:           s       - socket to use
0420     *                  byte    - a data byte to send
```

```
0421     *
0422     * Output:           TRUE if given byte was put in transmit buffer
0423     *                   FALSE if transmit buffer is full.
0424    *
0425     *****************************************************************/
0426    BOOL TCPPut(TCP_SOCKET s, BYTE byte)
0427    {
0428        WORD tempTxCount;       // This is a fix for HITECH compiler bug
0429        SOCKET_INFO* ps;
0430
0431        ps = &TCB[s];
0432
0433        if ( ps->TxBuffer == INVALID_BUFFER )
0434        {
0435            ps->TxBuffer = MACGetTxBuffer();
0436
0437            // This function is used to transmit data only.  And every data packet
0438            // must be ack'ed by remote node.  Until this packet is ack'ed by
0439            // remote node, we must preserve its content so that we can retransmit
0440            // if we need to.
0441            MACReserveTxBuffer(ps->TxBuffer);
0442            ps->TxCount = 0;
0443            IPSetTxBuffer(ps->TxBuffer, sizeof(TCP_HEADER));
0444        }
0445
0446        tempTxCount = ps->TxCount;
0447        if ( tempTxCount >= MAX_TCP_DATA_LEN )
0448            return FALSE;
0449
0450        ps->Flags.bIsTxInProgress = TRUE;
0451
0452        MACPut(byte);
0453
0454        // REMOVE
0455        //tempTxCount = ps->TxCount;
0456        tempTxCount++;
0457        ps->TxCount = tempTxCount;
0458
0459        //ps->TxCount++;
0460        //tempTxCount = ps->TxCount;
0461        if ( tempTxCount >= MAX_TCP_DATA_LEN )
0462            TCPFlush(s);
0463        //if ( TCB[s].TxCount >= MAX_TCP_DATA_LEN )
0464        //    TCPFlush(s);
0465
0466        return TRUE;
0467    }
0468
0469    /*****************************************************************
0470     * Function:         BOOL TCPDiscard(TCP_SOCKET s)
0471     *
0472     * PreCondition:     TCPInit() is already called.
0473     *
0474     * Input:            s       - socket
0475     *
0476     * Output:           TRUE if socket received data was discarded
0477     *                         FALSE if socket received data was already
0478     *                           discarded.
0479     *
0480     *****************************************************************/
0481    BOOL TCPDiscard(TCP_SOCKET s)
0482    {
0483        SOCKET_INFO* ps;
0484
0485        ps = &TCB[s];
0486
0487        if ( !ps->Flags.bIsGetReady )
0488            return FALSE;
0489
0490        MACDiscardRx();
0491        ps->Flags.bIsGetReady = FALSE;
0492
0493        return TRUE;
0494    }
0495
0496    /*****************************************************************
0497     * Function:         WORD TCPGetArray(TCP_SOCKET s, BYTE *buffer,
0498     *                                    WORD count)
0499     *
0500     * PreCondition:     TCPInit() is already called     AND
0501     *                   TCPIsGetReady(s) == TRUE
0502     *
0503     * Input:            s       - socket
0504     *                   buffer  - Buffer to hold received data.
```

```
0505     *                   count   - Buffer length
0506     *
0507     * Output:           Number of bytes loaded into buffer.
0508     *
0509     *****************************************************************/
0510    WORD TCPGetArray(TCP_SOCKET s, BYTE *buffer, WORD count)
0511    {
0512        SOCKET_INFO *ps;
0513        ps = &TCB[s];
0514
0515        if ( ps->Flags.bIsGetReady )
0516        {
0517            if ( ps->Flags.bFirstRead )
0518            {
0519                // Position read pointer to begining of correct
0520                // buffer.
0521                IPSetRxBuffer(sizeof(TCP_HEADER));
0522
0523                ps->Flags.bFirstRead = FALSE;
0524            }
0525
0526            ps->Flags.bIsTxInProgress = TRUE;
0527
0528            return MACGetArray(buffer, count);
0529        }
0530        else
0531            return 0;
0532    }
0533
0534
0535    /*****************************************************************
0536     * Function:         BOOL TCPGet(TCP_SOCKET s, BYTE *byte)
0537     *
0538     * PreCondition:     TCPInit() is already called     AND
0539     *                   TCPIsGetReady(s) == TRUE
0540     *
0541     * Input:            s       - socket
0542     *                   byte    - Pointer to a byte.
0543     *
0544     * Output:           TRUE if a byte was read.
0545     *                   FALSE if byte was not read.
0546     *
0547     *****************************************************************/
0548    BOOL TCPGet(TCP_SOCKET s, BYTE *byte)
0549    {
0550        SOCKET_INFO* ps;
0551
0552        ps = &TCB[s];
0553
0554        if ( ps->Flags.bIsGetReady )
0555        {
0556            if ( ps->Flags.bFirstRead )
0557            {
0558                // Position read pointer to begining of correct
0559                // buffer.
0560                IPSetRxBuffer(sizeof(TCP_HEADER));
0561                ps->Flags.bFirstRead = FALSE;
0562
0563            }
0564
0565            if ( ps->RxCount == 0 )
0566            {
0567                MACDiscardRx();
0568                ps->Flags.bIsGetReady = FALSE;
0569                return FALSE;
0570            }
0571
0572            ps->RxCount--;
0573            *byte = MACGet();
0574            return TRUE;
0575        }
0576        return FALSE;
0577    }
0578
0579
0580
0581    /*****************************************************************
0582     * Function:         BOOL TCPIsGetReady(TCP_SOCKET s)
0583     *
0584     * PreCondition:     TCPInit() is already called.
0585     *
0586     * Input:            s       - socket to test
0587     *
0588     * Output:           TRUE if socket 's' contains any data.
```

```
0589     *                   FALSE if socket 's' does not contain any data.
0590    *
0591     *****************************************************************/
0592    BOOL TCPIsGetReady(TCP_SOCKET s)
0593    {
0594        /*
0595         * A socket is said to be "Get" ready when it has already
0596         * received some data.  Sometime, a socket may be closed,
0597         * but it still may contain data.  Thus in order to ensure
0598         * reuse of a socket, caller must make sure that it reads
0599         * a socket, if is ready.
0600         */
0601        return (TCB[s].Flags.bIsGetReady );
0602    }
0603
0604    /*****************************************************************
0605     * Function:         void TCPTick(void)
0606     *
0607     * PreCondition:     TCPInit() is already called.
0608     *
0609     * Input:            None
0610     *
0611     * Output:           Each socket FSM is executed for any timeout
0612     *                   situation.
0613    *
0614     *****************************************************************/
0615    void TCPTick(void)
0616    {
0617    #if !defined(TCP_NO_WAIT_FOR_ACK)
0618        TCP_SOCKET s;
0619        TICK diffTicks;
0620        TICK tick;
0621        SOCKET_INFO* ps;
0622        DWORD seq;
0623        BYTE flags = 0x00;
0624
0625      /*
0626         * Periodically all "not closed" sockets must perform timed operation.
0627         */
0628        for ( s = 0; s < MAX_SOCKETS; s++ )
0629        {
0630            ps = &TCB[s];
0631
0632            if ( ps->Flags.bIsGetReady || ps->Flags.bIsTxInProgress )
0633                continue;
0634
0635            if ( (ps->smState == TCP_CLOSED) ||
0636                 (ps->smState == TCP_LISTEN &&
0637                  ps->Flags.bServer == TRUE) )
0638                continue;
0639
0640            tick = TickGet();
0641
0642            // Calculate timeout value for this socket.
0643            diffTicks = TickGetDiff(tick, ps->startTick);
0644
0645            // If timeout has not occured, do not do anything.
0646            if ( diffTicks <= ps->TimeOut )
0647                continue;
0648
0649            /*
0650             * All states require retransmission, so check for transmitter
0651             * availability right here - common for all.
0652             */
0653            if ( !IPIsTxReady() )
0654                return;
0655
0656            // Restart timeout reference.
0657            ps->startTick = TickGet();
0658
0659            // Update timeout value if there is need to wait longer.
0660            ps->TimeOut++;
0661
0662            // This will be one more attempt.
0663            ps->RetryCount++;
0664
0665            /*
0666             * A timeout has occured.  Respond to this timeout condition
0667             * depending on what state this socket is in.
0668             */
0669            switch(ps->smState)
0670            {
0671            case TCP_SYN_SENT:
0672                /*
```

```
0673                 * Keep sending SYN until we hear from remote node.
0674                 * This may be for infinite time, in that case
0675                 * caller must detect it and do something.
0676                 */
0677                flags = SYN | ACK;
0678                break;
0679
0680            case TCP_SYN_RCVD:
0681                /*
0682                 * We must receive ACK before timeout expires.
0683                 * If not, resend SYN+ACK.
0684                 * Abort, if maximum attempts counts are reached.
0685                 */
0686                if ( ps->RetryCount < MAX_RETRY_COUNTS )
0687                {
0688                    flags = SYN | ACK;
0689                }
0690                else
0691                    CloseSocket(ps);
0692                break;
0693
0694            case TCP_EST:
0695                /*
0696                 * Don't let this connection idle for very long time.
0697                 * If we did not receive or send any message before timeout
0698                 * expires, close this connection.
0699                 */
0700                if ( ps->RetryCount <= MAX_RETRY_COUNTS )
0701                {
0702                    if ( ps->TxBuffer != INVALID_BUFFER )
0703                    {
0704                        MACSetTxBuffer(ps->TxBuffer, 0);
0705                        MACFlush();
0706                    }
0707                    else
0708                        flags = ACK;
0709                }
0710                else
0711                {
0712                    // Forget about previous transmission.
0713                    if ( ps->TxBuffer != INVALID_BUFFER )
0714                        MACDiscardTx(ps->TxBuffer);
0715                    ps->TxBuffer = INVALID_BUFFER;
0716
0717                    // Request closure.
0718                    flags = FIN | ACK;
0719
0720                    ps->smState = TCP_FIN_WAIT_1;
0721                }
0722                break;
0723
0724            case TCP_FIN_WAIT_1:
0725            case TCP_LAST_ACK:
0726                /*
0727                 * If we do not receive any response to out close request,
0728                 * close it outselves.
0729                 */
0730                if ( ps->RetryCount > MAX_RETRY_COUNTS )
0731                    CloseSocket(ps);
0732                else
0733                    flags = FIN;
0734                break;
0735
0736            case TCP_CLOSING:
0737            case TCP_TIMED_WAIT:
0738                /*
0739                 * If we do not receive any response to out close request,
0740                 * close it outselves.
0741                 */
0742                if ( ps->RetryCount > MAX_RETRY_COUNTS )
0743                    CloseSocket(ps);
0744                else
0745                    flags = ACK;
0746                break;
0747            }
0748
0749            if ( flags > 0x00 )
0750            {
0751                if ( flags != ACK )
0752                    seq = ps->SND_SEQ++;
0753                else
0754                    seq = ps->SND_SEQ;
0755
0756                SendTCP(&ps->remote,
```

```
0757                        ps->localPort,
0758                        ps->remotePort,
0759                        seq,
0760                        ps->SND_ACK,
0761                        flags);
0762            }
0763        }
0764    #else
0765        return;
0766    #endif
0767    }
0768
0769    /*****************************************************************************
0770     * Function:         BOOL TCPProcess(NODE_INFO* remote,
0771     *                                   WORD len)
0772     *
0773     * PreCondition:     TCPInit() is already called     AND
0774     *                   TCP segment is ready in MAC buffer
0775     *
0776     * Input:            remote      - Remote node info
0777     *                   len         - Total length of TCP semgent.
0778     *
0779     * Output:           TRUE if this function has completed its task
0780     *                   FALSE otherwise
0781    *
0782     *****************************************************************************/
0783    BOOL TCPProcess(NODE_INFO *remote, WORD len)
0784    {
0785        TCP_HEADER      TCPHeader;
0786        PSEUDO_HEADER   pseudoHeader;
0787        TCP_SOCKET      socket;
0788        WORD_VAL        checksum;
0789        BYTE            optionsSize;
0790
0791
0792        // Retrieve TCP header.
0793        MACGetArray((BYTE*)&TCPHeader, sizeof(TCPHeader));
0794
0795        SwapTCPHeader(&TCPHeader);
0796
0797        // Calculate IP pseudoheader checksum.
0798        pseudoHeader.SourceAddress      = remote->IPAddr;
0799        pseudoHeader.DestAddress.v[0]   = MY_IP_BYTE1;
0800        pseudoHeader.DestAddress.v[1]   = MY_IP_BYTE2;
0801        pseudoHeader.DestAddress.v[2]   = MY_IP_BYTE3;
0802        pseudoHeader.DestAddress.v[3]   = MY_IP_BYTE4;
0803        pseudoHeader.Zero               = 0x0;
0804        pseudoHeader.Protocol           = IP_PROT_TCP;
0805        pseudoHeader.TCPLength          = len;
0806
0807        SwapPseudoTCPHeader(pseudoHeader);
0808
0809
0810        checksum.Val = ~CalcIPChecksum((BYTE*)&pseudoHeader,
0811                                       sizeof(pseudoHeader));
0812
0813        // Set TCP packet checksum = pseudo header checksum in MAC RAM.
0814        IPSetRxBuffer(16);
0815        MACPut(checksum.v[0]);
0816        MACPut(checksum.v[1]);
0817        IPSetRxBuffer(0);
0818
0819        // Now calculate TCP packet checksum in NIC RAM - including
0820        // pesudo header.
0821        checksum.Val = CalcTCPChecksum(len);
0822
0823        if ( checksum.Val != TCPHeader.Checksum )
0824        {
0825            MACDiscardRx();
0826            return TRUE;
0827        }
0828
0829        // Skip over options and retrieve all data bytes.
0830        optionsSize = (BYTE)((TCPHeader.DataOffset.Val << 2)-
0831                             sizeof(TCPHeader));
0832        len = len - optionsSize - sizeof(TCPHeader);
0833
0834        // Position packet read pointer to start of data area.
0835        IPSetRxBuffer((TCPHeader.DataOffset.Val << 2));
0836
0837        // Find matching socket.
0838        socket = FindMatchingSocket(&TCPHeader, remote);
0839        if ( socket < INVALID_SOCKET )
0840        {
```

```
0841            HandleTCPSeg(socket, remote, &TCPHeader, len);
0842        }
0843        else
0844        {
0845            MACDiscardRx();
0846
0847            if ( socket == UNKNOWN_SOCKET )
0848            {
0849
0850                TCPHeader.AckNumber += len;
0851                if ( TCPHeader.Flags.bits.flagSYN ||
0852                     TCPHeader.Flags.bits.flagFIN )
0853                    TCPHeader.AckNumber++;
0854
0855                SendTCP(remote,
0856                        TCPHeader.DestPort,
0857                        TCPHeader.SourcePort,
0858                        TCPHeader.AckNumber,
0859                        TCPHeader.SeqNumber,
0860                        RST);
0861            }
0862
0863        }
0864
0865        return TRUE;
0866    }
0867
0868    /*****************************************************************
0869     * Function:        static void TransmitTCP(NODE_INFO* remote
0870     *                                          TCP_PORT localPort,
0871     *                                          TCP_PORT remotePort,
0872     *                                          DWORD seq,
0873     *                                          DWORD ack,
0874     *                                          BYTE flags,
0875     *                                          BUFFER buffer,
0876     *                                          WORD len)
0877     *
0878     * PreCondition:    TCPInit() is already called     AND
0879     *                  TCPIsPutReady() == TRUE
0880     *
0881     * Input:           remote      - Remote node info
0882     *                  localPort   - Source port number
0883     *                  remotePort  - Destination port number
0884     *                  seq         - Segment sequence number
0885     *                  ack         - Segment acknowledge number
0886     *                  flags       - Segment flags
0887     *                  buffer      - Buffer to which this segment
0888     *                                is to be transmitted
0889     *                  len         - Total data length for this segment.
0890     *
0891     * Output:          A TCP segment is assembled and put to transmit.
0892     *
0893     *****************************************************************/
0894    static void TransmitTCP(NODE_INFO *remote,
0895                            TCP_PORT localPort,
0896                            TCP_PORT remotePort,
0897                            DWORD tseq,
0898                            DWORD tack,
0899                            BYTE flags,
0900                            BUFFER buffer,
0901                            WORD len)
0902    {
0903        WORD_VAL        checkSum;
0904        TCP_HEADER      header;
0905        TCP_OPTIONS     options;
0906        PSEUDO_HEADER   pseudoHeader;
0907
0908        while( !IPIsTxReady() );
0909
0910        header.SourcePort           = localPort;
0911        header.DestPort             = remotePort;
0912        header.SeqNumber            = tseq;
0913        header.AckNumber            = tack;
0914        header.Flags.bits.Reserved2 = 0;
0915        header.DataOffset.Reserved3 = 0;
0916        header.Flags.byte           = flags;
0917        // Receive window = MAC Free buffer size - TCP header (20) - IP header (20)
0918        //                  - ETHERNET header (14 if using NIC) .
0919        header.Window               = MACGetFreeRxSize();
0920    #if !defined(STACK_USE_SLIP)
0921        /*
0922         * Limit one segment at a time from remote host.
0923         * This limit increases overall throughput as remote host does not
0924         * flood us with packets and later retry with significant delay.
```

```
0925         */
0926        if ( header.Window >= MAC_RX_BUFFER_SIZE )
0927            header.Window = MAC_RX_BUFFER_SIZE;
0928
0929        else if ( header.Window > 54 )
0930        {
0931            header.Window -= 54;
0932        }
0933        else
0934            header.Window = 0;
0935    #else
0936        if ( header.Window > 40 )
0937        {
0938            header.Window -= 40;
0939        }
0940        else
0941            header.Window = 0;
0942    #endif
0943
0944        header.Checksum             = 0;
0945        header.UrgentPointer        = 0;
0946
0947        SwapTCPHeader(&header);
0948
0949        len += sizeof(header);
0950
0951        if ( flags & SYN )
0952        {
0953            len += sizeof(options);
0954            options.Kind = TCP_OPTIONS_MAX_SEG_SIZE;
0955            options.Length = 0x04;
0956
0957            // Load MSS in already swapped order.
0958            options.MaxSegSize.v[0]  = (MAC_RX_BUFFER_SIZE >> 8); // 0x05;
0959            options.MaxSegSize.v[1]  = (MAC_RX_BUFFER_SIZE & 0xff); // 0xb4;
0960
0961            header.DataOffset.Val   = (sizeof(header) + sizeof(options)) >> 2;
0962        }
0963        else
0964            header.DataOffset.Val   = sizeof(header) >> 2;
0965
0966
0967        // Calculate IP pseudoheader checksum.
0968        pseudoHeader.SourceAddress.v[0] = MY_IP_BYTE1;
0969        pseudoHeader.SourceAddress.v[1] = MY_IP_BYTE2;
0970        pseudoHeader.SourceAddress.v[2] = MY_IP_BYTE3;
0971        pseudoHeader.SourceAddress.v[3] = MY_IP_BYTE4;
0972        pseudoHeader.DestAddress    = remote->IPAddr;
0973        pseudoHeader.Zero           = 0x0;
0974        pseudoHeader.Protocol       = IP_PROT_TCP;
0975        pseudoHeader.TCPLength      = len;
0976
0977        SwapPseudoTCPHeader(pseudoHeader);
0978
0979        header.Checksum = ~CalcIPChecksum((BYTE*)&pseudoHeader,
0980                            sizeof(pseudoHeader));
0981        checkSum.Val = header.Checksum;
0982
0983        if ( buffer == INVALID_BUFFER )
0984            buffer = MACGetTxBuffer();
0985
0986        IPSetTxBuffer(buffer, 0);
0987
0988        // Write IP header.
0989        IPPutHeader(remote, IP_PROT_TCP, len);
0990        IPPutArray((BYTE*)&header, sizeof(header));
0991
0992        if ( flags & SYN )
0993            IPPutArray((BYTE*)&options, sizeof(options));
0994
0995        IPSetTxBuffer(buffer, 0);
0996
0997        checkSum.Val = CalcTCPChecksum(len);
0998
0999        // Update the checksum.
1000        IPSetTxBuffer(buffer, 16);
1001        MACPut(checkSum.v[1]);
1002        MACPut(checkSum.v[0]);
1003        MACSetTxBuffer(buffer, 0);
1004
1005        MACFlush();
1006    }
1007
1008    /*****************************************************************************
```

```
1009    * Function:        static WORD CalcTCPChecksum(WORD len)
1010    *
1011    * PreCondition:     TCPInit() is already called     AND
1012    *                   MAC buffer pointer set to starting of buffer
1013    *
1014    * Input:            len      - Total number of bytes to calculate
1015    *                              checksum for.
1016    *
1017    * Output:           16-bit checksum as defined by rfc 793.
1018    *
1019    *****************************************************************/
1020   static WORD CalcTCPChecksum(WORD len)
1021   {
1022       BOOL lbMSB = TRUE;
1023       WORD_VAL checkSum;
1024       BYTE Checkbyte;
1025
1026       checkSum.Val = 0;
1027
1028       while( len-- )
1029       {
1030           Checkbyte = MACGet();
1031
1032           if ( !lbMSB )
1033           {
1034               if ( (checkSum.v[0] = Checkbyte+checkSum.v[0]) < Checkbyte)
1035               {
1036                   if ( ++checkSum.v[1] == 0 )
1037                       checkSum.v[0]++;
1038               }
1039           }
1040           else
1041           {
1042               if ( (checkSum.v[1] = Checkbyte+checkSum.v[1]) < Checkbyte)
1043               {
1044                   if ( ++checkSum.v[0] == 0 )
1045                       checkSum.v[1]++;
1046               }
1047           }
1048
1049           lbMSB = !lbMSB;
1050       }
1051
1052       checkSum.v[1] = ~checkSum.v[1];
1053       checkSum.v[0] = ~checkSum.v[0];
1054       return checkSum.Val;
1055   }
1056
1057   /*****************************************************************
1058    * Function:        static TCP_SOCKET FindMatchingSocket(TCP_HEADER *h,
1059    *                                       NODE_INFO* remote)
1060    *
1061    * PreCondition:     TCPInit() is already called
1062    *
1063    * Input:            h           - TCP Header to be matched against.
1064    *                   remote      - Node who sent this header.
1065    *
1066    * Output:           A socket that matches with given header and remote
1067    *                   node is searched.
1068    *                   If such socket is found, its index is returned
1069    *                   else INVALID_SOCKET is returned.
1070    *
1071    *****************************************************************/
1072   static TCP_SOCKET FindMatchingSocket(TCP_HEADER *h, NODE_INFO *remote)
1073   {
1074       SOCKET_INFO *ps;
1075       TCP_SOCKET s;
1076       TCP_SOCKET partialMatch;
1077
1078       partialMatch = INVALID_SOCKET;
1079
1080       for ( s = 0; s < MAX_SOCKETS; s++ )
1081       {
1082           ps = &TCB[s];
1083
1084           if ( ps->smState != TCP_CLOSED )
1085           {
1086               if ( ps->localPort == h->DestPort )
1087               {
1088                   if ( ps->smState == TCP_LISTEN )
1089                       partialMatch = s;
1090
1091                   if ( ps->remotePort == h->SourcePort &&
1092                       ps->remote.IPAddr.Val == remote->IPAddr.Val )
```

```
1093                    {
1094                            return s;
1095                    }
1096                }
1097            }
1098        }
1099
1100        ps = &TCB[partialMatch];
1101
1102        if ( partialMatch != INVALID_SOCKET &&
1103             ps->smState == TCP_LISTEN )
1104        {
1105            memcpy((void*)&ps->remote, (void*)remote, sizeof(*remote));
1106            //ps->remote                = *remote;
1107            ps->localPort             = h->DestPort;
1108            ps->remotePort            = h->SourcePort;
1109            ps->Flags.bIsGetReady     = FALSE;
1110            ps->TxBuffer              = INVALID_BUFFER;
1111            ps->Flags.bIsPutReady     = TRUE;
1112
1113            return partialMatch;
1114        }
1115
1116        if ( partialMatch == INVALID_SOCKET )
1117            return UNKNOWN_SOCKET;
1118        else
1119            return INVALID_SOCKET;
1120    }
1121
1122    /*****************************************************************
1123     * Function:        static void SwapTCPHeader(TCP_HEADER* header)
1124     *
1125     * PreCondition:    None
1126     *
1127     * Input:           header      - TCP Header to be swapped.
1128     *
1129     * Output:          Given header is swapped.
1130     *
1131     *****************************************************************/
1132    static void SwapTCPHeader(TCP_HEADER* header)
1133    {
1134        header->SourcePort      = swaps(header->SourcePort);
1135        header->DestPort        = swaps(header->DestPort);
1136        header->SeqNumber       = swapl(header->SeqNumber);
1137        header->AckNumber       = swapl(header->AckNumber);
1138        header->Window          = swaps(header->Window);
1139        header->Checksum        = swaps(header->Checksum);
1140        header->UrgentPointer   = swaps(header->UrgentPointer);
1141    }
1142
1143    /*****************************************************************
1144     * Function:        static void CloseSocket(SOCKET_INFO* ps)
1145     *
1146     * PreCondition:    TCPInit() is already called
1147     *
1148     * Input:           ps  - Pointer to a socket info that is to be
1149     *                        closed.
1150     *
1151     * Output:          Given socket information is reset and any
1152     *                  buffer held by this socket is discarded.
1153     *
1154     *****************************************************************/
1155    static void CloseSocket(SOCKET_INFO* ps)
1156    {
1157        if ( ps->TxBuffer != INVALID_BUFFER )
1158        {
1159            MACDiscardTx(ps->TxBuffer);
1160            ps->TxBuffer            = INVALID_BUFFER;
1161            ps->Flags.bIsPutReady   = TRUE;
1162        }
1163
1164        ps->remote.IPAddr.Val = 0x00;
1165        ps->remotePort = 0x00;
1166        if ( ps->Flags.bIsGetReady )
1167        {
1168            MACDiscardRx();
1169        }
1170        ps->Flags.bIsGetReady       = FALSE;
1171        ps->TimeOut                 = TCP_START_TIMEOUT_VAL;
1172
1173        ps->Flags.bIsTxInProgress   = FALSE;
1174
1175        if ( ps->Flags.bServer )
1176            ps->smState = TCP_LISTEN;
```

```
1177        else
1178            ps->smState = TCP_CLOSED;
1179        return;
1180    }
1181
1182    /*****************************************************************
1183     * Function:        static void HandleTCPSeg(TCP_SOCKET s,
1184     *                                           NODE_INFO *remote,
1185     *                                           TCP_HEADER* h,
1186     *                                           WORD len)
1187     *
1188     * PreCondition:    TCPInit() is already called     AND
1189     *                  TCPProcess() is the caller.
1190     *
1191     * Input:           s           - Socket that owns this segment
1192     *                  remote      - Remote node info
1193     *                  h           - TCP Header
1194     *                  len         - Total buffer length.
1195     *
1196     * Output:          TCP FSM is executed on given socket with
1197     *                  given TCP segment.
1198     *
1199     *****************************************************************/
1200    static void HandleTCPSeg(TCP_SOCKET s,
1201                             NODE_INFO *remote,
1202                             TCP_HEADER *h,
1203                             WORD len)
1204    {
1205        DWORD ack;
1206        DWORD seq;
1207        DWORD prevAck, prevSeq;
1208        SOCKET_INFO *ps;
1209        BYTE flags = 0x00;
1210
1211
1212        ps = &TCB[s];
1213
1214        prevAck = ps->SND_ACK;
1215        prevSeq = ps->SND_SEQ;
1216
1217        ack = h->SeqNumber ;
1218        ack += (DWORD)len;
1219        seq = ps->SND_SEQ;
1220
1221        ps->RetryCount  = 0;
1222        ps->startTick   = TickGet();
1223        ps->TimeOut = TCP_START_TIMEOUT_VAL;
1224
1225        if ( ps->smState == TCP_LISTEN )
1226        {
1227            MACDiscardRx();
1228
1229            ps->SND_SEQ     = ++ISS;
1230            ps->SND_ACK     = ++ack;
1231            seq             = ps->SND_SEQ;
1232            ++ps->SND_SEQ;
1233            if ( h->Flags.bits.flagSYN )
1234            {
1235                memcpy((void*)&ps->remote, remote, sizeof(*remote));
1236                ps->remotePort  = h->SourcePort;
1237                flags           = SYN | ACK;
1238                ps->smState     = TCP_SYN_RCVD;
1239
1240            }
1241            else
1242            {
1243                flags               = RST;
1244                seq                 = ack;
1245                ack                 = h->SeqNumber;
1246                ps->remote.IPAddr.Val = 0x00;
1247            }
1248
1249        }
1250        else
1251        {
1252            /*
1253             * Reset FSM, if RST is received.
1254             */
1255            if ( h->Flags.bits.flagRST )
1256            {
1257                MACDiscardRx();
1258
1259                CloseSocket(ps);
1260
```

```
1261                return;
1262
1263            }
1264
1265            else if ( seq == h->AckNumber )
1266            {
1267                if ( ps->smState == TCP_SYN_RCVD )
1268                {
1269                    if ( h->Flags.bits.flagACK )
1270                    {
1271                        ps->SND_ACK = ack;
1272                        ps->RetryCount = 0;
1273                        ps->startTick = TickGet();
1274                        ps->smState = TCP_EST;
1275
1276                        if ( len > 0 )
1277                        {
1278                            ps->Flags.bIsGetReady   = TRUE;
1279                            ps->RxCount             = len;
1280                            ps->Flags.bFirstRead    = TRUE;
1281                        }
1282                        else
1283                            MACDiscardRx();
1284                    }
1285                    else
1286                    {
1287                        MACDiscardRx();
1288                    }
1289                }
1290                else if ( ps->smState == TCP_SYN_SENT )
1291                {
1292                    if ( h->Flags.bits.flagSYN )
1293                    {
1294                        ps->SND_ACK = ++ack;
1295                        if ( h->Flags.bits.flagACK )
1296                        {
1297                            flags = ACK;
1298                            ps->smState = TCP_EST;
1299                        }
1300                        else
1301                        {
1302                            flags = SYN | ACK;
1303                            ps->smState = TCP_SYN_RCVD;
1304                            ps->SND_SEQ = ++seq;
1305                        }
1306
1307                        if ( len > 0 )
1308                        {
1309                            ps->Flags.bIsGetReady   = TRUE;
1310                            ps->RxCount             = len;
1311                            ps->Flags.bFirstRead    = TRUE;
1312                        }
1313                        else
1314                            MACDiscardRx();
1315                    }
1316                    else
1317                    {
1318                        MACDiscardRx();
1319                    }
1320                }
1321                else
1322                {
1323                    if ( h->SeqNumber != ps->SND_ACK )
1324                    {
1325                        // Discard buffer.
1326                        MACDiscardRx();
1327                        return;
1328                    }
1329
1330                    ps->SND_ACK = ack;
1331
1332                    if ( ps->smState == TCP_EST )
1333                    {
1334                        if ( h->Flags.bits.flagACK )
1335                        {
1336                            if ( ps->TxBuffer != INVALID_BUFFER )
1337                            {
1338                                MACDiscardTx(ps->TxBuffer);
1339                                ps->TxBuffer            = INVALID_BUFFER;
1340                                ps->Flags.bIsPutReady   = TRUE;
1341                            }
1342                        }
1343
1344                        if ( h->Flags.bits.flagFIN )
```

```
1345                        {
1346                            flags = FIN | ACK;
1347                            seq = ps->SND_SEQ++;
1348                            ack = ++ps->SND_ACK;
1349                            ps->smState = TCP_LAST_ACK;
1350                        }
1351
1352
1353                        if ( len > 0 )
1354                        {
1355                            if ( !ps->Flags.bIsGetReady )
1356                            {
1357                                ps->Flags.bIsGetReady   = TRUE;
1358                                ps->RxCount             = len;
1359                                ps->Flags.bFirstRead    = TRUE;
1360
1361                                 // 4/1/02
1362                                flags = ACK;
1363                           }
1364                            else
1365                            {
1366
1367                             flags = 0x00;
1368                                ps->SND_SEQ = prevSeq;
1369                                ps->SND_ACK = prevAck;
1370
1371                                MACDiscardRx();
1372                            }
1373                        }
1374                        else
1375                        {
1376                            MACDiscardRx();
1377                        }
1378
1379
1380                    }
1381                    else if ( ps->smState == TCP_LAST_ACK )
1382                    {
1383                        MACDiscardRx();
1384
1385                        if ( h->Flags.bits.flagACK )
1386                        {
1387                            CloseSocket(ps);
1388                        }
1389                    }
1390                    else if ( ps->smState == TCP_FIN_WAIT_1 )
1391                    {
1392                        MACDiscardRx();
1393
1394                        if ( h->Flags.bits.flagFIN )
1395                        {
1396                            flags = ACK;
1397                            ack = ++ps->SND_ACK;
1398                            if ( h->Flags.bits.flagACK )
1399                            {
1400                                CloseSocket(ps);
1401                            }
1402                            else
1403                            {
1404                                ps->smState = TCP_CLOSING;
1405                            }
1406                        }
1407                    }
1408                    else if ( ps->smState == TCP_CLOSING )
1409                    {
1410                        MACDiscardRx();
1411
1412                        if ( h->Flags.bits.flagACK )
1413                        {
1414                            CloseSocket(ps);
1415                        }
1416                    }
1417                }
1418            }
1419            else
1420            {
1421                MACDiscardRx();
1422            }
1423        }
1424
1425        if ( flags > 0x00 )
1426        {
1427            SendTCP(remote,
1428                    h->DestPort,
```

```
1429                    h->SourcePort,
1430                    seq,
1431                    ack,
1432                    flags);
1433        }
1434    }
1435
```

```
0001  /*********************************************************************
0002  *       TCPIC Web Server Application
0003  *********************************************************************/
0004
0005  #define THIS_IS_STACK_APPLICATION
0006
0007  #define BAUD_RATE       (19200)     // bps
0008
0009  #define USART_USE_BRGH_LOW
0010  #if defined(USART_USE_BRGH_LOW)
0011      #define SPBRG_VAL   ( ((CLOCK_FREQ/BAUD_RATE)/64) - 1)
0012  #else
0013      #define SPBRG_VAL   ( ((CLOCK_FREQ/BAUD_RATE)/16) - 1)
0014  #endif
0015
0016  #if SPBRG_VAL > 255
0017      #error "Calculated SPBRG value is out of range for currnet CLOCK_FREQ."
0018  #endif
0019
0020  #include <string.h>
0021
0022  /*
0023   * These headers must be included for required defs.
0024   */
0025  #include "stacktsk.h"
0026  #include "tick.h"
0027
0028
0029  // #define STACK_USE_DHCP (0)
0030  // #define STACK_USE_IP_GLEANING (0)
0031
0032  #if defined(STACK_USE_DHCP)
0033  #include "dhcp.h"
0034  #endif
0035
0036  #if defined(STACK_USE_HTTP_SERVER)
0037  #include "http.h"
0038  #endif
0039
0040  #include "mpfs.h"
0041
0042  #if defined(STACK_USE_FTP_SERVER) && defined(MPFS_USE_EEPROM)
0043  #include "ftp.h"
0044  #endif
0045
0046  #include "xlcd.h"
0047
0048  #if defined(MPFS_USE_EEPROM)
0049  #include "xeeprom.h"
0050  #endif
0051
0052
0053  // For debug only.
0054  #include "tcp.h"
0055  #include "icmp.h"
0056
0057  #include "delay.h"
0058
0059  ROM char StartupMsg[] = "TCP/IC    v1.12";
0060
0061  #if defined(STACK_USE_DHCP) || defined(STACK_USE_IP_GLEANING)
0062  ROM char DHCPMsg[] = "DHCP/Gleaning...";
0063  #endif
0064
0065  ROM char SetupMsg[] = "RS232 Setup...";
0066  APP_CONFIG AppConfig;
0067
0068  BYTE myDHCPBindCount = 0;
0069  #if defined(STACK_USE_DHCP)
0070      extern BYTE DHCPBindCount;
0071  #else
0072      BYTE DHCPBindCount = 1;
0073  #endif
0074
0075  #if defined(HITECH_C18)
0076  __CONFIG(1, UNPROTECT & HS);
0077  __CONFIG(2, PWRTEN & BORDIS & WDTDIS);
0078  #endif
0079
0080  static void InitAppConfig(void);
0081  static void InitializeBoard(void);
0082  static void ProcessIO(void);
0083  void NotifyRemoteUser(void);
0084  static void DisplayIPValue(IP_ADDR *IPVal, BOOL bToLCD);
```

```
0085    static void SetConfig(void);
0086
0087
0088    #if defined(MPFS_USE_EEPROM)
0089    static BOOL DownloadMPFS(void);
0090    static void SaveAppConfig(void);
0091    #endif
0092
0093    #if defined(MCHP_C18)
0094        #pragma interrupt HighISR save=section(".tmpdata")
0095        void HighISR(void)
0096    #elif defined(HITECH_C18)
0097        #if defined(STACK_USE_SLIP)
0098            extern void MACISR(void);
0099        #endif
0100        void interrupt HighISR(void)
0101    #endif
0102    {
0103        TickUpdate();
0104
0105    #if defined(STACK_USE_SLIP)
0106        MACISR();
0107    #endif
0108    }
0109
0110    #if defined(MCHP_C18)
0111    #pragma code highVector=0x08
0112    void HighVector (void)
0113    {
0114        _asm goto HighISR _endasm
0115    }
0116    #pragma code /* return to default code section */
0117    #endif
0118
0119    static void USARTPut(BYTE c)
0120    {
0121        while( !TXSTA_TRMT);
0122        TXREG = c;
0123    }
0124
0125    static void USARTPutString(BYTE *s)
0126    {
0127        BYTE c;
0128
0129        while( (c = *s++) )
0130            USARTPut(c);
0131    }
0132
0133    #define USARTIsGetReady()    (PIR1_RCIF)
0134    #define USARTGet()           (RCREG)
0135
0136    /*
0137     * Main entry point.
0138     */
0139    void main(void)
0140    {
0141        TBLPTRU = 0x00;
0142
0143        /*
0144         * Initialize any application specific hardware.
0145         */
0146        InitializeBoard();
0147        TickInit();
0148        MPFSInit();
0149        InitAppConfig();
0150        if ( PORTB_RB5 == 0 )
0151        {
0152            XLCDGoto(1, 0);
0153            XLCDPutROMString(SetupMsg);
0154            SetConfig();
0155        }
0156
0157        StackInit();
0158
0159    #if defined(STACK_USE_HTTP_SERVER)
0160        HTTPInit();
0161    #endif
0162
0163    #if defined(STACK_USE_FTP_SERVER) && defined(MPFS_USE_EEPROM)
0164        FTPInit();
0165    #endif
0166
0167
0168    #if defined(STACK_USE_DHCP) || defined(STACK_USE_IP_GLEANING)
```

```
0169        if ( AppConfig.Flags.bIsDHCPEnabled )
0170        {
0171            XLCDGoto(1, 0);
0172            XLCDPutROMString(DHCPMsg);
0173        }
0174        else
0175        {
0176            myDHCPBindCount = 1;
0177    #if defined(STACK_USE_DHCP)
0178            DHCPDisable();
0179    #endif
0180        }
0181    #endif
0182        while(1)
0183        {
0184            StackTask();
0185    #if defined(STACK_USE_HTTP_SERVER)
0186            HTTPServer();
0187    #endif
0188    #if defined(STACK_USE_FTP_SERVER) && defined(MPFS_USE_EEPROM)
0189            FTPServer();
0190    #endif
0191            ProcessIO();
0192
0193            if ( DHCPBindCount != myDHCPBindCount )
0194            {
0195                DisplayIPValue(&AppConfig.MyIPAddr, TRUE);
0196                myDHCPBindCount = DHCPBindCount;
0197
0198                if ( AppConfig.Flags.bIsDHCPEnabled )
0199                {
0200                    XLCDGoto(1, 14);
0201                    if ( myDHCPBindCount < 0x0a )
0202                        XLCDPut(myDHCPBindCount + '0');
0203                    else
0204                        XLCDPut(myDHCPBindCount + 'A');
0205                }
0206            }
0207
0208        }
0209    }
0210
0211    ROM char blankLCDLine[] = "                ";
0212
0213    static void DisplayIPValue(IP_ADDR *IPVal, BOOL bToLCD)
0214    {
0215        char IPDigit[8];
0216
0217        if ( bToLCD )
0218        {
0219            XLCDGoto(1, 0);
0220            XLCDPutROMString(blankLCDLine);
0221
0222        }
0223
0224        XLCDGoto(1, 0);
0225
0226        itoa(IPVal->v[0], IPDigit);
0227        if ( bToLCD )
0228        {
0229            XLCDPutString(IPDigit);
0230            XLCDPut('.');
0231        }
0232        else
0233        {
0234            USARTPutString(IPDigit);
0235            USARTPut('.');
0236        }
0237
0238        itoa(IPVal->v[1], IPDigit);
0239        if ( bToLCD )
0240        {
0241            XLCDPutString(IPDigit);
0242            XLCDPut('.');
0243        }
0244        else
0245        {
0246            USARTPutString(IPDigit);
0247            USARTPut('.');
0248        }
0249
0250        itoa(IPVal->v[2], IPDigit);
0251        if ( bToLCD )
0252        {
```

```
0253            XLCDPutString(IPDigit);
0254            XLCDPut('.');
0255        }
0256        else
0257        {
0258            USARTPutString(IPDigit);
0259            USARTPut('.');
0260        }
0261
0262        itoa(IPVal->v[3], IPDigit);
0263        if ( bToLCD )
0264            XLCDPutString(IPDigit);
0265        else
0266            USARTPutString(IPDigit);
0267    }
0268
0269    static char AN0String[8];
0270    static char AN1String[8];
0271
0272
0273    static void ProcessIO(void)
0274    {
0275        WORD_VAL ADCResult;
0276        ADCON0      = 0b10000001;
0277
0278        ADCResult.v[0] = 100;
0279        while( ADCResult.v[0]-- );
0280
0281        ADCON0_GO   = 1;
0282        while( ADCON0_GO );
0283
0284        ADCResult.v[0] = ADRESL;
0285        ADCResult.v[1] = ADRESH;
0286        itoa(ADCResult.Val, AN0String);
0287        ADCON1      = 0b10000100;
0288        ADCON0      = 0b10001001;
0289        ADCResult.v[0] = 100;
0290        while( ADCResult.v[0]-- );
0291
0292       ADCON0_GO   = 1;
0293        while( ADCON0_GO );
0294        ADCResult.v[0] = ADRESL;
0295        ADCResult.v[1] = ADRESH;
0296
0297        itoa(ADCResult.Val, AN1String);
0298
0299        ADCON1      = 0b10001110;       // RA0 as analog input.
0300
0301    }
0302
0303    /*
0304     * CGI Command Codes.
0305     */
0306    #define CGI_CMD_DIGOUT      (0)
0307    #define CGI_CMD_LCDOUT      (1)
0308
0309    /*
0310     * CGI Variable codes. - There could be 0-255 variables.
0311     */
0312    #define VAR_LED_D5          (0)
0313    #define VAR_LED_D6          (1)
0314    #define VAR_ANAIN_AN0       (2)
0315    #define VAR_ANAIN_AN1       (3)
0316    #define VAR_DIGIN_RB5       (4)
0317    #define VAR_STROUT_LCD      (5)
0318    #define VAR_OUTPUT_RC0     (8)
0319    #define VAR_OUTPUT_RC1     (9)
0320
0321    /*****************************************************************
0322     * Function:         void HTTPExecCmd(BYTE** argv, BYTE argc)
0323     *
0324     * PreCondition:     None
0325     *
0326     * Input:            argv        - List of arguments
0327     *                   argc        - Argument count.
0328     *
0329     * Output:           None
0330     *
0331     *****************************************************************/
0332    #if defined(STACK_USE_HTTP_SERVER)
0333
0334    ROM char COMMANDS_OK_PAGE[] = "COMMANDS.CGI";
0335
0336    // Copy string with NULL termination.
```

```
0337    #define COMMANDS_OK_PAGE_LEN  (sizeof(COMMANDS_OK_PAGE))
0338
0339    ROM char CMD_UNKNOWN_PAGE[] = "INDEX.HTM";
0340
0341    // Copy string with NULL termination.
0342    #define CMD_UNKNOWN_PAGE_LEN    (sizeof(CMD_UNKNOWN_PAGE))
0343
0344    void HTTPExecCmd(BYTE** argv, BYTE argc)
0345    {
0346        BYTE command;
0347        BYTE var;
0348
0349        command = argv[0][0] - '0';
0350        switch(command)
0351        {
0352        case CGI_CMD_DIGOUT:
0353            /*
0354             * This DIGOUTS.CGI.  Any arguments with this file
0355             * must be about controlling digital outputs.
0356             */
0357
0358            /*
0359             * Identify the parameters.
0360             * Compare it in upper case format.
0361             */
0362            var = argv[1][0] - '0';
0363
0364            switch(var)
0365            {
0366            case VAR_LED_D5:
0367                /*
0368                 * This is "D5".
0369                 * Toggle D5.
0370                 */
0371                LATA3 ^= 1;
0372                break;
0373
0374            case VAR_LED_D6:
0375                /*
0376                 * This is "D6".
0377                 * Toggle it.
0378                 */
0379                LATA2 ^= 1;
0380                break;
0381
0382            case VAR_OUTPUT_RC0:
0383                // This is "RC0" - Toggle it
0384                LATCbits.LATC0 ^= 1;
0385                LATA3 ^= 1; // Also toggle other LED to indicate change !
0386                break;
0387
0388            case VAR_OUTPUT_RC1:
0389                // This is "RC1" - Toggle it
0390                LATCbits.LATC1 ^= 1;
0391                LATA3 ^= 1; // Also toggle other LED to indicate change !
0392                break;
0393             }
0394
0395             memcpypgm2ram(argv[0],
0396                  (const ROM char*)COMMANDS_OK_PAGE, COMMANDS_OK_PAGE_LEN);
0397             break;
0398
0399        case CGI_CMD_LCDOUT:
0400            /*
0401             * Note implemented.
0402             */
0403            break;
0404
0405        default:
0406            memcpypgm2ram((unsigned char*)argv[0],
0407                  (const ROM char*)CMD_UNKNOWN_PAGE, CMD_UNKNOWN_PAGE_LEN);
0408            break;
0409        }
0410    }
0411    #endif
0412
0413
0414    /*****************************************************************
0415     * Function:        WORD HTTPGetVar(BYTE var, WORD ref, BYTE* val)
0416     *
0417     * PreCondition:    None
0418     *
0419     * Input:           var         - Variable Identifier
0420     *                  ref         - Current callback reference with
```

```
0421     *                                   respect to 'var' variable.
0422     *                   val         - Buffer for value storage.
0423     *
0424     * Output:           Variable reference as required by application.
0425     *
0426     ********************************************************************/
0427    #if defined(STACK_USE_HTTP_SERVER)
0428    WORD HTTPGetVar(BYTE var, WORD ref, BYTE* val)
0429    {
0430        switch(var)
0431        {
0432        case VAR_LED_D5:
0433            if ( LATA3 )
0434                *val = '0';
0435            else
0436                *val = '1';
0437            break;
0438
0439        case VAR_LED_D6:
0440            if ( LATA2 )
0441                *val = '0';
0442            else
0443                *val = '1';
0444            break;
0445
0446        case VAR_OUTPUT_RC0:
0447            if ( LATCbits.LATC0 )
0448                *val = '0';
0449            else
0450                *val = '1';
0451            break;
0452
0453        case VAR_OUTPUT_RC1:
0454            if ( LATCbits.LATC1 )
0455                *val = '0';
0456            else
0457                *val = '1';
0458            break;
0459
0460        case VAR_ANAIN_AN0:
0461            if ( ref == HTTP_START_OF_VAR )
0462            {
0463                ref = (BYTE)0;
0464            }
0465            *val = AN0String[(BYTE)ref];
0466            if ( AN0String[(BYTE)ref] == '\0' )
0467                return HTTP_END_OF_VAR;
0468
0469            (BYTE)ref++;
0470            return ref;
0471
0472        case VAR_ANAIN_AN1:
0473            if ( ref == HTTP_START_OF_VAR )
0474            {
0475                ref = (BYTE)0;
0476            }
0477            *val = AN1String[(BYTE)ref];
0478            if ( AN1String[(BYTE)ref] == '\0' )
0479                return HTTP_END_OF_VAR;
0480
0481            (BYTE)ref++;
0482            return ref;
0483
0484        case VAR_DIGIN_RB5:
0485            if ( PORTB_RB5 )
0486                *val = '1';
0487            else
0488                *val = '0';
0489            break;
0490        }
0491
0492        return HTTP_END_OF_VAR;
0493    }
0494    #endif
0495
0496
0497    #if defined(STACK_USE_FTP_SERVER) && defined(MPFS_USE_EEPROM)
0498    ROM char FTP_USER_NAME[]     = "ftp";
0499    #undef FTP_USER_NAME_LEN
0500    #define FTP_USER_NAME_LEN    (sizeof(FTP_USER_NAME)-1)
0501
0502    ROM char FTP_USER_PASS[]     = "wassupzems00";
0503    #define FTP_USER_PASS_LEN    (sizeof(FTP_USER_PASS)-1)
0504
```

```
0505    BOOL FTPVerify(char *login, char *password)
0506    {
0507        if ( !memcmppgm2ram(login, FTP_USER_NAME, FTP_USER_NAME_LEN) )
0508        {
0509            if ( !memcmppgm2ram(password, FTP_USER_PASS, FTP_USER_PASS_LEN) )
0510                return TRUE;
0511        }
0512        return FALSE;
0513    }
0514    #endif
0515
0516    /*****************************************************************
0517     * Function:         void InitializeBoard(void)
0518     *
0519     * PreCondition:     None
0520     *
0521     * Input:            None
0522     *
0523     * Output:           None
0524     *
0525     *****************************************************************/
0526    static void InitializeBoard(void)
0527    {
0528        /*
0529         * Setup for PORTA.RA0 as analog input while rests
0530         * as digital i/o lines.
0531         */
0532        ADCON1  = 0b10001110;       // RA0 as analog input, Right justified
0533        TRISA   = 0x03;
0534       TRISC = 0x00;             // Set xxxxxx00
0535          LATCbits.LATC0 =  0;     // 1 = ON
0536          LATCbits.LATC1 =  0;     // 0 = OFF
0537
0538        /*
0539         * LCD is enabled using RA5.
0540         */
0541        PORTA_RA5 = 0;          // Disable LCD.
0542
0543        /*
0544         * Turn off the LED's.
0545         */
0546        LATA2 = 1;
0547        LATA3 = 1;
0548
0549        /*
0550         * External data EEPROM needs pull-ups, so enable internal
0551         * pull-ups.
0552         */
0553        INTCON2_RBPU = 0;
0554
0555        XLCDInit();
0556        XLCDGoto(0, 0);
0557        XLCDPutROMString(StartupMsg);
0558
0559
0560        TXSTA = 0b00100000;     // Low BRG speed
0561        RCSTA = 0b10010000;
0562        SPBRG = SPBRG_VAL;
0563
0564        T0CON = 0;
0565        INTCON_GIEH = 1;
0566        INTCON_GIEL = 1;
0567
0568    }
0569
0570    /*****************************************************************
0571     * Function:         void InitAppConfig(void)
0572     *
0573     * PreCondition:     MPFSInit() is already called.
0574     *
0575     * Input:            None
0576     *
0577     * Output:           Write/Read non-volatile config variables.
0578     *
0579     *****************************************************************/
0580    static void InitAppConfig(void)
0581    {
0582    #if defined(MPFS_USE_EEPROM)
0583        BYTE c;
0584        BYTE *p;
0585    #endif
0586        AppConfig.MyIPAddr.v[0]      = MY_DEFAULT_IP_ADDR_BYTE1;
0587        AppConfig.MyIPAddr.v[1]      = MY_DEFAULT_IP_ADDR_BYTE2;
0588        AppConfig.MyIPAddr.v[2]      = MY_DEFAULT_IP_ADDR_BYTE3;
```

```
0589        AppConfig.MyIPAddr.v[3]      = MY_DEFAULT_IP_ADDR_BYTE4;
0590
0591        AppConfig.MyMask.v[0]        = MY_DEFAULT_MASK_BYTE1;
0592        AppConfig.MyMask.v[1]        = MY_DEFAULT_MASK_BYTE2;
0593        AppConfig.MyMask.v[2]        = MY_DEFAULT_MASK_BYTE3;
0594        AppConfig.MyMask.v[3]        = MY_DEFAULT_MASK_BYTE4;
0595
0596        AppConfig.MyGateway.v[0]     = MY_DEFAULT_GATE_BYTE1;
0597        AppConfig.MyGateway.v[1]     = MY_DEFAULT_GATE_BYTE2;
0598        AppConfig.MyGateway.v[2]     = MY_DEFAULT_GATE_BYTE3;
0599        AppConfig.MyGateway.v[3]     = MY_DEFAULT_GATE_BYTE4;
0600
0601        AppConfig.MyMACAddr.v[0]     = MY_DEFAULT_MAC_BYTE1;
0602        AppConfig.MyMACAddr.v[1]     = MY_DEFAULT_MAC_BYTE2;
0603        AppConfig.MyMACAddr.v[2]     = MY_DEFAULT_MAC_BYTE3;
0604        AppConfig.MyMACAddr.v[3]     = MY_DEFAULT_MAC_BYTE4;
0605        AppConfig.MyMACAddr.v[4]     = MY_DEFAULT_MAC_BYTE5;
0606        AppConfig.MyMACAddr.v[5]     = MY_DEFAULT_MAC_BYTE6;
0607
0608    #if defined(STACK_USE_DHCP) || defined(STACK_USE_IP_GLEANING)
0609        AppConfig.Flags.bIsDHCPEnabled = TRUE;
0610    #else
0611        AppConfig.Flags.bIsDHCPEnabled = FALSE;
0612    #endif
0613
0614    #if defined(MPFS_USE_EEPROM)
0615        p = (BYTE*)&AppConfig;
0616
0617
0618        XEEBeginRead(EEPROM_CONTROL, 0x00);
0619        c = XEERead();
0620        XEEEndRead();
0621
0622        if ( c == 0x55 )
0623        {
0624            XEEBeginRead(EEPROM_CONTROL, 0x01);
0625            for ( c = 0; c < sizeof(AppConfig); c++ )
0626                *p++ = XEERead();
0627            XEEEndRead();
0628        }
0629        else
0630            SaveAppConfig();
0631    #endif
0632    }
0633
0634    #if defined(MPFS_USE_EEPROM)
0635    static void SaveAppConfig(void)
0636    {
0637        BYTE c;
0638        BYTE *p;
0639
0640        p = (BYTE*)&AppConfig;
0641        XEEBeginWrite(EEPROM_CONTROL, 0x00);
0642        XEEWrite(0x55);
0643        for ( c = 0; c < sizeof(AppConfig); c++ )
0644        {
0645            XEEWrite(*p++);
0646        }
0647
0648        XEEEndWrite();
0649    }
0650    #endif
0651
0652    ROM char menu[] =
0653        "\r\n\r\n\r\TCP/IC RS232 Configuration\r\n\r\n"
0654
0655        "\t1: Change Board serial number.\r\n"
0656        "\t2: Change default IP address.\r\n"
0657        "\t3: Change default gateway address.\r\n"
0658        "\t4: Change default subnet mask.\r\n"
0659        "\t5: Enable DHCP & IP Gleaning.\r\n"
0660        "\t6: Disable DHCP & IP Gleaning.\r\n"
0661        "\t7: Download MPFS image.\r\n"
0662        "\t8: Save & Quit.\r\n"
0663        "\r\n"
0664        "Enter a menu choice (1-8): ";
0665
0666    typedef enum _MENU_CMD
0667    {
0668        MENU_CMD_SERIAL_NUMBER          = '1',
0669        MENU_CMD_IP_ADDRESS,
0670        MENU_CMD_GATEWAY_ADDRESS,
0671        MENU_CMD_SUBNET_MASK,
0672        MENU_CMD_ENABLE_AUTO_CONFIG,
```

```
0673        MENU_CMD_DISABLE_AUTO_CONFIG,
0674        MENU_CMD_DOWNLOAD_MPFS,
0675        MENU_CMD_QUIT,
0676        MENU_CMD_INVALID
0677    } MENU_CMD;
0678
0679    ROM char* menuCommandPrompt[] =
0680    {
0681        "\r\nSerial Number (",
0682        "\r\nDefault IP Address (",
0683        "\r\nDefault Gateway Address (",
0684        "\r\nDefault Subnet Mask (",
0685        "\r\nDHCP & IP Gleaning enabled.\r\n",
0686        "\r\nDHCP & IP Gleaning disabled.\r\n",
0687        "\r\nReady to download MPFS image - Use Xmodem protocol.\r\n",
0688        "\r\nNow running application..."
0689    };
0690
0691    ROM char InvalidInputMsg[] = "\r\nInvalid input received - Input ignored.\r\n"
0692                                 "Press any key to continue...\r\n";
0693
0694    void USARTPutROMString(ROM char* str)
0695    {
0696        BYTE v;
0697
0698        while( v = *str++ )
0699            USARTPut(v);
0700    }
0701
0702
0703    BYTE USARTGetString(char *buffer, BYTE bufferLen)
0704    {
0705        BYTE v;
0706        BYTE count;
0707
0708        count = 0;
0709        do
0710        {
0711            while( !USARTIsGetReady() );
0712
0713            v = USARTGet();
0714
0715            if ( v == '\r' || v == '\n' )
0716                break;
0717
0718            count++;
0719            *buffer++ = v;
0720            *buffer = '\0';
0721            if ( bufferLen-- == 0 )
0722                break;
0723        } while(1);
0724        return count;
0725    }
0726
0727    BOOL StringToIPAddress(char *str, IP_ADDR *buffer)
0728    {
0729        BYTE v;
0730        char *temp;
0731        BYTE byteIndex;
0732
0733        temp = str;
0734        byteIndex = 0;
0735
0736        while( v = *str )
0737        {
0738            if ( v == '.' )
0739            {
0740                *str++ = '\0';
0741                buffer->v[byteIndex++] = atoi(temp);
0742                temp = str;
0743            }
0744            else if ( v < '0' || v > '9' )
0745                return FALSE;
0746
0747            str++;
0748        }
0749
0750        buffer->v[byteIndex] = atoi(temp);
0751
0752        return (byteIndex == 3);
0753    }
0754
0755
0756
```

```
0757    MENU_CMD GetMenuChoice(void)
0758    {
0759        BYTE c;
0760
0761        while ( !USARTIsGetReady() );
0762
0763        c = USARTGet();
0764
0765        if ( c >= '1' && c < MENU_CMD_INVALID )
0766            return c;
0767        else
0768            return MENU_CMD_INVALID;
0769    }
0770
0771    #define MAX_USER_RESPONSE_LEN    (20)
0772    void ExecuteMenuChoice(MENU_CMD choice)
0773    {
0774        char response[MAX_USER_RESPONSE_LEN];
0775        IP_ADDR tempIPValue;
0776        IP_ADDR *destIPValue;
0777
0778        USARTPut('\r');
0779        USARTPut('\n');
0780        USARTPutROMString(menuCommandPrompt[choice-'0'-1]);
0781
0782        switch(choice)
0783        {
0784        case MENU_CMD_SERIAL_NUMBER:
0785            itoa(AppConfig.SerialNumber.Val, response);
0786            USARTPutString(response);
0787            USARTPut(')');
0788            USARTPut(':');
0789            USARTPut(' ');
0790
0791            if ( USARTGetString(response, sizeof(response)) )
0792            {
0793                AppConfig.SerialNumber.Val = atoi(response);
0794
0795                AppConfig.MyMACAddr.v[4] = AppConfig.SerialNumber.v[1];
0796                AppConfig.MyMACAddr.v[5] = AppConfig.SerialNumber.v[0];
0797            }
0798            else
0799                goto HandleInvalidInput;
0800
0801            break;
0802
0803        case MENU_CMD_IP_ADDRESS:
0804            destIPValue = &AppConfig.MyIPAddr;
0805            goto ReadIPConfig;
0806
0807        case MENU_CMD_GATEWAY_ADDRESS:
0808            destIPValue = &AppConfig.MyGateway;
0809            goto ReadIPConfig;
0810
0811        case MENU_CMD_SUBNET_MASK:
0812            destIPValue = &AppConfig.MyMask;
0813
0814        ReadIPConfig:
0815            DisplayIPValue(destIPValue, FALSE);
0816            USARTPut(')');
0817            USARTPut(':');
0818            USARTPut(' ');
0819
0820            USARTGetString(response, sizeof(response));
0821
0822            if ( !StringToIPAddress(response, &tempIPValue) )
0823            {
0824    HandleInvalidInput:
0825                USARTPutROMString(InvalidInputMsg);
0826                while( !USARTIsGetReady() );
0827                USARTGet();
0828            }
0829            else
0830            {
0831                destIPValue->Val = tempIPValue.Val;
0832            }
0833            break;
0834
0835
0836        case MENU_CMD_ENABLE_AUTO_CONFIG:
0837            AppConfig.Flags.bIsDHCPEnabled = TRUE;
0838            break;
0839
0840        case MENU_CMD_DISABLE_AUTO_CONFIG:
```

```
0841            AppConfig.Flags.bIsDHCPEnabled = FALSE;
0842            break;
0843
0844        case MENU_CMD_DOWNLOAD_MPFS:
0845    #if defined(MPFS_USE_EEPROM)
0846            DownloadMPFS();
0847    #endif
0848            break;
0849
0850        case MENU_CMD_QUIT:
0851    #if defined(MPFS_USE_EEPROM)
0852            SaveAppConfig();
0853    #endif
0854            break;
0855        }
0856    }
0857
0858    static void SetConfig(void)
0859    {
0860        MENU_CMD choice;
0861
0862        do
0863        {
0864            USARTPutROMString(menu);
0865            choice = GetMenuChoice();
0866            if ( choice != MENU_CMD_INVALID )
0867                ExecuteMenuChoice(choice);
0868        } while(choice != MENU_CMD_QUIT);
0869
0870    }
0871
0872
0873    #if defined(MPFS_USE_EEPROM)
0874
0875    /*****************************************************************************
0876     * Function:        BOOL DownloadMPFS(void)
0877     *
0878     * PreCondition:    MPFSInit() is already called.
0879     *
0880     * Input:           None
0881     *
0882     * Output:          TRUE if successful
0883     *                  FALSE otherwise
0884     *
0885     *****************************************************************************/
0886    #define XMODEM_SOH      0x01
0887    #define XMODEM_EOT      0x04
0888    #define XMODEM_ACK      0x06
0889    #define XMODEM_NAK      0x15
0890    #define XMODEM_CAN      0x18
0891    #define XMODEM_BLOCK_LEN 128
0892
0893    static BOOL DownloadMPFS(void)
0894    {
0895        enum SM_MPFS
0896        {
0897            SM_MPFS_SOH,
0898            SM_MPFS_BLOCK,
0899            SM_MPFS_BLOCK_CMP,
0900            SM_MPFS_DATA,
0901        } state;
0902
0903        BYTE c;
0904        MPFS handle;
0905        BOOL lbDone;
0906        BYTE blockLen;
0907        BYTE lResult;
0908        BYTE tempData[XMODEM_BLOCK_LEN];
0909        TICK lastTick;
0910        TICK currentTick;
0911
0912        state = SM_MPFS_SOH;
0913        lbDone = FALSE;
0914
0915        handle = MPFSFormat();
0916
0917        lastTick = TickGet();
0918        do
0919        {
0920           TickUpdate();
0921
0922            currentTick = TickGet();
0923            if ( TickGetDiff(currentTick, lastTick) >= (TICK_SECOND/2) )
0924            {
```

```
0925                lastTick = TickGet();
0926                USARTPut(XMODEM_NAK);
0927                LATA2 ^= 1;
0928            }
0929
0930        } while( !USARTIsGetReady() );
0931
0932
0933        while(!lbDone)
0934        {
0935            TickUpdate();
0936
0937            if ( USARTIsGetReady() )
0938            {
0939                LATA2 ^= 1;
0940                c = USARTGet();
0941            }
0942            else
0943            {
0944                continue;
0945            }
0946
0947            switch(state)
0948            {
0949            default:
0950                if ( c == XMODEM_SOH )
0951                {
0952                    state = SM_MPFS_BLOCK;
0953                }
0954                else if ( c == XMODEM_EOT )
0955                {
0956                    LATA2 = 1;
0957
0958                    MPFSClose();
0959                    USARTPut(XMODEM_ACK);
0960                    lbDone = TRUE;
0961                }
0962                else
0963                    USARTPut(XMODEM_NAK);
0964
0965                break;
0966
0967            case SM_MPFS_BLOCK:
0968                lResult = XMODEM_ACK;
0969                blockLen = 0;
0970                state = SM_MPFS_BLOCK_CMP;
0971                break;
0972
0973            case SM_MPFS_BLOCK_CMP:
0974                state = SM_MPFS_DATA;
0975                break;
0976
0977            case SM_MPFS_DATA:
0978                tempData[blockLen++] = c;
0979                if ( blockLen > XMODEM_BLOCK_LEN )
0980                {
0981                    MPFSPutBegin(handle);
0982
0983                    lResult = XMODEM_ACK;
0984                    for ( c = 0; c < XMODEM_BLOCK_LEN; c++ )
0985                        MPFSPut(tempData[c]);
0986
0987                    handle = MPFSPutEnd();
0988
0989                    USARTPut(lResult);
0990                    state = SM_MPFS_SOH;
0991                }
0992                break;
0993
0994            }
0995
0996        }
0997
0998        return TRUE;
0999    }
1000
1001    void XLCDDelay15ms(void)
1002    {
1003        DelayMs(15);
1004    }
1005    void XLCDDelay4ms(void)
1006    {
1007        DelayMs(4);
1008    }
```

```
1009
1010    void XLCDDelay100us(void)
1011    {
1012        INTCON_GIEH = 0;
1013        Delay10us(1);
1014        INTCON_GIEH = 1;
1015    }
1016
```

```
001     /*****************************************************************************
002      *                  HTTP Implementation
003      *****************************************************************************/
004     #define THIS_IS_HTTP_SERVER
005
006     #include <string.h>
007
008     #include "stacktsk.h"
009     #include "http.h"
010     #include "mpfs.h"
011     #include "tcp.h"
012
013
014     #if !defined(STACK_USE_HTTP_SERVER)
015     #error HTTP Server module is not enabled.
016     #error Remove this file from your project to reduce your code size.
017     #endif
018
019
020
021
022     #if (MAC_TX_BUFFER_SIZE <= 130 || MAC_TX_BUFFER_SIZE > 1500 )
023     #error HTTP : Invalid MAC_TX_BUFFER_SIZE value specified.
024     #endif
025
026     /*  Each dynamic variable within a CGI file should be preceeded with  this char
027     #define HTTP_VAR_ESC_CHAR       '%'
028     #define HTTP_DYNAMIC_FILE_TYPE  (HTTP_CGI)
029
030     /*
031      * HTTP File Types
032      */
033     #define HTTP_TXT        (0)
034     #define HTTP_HTML       (1)
035     #define HTTP_GIF        (2)
036     #define HTTP_CGI        (3)
037     #define HTTP_JPG        (4)
038     #define HTTP_JAVA       (5)
039     #define HTTP_WAV        (6)
040     #define HTTP_UNKNOWN    (7)
041
042
043     #define FILE_EXT_LEN    (3)
044     typedef struct _FILE_TYPES
045     {
046         char fileExt[FILE_EXT_LEN+1];
047     } FILE_TYPES;
048
049     /*
050      * Each entry in this structure must be in UPPER case.
051      * Order of these entries must match with those defined
052      * by "HTTP File Types" defines.
053      */
054     static ROM FILE_TYPES httpFiles[] =
055     {
056         { "TXT" },          // HTTP_TXT
057         { "HTM" },          // HTTP_HTML
058         { "GIF" },          // HTTP_GIF
059         { "CGI" },          // HTTP_CGI
060         { "JPG" },          // HTTP_JPG
061         { "CLA" },          // HTTP_JAVA
062         { "WAV" },          // HTTP_WAV
063         { "SWF" }           // HTTP_SWF
064     };
065     #define TOTAL_FILE_TYPES        ( sizeof(httpFiles)/sizeof(httpFiles[0]) )
066
067
068     typedef struct _HTTP_CONTENT
069     {
070         ROM char typeString[20];
071     } HTTP_CONTENT;
072
073     /*
074      * Content entry order must match with those "HTTP File Types" define's.
075      */
076     static ROM HTTP_CONTENT httpContents[] =
077     {
078         { "text/plain" },           // HTTP_TXT
079         { "text/html" },            // HTTP_HTML
080         { "image/gif" },            // HTTP_GIF
081         { "text/html" },            // HTTP_CGI
082         { "image/jpeg" },           // HTTP_JPG
083         { "application/java-vm" },  // HTTP_JAVA
084         { "audio/x-wave" },          // HTTP_WAV
```

```
085         { "application/x-shockwave-flash" }   // MACROMEDIA_FLASH
086     };
087     #define TOTAL_HTTP_CONTENTS     ( sizeof(httpContents)/sizeof(httpConetents[0])
088
089     /*
090      * HTTP FSM states for each connection.
091      */
092     typedef enum _SM_HTTP
093     {
094         SM_HTTP_IDLE,
095         SM_HTTP_GET,
096         SM_HTTP_NOT_FOUND,
097         SM_HTTP_GET_READ,
098         SM_HTTP_GET_PASS,
099         SM_HTTP_GET_DLE,
100         SM_HTTP_GET_HANDLE,
101         SM_HTTP_GET_HANDLE_NEXT,
102         SM_HTTP_GET_VAR,
103         SM_HTTP_DISCONNECT,
104         SM_HTTP_DISCONNECT_WAIT,
105         SM_HTTP_HEADER,
106         SM_HTTP_DISCARD
107     } SM_HTTP;
108
109
110     /*
111      * Supported HTTP Commands
112      */
113     typedef enum _HTTP_COMMAND
114     {
115         HTTP_GET,
116         HTTP_POST,
117         HTTP_NOT_SUPPORTED,
118         HTTP_INVALID_COMMAND
119     } HTTP_COMMAND;
120
121     /*
122      * HTTP Connection Info - one for each connection.
123      */
124     typedef struct _HTTP_INFO
125     {
126         TCP_SOCKET socket;
127         MPFS file;
128         SM_HTTP smHTTP;
129         BYTE smHTTPGet;
130         WORD VarRef;
131         BYTE bProcess;
132         BYTE Variable;
133         BYTE fileType;
134     } HTTP_INFO;
135     typedef BYTE HTTP_HANDLE;
136
137
138     typedef enum
139     {
140         HTTP_NOT_FOUND,
141         HTTP_OK,
142         HTTP_HEADER_END,
143         HTTP_NOT_AVAILABLE
144     } HTTP_MESSAGES;
145
146     /*
147      * Following message order must match with that of HTTP_MESSAGES
148      * enum.
149      */
150     static ROM char *HTTPMessages[] =
151     {
152         "HTTP/1.0 404 Not found\r\n\r\nNot found.\r\n",
153         "HTTP/1.0 200 OK\r\n\Content-type: ",
154         "\r\n\r\n",
155         "HTTP/1.0 503 \r\n\r\nService Unavailable\r\n"
156     };
157
158
159     /*
160      * Standard HTTP messages.
161      */
162     ROM BYTE HTTP_OK_STRING[]                       =           \
163                             "HTTP/1.0 200 OK\r\nContent-type: ";
164     #define HTTP_OK_STRING_LEN                                  \
165                             (sizeof(HTTP_OK_STRING)-1)
166
167     ROM BYTE HTTP_HEADER_END_STRING[]               =           \
168                             "\r\n\r\n";
```

```
169     #define HTTP_HEADER_END_STRING_LEN                              \
170                             (sizeof(HTTP_HEADER_END_STRING)-1)
171
172     /*
173      * HTTP Command Strings
174      */
175     ROM BYTE HTTP_GET_STRING[]                      =               \
176                             "GET";
177     #define HTTP_GET_STRING_LEN                                     \
178                             (sizeof(HTTP_GET_STRING)-1)
179
180     /*
181      * Default HTML file.
182      */
183     ROM BYTE HTTP_DEFAULT_FILE_STRING[]             =               \
184                             "INDEX.HTM";
185     #define HTTP_DEFAULT_FILE_STRING_LEN                            \
186                             (sizeof(HTTP_DEFAULT_FILE_STRING)-1)
187
188
189     /*
190      * Maximum nuber of arguments supported by this HTTP Server.
191      */
192     #define MAX_HTTP_ARGS       (5)
193
194     /*
195      * Maximum HTML Command String length.
196      */
197     #define MAX_HTML_CMD_LEN    (80)
198
199
200     static HTTP_INFO HCB[MAX_HTTP_CONNECTIONS];
201
202
203     static void HTTPProcess(HTTP_HANDLE h);
204     static HTTP_COMMAND HTTPParse(BYTE *string,
205                                   BYTE** arg,
206                                   BYTE* argc,
207                                   BYTE* type);
208     static BOOL SendFile(HTTP_INFO* ph);
209
210
211
212
213     /*****************************************************************
214      * Function:        void HTTPInit(void)
215      *
216      * PreCondition:    TCP must already be initialized.
217      *
218      * Input:           None
219      *
220      * Output:          HTTP FSM and connections are initialized
221      *
222      *****************************************************************/
223     void HTTPInit(void)
224     {
225         BYTE i;
226
227         for ( i = 0; i <  MAX_HTTP_CONNECTIONS; i++ )
228         {
229             HCB[i].socket = TCPListen(HTTP_PORT);
230             HCB[i].smHTTP = SM_HTTP_IDLE;
231         }
232     }
233
234
235
236     /*****************************************************************
237      * Function:        void HTTPServer(void)
238      *
239      * PreCondition:    HTTPInit() must already be called.
240      *
241      * Input:           None
242      *
243      * Output:          Opened HTTP connections are served.
244      *
245      * Side Effects:    None
246      *
247      * Overview:        Browse through each connections and let it
248      *                  handle its connection.
249      *                  If a connection is not finished, do not process
250      *                  next connections.  This must be done, all
251      *                  connections use some static variables that are
252      *                  common.
```

```
253      *
254      *****************************************************************/
255     void HTTPServer(void)
256     {
257         BYTE conn;
258
259         for ( conn = 0;  conn < MAX_HTTP_CONNECTIONS; conn++ )
260             HTTPProcess(conn);
261     }
262
263
264     /*****************************************************************
265      * Function:        static BOOL HTTPProcess(HTTP_HANDLE h)
266      *
267      * PreCondition:    HTTPInit() called.
268      *
269      * Input:           h   -   Index to the handle which needs to be
270      *                          processed.
271      *
272      * Output:          Connection referred by 'h' is served.
273      *
274      *****************************************************************/
275     static void HTTPProcess(HTTP_HANDLE h)
276     {
277         BYTE httpData[MAX_HTML_CMD_LEN+1];
278         HTTP_COMMAND httpCommand;
279         WORD httpLength;
280         BOOL lbContinue;
281         BYTE *arg[MAX_HTTP_ARGS];
282         BYTE argc;
283         BYTE i;
284         HTTP_INFO* ph;
285         ROM char* romString;
286
287         ph = &HCB[h];
288
289         lbContinue = TRUE;
290         while( lbContinue )
291         {
292             lbContinue = FALSE;
293
294             if ( !TCPIsConnected(ph->socket) )
295             {
296                 ph->smHTTP = SM_HTTP_IDLE;
297                 break;
298             }
299
300
301             switch(ph->smHTTP)
302             {
303             case SM_HTTP_IDLE:
304                 if ( TCPIsGetReady(ph->socket) )
305                 {
306                     lbContinue = TRUE;
307
308                     httpLength = 0;
309                     while( httpLength < MAX_HTML_CMD_LEN &&
310                            TCPGet(ph->socket, &httpData[httpLength++]) );
311                     httpData[httpLength] = '\0';
312                     TCPDiscard(ph->socket);
313
314                     ph->smHTTP = SM_HTTP_NOT_FOUND;
315                     argc = MAX_HTTP_ARGS;
316                     httpCommand = HTTPParse(httpData, arg, &argc, &ph->fileType);
317                     if ( httpCommand == HTTP_GET )
318                     {
319                         if ( argc > 1 )
320                         {
321                             HTTPExecCmd(&arg[0], argc);
322                             ph->fileType = HTTP_CGI;
323                         }
324
325                         ph->file = MPFSOpen(arg[0]);
326                         if ( ph->file == MPFS_INVALID )
327                         {
328                             ph->Variable = HTTP_NOT_FOUND;
329                             ph->smHTTP = SM_HTTP_NOT_FOUND;
330                         }
331                         else if ( ph->file == MPFS_NOT_AVAILABLE )
332                         {
333                             ph->Variable = HTTP_NOT_AVAILABLE;
334                             ph->smHTTP = SM_HTTP_NOT_FOUND;
335                         }
336                         else
```

```
337                         {
338                             ph->smHTTP = SM_HTTP_HEADER;
339                         }
340                     }
341                 }
342                 break;
343
344             case SM_HTTP_NOT_FOUND:
345                 if ( TCPIsPutReady(ph->socket) )
346                 {
347                     romString = HTTPMessages[ph->Variable];
348
349                     while( (i = *romString++) )
350                         TCPPut(ph->socket, i);
351
352                     TCPFlush(ph->socket);
353                     ph->smHTTP = SM_HTTP_DISCONNECT;
354                 }
355                 break;
356
357             case SM_HTTP_HEADER:
358                 if ( TCPIsPutReady(ph->socket) )
359                 {
360                     lbContinue = TRUE;
361
362                     for ( i = 0; i < HTTP_OK_STRING_LEN; i++ )
363                         TCPPut(ph->socket, HTTP_OK_STRING[i]);
364
365                     romString = httpContents[ph->fileType].typeString;
366                     while( (i = *romString++) )
367                         TCPPut(ph->socket, i);
368
369                     for ( i = 0; i < HTTP_HEADER_END_STRING_LEN; i++ )
370                         TCPPut(ph->socket, HTTP_HEADER_END_STRING[i]);
371
372                     if ( ph->fileType == HTTP_DYNAMIC_FILE_TYPE )
373                         ph->bProcess = TRUE;
374                     else
375                         ph->bProcess = FALSE;
376
377                     ph->smHTTPGet = SM_HTTP_GET_READ;
378                     ph->smHTTP = SM_HTTP_GET;
379                 }
380                 break;
381
382
383             case SM_HTTP_GET:
384                 if ( TCPIsGetReady(ph->socket) )
385                     TCPDiscard(ph->socket);
386
387                 if ( SendFile(ph) )
388                 {
389                     MPFSClose();
390                     ph->smHTTP = SM_HTTP_DISCONNECT;
391
392                 }
393                 break;
394
395             case SM_HTTP_DISCONNECT:
396                 if ( TCPIsConnected(ph->socket) )
397                 {
398                     if ( TCPIsPutReady(ph->socket) )
399                     {
400                         TCPDisconnect(ph->socket);
401                         ph->smHTTP = SM_HTTP_DISCONNECT_WAIT;
402                     }
403                 }
404                 break;
405
406             }
407         }
408     }
409
410
411     /****************************************************************
412      * Function:        static BOOL SendFile(HTTP_INFO* ph)
413      *
414      * PreCondition:    None
415      *
416      * Input:           ph      -   A HTTP connection info.
417      *
418      * Output:          File reference by this connection is served.
419     *
420      ****************************************************************/
```

```
421     static BOOL SendFile(HTTP_INFO* ph)
422     {
423         BOOL lbTransmit;
424         BYTE c;
425
426         MPFSGetBegin(ph->file);
427
428         while( TCPIsPutReady(ph->socket) )
429         {
430             lbTransmit = FALSE;
431
432             if ( ph->smHTTPGet != SM_HTTP_GET_VAR )
433             {
434                 c = MPFSGet();
435                 if ( MPFSIsEOF() )
436                 {
437                     MPFSGetEnd();
438                     TCPFlush(ph->socket);
439                     return TRUE;
440                 }
441             }
442
443             if ( ph->bProcess )
444             {
445                 switch(ph->smHTTPGet)
446                 {
447                 case SM_HTTP_GET_READ:
448                     if ( c == HTTP_VAR_ESC_CHAR )
449                         ph->smHTTPGet = SM_HTTP_GET_DLE;
450                     else
451                         lbTransmit = TRUE;
452                     break;
453
454                 case SM_HTTP_GET_DLE:
455                     if ( c == HTTP_VAR_ESC_CHAR )
456                     {
457                         lbTransmit = TRUE;
458                         ph->smHTTPGet = SM_HTTP_GET_READ;
459                     }
460                     else
461                     {
462                         ph->Variable = (c - '0') << 4;
463                         ph->smHTTPGet = SM_HTTP_GET_HANDLE;
464                     }
465                     break;
466
467                 case SM_HTTP_GET_HANDLE:
468                     ph->Variable |= (c - '0');
469
470                     ph->smHTTPGet = SM_HTTP_GET_VAR;
471                     ph->VarRef = HTTP_START_OF_VAR;
472
473                     break;
474
475                 case SM_HTTP_GET_VAR:
476                     ph->VarRef = HTTPGetVar(ph->Variable, ph->VarRef, &c);
477                     lbTransmit = TRUE;
478                     if ( ph->VarRef == HTTP_END_OF_VAR )
479                         ph->smHTTPGet = SM_HTTP_GET_READ;
480                     break;
481
482                 default:
483                     while(1);
484                 }
485
486                 if ( lbTransmit )
487                     TCPPut(ph->socket, c);
488             }
489             else
490                 TCPPut(ph->socket, c);
491
492         }
493
494         ph->file = MPFSGetEnd();
495
496         return FALSE;
497     }
498
499     /*****************************************************************
500      * Function:        static HTTP_COMMAND HTTPParse(BYTE *string,
501      *                                                BYTE** arg,
502      *                                                BYTE* argc,
503      *                                                BYTE* type)
504      *
```

```
505      * PreCondition:    None
506      *
507      * Input:           string      - HTTP Command String
508      *                  arg         - List of string pointer to hold
509      *                                HTTP arguments.
510      *                  argc        - Pointer to hold total number of
511      *                                arguments in this command string/
512      *                  type        - Pointer to hold type of file
513      *                                received.
514      *                              Valid values are:
515      *                                  HTTP_TXT
516      *                                  HTTP_HTML
517      *                                  HTTP_GIF
518      *                                  HTTP_CGI
519      *                                  HTTP_UNKNOWN
520      *
521      * Output:          HTTP FSM and connections are initialized
522      *
523      *****************************************************************/
524     static HTTP_COMMAND HTTPParse(BYTE *string,
525                                   BYTE** arg,
526                                   BYTE* argc,
527                                   BYTE* type)
528     {
529         BYTE i;
530         BYTE smParse;
531         HTTP_COMMAND cmd;
532         BYTE *ext;
533         BYTE c;
534         ROM char* fileType;
535
536         enum
537         {
538             SM_PARSE_IDLE,
539             SM_PARSE_ARG,
540             SM_PARSE_ARG_FORMAT
541         };
542
543         smParse = SM_PARSE_IDLE;
544         ext = NULL;
545         i = 0;
546
547         // Only "GET" is supported for time being.
548         if ( !memcmppgm2ram(string, HTTP_GET_STRING, HTTP_GET_STRING_LEN) )
549         {
550             string += (HTTP_GET_STRING_LEN + 1);
551             cmd = HTTP_GET;
552         }
553         else
554         {
555             return HTTP_NOT_SUPPORTED;
556         }
557
558         // Skip white spaces.
559         while( *string == ' ' )
560             string++;
561
562         c = *string;
563
564         while ( c != ' ' &&  c != '\0' && c != '\r' && c != '\n' )
565
566         {
567             // Do not accept any more arguments than we haved designed to.
568             if ( i >= *argc )
569                 break;
570
571             switch(smParse)
572             {
573             case SM_PARSE_IDLE:
574                 arg[i] = string;
575                 c = *string;
576                 if ( c == '/' || c == '\\' )
577                     smParse = SM_PARSE_ARG;
578                 break;
579
580             case SM_PARSE_ARG:
581                 arg[i++] = string;
582                 smParse = SM_PARSE_ARG_FORMAT;
583
584             case SM_PARSE_ARG_FORMAT:
585                 c = *string;
586                 if ( c == '?' || c == '&' )
587                 {
588                     *string = '\0';
```

```
589                     smParse = SM_PARSE_ARG;
590                 }
591                 else
592                 {
593                     // Recover space characters.
594                     if ( c == '+' )
595                         *string = ' ';
596
597                     // Remember where file extension starts.
598                     else if ( c == '.' && i == 1 )
599                     {
600                         ext = ++string;
601                     }
602
603                     else if ( c == '=' )
604                     {
605                         *string = '\0';
606                         smParse = SM_PARSE_ARG;
607                     }
608
609                     // Only interested in file name - not a path.
610                     else if ( c == '/' || c == '\\' )
611                         arg[i-1] = string+1;
612
613                 }
614                 break;
615             }
616             string++;
617             c = *string;
618         }
619         *string = '\0';
620
621         *type = HTTP_UNKNOWN;
622         if ( ext != NULL )
623         {
624             ext = (BYTE*)strupr((char*)ext);
625
626             fileType = httpFiles[0].fileExt;
627             for ( c = 0; c < TOTAL_FILE_TYPES; c++ )
628             {
629                 if ( !memcmppgm2ram((void*)ext, (ROM void*)fileType, FILE_EXT_LEN) )
630                 {
631                     *type = c;
632                     break;
633                 }
634                 fileType += sizeof(FILE_TYPES);
635
636             }
637         }
638
639         if ( i == 0 )
640         {
641             memcpypgm2ram((char*)arg[0], HTTP_DEFAULT_FILE_STRING,
642                                          HTTP_DEFAULT_FILE_STRING_LEN);
643             arg[0][HTTP_DEFAULT_FILE_STRING_LEN] = '\0';
644             *type = HTTP_HTML;
645             i++;
646         }
647         *argc = i;
648
649         return cmd;
650     }
```

ADDENDUM III

Summary of Component Datasheets

MICROCHIP

24AA256/24LC256

256K I^2C^{TM} CMOS Serial EEPROM

DEVICE SELECTION TABLE

Part Number	Vcc Range	Max Clock Frequency	Temp Ranges
24AA256	1.8-5.5V	400 kHz†	I
24LC256	2.5-5.5V	400 kHz‡	I, E

† 100 kHz for Vcc < 2.5V.
‡ 100 kHz for E temperature range.

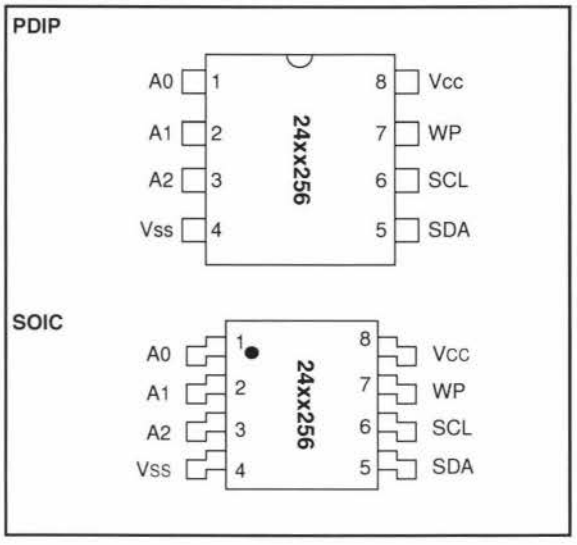
FEATURES

- Low power CMOS technology
  - Maximum write current 3 mA at 5.5V
  - Maximum read current 400 µA at 5.5V
  - Standby current 100 nA typical at 5.5V
- 2-wire serial interface bus, I²C compatible
- Cascadable for up to eight devices
- Self-timed ERASE/WRITE cycle
- 64-byte page-write mode available
- 5 ms max write-cycle time
- Hardware write protect for entire array
- Schmitt trigger inputs for noise suppression
- 100,000 erase/write cycles guaranteed
- Electrostatic discharge protection > 4000V
- Data retention > 200 years
- 8-pin PDIP and SOIC (208 mil) packages
- Temperature ranges:
  - Industrial (I): -40°C to +85°C
  - Automotive (E): -40°C to +125°C

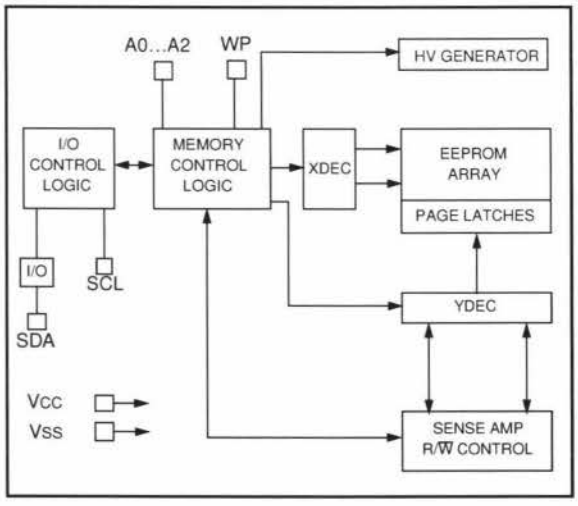
DESCRIPTION

The Microchip Technology Inc. 24AA256/24LC256 (24xx256*) is a 32K x 8 (256K bit) Serial Electrically Erasable PROM, capable of operation across a broad voltage range (1.8V to 5.5V). It has been developed for advanced, low power applications such as personal communications or data acquisition. This device also has a page-write capability of up to 64 bytes of data. This device is capable of both random and sequential reads up to the 256K boundary. Functional address lines allow up to eight devices on the same bus, for up to 2 Mbit address space. This device is available in the standard 8-pin plastic DIP, and 8-pin SOIC (208 mil) packages.

PACKAGE TYPE

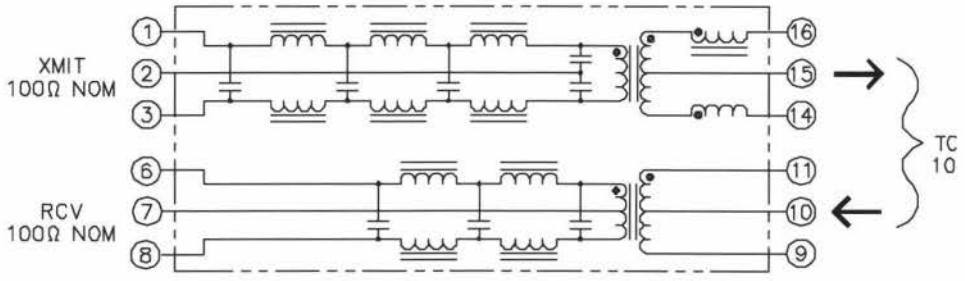


BLOCK DIAGRAM



I²C is a trademark of Philips Corporation.
*24xx256 is used in this document as a generic part number for the 24AA256/24LC256 devices.

Electrical



ELECTRICAL SPECIFICATIONS @ 25°C

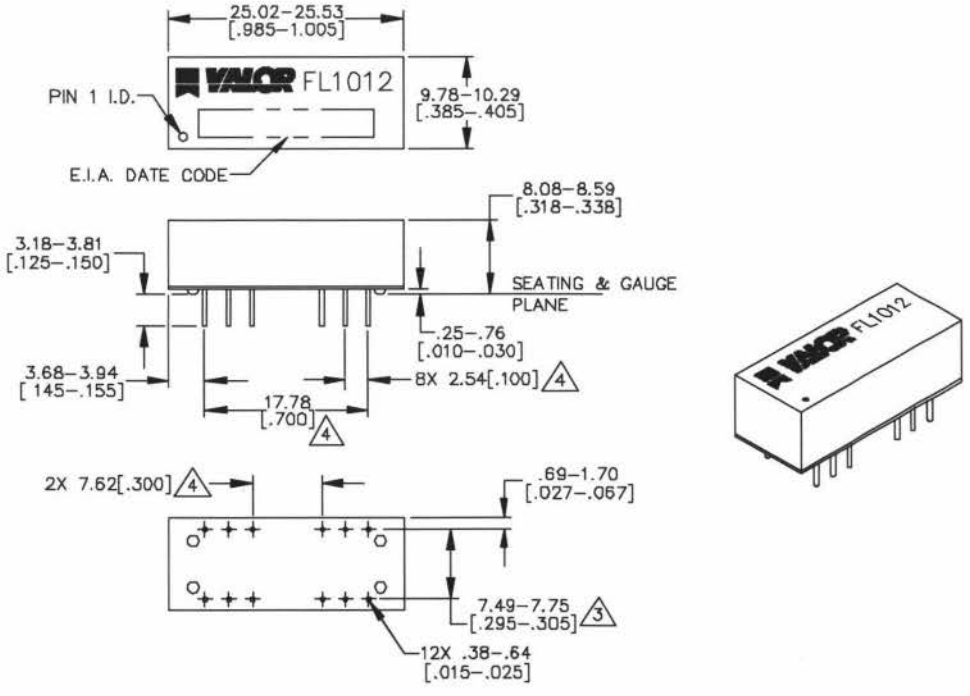
	TRANSMITTER FILTER:	RECEIVER FILTER:
TYPE	: 7 POLE,	: 5 POLE, BALANCED LOW-PASS, 100Ω
*CUT-OFF FREQ	: 17MHz ± 1MHz	: 17MHz ± 1MHz
INSERTION LOSS (1-10MHZ)	: 1.0dB MAX	: 0.8dB MAX
*ATTENUATION @ 20MHz	: 7dB MIN	: 6dB MIN
@ 25MHz	. 19dB MIN	: 14dB MIN
@ 30MHz	. 30dB MIN	: 20dB MIN
@ 40MHz	: 32dB MIN	: 31dB MIN
HIPOT	: 1500VRMS FOR 1 MINUTE PINS 1,2 & 3 TO 14,15 & 16	: 1500VRMS FOR 1 MINUTE PINS 6,7 & 8 TO 9,10 & 11
DCR	5Ω MAX (14-15)	.5Ω MAX (6-7) .1Ω MAX (9-10)
COMMON MODE REJECTION (30-100MHZ)	: 30dB MIN	: 30dB MIN
RETURN LOSS (5-10MHZ)	: 17dB MIN	:17dB MIN

(MEASURED WITH SOURCE IMPEDANCE RANGING FROM 85-100Ω)

TRANSMITTER TO RECEIVER CHARACTERISTICS

CROSS TALK ATTEN (1-10MHZ)	: BETWEEN CHANNELS : 200VRMS FOR 1 MINUTE	: 30dB MIN
*REFERENCED TO FILTER OUTPUT @ 5.0MHz.	PINS 1 & 3 TO 6 & 8 1 & 3 TO 9 & 11 6 & 8 TO 14 & 16	

Mechanical



MICROCHIP

PIC18FXX2

28/40-pin High Performance, Enhanced FLASH Microcontrollers with 10-Bit A/D

High Performance RISC CPU:

- C compiler optimized architecture/instruction set
  - Source code compatible with the PIC16C, PIC17C and PIC18C instruction sets
- Linear program memory addressing to 32 Kbytes
- Linear data memory addressing to 1.5 Kbytes

Device	On-Chip Program Memory		On-Chip RAM (bytes)	Data EEPROM (bytes)
	FLASH (bytes)	# Single Word Instructions		
PIC18F242	16K	8192	768	256
PIC18F252	32K	16384	1536	256
PIC18F442	16K	8192	768	256
PIC18F452	32K	16384	1536	256

- Up to 10 MIPs operation:
  - DC - 40 MHz osc./clock input
  - 4 MHz - 10 MHz osc./clock input with PLL active
- 16-bit wide instructions, 8-bit wide data path
- Priority levels for interrupts
- 8 x 8 Single Cycle Hardware Multiplier

Peripheral Features:

- High current sink/source 25 mA/25 mA
- Three external interrupt pins
- Timer0 module: 8-bit/16-bit timer/counter with 8-bit programmable prescaler
- Timer1 module: 16-bit timer/counter
- Timer2 module: 8-bit timer/counter with 8-bit period register (time-base for PWM)
- Timer3 module: 16-bit timer/counter
- Secondary oscillator clock option - Timer1/Timer3
- Two Capture/Compare/PWM (CCP) modules. CCP pins that can be configured as:
  - Capture input: capture is 16-bit, max. resolution 6.25 ns (TCY/16)
  - Compare is 16-bit, max. resolution 100 ns (TCY)
  - PWM output: PWM resolution is 1- to 10-bit, Max. PWM freq. @: 8-bit resolution = 156 kHz
    10-bit resolution = 39 kHz

Peripheral Features (Continued):

- Master Synchronous Serial Port (MSSP) module, Two modes of operation:
  - 3-wire SPI™ (supports all 4 SPI modes)
  - I2C™ Master and Slave mode
- Addressable USART module:
  - Supports RS-485 and RS-232
- Parallel Slave Port (PSP) module

Analog Features:

- Compatible 10-bit Analog-to-Digital Converter module (A/D) with:
  - Fast sampling rate
  - Conversion available during SLEEP
  - DNL = ±1 LSb, INL = ±1 LSb
- Programmable Low Voltage Detection (PLVD)
  - Supports interrupt on-Low Voltage Detection
- Programmable Brown-out Reset (BOR)

Special Microcontroller Features:

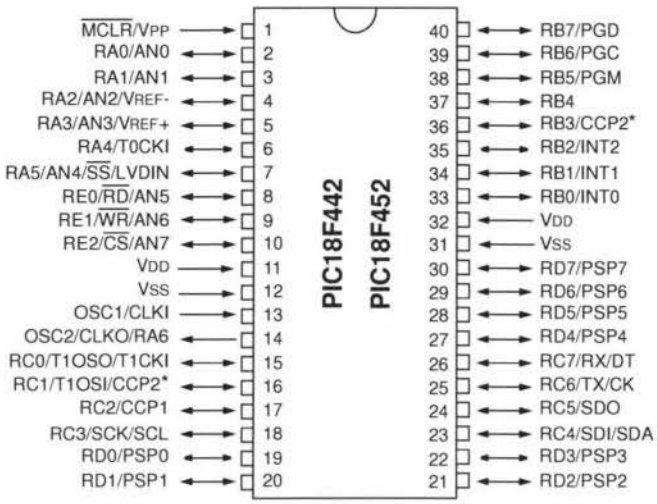
- 100,000 erase/write cycle Enhanced FLASH program memory typical
- 1,000,000 erase/write cycle Data EEPROM memory
- FLASH/Data EEPROM Retention: > 40 years
- Self-reprogrammable under software control
- Power-on Reset (POR), Power-up Timer (PWRT) and Oscillator Start-up Timer (OST)
- Watchdog Timer (WDT) with its own On-Chip RC Oscillator for reliable operation
- Programmable code protection
- Power saving SLEEP mode
- Selectable oscillator options including:
  - 4X Phase Lock Loop (of primary oscillator)
  - Secondary Oscillator (32 kHz) clock input
- Single supply 5V In-Circuit Serial Programming™ (ICSP™) via two pins
- In-Circuit Debug (ICD) via two pins

CMOS Technology:

- Low power, high speed FLASH/EEPROM technology
- Fully static design
- Wide operating voltage range (2.0V to 5.5V)
- Industrial and Extended temperature ranges

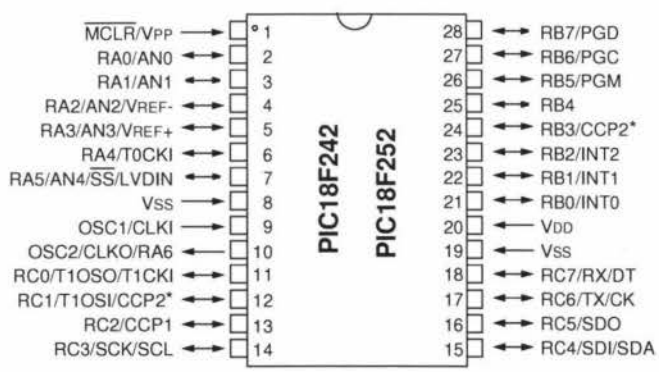
Pin Diagrams (Cont.'d)

DIP



Note: Pin compatible with 40-pin PIC16C7X devices.

DIP, SOIC



* RB3 is the alternate pin for the CCP2 pin multiplexing.

1.0 DEVICE OVERVIEW

This document contains device specific information for the following four devices:

1. PIC18F242
2. PIC18F252
3. PIC18F442
4. PIC18F452

These devices come in 28-pin and 40/44-pin packages. The 28-pin devices do not have a Parallel Slave Port (PSP) implemented and the number of Analog-to-Digital (A/D) converter input channels is reduced to 5. An overview of features is shown in Table 1-1.

The following two figures are device block diagrams sorted by pin count: 28-pin for Figure 1-1 and 40/44-pin for Figure 1-2. The 28-pin and 40/44-pin pinouts are listed in Table 1-2 and Table 1-3, respectively.

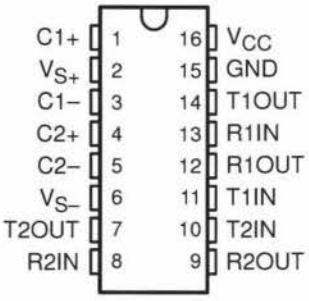
TABLE 1-1: DEVICE FEATURES

Features	PIC18F242	PIC18F252	PIC18F442	PIC18F452
Operating Frequency	DC - 40 MHz	DC - 40 MHz	DC - 40 MHz	DC - 40 MHz
Program Memory (Bytes)	16K	32K	16K	32K
Program Memory (Instructions)	8192	16384	8192	16384
Data Memory (Bytes)	768	1536	768	1536
Data EEPROM Memory (Bytes)	256	256	256	256
Interrupt Sources	17	17	18	18
I/O Ports	Ports A, B, C	Ports A, B, C	Ports A, B, C, D, E	Ports A, B, C, D, E
Timers	4	4	4	4
Capture/Compare/PWM Modules	2	2	2	2
Serial Communications	MSSP, Addressable USART	MSSP, Addressable USART	MSSP, Addressable USART	MSSP, Addressable USART
Parallel Communications	—	—	PSP	PSP
10-bit Analog-to-Digital Module	5 input channels	5 input channels	8 input channels	8 input channels
RESETS (and Delays)	POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT, OST)	POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT, OST)	POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT, OST)	POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT, OST)
Programmable Low Voltage Detect	Yes	Yes	Yes	Yes
Programmable Brown-out Reset	Yes	Yes	Yes	Yes
Instruction Set	75 Instructions	75 Instructions	75 Instructions	75 Instructions
Packages	28-pin DIP 28-pin SOIC	28-pin DIP 28-pin SOIC	40-pin DIP 44-pin PLCC 44-pin TQFP	40-pin DIP 44-pin PLCC 44-pin TQFP

SLLS047I – FEBRUARY 1989 – REVISED OCTOBER 2002

- **Meet or Exceed TIA/EIA-232-F and ITU Recommendation V.28**
- **Operate With Single 5-V Power Supply**
- **Operate Up to 120 kbit/s**
- **Two Drivers and Two Receivers**
- **±30-V Input Levels**
- **Low Supply Current . . . 8 mA Typical**
- **Designed to be Interchangeable With Maxim MAX232**
- **ESD Protection Exceeds JESD 22**
  - **2000-V Human-Body Model (A114-A)**
- **Applications**
  - **TIA/EIA-232-F**
  - **Battery-Powered Systems**
  - **Terminals**
  - **Modems**
  - **Computers**

MAX232 . . . D, DW, N, OR NS PACKAGE
MAX232I . . . D, DW, OR N PACKAGE
(TOP VIEW)



description/ordering information

The MAX232 is a dual driver/receiver that includes a capacitive voltage generator to supply EIA-232 voltage levels from a single 5-V supply. Each receiver converts EIA-232 inputs to 5-V TTL/CMOS levels. These receivers have a typical threshold of 1.3 V and a typical hysteresis of 0.5 V, and can accept ±30-V inputs. Each driver converts TTL/CMOS input levels into EIA-232 levels. The driver, receiver, and voltage-generator functions are available as cells in the Texas Instruments LinASIC™ library.

ORDERING INFORMATION

T_A	PACKAGE†		ORDERABLE PART NUMBER	TOP-SIDE MARKING
0°C to 70°C	PDIP (N)	Tube	MAX232N	MAX232N
	SOIC (D)	Tube	MAX232D	MAX232
		Tape and reel	MAX232DR	
	SOIC (DW)	Tube	MAX232DW	MAX232
		Tape and reel	MAX232DWR	
	SOP (NS)	Tape and reel	MAX232NSR	MAX232
–40°C to 85°C	PDIP (N)	Tube	MAX232IN	MAX232IN
	SOIC (D)	Tube	MAX232ID	MAX232I
		Tape and reel	MAX232IDR	
	SOIC (DW)	Tube	MAX232IDW	MAX232I
		Tape and reel	MAX232IDWR	

† Package drawings, standard packing quantities, thermal data, symbolization, and PCB design guidelines are available at www.ti.com/sc/package.

absolute maximum ratings over operating free-air temperature range (unless otherwise noted)†

Input supply voltage range, V_{CC} (see Note 1) . . . . . –0.3 V to 6 V
Positive output supply voltage range, V_{S+} . . . . . V_{CC} – 0.3 V to 15 V
Negative output supply voltage range, V_{S-} . . . . . –0.3 V to –15 V
Input voltage range, V_I: Driver . . . . . –0.3 V to V_{CC} + 0.3 V
Receiver . . . . . ±30 V
Output voltage range, V_O: T1OUT, T2OUT . . . . . V_{S-} – 0.3 V to V_{S+} + 0.3 V
R1OUT, R2OUT . . . . . –0.3 V to V_{CC} + 0.3 V
Short-circuit duration: T1OUT, T2OUT . . . . . Unlimited
Package thermal impedance, θ_{JA} (see Note 2): D package . . . . . 73°C/W
DW package . . . . . 57°C/W
N package . . . . . 67°C/W
NS package . . . . . 64°C/W
Lead temperature 1,6 mm (1/16 inch) from case for 10 seconds . . . . . 260°C
Storage temperature range, T_{stg} . . . . . –65°C to 150°C

† Stresses beyond those listed under "absolute maximum ratings" may cause permanent damage to the device. These are stress ratings only, and functional operation of the device at these or any other conditions beyond those indicated under "recommended operating conditions" is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.

NOTE 1: All voltage values are with respect to network ground terminal.
2. The package thermal impedance is calculated in accordance with JESD 51-7.

recommended operating conditions

			MIN	NOM	MAX	UNIT
V_{CC}	Supply voltage		4.5	5	5.5	V
V_{IH}	High-level input voltage (T1IN,T2IN)		2			V
V_{IL}	Low-level input voltage (T1IN, T2IN)				0.8	V
R1IN, R2IN	Receiver input voltage				±30	V
T_A	Operating free-air temperature	MAX232	0		70	°C
		MAX232I	–40		85	

electrical characteristics over recommended ranges of supply voltage and operating free-air temperature (unless otherwise noted) (see Note 3 and Figure 4)

PARAMETER		TEST CONDITIONS		MIN	TYP‡	MAX	UNIT
I_{CC}	Supply current	V_{CC} = 5.5 V, T_A = 25°C	All outputs open,		8	10	mA

‡ All typical values are at V_{CC} = 5 V and T_A = 25°C.
NOTE 3: Test conditions are C1–C4 = 1 µF at V_{CC} = 5 V ± 0.5 V.

DRIVER SECTION

electrical characteristics over recommended ranges of supply voltage and operating free-air temperature range (see Note 3)

PARAMETER			TEST CONDITIONS		MIN	TYP†	MAX	UNIT
V_{OH}	High-level output voltage	T1OUT, T2OUT	R_L = 3 kΩ to GND		5	7		V
V_{OL}	Low-level output voltage‡	T1OUT, T2OUT	R_L = 3 kΩ to GND			–7	–5	V
r_o	Output resistance	T1OUT, T2OUT	$V_{S+} = V_{S-} = 0$,	V_O = ±2 V	300			Ω
I_{OS}§	Short-circuit output current	T1OUT, T2OUT	V_{CC} = 5.5 V,	V_O = 0		±10		mA
I_{IS}	Short-circuit input current	T1IN, T2IN	V_I = 0				200	µA

† All typical values are at V_{CC} = 5 V, T_A = 25°C.
‡ The algebraic convention, in which the least positive (most negative) value is designated minimum, is used in this data sheet for logic voltage levels only.
§ Not more than one output should be shorted at a time.
NOTE 3: Test conditions are C1–C4 = 1 µF at V_{CC} = 5 V ± 0.5 V.

switching characteristics, V_{CC} = 5 V, T_A = 25°C (see Note 3)

PARAMETER		TEST CONDITIONS	MIN	TYP	MAX	UNIT
SR	Driver slew rate	R_L = 3 kΩ to 7 kΩ, See Figure 2			30	V/µs
SR(t)	Driver transition region slew rate	See Figure 3		3		V/µs
	Data rate	One TOUT switching		120		kbit/s

NOTE 3: Test conditions are C1–C4 = 1 µF at V_{CC} = 5 V ± 0.5 V.

RECEIVER SECTION

electrical characteristics over recommended ranges of supply voltage and operating free-air temperature range (see Note 3)

PARAMETER			TEST CONDITIONS		MIN	TYP†	MAX	UNIT
V_{OH}	High-level output voltage	R1OUT, R2OUT	I_{OH} = –1 mA		3.5			V
V_{OL}	Low-level output voltage‡	R1OUT, R2OUT	I_{OL} = 3.2 mA				0.4	V
V_{IT+}	Receiver positive-going input threshold voltage	R1IN, R2IN	V_{CC} = 5 V,	T_A = 25°C		1.7	2.4	V
V_{IT-}	Receiver negative-going input threshold voltage	R1IN, R2IN	V_{CC} = 5 V,	T_A = 25°C	0.8	1.2		V
V_{hys}	Input hysteresis voltage	R1IN, R2IN	V_{CC} = 5 V		0.2	0.5	1	V
r_i	Receiver input resistance	R1IN, R2IN	V_{CC} = 5,	T_A = 25°C	3	5	7	kΩ

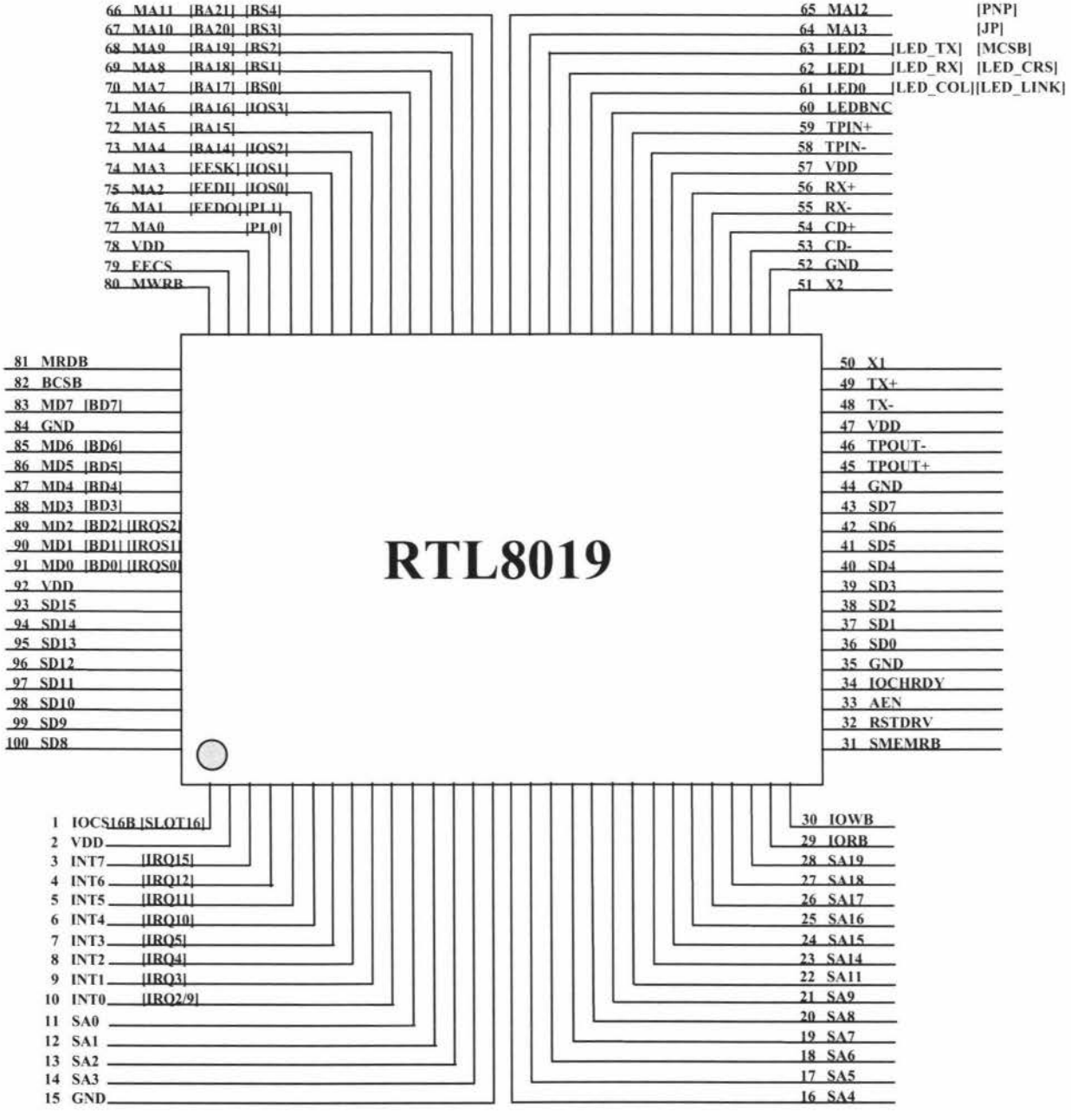
† All typical values are at V_{CC} = 5 V, T_A = 25°C.
‡ The algebraic convention, in which the least positive (most negative) value is designated minimum, is used in this data sheet for logic voltage levels only.
NOTE 3: Test conditions are C1–C4 = 1 µF at V_{CC} = 5 V ± 0.5 V.

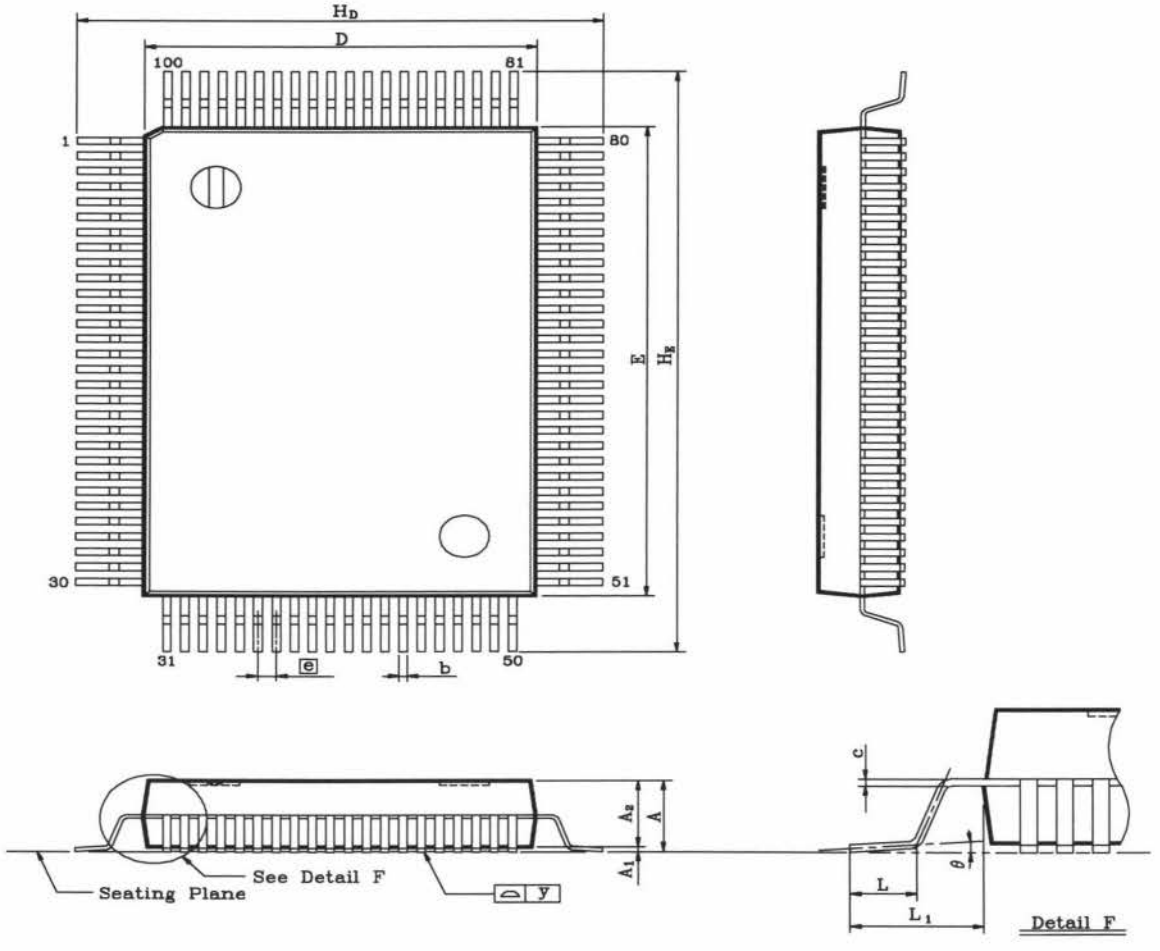
switching characteristics, V_{CC} = 5 V, T_A = 25°C (see Note 3 and Figure 1)

PARAMETER		TYP	UNIT
$t_{PLH(R)}$	Receiver propagation delay time, low- to high-level output	500	ns
$t_{PHL(R)}$	Receiver propagation delay time, high- to low-level output	500	ns

NOTE 3: Test conditions are C1–C4 = 1 µF at V_{CC} = 5 V ± 0.5 V.

3. PIN CONFIGURATION





Symbol	Dimension in mil			Dimension in mm		
	Min	Typ	Max	Min	Typ	Max
A	106.3	118.1	129.9	2.70	**3.00**	3.30
A1	4.3	20.1	35.8	0.11	**0.51**	0.91
A2	102.4	112.2	122.0	2.60	**2.85**	3.10
b	7.1	11.8	16.5	0.18	**0.30**	0.42
c	1.6	5.9	10.2	0.04	**0.15**	0.26
D	541.3	551.2	561.0	13.75	**14.00**	14.25
E	777.6	787.4	797.2	19.75	**20.00**	20.25
□	19.7	25.6	31.5	0.50	**0.65**	0.80
HD	726.4	740.2	753.9	18.45	**18.80**	19.15
HE	962.6	976.4	990.2	24.45	**24.80**	25.15
L	39.4	47.2	55.1	1.00	**1.20**	1.40
L1	88.6	94.5	104.3	2.25	**2.40**	2.65
y			3.9			0.10
	0		12	0		12

Note:

1. Dimension D & E do not include interlead flash.
2. Dimension b does not include dambar protrusion/intrusion.
3. ***Controlling dimension: Millimeter***
4. General appearance spec. should be based on final visual inspection spec.

TITLE : 100L QFP (14x20 mm**2) FOOTPRINT 4.8 mm PACKAGE OUTLINE DRAWING			
LEADFRAME MATERIAL:			
APPROVE		DWG NO.	
		REV NO.	
		SCALE	
CHECK	Ricardo Chen	DATE	
		SHT NO.	1 OF
REALTEK SEMI-CONDUCTOR CO., LTD			