Embedded Room-By-Room Heating Control System

Mr Michael John Curry

EO320

Mr Chris S Knight

24 April 2007

Final year report submitted in partial fulfilment of the requirements for the degree of:
Bachelor of Engineering (Honours) in Electronic & Computer Engineering

Disclaimer

I hereby certify that the attached report is my own work except where otherwise indicated. I have identified my sources of information; in particular I have put in quotation marks any passages that have been quoted word-for-word, and identified their origins.

Signed ……

Date ……

Abstract

This project looks at an implementation of an ethernet local area network using embedded microchip 'PIC' microcontrollers, whereby each room in a building is thermally controlled by an embedded ethernet node. Each room node is then in turn controlled by a central control node which also hosts an embedded web server, allowing a user to easily control the temperature of each room through an internet browser.

This dissertation accompanies the hardware and software implementation of two embedded microcontroller ethernet nodes, whereby each can manipulate a binary heater output, to meet a user specified target temperature by reading a thermal sensor. One of these nodes has also been configured to be able to 'discover' the other node on the network and send it a new target temperature, through an HTML interface (aka. web page).

This project covers both the design of the hardware and software, which must function closely together for an embedded solution to work.

Contents

List of Figures

List of Tables

Glossary

Term	Definition
\n or lf	ASCII character for line feed (hex code 0x0d)
\r or cr	ASCII character for carriage return (hex code 0x0a)
AJAX	Asynchronous JavaScript & XML
ASCII	American Standard Code for Information Interchange
CAN	Controller Area Network
CGI	Common Gateway Interface
CSS	Cascading Style Sheet
E^2PROM	See *EEPROM*
EEPROM	Electronically Erasable Programmable Read Only Memory
EMI	Electro-Magnetic Interference
HTML	Hyper-Text Markup Language
HTTP	Hyper-Text Transfer Protocol
IC	Integrated Circuit
ICD	In-Circuit Debugger / Debugging
ICSP	In-Circuit Serial Programming
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
ISO	International Standards Organisation
MCC18	Microchip C compiler for 18 series PICs
MCU	Microcontroller Unit
NIC	Network Interface Controller
OSI	Open Systems Interconnect
PCB	Printed Circuit Board
RFC	Request for comments
RM	Reference Model
RTC	Real Time Clock
RX	Receive / Receiver

Term	Definition
SPDIP	Slim Plastic Dual In-line Package
SPI	Serial Peripheral Interface
TCP	Transmission Control Protocol
TX	Transmit / Transmitter

Acknowledgements

I would like to firstly thank the dedication of the microchip support team for all their hard work in trying to solve the various problems I had with their TCP/IP stack.

Secondly, I would like to thank the staff of the University of Brighton for their support throughout the project.

1 Introduction

With the ever increasing amounts of integration of microprocessors and microcontrollers in electrical goods in today's world and the ever growing adoption of cheap 'smart home' solutions, it makes sense to integrate some form of processing power into the heaters in each room of an house. With the popularity of ethernet (IEEE 802.3) as an home networking medium, it makes sense to use this technology. There are several ways this could be achieved, even in an home without any wired ethernet cables. Two existing technologies are WiFi (IEEE 802.11) and ethernet over mains (known by several names by different manufactures). The second of these two would be very easy to use, as it provides a RJ-45 socket for the network device to use, whereas the second option would involve having to integrate the WiFi receiver / transmitter into each node, which is beyond the scope of this project.

A project involving several PIC microcontrollers talking to each other over ethernet has not been attempted before by the university, although one PIC acting as a standalone web server, with controls for various input and output transducers, has been completed successfully but used a different network interface controller (NIC). However, since that project was completed, the microchip TCP/IP stack has gone through several revisions with the latest being v4.00RC (beta), which was supplied through support.microchip.com whilst having trouble trying to get various parts of the stack to work properly.

Room-by-room temperature control has been chosen as the purpose for these interconnected microcontrollers, due to the current popularity of energy saving, 'green' technology. By controlling the temperature of each room individually, rather than a single thermostat for the whole building or floor, can significantly save energy bills and with the added convenience of being controlled through one central website would allow the busiest of users to only heat the rooms needed, at the times when they are needed. If this website is also made available to the internet through a permanently connected home broadband connection, the home owner would then have increased control over the temperature in their rooms, by being able to control

their temperature from anywhere in the world. An example of this is the scenario of when the user extends or shortens a planned holiday and can simply connect to their house to update its heating pattern to adapt to their latest needs.

2 Project Aims, Objectives and Technical Overview

2.1 Aims

To develop an embedded web based programmable room-by-room heating control system, using an ethernet local area network

2.2 Objectives

- Research temperature measurement and control
- Research mains heater control
- Research previous final year students' ethernet based projects and take heed of the problems they encountered
- Research and utilise in-circuit debugging (ICD) and in-circuit serial programming (ICSP)
- To research & utilise the C18 microchip compiler
- Research transmission control / internet protocol (TCP/IP)
- Utilise Microchip's TCP/IP stack
- Research and utilise the dynamic host control protocol (DHCP) for automatic internet protocol (IP) address allocation
- Research the hyper-text transfer protocol (HTTP) and implement embedded web server to host an HTML front end
- To design the system in a modular form
- To design, build and test the hardware and software to form a solution to the project aim

2.3 Technical Overview

This is an embedded system, to perform the heating part of a 'smart home.' All the functions will be controlled and monitored by a central node that monitors room nodes and sets the required temperature set-point. It communicates with the room nodes as a TCP client on the ethernet network. The central node also provides an embedded web server which the user can log-in to from within the house's LAN or

potentially the WWW to control the system. The web page(s) will allow monitoring of each room temperature and the setting of the target temperature. For this prototype model, there will be two rooms with the potential to have more rooms added to facilitate a larger house. Each room node will measure the temperature level and drive the ‘heater’ in such a way as to achieve the required set-point. The room node will act as a TCP server that will return values to the central node and receive the required set-point. For the exhibition of the project, each room will be a suitably sized cardboard box with a 40W light bulb as the heater.

3 Background and Research

3.1 Embedded Web Servers

An embedded server is a device that is small in size and cost, capable of communicating with a data network. Primarily, embedded systems are used in process control, whereby the user may request an environment to have certain properties and the system would need to manage peripherals connected to it to best fit the user's requirements. With the wide-spread use of the internet across the world, if the control system has an embedded web server built in, the user has the power to adjust variables potentially from anywhere in the world.

However, the main disadvantages of using an embedded system to host a web server, rather than a PC are the limited processing power and memory size. Nevertheless, with the rapid increase of silicon density and ongoing microprocessor / microcontroller technology, it is foreseeable that these problems can be overcome although there would be significant impact on the price of the overall product, when using a high specification system.

3.2 Review of previous embedded-ethernet projects

The project that is most similar to this one was carried out two years ago by Kristyan Osborne, with the title 'Embedded web server for remote control of sensors over the internet.' The main differences between that project and this are as follows:

Item \ Project Student	Kristyan Osborne	Michael Curry
Network controller	Realtek RTL8019AS	Microchip ENC28J60
Number of nodes	1	At least 2
Microchip TCP/IP stack version	v2.20.04.01	v4.00RC (beta)
Microchip PIC	p18f452	p18f4685

Table 1 - Comparison of Projects

One of the major potential set backs that Kristyan faced, was the need to make an adaptor board for the Realtek network interface controller (NIC), since it is only available in surface-mount form. This problem was avoided for this project, by using a microchip ENC28J60 SPI ethernet controller (which was still in development at the

time of Kristyan's project) in a slim plastic dual in-line package (SPDIP), making it easy to use for a prototype on a solder-less breadboard. Since the latest versions of the microchip TCP/IP stack support this chip, the substitution should be seamless. However, the demo board (PICDEM.net) that was previously used is now out of date, with the PICDEM.net 2 board being its successor. This new board contains both the ENC28J60 as well as a PIC18F97J60 – one of the new surface mount PICs with an integrated ethernet controller. Unfortunately, it is not within the project budget to purchase one of these boards, and all code will need to be run on a purpose built prototype node.

There have also been several advancements to the microchip TCP/IP stack since Kristyan attempted the project, including fixes for bugs (such as the DHCP lease time issue that was encountered previously) and new modules such as NetBIOS, SMTP, Telnet and templates for generic TCP clients and servers. However, since the version being used is a pre-release (released by microchip support, for use in this project whilst trying to get v3.75 to work), there were also new software bugs and unfinished new modules encountered.

3.3 In-Circuit Debugging and Programming

Certain microchip peripheral interface controllers (PICs) have an in-circuit serial programming (ICSP) capability, which allows programmers to change the programme memory of the device, without physically removing the microcontroller unit (MCU) from the target board. This can be achieved using only 5 wires – Vdd, Vss, Vpp, PGD and PGC. Vdd allows the circuit programmer module to either detect whether there is power on the target board or, if a 5 volt circuit with low current consumption, power the target board. The Vdd line may be omitted, but will cause errors to be triggered in the integrated development environment (IDE), since the ICSP module will report the target board is not powered. Vss provides a ground reference / return. Vpp is used to hard-reset the target board and is required in the PIC programming routine. PGD and PGC are used to clock data to and from the PIC allowing the programme and EEPROM data and to be written, read and verified.

In-circuit debugging (ICD) uses the same 5 lines as ICSP, but allows the programme developer to step through programmes running on the target device and directly read and write to registers on the MCU, thus making debugging code very easy. However,

when the PIC is running in debug mode, the operation speed is reduced significantly. In the case of this project, making running the TCP/IP stack in debug mode impossible. Nevertheless, portions of code may be loaded into a separate project for debugging purposes, should the reason for a routine not working become unclear.

3.4 Microchip 'MCC18' C compiler

Since the microchip TCP/IP stack is written in C, a C compiler must be used to convert the programme to machine code. Fortunately, there is a free student edition of the microchip C compiler for the 18 series of PIC (a.k.a. MCC18). The only differences to a C compiler for the PC are the 'additional' library functions.

- Not all the functions usually available are present
- Some of the functions that are present have more limited functionality than their counterparts on PC C compilers
- Data-types must be handled correctly, especially when data-passing between functions, e.g. when comparing a string, two variables must be passed – strcmp(str1,str2); and not strcmp(str1,"teststring");

3.5 ISO-OSI 7 layer reference model & TCP/IP stack model

The industry standards organisation open systems interconnect (a.k.a. ISO-OSI) seven layer reference model. The seven layers are split up as follows:

- Layer 7: Application layer – this is the layer to which the 'user' interfaces to.
- Layer 6: Presentation layer – this layer transforms data received from lower layers into a form usable by the application layer
- Layer 5: Session layer – this layer controls the establishment and termination of connections (or sessions) between clients.
- Layer 4: Transport layer – this allows reliable transmission of messages to occur, relieving upper layer protocols of the task.
- Layer 3: Network layer – this allows variable lengths of message and segmentation / desegmentation of oversized packets to be transferred from source to destination over one or more networks. Routing of packets between networks uses this layer's addressing.

- Layer 2: Data link layer – this defines the encoding / decoding of the data to be passed to and from the physical layer. It also involves some form of node addressing facilitating the transfer of data from one node to another. In the case of 10Base-T ethernet, the data is encoded in Manchester code and MAC addresses are used to identify each node.
- Layer 1: Physical layer – this defines the physical and electrical specifications of the network medium for which data signals are transmitted and received through.

Based roughly on [WP1] and [STA07] descriptions

The TCP/IP 5 layer model is based on the ISO-OSI model, with layers 4 & 5 and layers 6 & 7 combined. However, there are also slight overlaps between the layers
Side by side, the two models roughly compare thusly:

<table>
<tr><th colspan="2">ISO-OSI RM</th><th colspan="2">TCP/IP</th></tr>
<tr><td>7</td><td>Application layer</td><td rowspan="2" colspan="2">Application layer</td></tr>
<tr><td>6</td><td>Presentation layer</td></tr>
<tr><td>5</td><td>Session layer</td><td rowspan="2" colspan="2">Transport layer</td></tr>
<tr><td>4</td><td>Transport layer</td></tr>
<tr><td>3</td><td>Network layer</td><td colspan="2">Internetworking layer</td></tr>
<tr><td>2</td><td>Data link layer</td><td>Data link layer</td><td rowspan="2">A.k.a. Network Access layer</td></tr>
<tr><td>1</td><td>Physical layer</td><td>Physical layer</td></tr>
</table>

Table 2 - Comparison of ISO-OSI and TCP/IP layers

3.6 Transmission Control Protocol / Internet Protocol suite

The portions on TCP and IP in this section have been adapted from another dissertation written by the author, for the module EO306 Data Communications.

3.6.1 Internet Protocol Version 4 (IPv4, OSI layer 3) – RFC 791

This is a data-oriented protocol, which is the dominant network layer protocol used on the internet and may be used on any packed switched network. Since it is data oriented (opposed to connection-oriented), none of the data sent is guaranteed to

arrive at all, in the correct order or error free. This is where an upper layer protocol such as TCP or UDP must be used.

Each node on an IP network has a unique 32 bit address assigned to it. This is usually divided up into four bytes and it is common practise to be displayed as a dotted decimal quad. Other representations include dotted-hexadecimal, dotted-octal, raw hexadecimal, raw decimal and raw octal.

Out of the possible 4,294,967,296 (232) addresses potentially available, there are certain ranges (or blocks) that are reserved for specific use. Examples of this are for private networks, local loop-back and broadcast.

Based roughly on [WP2] and [STA07] descriptions

3.6.2 Transmission Control Protocol over IPv4 (TCP, OSI layers 4&5) – RFC 793

This is a connection-oriented protocol, which allows network nodes to establish a reliable connection with one another using a method known as socketing. Socketing allows multiple servers to run on a single node, by assigning the different services unique ports.

Before data from an higher level protocol can be sent, TCP must first establish a connection with the other end, using a method known as the 'three way handshake.' First, the client will send a packet with the SYN flag set. If the server receives this packet, it will reply with a packet with the SYN and ACK flags set. If the client receives this packet successfully, it will respond by sending a packet with just the ACK flag set. The connection is now established and the higher level protocol may now send data.

Since the protocol is connection oriented, all the packets sent must be acknowledged. If an acknowledgement of a packet is not received, the sender will timeout and resend the packet.

Once all the data has been sent from the higher level protocol, the link must be closed. One method is by using a three way handshake. In this case, the flags set in order are FIN, FIN+ACK, ACK.

Another method is a four way handshake, where one end sends a packet with the FIN flag set, the other end replies with ACK. At this point, only one way communication

may take place. Once the other end is ready to terminate the link, it will send a packet with the FIN flag set, to which the other end will reply ACK.

Based roughly on [WP3] and [STA07] descriptions

The key advantages of TCP over IP alone are:

- Multiple servers running behind one node (socketing)
- Data from higher level protocols gets checked for errors
- All packets are acknowledged
    - Lost packets get retransmitted
- Segmented data easily put back in the correct order

3.6.3 User Datagram Protocol (UDP, OSI Layer 4) – RFC 768

This protocol is designed to provide minimum-latency data transfer. It uses the same socketing method as TCP to allow multiple connections to the same node. However, to reduce latency, the packets sent are not acknowledged therefore making any data being transmitted over an unreliable medium prone to packet loss and packet disordering, although the data that is received will be accurate since the checksum in the UDP header is for the data as well as the header.

The key advantages of using UDP over IP alone are:

- Multiple servers running behind one node (socketing)
- Data from higher level protocols gets checked for errors

The key advantages of using UDP in place of TCP are:

- Reduced latency (no three way handshake to initiate connection and no acknowledgements of individual packets)

The key disadvantages of using UDP in place of TCP are:

- Since there is no built in packet ordering strategy, if the order of the data is significant, an higher level protocol must deal with this
- Since there is no packet acknowledgement, if packet loss is not acceptable, an higher level protocol must deal with this

3.7 Microchip TCP/IP stack

The Microchip TCP/IP stack is a modular, highly customisable piece of code that allows a TCP/IP stack to run on a PIC, with just the services required.

High level modules available in v4.00RC of the stack include:

- Announce (Announce.c)
- Address Resolution Protocol (ARP.c)
- Dynamic Host Control Protocol (DHCP.c)
- Domain Name System (DNS.c)
- File Transfer Protocol (FTP.c)
- Generic TCP Client (GenericTCPClient.c)
- Generic TCP Server (GenericTCPServer.c)
- Hypertext Transfer Protocol (HTTP.c)
- Internet Control Messaging Protocol (ICMP.c)
- NetBIOS Name System (NBNS.c)
- Simple Mail Transfer Protocol (SMTP.c)
- Simple Network Management Protocol (SNMP.c)
- Telnet (Telnet.c)
- Trivial File Transfer Protocol (TFTP.c)

These modules in turn use the following modules to access the network:

- Transport Control Protocol (TCP.c)
- Internet Protocol (IP.c)
- User Data Protocol (UDP.c)
- Serial Line Internet Protocol (SLIP.c)

There are also other modules, used for the control of external peripherals, including the ethernet controller and helper functions to allow the stack to operate efficiently:

- General Delay Routines (Delay.c)
- Medium Access Control Layer for Microchip ENC28J60 (ENC28J60.c)
- Medium Access Control Layer for Microchip PIC18F97J60 (ETH97J60.c)
- Helper Functions for the TCP/IP stack (Helpers.c)

- Data I2C EEPROM Access Routines (I2CEEPROM.c)
- LCD Access Routines (LCDBlocking.c)
- Microchip File System Access API (MPFS.c)
- Medium Access Control Layer for Realtek RTL8019AS (RTL8019AS.c)
- Data SPI EEPROM Access Routines (SPIEEPROM.c)
- TCP/IP Stack Manager (StackTsk.c)
- Tick Manager, for loose timekeeping (Tick.c)
- UART access routines (UART.c)

The modules from the stack that will be used for this project (with the exception of the helper functions) have a software structure as follows:

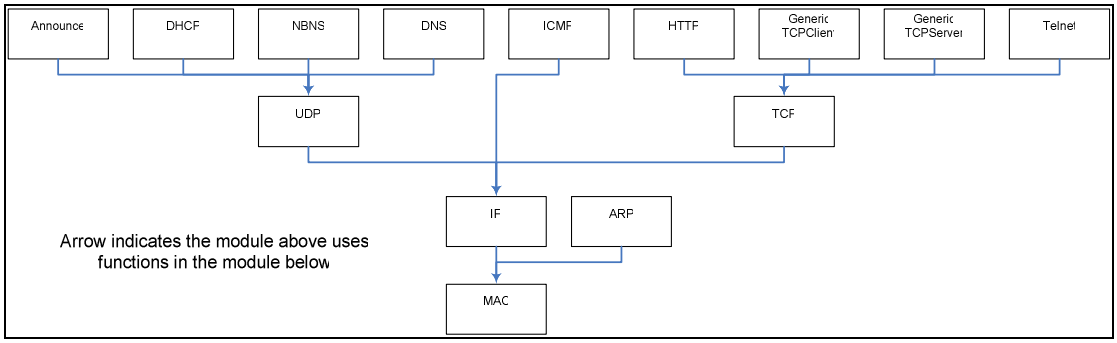
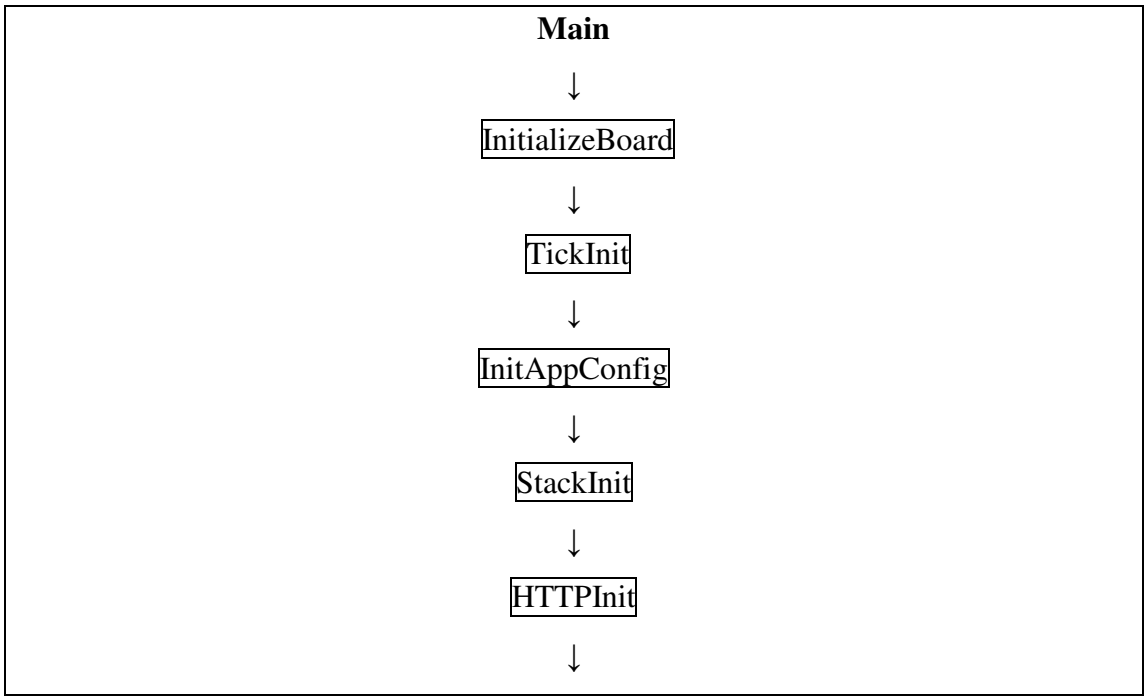


Figure 1 - Stack structure chart

However, the actual flow of the software is sequential, using the following sequence:



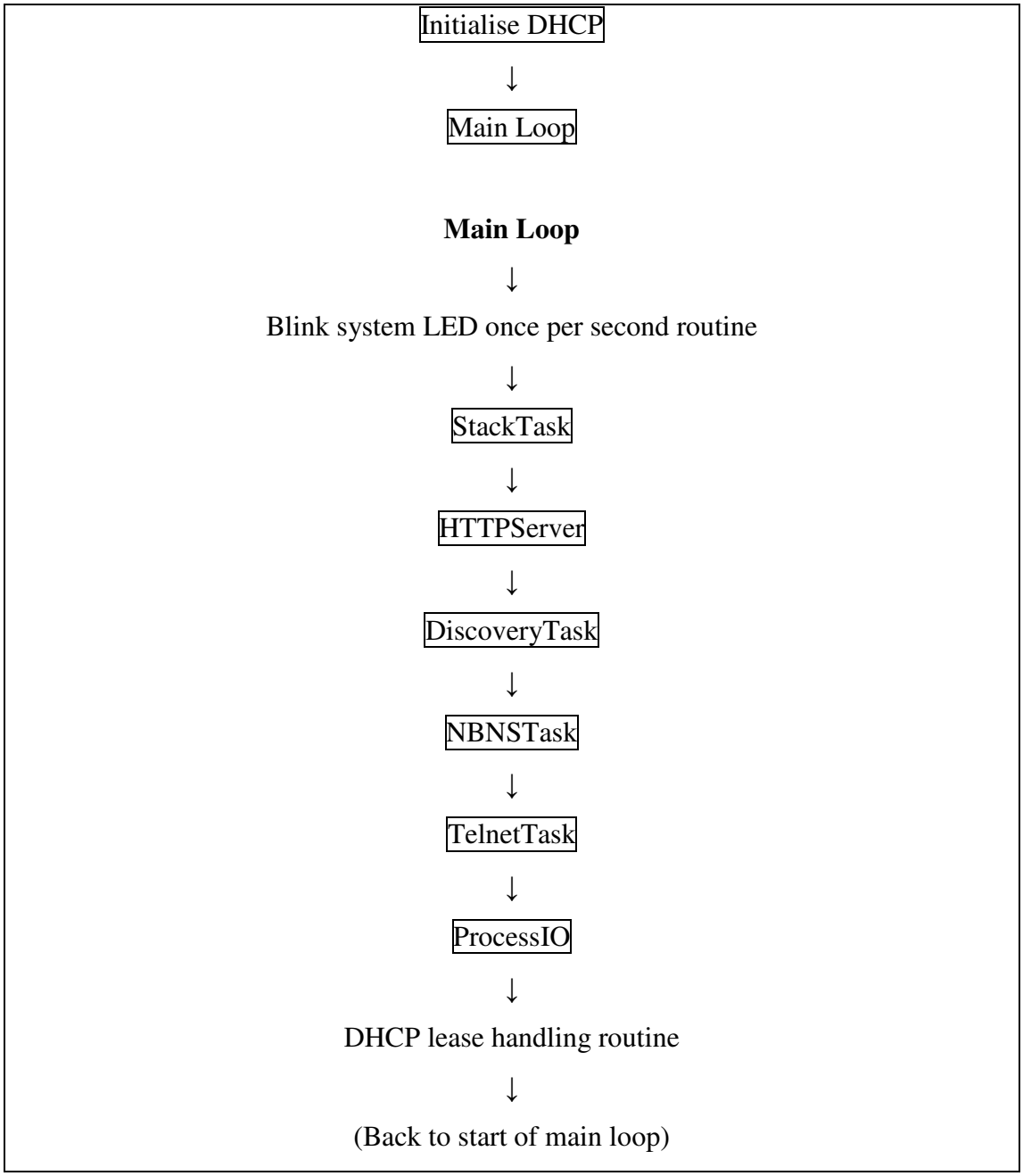


Figure 2 - Stack flowchart

With the stack written in this modular fashion, the task of creating additional functions to do tasks specific to this project is made easy. By basing code on the GenericTCPServer module, a fully customised TCP socket can be created, using a suitable free port. To include this within the running of the stack, the function call may be placed after the ProcessIO task.

4 Design of artefact

4.1 Possible room-node network mediums

4.1.1 X.10

Advantages	Disadvantages
• Can be modulated onto mains wiring • Hence potentially cheaper set-up costs • Other home-automation products use this medium allowing potential integration to an existing home automation network	• If adopted on a large scale, neighbours on the same electricity phase would also be part of the network (privacy issues for the mains wiring option) • Cabling costs if mains not used • Slow connections • Protocol translation / extra coding involved to use in conjunction with TCP/IP stack

Table 3 - X.10 advantages / disadvantages

4.1.2 CAN (Controller Area Network)

Advantages	Disadvantages
• Network protocol specifically designed for real-time automation tasks (e.g. cars)	• Protocol translation / extra coding involved to use in conjunction with TCP/IP stack • Additional costs of wiring for all users

Table 4 - CAN advantages / disadvantages

4.1.3 IEEE 802.3 Ethernet (wired)

Advantages	Disadvantages
• Very popular network medium • All modern computers in	• Additional costs of wiring for the majority of users, unless an

production have out-of-the-box access to this network medium • Many ‘smart-homes’ are pre-wired with ethernet cabling	additional ethernet over mains module was bought.

Table 5 - Ethernet advantages / disadvantages

4.1.4 IEEE 802.11 Ethernet (WiFi)

Advantages	Disadvantages
• No costs involved in cabling • Hardware readily available	• Poor signal coverage over a large area • Unreliable connections in poor signal areas • At least one wired to wireless bridge must be used to allow nodes to communicate • More complex setting up required of the end user / installation technician • A lot more time required to interface to project (making this outside of this project’s scope)

Table 6 - WiFi advantages / disadvantages

4.1.5 Conclusion

Since the microchip TCP/IP stack is written for ethernet and the popularity of ethernet as a network medium, wired ethernet was chosen to be used. It is also the case that many newly built smart homes are being pre-networked with wired ethernet thus making installation of a product of this sort very simple.

4.2 Physical ethernet connections

4.2.1 Specification outlined within ENC28J60 data sheet

Within the datasheet for the ENC28J60 is the following circuit diagram regarding external connections for the chip:

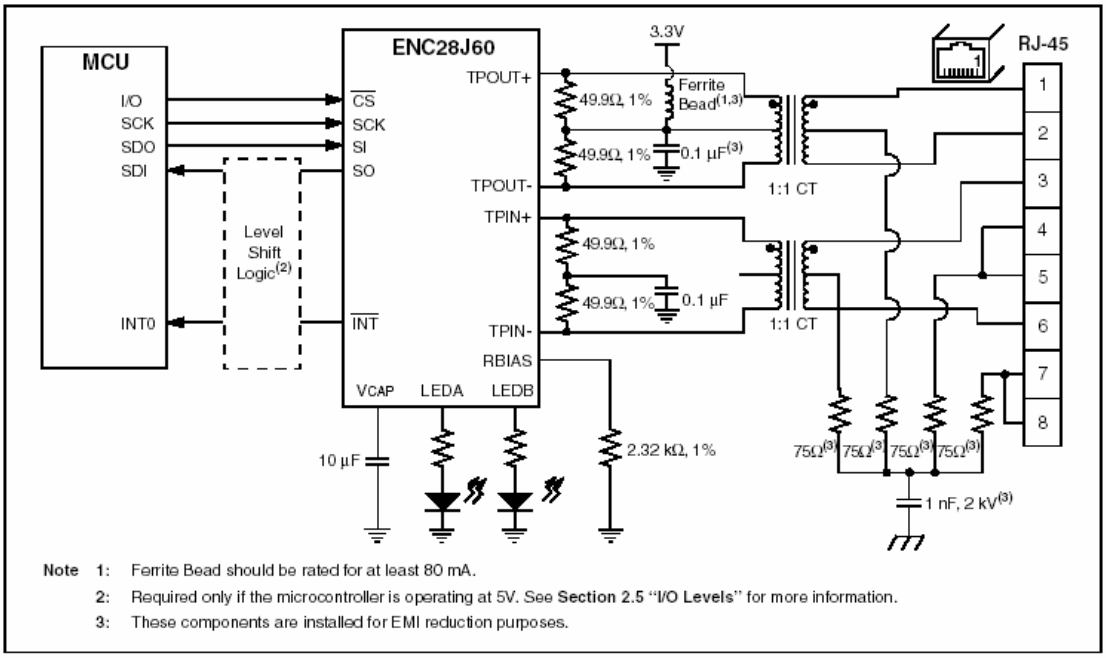


Figure 3 - ENC28J60 Ethernet Termination and External Connections [MC1]

After some quick initial research, it became apparent that there was an array of RJ-45 sockets with integrated magnetics (i.e. transformers / inductors) and this arrangement was known as 'tab-down'. However, it also became clear after looking in the RS and Farnell catalogues that these components had several variations, namely with different turns ratios on the transmit (TX) and receive (RX) transformers. It also became apparent that these parts may be difficult to source since despite having the lines in the catalogues, when checking availability online they showed up as being out of stock. Nevertheless, a line that was in stock and of the correct specification was eventually found and the parts were ordered and promptly delivered.

The only other miscellaneous part from this circuit was the ferrite bead. After a quick search within the microchip user-support forums, it became clear that the inductance value of the ferrite bead was not at all critical since it was just used to prevent electro-magnetic interference (EMI).

4.3 Selection of a suitable microcontroller

Since microchip offer a free sampling service for developers, the only constricting points to consider are the following:

- Must be an 18 series PIC since the TCP/IP stack is written in C, and the microchip C compiler is written for the 18 series PICs
- Must have sufficient programme memory to store the TCP/IP stack, web pages (for the central server) and additional modules needed for the task in hand
- Internal EEPROM would be an advantage for storing variables that are required to be retained in the event of power loss
- Must be able to run a 3.3v, since having multiple voltages in the circuit would require a more complex power supply, hence increasing costs

4.4 Thermal sensing

Since components used to accurately sense temperature, such as thermocouples, rarely have a linear relationship between the measured result and the actual temperature, a decision was made to use a pre-made, ready linearised thermally sensing integrated circuit (IC) with a serial digital output. Since the ENC28J60 ethernet controller uses an SPI bus, it made sense to choose a thermal sensor that uses the same technology. The microchip TC77 is an example of this, with free samples available from the microchip website for prototyping purposes. However, it is only available in a surface mount form factor, causing a dual-in-line adaptor board to be sought to allow usage of this chip on the solder-less prototype board being used. Nevertheless, since the final design for a room node is intended to be on PCB, a chip of this form would not a problem.

4.5 Mains switching

Since the circuit will be running on 3.3v, a relay with a suitable coil voltage rating and a contact rating of 220-240 volts must be chosen. It is also important to remember that the final product would be intended to power mains heaters, which have high current consumptions (up to 13 amps) hence a relay for the final product should have a power rating of at least 3.1kW.

During testing of a 5 volt relay that had been acquired, it became apparent that although it would be suitable to be powered using a 3v supply, the PIC would not be able to supply enough current to the coil (see section 11.1 (Appendix A) for experiment results), so a switching transistor circuit had to be designed to buffer the output and allow the relay to be driven correctly without causing damage to the PIC. Hence the following "emitter follower" buffer circuit was designed:

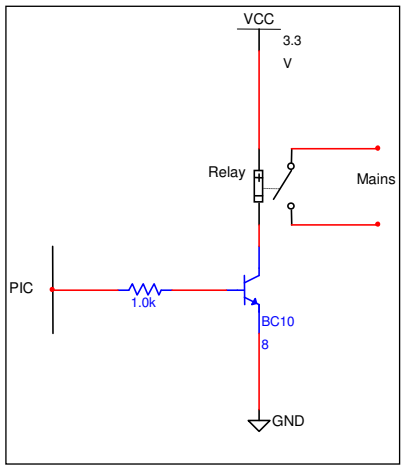


Figure 4 - Collector follower transistor buffer

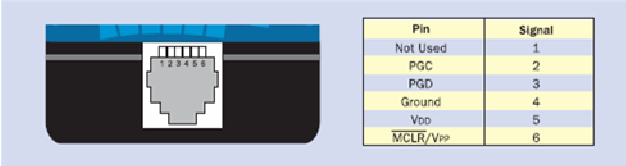
The PIC could now be used for the switching the relay.

4.6 Miscellaneous

There are certain other parts of the microchip which must also be dealt with, that have not been mentioned yet in this section. The most important area is the power supply. Both the PIC18LF4685 and the ENC28J60 have multiple pins that are used to supply power and they must all be connected (i.e. assuming connecting only one Vdd line and one Vss line to the chip will not work).

4.7 In-circuit serial programming / debugging

Using the microchip ICD2 in-circuit serial programmer / in-circuit debugger, allows a lot of time and effort to be saved. To connect the debugger the following 5 lines from the RJ-12 connector must be connected:



Pin	Signal
Not Used	1
PGC	2
PGD	3
Ground	4
VDD	5
MCLR/VPP	6

Figure 5 - ICD2 RJ-12 Pinout

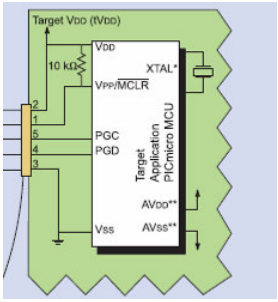


Figure 6 - ICD2 PIC connections

[MC3]

Since RB6 and RB7 are not being used with the design for this project, the connection of the ICD2 is simple, although in a more complex circuit where these pins are required, the peripherals connected to these two pins would need to be isolated whilst programming is in progress.

4.8 Circuit Block Diagram

Using the information already discussed, a block diagram for the circuit may now be produced:

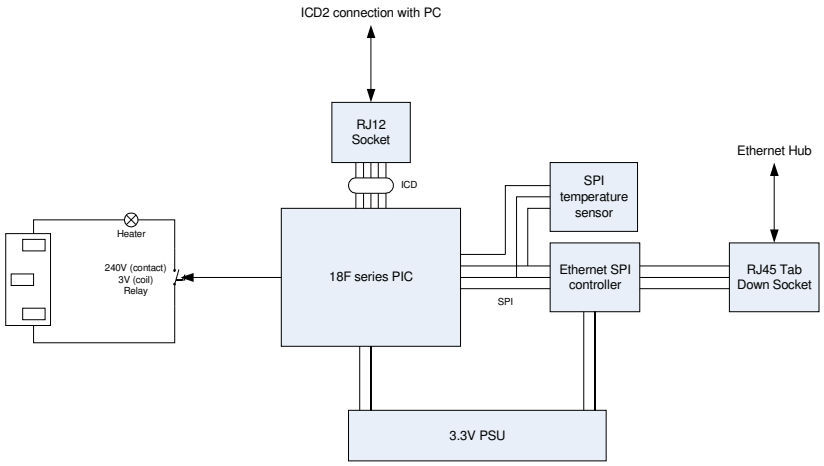


Figure 7 - Circuit Block Diagram

For a full circuit diagram, please see section 11.2 (Appendix B).

4.9 Design & modification of TCP/IP stack modules

Firstly, it is important to remember that the modules are in effect acting as part of a multi-tasking system and the minimum amount of time needs to be spent in each module. This is why all the modules have been written by using a state machine. This allows each module to remember where in a process it was, whilst allowing other modules to carry out their tasks when they would otherwise be waiting.

4.9.1 Modification of microchip's existing modules

4.9.1.1 Telnet

This module is new for version 4 of the microchip TCP/IP stack and since the stack was still pre-release when I received it, this module was not complete. With a couple of week's work, the user and password was authenticated and a basic command line was written. This provided a very useful platform to write commands for testing the custom modules that were created.

Commands implemented:

- help – display the available commands
- values – run the original task, where the status of AN0, switches and LEDs are reported
- passwd – update the telnet password (and store to internal EEPROM)
- newnbn – update the NetBIOS Name of node
    - This also involved converting to upper case, limiting to 15 characters and padding with 0x20 (space) if the supplied name is less than 15 characters. *See section 4.9.3.3 for details of this function.*
- sendtemp – accept a user inputted IP address and new target temperature value, and then trigger the transmission of the value
    - This involved converting the user entered IP address string of characters into a 32 bit hexadecimal IP address – see section 4.9.3 for details of this function.
- discover – initialise the discovery process, or report that it is already in progress
- quit – terminate the connection

4.9.1.2 HTTP

There were only a few modifications made to the HTTP module itself:

- HTTP headers updated to include the 'server' field
- .htm files allowed to have CGI content as well as .cgi files
    - This was required to allow the index page to have dynamic content
- Error 404 (not found) page updated to return a properly formatted HTML page rather than literally 'Not found'.

4.9.1.3 HTTP CGI

The Common Gateway Interface (CGI) used by the maindemo module, was pretty straight forward to understand, by looking at the existing examples.

CGI variables within the HTML are formatted %00 to %ff and when the HTTP module calls HTTPGetVar, the variable 'var' (of data-type 'BYTE') will contain the variable number (e.g. if %2e was found within a page, 'var' would contain 0x2e). One of the limitations of the function HTTPGetVar, is that only one byte can be returned at a time, although this function will be repeatedly called until HTTP_END_OF_VAR is returned, allowing a string of characters to be inserted into the dynamic HTML file.

The dynamic variables that were implemented were as follows:

- %20 – VAR_CURRENT_TEMP – Temperature reported from the local TC77 thermal sensor
- %21 – VAR_NODE_NAME – The local node's NetBIOS Name
- %22 – VAR_HEATER_STATUS – Reports whether the local heater is 'on' or 'off'
- %23 – VAR_TARGET_TEMP – Reports the local node's target temperature for the room
- %24 – VAR_REMOTE_TEMP – Returns the temperature reported from the last SendTemp operation
- %25 – VAR_DISCOVERED_NODES – Returns the string generated by the Discover module, reporting the NetBIOS Name and IP addresses of the nodes discovered on the local ethernet subnet

- %26 – VAR_DISCOVER_STATUS – Reports whether the discovery process (is) ‘’ or (is) ‘ not’ in progress

4.9.1.4 HTTP commands

HTTP commands are the variables that are sent on the end of the URI and can be easily generated by forms on HTML web pages. For example, to send a command, the URL to be open would be HTTP://domainname/cmdname?varname=var.

Once the variables have been received, it is a relatively easy task of reading the command name, variable name and variable from within the HTTPExecCmd function, using the ‘argv’ array passed. This array is formatted with argv[0] being the command name, argv[1] being the first variable name, argv[2] being the first variable, argv[3] being the second variable name, argv[4] being the second variable and so on.

The additional commands that were implemented were as follows:

- X*?Y*=TargetTemp
    - Accepts a new set point for the local node
        - Copies TargetTemp to the TempControl process variable
    - Returns index.htm
    - Since only the first character of the Command name and Variable name are tested, a more meaningful name may be inserted in place of the *
- Discovery?novariables
    - Initiates the discovery process
    - Despite not taking any variables, one must be supplied to trigger the HTTPExecCmd function to be called
    - Returns discover.htm
- SendTemp?I*=RemoteIP&T*=NewTargetTemp
    - Initiates the SendTemp process
    - Converts the RemoteIP to a 32 bit hexadecimal IP address (identical routine used in the telnet module) and copies the NewTargetTemp string to the SendTemp process variables
        - Special care had to be taken to trap if T came before I also
    - Returns index.htm

- Since only the first character of the variable name is tested, a more meaningful name may be inserted in place of the *
  - e.g. IP and Temperature

4.9.2 New stack modules written specifically for the project

4.9.2.1 GetTemp – Read temperature from TC77 SPI thermal sensor

Two bytes of data were read from the SPI bus, triggered by the chip select line going low and two 'dummy' bytes being sent. The three least significant bits of the data then had to be shifted 'out', as they were always set to '1'. The remaining data was then a 13 bit signed integer, with each step being worth 0.625°C. However a simple multiplication by 0.625, with the result being stored as a floating point was not all the conversion required before the result could be used, since a floating point number to string conversion function is not available within the MCC18 library functions. Therefore, a function was written to allow a floating point number of -99.9 to +99.9 to be converted to a string (please see section 4.9.3 for details of this function). This string was then stored as 'strTemp'.

Please see section 11.3 (Appendix C) for the complete c code listing of this module.

4.9.2.2 SendTemp – Open connection to remote node to update target temperature and read remote temperature

This module was roughly based on the GenericTCPClient module written by microchip. However, since the GenericTCPClient module is triggered by a button being pressed, a new global flag was created. When this flag was false, the state of the module would be forced to 'home' and would then remain there until the flag was set to 'true'. Once this had been triggered, the module would then use the ARP module's ARPResolve to resolve the MAC of the specified remote IP. Once this had been received, the module would continue to use the TCP module's 'TCPConnect' function (N.B. stack client mode must be enabled to use these functions) to establish a connection with the remote node. Once the socket had been established, the new set point would be sent, followed by '\r\n' (carriage return, line feed) and the response

from the remote node (i.e. the current temperature there) collected and stored. The socket could then be disconnected, using the TCP module's TCPDisconnect function.
Please see section 11.4 (Appendix D) for the complete c code listing of this module.

4.9.2.3 TempControl – Controls the state of the heater output

This module took the current temperature string (strTemp) produced by the GetTemp module and the target temperature string taken either from the HTTP command or the custom TCP socket module and using the microchip library function 'atof' converted the strings into floating point numbers. The following procedure could then be used:

```
PROCEDURE TemperatureControl
       IF {(Current Temperature) > (Target Temperature + Hysteresis)}
              Ensure the heater is switched off
       ELSE IF {(Current Temperature) < (Target Temperature – Hysteresis)}
              Ensure the heater is switched on
       END IF
END PROCEDURE
```

For the purpose of demonstrating the code worked, an hysteresis value of 0.5 °C was used. However, this value would need to be a little higher to be used in a real room, since temperature can fluctuate due to air movement cause by, for example human presence and doors / windows being opened.
Please see section 11.5 (Appendix E) for the complete c code listing of this module.

4.9.2.4 Socket – Allow the central node to update the local node's target temperature and read the current temperature

The original plan for this stack module was to base it on the GenericTCPServer, but since it was designed to process one character at a time (taking a letter and relaying its uppercase version back), it made more sense to use the telnet module as a basis, since there was a string acquisition technique already built in. After the extensive work on the telnet module, a very high level of familiarity had been built up, making the job of adapting the username entry into a target temperature acquisition, creating an additional state to report the node's current temperature and removing the additional states an easy task. The only piece of additional intelligence that was required, was to

not update the temperature set point, if either an 'empty' target temperature had been sent (i.e. no characters had been sent, only '\r\n' / carriage return, line feed).

Please see section 11.6 (Appendix F) for the complete c code listing of this module.

4.9.2.5 Discover – Allow the discovery of other microchip nodes in the IP subnet, presumably acting as room nodes

Due to the nature of this module, the only piece of code that could be used as a guide is the code in announce.c (i.e. the code that sends out a response to discovery requests). Since there was no code to use as a direct comparison, a lot more effort was required to think around the problem to get working code. However, by using the same state machine methodology as the other stack modules and some carefully chosen variables, it was possible to implement the following function:

```
PROCEDURE Remote Node Discovery
       Open UDP socket (IP & MAC all 1s, src port 31337, dest port 30303)
       Send out discovery packet
       Initialise timer
       DO
              IF (a response packet has been received) THEN
                     Save remote node's NetBIOS name and IP address in the
                     next available slot in the discovered nodes array
       WHILE (timer < 10 seconds)
       Fill 'undiscovered' spaces in the discovered nodes array with blanks
       Close UDP socket
END PROCEDURE
```

Since there are only limited resources on the PIC, it was apparent that the array of discovered nodes was quite large (using 16 bytes for the NetBIOS Name and the microchip struct IP_ADDR for the IP address), so a limit of 5 discovery slots was created. However, using a PIC with more resources than the 18F4685, or EEPROM would allow this limit to be extended. However, to use internal or external EEPROM would further complicate the procedure, also causing a reduction of the stack's operation speed.

Please see section 11.7 (Appendix G) for the complete c code listing of this module.

4.9.3 Additional functions written for the task

4.9.3.1 Converting a float to a string (within the range -99.9 to +99.9)

```
// Floating is the floating point number to be converted
Whole = Floating; // Whole is now a rounded down integer value of the #
if(whole >= 0)
       string[0] = '+';
else
{
       string[0] = '-';
       whole = whole * -1;
}
string[1] = Floating / 10 + 48;   // Tens
string[2] = whole % 10 + 48;      // Units
string[3] = '.';
Floating -= whole;
string[4] = Floating * 10 + 48;   // Tenths
string[5] = '\0';
```

Points worth noting about this code, are that:

- Adding 48 at the end of each conversion line turns the integer (assuming it is less than 10) into an ASCII character
- During the conversion, the value in Floating is destroyed, so if that floating point number is required further down, a copy must be made
- All positive numbers have a leading +
- For numbers between -9.9 and +9.9 there will be a leading zero

4.9.3.2 Converting a user entered IP address decimal dotted quad string to a 32-bit hexadecimal representation

```
// N.B. microchip stack stores the hex IP address in the following format:
// e.g. decimal dotted quad 2.4.3.7 => 0x07030402
Server.IPAddr.Val = 0x00000000;  // Initialise the hex IP address variable
StartPtr=0;                      // Initialise pointers
CurrentPtr=-1;
do
{
       CurrentPtr++;
       // Find the next dot in the string, or end of string (0x00)
       if((StringIP[CurrentPtr] == '.') || (StringIP[CurrentPtr] == '\0'))
       {
              Counter1=0;
              // Extract the number from the last dot and the current dot
              for(Counter=StartPtr;Counter<CurrentPtr;Counter++)
                     TempStr[Counter1++] = StringIP[Counter];
              TempStr[Counter1] = '\0';
              // Convert the extracted number and shift to the top byte of a
              // temporary 32 bit number
              Temp = atoi(TempStr) * 0x1000000;
              // Shift the data already in the hex IP address down a byte
              // and insert the current byte
              Server.IPAddr.Val = Server.IPAddr.Val / 0x100 | Temp;
              StartPtr = CurrentPtr + 1;
       }
} while(StringIP[CurrentPtr] != '\0');
```

Points worth noting about this code are that:

- It is assumed that it is a decimal dotted quad that has been fed to the function – if there are more than 4 numbers separated by a '.', then the last 4 should be stored in the 32 bit representation
- If a non-numeric (decimal) value is entered, the microchip atoi function will return a zero
- If a number greater than 255 is entered, the value will 'overflow' and only the 8 least significant bits will be stored in the 32 bit IP address
    - In mathematical terms, keep subtracting 256 until the result is in the range 0 - 255

4.9.3.3 Padding of a 16 byte string with 0x20 (space) if less than 15 characters stored

(i.e. for the correct storage of a NetBIOS name)

```
while(DataPtr < (NewNBN + sizeof(NewNBN)-1))
       *DataPtr++ = 0x20;
*DataPtr = 0; // Terminate the string
```

Points worth noting about this code:

- It is assumed the DataPtr is pointing at the position after the last entered character of the new NetBIOS name (NewNBN)
    - Thus allowing it to sit perfectly after the acquisition of a new NetBIOS name in the telnet module

4.9.3.4 Reading and Writing Internal EEPROM (single char)

This was achieved using the code provided from microchip. Despite looking identical in functionality to the code I had written using the PIC datasheet, this code works!

```
//=================================
// EEPROM read routine

unsigned char ReadEEPROM(unsigned int Address)
{
        // Load the high byte of the EEPROM address
        EEADRH = (unsigned char)(Address>>8);
        EEADR = (unsigned char)Address; // Load the low byte of the EEPROM address
        EECON1bits.RD = 1; // Do the read
        return EEDATA; // Return with the data
}

//=================================
// EEPROM write routine

void WriteEEPROM(unsigned int Address, unsigned char Data)
{
        // Variable to save Global Interrupt Enable bit
        static unsigned char GIE_Status;
        // Load the high byte of the EEPROM address
        EEADRH = (unsigned char)(Address>>8);
        EEADR = (unsigned char)Address; // Load the low byte of the EEPROM address
        EEDATA = Data; // Load the EEPROM data
        EECON1bits.WREN = 1; // Enable EEPROM writes
        GIE_Status = INTCONbits.GIE; // Save the Global Interrupt Enable bit
        INTCONbits.GIE = 0; // Disable global interrupts
        EECON2 = 0x55; // Required sequence to start the write cycle
        EECON2 = 0xAA; // Required sequence to start the write cycle
        EECON1bits.WR = 1; // Required sequence to start the write cycle
        INTCONbits.GIE = GIE_Status; // Restore the Global Interrupt Enable bit
        EECON1bits.WREN = 0; // Disable EEPROM writes
        while (EECON1bits.WR); // Wait for the write cycle to complete
}
```

[mc2]

4.9.4 Bugs within v4.00RC (beta) of the TCP/IP stack

4.9.4.1 TCP socket pointer error

Whilst building the website for the central node, it was discovered that if an external file was linked into the web page (e.g. an image or a cascading style sheet (CSS)) and whilst the linked file would load properly, the first segmented packet of the HTML file would be re-transmitted repeatedly. A support ticket was then raised with microchip and 8 days later a response with the solution was given:

> Dear Sir,
>
> There is a bug in the TCP.c module of 4.00RC. Specifically, on line 220 in the TCPInit() function, there is following
> ps->bufferTxStart = BASE_TCB_ADDR + sizeof(TCB) + hTCP*(sizeof(TCB) + (TCP_TX_FIFO_SIZE+1) + TCP_RX_FIFO_SIZE);
>
> This calculation is off by one, which causes memory to get corrupted in an adjacent socket. Change the line to this:
> ps->bufferTxStart = BASE_TCB_ADDR + sizeof(TCB) + hTCP*(sizeof(TCB) + (TCP_TX_FIFO_SIZE+1) + TCP_RX_FIFO_SIZE+1);
>
> Making this correction will probably fix the problem.
> I hope details mentioned above will certainly help you or please do get in touch with us for further technical support on same query.
>
> Thanks & Regards,
>
> Microchip Technical Support

With line 220 of TCP.c updated, this problem disappeared.

4.9.4.2 Intermittent UDP error

This bug involved the application data section of the UDP packet to occasionally be shifted by 10 bytes. This caused problems with all modules that use UDP (i.e. announce, DHCP and the custom discovery module written). Because of this, corrupted DHCP packets would not be interpreted as they were intended, instead being seen as nonsense or 'boot request' packets. In the case of the announce and discovery modules, the broadcast and response packets would not be interpreted correctly by the receiving device – the NetBIOS name would be shown with nonsense data preceding it and the discover packets would not begin with a 'd', hence causing remote nodes not to reply.

Again, a support ticket was raised with microchip but this time the response was to try using a new version of the stack (v4.02). Since at this stage of the project there was not enough time to learn the new intricacies of this version, a decision was made to try replacing just the UDP.c module. However, the intermittent error still remained. Had there been more time available to the project at this time, the next step would be to integrate the project into the new stack version, as it is possible that the error could be being cause by a module lower down in the stack.

5 Testing of programme and development of artefact

5.1 Major Problems encountered

5.1.1 Problem 1

5.1.1.1 Symptoms

- Programme code not being verified
- 25MHz crystal not being used, rather a slower internal RC clock showing on OSC1

5.1.1.2 Cause / Solution

The first in-circuit debugger / serial programmer was a 'clone' of the microchip ICD2 unit manufactured by etekronics, purchased from eBay for £29.00 (inc. shipping). However, it quickly became apparent that it was not dealing with the 3.3v levels used on the target board and despite efforts to try and use pull-up resistors, the code would never validate. This programmer would also apparently not set any of the configuration bits giving the symptoms of not using the 25MHz crystal, rather using a much slower internal RC clock. It was eventually decided that a new programmer / debugger would have to be found. Since the official microchip ICD2 units retail at £90 and university did not have one, another clone was found on eBay, this time manufactured by Sivava, which specifically had support for low voltage applications within its specifications. This unit cost £43.13 (inc. shipping) and has the additional feature of using USB rather than RS232, allowing faster data transfer. As soon as this second programmer was plugged in and a programme sent to the target, packets could be seen being sent / received using wireshark.

5.1.2 Problem 2

5.1.2.1 Symptoms

- Not accepting DHCP offers
- When using static IP address, packets with invalid checksums being sent out (and presumably incoming packets failing validation)

5.1.2.2 Cause / Solution

The TC77 SPI thermal sensor was built onto the first prototype board, expecting there to be no disruption, as it wasn't going to be used until a later stage. However, after careful testing, it was eventually found that contrary to what the programme should've been doing, the chip select lines were not correct and the TC77 was corrupting data on the bus. As a temporary measure, the chip was removed to allow the basic TCP/IP stack to be tested. As a result, the TCP/IP stack began working correctly, with valid checksums and DHCP offers being accepted.

As a solution to allow the TC77 to be used, a different method of setting the chip select lines was employed and another spare pin was used:

Old method:

```
PORTBbits.RB5 = 0;            // Set RB3 (TC77 CS)
PORTBbits.RB5 = 1;            // Clear RB3 (TC77 CS)
```

New method:

```
PORTB = PORTB | 0b00001000;   // Set RB3 (TC77 CS)
PORTB = PORTB & 0b11110111;   // Clear RB3 (TC77 CS)
```

It is unclear why the old method was not working but using the new method allowed the TC77 thermal sensor to be used on the same SPI bus as the ENC28J60 ethernet controller.

These two problems combined caused over two month's worth of project time to effectively be wasted, trying to debug a problem that was assumed to be software, which was in fact hardware. However, given this knowledge, future students working with similar 3.3v and/or SPI hardware whom encounter such symptoms, may take heed of the problems encountered here, hence allowing more time to be spent developing code, rather than unnecessarily debugging it.

5.2 Debugging using microchip ICD2

Using an in-circuit debugger proved very useful, whilst trying to understand how to use the SPI bus, when talking to the TC77 thermal sensor. It allows real-time reading and writing of internal registers on the PIC, including registers that control the state of the I/O ports. However, since the speed of operation is considerably decreased whilst 'running' the programme in debug mode, with the TCP/IP stack being such a large programme, it was impossible to set a checkpoint, let the programme run and wait for the break to be hit, whilst allowing the programme to otherwise function normally. Had the TCP/IP stack been configured to use a static IP address, the main loop of the stack may have eventually been entered in debug mode, except the node would be unusable, with clients trying to connect simply having their connections terminated whilst waiting (i.e. 'timing out').

Instead, a more useful method of debugging stack modules was to interface the PIC's built in UART to a PC's RS232 port, using a MAX233 level converter. Variables could then be relayed using a simple 'printf' command to a virtual terminal allowing problems to be narrowed down more efficiently. However, using methods like this can be at a risk, since they will bloat out the programme code and slightly slow down the operation of the stack if they are not removed once the glitch has been solved.

5.3 Debugging of glitches in code written

5.3.1 Reading Temperature

Once it had been realised that the chip select lines were not being driven correctly, it was relatively easy to read a value from the TC77.
However, the pre-written microchip c functions turned out not to be compatible with how the TCP/IP stack compiler options were set up, so custom modules were written, following the instructions on using the SPI from the datasheet.

Once values were being read and converted into a temperature, it was apparent that they were not particular steady, and jumping around quite a bit, about a central value. Therefore, a 'rolling average' piece of code was written to help solve the problem, by smoothing out the ripple. An array of 16 numbers (excluding repeat readings, where the ADC in the thermal sensor had not converted a new value) was filled, with the average value of the array taken to be passed for conversion to a string.

5.3.2 Telnet

One of the major flaws of the original telnet module supplied in the v4.00RC (beta) stack was the initialisation of the data pointer for the collection of user and password details. The solution to this was to set the pointer in the state before that where the next pressed key.
The second problem was that the backspace key was being accepted as a normal character, therefore not allowing the user to correct a mistake they may have made whilst entering their credentials. However, the solution was not as simple as just decrementing the pointer, as that could easily allow data before the variable to be corrupted. Therefore a trap to detect whether the data pointer was at the start of the string needed to be included also. Using this method to read the username was then used again in the collection of other variables, such as the command line commands.

5.3.3 Internal EEPROM read/writing

Apart from the problem aforementioned in section 4.9.3.4, there was one strange property of this code that existed to allow it to work. Without an immediately obvious explanation, it would appear that the function must be in the same file as the function calling it. The consequence of this meant that a second identical pair of functions (with different names) needed to be created to allow the telnet module to read and edit the telnet password as well as maindemo reading the NetBIOS name.

5.3.4 Custom socket for updating of target temperature set point

Once the string acquisition had been mastered in the telnet module, since this module used the same technique to retrieve the new target temperature set point, any problems that were being had here could be rectified by referring to the methodology used within the telnet module.

5.3.5 Client for changing set point on remote node

The only problem encountered here, was forgetting to handle a data pointer as a static variable, hence causing random pieces of memory to be written with the received data, rather than the required string, giving the appearance that the string had not been updated

5.3.6 HTTP

Troubles encountered here included:

- Incorrectly formed strings being passed back to the webpage
    - Usually due to the string not being parsed correctly causing the end not to be found, hence HTTP_END_OF_VAR not being passed back.
- When visiting the root, or home page of the server (i.e. the URI '/', returning index.htm), the CGI variables would not be replaced
    - Hence HTTP module updated to allow .htm files as well as .cgi files to contain the dynamic variables

5.3.7 Remote node discovery

The majority of problems encountered here, were due to the aforementioned UDP module bug, where the data section of the packet would be occasionally offset by 10 bytes. This caused a lot of time to be spent trying to debug code that was not at fault. However, there was also the same mistake made as when receiving the remote node's current temperature – certain variables were not being stored a static datatypes, causing their contents to be lost whilst the processor was completing other stack tasks. Nevertheless, when a properly formed packet was sent out (i.e. the data contents were not offset), a reply from the other node built was received and (assuming the response packet was also correctly formed) its NetBIOS name stored in the array, which could be viewed through the index.htm page of the web server.

The only remaining problem that did not get solved was the acquisition of the replying node's IP address. Despite the fact that the remote IP address of the replying node should be stored in the UDP socket information, the information that was actually relayed back to the discovery module was an IP address of 0.0.0.0. However, since this module was brought to this level of functionality, so near to the end of this project's timescale, there was not enough time to raise a support ticket with the microchip support staff to find the correct way, if it is at all possible, to read the remote node's IP address.

5.4 Testing of temperature control

To test whether the room node was truly capable of controlling the temperature of an environment, a test-bed platform was constructed using a cardboard box and a 40W light bulb as an heat source. Since the cardboard box is a lot smaller that an actual room, the heating from the bulb could show its effect in a smaller timeframe, hence allowing a faster and more efficient testing strategy to take place.

However, since cardboard is an inflammable substance, care had to be taken not to leave the test-bed unattended whilst the bulb was switched on.

Below, Figure 8 and Figure 9 show the constructed test-bed:

Figure 8 – Test-bed, with heat source off

Figure 9 – Test-bed, with heat source on

6 Costing

Item	Rough Price (£)
ENC28J60	2.00
PIC18F…	3.50
TC77	0.33
RJ45 Socket	3.00
Mains Relay	2.50
Regulated 3v Power Supply	3.00
PCB	1.00 (guestimate for mass production)
Resistors / Capacitors / Crystals etc	2.00
Casing	1.00 (guestimate for mass production)
Rough Total:	**18.33**

However, apart from the man-hours spent on this project, there was also money spent on the following:

- Incompatible ICD2 clone - £29
- 3.3v compatible full speed ICD2 clone - £43.13
- MAX233 chip (TTL to RS232 level converter) – £5.51
- 9-way D Sub connector (minimum pack of 5) – £5.96
- 40W Light bulb, bulb holder, mains cord & plug ~ £10
- Solderless breadboard ~ £15
- **Total spent on additional R&D costs: £108.60**

Since two room nodes were built and the microchip products were obtained free of charge through microchip samples, which brings a total cost of the project to **£129.60**.

7 Conclusions

This project has proven itself to be a much bigger challenge than first envisioned. There were some basic circuit building techniques that were ignored (i.e. building the TC77 thermal sensor into the circuit, assuming it would not affect the circuit) hence causing progress to be delayed significantly.
Nevertheless, once hardware problems had been solved (or specific traits recognised as being due to hardware faults – i.e. bad physical connections causing the ethernet link to the hub to keep being dropped), the project gave good experience for using the c programming language for more than simple laboratory exercises. Despite still making the occasional syntax error (e.g. missing a semi colon from the end of a command line), by the end of the project it was found to be a lot easier to write sections of code without having to refer to example code written by other authors, allowing [DUM94] to stay on the shelf more often.

Another large problem encountered during this project was the importance of time management – since there was a much larger work load being experienced due to other final year modules (compared to previous years of the degree course), it became a struggle at times to set aside time for the project. This is reflected in the modifications to the Gantt chart made at the second assessment stage, with the realisation that there would not be enough time to complete the project to a standard first envisioned at the initial planning stages of the project.
Please see section 11.8 (Appendix H) for the final project gantt chart.

One of the most enjoyable parts of the project, was being able to construct a TCP server and client, since the author had never had the tools to do this before, using a PC. Using the microchip TCP/IP stack has brought to life the importance and power of how using a stacked approach to a networking can make programming new modules a lot easier, at any level of the stack. An example of this is in one of the inherent designs of the stack, where the use of several network interfaces are supported (built in to a PIC, the microchip ENC28J60 and the Realtek RTL8019AS), where to change which interface is used is a simple case of including a different file in the project and enabling it through a #define command.

8 Suggestions for Further Work or Recommendations

8.1 Specific to this application

- Advanced configuration via web
    - Modification of NetBIOS name
    - Option to use fixed IP
- Customisable hysteresis for room temperature control
- Real time clock (RTC) integration
    - Allowing time / date presetting for different temperature set points for different rooms
- Data-log facility to see a room's temperature history
- Integration of an energy measurement device, to allow the user to see how much electricity is being used in heating the room
- Interfacing of radiator valve actuators
- Modification of temperature control algorithm to allow variable flow to a radiator / variable temperature of heater (i.e. use less energy when close to target temperature)

8.2 Applicable to a generic PIC ethernet project

- University to invest in PICDEM.net 2 board(s) with the ENC28J60 chip onboard, to eliminate the possibility of bad connections on the prototype board whilst debugging code.
    - Since in-circuit serial programming was so useful to this project, the product recommendation would be microchip part number DV164006 – MPLAB ICD2 evaluation kit, which includes a PICDEM.net 2 board
- Investigation into the newly released version 4.02 of the microchip TCP/IP stack

8.3 URL for future project students interested in this project

If you are reading this report, considering whether you wish to carry out a project of a similar manner, you may be interested to visit `http://uni.mcurry.co.uk/FYP/` which I anticipate to use to put more details about this project, including progress notes, MPLAB project files, full source c code listings and some pictures.

9 References

[MC1] Microchip ENC28J60 data sheet
http://ww1.microchip.com/downloads/en/DeviceDoc/39662b.pdf

[MC2] Internal EEPROM read and write functions from the microchip knowledge base
http://support2.microchip.com/KBSearch/KB_StdProb.aspx?ID=SQ6UJ9A0004DV

[MC3] Microchip 'Using MPLAB® ICD 2 Poster'
http://ww1.microchip.com/downloads/en/DeviceDoc/51265g.pdf

[WP1] 'Wikipedia' article on the ISO-OSI 7 Layer Reference Model
http://en.wikipedia.org/w/index.php?title=OSI_model&oldid=125297235

[WP2] 'Wikipedia' article on Internet Protocol
http://en.wikipedia.org/w/index.php?title=Internet_Protocol&oldid=125265480

[WP3] 'Wikipedia' article on Transmission Control Protocol
http://en.wikipedia.org/w/index.php?title=Transmission_Control_Protocol&oldid=125480812

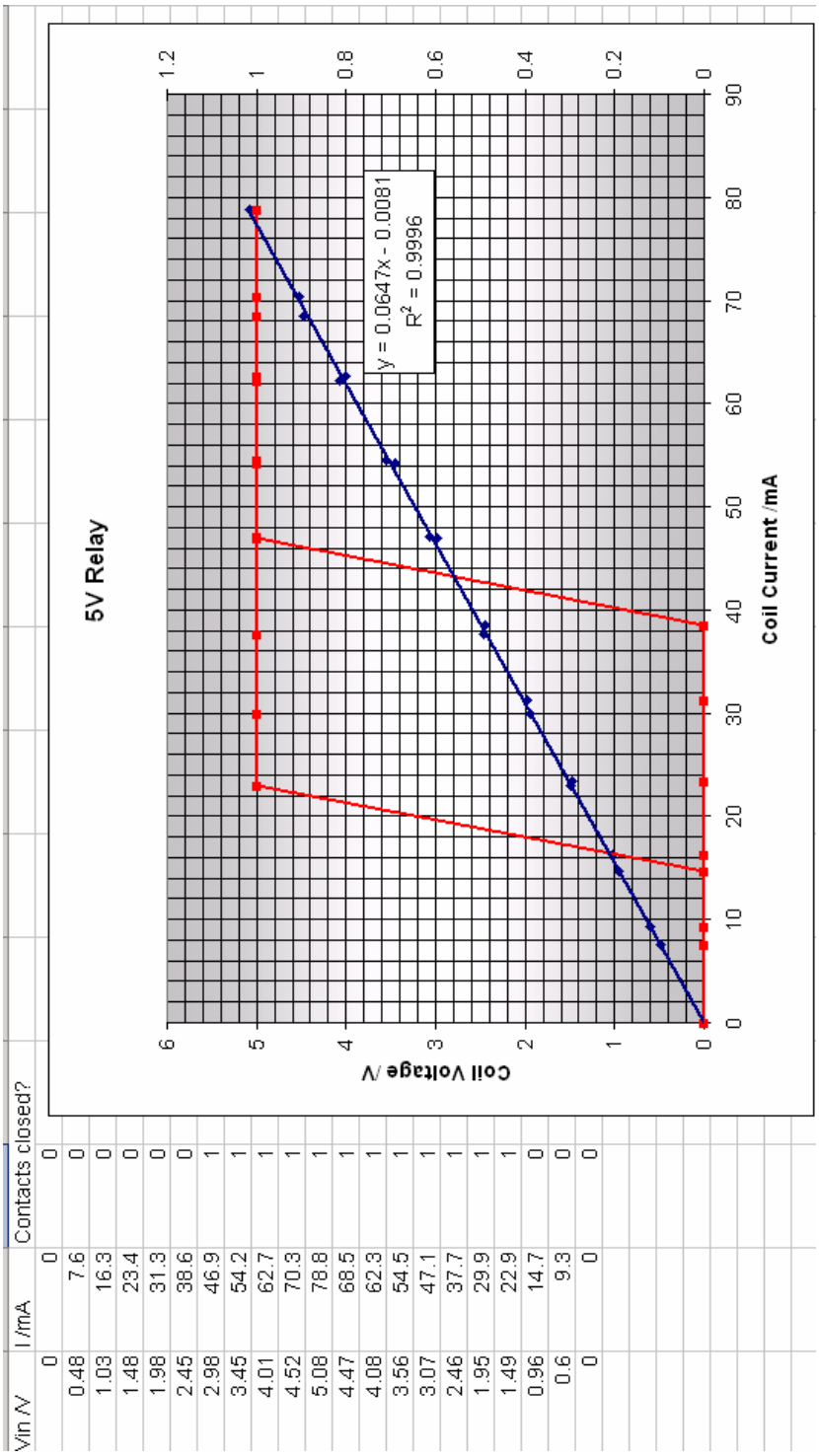
10 Bibliography

[DUM94] C For Dummies Volume One; GOOKIN, Dan; ISBN 1-878058-78-9

[STA07] Data and Computer Communications, Eighth Edition; STALLINGS, William; ISBN 0-13-243310-9

11 Appendices

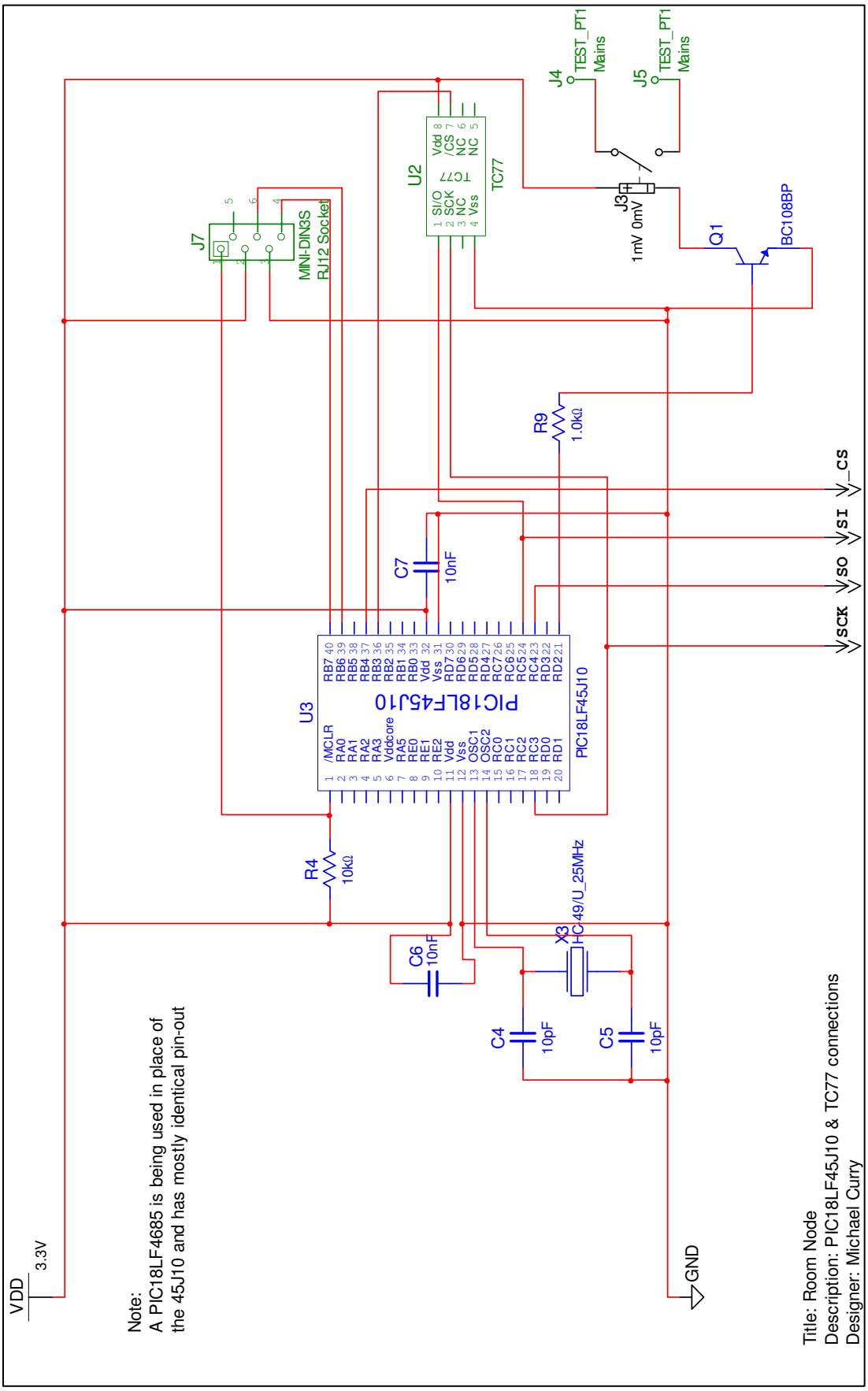
11.1 Appendix A – 5v relay suitability testing

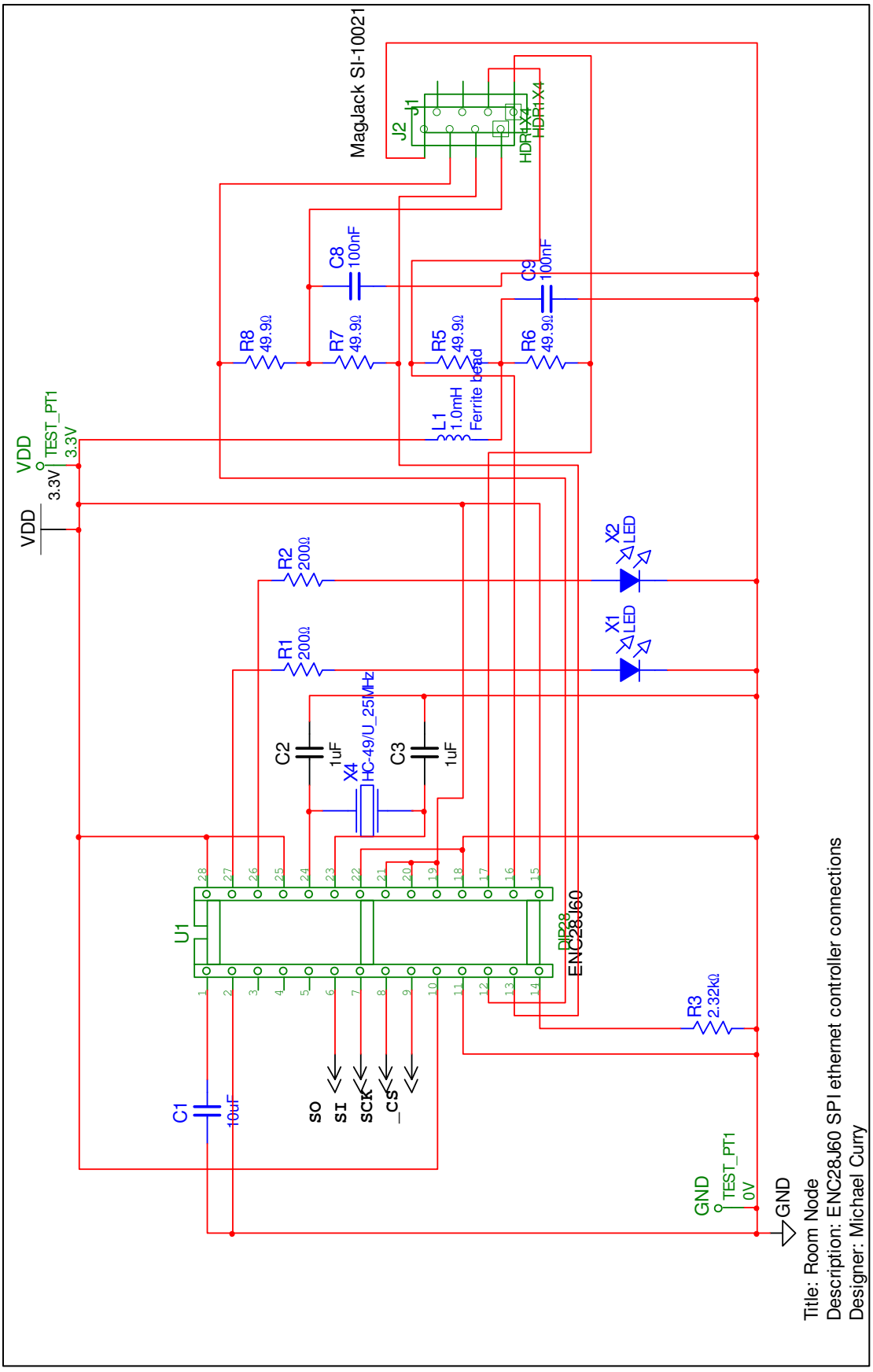


Vin /V	I /mA	Contacts closed?
0	0	0
0.48	7.6	0
1.03	16.3	0
1.48	23.4	0
1.98	31.3	0
2.45	38.6	0
2.98	46.9	1
3.45	54.2	1
4.01	62.7	1
4.52	70.3	1
5.08	78.8	1
4.47	68.5	1
4.08	62.3	1
3.56	54.5	1
3.07	47.1	1
2.46	37.7	1
1.95	29.9	1
1.49	22.9	1
0.96	14.7	0
0.6	9.3	0
0	0	0

Results from the experimentation on the 5v mains relay for suitability testing with 3.3v microcontroller. The red line indicates the state of the relay contacts and the blue line shows the coil's response.

11.2 Appendix B – Room Node Circuit Diagram




Maglack SI-10021
J1
J2
HDR1X4
HDR1X4
C8
100nF
C9
100nF
R8
49.9Ω
R7
49.9Ω
R5
49.9Ω
R6
49.9Ω
L1
1.0mH
Ferrite bead
VDD
TEST_PT1
3.3V
VDD
3.3V
R2
200Ω
R1
200Ω
X2
LED
X1
LED
C2
1uF
X4
HC-49/U_25MHz
C3
1uF
U1
DIP28
ENC28J60
R3
2.32kΩ
C1
10uF
SO
SI
SCK
_CS
GND
TEST_PT1
0V
GND
Title: Room Node
Description: ENC28J60 SPI ethernet controller connections
Designer: Michael Curry

11.3 Appendix C – GetTemp.c listing

```
/*****************************************************************************
*         Module to retrieve temperature from TC77 SPI thermal sensor         *
*****************************************************************************
*    Programme Information                                                    *
*        Author:         Michael Curry                                        *
*        Last Modified:    2007-03-23                                         *
*****************************************************************************
*    Programme Requirements                                                   *
*        TC77 /CS line connected to RB3 and ENC28J60 /CS line connected to RB4*
*        The SPI interface has already been initialised                       *
*****************************************************************************
*    Inputs                                                                   *
*        None                                                                 *
*    Outputs                                                                  *
*        None                                                                 *
*    External Variables                                                       *
*        strTemp[6] - string value of the temperature in Celsius              *
*****************************************************************************
*    Additional Programme Information                                         *
*        Rolling average taken & repeat values ignored (i.e. if the sigma-    *
*        delta converter has not produced a new reading)                      *
*****************************************************************************/

#include "p18f4685.h"

extern char strTemp[6];
//extern int tmp,tmp1,rollav[20],countav;

void GetTemp(void)
{
    static int tmp,tmp1,rollav[20],countav=0;
    float Temp;
    int whole, tenth, tens;
    unsigned int units, tenths;

    tmp1 = tmp;
    PORTB = 0b00010000;           // CS for TC77 enabled (init comms)
    SSPBUF = 0x00;                // Send dummy byte to receive byte
    while(!SSPSTATbits.BF);       // Wait for byte to be received
    tmp = SSPBUF * 0x100;         // Receive first byte (and shift to upper byte)

    SSPBUF = 0;                   // Send dummy byte to receive byte
    while(!SSPSTATbits.BF);       // Wait for byte to be received
    tmp = tmp | SSPBUF;           // Receive second byte

    PORTB = 0b00011000;            // CS for TC77 disabled

    tmp >>=3;                    // Shift out the bottom 3 bits (always 1)
```

```
    if(tmp1 != tmp)                // Ignore repeat readings
    {
        rollav[countav] = tmp;
        countav++;
    }

    if(countav == 16)
        countav = 0;

    rollav[16] = (rollav[0] + rollav[1] + rollav[2] + rollav[3])/4;
    rollav[17] = (rollav[4] + rollav[5] + rollav[6] + rollav[7])/4;
    rollav[18] = (rollav[8] + rollav[9] + rollav[10] + rollav[11])/4;
    rollav[19] = (rollav[12] + rollav[13] + rollav[14] + rollav[15])/4;
    rollav[16] = (rollav[16] + rollav[17] + rollav[18] + rollav[19])/4;

    Temp = 0.0625 * rollav[16];            // Convert to a meaningful temperature
    whole = Temp;
    {
        if(whole >= 0)
            strTemp[0] = '+';
        else
        {
            strTemp[0] = '-';
            whole = whole * -1;
        }
        strTemp[1] = Temp / 10 + 48;        // Tens
        //units = x - (tens-48) * 10 + 48;
        strTemp[2] = whole % 10 + 48;       // Units
        Temp -= whole;
        strTemp[4] = Temp * 10 + 48;        // Tenths

        //strTemp[1] = (char) tens;
        //strTemp[2] = (char) units;
        strTemp[3] = '.';
        //strTemp[4] = (char) tenths;
        strTemp[5] = '\0';
    }
    //printf("\r0x%.4x | %s",rollav[16],strTemp); // used in debugging
    return;
}
```

11.4 Appendix D – SendTemp.c listing

```
// Send new SetPoint to another node
// Based around the GenericTCPClient module

#include "TCPIP.h"

extern BOOL SendTempStatus;
extern NODE_INFO SendTempServer;
extern char SendTempTemp[32];
extern BYTE ReportedTemp[8];

void SendTemp(void);

void SendTemp(void)
{
    WORD                ServerPort = 31337;
    BYTE                i;
    static BYTE         *StringPtr;
    static TCP_SOCKET   MySocket = INVALID_SOCKET;
    static TICK         Timer;
    static BYTE         Counter;

    static enum _SendTempState
    {
        SM_HOME = 0,
        SM_ARP_START_RESOLVE,
        SM_ARP_RESOLVE,
        SM_SOCKET_OBTAIN,
        SM_SOCKET_OBTAINED,
        SM_PROCESS_RESPONSE,
        SM_DISCONNECT,
        SM_DONE
    } SendTempState = SM_HOME;

    switch(SendTempState)
    {
        case SM_HOME:
            if(SendTempStatus != 0)
            {
                SendTempState++;
                Counter = 0;
            }
            break;

        case SM_ARP_START_RESOLVE:
            if(SendTempStatus == 0)
                SendTempState = SM_DISCONNECT;
```

```
            // Obtain the MAC address associated with the server's IP address (either
direct MAC address on same subnet, or the MAC address of the Gateway machine)
            ARPResolve(&SendTempServer.IPAddr);
            Timer = TickGet();
            SendTempState++;
            break;

        case SM_ARP_RESOLVE:
            if(SendTempStatus == 0)
                SendTempState = SM_DISCONNECT;
            // Wait for the MAC address to finish being obtained
            if(!ARPIsResolved(&SendTempServer.IPAddr, &SendTempServer.MACAddr))
            {
                // Time out if too much time is spent in this state
                if(TickGet()-Timer > 1*TICK_SECOND)
                {
                    // Retransmit ARP request
                    SendTempState--;

                    // Make a note of how many times we've tried to re-transmit
                    Counter++;
                }
                if(Counter > 5)    // We may want to change this value
                {
                    // We've tried and got nowhere - back home we go
                    SendTempState = SM_DISCONNECT;
                    SendTempStatus = 0;
                    // Do we want to tell anything we failed...?
                }
                break;
            }
            Counter = 0;
            SendTempState++;

        case SM_SOCKET_OBTAIN:
            if(SendTempStatus == 0)
                SendTempState = SM_DISCONNECT;
            // Connect a socket to the remote TCP server
            MySocket = TCPConnect(&SendTempServer, ServerPort);

            Counter++;
            if(Counter > 5)
            {
                // We've tried and got nowhere - back home we go
                SendTempState = SM_DISCONNECT;
                SendTempStatus = 0;
                // Do we want to tell anything we failed...?
                break;
            }

            // Abort operation if no TCP sockets are available
```

```
        // If this ever happens, incrementing MAX_TCP_SOCKETS in
        // StackTsk.h may help (at the expense of more global memory
        // resources).
        if(MySocket == INVALID_SOCKET)
            break;

        SendTempState++;
        Timer = TickGet();
        break;

    case SM_SOCKET_OBTAINED:
        if(SendTempStatus == 0)
            SendTempState = SM_DISCONNECT;
        // Wait for the remote server to accept our connection request
        if(!TCPIsConnected(MySocket))
        {
            // Time out if too much time is spent in this state
            if(TickGet()-Timer > 5*TICK_SECOND)
            {
                // Close the socket so it can be used by other modules
                TCPDisconnect(MySocket);
                MySocket = INVALID_SOCKET;
                SendTempState--;
            }
            break;
        }

        Timer = TickGet();

        // Make certain the socket can be written to
        if(!TCPIsPutReady(MySocket))
            break;

        // Place the application protocol data into the transmit buffer.
        {
            StringPtr = SendTempTemp;
            for(i = 0; i < strlen(SendTempTemp); i++)
            {
                TCPPut(MySocket, *StringPtr++);
            }
            TCPPut(MySocket, '\r');
            TCPPut(MySocket, '\n');
        }
        // Send the packet
        TCPFlush(MySocket);
        ReportedTemp[5] = '\0';
        StringPtr = ReportedTemp;
        SendTempState++;
        break;

    case SM_PROCESS_RESPONSE:
```

```
            if(SendTempStatus == 0)
            {
                SendTempState = SM_DISCONNECT;
                break;
            }

            // Check to see if the remote node has disconnected from us or sent us any
application data
            if((!TCPIsConnected(MySocket)) && (StringPtr - ReportedTemp > 5))
            {
                *StringPtr = '\0';
                SendTempState++;
                break;
            }

            if(!TCPIsGetReady(MySocket))
                break;

            // Obtain the server reply
            while(TCPGet(MySocket, &i))
            {
                *StringPtr++ = i;
            }

            break;

        case SM_DISCONNECT:
            // Close the socket so it can be used by other modules
            TCPDisconnect(MySocket);
            MySocket = INVALID_SOCKET;
            SendTempState = SM_DONE;

        case SM_DONE:
            SendTempStatus = FALSE;
            SendTempState = SM_HOME;
            break;
    }
}
```

11.5 Appendix E – TempControl.c listing

```
// Temperature Control Stack Module

#include "TCPIP.h"
#include "stdio.h"
#include "stdlib.h"

extern BYTE SetPoint[32];
extern char strTemp[6];
extern BOOL Heat;

void TempControl(void);

void TempControl(void)
{
    signed float Current, Target, Hysteresis=0.5;
    char test[7];

    Current = atof(strTemp);
    Target = atof(SetPoint);
    if(Target > 70.0)       // N.B. ENC28J60's temperature limit is 0-70C!
        Target = 70.0;      // and by design, a negative value cannot be set

    if (Current > (Target + Hysteresis))
    {
        // Time to switch the heater off
        PORTDbits.RD2 = 0;
        Heat = FALSE;
    }
    else if (Current < (Target - Hysteresis))
    {
        // Time to switch the heater on
        PORTDbits.RD2 = 1;
        Heat = TRUE;
    }
}
```

11.6 Appendix F – Socket.c listing

```
/*********************************************************************
 *                                                                   *
 *     Room node socket, based on microchip Telnet Server module     *
 *                                                                   *
 *********************************************************************/

#include "TCPIP.h"
#include "string.h"
#include "stdlib.h"

#define PORT 31337

extern BYTE SetPoint[32];
extern BYTE strTemp[8];

void MikesTask(void); // Prototype

void MikesTask(void)
{
    BYTE                i;
    ROM BYTE            *ROMPtr;
    BYTE                *RAMPtr;
    static BYTE         Data[32];
    static BYTE         *DataPtr;
    static TICK         Timer;
    static TCP_SOCKET   NewSocket = INVALID_SOCKET;
    static NODE_INFO    Remote;
    static double temp;

    static enum _MikesState
    {
        SM_HOME = 0,
        SM_Welcome,
        SM_Get_Setpoint,
        SM_Validate_SP,
        SM_Return_Temp,
        SM_DISCONNECT,
    }   MikesState = SM_HOME;


    switch(MikesState)
    {
        case SM_HOME:
```

```
            // Connect a socket to the remote TCP server
            NewSocket = TCPListen(PORT);

            // Abort operation if no TCP sockets are available
            // If this ever happens, incrementing MAX_TCP_SOCKETS in
            // StackTsk.h may help (at the expense of more global memory
            // resources).
            if(NewSocket == INVALID_SOCKET)
                break;

            memcpypgm2ram(Data,0x00,32);
            MikesState++;
            Timer = TickGet();
            break;

        case SM_Welcome:
            // Wait for the remote client to connect to us
            if(!TCPIsConnected(NewSocket))
                break;

            Timer = TickGet();

            // Initialise the data pointer BEFORE you are in the next state!
            DataPtr = Data;
            MikesState++;

        case SM_Get_Setpoint:
            if(!TCPIsConnected(NewSocket))
            {
                MikesState = SM_Welcome;
                break;
            }

            // Retrieve the set point
            while(TCPGet(NewSocket, &i))
            {
            if(((i != '\r' && i != '\n') && ((i >= '0' && i <= '9') || i ==
'.')) && (DataPtr != (Data + sizeof(Data))))
                    *DataPtr++ = i;
                else if((i == 0x08) && (DataPtr > Data)) // Backspace
                    DataPtr--;

                // Stop parsing and advance if linefeed encountered
                if(i == '\r')
                {
```

```
                    *DataPtr = '\0';        // End of the string
                    MikesState++;
                    break;
                }
            }

            break;


        case SM_Validate_SP:
            temp = atof(Data);
            if((Data[0] != 0x00) )
            {
                strcpy(SetPoint,Data);
                TCPFlush(NewSocket);
            }
            MikesState++;
            break;

        case SM_Return_Temp:
            if(!TCPIsConnected(NewSocket))
            {
                MikesState = SM_Welcome;
                break;
            }

            TCPPutString(NewSocket, strTemp);
            TCPFlush(NewSocket);
            MikesState++;


        case SM_DISCONNECT:
            // Close the socket so it can be used by other modules
            // For this application, we wish to stay connected, but this
state will still get entered if the remote server decides to disconnect
            TCPDisconnect(NewSocket);
            memcpyram2pgm(0x00,Data,32);
            MikesState = SM_Welcome;
            break;
    }
}
```

11.7 Appendix G – discover.c listing

```
// Discovery module - discovering which other nodes are out there
//
// Whether I can get this working is another matter though...
//
// Broadcast packet is sent to 255.255.255.255
// Packets sent out in reply are only addressed to the discoverer's IP
// UDP Port is always 30303 (source and destination)
// UDP source port can be different (duh!)
// Received packet will have the first 15 bytes as the NetBIOS name,
// followed by \r\n, then the MAC address followed by \r\n
// Anything after this is the 'other info' field (e.g. DHCP/Power event
occurred)
//

#include "TCPIP.h"

// This must begin with 'D'... I think that is all!
ROM BYTE strDiscover[] = "Discovery: Who is out there?";

typedef struct _MikeNode
{
    IP_ADDR     IPAddr;
    BYTE        NetBIOSName[16];
} MikeNode;
extern MikeNode DiscoveredNodes[];   //Contents being IPAddr and NetBIOSName
extern BOOL DiscoverNodes;

void DiscoverNodesProcess(void); // prototype
void DiscoverNodesProcess(void)
{
    static UDP_SOCKET   MySocket;
    NODE_INFO           Remote;
    BYTE                i;
    static BYTE         x,y,timeout;
    static ROM BYTE     *ROMDataPtr;
    static BYTE         *DataPtr;
    static TICK         Timer;

    static enum {
        SM_DISCOVER_HOME = 0,
        SM_DISCOVER_BROADCAST,
        SM_DISCOVER_LISTEN,
        SM_DISCOVER_SOMETHING,
```

```
    SM_DISCOVER_CLOSE
} SM_DISCOVER = SM_DISCOVER_HOME;

switch(SM_DISCOVER)
{
    case SM_DISCOVER_HOME:
        if(DiscoverNodes != 0)
            SM_DISCOVER++;
        x=0;
        timeout=1;
        break;

    case SM_DISCOVER_BROADCAST:
        if(DiscoverNodes == 0)
        {
            SM_DISCOVER = SM_DISCOVER_CLOSE;
            break;
        }

        // Set the socket's destination to be a broadcast over our IP
        // subnet
        // Set the MAC & IP destination to be a broadcast (all 1s)
        memset(&Remote, 0xFF, sizeof(Remote));

        // Open a UDP socket for outbound transmission
        MySocket = UDPOpen(31337, &Remote, 30303);

        if( MySocket == INVALID_UDP_SOCKET )
            break;

        // Make certain the socket can be written to
        while( !UDPIsPutReady(MySocket) );

        // Since I was having so much trouble with the bug in the UDP
        // module, several methods were used to try and get the string
        // be sent out correctly, before realising it was not a fault
        // with my code, rather with the microchip stack!

        // Put the discovery string in the packet
        //DataPtr = strDiscover;
        /*
            ROM BYTE *ROMPtr = "Discovery: Anyone there?";
            while(*ROMPtr)
            {
                UDPPut(*ROMPtr++);
```

```
        }
    }*/
    UDPPut('D');
    UDPPut('i');
    UDPPut('s');
    UDPPut('c');
    UDPPut('o');
    UDPPut('v');
    UDPPut('e');
    UDPPut('r');
    UDPPut('y');
    UDPPut(':');
    UDPPut(' ');
    UDPPut('A');
    UDPPut('n');
    UDPPut('y');
    UDPPut('o');
    UDPPut('n');
    UDPPut('e');
    UDPPut(' ');
    UDPPut('t');
    UDPPut('h');
    UDPPut('e');
    UDPPut('r');
    UDPPut('e');
    UDPPut('?');

    // Send the packet
    UDPFlush();

    Timer = TickGet();
    y=0;
    SM_DISCOVER++;
    break;

case SM_DISCOVER_LISTEN:
    if((DiscoverNodes == 0) || (TickGet()-Timer > 10*TICK_SECOND))
    {
        SM_DISCOVER = SM_DISCOVER_CLOSE;
        break;
    }
    // Do nothing if no data is waiting
    if(!UDPIsGetReady(MySocket))
        return;
```

```
            // Progress to the next state if something is RX
            SM_DISCOVER++;

        case SM_DISCOVER_SOMETHING:
            if((DiscoverNodes == 0) || (TickGet()-Timer > 10*TICK_SECOND))
            {
                SM_DISCOVER = SM_DISCOVER_CLOSE;
                break;
            }

            DataPtr = DiscoveredNodes[y].NetBIOSName;

            for(x=0;x<15;x++)
            {
                //while(!UDPIsGetReady(MySocket));
                UDPGet(&i);
                //*DataPtr++ = i;
                DiscoveredNodes[y].NetBIOSName[x] = i;
            }
            DiscoveredNodes[y].NetBIOSName[16] = '\0';
            DiscoveredNodes[y].IPAddr.Val = Remote.IPAddr.Val;
            y++;
            UDPDiscard();
            SM_DISCOVER--;
            break;

        case SM_DISCOVER_CLOSE:
            DiscoverNodes = 0;
            if(y < 5)
            {
                for(y=y;y<5;y++)    // Fill unused space with blanks
                {
                    BYTE strNotFound[] = "No Node Found";
                    strcpy(DiscoveredNodes[y].NetBIOSName,strNotFound);
                    DiscoveredNodes[y].IPAddr.Val = 0xffffffff;
                }
            }
            SM_DISCOVER = SM_DISCOVER_HOME;
            UDPClose(MySocket);
            break;
    }
}
```

11.8 Appendix H – Project Gantt Chart