Electronic Engine Controller Simulation and Emulation With Ethernet Connectivity

By

Josh Blackann

Submitted in Partial Fulfillment of the Requirements

For the Degree of

Master of Science in Engineering

In the

Electrical Engineering Program

Youngstown State University

May, 2011

Electronic Engine Controller Simulation and Emulation
With Ethernet Connectivity

Josh Blackann

Signature:

Josh Blackann, Student Date

Approvals:

Dr. Faramarz Mossayebi, Thesis Advisor Date

Dr. Jalal Jalali, Committee Member Date

Dr. Frank Li, Committee Member Date

Peter J. Kasvinsky, Dean of Graduate Studies & Research Date

ABSTRACT

Electronic engine controllers are key components to the overall function and success of many different applications, such as automotive or other power generation equipment. This research documents the design and development of an electronic engine controller based on a microcontroller development kit. The controller application maintains a constant RPM output out of a generator based on varying load conditions. A secondary circuit, an engine simulator, was developed which coupled with the engine controller provides a complete test bed to prove its functionality. Ethernet connectivity was integrated into this project to allow the development boards to access the internet to be viewed remotely or to modify and control the operation of the application through a webpage.

The implementation of the test bed associated with this work demonstrates the ability to build and simulate a low cost electronic engine controller circuit using standard microcontroller evaluation kits. The verification of experimental results for monitoring and control of the modeled engine through a webpage is demonstrated by inclusion of numerous screen captures from an oscilloscope.

The test bed developed in this research work can be expanded and enhanced to model more complex engines. Additionally, the controller aspect of this experimental setup can be further modified to test and verify more advanced real time control algorithms.

ACKNOWLEDGEMENTS

To my family; Alexa, Owen, and Finley,

For all of their understanding and support as I worked countless hours on my electronic engine controller thesis. Without your support, I would not have been able to complete this work.

To Dr. Mossayebi,

For the guidance and encouragement to complete my thesis.

To my co-workers (past and present),

For sharing their experience and knowledge to help guide me through my continual learning of embedded microcontrollers.

To Peter Reen,

For his insights into Ethernet products and applications from Microchip.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Introduction

An electronic engine controller is a device that controls the ignition and other parameters required for operation of an engine. The controller is the brain of an engine with its ability to start and stop the engine. Additionally, it regulates the speed, monitors various sensors, and provides other safety features. Electronic engine controllers (EEC) consist typically of a microcontroller as well as support circuitry which run algorithms or programs to manage the overall operation of an engine. Internal combustion engines and other various incarnations of engines benefit from the management of an EEC.

Microcontrollers are integrated circuits which have processing capabilities, memory storage, and hardware peripherals combined in a single package. These specialized electronics perform different functions or tasks as defined by the memory of the device. This allows products to have an element of intelligence that cannot be accomplished with discrete electronics. This ability to add "smarts" to products has opened the door for embedded electronics, enabling technologies such as handheld GPS and smart phones. Also, with increased use of embedded microcontrollers, products have been designed to connect to the internet for remote communication or operation, and even software updates. This thesis will explore a specific application of microcontrollers as well as the addition of Ethernet connectivity to provide internet access.

1.2 Application Background

Prior to the adoption of EECs, engine control was typically performed by elaborate mechanical systems which may not have functioned as desired in all situations. Purely mechanical control systems were the first type of method utilized for controlling engines. As laws and regulations have changed over time, new emission standards have been put in place that required engines to operate with a higher degree of fuel performance. In 1987, the Detroit Diesel series made its debut. It was a four-stroke-cycle diesel, and it was offered as a fuel-injected, electronically-controlled engine designed to meet the new emission requirements [1]. This began the shift to electronically controlled engines, allowing intelligence to be added to the overall system.

An engine managed by an EEC can be programmed to handle different situations based on climate, mode of operation, and various changes from other sensors in the overall system. The ever increasing integration of microcontrollers into everyday products and systems provides an excellent opportunity to create an electronic engine controller based on an evaluation board from a microcontroller vendor. The implementation of electronics controllers has improved both the efficiency and the performance of engines. This thesis focuses on the design of a basic electronic engine controller as well as a simulator circuit which provides the necessary feedback for the controller to function properly. This electronic engine controller is not aimed at any particular market (for example, automotive) but it could be used as a starting point for development of a custom controller for any engine.

1.3 Organization

The first chapter of this thesis is an introduction into the topic of engine controllers and microcontrollers. The second chapter contains a more detailed examination of microcontrollers from Microchip and covers the application design as well as the tools utilized for development. An overview of the functionality of the electronic engine controller and its implementation is described in chapter three. The fourth chapter details the simulator circuit which tested the operation of the controller. An introduction to TCP/IP and the Microchip stack are provided in chapter five, as well as a description about how Ethernet connectivity was incorporated in both the controller and simulator applications. The sixth and final chapter contains conclusions and future development opportunities. Appendices follow which detail the different development tools and software utilized, circuit schematics, other background information, as well as the source code for the application.

Chapter 2

Hardware and Application Overview

2.1 Electronics Development Tools

The design and development of electronic circuits is no small undertaking. To develop an electronic engine controller, an evaluation kit from an embedded microcontroller manufacturer was utilized to speed up implementation. Development kits from Microchip Technology provided an excellent starting point to develop a project or proof of concept utilizing microcontrollers. Microchip Technology Inc. is a leading manufacturer of microcontrollers and analog semiconductors [2]. Microchip creates various versions of microcontrollers: 8-bit, 16-bit, and 32-bit variants. Numerous development tools provide entry level options for learning as well as more advanced kits for creating new products. The PIC18 Explorer board (referred to as PIC18 EB) is a demo board for the PIC18 MCU family which is the highest performance 8-bit architecture in the Microchip product microcontroller line [3]. The PIC18 EB provided evaluation of the PIC18 hardware and its ability to operate with extra peripherals. Sample source code is also available with the PIC18 EB that provides some examples of what could be done with the PIC18 hardware. A voltmeter, temperature sensor, and a real time clock are some examples of the applications on the demo board.

Microchip provides a free development suite, MPLAB IDE, for developing and creating embedded applications (see section documentation for further information) [4]. Firmware can be written in numerous different programming languages; Assembly, BASIC, and C. Additionally, numerous third party companies provide tools and support for Microchip products, such as Basic Micro Inc and CCS, Inc.

The Microchip PICkit 3 is an in-circuit debugger/programmer which provides access to debugging logic incorporated into each chip with Flash memory [5]. The PICkit 3 can program the PIC18EB or other devices and can even function as a production programmer. The debugging capability of the PICkit 3 is invaluable. It provided much needed insight into the inner workings of the processor allowing a developer to single step through code to analyze what functionality the microcontroller is currently processing.

Figure 2.1 PICkit 3 Debugger/Programmer

2.2 Implementation

The electronic engine controller needs to manage the speed of the engine while it performs a variety of other functions. The engine speed is assumed to be varying from 100 to 3600 revolutions per minute (RPM) based on input settings. As the controller circuit monitors the RPM, it reports calculations via the onboard LCD on the PIC18EB. With the addition of internet connectivity, the device communicates its status via a web interface. The RPM rate of the engine deviates from its prescribed RPM setting due to different loading conditions. The controller then modifies its output to bring the engine back into regulation, either by increasing or decreasing the time period which the device fires its outputs, or by adjusting the duty cycle of injector on time.

To create an electronic engine controller, ideally it would be best to have an engine for development but this was not a feasible option due to the cost and test setup requirements. One PIC18EB was configured to simulate an engine and another PIC18 EB was utilized to create the engine controller. The PIC18EB has a wide variety of hardware components which provided an excellent opportunity for development of a prototype application. The evaluation board supports Plug-In Modules (PIMs), allowing different PIC18 microcontrollers to be swapped in and out. An LCD module was integrated to display different information about the microcontroller functions. Additionally, there is a 6 pin male header for a PICkit2/3 interface for programming and debugging. Other useful features of the PIC18 EB include 2 push buttons, 8 LEDs, a 256k-bit (32K Byte) EEPROM, an analog temperature sensor, PICtail daughter board connector socket, a potentiometer for analog inputs and several other components. There is also a small prototyping area which was utilized for connecting additional push buttons and other circuitry.

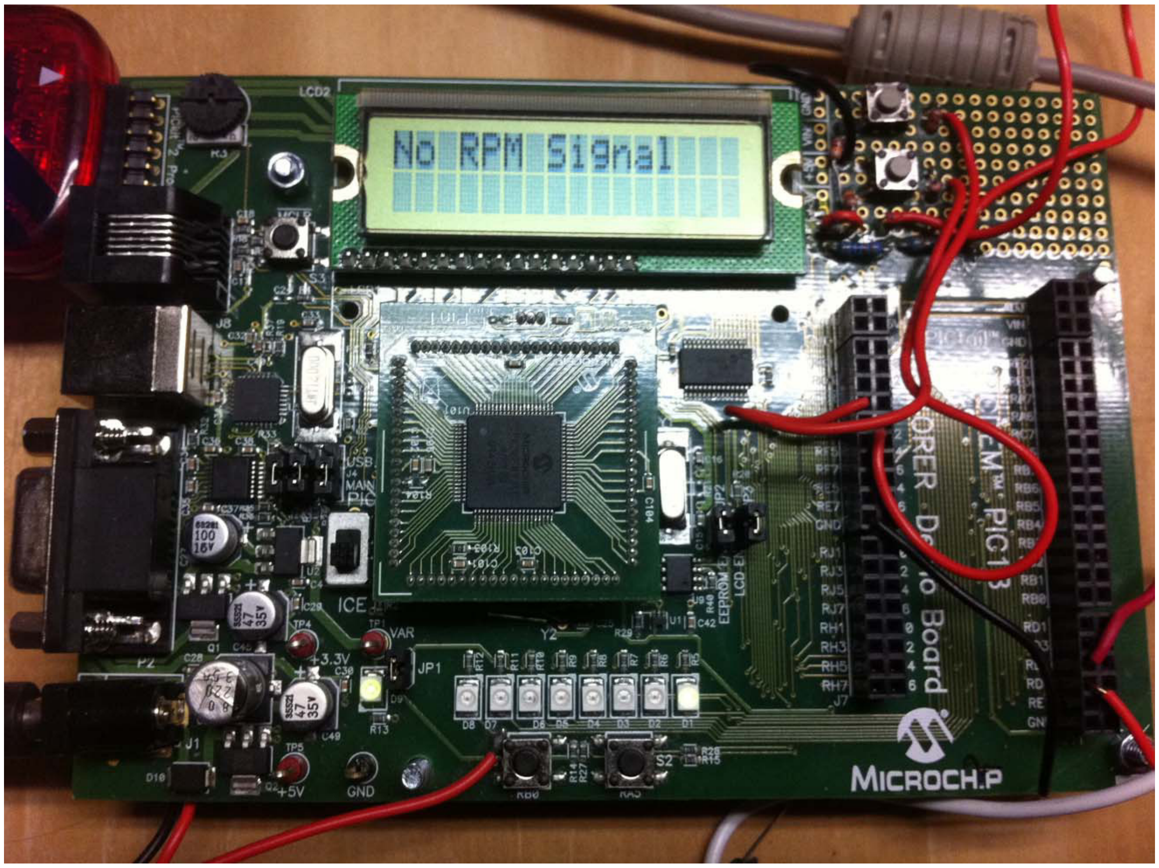


Figure 2.2 PIC18 Explorer Board

A PIM with the PIC18F87J11 (referenced as the 87J11) from Microchip was the microcontroller of choice. The 87J11 is a low-voltage microcontroller in the PIC18 family with 128 Kbytes of Flash program memory and 3930 bytes of SRAM, providing ample resources for developing applications. Nine different ports control the functionality of 68 input/outputs pins, of which there is a subset of 15 pins which can record analog values and then convert this to 10-bit digital input value. [6] The 87J11 has many other features and peripherals that proved useful while developing applications. For instance, the Master Synchronous Serial Port (MSSP) handles communication between the 87J11 and external ICs, such as the SPI I/O expander for communication with the LCD, the external EEPROM (U9 on the board), and other devices which can be plugged into the PICtail daughter board connector socket. The 87J11 also has several hardware timers that can be configured to create specific timing functions as further explained in this work.

To verify that the hardware worked properly, sample source code for the PIC18EB was programmed into the 87J11. The demo emulated a voltmeter by displaying the analog voltage across a potentiometer on the LCD, as well as displaying the temperature from an onboard temperature sensor and a real time clock. The sample source code was written in Assembly. The code was ported to C for better readability. The applications for this controller and simulator were created using the C programming language with C18 Compile from Microchip. For more information on C18, see Microchip's website [7].

Block diagrams and flow charts were developed as a "roadmap" for the embedded application. Several different block diagrams were created using Microsoft Visio to detail the communication from the PC to development and the interactions between evaluation kits. See figure 2.3 for a representation of the overall system design of different components that must operate with one another. The electronic engine controller initialization sequence is illustrated in Figure 2.4, and the complete operation of the engine controller can be found in Figure 2.5. A flowchart for the simulator board is provided in Figure 2.6. These high-level diagrams provide a method of visualizing the overall objective of the entire system as well as the multiple facets of the firmware application.

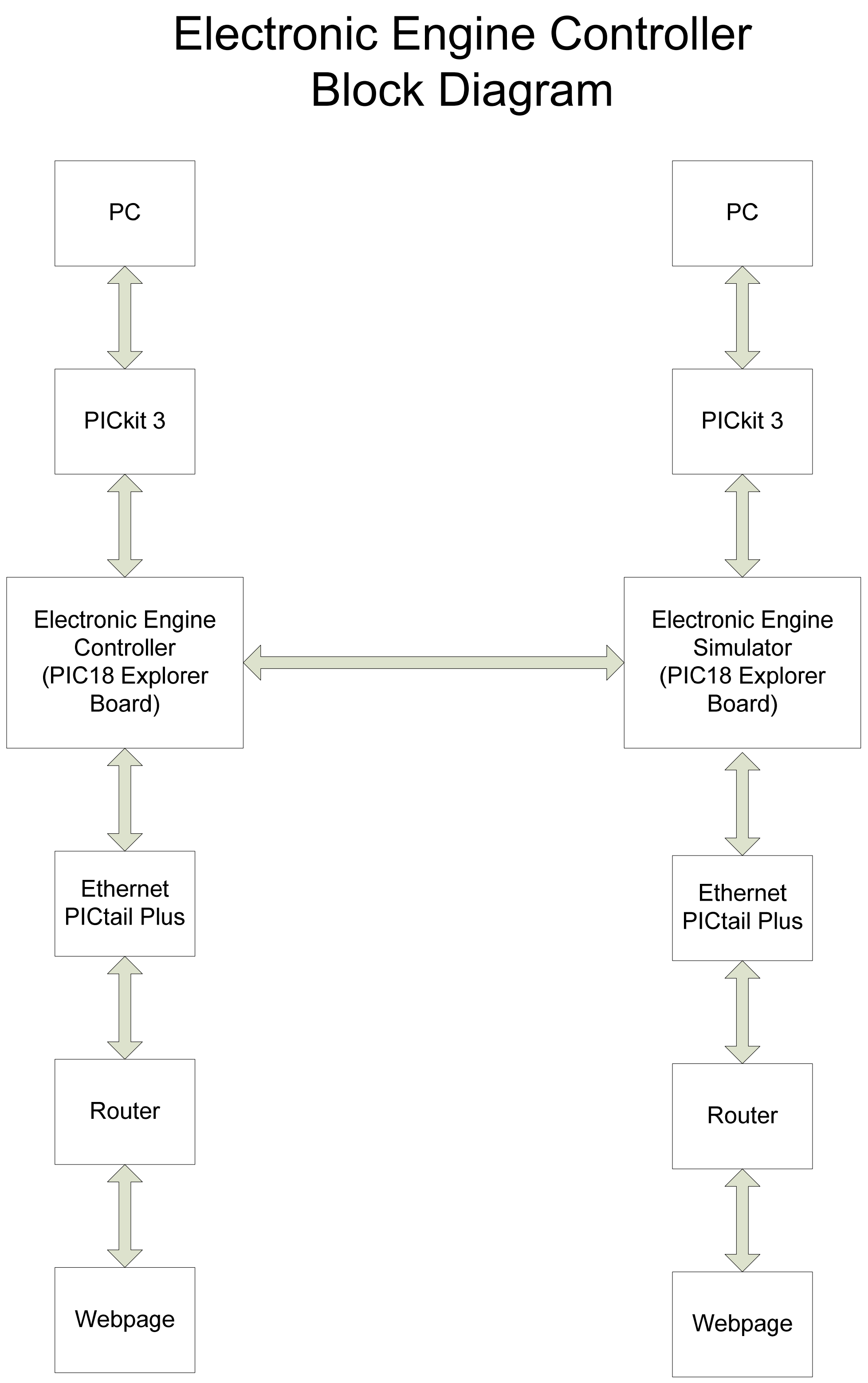

Electronic Engine Controller
Block Diagram
PC
PICkit 3
Electronic Engine Controller (PIC18 Explorer Board)
Ethernet PICtail Plus
Router
Webpage
PC
PICkit 3
Electronic Engine Simulator (PIC18 Explorer Board)
Ethernet PICtail Plus
Router
Webpage

Figure 2.3 Overall Electronic Engine Controller Block Diagram

Electronic Engine Controller
Flowchart

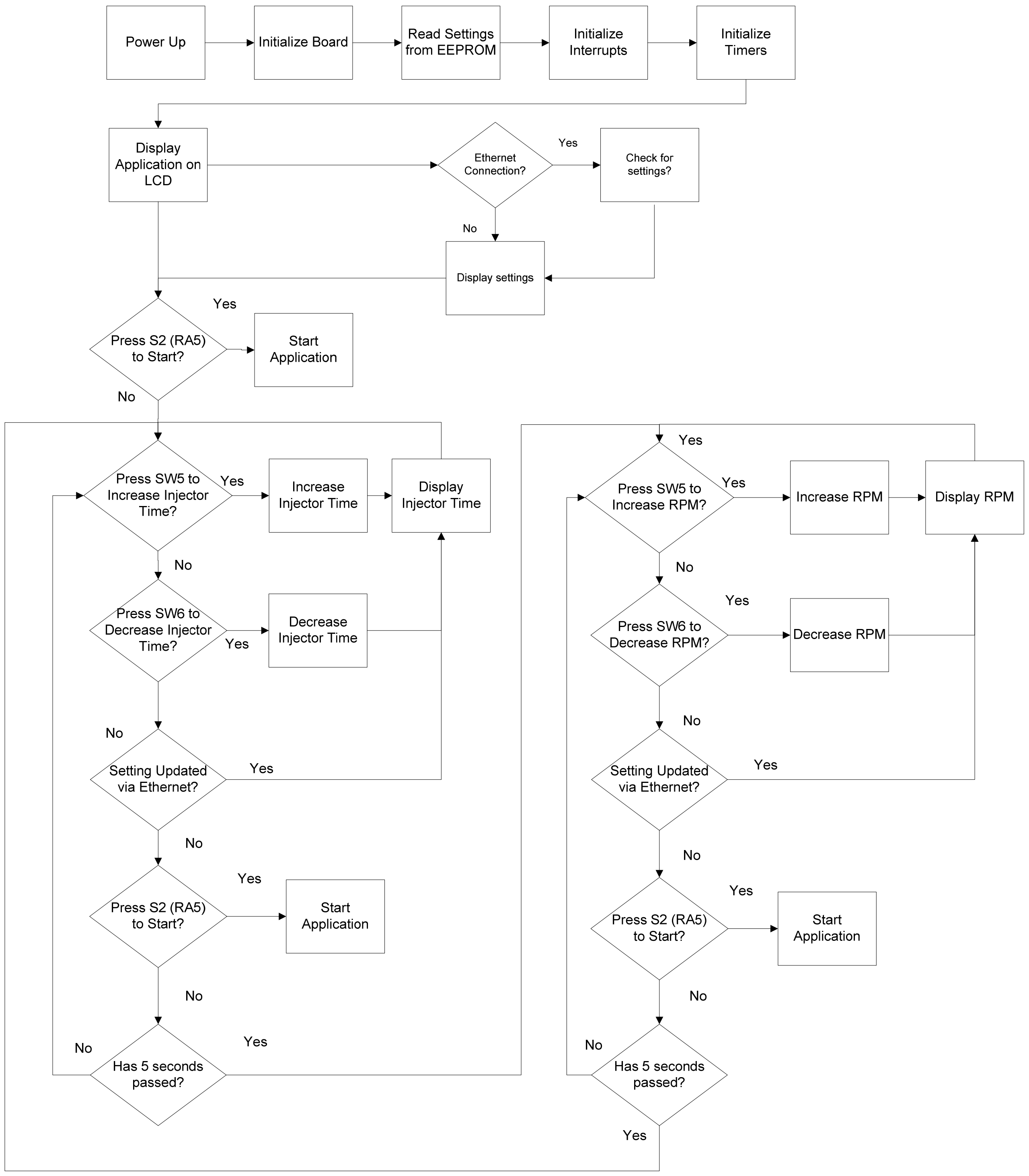


Figure 2.4 Electronic Engine Controller Initialization

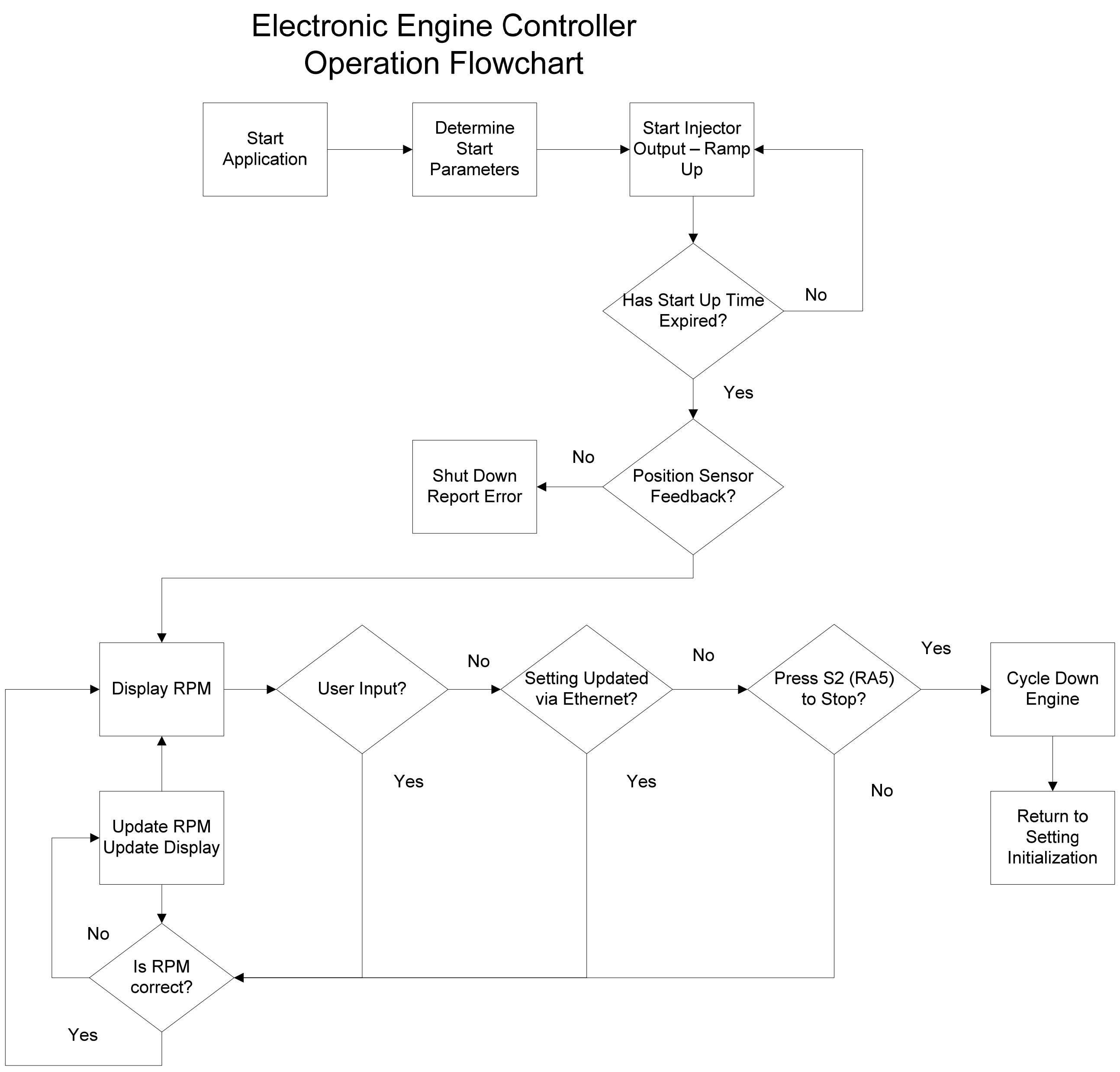

Electronic Engine Controller Operation Flowchart
Start Application
Determine Start Parameters
Start Injector Output – Ramp Up
Has Start Up Time Expired?
No
Yes
Position Sensor Feedback?
No
Shut Down Report Error
Display RPM
User Input?
No
Yes
Setting Updated via Ethernet?
No
Yes
Press S2 (RA5) to Stop?
Yes
No
Cycle Down Engine
Return to Setting Initialization
Update RPM Update Display
Is RPM correct?
No
Yes

Figure 2.5 Electronic Engine Controller Operation Flowchart

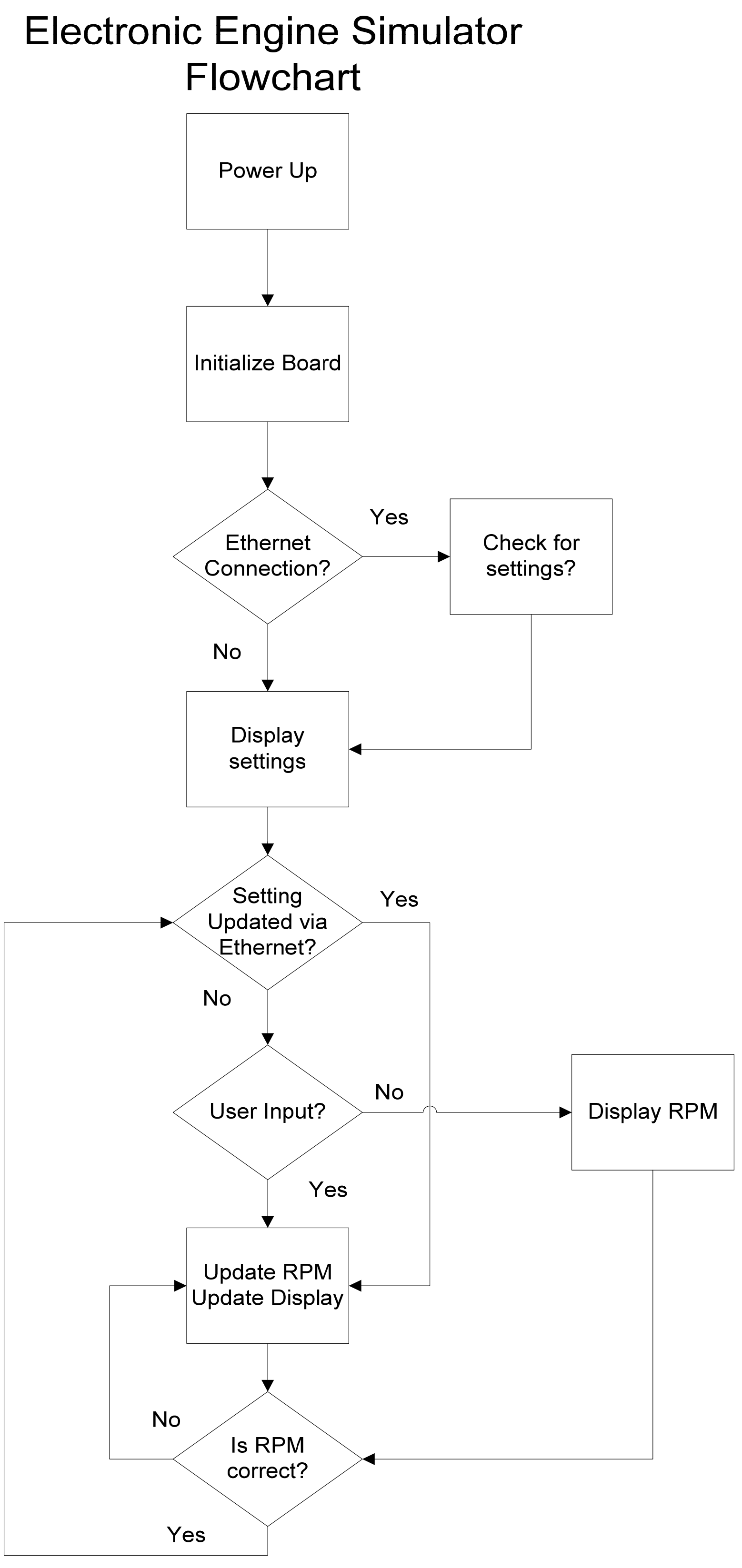


Figure 2.6 Electronic Engine Simulator Flowchart

Chapter 3

Development of Electronic Engine Controller

3.1 Application Overview

The main goal of this work was the development of an engine controller which adjusts its output to control the speed at which the simulator operates. The engine controller generates a specific RPM rate and monitors the feedback signal from the simulator circuit. Then, based on its settings, varies the RPM signal to either slow down or speed up the engine. Numerous different types of engines exist in varying degrees of complexity. The Moore Engine was selected as a reference design for this controller based on its simplicity. It consists of one cylinder which would turn a crankshaft which turns a flywheel, moving the piston back and forth. Injectors located at the top of the cylinder are electronically controlled, opening to allow pressurized air or gas to drive the piston down causing the flywheel to rotate. The flywheel has magnets placed upon it surface which pass a fixed proximity sensor. Eight magnets are evenly positioned 45 degrees apart around the perimeter. One additional magnet is placed on the flywheel which indicates when the piston is at the top of its stroke. The position sensors are aligned such that one position sensor is located at the same degree location as the injector signal. Another sensor is located 180 degrees of the injector which indicated the piston was at the bottom of its stroke.

3.2 Controller Detail

Upon power up, the engine controller displays its application name on the LCD module on the PIC18EB. The engine controller program utilizes three push buttons to modify the circuit's operation; S2, S5, and S6. The push buttons are connected to general purpose input/output (I/O) pins which are configured as inputs. The S2 button is connected to Port A pin 5 or RA5; S5 is connected to Port F pin 1 or RF1; and S6 is connected to Port F pin 2 or RF2. The LCD displays the previously saved settings for the desired RPM rate as well as the injector time. These settings are stored on an external EEPROM, 25LCD256, U9, making the settings non-volatile [8]. For example, the controller may boot up displaying the injector time, which would be the time in milliseconds that the injector would be on, and the desired RPM rate. Each parameter can be varied by pressing S5 to increase the time or RPM, or by pressing S6 to decrease the RPM rate. If the buttons are activated for an extended amount of time, the device counts by larger intervals. For example, individual button presses only adjusted a parameter up or down by a count of one. If a button is held down and the device counted to a value of 10 by increments of 1, the device would then begin to count by intervals of 10. This allows settings to be configured faster by counting the RPM rate in values of 10s or 100s to cover the range of 100 to 3600. The injector time can also be set in a similar manner. As designed the injectors would only stay on for a maximum of 50% of the operating RPM or considered a 50% duty cycle. The injector duty cycle can be varied from 1 to 50% in increments of 1. Once the settings are configured as desired, S2 can be pressed to begin the motor controller application.

The injectors are fired to get the engine turning. A startup sequence ramps the simulator or actual engine up to the desired speed. The method used for the startup sequence was to take the desired RPM and calculate steps for the controller. The controller then outputs the calculated RPM steps until the engine reaches the desired

RPM setting. As referenced in section 3.1 about the Moore engine, the engine operation is monitored by a number of sensors located on the flywheel which indicated the rate at which the engine is turned, as well as position. One additional sensor is attached to the flywheel which, when in the correct position, indicates when the injector should fire. The position sensors are monitored by connecting inputs RB0 and RB3. These pins have external interrupt capability meaning that other circuitry can be connected to these pins to trigger the RPM calculation. When a falling edge trigger occurs on RB0, an interrupt function increments a position sensor count. The same is true for RB3, but this indicates that the flywheel is in the correct position to enable the injector to fire.

The first operation the controller performs is to determine and calculate the cycle time for a specific RPM rate. Using Excel, a spreadsheet was created which had the RPM rates as well as different timing parameters to help determine what timing function would be required. (See appendix for calculations)

$$\text{RPM} = \text{cyc} / \text{min} \tag{3.1}$$

In which RPM = revolutions per minute

cyc = engine cycles or revolutions

min = minutes

To calculate the required RPM signal, the Timer1 module is used in 16-bit counter mode. The 87J11 has several clock sources available (see section 2.0 of the 87J11 datasheet [6]). To utilize the PIC18EB with the Ethernet expansion board, the on board 10MHz crystal was required as well as enabling the High Speed Crystal with Software Phase Locked Loop (HSPLL), which multiplies the oscillating source by 4 up to 40 MHz. The correct cycle time is created by loading a value into Timer1 which will increment until it overflows, rolling over from 65535 to 0, causing a Timer 1 overflow interrupt to occur.

$$T1RPM = Tpc / T1pre * Icy / 2 \qquad (3.2)$$

In which T1RPM = number of Timer 1 must count to overflow.

Tpc	= Time per cycle.
T1pre	= Timer1 Prescaler.
Icy	= Instruction cycle frequency where the oscillator frequency, Fosc /4 or 40MHz/4, or equal to 10 MHz.

Divided by 2 as this is one half of RPM frequency

For example, to create an RPM of 600, the time per cycle was 0.1 seconds / (prescaler of 8) * 10M (40M/4) /2, Timer1 needed to count 62,500 times. Timer1 must be loaded with an initial value such that after Timer1 increments 62,500 times it will overflow and cause an interrupt. Timer1 is a 16-bit timer, so it can count to 65535, thus the correct value to load is 65535 - 62500 = 3035. The excel spreadsheet found in the appendix D provides details of all of the calculations. When generating all of the counts, it was evident that as the RPM rate increased the number of counts between RPM decreased, creating slight timing inaccuracies. Choosing values which were as close to the 16-bit limit (65535) but landed on a RPM step boundary (multiples of 300) provided the most accurate results. This method worked for calculating higher RPM rates (600 and above) but the count was too large for slower RPMs. In this case, a slower clock source was required. On the PIC18EB, there is an external 32kHz clock attached to the Timer1 input pins. The same equation 3.2 was used with the 32kHz clock to generate

RPM rates of 16 to 599. The table below shows the different parameters used for calculating the RPM.

Table 3.1 Parameters used for calculating RPM

RPM Range	Clock Source	Prescaler
16 to 599	32 kHz	1:1
600 to 1199	40 MHz	1:8
1200 to 2399	40 MHz	1:4
2400 to 3600	40 MHz	1:2

This method used for determining the RPM is fairly accurate but it does have a small error when calculating higher frequencies (> 1200). The timing error is due to the overhead required when servicing the Timer1 interrupt overflow. For the purposes of this project, there was no need to correct for this error but there was the need to understand that it exists in the event that the system requires more accuracy.

As the injectors are activated, the engine cycles and the simulator's position sensor signal acts as a feedback signal which is used to determine the motor's speed. To record the simulator's RPM, the engine controller could have used several different methods. One method, not implemented in this work, would have sampled the input signal and used a timer to count the time that elapsed between the feedback signal toggling back and forth. Another method, which is implemented, is to record the number of times the sensor has toggled over a specific period of time. The TCP/IP stack includes a tick timer function using the Timer0 module which will be used when Ethernet connectivity is added later. Timer 0 uses the internal instruction cycle clock source and operates as a 16 bit counter. The RB0 input pin is configured as an input with an interrupt on falling edge trigger, so each time a position sensor is detected, a count is incremented. On the flywheel of the simulated engine there are 8 position sensors which were evenly spaced out and one additional injector sensor. Once a second the 87J11 takes the total number of position sensor counts and multiplies that number by 60. After several successive count recordings, the board will then build up a more accurate representation of the RPM rate of the engine. This successive calculation will be ongoing thus the RPM is averaging the engine speed.

$$\text{RPM ct} = \text{pct} * 60 / 8 \tag{3.3}$$

In which RPMct = RPM count

pct = position count is number of RB0 interrupt triggers recorder over 1 second

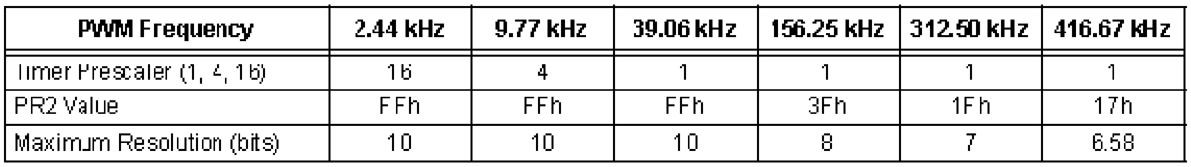
Multiplied by 60 to give the total count for one minute

Divided by 8 as there are 8 sensor positions per revolution.

Once this RPM rate is calculated, the 87J11 displays the RPM on its LCD module. If the calculated RPM rate is less than the desired rate, the controller then attempts to increase the RPM by turning the injectors on for a slightly longer time or by increasing its own RPM rate. If the opposite occurred, where the calculated RPM rate is higher than the desired rate, the controller attempts to decrease the RPM rate by firing the injectors for a slightly shorter time or by decreasing its own RPM rate. The RPM rate was varied back and forth to simulate different load conditions being applied to the motor. The further out of tolerance the simulator was, the more the controller tried to modify its operation.

For the circuit to vary the injector duty cycle, the Compare/Capture/PWM hardware peripheral in the 87J11 can be utilized. At first glance, the PWM feature seems best suited for manipulating the duty cycle of the injectors. Upon further examination of the PWM module, it is evident that the PWM module operates at a much higher frequency when the 87J11 is configured to operate at 40 MHz.

Table 3.2 Example PWM Frequencies and Resolutions at 40 MHz [6]



PWM Frequency	2.44 kHz	9.77 kHz	39.06 kHz	156.25 kHz	312.50 kHz	416.67 kHz
Timer Prescaler (1, 4, 16)	16	4	1	1	1	1
PR2 Value	FFh	FFh	FFh	3Fh	1Fh	17h
Maximum Resolution (bits)	10	10	10	8	7	6.58

As the max operating speed of the controller is 3600 RPM or a signal frequency of 60 Hz, the PWM is not suitable for modifying the duty cycle of the injector. To implement the duty cycle calculation, the compare feature can be utilized. Timer1 is used to generate the RPM signal which creates a 50% duty cycle. To make the duty cycle smaller, the compare feature can be used to cause an interrupt to occur when Timer1 reaches a number smaller than that required to cause an overflow generating the RPM signal. Thus to make the calculation, we must take the Timer1 RPM value and subtract it from the Timer1 maximum count (65535). This value is then multiplied by two and the multiplied by the duty cycle. The result of this is then added to the initial value used with Timer1 to generate the RPM signal. The result is then stored in CCPR4 which is a 16-bit register which is continually compared to Timer1.

$$\text{PWM} = (((\text{T1Ovr} - \text{T1RPM}) * 2) * \text{DCy}) + \text{T1RPM} \qquad (3.4)$$

In which, PWM = value timer 1 needs to count to for specific duty cycle

T1Ovr = Timer1 overflows at count of 65535

T1RPM = Timer1 RPM value (defined in eq 3.2)

Divided by 2 as this is only one half of RPM signal

Dcy = duty cycle

For example, if RPM is 700 and duty cycle is to be 20%, the value used for Timer1 is 53571.

So 65535 – 11964= 53571

(53571 * 2) * 20/100 = 21428

Thus PWM value = 21428 + 11964 = 33392

Thus a value 33392 will need to be placed in CCPR4 and when Timer1 reaches this count an interrupt will occur which will turn off the injector if it is on, creating a RPM signal with a 20% duty cycle.

The S2 button can be pressed at anytime to cause the engine to shut down. This causes the application to break from its normal controlling mode and return to its initial parameter setup state.

Chapter 4

Development of Electronic Engine Simulator

4.1 Engine Simulator Application Overview

With firmware for the controller developed, simulator firmware was required to provide the sensor feedback signal. The simulator circuit is the inverse of the controller application. The simulator analyzed the input RPM signal and then generates a position sensor feedback signal. As stated previously, the RPM signal would be when the injector should be fired, a signal occurring once per revolution. The position sensor would be triggered at a rate eight times faster than that of the RPM (injector) signal.

4.2 Simulator Detail

The 87J11 used on the simulator PIC18EB would need to capture the input RPM signal on the RB0 pin. As stated in the controller section 3.2, this pin was configured for capturing external trigger signals. Based on the input signal, the 87J11then generates the resulting position feedback signal required by the controller. The input RPM signal was recorded on the simulator in the same manner as feedback signal was analyzed on the controller. The simulator recorded the RPM by counting the number of trigger signals that occurred over a one second time interval and then added several samples together to generate an accurate RPM. The calculation was as follows:

$$\text{RPMcalc} = \text{RPMin} * 60 \qquad (4.1)$$

In which RPMCalc = RPM value calculated

RPMin = RPM input signal count from controller (recorded over 1 second)

Multiplied by 60 as to determine RPM rate for a minute

The difference between this calculation and the one performed by the engine controller is the simulator does not need to divide by eight, as only one injector trigger would occur once per revolution.

With the RPM calculation, the simulator 87J11 computes the necessary timing functions for the position sensor signal. This calculation is similar to RPM signal generated via Timer3 on the engine controller side. The simulator 87J11 uses Timer3 as a 16-bit counter to generate the feedback signal. Timer3 needs to be loaded with a value which will continue to increment until it overflows, rolling over from 65535 to 0, causing an interrupt to occur.

$$T3Cnt = Tpc / T3pre * Icy / 2 / 8 \tag{4.2}$$

In which T3Cnt = number of times Timer 3 must count to overflow.

Tpc = Time per cycle.

T3pre = Timer3 Prescaler.

Icy = Instruction cycle frequency where the oscillator frequency, Fosc /4 or 40MHz/4, or equal to 10 MHz.

Divided by 2 as this is one half of RPM frequency

Divided by 8 as there are 8 position sensors

This calculation was also performed in an excel spreadsheet to generate the correct counts required for synthesizing the appropriate signal. These values can be found in the spreadsheet located in Appendix D.

To verify that the calculations were correct, the simulator generated an RPM signal of 176 for the controller which was a calculated frequency of 2.933 Hz. The simulator also generated the position sensor signal which was at a frequency 8 times faster, with a calculated frequency of 23.466 Hz or a 42.6 mS period. Using an oscilloscope, the signal was captured and measured. As depicted in Figure 4.1, the injector signal (yellow, CH1) occurs once every eighth position sensor pulse (blue, CH2). The scope calculates the period of the position sensor signal as 42.60 mS with a frequency of 23.47 Hz, matching that of the spreadsheet calculations.

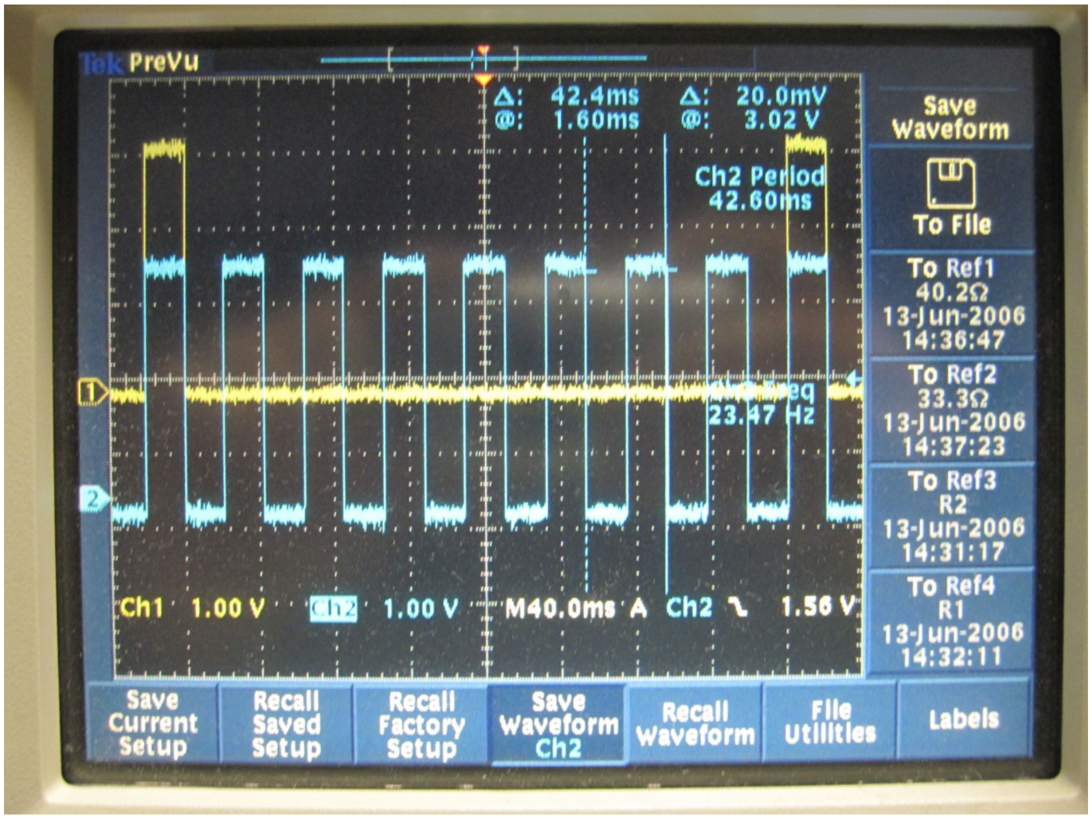


Figure 4.1 Screen shot with RPM signal and Position sensor signal

Utilizing the calculations from the spreadsheet, the values listed in Table 4.1 were used to determine the prescaler settings required to generate the position sensor signal.

The higher frequency position sensor signal requires that a high clock source and prescaler be used to generate the correct signal.

Table 4.1 Parameters used for calculating Feedback Position Sensor Signal

Position sensor signal rate	Clock Source	Prescaler
75 to 299	40 MHz	1:8
300 to 599	40 MHz	1:2
600 to 3600	40 MHz	1:1

The Timer 1 module is configured in the same manner as the controller application; creating an interrupt once a second, and toggling LED1 or RD0. The position sensor signal can be seen on LED 6 or RD5 pin. The injector signal can be seen on LED8 or RD7.

For the simulator to test the controller, the circuit had to modify its position sensor feedback simulating a load being placed on or removed from the system. This simulated the effect of the position sensor signal either decreasing or increasing, respectively. To modify this parameter, values can be changed through the TCP/IP application (as depicted in Chapter 5 TCP Integration) or another input can be utilized. The onboard potentiometer, R3, is used as an input which can be manually varied to simulate the various loading conditions. The potentiometer provides an analog input voltage to the RA0 pin of the 87J11. The on chip Analog to Digital Converter (AD) module is used to sample the analog voltage. This sample is converted into a 10-bit (1024 step) digital number which is then processed (for more information on the Analog to Digital Converter, please refer to section 21 of the 87J11 datasheet [6]). The potentiometer is set to the midpoint, and a digital value of 512 is determined. This was considered to be the condition where the loading was at a steady state. Turning the potentiometer counter-clockwise causes the analog voltage to decrease, which represents the system slowing due to it being loaded down. If the potentiometer is turned the other direction (clockwise) the analog voltage would increase, indicating the RPM is increasing and the load is being reduced. In either case, the change in analog voltage is constantly calculated. Based on the amount of variation from the midpoint value, the controller then modifies its output signal to bring the simulator back within its specification. The LCD module displayed whether the system was being loaded down or the load was being reduced.

The variation in load is determined by converting the analog voltage into different step ranges. The ranges were centered about the midpoint of the potentiometer (a value of 512). A range of 100 provided a stable midpoint, ranging from 462 to 562. Step sizes of 100 from the midpoint range value were calculated and used to determine the loading condition of the simulator. See the figure 4.2 below which shows the different steps.

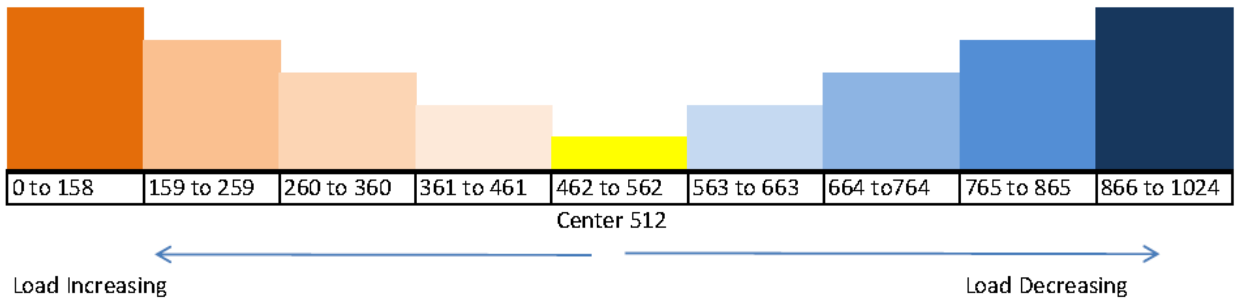


Figure 4.2 Digital values corresponding to analog values simulating load variations

If the load increases to Load Increase Step 1 (between 361 and 461), the RPM feedback signal will slow down by approximately 3%. If the loading continues to increase to Load Step 2 (260 to 360), the RPM signal will slow by approximately 6%. If further load still occurs, the RPM signal will slow by 12% or 25%, whether it enters Load

Inc Step or Step 4, respectively. If the load is decreased the same changes will be seen but the RPM signal will increase.

An accurate analog value was obtained by taking an average of eight recordings. The 87J11 uses the calculated result to determine which load condition the simulator is currently in. The analog to digital module continually sampled the analog voltage on RA0 and modified the feedback signal as required.

Chapter 5

Connecting to the World with Ethernet

5.1 TCP/IP Introduction

After the simulator and engine controller board were successfully developed, Ethernet connectivity was added to allow communication to the outside world. Ethernet is a data link and physical layer protocol defined by IEEE 802.3 specification [31 re-order]. With the addition of an Ethernet connection, the PIC18EB can be plugged into an internet connection and accessed through a network or remotely through the internet by adding a web server interface. Microchip has a section of its website dedicated to the development of Ethernet products (see Microchip Ethernet Solutions [9]). The Fast 100 Mbps Ethernet PICtail Plus (EPP) Daughter board is made by Microchip which plugs into the PIC18EB to provide an Ethernet link [10]. At the heart of the EPP is the ENC624J600, which is a stand-alone 10/100 Base-T Ethernet interface controller with integrated Media Access Control (MAC) & physical data link layer (PHY) [11]. Microchip offers a free licensed TCP/IP stack optimized for the PIC18, PIC24, dsPIC and PIC32 microcontroller families [12]. The TCP/IP stack is included in the Microchip Applications Library which also had numerous examples of TCP/IP applications. The TCP/IP ENCX24J600 Demo App was chosen as a starting point for development using Ethernet that used the EPP. Also included with the Applications Library is a help file which provides documentation and a walk-through of the various TCP/IP examples [13]. A review of Microchip's Application notes details the stack in more detail and presents the theory of operation [14, 15].

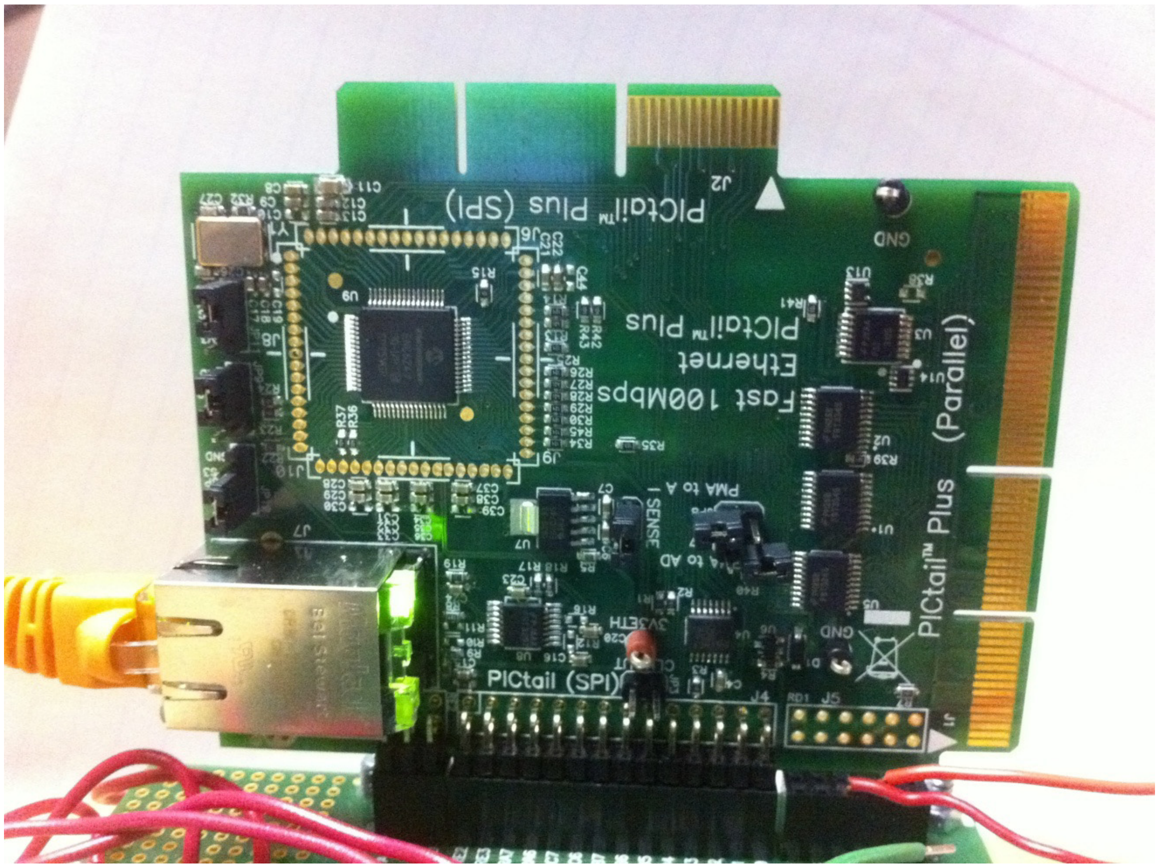


Figure 5.1 Ethernet PICtail Plus (EPP) Daughter Board

5.2 TCP/IP Stack

The Microchip TCP/IP stack currently supports the TCP and UDP transport layer modules, the IPv4 (and part of the ICMP) Internet Layer modules, the ARP link layer modules, and a variety of application layer modules[13]. The stack includes more than 60 C files as well as more than 70 header files for implementing the various different application layers or other functionality found in TCP/IP. Only a specific sub-set of the provided TCP/IP stack is required for every project, see the list below [13].

- A main file- this is the file with the application code in it.

- ARP.c and ARP.h- These files are used by the stack to discover the MAC address associated with a given IP address.
- Delay.c and Delay.h – These files are used to provide delays for some stack functions. Note that it would be best to not use these delays in your own code, as they do create blocking conditions.
- Physical layer files – These files are used to enable a specified physical layer. More information on which files to include can be found in the Hardware Configuration section.
- Helpers.c and Helpers.h – These files contain helper functions used for miscellaneous stack tasks.
- IP.c and IP.h – These files provide internet layer functionality for the stack.
- StackTsk.c and StackTsk.h – These files contain the code to initialize the stack and perform the callbacks that keep the stack going.
- Tick.c and Tick.h – These files implement a tick timer that is used to implement some timing functionality within the stack.
- HardwareProfile.h – This configuration file is used to set up hardware options.
- TCPIPConfig.h – This configuration file is used to set up firmware options.
- MAC.h – This header file provides macros and structures relating to the hardware MAC layer.
- TCPIP.h – This is the primary include file for the stack.The main file should include TCPIP.h.

5.3 TCIP/IP Integration

The ENC Demo App demonstrates the ability to use the PIC18 as a web server, e-mail client, and for building custom applications. The EPP was plugged into the PICtail expansion header and the demo was loaded onto the PIC18EB via the PICkit3. An Ethernet connection was made between the EPP and a network router as well as a PC. With the firmware updated on the 87J11, the next step was to load the web server file on to the board. Included with the TCP/IP stack, is the MPFS2 Utility which packages webpages into a format for efficient storage in an embedded application. Microchip provides a demo file which can be uploaded into the EEPROM on the PIC18EB. The MPFS2 utility was used to load the demo webpage onto the PIC18EB. Then using an internet browser, it was possible to navigate to the PIC18EB web server by navigating to http://mchpboard (or the programmed host name) or by also typing the IP address of the evaluation kit. The sample code provided a function for displaying its current IP address on the LCD on the PIC18EB. Figure 5.2 depicts the setup of the PIC18EB with the Ethernet PICtail Plus board connected as well as a PICkit 3 programmer attached for programming and debugging the source code. Figure 5.3 shows a screenshot of the example web server page from the Microchip example.

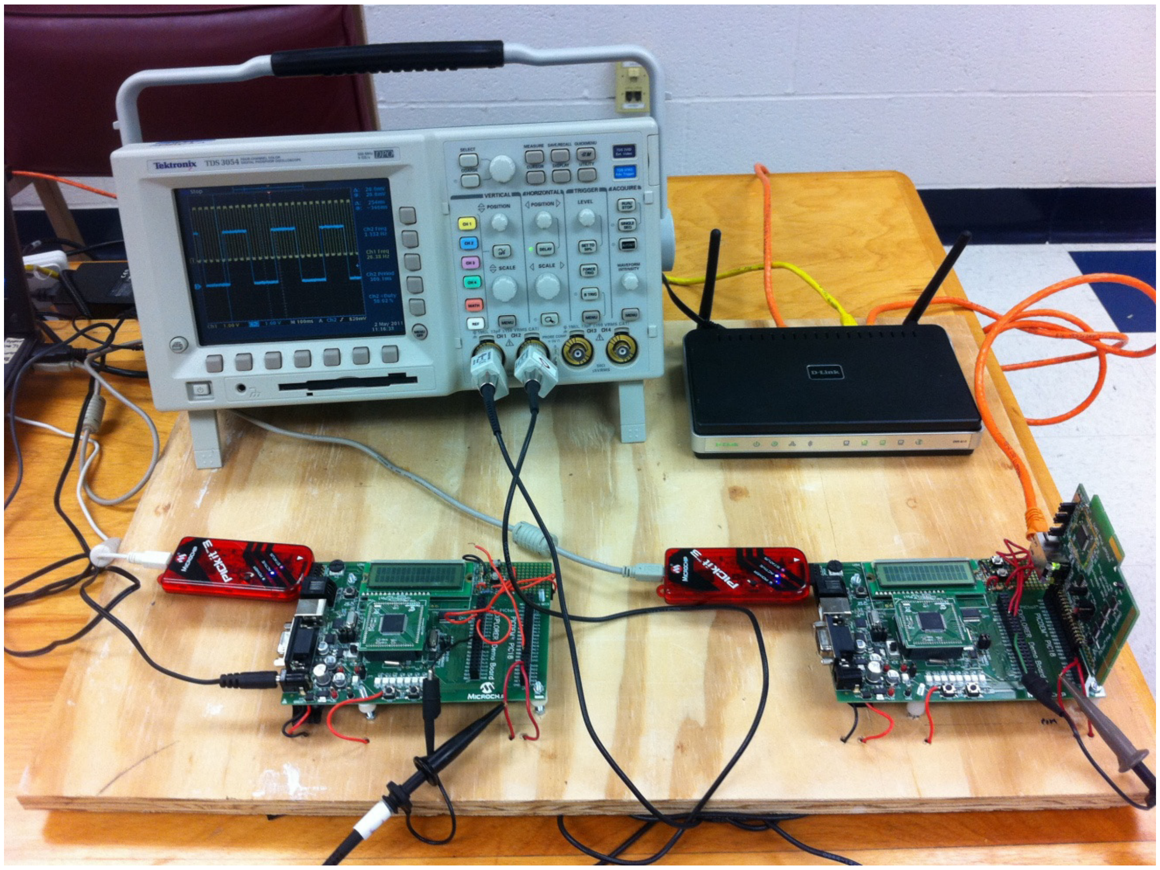
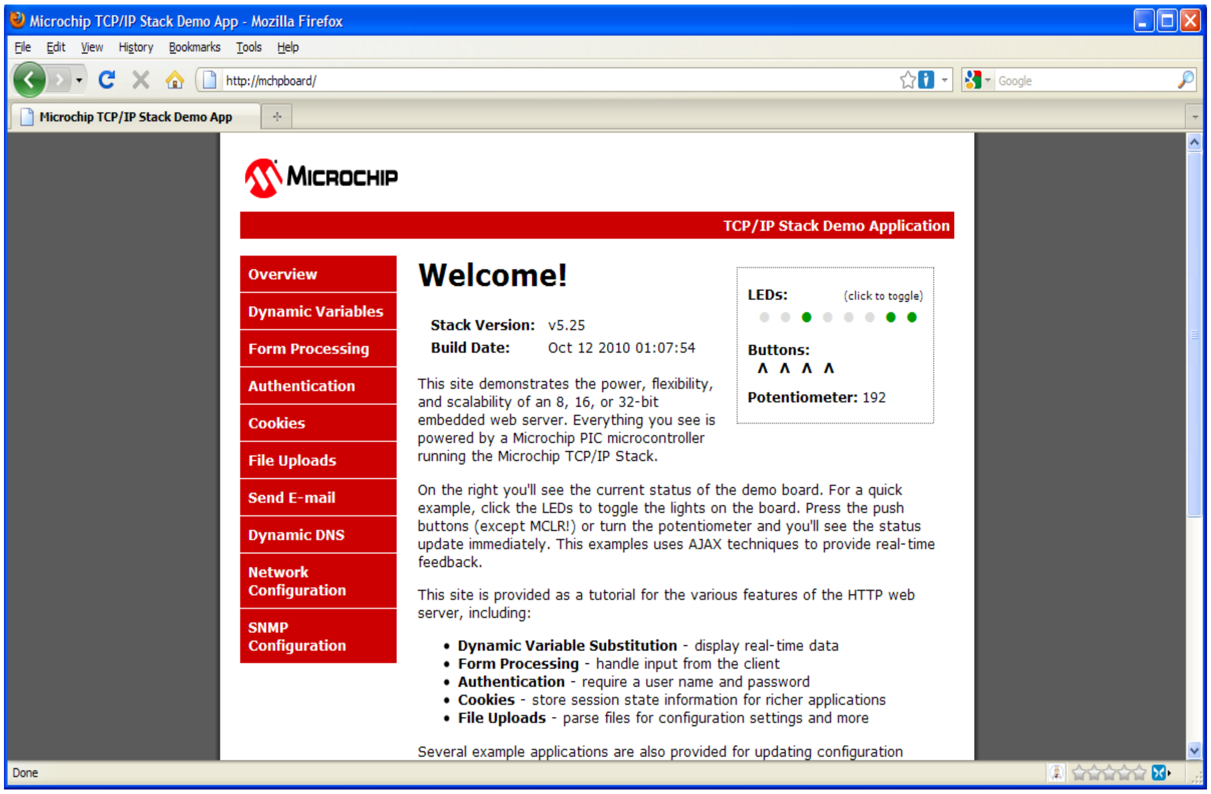


Figure 5.2 Test Setup, PIC18EB with PICkit3 and Ethernet PICtail Plus Board



Welcome!

Stack Version: v5.25
Build Date: Oct 12 2010 01:07:54

LEDs: (click to toggle)

Buttons: ∧ ∧ ∧ ∧

Potentiometer: 192

This site demonstrates the power, flexibility, and scalability of an 8, 16, or 32-bit embedded web server. Everything you see is powered by a Microchip PIC microcontroller running the Microchip TCP/IP Stack.

On the right you'll see the current status of the demo board. For a quick example, click the LEDs to toggle the lights on the board. Press the push buttons (except MCLR!) or turn the potentiometer and you'll see the status update immediately. This examples uses AJAX techniques to provide real-time feedback.

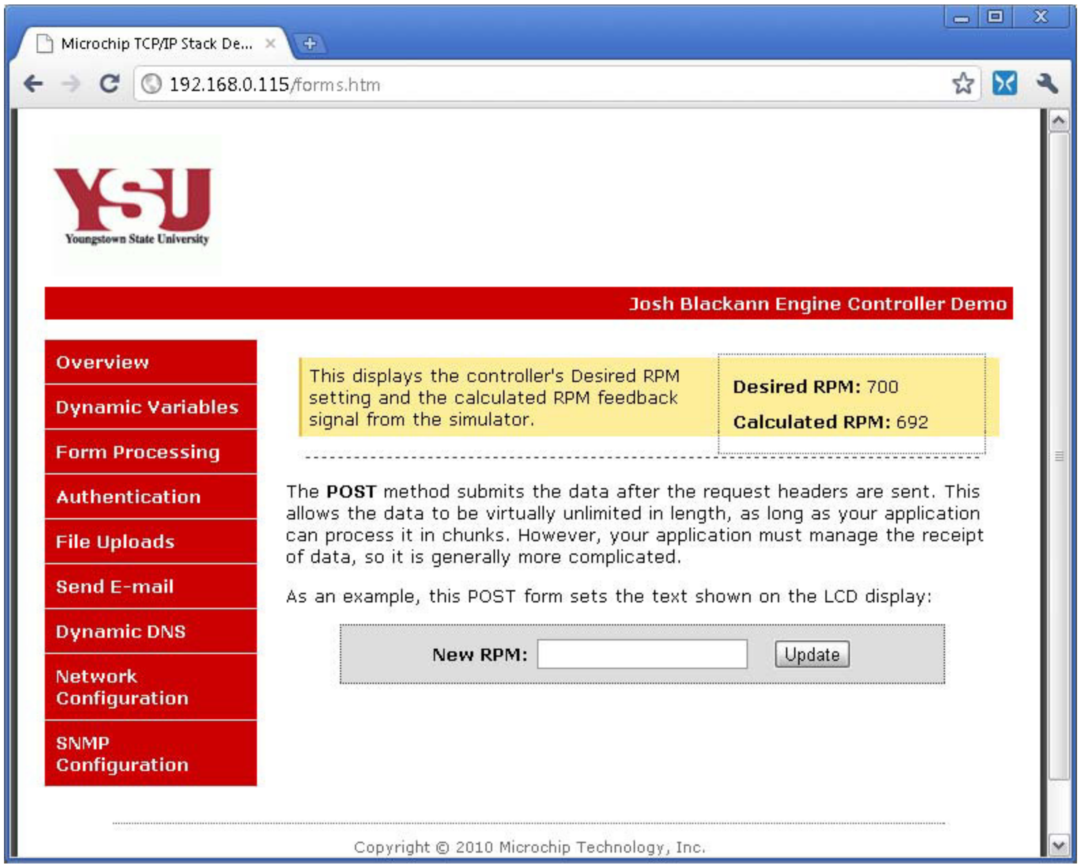
This site is provided as a tutorial for the various features of the HTTP web server, including:

- **Dynamic Variable Substitution** - display real-time data
- **Form Processing** - handle input from the client
- **Authentication** - require a user name and password
- **Cookies** - store session state information for richer applications
- **File Uploads** - parse files for configuration settings and more

Several example applications are also provided for updating configuration

Figure 5.3 Web Server on 87Jll running ENC Demo Application

Microchip provided the source code for the web server example. A free HyperText Markup Language (HTML) editor, Crimson Editor, was used to modify the source code for the web server to develop a custom webpage interface to the application as shown in Figure 5.4.


Microchip TCP/IP Stack De...

192.168.0.115/forms.htm

Josh Blackann Engine Controller Demo

Overview
Dynamic Variables
Form Processing
Authentication
File Uploads
Send E-mail
Dynamic DNS
Network Configuration
SNMP Configuration

This displays the controller's Desired RPM setting and the calculated RPM feedback signal from the simulator.

Desired RPM: 700

Calculated RPM: 692

The **POST** method submits the data after the request headers are sent. This allows the data to be virtually unlimited in length, as long as your application can process it in chunks. However, your application must manage the receipt of data, so it is generally more complicated.

As an example, this POST form sets the text shown on the LCD display:

New RPM: [Update]

Copyright © 2010 Microchip Technology, Inc.

Figure 5.4. Modified web server on the 87J11.

The sample web server demo for the PIC18EB worked but the LCD did not function properly requiring configuration of the SPI port on the 87J11.

With the electronic engine controller and simulator applications previously developed, each application was then separately combined with the ENC demo to add Ethernet connectivity. The ENC demo was written such that it could be configured for a large number of different microcontrollers (from the PIC18 to PIC24 to PIC32 devices) making it a complex undertaking to determine which code was relevant to the 87J11. A flowchart was developed for the engine controller with the integrated TCP/IP stack to help provide an outline as to where the controller functionality should be implemented. Figure 5.5 shows the flowchart of the controller combined with the ENCX24J600 demo.

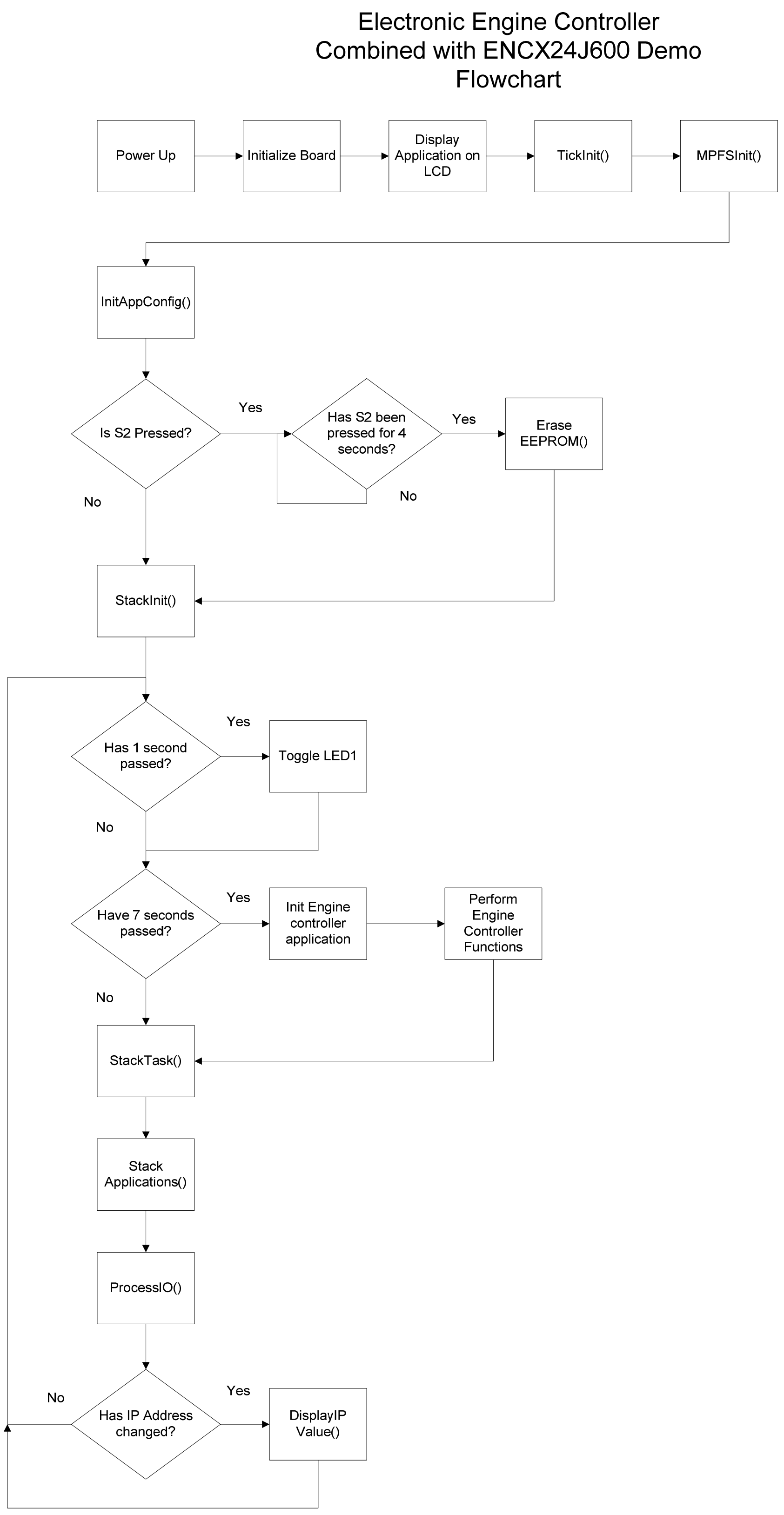


Figure 5.5 Flowchart for Engine Controller with ENCX24J600 Demo.

After analysis of the demo application, the controller code was moved into the MainDemo.c file of the ENC demo application. Several other functions that are noteworthy in the demo are the following functions:

TickInit() – which initializes Timer0 which is used for generating various timing functions used throughout the stack.

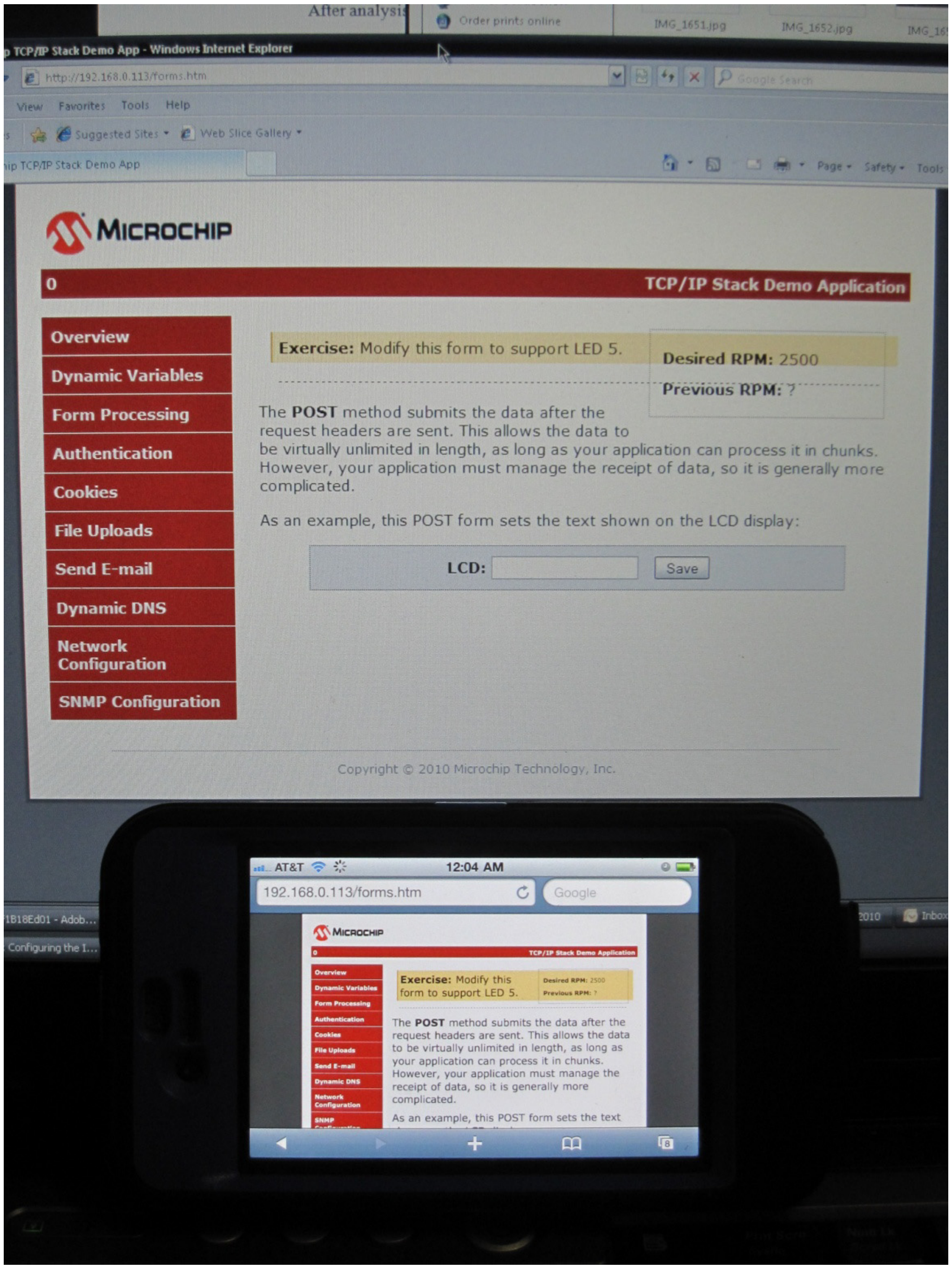
MPFSInit() – which is the function which initializes the device to use the external EEPROM to store the web server file information.

DisplayIPValue() – which is used to display that a new IP address has been assigned to the device.

After the code was combined and tested, the controller application for the 87J11 functioned with the ENC demo to display information via a webpage. The webpage interface from the ENC demo is a good starting point but much of that functionality is not required for interfacing with the controller board.

5.4 Application Setup

Setting up the TCP/IP demo to function on different networks might be a cumbersome task. On a home network, the demo application does not require any modification as the AutoIP.c file was used which obtains a valid IP address automatically. When the board boots up, it initially displays a default IP address of 169.254.1.1 on the LCD. Once a valid IP address is obtained, the LCD is updated to display the valid IP address. For instance, with the board connected to a home network via a router, the IP address is configured as 192.168.0.113, which is a randomly assigned IP address that the router has available. With a PC connected to network, it is possible to navigate to the board by typing in the IP address or the hostname, which in the case of this board is mchpboard. Using a smart phone such as an iPhone, which is connected to the same network, it is possible to access the board by directing the browser to the IP address of the demo kit as shown in Figure 5.6.



Exercise: Modify this form to support LED 5.

Desired RPM: 2500

Previous RPM: ?

The **POST** method submits the data after the request headers are sent. This allows the data to be virtually unlimited in length, as long as your application can process it in chunks. However, your application must manage the receipt of data, so it is generally more complicated.

As an example, this POST form sets the text shown on the LCD display:

LCD: Save

Copyright © 2010 Microchip Technology, Inc.

Figure 5.6. Navigation to the webpage via laptop and browser on a smart phone

Getting the PIC18EB boards to connect to a protected network such as Youngstown State University (YSU) requires some modifications to be performed. YSU's network requires specific IP address to be made available. After receiving valid IP addresses from the YSU Information Technology (IT) department, the PIC18EB simulator and controller need to be updated with the new IP address. The parameters that need modification reside in TCPIPConfig.h. Below is the default IP address, Default Mask and Default Gate which will need to be modified.

```
#define MY_DEFAULT_IP_ADDR_BYTE1        (169ul)
#define MY_DEFAULT_IP_ADDR_BYTE2        (254ul)
#define MY_DEFAULT_IP_ADDR_BYTE3        (1ul)
#define MY_DEFAULT_IP_ADDR_BYTE4        (1ul)

#define MY_DEFAULT_MASK_BYTE1          (255ul)
#define MY_DEFAULT_MASK_BYTE2          (255ul)
#define MY_DEFAULT_MASK_BYTE3          (0ul)
#define MY_DEFAULT_MASK_BYTE4          (0ul)

#define MY_DEFAULT_GATE_BYTE1          (169ul)
#define MY_DEFAULT_GATE_BYTE2          (254ul)
#define MY_DEFAULT_GATE_BYTE3          (1ul)
#define MY_DEFAULT_GATE_BYTE4          (1ul)
```

Additionally a couple modules will need to be disabled by commenting out them.

```
//#define STACK_USE_AUTO_IP             // Dynamic link-layer IP address automatic configuration protocol
//#define STACK_USE_DHCP_CLIENT     // Dynamic Host Configuration Protocol client for obtaining IP address and other parameters
```

Making these modifications, communication to the controller and simulator was possible attached to YSU network as well connected to a home network. To make these changes back and forth as easy as possible, the use of #define statements were added which could be configured based on what the application setup requires.

The following #define statement controls how the application is built.

```
#define WORKING_AT_HOME      1
#define WORKING_AT_YSU       0
```

Chapter 6

Conclusions and Results from Project Development

6.1 Conclusions

This work demonstrates that an electronic engine controller can be simulated successfully in the absence of having a complete system for development. Using development kits from Microchip, it is possible to create an application to simulate the operation of an engine as well as a separate project which will monitor and control the simulator without having to design and build hardware saving time and money. Another benefit of the development kits is that demo code can be used as a starting point for creating new applications. By using the TCP/IP stack provided free of charge from Microchip, a substantial amount of time and effort was saved.

The development of this work was an excellent opportunity to create an application using as many hardware peripherals as possible. Some sample code from Microchip was utilized as a starting point but was greatly expanding upon to complete my specific application. Using the 87J11, I had a chance to learn more about the most advanced 8-bit family of microcontrollers from Microchip. Integrating Ethernet connectivity into the engine controller application required significant review of the Microchip TCP/IP stack and example source code. Without the provided TCP/IP stack the required development time and effort to implement Ethernet connectivity would not have been possible for this work. As I continue to work with microcontrollers in my career, I will need to build more complex applications which require real-time and remote configurability. The addition of Ethernet proved to be an excellent learning exercise which adds real value to the application.

Electronic engine control will continue to be area which will strive for improvements in fuel economy and performance to meet the future requirements as energy concerns increase. The development performed in this work can be further expanded to more complicated engine designs.

6.2 Future Development

While the main goal of this work was to demonstrate the development of an electronic engine controller via a low cost microcontroller development board, additional features could be implemented to provide a more robust application. Future enhancements could include migrating to a higher performance microcontroller such as the 16-bit or 32-bit families which have more flash and RAM memory available as well as more advanced peripherals. Adding the TCPIP bootloader functionality would allow remote firmware upgrades or modifications. Also adding encryption or a secure connection method to the device would offer protection keeping other outside sources from gaining access to controller or device. Additionally, configuring the device to use a dynamic DNS service would allow the device to be accessed as long as it was plugged into a working internet connection.

The integration of other sensors including temperature sensors or pressures sensors could prove useful in developing a full solution to operate an engine. Once a complete system is defined a custom PCB could be designed to provide the required functionality at a lower cost.

Appendices

Appendix A

A.1 Documentation and Software Development Tools

Working on a project with the magnitude of a Master's thesis required that information be collected and compiled continually. There were numerous different applications that proved quite useful for project management. The following applications were key components to the success of this project.

Microsoft's OneNote application was a great tool for collecting and storing information [16]. OneNote worked very similar to a digital notebook providing the ability to create numerous different sections within a notebook for different subjects. It provided many of standard features that are found in numerous other Microsoft products. A key feature that makes OneNote extremely useful is the ability to do a search through all of the content that has been entered. OneNote provided the ability to track progress and keep detailed notes about subjects that might be forgotten.

Google Docs was another useful piece of Software developed by Google which combines much of the functionality found in Microsoft's Office products. Google Docs was used to create and maintain documents, spreadsheets, presentations and other items via any computer which is connected to the internet [17]. The service is free and can be used to share items with other to collaborate on or just as a means of building an online journal.

Another useful application was Microsoft Visio [18], which can be useful for developing block diagrams or creating flow charts providing a visual representation of the different requirements for a project. Drawing higher level presentations of the overall structure of my project provided a framework to follow when working on device firmware. The built-in libraries of Visio offered a large and diverse number of tools which helped create useful diagrams quickly. Visio could pull information from a variety of other Microsoft products or other database applications if necessary.

Tortoise SVN was a source control application which works inside of Windows to keep track of changes to different files [19]. This was used to track modifications of the firmware as the software development moved through its life cycle. TSVN created a repository which acts as a storage location for the countless changes made to source code files. Initially all of the source code was added and committed into the new repository. As changes were made to files in the repository, TSVN marked the files as out of date, providing a user with instant knowledge that files were different from the repository. TSVN allowed bookmarks or revision points to be created by committing files to the repository. As changes are submitted to the repository, notes were also included which provide a description of modifications. TSVN provided a means for keeping track of various different changes that were made to firmware files and the ability to revert to older revision of a file if required.

KDiff3 was another invaluable application for software development. KDiff3 is an open source application which can compare or merge files together [20]. This was especially useful when comparing different revisions of the same file. For example, the demo code from Microchip can be compared to the application code developed and the changes are highlighted by the application. KDiff3 allowed fast navigation through source code to detect the simplest of differences from one file to another. There are other applications similar to this which can be purchased. BeyondCompare is another application that can perform similar and more advanced tasks for purchase [21].

A schematic capture program was essential to create the electrical design as the project began as an idea and moved into a physical implementation. OrCAD [22] and Mentor Graphics [23] were two variations which are industry standards but all come with a high price. There were a couple different Open Source applications which can be downloaded and used for free, TinyCAD [24] is just one example. Many schematic capture programs also come with other software to transform the design into a PCB or to perform simulations of the circuit to verify its performance.

Microchip's MPLAB Integrated Development Envirornment (IDE) is a free, integrated toolset for the development of embedded applications employing Microchip's PIC® and dsPIC® microcontrollers. MPLAB IDE also serves as a single, unified graphical user interface for additional Microchip and third party software and hardware development tools [4]. All code development was done through MPLAB IDE using C18 [7].

Crimson Editor is an open-source multi-purpose, functional text editor [25]. This was used to modify the html code which is used for the web interface.

Appendix B

Equipment

Explorer 18 Development board – development board from Microchip for evaluating the PIC18 8-bit microcontrollers [3]

PICkit 3 – allows programming and debugging of PIC microcontrollers through MPLAB IDE [5]

Fast 100 Mbps Ethernet PICtail Plus Daughter – a plug-in board for the PIC18EB which allows development of the Ethernet applications [10]

Tektronix – TDS3054 – Four channel Color Digital Phosphor Oscilloscope – used for debugging data and logic signals [26]

DIR-615 Wireless N 300 Router – provides internet connectivity between PC and the PIC18EB with EPP [27]

Appendix C

C.1 Abbreviations

PIC18EB – PIC18 Explorer Board. Evaluation kit from Microchip Technology for evaluating the PIC18 family of microcontrollers

PIM – Plug-In Module. A plug in module which has a PIC18 or other Microchip microcontroller which can be used for evaluating different microcontroller solutions

SPI – Serial Peripheral Interface. Part of the Master Synchronous Serial Port Module (MSSP) which is for communicating with our devices or microcontrollers.

ADC- Analog to Digital Converter. A module within the microcontroller which converts an analog input signal into a 10-bit digital number.

Pot – potentiometer. A resistor which can act as an adjustable voltage divider

TSVN – Tortoise SVN. Source control software for capturing revisions to different software files.

PCB – print circuit board. An electrical circuit which is populated with electronic components.

EPP - Fast 100 Mbps Ethernet PICtail Plus Daughter, Ethernet evaluation board which works with the PIC18EB

PLL – Phase Locked Loop circuit which enables a lower frequency oscillator to be used and runs the device at a higher frequency.

TCP– Transmission Control Protocol – provides reliable communication to applications

IP – Internet Protocol – Refers to IPv4 or IPv6

HTML - HyperText Markup Language – one of the primary languages used for webpages

C.2 Definitions

EEPROM - **E**lectrically **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory. Memory that retains its stored information even when power has been removed.

Microcontroller – an integrated circuit which runs firmware to perform a specific function or application. Generally microcontrollers have on board program memory, data memory, and peripherals to make a complete device

Appendix D

D.1 Excel RPM Calculation Values

Injector RPM compared to Position Sensor RPM

		Position Sensor
	Injector	8 times faster than injector
Frequency	RPM	Frequency
0.016666667	1	0.133333333
0.033333333	2	0.266666667
0.05	3	0.4
0.066666667	4	0.533333333
0.083333333	5	0.666666667
0.1	6	0.8
0.116666667	7	0.933333333
0.133333333	8	1.066666667
0.15	9	1.2
9.85	591	78.8
9.866666667	592	78.93333333
9.883333333	593	79.06666667
9.9	594	79.2
9.916666667	595	79.33333333
9.933333333	596	79.46666667
9.95	597	79.6
9.966666667	598	79.73333333
9.983333333	599	79.86666667
10	600	80
10.01666667	601	80.13333333
10.03333333	602	80.26666667
10.05	603	80.4
10.06666667	604	80.53333333
10.08333333	605	80.66666667
10.1	606	80.8
10.11666667	607	80.93333333

Calculated Timer 1 load values for creating RPM Signal

	Timer 1 Controller Calculations			
	Number Divided by 2			
	Timer 1 use 32.768 kHz	Timer 1 using 40MHz	Timer 1 using 40MHz	Timer 1 using 40MHz
	prescaler 1	prescaler 2	prescaler 4	prescaler 8
RPM	Counts required	Counts required	Counts required	Counts required
1	983040	150000000	75000000	37500000
2	491520	75000000	37500000	18750000
3	327680	50000000	25000000	12500000
4	245760	37500000	18750000	9375000
5	196608	30000000	15000000	7500000
14	70217.14286	10714285.71	5357142.857	2678571.429
15	65536	10000000	5000000	2500000
16	61440	9375000	4687500	2343750
17	57825.88235	8823529.412	4411764.706	2205882.353
18	54613.33333	8333333.333	4166666.667	2083333.333
597	1646.633166	251256.2814	125628.1407	62814.07035
598	1643.879599	250836.1204	125418.0602	62709.0301
599	1641.135225	250417.3623	125208.6811	62604.34057
600	1638.4	250000	125000	62500
601	1635.673877	249584.0266	124792.0133	62396.00666
1198	820.5676127	125208.6811	62604.34057	31302.17028
1199	819.883236	125104.2535	62552.12677	31276.06339
1200	819.2	125000	62500	31250
1201	818.5179017	124895.9201	62447.96003	31223.98002
1202	817.8369384	124792.0133	62396.00666	31198.00333
2397	410.1126408	62578.22278	31289.11139	15644.55569
2398	409.941618	62552.12677	31276.06339	15638.03169
2399	409.7707378	62526.05252	31263.02626	15631.51313
2400	409.6	62500	31250	15625
2401	409.4294044	62473.96918	31236.98459	15618.49229
2402	409.2589509	62447.96003	31223.98002	15611.99001
2403	409.0886392	62421.97253	31210.98627	15605.49313
3598	273.2184547	41689.82768	20844.91384	10422.45692
3599	273.1425396	41678.24396	20839.12198	10419.56099
3600	273.0666667	41666.66667	20833.33333	10416.66667

Calculated Timer 3 load values for creating Position Sensor Feedback Signal

	Calculations for Simulator		
	Timer1 = 65535 - counts required		
	Fosc/4		
	number divided by 4 divided by 2		
	Divide by 8 as this would be the time between sensor outputs		
	Timer 1 using 40MHz	Timer 1 using 40MHz	Timer 1 using 40MHz
	prescaler 1	prescaler 2	prescaler 8
RPM	Counts required	Counts required	Counts required
1	37500000	18750000	4687500
2	18750000	9375000	2343750
3	12500000	6250000	1562500
4	9375000	4687500	1171875
5	7500000	3750000	937500
75	500000	250000	62500
76	493421.0526	246710.5263	61677.63158
77	487012.987	243506.4935	60876.62338
78	480769.2308	240384.6154	60096.15385
79	474683.5443	237341.7722	59335.44304
296	126689.1892	63344.59459	15836.14865
297	126262.6263	63131.31313	15782.82828
298	125838.9262	62919.46309	15729.86577
299	125418.0602	62709.0301	15677.25753
300	125000	62500	15625
301	124584.7176	62292.3588	15573.0897
302	124172.1854	62086.09272	15521.52318
303	123762.3762	61881.18812	15470.29703
597	62814.07035	31407.03518	7851.758794
598	62709.0301	31354.51505	7838.628763
599	62604.34057	31302.17028	7825.542571
600	62500	31250	7812.5
601	62396.00666	31198.00333	7799.500832
602	62292.3588	31146.1794	7786.54485
603	62189.05473	31094.52736	7773.631841
3598	10422.45692	5211.22846	1302.807115
3599	10419.56099	5209.780495	1302.445124
3600	10416.66667	5208.333333	1302.083333

Appendix E

E.1 Electronic Engine Controller Firmware

The following segments of source code were used in the Engine Controller application

Interrupt functions

```
// PIC18 Interrupt Service Routines

#pragma interruptlow HighISR
void HighISR(void)
#endif
{
         #if defined(STACK_USE_UARTZTCP_BRIDGE)
        UART2TCPBridgeISR();
        #endif

        #if defined(WF_CS_TRIS)
        WFEintISR();
        #endif // WF_CS_TRIS

        // Check for INT0 interrupt
        // this will most likely be the top of stroke or position
        // sensor interrupt
        if (INTCONbits.INT0IF)              // Position sensor fired
        {
                LED3 = On;
                // clear (reset) flag
                INTCONbits.INT0IF = 0;
                eventsBuff1.eRB0Pressed = 1;        // set flag to show button was pressed
                                                    // sensor activated
                positionSensor++;                   // increment position sensor
                pulseCount++;
                LED3 = Off;
        }

        if      (INTCON3bits.INT3IF)                        // top of stroke indicator
        {
                INTCON3bits.INT3IF = 0;                     // clear INT3 interrupt flag
                LED5 = On;
                eventsBuff1.eTopOfStroke = 1;               // set flag to show
                                                            // sensor activated
                positionSensor = 0;
        }

        // Check for Timer0 Interrupt
        if  (INTCONbits.TMR0IF)                             // did 1mS timer elapse
        {
                //eventsBuff1bits.eTMR0Overflow = 1;
        TickUpdate();
        INTCONbits.TMR0IF = 0;      // clear (reset) flag
}

// a compare interrupt occured, use for generating PWM with RPM out
if(PIR3bits.CCP4IF)
{
        LED4 = !LED4;
        PIR3bits.CCP4IF = 0;        // clear Timer1/Timer3 compare interrupt flag
        if(LED6 == On)
        {
                LED6 = Off;         // based on capture 4 module and Timer1 compare
                                    // turn off Output to create a PWM
        }
}

// timer 1 used to generate the different RPM rate
if (PIR1bits.TMR1IF)        // Timer 1, new ocillation required
{
        reloadTimer1();
        PIR1bits.TMR1IF  = 0;       // clear interrupt flag
        PIE1bits.TMR1IE = 1;        // enable the TMR1 overflow interrup
        if(injectorState == Off)
        {
                LED6 = On;
                injectorState = On;
        }
        else if(injectorState == On)
        {
                LED6 = Off;
                injectorState = Off;
        }
}

// check A to D flag to see if conversion completed
if (PIR1bits.ADIF)          // Check A to D flag
{
        PIR1bits.ADIF = 0;                  // clear A to D conversion complete flag
        LED2 = !LED2;
}

// check if Timer3 overflowed
// creates a 10mS timer
if (PIR2bits.TMR3IF)        // Check Timer 3 flag
{
        PIR2bits.TMR3IF = 0;                // clear Timer3 overflow flag
        eventsBuff1.eTMR3Overflow = 1;
        LED7 = !LED7;
        count100times++;
        if(count100times == 100)
        {
                secondCount++;
                eventsBuff1.e1SecTimerOverflow = 1;
```

```
					count100times = 0;
					LED0_IO ^= 1;
				}
				reloadTimer3();
			}
		}		// end of interrupt vectors


		*********
		// This is where the RPM signal is generated
		if(running == 1)
		{
			if(initial_startup == 1)
			{
				RPM = 0;
				LCDRPMCount(RPM);
				sprintf((char *)LCDText, (far rom char*)"Engine Startup");
				LCDUpdate();
				temp_int = desiredRPM / 100;
				desiredRPMchar = (unsigned char) temp_int;
				LED6 = On;
				exit = 0;
				running_mode = STARTUP_MODE_IN_STARTUP;
				// determine how many steps should be taken to accelerate the device up to speed
				temp_RPM_startup_steps = determineRPMStartupSteps(desiredRPM);
				// determine the startup step times
				determineRPMStartupValues(temp_RPM_startup_steps);

				i = 0;
				/* if any startup values are less than 16,
				 * update startup values to be larger than 16
				 * as 16 is as low of a frequency the controller can generate
				 */
				if(temp_RPM_startup_values[i] < 16)
				{
					temp_RPM_startup_values[i] = 16;
					reloadTimer1Value = determineTimer1Setup(temp_RPM_startup_values[i]);
				}
				initial_startup = 0;
				startup_Time = TickGet();  // store the current time
				T1CONbits.TMR1ON = 1;			//turn on Timer1
			}
			if(running_mode == STARTUP_MODE_IN_STARTUP)
			{
				switch(running_mode)
				{
					case STARTUP_MODE_IN_STARTUP:
					{
						TickCount = TickGet();
						// sit in each startup for 5 seconds or if the RPM rate increases above
						//50% of current step, advance to next step
						if(((startup_Time + 10*TICK_SECOND) <= (TickGet())) || ((RPM << 1) >=
						temp_RPM_startup_values[i]))
```

```
          {
                    i++;        // increment to next startup value
                    startup_Time = TickGet();  // store the current time again
                    if(i > (temp_RPM_startup_steps -1))
                    {
                              running_mode = UP_TO_SPEED_IN_STARTUP;
                              reloadTimer1Value =
                              determineTimer1Setup(desiredRPM);

                              /* to create a PWM signal, compare module will be
                               * used  CCP4CON will need to match timer1 then
                               * cause an interrupt.Max duty cycle is currently 50%
                              */
                              PWMvalue = (65535 - reloadTimer1Value) << 1;
                              PWMvalue = PWMvalue * injectorTime;
                              PWMvalue =((PWMvalue / 100) +
                              reloadTimer1Value);
                              CCPR4 =(unsigned int) PWMvalue;
                              CCP4CON = 0x0A;             // enable Compare mode:
                                                          // generate software int
                                                          // on compare
                              PIE3bits.CCP4IE = 1;        // CCP4 interrupt enable
                              break;
                    }
/* If any startup values are less than 16,
 * update startup values to be larger than 16
 * as 16 is as low of a frequency the controller can generate.
 * Not sure why this is here and also above
 */
                    if(temp_RPM_startup_values[i] < 16)
                    {
                              temp_RPM_startup_values[i] = 16;
                              reloadTimer1Value =
                    determineTimer1Setup(temp_RPM_startup_values[i]);
                    }
                    /*  Startup value is larger than 16, calculate value required for
                    * Timer1  to generate the correct RPM
                    */
                    else
                    {
                              reloadTimer1Value =
                    determineTimer1Setup(temp_RPM_startup_values[i]);
                    }
          }
}
          break;

/* Motor controller has gone through it startup sequence and should be
 * at full desired speed
 */
case UP_TO_SPEED_IN_STARTUP:
{
          initial_startup = 0;
```

```
                                    sprintf((char *)LCDText, (far rom char*)"Eng Up to Speed");
                                    LCDUpdate();
                                    running_mode = STARTUP_MODE;
                                    temp_Timer1RPMCalc = determineTimer1Setup(desiredRPM);
                              }
                                    break;

                        }
            }           // running_mode == STARTUP_MODE_IN_STARTUP)

      if(eventsBuff1.e1SecTimerOverflow == 1)
      {
            //perform different operations based on 1 second increments
            // check to see if we had a new RPM setting via TCPIP
            if(eventsBuff1.eRPMChangedViaTCPIP == 1)
            {
                  eventsBuff1.eRPMChangedViaTCPIP = 0;
                  // determine if new RPM is faster or slower than previous RPM
                  if(desiredRPM > previousDesiredRPM)
                  {
                        // new RPM is faster than previous setting
                        running_mode = NEW_RPM_VIA_TCPIP;
                  }
                  else if(desiredRPM < previousDesiredRPM)
                  {
                        // new RPM is slower than previous setting
                        running_mode = NEW_RPM_VIA_TCPIP;
                  }
                  modifiedPreviousDesiredRPM = previousDesiredRPM;
            }

            // new RPM has been set, slow down or speed up as required.
            if(running_mode == NEW_RPM_VIA_TCPIP)
            {
                  unsigned int newRPMDifference;
                  if(desiredRPM > modifiedPreviousDesiredRPM)
                  {
                        newRPMDifference = desiredRPM - modifiedPreviousDesiredRPM;
                        if(newRPMDifference > 100)
                        {
                              modifiedPreviousDesiredRPM = modifiedPreviousDesiredRPM + 50;
                              reloadTimer1Value =
                              determineTimer1Setup(modifiedPreviousDesiredRPM);
                        }
                        else if(newRPMDifference > 50)
                        {
                              modifiedPreviousDesiredRPM = modifiedPreviousDesiredRPM + 25;
                              reloadTimer1Value =
                              determineTimer1Setup(modifiedPreviousDesiredRPM);
                        }
                        else if(newRPMDifference > 25)
                        {
                              modifiedPreviousDesiredRPM = modifiedPreviousDesiredRPM + 10;
```

```
				reloadTimer1Value =
				determineTimer1Setup(modifiedPreviousDesiredRPM);
			}
			else if(newRPMDifference > 10)
			{
				modifiedPreviousDesiredRPM = modifiedPreviousDesiredRPM + 5;
				reloadTimer1Value =
				determineTimer1Setup(modifiedPreviousDesiredRPM);
			}
			else if(newRPMDifference > 0)
			{
				modifiedPreviousDesiredRPM = modifiedPreviousDesiredRPM + 1;
				reloadTimer1Value =
				determineTimer1Setup(modifiedPreviousDesiredRPM);
			}
		}
		else if(desiredRPM < modifiedPreviousDesiredRPM)
		{
			newRPMDifference =  modifiedPreviousDesiredRPM - desiredRPM;
			if(newRPMDifference > 100)
			{
				modifiedPreviousDesiredRPM = modifiedPreviousDesiredRPM - 50;
				reloadTimer1Value =
				determineTimer1Setup(modifiedPreviousDesiredRPM);
			}
			else if(newRPMDifference > 50)
			{
				modifiedPreviousDesiredRPM = modifiedPreviousDesiredRPM - 25;
				reloadTimer1Value =
				determineTimer1Setup(modifiedPreviousDesiredRPM);
			}
			else if(newRPMDifference > 25)
			{
				modifiedPreviousDesiredRPM = modifiedPreviousDesiredRPM - 10;
				reloadTimer1Value =
				determineTimer1Setup(modifiedPreviousDesiredRPM);
			}
			else if(newRPMDifference > 10)
			{
				modifiedPreviousDesiredRPM = modifiedPreviousDesiredRPM - 5;
				reloadTimer1Value =
				determineTimer1Setup(modifiedPreviousDesiredRPM);
			}
			else if(newRPMDifference > 0)
			{
				modifiedPreviousDesiredRPM = modifiedPreviousDesiredRPM - 1;
				reloadTimer1Value =
				determineTimer1Setup(modifiedPreviousDesiredRPM);
			}
		}
		else if(desiredRPM == modifiedPreviousDesiredRPM)
		{
			running_mode = UP_TO_SPEED_IN_STARTUP;		// ramping RPM finished
```

```
                previousRunning_mode = NEW_RPM_VIA_TCPIP;
            }
    }

    if(running_mode == UP_TO_SPEED_IN_STARTUP)
    {
            // check to see if RPM is going faster or slower than the desired RPM
            static unsigned char underSpeed = 0;        // engine load down
            static unsigned char overSpeed = 0;         // engine load has been reduced

            if(previousRunning_mode != UP_TO_SPEED_IN_STARTUP)
            {
                initial_startup = 0;
                sprintf((char *)LCDText, (far rom char*)"Eng Up to Speed");
                LCDUpdate();
                //running_mode = STARTUP_MODE;
                previousRunning_mode = UP_TO_SPEED_IN_STARTUP;
            }

            temp_Timer1RPMCalc = determineTempTimer1Setup(desiredRPM);
            fourPercentOfDesiredRPM = desiredRPM >> 4;                  // divide by 16

            // check to see if RPM is higher than desired RPM
            if(calculated_RPM > (desiredRPM + fourPercentOfDesiredRPM))         // if cal RPM is
                                                    //6% higher or more than desired RPM
            {
                overSpeed++;                        // if overspeed for a couple cycles
                if(overSpeed == 10)                 // do something to slow engine down
                {
                    running_mode = GOING_FASTER_THAN_DESIRED;
                }
            }
            else if (calculated_RPM < (desiredRPM - fourPercentOfDesiredRPM))   // if cal RPM is
                                                    //6% lower or smaller than desired RPM
            {
                underSpeed++;                       // if underspeed for a couple cycles
                if(underSpeed == 10)        // do something to slow engine down
                {
                    running_mode = GOING_SLOWER_THAN_DESIRED;
                overSpeed = 0;
                }
            }
            else
            {
                underSpeed = 0;                     // clear as RPM is within tolerance
                overSpeed = 0;
            }
    }       // running_mode == UP_TO_SPEED_IN_STARTUP

    // if RPM is faster than desired SLOW down
    if(running_mode == GOING_FASTER_THAN_DESIRED)
    {
            unsigned int twentyPerBelowDesiredRPM;
```

```
}
        static unsigned int tempRPMDecrease = 0;
        static unsigned char RPMstuckTooFast = 0;
        static unsigned char slowDecrease = 0;

        if(previousRunning_mode != GOING_FASTER_THAN_DESIRED)
        {
                sprintf((char *)LCDText, (far rom char*)"Slowing Eng RPM");
                LCDUpdate();
                previousRunning_mode = GOING_FASTER_THAN_DESIRED;
                tempRPMDecrease = desiredRPM - 2;
        }
        tempRPMdifference = calculated_RPM - desiredRPM;        // what is the difference
                                                                // between desired and calculated
        // engine should not slow down more than 20 below desired RPM
        // calculate what RPM would be 20 percent below desired RPM

        twentyPerBelowDesiredRPM = (desiredRPM * 4) / 5;
        // have we slowed down the RPM by 20% less than desired or more
        if(desiredRPM > twentyPerBelowDesiredRPM)
        {
                // RPM has not slowed by 20%, continue to slow RPM
                if(slowDecrease == 3)
                {
                        tempRPMDecrease-=1;
                        slowDecrease = 0;
                }
                slowDecrease++;
                temp_Timer1RPMCalc = determineTempTimer1Setup(tempRPMDecrease);
                reloadTimer1Value = temp_Timer1RPMCalc;

        }
        else
        {
                // if we have slowed RPM by 20% and still not back to speed
                // what a couple more cycles than indicate Stuck
                RPMstuckTooFast++;
                if(RPMstuckTooFast == 5)
                {
                        sprintf((char *)LCDText, (far rom char*)"Eng RPM Too Fast");
                        LCDUpdate();
                }
        }

        // don't slow down more than 20% of desired          RPM
        // take small steps to try to bring system back into balance
        if(calculated_RPM < (desiredRPM + 10))
        {
                running_mode = UP_TO_SPEED_IN_STARTUP;
                tempRPMDecrease = 0;
                RPMstuckTooFast = 0;
        }
}
```

```
// if RPM is slower than desired SPEED Up
if(running_mode == GOING_SLOWER_THAN_DESIRED)
{
        unsigned int twentyPerAboveDesiredRPM;
        static unsigned int tempRPMIncrease = 0;
        static unsigned char RPMstuckTooSlow = 0;
        static unsigned char slowIncrease = 0;
        if(previousRunning_mode != GOING_SLOWER_THAN_DESIRED)
        {
                // Display controller is attempting to speed up.
                sprintf((char *)LCDText, (far rom char*)"Acceling Eng RPM");
                LCDUpdate();
                tempRPMIncrease = desiredRPM + 2;
                previousRunning_mode = GOING_SLOWER_THAN_DESIRED;
        }

        tempRPMdifference = desiredRPM - calculated_RPM; // what is the difference
                                                         // between desired and calculated

        // engine should not slow down more than 20 below desired RPM
        // calculate what RPM would be 20 percent above desired RPM
        twentyPerAboveDesiredRPM = (desiredRPM * 6) / 5;

        if(calculated_RPM < twentyPerAboveDesiredRPM)
        // don't speed up more than 20% above of desired RPM
        //if(tempRPMdifference < (twentyPerBelowDesiredRPM - desiredRPM))
        {
                // RPM has not sped up by 20%, continue to speed up RPM
                //tempRPMIncrease = calculated_RPM + 2;
                if(slowIncrease == 3)
                {
                        tempRPMIncrease+=1;
                        slowIncrease = 0;
                }
                slowIncrease++;
                temp_Timer1RPMCalc = determineTempTimer1Setup(tempRPMIncrease);
                reloadTimer1Value = temp_Timer1RPMCalc;
        }
        else
        {
                // if we have sped up RPM by 20% and still not back to speed
                // what a couple more cycles than indicate Stuck
                RPMstuckTooSlow++;
                if(RPMstuckTooSlow == 5)
                {
                        sprintf((char *)LCDText, (far rom char*)"Eng RPM Too Slow");
                        LCDUpdate();
                }
        }
        // take small steps to try to bring system back into balance

        if(calculated_RPM > (desiredRPM - 10))
        {
```

```
                        running_mode = UP_TO_SPEED_IN_STARTUP;
                        tempRPMIncrease = 0;
                        RPMstuckTooSlow = 0;
                }
        }
}

if(eventsBuff1.e1SecTimerOverflow == 1)                // if 1 second elasped do something
{
        eventsBuff1.e1SecTimerOverflow = 0;            // clear until 1 second timer
                                                       // overflows again
        countsPer1Sec = pulseCount;         // copy into other variable to performance calculation
        pulseCount = 0;                     // reset pulse count
        calculated_RPM = calculateRPM(countsPer1Sec);
        // if no PulseCount recorded for more than 5 seconds, assume no signal
        if(countsPer1Sec == 0 )
        {
                if(no_RPM_Five_Times == 5)
                {
                        reset_counts = 1;              // set so next time a RPM is to be
                                                       // calculated all values will be
                }
                else
                {
                        no_RPM_Five_Times++;           // increment how many times no
                                                       //feedback signal recorded
                }
        }
        else
        {
                no_RPM_Five_Times = 0;                 // clear as we have a feedback signal
        }

        calculated_RPM = calculated_RPM >> 3;
                LCDRPMCount(calculated_RPM);           // from timing measurement this takes
                                                       //approximately 15.4mS
        }



        if(PUSH_BUTTON_2 == 0)
        {
                delay_ms(30);                          // debounce for 30ms
                if(PUSH_BUTTON_2 == 0)
                {
                        INJECTOR1 = Off;               // turn injector off
                        INJECTOR2 = Off;
                        LED5 = Off;                            // turn off injector
                        LED6 = Off;                            // turn off RPM generation signal
                        T1CONbits.TMR1ON = 0;                  //turn off Timer1 to stop creating RPM
                                                               // Signal
                        PIE3bits.CCP4IE = 0;                   // disable CCP4 interrupt (used for PWM)
                        // turn off any other connections which are used with the injectors
                        LCDDisplay((char *)"Output Stopped", 0, TRUE);
                reset_counts = 1;               // reset RPM counter values
                exit = 1;           // button still low, set to exit
            }
            setup = 1;
            running = 0;
            initial_startup = 0;
            menu_state = INITIAL_MENU_STATE;
            while(PUSH_BUTTON_2 == 0);
            delay_ms(100);
            LCDDisplay((char *)"Return to Setup", 0, TRUE);
        }
}

*******
// this function reloads Timer1
void reloadTimer1(void)    //unsigned int reloadTimer1Value)
{
        // might be a problem loading TIMER1 due to Errata
        T1CONbits.TMR1ON = 0;   //      turn off Timer 1
        iu.i = reloadTimer1Value;
        TMR1H = iu.bytes[1];
        TMR1L = iu.bytes[0];
        T1CONbits.TMR1ON = 1;   //      turn on Timer 1
}

*******
// this function reloads Timer3
void reloadTimer3(void)    //unsigned int reloadTimer1Value)
{
        // might be a problem loading TIMER1 due to Errata
        T3CONbits.TMR3ON = 0;   //      turn off Timer 3
        TMR3H = 0xCF;           // load TMR0H for a 1mS time delay
        TMR3L = 0xC1;           // must have CLK at 40 MHz, 10 MHz with PLL * 4
        T3CONbits.TMR3ON = 1;   //      turn on Timer 3
}

*******
// this function calculates Timer1 count value which creates RPM signal
unsigned int determineTimer1Setup(unsigned int rpm)
{
        // calculate what value needs to be loaded into Timer1
        // to cause the correct overflow value to create correct frequency
        unsigned long multResult;
        unsigned int timer1LoadValue;
        unsigned char timer1Mode;

        // determine prescaler based on rpm range
        // prescaler set to 1
        if ((rpm >= 16) && (rpm <= 599))     //if using 10 MHz with PLL = 40 MHz
        {
                T1CONbits.T1CKPS1 = 0;          // 1:1 prescaler
                T1CONbits.T1CKPS0 = 0;
                timer1Mode = TIMER1_32KHZ_MODE;
```

```
		T1CONbits.TMR1CS = 1;			// use external 32 kHz clock
	}

	// use 40MHz internal clock (PLL enabled)
	// prescaler set to 8
	else if((rpm >= 600) && (rpm <= 1199))
	{
		T1CONbits.T1CKPS1 = 1;			// 1:8 prescaler
		T1CONbits.T1CKPS0 = 1;
		timer1Mode = TIMER1_INTERNAL_MODE;
		T1CONbits.TMR1CS = 0;			// use internal clock
	}
	// prescaler set to 4
	else if((rpm >= 1200) && (rpm <= 2399))
	{
		T1CONbits.T1CKPS1 = 1;			// 1:4 prescaler
		T1CONbits.T1CKPS0 = 0;
		timer1Mode = TIMER1_INTERNAL_MODE;
		T1CONbits.TMR1CS = 0;			// use internal clock
	}
	// prescaler set to 2
	else if((rpm >= 2400) && (rpm <= 3600))
	{
		T1CONbits.T1CKPS1 = 0;			// 1:2 prescaler
		T1CONbits.T1CKPS0 = 1;
		timer1Mode = TIMER1_INTERNAL_MODE;
		T1CONbits.TMR1CS = 0;			// use internal clock
	}

	if(timer1Mode == TIMER1_32KHZ_MODE)
	{
		// if prescaler set to 1
		if((T1CONbits.T1CKPS1 == 0) && (T1CONbits.T1CKPS0 == 0))
		{
			multResult = 983040 / rpm;
		}
		timer1LoadValue = 65535 - (unsigned int) multResult;
		return timer1LoadValue;
	}

	if(timer1Mode == TIMER1_INTERNAL_MODE)
	{
		// based on prescaler, determine number of counts required for Timer1
		// if prescaler set to 8
		if((T1CONbits.T1CKPS1 == 1) && (T1CONbits.T1CKPS0 == 1))
		{
			multResult = 37500000 / rpm;		// if INTOSC is 10MHz with PLL en = 40 MHz
		}
		// if prescaler set to 4
		else if((T1CONbits.T1CKPS1 == 1) && (T1CONbits.T1CKPS0 == 0))
		{
			multResult = 75000000 / rpm;		// if INTOSC is 10MHz with PLL en = 40 MHZ
		}
```

```
                    // if prescaler set to 2
                    else if((T1CONbits.T1CKPS1 == 0) && (T1CONbits.T1CKPS0 == 1))
                    {
                              multResult = 150000000 / rpm;       // if INTOSC is 10MHz with PLL en = 40 MHZ
                    }
                    timer1LoadValue = 65535 - (unsigned int) multResult;
                    return timer1LoadValue;
          }
}         // determineTimer1Setup


******
// this function calculates the feedback position RPM

unsigned int calculateRPM(unsigned int countsPer1Sec)
{
          unsigned char static calculate_RPM_Pointer = 0;
          unsigned char static cal_RPM_Pointers_Filled = 0;
          unsigned int static RPM_count_array[10] = {0,0,0,0,0,0,0,0,0,0};
          unsigned int cal_RPM_Result;
          unsigned int cal_RPM_Result_Previous;
          unsigned int previous_RPM_count_array_element;
          unsigned int remainder = 0;
          unsigned char i;

          if(reset_counts == 1)                        // if restart has occured clear all recordings
          {
                    calculate_RPM_Pointer = 0;
                    cal_RPM_Pointers_Filled = 0;
                    for(i = 0; i < 10; i++)
                    {
                              RPM_count_array[i] = 0;
                    }
                    reset_counts = 0;
          }

          // use if you want to determine RPM change over 15 continuous samples
          cal_RPM_Result_Previous = calculated_RPM;
          previous_RPM_count_array_element = RPM_count_array[calculate_RPM_Pointer];
          RPM_count_array[calculate_RPM_Pointer] = countsPer1Sec;
          calculate_RPM_Pointer++;
          if(countsPer1Sec >0)
          {
                    cal_RPM_Pointers_Filled++;          // only add count if anything other than 0
          }
          else
          {
                    if(previous_RPM_count_array_element > 0)            // erase a pointer unless no
                                                                        //pointers captured
                    {
                              cal_RPM_Pointers_Filled--;
                    }
```

```
}

cal_RPM_Result = 0;
if(calculate_RPM_Pointer == 10)
{
            calculate_RPM_Pointer = 0;
}
if(cal_RPM_Pointers_Filled == 11)
{
            cal_RPM_Pointers_Filled = 10;
}
if(cal_RPM_Pointers_Filled > 0)
{
            for(i = 0; i < 10; i++)            // used to be cal_RPM_Pointers_Filled, can't current assume
                                               //filled from [0]
            {
                        cal_RPM_Result += RPM_count_array[i];
            }
            if(cal_RPM_Pointers_Filled == 10)
            {
                        return    cal_RPM_Result = cal_RPM_Result * 6;
            }
            else if(cal_RPM_Pointers_Filled == 9)
            {
                        remainder = (cal_RPM_Result <<1)/3;                    //here is the 0.66
                        return    cal_RPM_Result = (cal_RPM_Result * 6) + remainder;            // 6.66
            }
            else if(cal_RPM_Pointers_Filled == 8)
            {
                        remainder = (cal_RPM_Result >> 1);                    //here is the 0.5
                        return    cal_RPM_Result = (cal_RPM_Result * 7) + remainder;            //7.5
            }
            else if(cal_RPM_Pointers_Filled == 7)
            {
                        remainder = (cal_RPM_Result >> 1);                    //here is the 0.5
                        return    cal_RPM_Result = (cal_RPM_Result << 3) + remainder;          //8.57...
            }
            else if(cal_RPM_Pointers_Filled == 6)
            {
                        return    cal_RPM_Result = cal_RPM_Result * 10;
            }
            else if(cal_RPM_Pointers_Filled == 5)
            {
                        return    cal_RPM_Result = cal_RPM_Result * 12;
            }
            else if(cal_RPM_Pointers_Filled == 4)
            {
                        return    cal_RPM_Result = cal_RPM_Result * 15;
            }
            else if(cal_RPM_Pointers_Filled == 3)
            {
                        return    cal_RPM_Result = cal_RPM_Result * 20;
            }
```

```
		else if(cal_RPM_Pointers_Filled == 2)
		{
			return	cal_RPM_Result = cal_RPM_Result * 30;
		}
		else if(cal_RPM_Pointers_Filled == 1)
		{
			return	cal_RPM_Result = cal_RPM_Result * 60;
		}
	}		// if(cal_RPM_Pointers_Filled > 0)
	else			// if no calculated pointers are filled
	{
		return	cal_RPM_Result = 0;
	}
} //calculateRPM()

*********
// this function determines the startup step values during initial engine ramp up
unsigned char determineRPMStartupSteps(unsigned int temp_RPM)
{
	if(temp_RPM > 0x1000)
	{
		return 0x09;
	}
	else if(temp_RPM > 0x0800)
	{
		return 0x08;
	}
	else if(temp_RPM > 0x0400)
	{
		return 0x07;
	}
	else if(temp_RPM > 0x0200)
	{
		return 0x06;
	}
	else if(temp_RPM > 0x0100)
	{
		return 0x05;
	}
	else if(temp_RPM > 0x0080)
	{
		return 0x04;
	}
	else if(temp_RPM > 0x0040)
	{
		return 0x03;
	}
	else if(temp_RPM > 0x0020)
	{
		return 0x02;
	}
	else if(temp_RPM > 0x0010)
	{
```

```
                return 0x01;
        }
}

void determineRPMStartupValues(unsigned char temp_RPM_steps)
{
        unsigned char i;
        for(i = 0; i < temp_RPM_steps; i++)
        {
                temp_RPM_startup_values[i] = (desiredRPM >> (temp_RPM_steps - i));
        }
}
```

E.2 Electronic Engine Simulator Firmware

```
// -------------------- Interrupt Service Routines --------------------------
#pragma interrupt InterruptServiceHigh  // "interrupt" pragma also for high priority
void InterruptServiceHigh(void)
{
	 // Check to see what caused the interrupt
	 // (Necessary when more than 1 interrupt at a priority level)

	// Check for INT0 interrupt
	// this will most likely be the top of stroke or position
	// sensor interrupt
	if (INTCONbits.INT0IF)				// Position sensor fired
	 {
		INTCONbits.INT0IF = 0;			// clear (reset) flag
		eventsBuff1bits.eRB0Pressed = 1;	// set flag to show button was pressed
							// sensor activated
		positionSensor++;			// increment position sensor
		pulseCount++;
	}

	if(INTCON3bits.INT3IF)				// top of stroke indicator
	{
		INTCON3bits.INT3IF = 0;				// clear INT3 interrupt flag
		LED5 = On;
		eventsBuff1bits.eTopOfStroke = 1;		// set flag to show  sensor activated
		positionSensor = 0;
	}

	// check timer1 overflow to see if 1 second has elapsed
	if (PIR1bits.TMR1IF)		// Timer 1, did 1 sec timer elapse
	{
		//TMR1H |= 0x80;				// preload 1 sec overflow
		PIR1bits.TMR1IF  = 0;	// clear interrupt flag
		PIE1bits.TMR1IE = 1;	// enable the TMR1 overflow interrup
		TMR1H |= 0x80;			// preload 1 sec overflow
		LED1 = !LED1;
		eventsBuff1bits.e1SecTimerOverflow = 1;		//set event flag
	}

	// check to see if A to D conversion is complete
	if (PIR1bits.ADIF)		// A to D conversion complete?
	{
		PIR1bits.ADIF = 0;		// clear A to D conversion complete flag
	}

	//check to see if position sensor signal must toggle
	if(PIR2bits.TMR3IF)
	{
		INJECTOR_OUTPUT_LED = Off;
		PIR2bits.TMR3IF = 0;				// clear Timer 3 overflow flag
```

```
                reloadTimer3();
                #if TIMER3_RPM_GENERATOR
                POSITION_SENSOR_LED = !POSITION_SENSOR_LED;   // toggle position sensor LED
                positionSensorToggleCount++;
                if((positionSensorToggleCount == 16))          // && (POSITION_SENSOR_LED == On))
                {
                        positionSensorToggleCount = 0;
                        INJECTOR_OUTPUT_LED = On;
                }
                #endif
        }
}  // return from high-priority interrupt

************
// This is the main function
void main (void)
{
        unsigned char load_Modifier;  // used to determine how RPM signal should be modified
        unsigned char load_Modifier_Add; // use to indicate must add load value to RPM
                                         // value to make a slower RPM rate
        unsigned char load_Modifier_Subtract;        // use to indicate must subtract load value to RPM
                                                     // value to make a faster RPM rate
        unsigned long multResult;
        unsigned int timer3LoadValue;
        unsigned char timer3Mode;
        unsigned int rpm;

        OSCTUNEbits.PLLEN = 1;                 // enable the PLL
        initIO();
        LCDInit();
        LCDDisplay((char *)"Engine Simulator", 0, TRUE);
        positionSensorToggleCount = 1;
        RPM_Mode_state = 1;
        enter = 1;
        RPMChanged = YES;
        initInterrupts();    // setup interrupts
        INTCONbits.GIE = 1;                    // enable all interrupts
        eventsBuff1bits.eStartup = 1;  // set bit to indicate we are in startup mode
        timer3Init();
        seconds_Passed = 0;
        LCDErase();
        LCDUpdate();
        eventsBuff2bits.eNoRPMSignal = 1;

        while(1) // infinite loop here
        {
                if(eventsBuff1bits.e1SecTimerOverflow == 1)
                {
                        if(eventsBuff1bits.eStartup == 1)     // perform start up ramping
                        {
                                if(calculated_RPM > 0)
                                {
                                        eventsBuff2bits.eNoRPMSignal = 0;
```

```
						// clear to indicate there is an RPM
				LCDEraseLine1();
				seconds_Passed++;
				if(seconds_Passed > 10)
				{
					eventsBuff1bits.eStartup = 0;		// clear bit to no longer
										//perform startup functions
				}
				timer3Value = determineTimer3Setup(calculated_RPM);
				timer3Load(timer3Value);
			}
		}

		// if no RPM signal stop pulsing output and display not signal
		if(calculated_RPM == 0)
		{
			T3CONbits.TMR3ON = 0;			// turn off timer 3
			LED6 = Off;				// turn off feed back RPM signal
			eventsBuff2bits.eNoRPMSignal = 1;// set to indicate no RPM signal
			LCDErase();
			sprintf((char *)LCDText, (far rom char*)"No RPM Signal");
			LCDUpdate();
		}

		// if not in start up and the previous calculated RPM does not match the current
		// calculated RPM and 1 sec has transpired
		// determine new RPM rate.
		if((eventsBuff1bits.eStartup == 0) && (previous_calculated_RPM != calculated_RPM))
		{
			if(load_Modifier_Add == 1)		// load increasing
			{
				// add modifier value to slow down output RPM
				calculated_RPM = calculated_RPM + load_Modifier_value;
			}
			if(load_Modifier_Subtract == 1)	// load decreasing
			{
				// subtract modifier value to slow down output RPM
				calculated_RPM = calculated_RPM - load_Modifier_value;
			}
			timer3Value = determineTimer3Setup(calculated_RPM);
			timer3Load(timer3Value);
		}

		// calculate RPM based on 1 second checks
		eventsBuff1bits.e1SecTimerOverflow = 0;	// clear until 1 second timer
							// overflows again
		countsPer1Sec = pulseCount;		// copy into other variable to
							// performance calculation
		// if no pulseCount recorded for more than 5 seconds, assuming no signal
		if(countsPer1Sec == 0)
		{
			if(no_RPM_Five_Times == 10)
			{
```

```
                        eventsBuff2bits.eNoRPMSignal = 1;// set to indicate no RPM signal
                        eventsBuff1bits.eStartup = 1;         // set startup bit to
                                                              // initiate start up again
                        eventsBuff2bits.eResetRPMCountArray = 1; // set to Clear
                                                                 //RPM Count Array
                        seconds_Passed = 0;
                        LCDErase();
                        LCDUpdate();
                        no_RPM_Five_Times++;   //push it over 10 so this
                                               //doesn't keep clearing screen
                }
                else
                {
                        no_RPM_Five_Times++;
                        if(no_RPM_Five_Times == 12)  // never count beyond 12
                        {
                                no_RPM_Five_Times = 11;
                        }
                }
        }
        else
        {
                no_RPM_Five_Times = 0;  // clear as we have some RPM signal
        }

        previous_calculated_RPM = calculated_RPM;           // store Previous Calculated RPM
        calculated_RPM = calculateRPM(countsPer1Sec);
        pulseCount = 0;                                     // reset pulse count

        if(calculated_RPM > 0)      // only update LCD if the calculated RPM is greater than 0
        {
                LCDRPMInputLine1Display(calculated_RPM);
        }
        LED4 = Off;
        eventsBuff1bits.eCheckAtoD = 1;

        // check A to D to determine if load increaseing or decreasing
        if((eventsBuff1bits.eCheckAtoD == 1) && (eventsBuff1bits.eStartup == 0))
        {
                eventsBuff1bits.eCheckAtoD = 0;    // clear event flag
                load_Modifier = determineLoadCondition();
        }

        // only determine loading condition after startup
        if(eventsBuff1bits.eStartup == 0)
        {
                if(load_Modifier == LOAD_BALANCED)
                {
                        load_Modifier_Add = 0;
                        load_Modifier_Subtract = 0;
                }
                else       //change the RPM feedback signal based on it loading condition
                {
```

```
// is the load increasing?
if((load_Modifier >= LOAD_INCREASING_STEP1) && (load_Modifier <=
LOAD_INCREASING_STEP4))
{
        // load is increasing, Slow down the RPM feedback signal
        // appropriately
        load_Modifier_Subtract = 1;// set to indicate must subtract
        // load value to RPM value to make a slower RPM rate
        load_Modifier_Add = 0;              // clear flag as to indicate
                                            // load is not decreasing
        if(load_Modifier == LOAD_INCREASING_STEP1)
        {
                load_Modifier_value = calculated_RPM >> 5;
                // shift calc_RPM by 5, (divide by 32) to create a
                // 3.125% adder
        }
        else if(load_Modifier == LOAD_INCREASING_STEP2)
        {
                load_Modifier_value = calculated_RPM >> 4;
                // shift calc_RPM by 4, (divide by 16) to create a
                // 6.25% adder
        }
        else if(load_Modifier == LOAD_INCREASING_STEP3)
        {
                load_Modifier_value = calculated_RPM >> 3;
                // shift calc_RPM by 3, (divide by 8) to create a
                // 12.5% adder
        }
        else if(load_Modifier == LOAD_INCREASING_STEP4)
        {
                load_Modifier_value = calculated_RPM >> 2;
                // shift calc_RPM by 2, (divide by 4) to create a
                // 25% adder
        }
        else
        {
                // should never end up here, invalid
                // load_Modifier value
                load_Modifier_Subtract = 0;
                // clear flag as to indicate load is not decreasing
                load_Modifier_Add = 0;
        }
}
// is load decreasing?
else if((load_Modifier >= LOAD_DECREASING_STEP1) &&
(load_Modifier <= LOAD_DECREASING_STEP4))
{
        // load is decreasing, Speed up the RPM feedback signal
        // appropriately

        load_Modifier_Subtract = 0;         // clear flag as to indicate
        // load is not decreasing
        load_Modifier_Add = 1;    // set to indicate must add load
```

```
                                        // value to RPM value to make a faster RPM rate
                                if(load_Modifier == LOAD_DECREASING_STEP1)
                                {
                                        load_Modifier_value = calculated_RPM >> 5;
                                        // shift calc_RPM by 5, (divide by 32) to create a
                                        // 3.125% adder
                                }
                                else if(load_Modifier == LOAD_DECREASING_STEP2)
                                {
                                        load_Modifier_value = calculated_RPM >> 4;
                                        // shift calc_RPM by 4, (divide by 16) to create a
                                        // 6.25% adder
                                }
                                else if(load_Modifier == LOAD_DECREASING_STEP3)
                                {
                                        load_Modifier_value = calculated_RPM >> 3;
                                        // shift calc_RPM by 3, (divide by 8) to create a
                                        // 12.5% adder
                                }
                                else if(load_Modifier == LOAD_DECREASING_STEP4)
                                {
                                        load_Modifier_value = calculated_RPM >> 2;
                                        // shift calc_RPM by 2, (divide by 4) to create a
                                        // 25% adder
                                }
                                else
                                {
                                        // should never end up here, invalid
                                        //  load_Modifier value
                                        load_Modifier_Subtract = 0;
                                        // clear flag as to indicate load is not decreasing
                                        load_Modifier_Add = 0;
                                }
                        }
                        else
                        {
                                // should never end up here, invalid load_Modifier value
                                load_Modifier_Subtract = 0;
                                // clear flag as to indicate load is not decreasing
                                load_Modifier_Add = 0;
                        }
                }       // load modifier calcualtion
        }               // if(eventsBuff1bits.eStartup == 0)
    }
}       // main
****************
```

```
// This function determines the Timer3 value to create the Position sensor feedback signal
unsigned int determineTimer3Setup(unsigned int rpm)
{
	//INJECTOR_RPM may not work as Timer 3 uses the same clock as Timer 1 which
	// may not be the desired clock source

	// calculate what value needs to be loaded into Timer3
	// to cause the correct overflow value to create correct frequency
	unsigned long multResult;
	unsigned int timer3LoadValue;
	unsigned char timer3Mode;

	// determine prescaler based on rpm range
	// prescaler set to 1
	if((rpm < 75) && (rpm > 0))
	{
		T3CONbits.T3CKPS1 = 1;		// 1:8 prescaler
		T3CONbits.T3CKPS0 = 1;
		timer3Mode = TIMER1_32KHZ_MODE;
		T3CONbits.TMR3CS = 3;		// use external 32 kHz clock
	}
	else if ((rpm >= 75) && (rpm <= 299))
	{
		T3CONbits.T3CKPS1 = 1;		// 1:8 prescaler
		T3CONbits.T3CKPS0 = 1;
		timer3Mode = TIMER1_INTERNAL_MODE;
		T3CONbits.TMR3CS = 0;		// use internal clock
	}
	// prescaler set to 2
	else if((rpm >= 300) && (rpm <= 599))
	{
		T3CONbits.T3CKPS1 = 0;		// 1:2 prescaler
		T3CONbits.T3CKPS0 = 1;
		timer3Mode = TIMER1_INTERNAL_MODE;
		T3CONbits.TMR3CS = 0;		// use internal clock
	}
	// prescaler set to 1
	else if ((rpm >= 600) && (rpm <= 3600))
	{
		T3CONbits.T3CKPS1 = 0;		// 1:1 prescaler
		T3CONbits.T3CKPS0 = 0;
		timer3Mode = TIMER1_INTERNAL_MODE;
		T3CONbits.TMR3CS = 0;		// use internal clock
	}
	if(timer3Mode == TIMER1_32KHZ_MODE)
	{
		// based on prescaler, determine number of counts required for Timer3
		// if prescaler set to 8
		if((T3CONbits.T3CKPS1 == 1) && (T3CONbits.T3CKPS0 == 1))
		{
			multResult = 15360 / rpm;
		}
	}
```

```
        else if(timer3Mode == TIMER1_INTERNAL_MODE)
        {
                // system clock is 10MHz but using PLL so Freq is 40 MHz
                // based on prescaler, determine number of counts required for Timer1
                // if prescaler set to 8
                 if((T3CONbits.T3CKPS1 == 1) && (T3CONbits.T3CKPS0 == 1))
                {
                        multResult = 4687500 / rpm;
                }
                // if prescaler set to 2
                else if((T3CONbits.T3CKPS1 == 0) && (T3CONbits.T3CKPS0 == 1))
                {
                        multResult = 18750000 / rpm;        // if INTOSC is 10MHz
                }
                // if prescaler set to 1
                else if((T3CONbits.T3CKPS1 == 0) && (T3CONbits.T3CKPS0 == 0))
                {
                        multResult = 37500000 / rpm;        // if INTOSC is 10MHz
                }
        }
        timer3LoadValue = 65535 - (unsigned int) multResult;
        reloadTimer3Value = timer3LoadValue;
        return timer3LoadValue;
}

************
// This is how the simulator calculates what RPM speed is being generated by the controller
unsigned int calculateRPM(unsigned int countsPer1Sec)
{
        unsigned char static calculate_RPM_Pointer = 0;
        unsigned char static cal_RPM_Pointers_Filled = 0;
        unsigned int static RPM_count_array[10] = {0,0,0,0,0,0,0,0,0,0};
        unsigned int cal_RPM_Result;
        unsigned int cal_RPM_Result_Previous;
        unsigned char previous_RPM_count_array_element;
        unsigned int remainder = 0;
        unsigned char i;

        // reset calculated RPM
        if(eventsBuff2bits.eResetRPMCountArray == 1)
        {
                eventsBuff2bits.eResetRPMCountArray = 0;
                calculate_RPM_Pointer = 0;
                cal_RPM_Pointers_Filled = 0;
                for(i = 0;i < 10; i++)
                {
                        RPM_count_array[i] = 0;
                }
        }

        cal_RPM_Result_Previous = calculated_RPM;

        previous_RPM_count_array_element = RPM_count_array[calculate_RPM_Pointer];
```

```
RPM_count_array[calculate_RPM_Pointer] = countsPer1Sec;
calculate_RPM_Pointer++;
if(countsPer1Sec > 0)
{
        cal_RPM_Pointers_Filled++;          // only add count if anything other than 0
}
else
{
        if(previous_RPM_count_array_element > 0)            //erase a pointer unless no
                                                            // pointers captured
        {
                cal_RPM_Pointers_Filled--;
        }
}
cal_RPM_Result = 0;

if(calculate_RPM_Pointer == 10)
{
        calculate_RPM_Pointer = 0 ;
}
if(cal_RPM_Pointers_Filled == 11)
{
        cal_RPM_Pointers_Filled = 10;
}
if(cal_RPM_Pointers_Filled > 0)
{
        for(i = 0; i < 10; i++)         //          // changed from only counting filled
                                                                    // number of pointers
        {
                cal_RPM_Result += RPM_count_array[i];
        }
        if(cal_RPM_Pointers_Filled == 10)
        {
                return    cal_RPM_Result = cal_RPM_Result * 6;
        }
        else if(cal_RPM_Pointers_Filled == 9)
        {
                remainder = (cal_RPM_Result << 1)/3;                // here is the 0.66
                return    cal_RPM_Result = (cal_RPM_Result * 6) + remainder;          //6.66
        }
        else if(cal_RPM_Pointers_Filled == 8)
        {
                remainder = cal_RPM_Result >> 1;            // here is the half
                return    cal_RPM_Result = (cal_RPM_Result * 7) + remainder;          //7.5
        }
        else if(cal_RPM_Pointers_Filled == 7)
        {
                remainder = cal_RPM_Result >> 1;            // here is the half
                return    cal_RPM_Result = (cal_RPM_Result << 3) + remainder;         //8.57...
        }
        else if(cal_RPM_Pointers_Filled == 6)
        {
                return    cal_RPM_Result = cal_RPM_Result * 10;
```

```
				}
				else if(cal_RPM_Pointers_Filled == 5)
				{
					return	cal_RPM_Result = cal_RPM_Result * 12;
				}
				else if(cal_RPM_Pointers_Filled == 4)
				{
					return	cal_RPM_Result = cal_RPM_Result * 15;
				}
				else if(cal_RPM_Pointers_Filled == 3)
				{
					return	cal_RPM_Result = cal_RPM_Result * 20;
				}
				else if(cal_RPM_Pointers_Filled == 2)
				{
					return	cal_RPM_Result = cal_RPM_Result * 30;
				}
				else if(cal_RPM_Pointers_Filled == 1)
				{
					return	cal_RPM_Result = cal_RPM_Result * 60;
				}
			}
			else		// if no calculated pointers are filled
			{
				return cal_RPM_Result = 0;
			}
		} //calculateRPM()
		**************
		// This is how the device checks the Potentiometer analog voltage
		void SetupAtoD (void)
		{
			ADCON0 = 0b00000000;	// VCFG1 = 0 : VRef- source = Avss
										// VCFG0 = 0 : VRef+ source = Avdd
										// analog channel select AN0
										// A to D idle
										// A to D Module disabled
			ADCON1 = 0b10001110;	// ADFM = 1 : Right justified
										// ADCALnormal A/D conversion
										// ACQT2:ACQT0 = 001 : 2 TAD
										// ADCS2:ADCS0 = 110 : FOSC/64
			WDTCONbits.ADSHR = 1;
			ANCON0 = 0b11111110;	// all pins digital except AN0
			ANCON1 = 0b11111111;	// all pins digital
			WDTCONbits.ADSHR = 0;
		}

		unsigned int ReadPot(void)
		{
			unsigned int current_ad_value;
			unsigned int temp_ad_value;
			unsigned char i;

			temp_ad_value = 0;
```

```
    /* start the ADC conversion */
        for(i = 0; i < 8; i++)
        {
                ADCON0bits.GO = 1;                          // start A to D conversion
                _delay_100us();
                while (ADCON0bits.GO)
                        ;
                current_ad_value = ADRES;
                temp_ad_value += current_ad_value;
        }
        current_ad_value = temp_ad_value >> 3;              // shift by 3 to divide by 8
        return current_ad_value;
}


**************
// This function determines if the analog reading indicates the device should be in a specific load
// condition
unsigned char determineLoadCondition(void)
{
        unsigned int loadStepValue;
        unsigned char loadStepSetting;

        loadStepValue = ReadPot();
        if((loadStepValue >= LOAD_BALANCED_MIN) && (loadStepValue <= LOAD_BALANCED_MAX))
        {
                loadStepSetting = LOAD_BALANCED;
        }
        else if(loadStepValue < LOAD_BALANCED_MIN)
        {
                if((loadStepValue >= LOAD_INCREASING_STEP4_MIN) && (loadStepValue <=
                LOAD_INCREASING_STEP4_MAX))
                {
                        loadStepSetting = LOAD_INCREASING_STEP4;
                }
                else if((loadStepValue >= LOAD_INCREASING_STEP3_MIN) && (loadStepValue <=
                LOAD_INCREASING_STEP3_MAX))
                {
                        loadStepSetting = LOAD_INCREASING_STEP3;
                }
                else if((loadStepValue >= LOAD_INCREASING_STEP2_MIN) && (loadStepValue <=
                LOAD_INCREASING_STEP2_MAX))
                {
                        loadStepSetting = LOAD_INCREASING_STEP2;
                }
                else if((loadStepValue >= LOAD_INCREASING_STEP1_MIN) && (loadStepValue <=
                LOAD_INCREASING_STEP1_MAX))
                {
                        loadStepSetting = LOAD_INCREASING_STEP1;
                }
        }
        else if(loadStepValue > LOAD_BALANCED_MAX)
        {
                if((loadStepValue >= LOAD_DECREASING_STEP4_MIN) && (loadStepValue <=
                LOAD_DECREASING_STEP4_MAX))
                {
                        loadStepSetting = LOAD_DECREASING_STEP4;
                }
                else if((loadStepValue >= LOAD_DECREASING_STEP3_MIN) && (loadStepValue <=
                LOAD_DECREASING_STEP3_MAX))
                {
                        loadStepSetting = LOAD_DECREASING_STEP3;
                }
                else if((loadStepValue >= LOAD_DECREASING_STEP2_MIN) && (loadStepValue <=
                LOAD_DECREASING_STEP2_MAX))
                {
                        loadStepSetting = LOAD_DECREASING_STEP2;
                }
                else if((loadStepValue >= LOAD_DECREASING_STEP1_MIN) && (loadStepValue <=
                LOAD_DECREASING_STEP1_MAX))
                {
                        loadStepSetting = LOAD_DECREASING_STEP1;
                }
        }
        LCDLoadCondition(loadStepSetting);
        return loadStepSetting;
}
```

E.3 TCP/IP Stack

The following source code are some of modifications to the ENCX24J600 Demo example from Microchip's TCP/IP stack

From the CustomHTTPApp.c from the ENCX24J600 Demo

```
//Locates the 'lcd' parameter and uses it to update the text displayed on the board's LCD display.
static HTTP_IO_RESULT HTTPPostLCD(void)
{
	BYTE* cDest;
	extern unsigned int previousDesiredRPM;
	extern unsigned int desiredRPM;

	int  convint;
	int  atoi( const char * s );

	#define SM_POST_LCD_READ_NAME			(0u)
	#define SM_POST_LCD_READ_VALUE			(1u)

	switch(curHTTP.smPost)
	{
		// Find the name
		case SM_POST_LCD_READ_NAME:

			// Read a name
			if(HTTPReadPostName(curHTTP.data, HTTP_MAX_DATA_LEN) ==
			HTTP_READ_INCOMPLETE)
				return HTTP_IO_NEED_DATA;

			curHTTP.smPost = SM_POST_LCD_READ_VALUE;
			// No break...continue reading value

		// Found the value, so store the LCD and return
		case SM_POST_LCD_READ_VALUE:

			// If value is expected, read it to data buffer,
			// otherwise ignore it (by reading to NULL)
			if(!strcmppgm2ram((char*)curHTTP.data, (ROM char*)"lcd"))
				cDest = curHTTP.data;
			else
				cDest = NULL;

			// Read a value string
			if(HTTPReadPostValue(cDest, HTTP_MAX_DATA_LEN) ==
			HTTP_READ_INCOMPLETE)
				return HTTP_IO_NEED_DATA;

			// If this was an unexpected value, look for a new name
			if(!cDest)
			{
				curHTTP.smPost = SM_POST_LCD_READ_NAME;
```

```
                    break;
                }

                // Copy up to 32 characters to the LCD
                if(strlen((char*)cDest) < 32u)
                {
                    memset(LCDText, ' ', 32);
                    strcpy((char*)LCDText, (char*)cDest);
                }
                else
                {
                    memcpy(LCDText, (void *)cDest, 32);
                }
                convint = atoi( LCDText );
                previousDesiredRPM = desiredRPM;
                desiredRPM = convint;
                // set event flag to indicate new RPM change
                eventsBuff1.eRPMChangedViaTCPIP = 1;
                LCDUpdate();

                // This is the only expected value, so callback is done
                strcpypgm2ram((char*)curHTTP.data, (ROM void*)"/forms.htm");
                curHTTP.httpStatus = HTTP_REDIRECT;
                return HTTP_IO_DONE;
        }

        // Default assumes that we're returning for state machine convenience.
        // Function will be called again later.
        return HTTP_IO_WAITING;
}

**************
// The following functions convert variables into strings to send out to the web
void HTTPPrint_pot(void)
{
        BYTE AN0String[8];

        WORD ADval;
        extern unsigned int desiredRPM;

        uitoa(desiredRPM, AN0String);
        TCPPutString(sktHTTP, AN0String);

}

void HTTPPrint_pot1(void)
{
        BYTE AN1String[8];
        extern unsigned int calculated_RPM;

        uitoa(calculated_RPM, AN1String);
        TCPPutString(sktHTTP, AN1String);
```

```
}
```

****

The following items are from the MainDemo.c File

```
// Writes an IP address to the LCD display and the UART as available
void DisplayIPValue(IP_ADDR IPVal)
{
//          printf("%u.%u.%u.%u", IPVal.v[0], IPVal.v[1], IPVal.v[2], IPVal.v[3]);
             BYTE IPDigit[4];
            BYTE i;
            BYTE j;
            BYTE LCDPos=16;

            for(i = 0; i < sizeof(IP_ADDR); i++)
            {
                         uitoa((WORD)IPVal.v[i], IPDigit);

                        putsUART((char *) IPDigit);

                        for(j = 0; j < strlen((char*)IPDigit); j++)
                        {
                                    LCDText[LCDPos++] = IPDigit[j];
                        }
                        if(i == sizeof(IP_ADDR)-1)
                                    break;
                        LCDText[LCDPos++] = '.';



                        while(BusyUART());
                        WriteUART('.');
            }

            if(LCDPos < 32u)
            {
                        LCDText[LCDPos] = 0;
            }
            LCDUpdate();
}

//This is from main()
{
            // Initialize application specific hardware
            InitializeBoard();
            PMCONH = 0;
            TRISCbits.TRISC1 = 0;
            PMEL = 0;           // function as port IO
            PADCFG1 = 0xFF; // PMP uses TTL input buffers
            //Enable SPI
             SSP1STAT = 0x40;                       // was 0xC0 but ASM file has 0x40
             SSP1CON1 = 0x22;                       // was 0x21 but ASM file has 0x22
```

```
enterInit = 1;

LCDInit();
LCDDisplay((char *)"Ethernet Demo", 0, TRUE);

strcpypgm2ram((char*)LCDText, "TCPStack " TCPIP_STACK_VERSION "  "
"               ");
LCDUpdate();

// Initialize stack-related hardware components that may be
// required by the UART configuration routines
TickInit();

MPFSInit();          // Initializes the MPFS module.

// Initialize Stack and application related NV variables into AppConfig.
InitAppConfig();

 // Initiates board setup process if button is depressed
// on startup
  if(BUTTON0_IO == 0u)
 {
        #if defined(EEPROM_CS_TRIS) || defined(SPIFLASH_CS_TRIS)
        // Invalidate the EEPROM contents if BUTTON0 is held down for more than 4 seconds
        DWORD StartTime = TickGet();
        LED_PUT(0x00);

        while(BUTTON0_IO == 0u)
        {
                if(TickGet() - StartTime > 4*TICK_SECOND)
                {
                #if defined(EEPROM_CS_TRIS)
                 XEEBeginWrite(0x0000);
                 XEEWrite(0xFF);
                 XEEEndWrite();
                 #elif defined(SPIFLASH_CS_TRIS)
                 SPIFlashBeginWrite(0x0000);
                 SPIFlashWrite(0xFF);
                 #endif

                putrsUART("\r\n\r\nBUTTON0 held for more than 4 seconds.  Default settings
                restored.\r\n\r\n");

                LED_PUT(0x0F);
                while((LONG)(TickGet() - StartTime) <= (LONG)(9*TICK_SECOND/2));
                LED_PUT(0x00);
                while(BUTTON0_IO == 0u);
                Reset();
                break;
                }
        }
        #endif
```

```
            DoUARTConfig();
}

// Initialize core stack layers (MAC, ARP, TCP, UDP) and
// application modules (HTTP, SNMP, etc.)
StackInit();

// Now that all items are initialized, begin the co-operative
// multitasking loop.  This infinite loop will continuously
// execute all stack-related tasks, as well as your own
// application's functions.  Custom functions should be added
// at the end of this loop.
// Note that this is a "co-operative mult-tasking" mechanism
// where every task performs its tasks (whether all in one shot
// or part of it) and returns so that other tasks can do their
// job.
// If a task needs very long time to do its job, it must be broken
 // down into smaller pieces so that other tasks can have CPU time.
while(1)
{

            /*
             *  This is where the engine controller application code would go
             *
             */

            // This task performs normal stack task including checking
            // for incoming packet, type of packet and calling
            // appropriate stack entity to process it.
             StackTask();

            // This tasks invokes each of the core stack application tasks
             StackApplications();

            // Process application specific tasks here.
            // For this demo app, this will include the Generic TCP
            // client and servers, and the SNMP, Ping, and SNMP Trap
            // demos.  Following that, we will process any IO from
            // the inputs on the board itself.
            // Any custom modules or processing you need to do should
            // go here.
            #if defined(STACK_USE_GENERIC_TCP_CLIENT_EXAMPLE)
            GenericTCPClient();
            #endif

            #if defined(STACK_USE_GENERIC_TCP_SERVER_EXAMPLE)
            GenericTCPServer();
            #endif

            #if defined(STACK_USE_SMTP_CLIENT)
            SMTPDemo();
            #endif
```

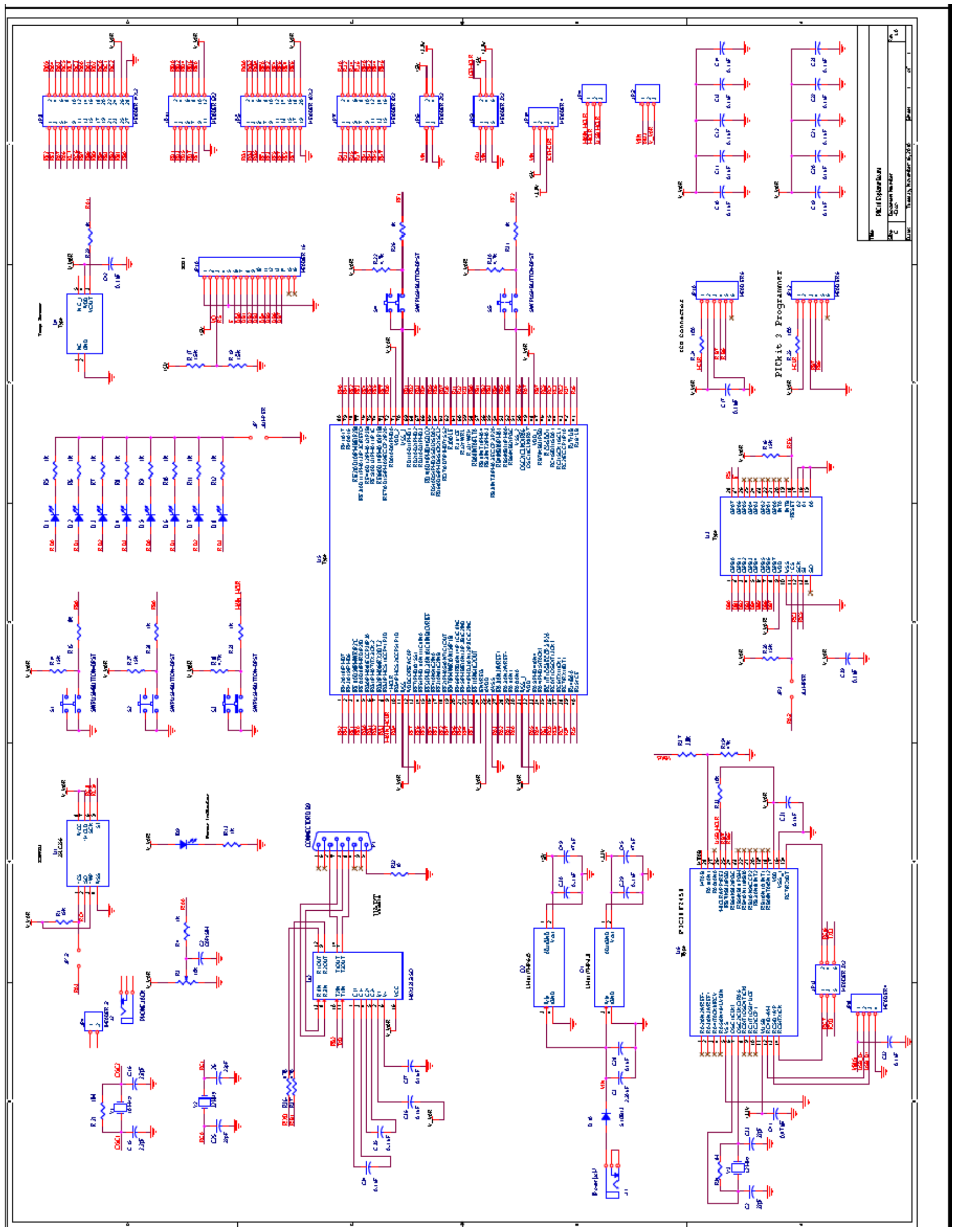
```
			#if defined(STACK_USE_SNMP_SERVER) && !defined(SNMP_TRAP_DISABLED)
			//User should use one of the following SNMP demo
			// This routine demonstrates V1 or V2 trap formats with one variable binding.
			SNMPTrapDemo();
			#if defined(SNMP_STACK_USE_V2_TRAP)
			//This routine provides V2 format notifications with multiple (3) variable bindings
			//User should modify this routine to send v2 trap format notifications with the
			//required varbinds.
			//SNMPV2TrapDemo();
			#endif
			if(gSendTrapFlag)
				SNMPSendTrap();
			#endif

			#if defined(STACK_USE_BERKELEY_API)
			BerkeleyTCPClientDemo();
			BerkeleyTCPServerDemo();
			BerkeleyUDPClientDemo();
			#endif

			ProcessIO();
			// If the local IP address has changed (ex: due to DHCP lease change)
			 // write the new IP address to the LCD display, UART, and Announce
			 // service
			if(dwLastIP != AppConfig.MyIPAddr.Val)
			{
				dwLastIP = AppConfig.MyIPAddr.Val;

				putrsUART((ROM char*)"\r\nNew IP Address: ");
				DisplayIPValue(AppConfig.MyIPAddr);
				putrsUART((ROM char*)"\r\n");

				#if defined(STACK_USE_ANNOUNCE)
					AnnounceIP();
				#endif

			}
		}
	}
```

Appendix F

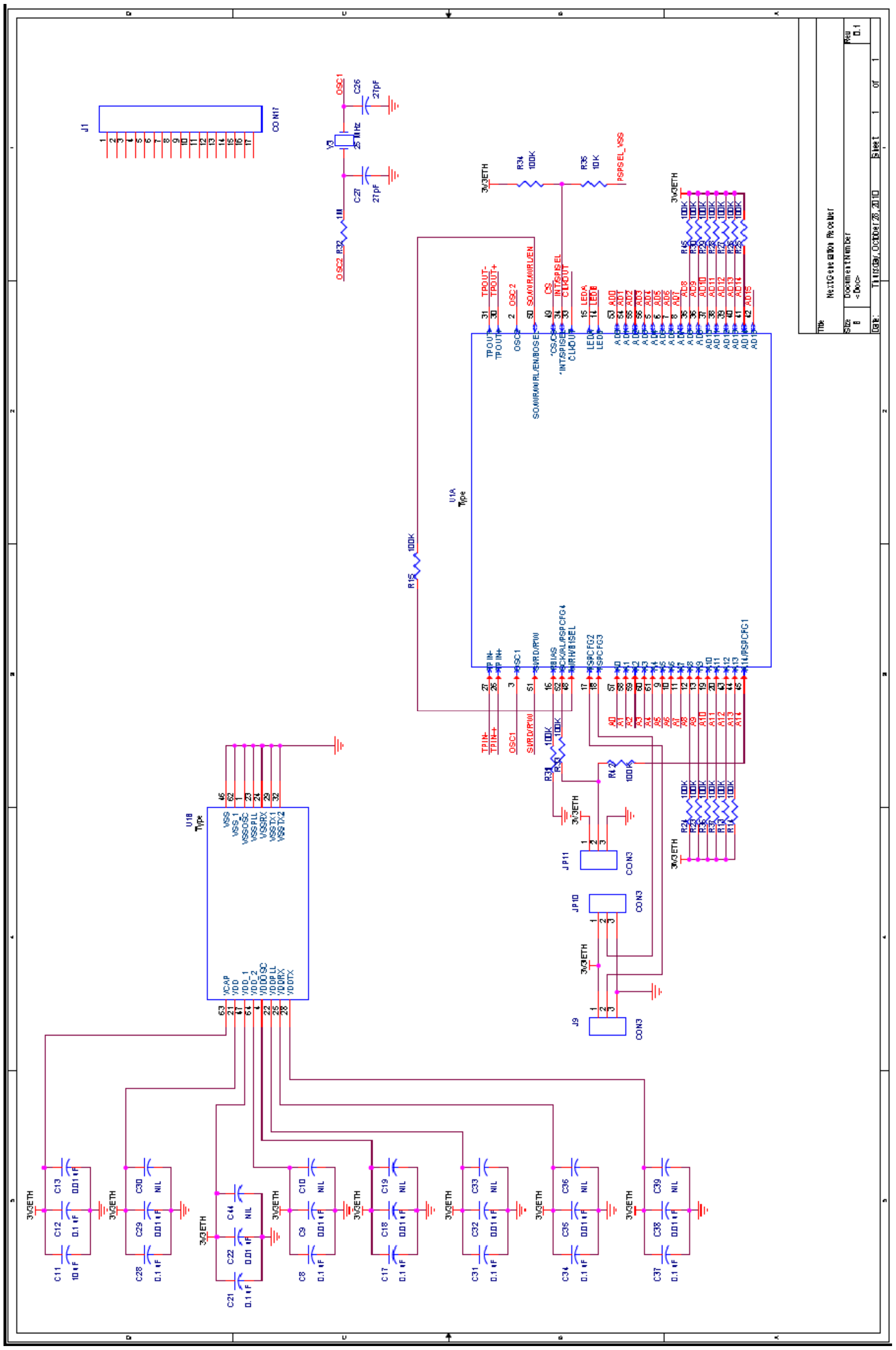
Schematics

The schematics are based on the design of the evaluation kit from Microchip. This does cover the entire circuit as components which were not required for this application were not included.

F.1 PIC18 Explorer Board Schematic



F.2 Fast 100Mbps Ethernet PICtail Plus Schematic



Fast 100 Mbps Ethernet PICtail Plus Schematic

Appendix G

G.1 Professional Background

I have been working and designing electronics with embedded technology since 2003. My career started with Canfield Connector as Project Engineer. Canfield Connector is a manufacturer of interconnection devices, electronic timers, connectors, sensors, modules and specialty electronic devices targeted at the fluid power industry [28]. As a Project Engineer for Canfield Connector, I was tasked with creating new products that required more intelligence than discrete electrical components provide. By incorporating microcontrollers into products, applications were given a degree of intelligence to perform a variety of different tasks based on different input and output conditions. Canfield provided a great opportunity to learning the basic engineering design concepts. Another role as project engineer was to follow a product through its complete design cycle, from the initial concept to sealing the box prior to shipment to the customer. Canfield has its own production capabilities and lab facilities, including surface mount board manufacturing and automated testing. After working approximately five years at Canfield Connector, I decided to switch employers and joined Turning Technologies.

Turning Technologies is the global leader in the Audience Response System market [29]. Turning provides both software and hardware solutions for presenters to engage their audience by asking questions and then gather the audience's responses. My position at Turning Technologies is a Hardware Design Engineer developing new hardware solutions, including new keypads and receivers. My main responsibilities include schematic and circuit design, embedded software development, testing products and managing contract manufacturing relationships. Previous experience with microcontrollers has proved invaluable at Turning Technologies. I continue learn more every day at Turning Technologies as I help to create the next audience response products.

References

1 A History of Electronic Engine Control.
http://www.allbusiness.com/legal/environmental-law-air-quality-regulation/15015826-1.html

2 Microchip Technology, Inc.
www.microchip.com

3 PIC18 Explorer Board.
http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en535770

4 MPLAB IDE
http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en019469&part=SW007002

5 PICkit 3
http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en538340

6 PIC18F87J11 Datasheet
http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en027149

7 C18 Compiler
http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en010014

8 25LC256 Datasheet
http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en023856

9 Microchip Ethernet Solutions
http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2504

10 Fast 100 Mbps Ethernet PICtail Plus Daughter Board
http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en543132

11 ENC624J600
http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en541877

12 TCP/IP Stack for PIC18, PIC24, dsPIC & PIC32
http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2680&dDocName=en537041

13 TCP/IP Stack Help file.
Included in the Microchip Applications Library as TCPIP Stack Help.chm 22

14 Ethernet Theory of Operation – App Note 1120 from Microchip Technology
http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en533903

15 Microchip TCP/IP Stack Application Note – App Note AN833 from Microchip Technology
http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en011993

16 Microsoft OneNote
http://office.microsoft.com/en-us/onenote/

17 Google Docs
http://googledocs.blogspot.com/

18 Microsoft Visio
http://office.microsoft.com/en-us/visio/

19 Tortoise SVN
http://tortoisesvn.net/

20 KDiff3
http://kdiff3.sourceforge.net/

21 Beyond Compare
http://www.scootersoftware.com/

22 OrCAD
http://www.cadence.com/products/orcad/pages/default.aspx

23 Mentor Graphics
http://www.mentor.com/

24 TinyCAD
http://tinycad.sourceforge.net/

25 Crimson Editor
http://www.emeraldeditor.com/

26 Tektronix TDS3054
http://www2.tek.com/cmswpt/psdetails.lotr?ct=PS&cs=psu&ci=14531&lc=EN

27 DIR-615, D-Link Wireless N 300 Router
http://www.dlink.com/products/?pid=565

28 Canfield Connector
www.canfieldconnector.com

29 Turning Technologies, LLC.
www.turningtechnologies.com

30 PICDEM™ PIC18 Explorer Demonstration Board User's Guide
http://ww1.microchip.com/downloads/en/DeviceDoc/51721b.pdf

31 IEEE 802.3 Specification
http://standards.ieee.org/about/get/802/802.3.html