The not so Fast and not so Furious*: SAE Telemetry System Design Report

Starring:
Laxmikant Joshi (aka Vin Diesel*), lkj10
Sangeeta Das (aka Michelle Rodriguez*), sd571
Stephen Tarzia (aka Ja Rule*), spt2002
Waclaw Godycki (aka Paul Walker*), wag47

Project Overview

This project is motivated by the Society of Automotive Engineers (SAE) student design competition. The premise of the event is to design, fabricate, and test a formula racecar for national competition. For the 2004 vehicle it was desired to develop a real-time data acquisition and telemetry system for use with the car. The telemetry system will be used to monitor various parameters related to the engine performance. The data is sampled using sensors and other hardware on board the car while the car is in motion. It is then processed by an embedded microcontroller and wirelessly transmitted to a base-station in real-time where it can be further processed in software and viewed by engineers. These data can serve as testing data for the engineering team and as a performance indicator. This information can also be used to make adjustments to the car (such as to the fuel controller) to alter its performance.

About the vehicle:

The vehicle is a formula style racecar built with a space-frame chassis powered by a 600cc motorcycle engine. For a summary of how the engine works you can consult howstuffworks.com. See the glossary below for definitions of the terms in italics. The engine is controlled by another commercially obtainable embedded electronic fuel injection (EFI) controller. The EFI controller uses various sensor data (see below) to determine how much fuel to put into the engine's combustion chamber on every cycle of the engine. On every cycle of the engine, the fuel controller processes the sensor data and uses some user specified information (namely a *base fuel map/load table*) to generate pulses used to control a bank of fuel injectors. The pulse width determines the duration an injector stays open. The longer an injector stays open, the more fuel goes into the engine for combustion. In general, for every engine speed value (aka *load site*) there is an optimal *Volumetric Efficiency* (VE) that allows the engine to achieve maximum performance. The larger the VE is, the greater the load on the engine. Maximum engine torque occurs at the highest VE load site. It is desired to maximize torque through all load sites, but there are mechanical constraints that make this difficult to do. However, the ideal can be approached through proper tuning of the engine

*Denotes a reference to the movie: *The Fast and the Furious,* copyright 2001, Universal Studios

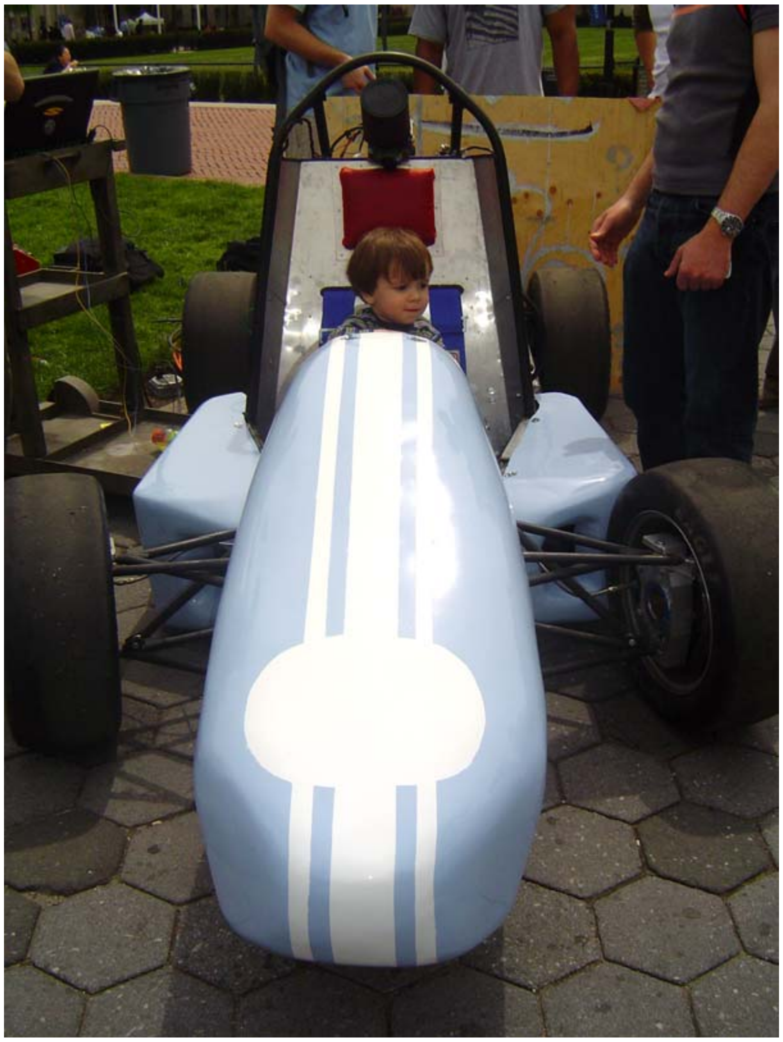


Figure 1: Photo of 2004 SAE vehicle

Data Channels that were implemented:

1. Engine speed (RPM)
2. Manifold Air Pressure (MAP)
3. Engine Temperature
4. Air/Fuel Mixture (AF)
5. Throttle Position (TPS)
6. Fuel Consumption

Motivation for the data: Items 1-5 are important parameters related to engine performance. These are the main parameters that the EFI controller uses to calculate pulse widths. Fuel consumption is important to know for making vehicle range calculations. It is very difficult to implement mechanically, but is simple to do electronically. Fuel Consumption = Σ (pulse width x fuel injector flow rate x 2). The additional factor of two is needed since the engine is a *batch fire* system.

Sensor implementation on the vehicle:

1. Engine RPM is measured on the vehicle using a magnetic reluctance sensor. A steel timing wheel is attached to the crankshaft of the engine. It has pattern of 12 teeth, with one tooth missing, at a fixed spacing. As the crankshaft rotates, the teeth pass under a magnetic reluctance sensor, which causes a sinusoidal voltage signal to develop. The RPM is a function to the instantaneous frequency of this signal.
2. The MAP sensor produces a linear 0-5 V signal which is proportional to the absolute air pressure inside the air intake manifold.
3. The Engine Temperature sensor is a thermistor.
4. The AF sensor produces a linear 0-5V signal proportional to the air fuel mixture extrapolated from the proportion of oxygen in the exhaust header. This is a measure of the *stoichiometric efficiency* of the engine.
5. Fuel Consumption will be calculated by measuring pulse widths of the outputs to the fuel injectors from the EFI controller.

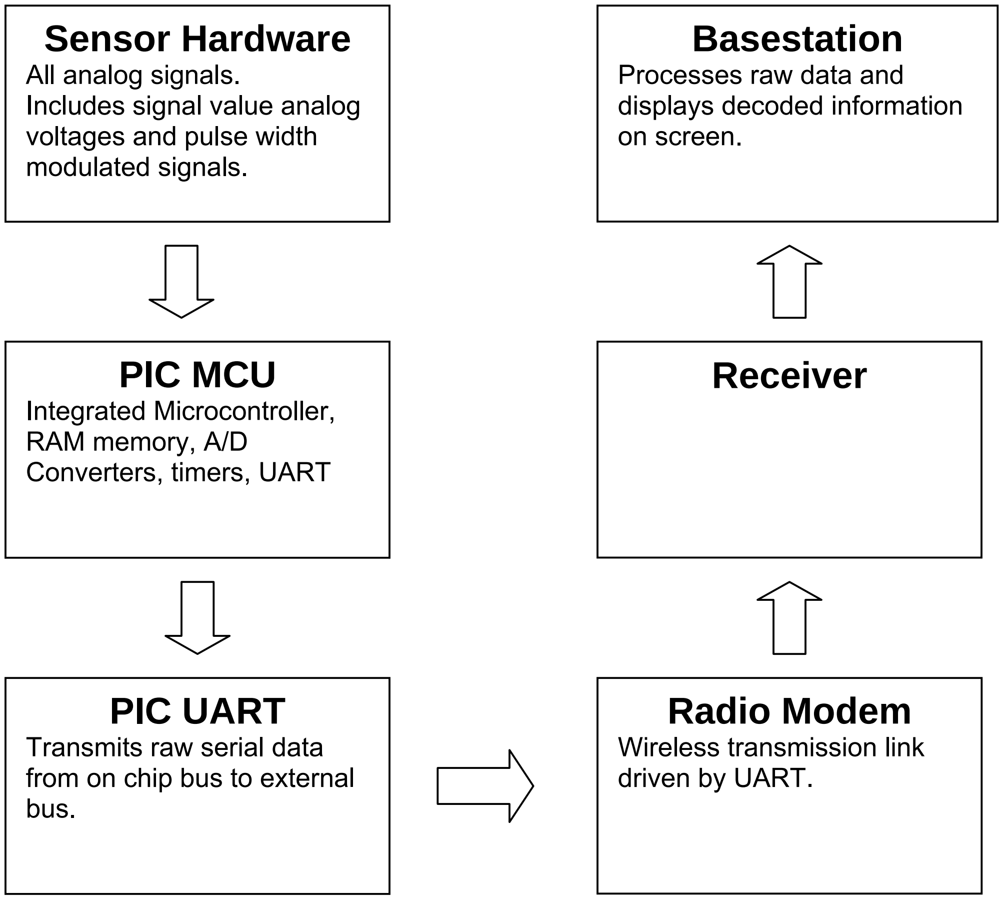
Short Glossary

- Batch fire – ignition system where two (or more) pistons are fired at the same time during one cycle. This is opposed to sequential ignition where only one piston in the engine is fired per cycle.
- Base fuel map/load table – a lookup table of user specified values that the EFI computer uses to calculate pulse widths. The user specifies how much fuel s/he would like the computer to put in under general loading conditions and the computer adjusts the value based on secondary lookup tables and sensor data. It is usually set by tuning the engine on a dynamometer.
- Soichiometric efficiency – the actual air/fuel ratio that was present in the engine during the combustion stroke.
- Volumetric efficiency (VE) – This is a measure of how much fuel the engine should burn effectively. It is the ratio of the amount of volume of air in a cylinder divided by the maximum theoretical capacity of the cylinder.
- Load site – the theoretical VE for a given engine speed range (usually at intervals of 50 RPM)
- Batch fire – ignition system where two (or more) fuel injectors are fired at the same time during one cycle. This is opposed to sequential ignition where only one piston in the engine is fired per cycle. A batch fire system requires fewer injector control lines that a sequential system.
- Base fuel map/load table – a lookup table of user specified values that the EFI computer uses to calculate pulse widths. The user specifies how much fuel s/he would like the computer to put in under general loading conditions and the computer adjusts the value based on secondary lookup tables and sensor data. It is usually set by tuning the engine on a dynamometer and through road testing.
- Soichiometric efficiency – the actual air/fuel ratio that was present in the engine during the combustion stroke.

Implementation Overview

Analog signals are generated by sensors located on various parts of the engine and chassis or by outputs of the EFI ECU. These signals required some analog circuitry for signal conditioning and buffering before they were fed to the ADC or timer channels of the PIC Microcontroller (MCU). Discrete valued signals (such as thermistors) will go to ADC channels, and the time based signals will go to timer channels. The data is converted to 8-bit PCM words by the MCU. The basis of the processing is a simple polling loop. On every polling loop cycle, the MCU will initiate A/D conversion on each ADC channel, store the results in memory, store pulse width timer data to memory, then construct a frame of data by retrieving the most recent data from memory and sequentially transmitting it to the radio modem via the UART. The MCU has a tri-state bus, with all data path and peripheral elements on the bus. All peripherals are controlled by memory mapped control registers. The radio modem link sends the data to a receiver at the base station, most likely a laptop computer. The raw data is be processed by the laptop on site and displayed in user readable text format.

Simplified Block Dataflow Diagram


Sensor Hardware
All analog signals.
Includes signal value analog voltages and pulse width modulated signals.
PIC MCU
Integrated Microcontroller, RAM memory, A/D Converters, timers, UART
PIC UART
Transmits raw serial data from on chip bus to external bus.
Radio Modem
Wireless transmission link driven by UART.
Receiver
Basestation
Processes raw data and displays decoded information on screen.

Sensors and analog interface circuitry

Since all sensor signals are piggybacked from the fuel controller which is built into the engine, all voltage signals had to be buffered with high input impedance op-amp circuitry to ensure proper signal strength and integrity. Some signals (e.g. temperature) are discrete valued voltage signals. Others, such as RPM and injector controls, are time based signals that require pulse width measurement. All signals are referenced to a common ground point. The injector pulse control signals were slightly trickier to interface with since there are two of them. However, only one of them is on at a given instant of time. All that is needed for fuel consumption is the running sum of the pulse widths. The two signals feeding the digital timer will simply be added using a digital NAND gate so only one PIC timer channel is needed. This signal is also fed into an AND gate with the chip clock signal because of the nature of the measurement scheme. Below is a detailed description about the implementation of each analog circuit channel. (See the final schematics on pages 13-14).

- Discrete Voltage Buffers

A Burr-Brown OPA340 opamp in unity gain configuration was used as a voltage buffer (figure 2). The MOS input stage on this opamp provided a high input impedance, and the chip operated on a single ended 5V supply. While it was not implemented, a low pass filtering capacitor could have been put at the input if noise from the car became a problem. Five of these circuits were built, but only four were actually used for the system allowing for expandability.

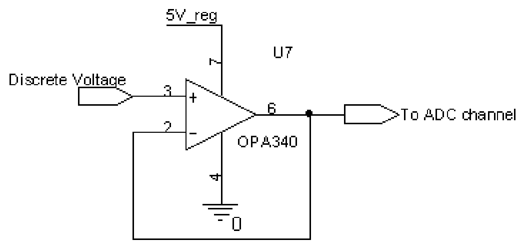


Figure 2: Discrete Voltage Buffers

- Fuel Injector Pulse Buffer

The fuel injector control signal is an output of the fuel controller ECU. The ECU has a transistor switch attached to the battery which connects the battery to the fuel injector solenoid whenever fuel is delivered to the engine. We modeled the solenoid as an inductor connected to the battery through a switch. By the nature of the solenoid switch, a 12V high signal represented a closed injector and the low voltage signal represented an open injector (closed switch).

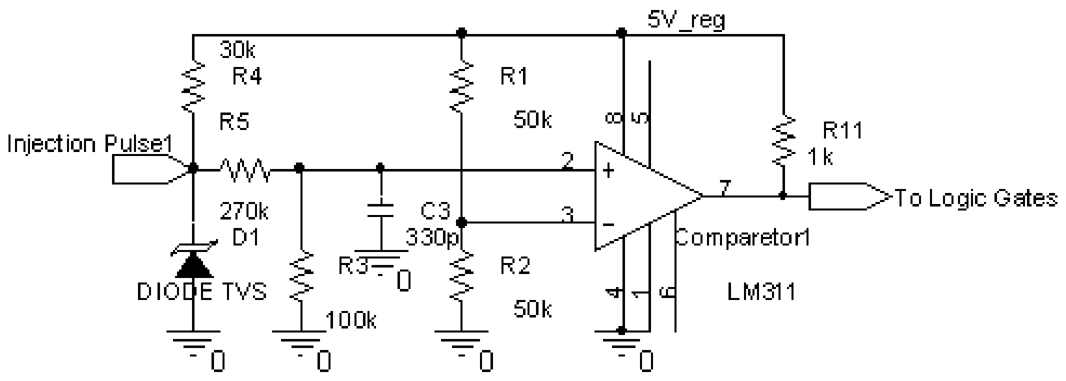


Figure 3: Injector Pulse Buffer Circuit

We measured the EFI signal going to the injector with a scope and found that a high signal was near 12V and a voltage low signal was 2 - 4V. It did not pull down all the way to ground. Also, the maximum allowable input voltage to the LM311 was VCC+0.3V (5.3V). Resistors R 3,4,5 served as a 1:4 voltage divider to reduce the dynamic range of the EFI signal. A 12V high signal was divided down to 4V and a 2 -

4V signal representing a low signal, was be cut down to 0.5-1V. This output was fed to a voltage comparator. The comparator compared this signal to a voltage at VCC/2 (2.5V). The reference was provided by R1 and R2. The output then saturated to either VCC or GND depending on whether the EFI signal was high or low.

A major design issue was large flyback voltage from the injector solenoid. When the EFI signal switched from low to high, there was a 300V Ldi/dt flyback voltage spike generated by the solenoid that had to be shunted somehow otherwise even the voltage divider could not clamp the input properly. To solve this, a 20V Transient Voltage Suppressor (TVS) zener diode was used. It is designed to turn on quickly to shunt any noise spikes above 20V to ground. This clamped the voltage and the input voltage divider further cut this voltage down to the allowable comparator input range. C3 further served as a high frequency noise filter as well.

The injectors themselves drew nearly 1A of current through the ECU and in order to prevent overloading of the ECU, the overall circuit impendence in parallel with the injector solenoid impedance had to be greater than 8 Ohms. The solenoid winding impedance was 15 Ohms. When D1 was off, the buffer input impedance was much higher than the solenoid impedance. However when D1 turned on, the only buffer impedance was whatever was ahead of D1. This is why the 300k for the voltage divider was split among two resistors. R4 served not only as a voltage divider but also impedance padding.

There were two EFI pulses that were out of phase with each other. Since the overall running sum of all the pulse widths was going to be calculated, the two EFI pulses were combined with a CMOS NAND gate. The desired logic table was as follows:

EFI 1	EFI 2	Output
1	1	X (not allowed)
1	0	1
0	1	1
0	0	0

- Timer Circuit

In order to use the MCU's timer circuitry to measure the EFI signal's pulse width, the signal had to be chopped so that it could trigger the counter in the MCU. The pulse widths vary from approximately .5ms to 2.5ms. Initially, the clock output from the MCU was going to be used to accomplish that. However, during bench testing it turned out that the MCU clock output is unable to drive much and the voltage levels are far from ideal TTL. After trying to use a few different simple analog buffers without success, a simpler approach used. Using a NE955 precision timer chip a square wave generator was built and used to chop EFI signal. For a 1ms pulse, sufficient accuracy is obtained by using 100-kHz chopper. Hence, the component values used for the resistors and the capacitor in the timer, which operates on Schmitt trigger principle.

- RPM Buffer Circuit

By the nature of the RPM sensor, the RPM signal is a non-constant amplitude frequency modulated signal and can have a large dynamic range. A magnetic reluctance sensor causes the amplitude of the signal to vary significantly with engine speed. In a way it acts as a generator. This has to be considered so as not to overload the OPA340 input.

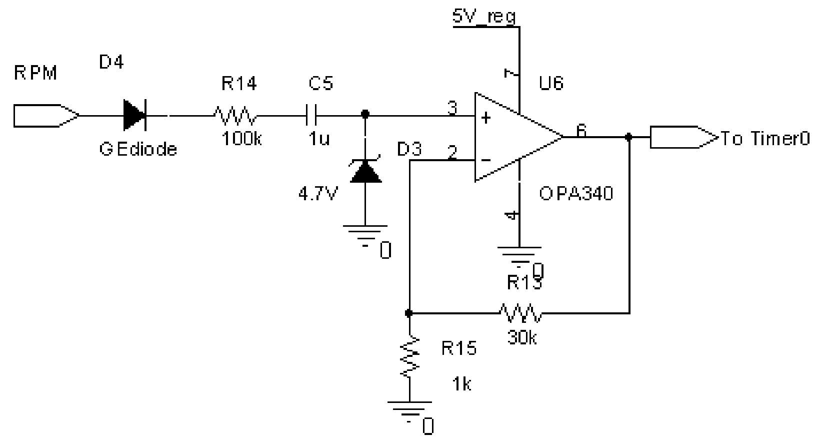


Figure 4: RPM Buffer Circuit

Figure 4 is basically a 5V hard limiter that is used to saturate the RPM signal to constant amplitude. While the original signal looks sinusoidal (although distorted and noisy), the amplitude values are irrelevant and the frequency data is preserved by the hard limiter and outputs to the MCU as a square wave. Its frequency varies with RPM. To sample the signal, first a low voltage drop germanium diode is used to reject all negative half cycles followed by a 100-kOhm resistor to provide sufficient signal separation between the ECU circuitry and the following clamping stage. Any signal with amplitude greater than 4.7V is clamped to by a 4.7 volt Zener diode, which is then amplified, or rather saturated with a high gain opamp, to a 0-5 volt square wave that can be processed by the MCU. An interesting problem arose while testing the circuit. The opamp would saturate even if for the negative cycles. Although the signal from the sensor is purely AC, but once the signal is rectified, it has a non-zero DC component that would saturate the output high. In order to prevent this, a 1uF AC coupling cap is used. Furthermore, the zener also prevents the input to the opamp from going more than –0.5V below ground and above 4.7V.

<u>Power Supply</u>

A well regulated power supply was needed to power the telemetry system. The standard automotive power supply (a nominally 12V lead-acid battery) is charged by an alternator and the actual battery voltage can vary wildly depending on a myriad of conditions. It was primarily intended to drive highly supply tolerant circuits and high power circuits like starter motors and ignition coils. All the opamps, the MCU, and the wireless modem required a single ended 5V supply. A simple low dropout regulator (the LM2940) was used. It is capable of regulating a 12V supply and can source up to 1A of current. The worst case power budget was divided as follows:

Component	Maximum current draw
MCU	300 mA
Wireless Radio Modem	150 mA
Analog Circuitry	50 mA
Total	**500mA**

500 mA at 5V (2.5 Watts) was well within the limits of the power dissipation of the LM2940 with an adequate heat sink and easily supplied by the car battery. The actual current draw from the battery was nearly 1A and much of it was dissipated by the regulator. A 10uF capacitor was used at the input of the regulator to serve provide noise filtering. A 47uF capacitor was used at the output to provide decoupling as well as to further regulate any voltage drop if a sudden burst of current was needed. 10nF ceramic capacitors were used on the supply pins of the analog opamps to further filter any high frequency noise and EMI on the supply pins.

PIC MCU and Programming

At the heart of the racecar data acquisition and telemetry system is a Microchip Technologies PIC16F77 flash programmable microprocessor. The PIC is clocked at 4MHz with an instruction execution rate of 1MHz. We utilized three of the PIC's on-chip peripherals: the ADC, the timers, and the UART.

The PIC's ADC has eight pin-inputs which are selected through an internal MUX. In the polling loop, this MUX cycles through all eight of the ADC channels, storing the acquired data in an array in the on-chip data memory.

We used the PIC's two timers as counters controlled by external clocks. The clock for timer0 is the rpm signal and the clock for timer1 is the EFI signal chopped by an oscillator. At the end of the PIC's polling phase, the data from these two timers is stored in memory (overwriting ADC channels 3 and 4, whose input pins are grounded). The value read from timer0 is proportional to the RPM rate of the motor and the value read from timer1 is proportional to the fuel consumption during the last program loop.

After the polling phase, there is a transmission phase. In this stage, the 8 bytes of acquired data are sent consecutively to the UART's transmission register. However, this data is not sent directly. First, each byte is converted into its two-ASCII-character hexadecimal representation. The main reason for this is to allow the receiver to recognize distorted data. Anything received data outside of the small range '0'-'9' or 'A'-'F' can be immediately recognized as bogus. This also made the transmitted data more readable during development. A 0x0D header byte was transmitted between each 16-byte data packet.

Once the transmission register is loaded, the UART peripheral handles the bit transmission and sets a flag bit when complete. Program execution can continue immediately after the transmission register is loaded. In steady state, the loop time is determined by the baud rate of the modem; in our case this was 9600 baud times 170 bits. This is an accurate and constant time interval because we are doing half-duplex transmission. Since the transmission time for one byte is much longer that the polling time, all polling occurs during the transmission of one character. The UART is always transmitting.

Wireless radio link and RS232 voltage levels

A Maxstream 9X PKG-R serial radio modem link is used to transmit the data sampled and processed by the MCU. It has a maximum of 9600kps of serial data throughput and can transmit data wirelessly up to 3km with a direct line of sight. It transmits in the 900 MHz unlicensed band and utilizes a frequency hopping spread spectrum transmission scheme. The modem, as it comes in the package, interfaces via standard RS-232 voltage levels. We first thought that we would need a level converter since the MCU serial output is CMOS levels. It turned out that the modem can be interfaced to CMOS circuits once it is stripped from the package. However, a very well regulated power supply has to be used for the sensitive modem circuitry. On the receiver side, the standard packaging was used to interface to the RS-232 serial input of a laptop computer.

Fabrication and PCB layout

This system was intended to be used for vehicle racing applications; this is a rather harsh environment. We made a study enclosure out of aluminum machined on a two axis milling machine to provide protection to the circuitry and developed a PCB (printed circuit board). PCB layout was done using CadSoft EAGLE version 4.03. First a schematic was entered. Then packages were chosen and laid out on the board space manually. Floorplanning and component placement was very important when trying to maximize trace planarity and for separating noise sensitive components. Each sub-circuit was laid out individually and then duplicated where applicable (there are two exact copies of the EFI circuit and five opamp buffers).

After part placement, the critical traces were laid out manually. These included the main power lines, which were routed with wider traces. Power supply traces were made twice as wide as the signal traces to alleviate IR drops on the traces. The remaining traces were laid out by the software's autorouter. The autorouter allows the user to assign costs to different board features such as trace turns, vias, particular layers, etc. We used these tweaks to have the autorouter produce a two sided trace layout with as few top-side traces as possible. Since we were making a single sided board, the top-layer traces would be wired with jumpers. There are about two dozen of these jumpers on our board. The PCB was a single layer board that we made in-house using a negative mask transfer process. The board was developed in sodium carbonate solution and etched with ferric chloride. Finally the holes for components were drilled out on a two axis milling machine.

User Interface

Setting up the environment was the most challenging aspect of developing the user interface. Various methods of reading data from an RS232 port were researched. Java offered a package dedicated to reading and writing from serial ports, called Javax, which had built in functionalities such as buffers and streams, and could easily be used to make a GUI interface. Unfortunately, the technology had little support and even less documentation. A more mainstream approach was C. Hardware resources were limited to notebooks running Windows operating systems, and therefore Cygwin, an application simulating a Linux environment in Windows, was used to read from the RS232 port. Although the reads were "successful" Cygwin required quite a bit of overhead, and was not able to support the high frequency of incoming data. Thus, a traditional Linux environment was used to access and read from the RS232 port.

The user interface program is used to process the data coming from the PIC MCU, as well as display it to the user. Methods of concurrently reading and processing the data were tried, but were not necessary because all data could be read and processed in ample time. However, attention to efficiency in execution was considered when developing the code.

The program received packets consisting of 8 fields of data, with each piece of data being received as two ASCII values. Incomplete packets were checked for and discarded and only complete packets were processed. All valid data read from the RS232 port was stored in two data files: raw.txt, which displays the raw data in ASCII format, as well as processed.txt, which shows the processed but unformatted data outputted from the program. Every tenth packet was displayed to the user so that the user could mentally process and visually see the change in data. A simple two line display was outputted to the user in the format:

```
TPS    TIMER0        TIMER1        TEMP MAP   A/F
88%    0.00          0.001         3.40V 4.00V 1.00V
```

The program was tested using the PIC to process random signals in the range of 0V to 5V, simulating possible expected data from the car. Voltages were also manually increased and decreased to test if the program correctly processed the data.

Testing

After looking at the raw signals while the car was running and designing the proper analog circuitry, we tested each of the components on the bench. Once, the unity gain, EFI pulse and RPM buffers were working, we connected them again to the running car and observed the output on the scope to verify that we get the required voltage levels and signal waveforms, while still powering from a bench supply. The voltage regulator circuit was tested independently to see if the voltage range as required by the modem and the PIC would be maintained, even if the current requirement goes as high as 500mA. After this, operation of all analog parts together as well as the modem and the PIC was tested successfully with the car's supply. After this step we were confident that the circuit would work and we proceeded to print the board and solder all the components together. After debugging we made sure that made sure again that the transmission could be established from the PCB. Serial data transmission was verified using randomly generated values on a breadboard as well as with the system connected to the vehicle transmitting easily verifiable static data.

The Epilouge

This was a true embedded system, complete with analog hardware, microcontroller firmware, front-end user software, and an aluminum box to boot to embed the whole thing into a peripheral that doesn't look anything like a computer—a formula racecar. It was important to test things on the bench piece by piece to make the debugging process easier. That was the only thing that maintained the sanity of all involved. By testing a single buffer, then two buffers, then a board, and then bench testing the entire system, we could be confident that the system would work before putting it into the car. However there were things that could not be simulated or predicted very accurately such as power supply noise, EMI noise, heat, and vibration. Oddly enough, on the final project demo, the board actually powered up, data was being transmitted and nothing caught on fire. But of course in true cynic's fashion the actual car that this system was embedded into did not start up for the test due to faulty spark plugs. (Sorry Prof. Edwards; We'll work on that one).

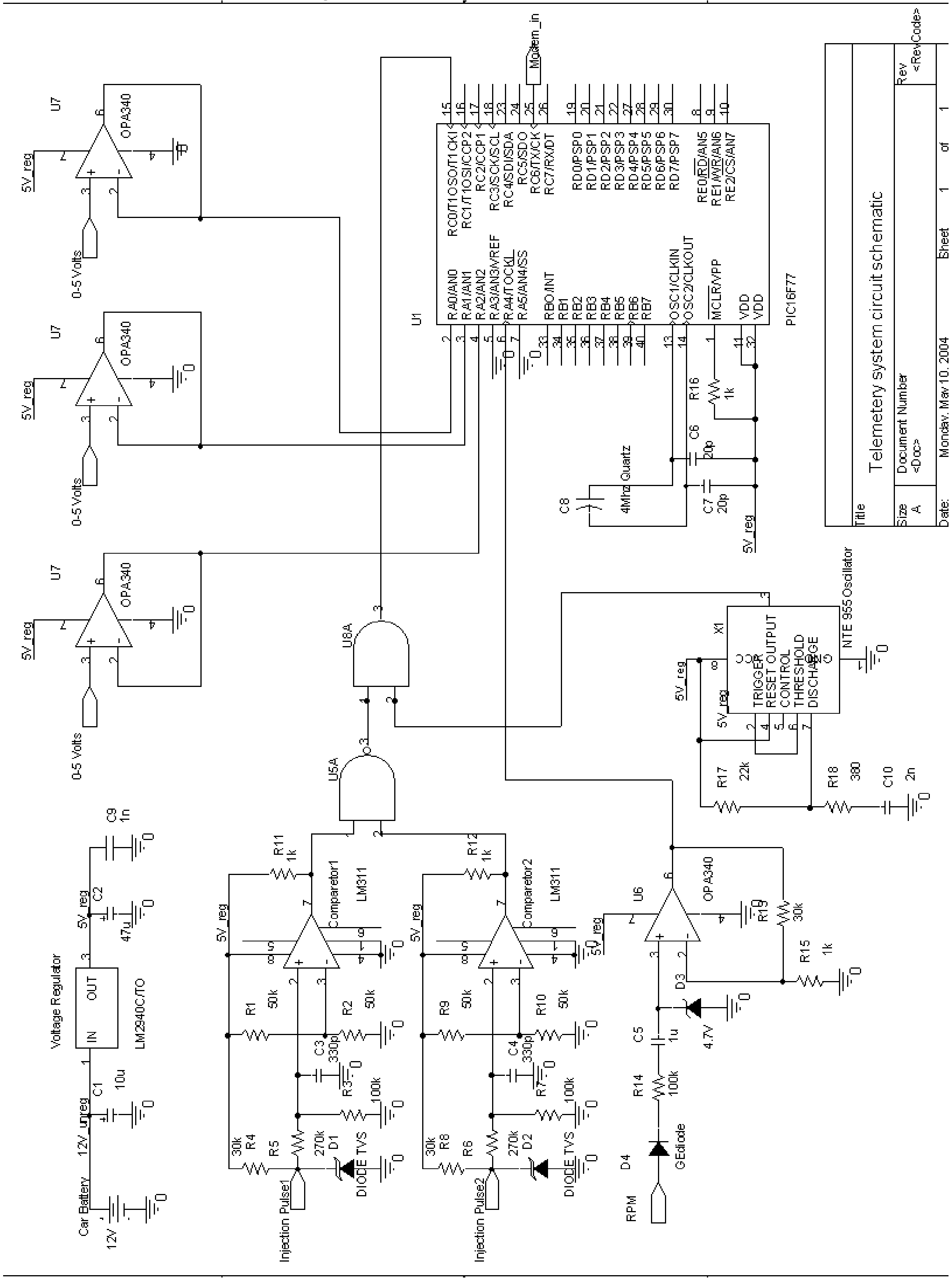
"I live my life a quarter mile at a time. For those 10 seconds or less, I'm free."
*–Vin Diesel, The Fast and the Furious**

Special Thanks:

- The Mechanical Engineering Department and the Columbia SAE for funding this project.
- Brian Lewis and his company, Performance Electronics LTD, for his gracious consultation and information about interfacing to the EFI controller.
- Nick Rivera and the Plasma Physics Laboratory for allowing us to use their facilities to print our board and for the occasional capacitor.
- John Kazana and the Electrical Engineering department for use of their equipment.
- The entire cast of *The Fast and the Furious* copyright 2001, Universal Studios for the inspiration (a great movie about cars!)

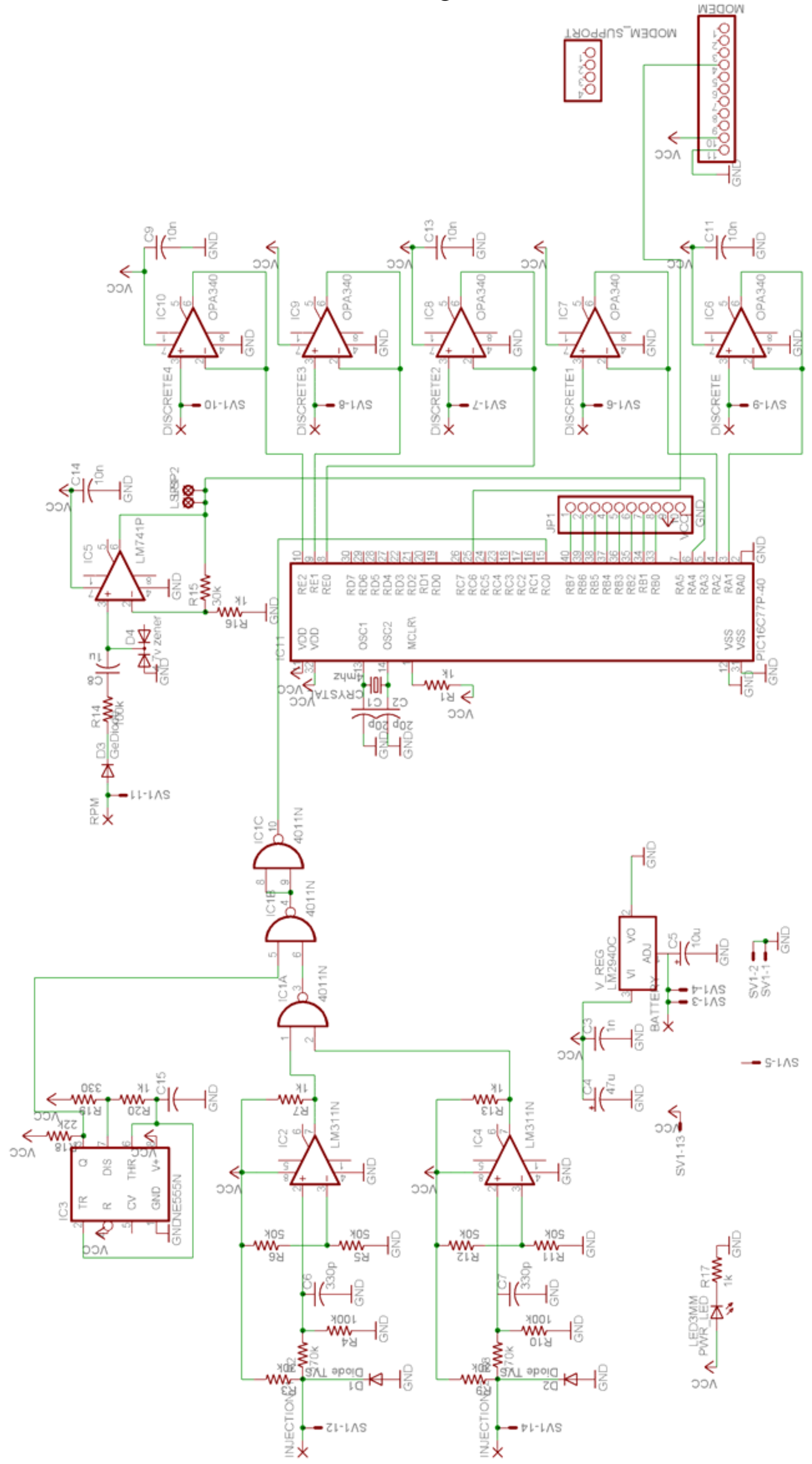
Figures Included Below:

Figure 5: Conceptual Schematic



13

Figure 6: Board Schematic



Note:

There are some inconsistencies between the part numbers on the board layout and on the schematic. The reason for this is that the CAD software did not have these parts in its library. Substitutions with identical packages were used.

Figure 7: PCB Layout

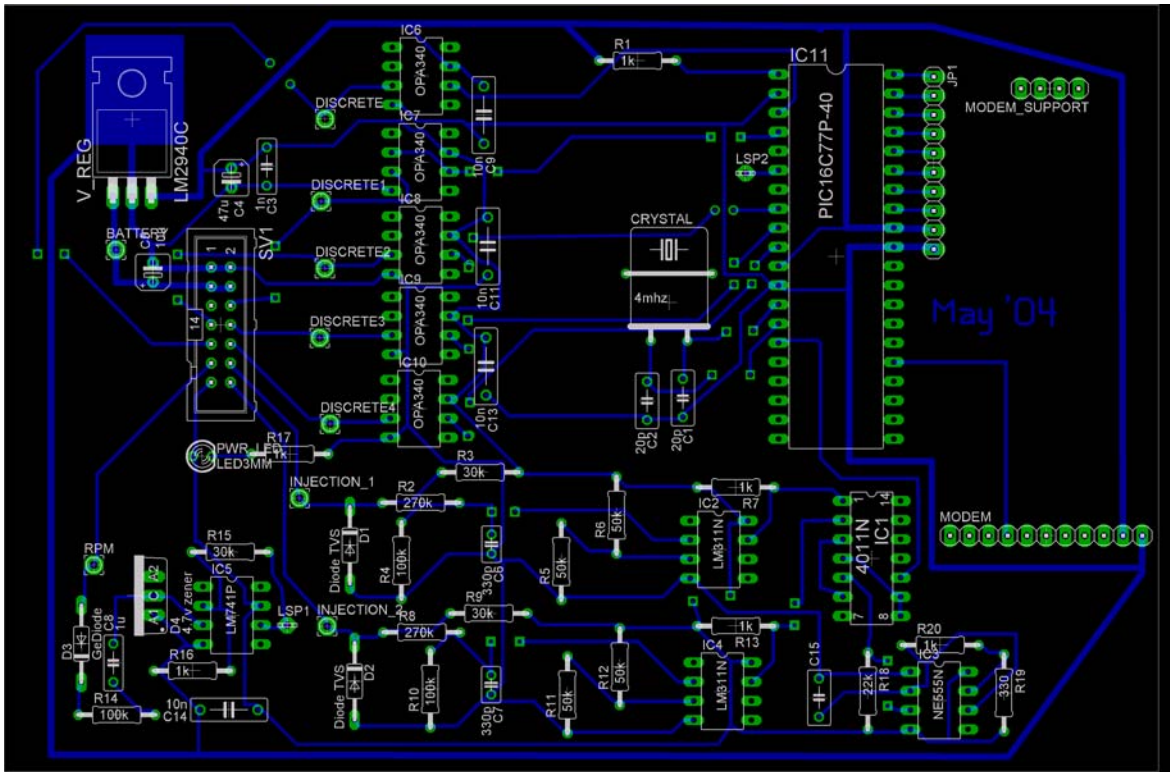


Figure 8: PCB Mask

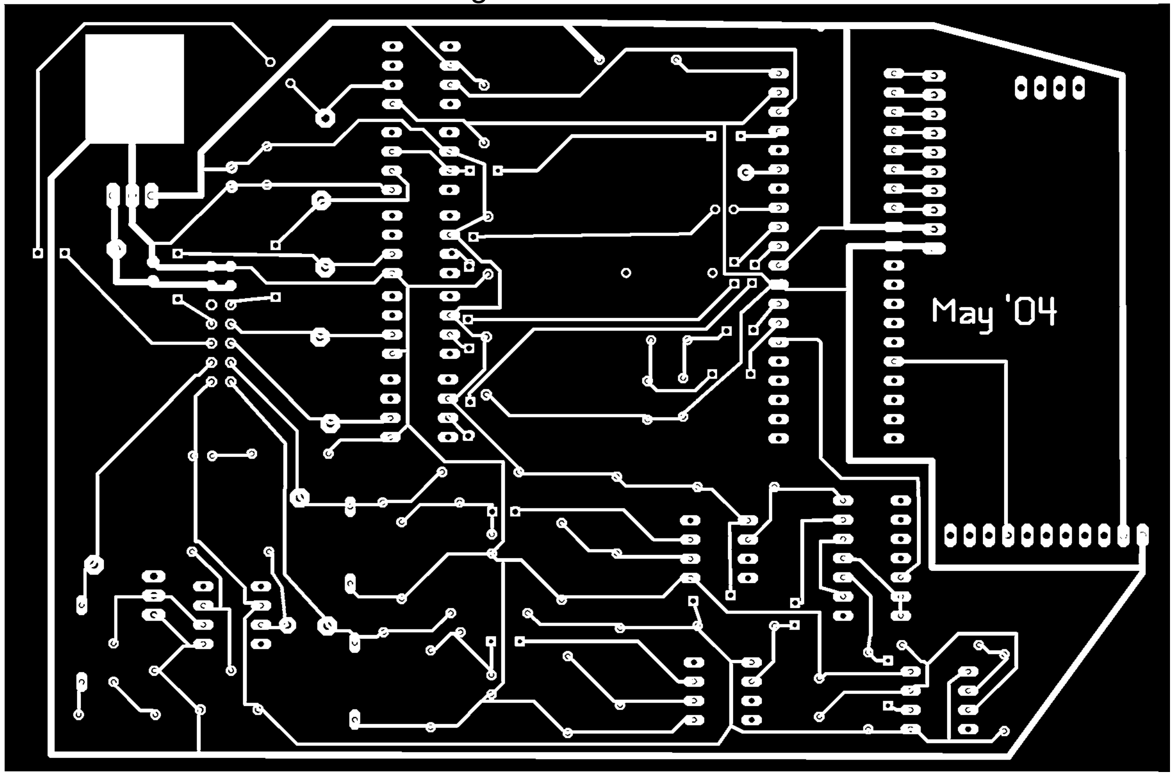


Figure 9: PIC16F77 Assembly Code Listing

```
;4840 project, may 2004

; GLOBALS
      list  p=16f77
      include     <p16f77.inc>

      __CONFIG  _CP_OFF & _WDT_ON & _BODEN_ON & _PWRTE_ON & _XT_OSC

c1    equ   0x20  ; define ASM alias
tmp   equ   0x21
tmp2  equ   0x22
tmp3  equ   0x23
adc0  equ   0x30
adc1  equ   0x31
adc2  equ   0x32
tmr0  equ   0x33
tmr1  equ   0x34
adc5  equ   0x35
adc6  equ   0x36
adc7  equ   0x37

      org   0x00  ; set program origin
reset GOTO  start

      org   0x04
start
;DISABLE INTERRUPTS
      CLRF INTCON

;INIT UART, ASSUME Fosc=4 MHZ
      BSF   STATUS,RP0  ;BANK1
      MOVLW D'25' ; 9600 BAUD WITH 4MHZ CLK
      MOVWF SPBRG ;BAUD RATE GENERATOR REGISTER
      MOVLW 0x24    ;'BRGH'=1 FOR HIGH SPEED
      MOVWF TXSTA ;XMIT STATUS REG
      BCF   STATUS,RP0  ;BANK0
      MOVLW B'10000000' ;'SPEN'
      MOVWF RCSTA ;RCSTATUS REG

;ADC INIT
      BSF   STATUS,RP0  ;BANK1
      CLRF  ADCON1      ;INIT ALL 8 ADC PINS
                  ;WITH VDD FOR VREF
      BCF   STATUS,RP0  ;BANK0
      MOVLW B'01000001'
      MOVWF ADCON0      ;CHANNEL0, ADC ON, FOSC/8

;PORTB INIT,
;this I/O port was used for debugginng output
      BSF   STATUS,RP0  ;BANK1
      CLRF  TRISB ;SET ALL 8BITS FOR OUTPUT
      BCF   STATUS,RP0  ;BANK0
      CLRF  PORTB ;CLEAR PORTB TO INIT
```

```
;TIMER0 INIT
      BSF   STATUS,RP0  ;BANK1
      BCF   STATUS,RP0  ;BANK0
      CLRF  TMR0  ; CLEAR TIMER

loop
;This the the main program loop.  This loop consists of a
;polling stage followed by a transmission stage.

;here we set the indirect address register to point to the
;beginning of the acquired data array
      MOVLW adc0
      MOVWF FSR   ;LOAD FSR W/ADC0 ADDR

adc_poll
;this is the start of the polling stage
;The following code is executed for each of the 8 ADC channels
;Each iteration, the data from adcY is stored in mem[0x3Y].
      ;DELAY FOR ACQUISITION
      CLRF  tmp
HERE1 INCFSZ      tmp,1
      GOTO  HERE1 ;;;

      ;START CONVERSION
      BSF   ADCON0,GO
wait1 BTFSC ADCON0,GO
      GOTO  wait1
      MOVF  ADRES,0
      MOVWF INDF  ;POINTS TO DATA ARRAY

      BTFSS ADCON0,3
      GOTO  adc_not_done
      BTFSS ADCON0,4
      GOTO  adc_not_done
      BTFSS ADCON0,5
      GOTO  adc_not_done
      MOVLW B'11000111'
      ANDWF ADCON0,1  ;CLEAR CHANNEL BITS
      GOTO  adc_done

adc_not_done
      MOVF  ADCON0,0    ;INCREMENT ADC
      ADDLW B'00001000' ; CHANNEL
      MOVWF ADCON0            ;

      INCF  FSR,1
      GOTO  adc_poll

adc_done
      INCF  c1,1  ;INCREMENT c1

      ;OUTPUT TO PORTB FOR DEBUGGING
      MOVF  c1,0
      MOVWF PORTB

timer_poll
;Timer polling is trivial, just copy from one
```

```
;register to another.
;Timer0 is reset but Timer1 is not
      MOVF  TMR0,0
      MOVWF tmr0
      CLRF  TMR0

      BCF   T1CON, TMR1ON     ;turn off timer
      MOVF  TMR1L,0
      MOVWF tmr1
      CLRF  TMR1L
      CLRF  TMR1H
      MOVLW B'00111110' ;1:8 PRESCALAR, EXTERNAL CLOCK, synchronous
      MOVWF T1CON       ;
      BSF   T1CON, TMR1ON     ;turn on timer


; XMIT PHASE
;Here the data transmission stage begins.  All the acquired
;data was stored in mem[0x30-0x37].  Here we simply iterate
;through this data, again using the indirect address register.
;
;Before loading the UART transmission register TXREG we wait
;for the "ready" bit to be set, TXIF.  When this bit is set,
;we know that the previous byte of data has been sent from
;TXREG to the UART.

      ;XMIT HEADER BYTE 0x0D
wait4 BTFSS PIR1,TXIF
      GOTO wait4  ;;;
      MOVLW 0x0D
      MOVWF TXREG       ;LOAD W into TXREG

      ;XMIT ADC DATA
      CLRF  tmp2
      MOVLW adc0
      MOVWF FSR   ;LOAD FSR W/ADC0 ADDR
array_xmit
      SWAPF INDF,1
      MOVF  INDF,0
      ANDLW B'00001111'
      ADDLW 0x30  ;ASCII VALUE OF '0'
      MOVWF tmp
      BTFSS tmp,3
      GOTO  wait3
      ADDLW 0x07  ;OFFSET BETWEEN '9' AND 'A'
      BTFSC tmp,1 ;UNDO IF '8' OR '9'
      GOTO  wait3 ; .
      BTFSC tmp,2 ; .
      GOTO  wait3 ; .
      ADDLW 0xF9  ; . SUBTRACT 0x07

wait3 BTFSS PIR1,TXIF   ;'TXIF'
      GOTO  wait3 ;;;
      MOVWF TXREG       ;LOAD W into TXREG

      INCF  tmp2,1
      BTFSC tmp2,0            ;IF FIRST ITERATION
```

```
GOTO  array_xmit

INCF  FSR,1
BTFSS FSR,3 ;SET AFTER 8 ITERATIONS
GOTO  array_xmit

GOTO  loop
end
```

Figure 10: User Interface Listing

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/select.h>
#include <fcntl.h>
#include <termios.h>
#include <string.h>
#include <errno.h>

#define DEBUG 0
#define BSIZE 1700       /*this is actually the last index in buffer*/
#define PSIZE 17         /*packet size (includeing \r in header)*/
#define DATINPACK 8      /*number of data lines in packet*/
#define VPI .0195        /*voltage per incriment*/
#define RPMCONV 282.353  /*conversion rate for v -> RPM*/
#define TPSOFF 0.51   /*voltage offset for throttle*/
#define TPSCONV 0.25     /*throttle conversion rate*/
#define MAPCONV 2.72     /*manifold absolute pressure conversion rate*/
#define MAPOFF 1.51      /*V offset for manifold absolute pressure*/
#define PRESCALER 8      /*Prescaling used to encode fuel consumption*/
#define T1FREQ 100000    /*timer one clock freq*/
#define FLOWRT 0.004505  /*flowrate for timer 1*/
#define OUTPUTCNTR 10          /*used to printf every 10th packet*/

/*
 * Code to open and read RS232 is modified code from RS232.c written by
 * Cristian Petrus Soviani
 */

int main () {

  /*variables for opening/reading RS232*/
  int fd, nbrx, nbtx, ptx, prx, nb;
  fd_set rfdsin, rfdsout;
  char buffer [BSIZE+1];
  int bufitr = 0;
  struct termios my_termios;

  int i;
  int j;
  int packetcntr = 0;
  int ifprint = 1; /*0 if it hsould print*/
  unsigned int first;
  unsigned int second;//used for getting data
  int fndx; //used to get first char
  int sndx; //used to get second char
  int cindex; //index to cross check if a packet is incomplete;
  FILE *fp;
  FILE *f2;
  unsigned int temp;
  double mydata;
  double myoutput;
  double totalFC = 0;
```

```
fp = fopen ("raw.txt", "w");
f2 = fopen ("processed.txt", "w");
fd = open ("/dev/ttyS0", O_RDWR | O_NOCTTY);


 //configure serial port
tcgetattr(fd, &my_termios);
tcflush(fd, TCIFLUSH);
my_termios.c_cflag = B9600 | CS8 | CREAD | CLOCAL | HUPCL;
cfsetospeed (&my_termios, B9600);
tcsetattr(fd, TCSANOW, &my_termios);

/*print header*/
printf ("\n\tTPS\tTIMER0\tTIMER1\tTEMP\tMAP\tA/F\n");

/*read first potential packet*/
for (bufitr = 0; bufitr<PSIZE; bufitr++) {
      read (fd, &buffer[bufitr], 1);
           fprintf (fp, "%c", buffer[bufitr]);
}

/*for all subsequent packets*/
while (1) {
  read (fd, &buffer[bufitr], 1);
  fprintf (fp, "%c", buffer[bufitr]);

  if (buffer[bufitr] == '\r') {
    /*set compare index*/
    if (bufitr < PSIZE)
           cindex = BSIZE+1 +bufitr - PSIZE;
    else
           cindex = bufitr - PSIZE;

    /*process if complete*/
    if (buffer[bufitr] == buffer[cindex]){
           fndx = cindex+5;
           /* process data - FIRST TWO CHANNELS IGNORED!*/
           for (j=2; j<DATINPACK; j++) {
             /*check if near end of buffer*/
             if (fndx<BSIZE)
               sndx=fndx +1;
             else if (fndx==BSIZE)
               sndx = 0;
             else if (fndx>BSIZE) {
               fndx = 0;
               sndx = 1;
             }

             /*convert first char to hex*/
             if (buffer[fndx] >= '0' && buffer[fndx] <= '9')
               first = buffer[fndx] - 48;
             else if (buffer[fndx] >= 'A' && buffer[fndx] <= 'F')
               first = buffer[fndx] - 55;

             /*convert sec char to hex*/
             if (buffer[sndx] >= '0' && buffer[sndx] <= '9')
```

```
          second = buffer[sndx] - 48;
        else if (buffer[sndx] >= 'A' && buffer[sndx] <= 'F')
          second = buffer[sndx] - 55;

        first = first<<4;
        first = first | second;
        mydata = first*VPI;

        switch (j) {
        case 2:
          myoutput = (mydata - TPSOFF) * TPSCONV * 100;
          if (ifprint == 0)
                printf ("\t%.0f%%", myoutput);
          break;
        case 3:
          myoutput = mydata * RPMCONV;
          if (ifprint == 0)
                printf ("\t%.0f", myoutput);
          break;
        case 4:
          myoutput = mydata * PRESCALER * FLOWRT / T1FREQ;
          totalFC += myoutput;
          if (ifprint == 0)
                printf ("\t%.3f", totalFC);
          break;
        case 5:
          myoutput = mydata;
          if (ifprint == 0)
                printf ("\t%.2f", myoutput);
          break;
        case 6:
          myoutput = mydata * MAPCONV + MAPOFF;
          if (ifprint == 0)
                printf ("\t%.2f", myoutput);
          break;
        case 7:
          myoutput = mydata;
          if (ifprint == 0)
                printf ("\t%.2f", myoutput);
          break;
        } /*end switch*/

        fprintf (f2, "%f ", myoutput);

        //iterate internal buffer iteraror
        fndx +=2;

      }/*end for*/
      fprintf(f2, "\n");
      printf("\r");

      packetcntr++;
      ifprint = packetcntr % OUTPUTCNTR;
  }/*end check for complete packet*/
}/*continue reading*/
```

```
    /*iterate buffer*/
    if (bufitr == (BSIZE)) /*circle around if end of buffer*/
      bufitr = 0;
    else
      bufitr++;
  }
  printf("bye\n");

  close (fd);
  fclose(fp);
}
```

Figure 11: Sample output from user interface software

```
68.175000 0.000000 0.000002 2.574000 8.246080 2.496000
61.350000 0.000000 0.000001 2.691000 8.458240 2.515500
61.350000 0.000000 0.000001 2.769000 8.617360 2.574000
61.350000 0.000000 0.000001 2.769000 8.776480 2.613000
61.350000 0.000000 0.000001 2.769000 8.776480 2.613000
60.862500 0.000000 0.000001 2.730000 8.723440 2.613000
61.350000 0.000000 0.000001 2.749500 8.723440 2.613000
61.350000 0.000000 0.000001 2.749500 8.723440 2.613000
60.862500 0.000000 0.000001 2.749500 8.829520 2.652000
60.862500 0.000000 0.000001 2.691000 8.723440 2.593500
60.862500 0.000000 0.000001 2.749500 8.723440 2.613000
60.862500 0.000000 0.000001 2.749500 8.776480 2.613000
60.862500 0.000000 0.000001 2.749500 8.776480 2.632500
60.862500 0.000000 0.000001 2.730000 8.723440 2.613000
60.862500 0.000000 0.000001 2.749500 8.776480 2.632500
67.687500 0.000000 0.000001 2.847000 8.988640 2.691000
61.350000 0.000000 0.000001 2.769000 8.935600 2.652000
61.350000 0.000000 0.000001 2.710500 8.723440 2.593500
61.350000 0.000000 0.000001 2.730000 8.723440 2.593500
61.350000 0.000000 0.000001 2.769000 8.776480 2.613000
61.350000 0.000000 0.000001 2.749500 8.776480 2.632500
61.350000 0.000000 0.000001 2.730000 8.723440 2.593500
61.350000 0.000000 0.000001 2.749500 8.829520 2.632500
61.350000 0.000000 0.000001 2.730000 8.776480 2.613000
61.350000 0.000000 0.000001 2.730000 8.776480 2.613000
61.350000 0.000000 0.000001 2.691000 8.670400 2.574000
61.350000 0.000000 0.000001 2.749500 8.723440 2.593500
61.350000 0.000000 0.000001 2.769000 8.776480 2.632500
61.350000 0.000000 0.000001 2.749500 8.829520 2.632500
61.837500 0.000000 0.000001 2.730000 8.723440 2.613000
61.350000 0.000000 0.000001 2.769000 8.776480 2.632500
60.862500 0.000000 0.000001 2.749500 8.776480 2.613000
60.862500 0.000000 0.000001 2.749500 8.776480 2.613000
60.862500 0.000000 0.000001 2.652000 8.564320 2.554500
60.862500 0.000000 0.000001 2.749500 8.670400 2.574000
60.862500 0.000000 0.000001 2.730000 8.776480 2.613000
60.862500 0.000000 0.000001 2.749500 8.723440 2.632500
60.862500 0.000000 0.000001 2.730000 8.776480 2.613000
60.862500 0.000000 0.000001 2.730000 8.776480 2.632500
60.862500 0.000000 0.000001 2.749500 8.723440 2.613000
60.862500 0.000000 0.000001 2.730000 8.776480 2.613000
60.862500 0.000000 0.000001 2.730000 8.723440 2.593500
60.862500 0.000000 0.000001 2.749500 8.670400 2.593500
60.862500 0.000000 0.000001 2.769000 8.776480 2.632500
60.862500 0.000000 0.000001 2.730000 8.829520 2.632500
60.862500 0.000000 0.000001 2.730000 8.617360 2.574000
60.862500 0.000000 0.000001 2.749500 8.829520 2.632500
60.862500 0.000000 0.000001 2.749500 8.723440 2.593500
60.375000 0.000000 0.000001 2.749500 8.829520 2.652000
60.862500 0.000000 0.000001 2.671500 8.617360 2.574000
60.862500 0.000000 0.000001 2.710500 8.670400 2.574000
60.862500 0.000000 0.000001 2.749500 8.723440 2.593500
```