Luke Skaff
Automotive Diagnostic Interface
Senior Design Project
Old Dominion University
April 23, 2007

Abstract:

A microcontroller based automotive computer interface is described. The device utilizes a propriety serial interface to connect to an automotive engine control computer and retrieve current engine conditions. This information is then processed and output to a LCD. The hardware, software, and interface details are described.

Overview, History, Background, Introduction:

Project Overview:

In this project an interface and display was designed to retrieve automotive diagnostic data from a late 1980's to early 1990's General Motors automotive engine control unit / computer known as an ECU. This interface utilizes an Atmel AVR 8-bit microcontroller to perform serial communication over a one wire serial interface with the ECU. This diagnostic data is then processed by the AVR microcontroller and outputted to a LCD in an easy to read format for the user to view.

History:

The emissions laws put into action in the 1970's forced automobile manufacturers to search for new ways to make engines more efficient and cleaner. About the same time microcontrollers and embedded computers were starting to gain more market and become lower in cost. Automobile manufactures started experimenting with the use of microcontroller based embedded computers to control automobile engines around this time. A few cars were released with extremely primitive computer controlled fuel injection in the late 1970's including one by the General Motors (GM) Cadillac division. These systems where problematic and had great room for improvement. The main limitation at the time was the lack of computing power of microcontrollers and the inability to deal with the temperature extremes of an automotive application. The early to mid 1980's was a large crossover period where many cars went from strictly carbureted engines of the past to computer controlled carburetors and fuel injection. By the late

1980’s almost all cars and trucks were switched over to some form of computer controlled fuel injection.

A problem that arose very quickly was the lack of a standard for which mechanics could retrieve data from the onboard computers to diagnose problems. Some systems did not output any data that could be read by a mechanic. It was not until 1987 when California Air Resources Board (CARB) required that all new vehicles sold in California starting in the 1988 manufacturer's year have some basic On-Board Diagnostics (OBD) capability. This was a loose standard and every auto manufacturer went about implementing OBD in a very different way. In 1994 The Society of Automotive Engineers (SAE) mandated a cross-platform standard that all automotive manufacturers must follow starting in the 1996 production year called OBD-II. The on board diagnostics systems of previous years are commonly referred to as Pre-OBD or OBD-I.

The complexity of the on-board automotive control computer has increased significantly in the past two decades. The first engine computers were 8-bit microprocessors running code written in assembly. Some of these computers also did not have full control over very basic engine functions such as fuel and spark. Modern automobile control computers are 32-bit with code written in high level programming languages, for example C, and control every aspect of engine operation along with transmission, differentials, anti-lock brakes, traction control, and more.

Background on Project:

This project will focus on interfacing with an onboard automotive computer using the GM later 8192 baud OBD-I interface. GM started using the 8192 baud OBD-I interface in 1986 and continued using it widely until 1995 when a universal interface and protocol was mandated by SAE. The OBD-I interface on GM cars is a proprietary GM interface. There are many ways to retrieve data from the onboard engine control computer, commonly called Engine Control Unit (ECU) and it will be referred to as the ECU in the remainder of this report. There are many other names this onboard computer is called, the most popular being Engine Control Module (ECM) and Power-train Control Module (PCM) which refers to an ECU/ECM that also controls other parts of the power-train such as the transmission.

The ECU chosen for this project is one found is many late 1980's and early 1990's GM vehicles and is know by its part number, 1227730. The ECU has a customized version of the Motorola 6811 processor and runs code written in Assembly. The processor runs at 8.388 Mhz's, has 2 kilobytes of RAM, and a 32 kilobyte UV EPROM for code and calibration data. This code has since been disassembled and commented by auto hobbyists for the general public use. GM has platform dependent code which is vehicle specific and is called code "masks". The specific code mask a vehicle runs is determined by engine type, transmission configuration, which ECU is used in the vehicle, and other vehicle features. The code mask used in this project is the one used on the 1990-1992 Pontiac Firebird, 1990-1992 Chevrolet Camaro, and the 1990-1991 Chevrolet Corvette. ECU code along with engine specific data such as fuel maps are hard coded into a removable EPROM chip inside the ECU, which is called the "MemCal", which stands for Memory Calibration Unit. This made it easy for GM to use the same ECU across different platforms and for technicians to upgrade the calibration in case a problem was found after production.

Introduction:

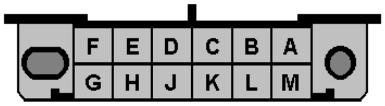
The purpose of the device designed in this project is to allow a user to retrieve information of the current status of their automobile ECU without the use of extremely expensive shop diagnostic computers. There are many devices on the market that allow home users to connect their laptops to their car ECU's and retrieve data via computer software. The purpose of this device is to eliminate the need for the laptop and have a microcontroller based circuit do the communication with the ECU and output the data to a LCD.

Discussion & Analysis:

Proprietary General Motors Assembly Line Diagnostic Link (ALDL):

As mentioned above, GM used its own proprietary communication method for communicating with its ECU's. Each GM OBD-I automobile has a connector that is called the Assembly Line Diagnostic Link (ALDL) connector seen in Figure 1 which is wired to the ECU. Mechanics can plug their shop diagnostic computer into this connector to communicate with the onboard ECU and retrieve engine sensor data and ECU error codes. During the period of the loose OBD-I standard; GM used two interfaces, the first of which was a 160 baud interface which was later replaced by an 8192 baud interface. The 8192 baud interface will be utilized in this project. The term interface refers to the communication as a whole including communication hardware, the physical communication method / protocol, and the software protocol. The protocol is the set of

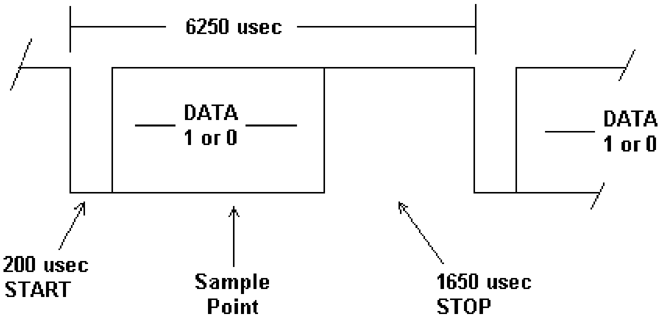
Figure 1:
ALDL connector



standards that must be followed when designing the hardware and software to communicate with the ECU properly and reliably.

The GM 160 baud OBD-I ALDL interface was the first of two OBD-I interfaces and uses synchronous serial communication utilizing one data line. This interface and data protocol does not allow for two way commutation with the ECU. The ECU constantly outputs diagnostic data at a rate of 160 baud over one wire on pin A of the ALDL connector. The receiver must be synced to receive at 160 baud in order to correctly receive the data bits at the "Sample Point" in Figure 2. Each data bit is synced with a falling start edge on the wave form before the data bit is sent. The drawbacks of this interface are the slow data speed and the inability for the diagnostic machine to send data to the ECU.

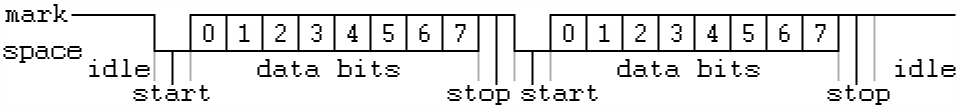
Figure 2:
160 Baud ODB-I protocol [7]



The GM 8192 baud OBD-I ALDL interface was the second and last of the two OBD-I interfaces. The 8192 baud interface communicates over pin M of the ALDL connector. The 8192 baud interface is more complex, faster, allows for more control of diagnostic data, and more diagnostic data to be retrieved from the ECU. The 8192 baud interface only uses one wire like the 160 baud interface but uses an asynchronous serial communication method and allows for two way communications over a single data line. The 8192 baud interface protocol is also a master/slave protocol and allows for multiple devices internal and external to the automobile to be connected to it. The ECUs that implement a 8192 baud interface do not constantly output data like the 160 baud interface; instead the attached device must send a short message requesting data from the ECU and the ECU will respond with a 60+ byte burst of data depending on the model of automobile.

The 8192 baud ALDL interface uses an asynchronous serial communication method as a means to transfer data over the data line. The 8192 baud interface also has a predetermined software communication procedure which can be considered the 8192 baud protocol. The 8192 baud interface uses asynchronous serial communication to perform the physical task of transmitting and receiving data. Asynchronous serial communication uses a start signal prior to each byte and a stop signal after each byte of data sent. This is the same method a serial RS-232 port on a computer uses with the difference that a RS-232 port has a separate transmit and receive line. In asynchronous serial communication the number of bits to be transmitted between start and stop bits, number of stop bits, parity options, and baud rate must be defined prior to any communication. The 8192 baud ALDL interface uses eight bits with one stop bit and no parity as seen in the timing diagram of Figure 3 below. The 8192 baud protocol uses an off standard baud rate of 8192 samples per second as the name implies. The closest standardized baud rate is 9600, this requires more effort in the hardware and software design to accommodate this off standard baud rate which is discussed later in this document.

Figure 3:
Asynchronous serial communication [1]



The data line is held high when inactive (idle). The transmission device pulls the data line low before transmitting data to trigger the receiving device to start sampling the data. The receiving device starts sampling the data line at the preset sample (baud) rate until it detects the stop bit. The connection is then resynchronized at the next start bit.

The ECU has six documented interface modes; each mode has a different command set, performs a different operation on the ECU, and receives different responses from the ECU. The six modes are mode 0, mode 1, mode 2, mode 3, mode 4, and mode 10. The function of these modes is discussed in detail later in this document. The modes are triggered by sending the ECU a specific command set.

For reference, refer to Appendix A for the list of commands each ECU mode is activated by. The most basic command set will include in this order: a message ID byte, message length byte, mode byte, and a checksum byte. The more complex commands will have other data bytes transmitted after the mode byte and before the checksum byte. The first command in the command set is the message ID byte which lets the ECU know what category of command it is receiving. Since all the commands listed above and in Appendix A are diagnostic commands they all have the same message ID byte of 0xF4. The next command in the command set is the message length byte. The most basic command set has a base message length byte of 0x56 and any other commands or data transmitted increments the message length number per extra byte transmitted. The mode byte is simply the desired ECU operation mode number. Other data bytes must be transmitted in the more complex data modes which are: mode 2, mode 3, and mode 4. The last byte to be transmitted is the checksum byte. The checksum byte is the one’s complement of the sum of all bytes transmitted.

Mode 0 is used by diagnostic equipment attached to the ALDL port to stop any communication on the data line. This would include in-car systems such as a body (suspension) computer and dash board modules which retrieve data from the ECU constantly over the diagnostic line. This communication must be stopped for the diagnostic equipment to accesses data from the ECU at full speed and constantly.

Mode 1 is used to retrieve all diagnostic data from the ECU. This operation mode is used in the field by test equipment; it is also used in this project. In this mode the diagnostic equipment or device will send the mode 1 commands over the data line and the ECU will reply with 64 bytes of diagnostic data. The bytes of data returned is dependent on the ECU and the code mask General Motors is running on the ECU, but the ECU chosen in this project returns 64 bytes of diagnostic data.

Mode 2 is used to dump 60 bytes of memory from the ECU to the ALDL data line starting with a user or device defined address. This is used mainly for debugging purposes and has little or no use for the average mechanic or technician. The most significant byte (MSB) of the desired memory start address and least significant byte (LSB) is transmitted after the mode 2 byte to tell the ECU which address to start the memory dump at. The ECU will then reply with the mode 2 command set and the desired 60 bytes of data.

Mode 3 will perform a dump of any eight defined addresses. The eight desired addresses are transmitted after the mode 3 command. These addresses are transmitted with the most significant byte first followed by the least significant byte. The ECU will then reply with the mode 3 command set and the desired 8 bytes of data. These addresses can be any addresses in the whole system including registers, RAM, and ROM. Mode 3 is also used in debugging and had little or no use for the average mechanic or technician.

Mode 4 is a controller mode where the user may change engine fuel, spark, and other engine parameters. This is a partially implemented feature and does not work on the production code mask used in this project. For this reason little is known how to operate this mode or how this mode is supposed to be commanded. It seems to be a GM development feature that was deactivated on the production code to avoid engine damage by untrained users.

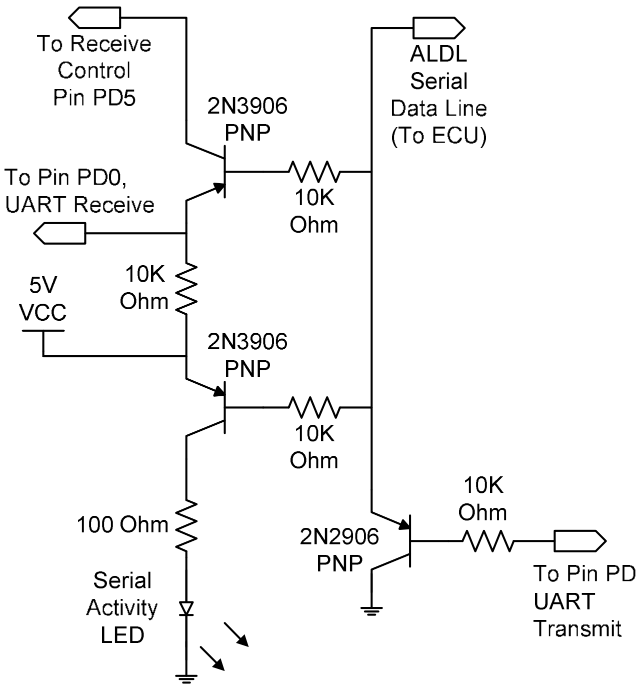
Mode 10 is used to clear any trouble codes the ECU has stored. A trouble code is a code stored in the ECU memory when it detects a fault or error in any of its sensor readings. This is commonly seen by the end user as a "service engine soon" light on the dash board. The ECU code mask used in the project has 64 trouble codes stored in 8 bytes, each bit representing a trouble code. After a mechanic or technician has retrieved these codes from the ALDL diagnostic port and repaired the problem the mode 10 command set can be transmitted to the ECU to clear stored trouble codes. This task can also be accomplished by removing battery power to the ECU.

Circuit Design:

An Atmel Mega324P AVR microcontroller was selected in this project to communicate and process the ECU data. This data is then output to a Hitachi 44780 controller based 4x20 LCD. The details of the AVR microcontroller and why it was chosen over similar microcontrollers, like a Microchip PIC microcontroller, is discussed below because is it outside the scope of this section. The Mega324P AVR is the most powerful AVR in a 40 pin DIP package, in production at the time of this report.

The Universal Asynchronous Receiver/Transmitter (UART) built into the Mega324P is designed to work with serial communication methods that use separate transmit and receive data lines such as SCI and SPI. The Mega324P, as with most microprocessors, is not designed to work with the off standard one wire interface used in this project. An interface circuit is needed to convert the one ALDL serial line to two data lines: a transmit line and a receive line.

Figure 4: UART Interface Circuit



One design issue with the microcontroller UART is that the transmit pin on the microcontroller is held high when inactive. Holding the serial line high when inactive is extremely common and used in almost all serial communication methods. If the transmit pin of the microprocessor was connected directly to the ECU serial line it would be held high when the ECU was trying to bring the line low for communication. A transistor is used on pin PD1 of the microcontroller, as seen in Figure 4, to isolate the transmit line. The ALDL serial line is held high by the ECU, the

PNP transistor only brings the serial data line low when the transmit pin on the microcontroller is brought low.

In order to prevent the receive UART from being filled with data that is being transmitted from the AVR microcontroller a transistor is use to isolate the receive line. The microcontroller sets pin PD5 high when transmitting so the receive UART on pin PD0 does not see the data being transmitted. When the microcontroller brings pin PD5 low the transistor is able to pull the receive UART pin, PD0, low when the ALDL line is low. An ALDL serial activity LED was added for debugging purposes. The LED comes on when the ALDL serial line is pulled low and remains off when there is no activity on the line, since the inactive state of a serial line is high.

To accommodate for the off standard baud rate of 8192 a crystal oscillator of 19.6608 Mhz was selected. At this clock rate the divider in the UART baud rate register, defined as UBRR, works out to zero percent error. As seen in the equation below an exact baud rate of 8192 is achieved with a UBRR register setting of 149 and a crystal oscillator of 19.6608 Mhz.

$$\text{BaudRate} = \frac{\text{Oscillator Frequency}}{16 \times (\text{UBRR} + 1)} = \frac{19660800\ \text{Mhz}}{16 \times (149 + 1)} = 8192$$

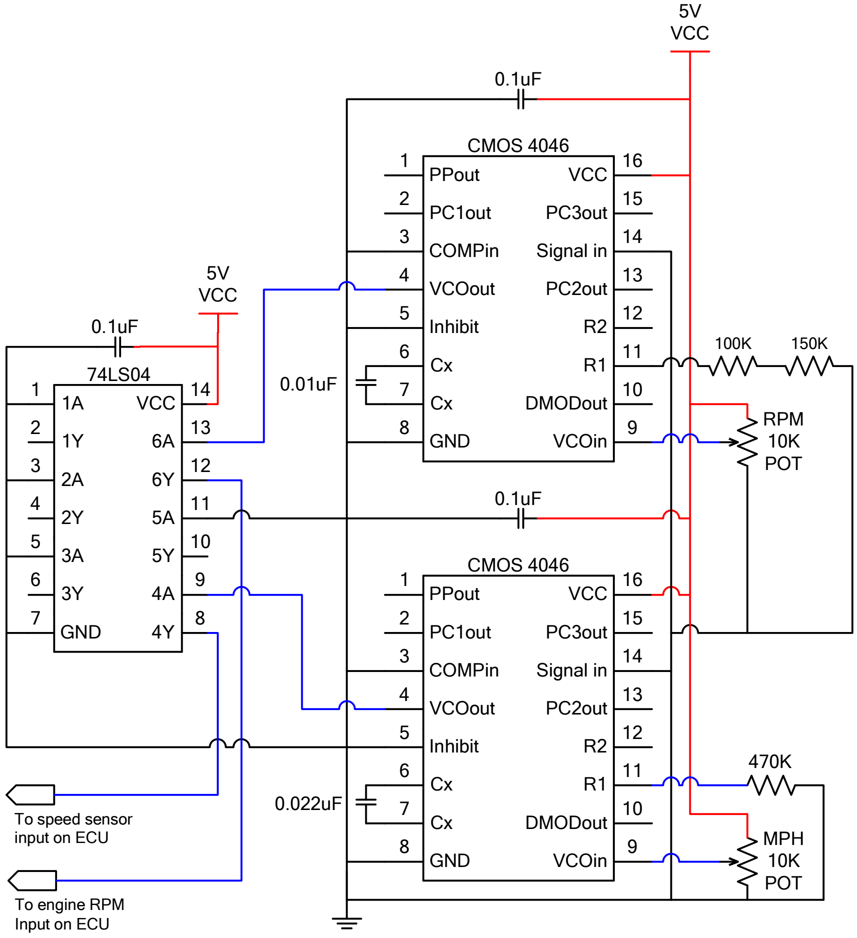
According to the microcontroller data sheet a percentage of error up to two percent is acceptable in most situations but in this project unnecessary error was eliminated for completeness and to reduce potential problems.

Automotive Computer Test Bench:

In order to make this project feasible during software development a method for running the automotive ECU on a bench had to be created. Each engine sensor the ECU reads is simulated so the ECU will not go into an error running state. Some sensors are simply variable resistors such as the temperature sensors and position sensors, so these could be simulated with a potentiometer. The ECU outputs a 5 volt reference for all the resistance based circuits. For the temperature sensors and position sensors, potentiometer resistance values were selected based on the resistance range of the original sensor the potentiometer was replacing. Two variable frequency square wave pulse generation circuits had to be designed to simulate the vehicle speed sensor and the engine RPM input. The frequency of each square wave is directly proportional to the speed in MPH and engine revolutions in RPM the ECU reads.

Designing the circuits to simulate the vehicle speed sensor and engine RPM output required more then just a potentiometer. Experimentation was done with using a 555 Timer circuit and using a set capacitance value with a potentiometer in the RC section of the 555 Timer circuit. The result was the inability to produce a wide enough frequency output range and the inability to bring the output down to zero hertz for an off state. So a Voltage Controlled Oscillator (VCO) IC was selected to perform the task. A low cost CMOS 4046 Phase-Locked Loop with VCO IC was selected for the job. The inputs the for phase-locked loop portion of the IC are tied to an inactive state and affectively disabled. In CMOS all inactive inputs must be tied low to avoid damage to the IC and to produce reliable results. The VCO portion of the IC was used to achieve the desired square wave output. Since the IC was CMOS and can only output small output currents in the 1-5mA range, the output was run through a TTL 74LS04 inverter which can provide up to 20mA output which is more suitable for the ignition module signal load. An inverter was selected due to easy availability. A TTL buffer or many other chips could be used to achieve the same affect.

Figure 5:
RPM and MPH input
Square wave generator circuit



The circuit seen in Figure 5 is the finalized design of the square wave generator circuit. It consists of two variable square wave outputs, one for the RPM input and one for the MPH input on the ECU. The output frequency is determined by the capacitance between pin 6 (Cx) and 7 (Cx), the resistance on pin 11 (R1) and 12 (R2), and the voltage present on pin 9 (VCOin). Before determining the correct values to be placed on these pins the desired frequency range to be inputted into the ECU must first be determined.

The maximum RPM the ECU can read is 6375 RPM's. This is because the engine RPM is stored in one byte of memory with a multiplier of 25. This results in 255 multiplied by 25 which equals 6375 RPM. The engine the ECU operates is only capable of 6000 RPM's so this limit is not a problem. Some ECU's and or code masks can read up to 9000 RPM's so it was decided to make the RPM generator circuit output a maximum frequency equivalent to 9000 RPM's for future uses. The distributor which generates the RPM signal for the ECU is connected to a camshaft which rotates at half the speed of the engine output crankshaft. The distributor outputs one pulse per cylinder fired. So the distributor is outputting 4 pulses per engine rotation on a V8 engine. The maximum frequency desired for the circuit to produce is calculated below:

$$\frac{9000\text{ RPM}}{60\text{ Seconds in a minute}} = 150\text{ Rotations per Second}$$

$$150\text{ Rotations per Second} \times 4\text{ Pulses per Revolution} = 600\text{Hz}$$

The maximum speed the ECU can read is 255 MPH since it is an 8-bit computer and the speed is stored in one byte of memory. The vehicle speed sensor outputs 2000 pulses per minute at a speed of one mile per minute. A speed of one mile per minute is 60 miles per hour (MPH). So the pulse per second (Hz) at 60 miles per house is 2000 divided by 60 which equals 33.3 Hz. Using this ratio it can be determined that a pulse of 141.6 Hz is needed to max out the ECU's MPH reading of 255 MPH. Of course this speed would never be achieved in real life but for testing purposes the full range of the ECU is utilized. It was chosen to round up the MPH pulse to a max of 150Hz for simplicity.

To calculate what resistor and capacitor values were needed to achieve these desired frequencies a trial and error method was used. The circuit was built on a bread board and an oscilloscope was used to measure the output frequency. The starting test values where chosen from the 4046 datasheets graphs and final values where chosen after educated guesses from the results of the pervious test.

AVR Microcontroller:

The Atmel AVR is an 8-bit microcontroller with Harvard architecture that runs a RISC instruction set. The Atmel AVR was designed to minimize code storage size and execution time of code written in assembly and C. This microcontroller is very low cost, starting at fifty cents for one unit for its simplest model. The AVR also has free development software platform and low cost development equipment, for example the programmer is twenty dollars and the full hardware development kit is eighty.

The AVR's strongest competitor is the Microchip PIC 8-bit microcontroller [8]. An AVR microcontroller was chosen over the PIC microcontroller which is taught in the EET curriculum due to the reasons presented below. The AVR has true single cycle execution unlike the PIC which divides the clock frequency by a factor of four. So the AVR runs four times faster at the same crystal speed of a PIC. The AVR microcontroller boasts 131 instructions compared to the PIC which has 35. This allows for smaller code length to accomplish the same task. Also unlike the one working register in the PIC the AVR has 32 which allows for more efficient code to be written. The PIC has one pointer and the AVR has three. Another drawback of the PIC is that the stack only has a depth of eight in the most common 16C and 16F series. The AVR stack is only limited to the amount of free memory so more reusable code can be written reducing code complexity and development time.

Software code:

The code for this project was written in C and compiled with a free open source compiler named WinAVR. WinAVR is a part of AVR studio, Atmel's development software. Programming in C allows for much faster development time, easier to follow code, and the use of more complex operations with greater ease. The drawback of C code is the larger code length when downward compiled to machine code with a compiler and a longer execution time when compared to efficient code written in assembly. This only becomes a problem in extremely time critical tasks. For most cases the time difference is unnoticeable and the memory use is not a problem. Many companies develop microcontroller based software in C do its shorter development time and ease of maintenance.

The entire code for this project can be viewed in appendix E. The code is split up into many functions to perform specific tasks. There are four main tasks accomplished by the code. The first is to initialize the command registers of the microcontroller and initialize the LCD. Secondly the car and diagnostic data is retrieved from the ECU and stored in the AVR's memory. Third the stored car data is processed and calculations are made to convert the raw values to the real world the sensors are reading. Fourth the processed data is outputted to the LCD. The second through the fourth task are looped continually so the LCD will always have the most up to date information from the ECU.

The initialization process of the code is used to configure the microprocessor and the LCD before use. The I/O ports of the microcontroller are configured for their proper function either input or output. The UART is configured for the proper baud rate, bit size, and number of stop bits. The 8-bit operation mode command is transmitted to the LCD along with commands to reset the display and place the cursor at the being of the display.

The second main task of the code operation is retrieving car and diagnostic data from the ECU which is comprised of a few functions. The first thing the code must do is send out the mode 1 command set to the ECU which commands the ECU to return a dump of the diagnostic data. This data is then stored one byte at a time in an array to be used at a later. Timeout detection is also built into the receive routine so if a transmission error occurs the code will timeout instead of getting hung up in the receive loop.

The third main task of the code operation is to process the stored diagnostic data from the previous task. The diagnostic data received from the ECU is in a raw, unprocessed format, and must be processed to produce data that makes sense to the end user. The data has set dividers and multipliers that must be performed to the raw values to produce the real world value the sensor is truly reading. These values are then converted to ASCII for output to the LCD.

The forth and final step is to output the processed ASCII data to the LCD. This includes putting headings in front of values so the users can identify what value they are reading. The data outputted to the user are: engine RPM, vehicle MPH, fuel block value know as BLM, coolant temperature (CTS), intake manifold air pressure (MAP), throttle position percent (TPS), battery voltage, and ECU trouble codes.

Conclusion:

This project as a whole successfully allows a user to read diagnostic data from an automotive ECU from a 1990-1992 Pontiac Firebird, 1990-1992 Chevrolet Camaro, and the 1990-1991 Chevrolet Corvette. This device performs the task of expensive diagnostic equipment with low cost parts. The project had many obstacles to overcome such as coming up with a method to convert a two wire UART to a one wire serial system, learning embedded C code used on an AVR, and the interfacing scheme used to communicate with the ECU. All these obstacles were overcome and the project was a success. More than the target 100 hours, stated in the project guidelines, were put into making this project operational but the material learned and the expanded education made it worthwhile.

References:

1) *Asynchronous serial communication*. (n.d.). Wikipedia. Retrieved March 24, 2007, from http://en.wikipedia.org/wiki/Asynchronous_serial_communication

2) *Atmel AVR*. (n.d.). Wikipedia. Retrieved February 25, 2007, from http://en.wikipedia.org/wiki/Atmel_AVR

3) *AVR 8-Bit RISC*. (n.d.). Atmel. Retrieved February 25, 2007, from http://www.atmel.com/products/avr/

4) Barnett, R., O'Cull, L., & Cox, S. (2007). *Embedded C Programming and the Atmel AVR* (2nd ed.). Clifton Park, NY: Thomson Delmar Learning.

5) *Error Codes from Service Engine Light*. (n.d.). Retrieved April 14, 2007, from http://www.thirdgen.org/service-engine-light-error-codes

6) Gadre, D. V. (2001). *Programming and Customizing the AVR Microcontroller*. New York, NY: McGraw-Hill.

7) *GM Diagnostics*. (n.d.). ECM Guy. Retrieved March 10, 2007, from http://www.geocities.com/ecmguy.geo/diagnostics/do_diag.html

8) Mitchell. (2006). *Automotive Shop Manuals*. Mitchell.

9) Morton, J. (2002). *AVR an Introductory Course*. Woburn, MA: Newnes.

10) *On-Board Diagnostics*. (n.d.). Wikipedia. Retrieved March 10, 2007, from http://en.wikipedia.org/wiki/On_Board_Diagnostics

11) Pardue, J. (2005). *C Programming for Microcontrollers*. Knoxville, TN: Smiley Micros.

12) *Win AVR Compiler*. (n.d.). Retrieved February 25, 2007, from http://winavr.sourceforge.net/

Appendices:

Appendix A - ALDL Mode Command Table [7]

	Mode 0	Mode 1	Mode 2	Mode 3	Mode 4	Mode 10
Comments:	Clear Communications	Diagnostic Data Request	60 Byte Data Dump with Start Address	Dump 8 Defined Addresses	Controller mode	Clear Error Mode
Command						
Message ID:	0xF4	0xF4	0xF4	0xF4	N/A	0xF4
Message Length:	0x56	0x56	0x58	0x65	N/A	0x56
Mode:	0x00	0x01	0x02	0x03	N/A	0x0A
Other Data:			Address MSB Address LSB	Address 1 MSB Address 2 LSB .. .. Address 8 MSB Address 8 LSB	Undocumented	
Checksum sent:	0xB5	0xB4	Calculated	Calculated	Calculated	0xAB
ECU Response						
Message ID:	0xF4	0xF4	0xF4	0xF4	N/A	0xF4
Message Length:	0x56	0x95	0x96	0x63	N/A	0x56
Mode:	0x00	0x01	0x02	0x03	N/A	0x0A
Other Data:		Data Byte 1 .. .. Data Byte 63	Byte 1 .. .. Byte 60	Byte 1 .. .. Byte 8	Undocumented	
Checksum sent:	0xB5	Calculated	Calculated	Calculated	Calculated	0xAB

Note 1: All commands are hex values

Note 2: A checksum of "Caluclated" means the checksum much be calculated by the transmitter at time of transmission

<u>Appendix B</u> – ECU Diagnostic Data Stream Returned in Mode 1 [7]

		Function	Equation
1		**EPROM ID, (MSB)**	
2		**EPROM ID, (LSB)**	
3		**MALFFLG1 MALFUNCTION WORD 1**	
	b0	CODE 23, MAT SENSOR LOW	
	b1	CODE 22, TPS LOW	
	b2	CODE 21, TPS HIGH	
	b3	CODE 16, NOT USED	
	b4	CODE 15,COOLANT SENSOR LOW TEMP	
	b5	CODE 14, COOLANT SENSOR HIGH TEMP	
	b6	CODE 13, o2 SENSOR	
	b7	CODE 12, NO DPR's	
4		**ERROR FLAG 2**	
	b0	CODE 35 not used	
	b1	CODE 34 MAP SENSOR LOW	
	b2	CODE 33 MAP SENSOR HIGH	
	b3	CODE 32 EGR DIAGNOSTIC	
	b4	CODE 31 not used	
	b5	CODE 26 not used	
	b6	CODE 25 MAT SENSOR HIGH	
	b7	CODE 24 Vss	
5		**ERROR FLAG 3**	
	b0	CODE 51 EPROM ERROR	
	b1	CODE 46 VATS FAILED	
	b2	CODE 45 o2 RICH	
	b3	CODE 44 o2 SENSOR LEAN	
	b4	CODE 43 ESC FAILURE	
	b5	CODE 42 EST MONITOR ERROR	
	b6	CODE 41 CYLINDER SELECT ERROR	
	b7	CODE 36 not used	
6		**ERROR FLAG 4**	
	b0	CODE 63 NOT USED	
	b1	CODE 62 OIL TEMP HIGH	
	b2	CODE 61 not used	
	b3	CODE 56 not used	
	b4	CODE 55 not used	
	b5	CODE 54 FUEL PUMP VOLTAGE	
	b6	CODE 53 OVER VOLTAGE	
	b7	CODE 52 OIL TEMP LOW	
7		**ERROR FLAG 5**	
	b0	not used	

	b1	not used	
	b2	not used	
	b3	not used	
	b4	not used	
	b5	CODE 66 not used	
	b6	CODE 65 not used	
	b7	CODE 64 not used	
8		**COOLANT TEMPERATURE, A/D COUNTS**	Deg c = n x.75 - 40
9		**START UP COOLANT TEMPERATURE**	Deg c = n x.75 - 40
10		**TPS A/D COUNTS**	VDC = n x (5/255)
11		**ENGINE SPEED (RPM)**	RPM = n x 25
12		**NEW DRP, TIME BETWEEN REFERENCE PULSES (MSB)**	
13		**NEW DRP+1 TIME BETWEEN REFERENCE PULSES (LSB)**	usec = ([n13]*256 + [n14]) x 15.26
14		**MPH/1**	
15		**NVMW2, NON-VOLATILE MODE WORD 2**	
	b0	not used	
	b1	not used	
	b2	not used	
	b3	1 = PLUGGABLE MEMORY FAILURE (err51)	
	b4	not used	
	b5	1 = VATS OK	
	b6	not used	
	b7	1 = ESC ENABLED BY DELTA COOLANT	
16		**ENG/Vss RATIO TO DETERMINE GEAR**	N = RPM/MPH
17		**OXYGEN SENSOR**	VOLTAGE = N x 4.42
18		**o2 SENSOR RICH/LEAN TRANSITION COUNTER**	
19		**BASE PULSE (FUEL) C/L FINE CORRECTION**	
20		**BLM**	
21		**BLM CELL Number**	
22		**CLOSED LOOP INTEGRATOR**	
23		**IDLE SPEED, PRESENT IAC MOTOR POSITION**	Steps
24		**SCALED TPS, auto zero**	%TPS = N/2.56
25		**DESIRED IDLE SPEED, RPM/12.5**	
26		**MANIFOLD AIR PRESSURE, A/D CTS**	VOLTS = N x (5/255)
27		**SC1 SDI STATUS OF SC1 INPUT DISCRETES**	
	b0	NOT USED	
	b1	A/C LOW PRESSURE SWITCH	
	b2	SECOND GEAR	
	b3	NOT USED	
	b4	NOT USED	
	b5	A/C REQUEST (0 = A/C REQUESTED)	
	b6	NOT USED	
	b7	2ND FAN REQUEST	

28		**FMD SDI INPUT STATES TO FMD VIA SSR**	
	b0	COOLANT SWITCH (1 = 348 ohm, 0 = 4 K)	
	b1	COP2 (< 54 usec BETWEEN FALLING EDGES)	
	b2	EST ENABLE	
	b3	PORT, PIN8	
	b4	FUEL PUMP ENABLE	
	b5	not used	
	b6	IRQ ENABLE	
	b7	DATA STEER (0 = BYTE 1, 1 = BYTE 2)	
29	1 l	**NVMW1 NON-VOLATILE MODE WORD**	
30		**MAT, A/D COUNTS**	Table Lookup
31		**EGR DUTY CYCLE**	%DC = N/2.56
32		**CHARCOAL CANISTER PURGE DUTY CYCLE**	%DC = N/2.56
33		DIAGMW2 DIAGNOSTIC MODE WORD 2 (CURRENT MALF FLAGS)	
	b0	1 = err 41 INDICATED (CYLINDER SELECT ERR)	
	b1	1 = err 25 THIS PASS INDICATED	
	b2	1 = REF PULSE IN CURRENT 100 msec.	
	b3	1 = DRP IN LAST 100 msec.	
	b4	1 = err 54 LOCKED IN	
	b5	E = err 54 PRESENT	
	b6	b6 1 = PASSED err 54A	
	b7	1 = ESC ENABLED	
34		**BATTERY VOLTAGE, A/D COUNTS**	Vbatt = n/10
35		**FUEL PUMP POWER**	Vbatt = n/10
36		**DIAGMW4 DIAGNOSTIC MODE WORD 4 (CURRENT MALF FLAGS)**	
	b0	1 = MALF 32 ACTIVE	
	b1	not used	
	b2	1 = EGR DIAGNOSTIC TEST IN WORK	
	b3	OPTION FOR 1 PASS	
	b4	TEST CYCLE TIME FLAG	
	b5	1 = A/C FIRST PASS WITH HIGH MPH	
	b6	1 = A/C CLUTCH DISABLED DUE TO HIGH MPH	
	b7	1 = err 52 or 62 PRESENT	
37		**MIN LEARNED IAC POSITION (KEEP ALIVE)**	STEPS
38		**LINEARIZED OIL TEMP (MSB)**	Deg c = n x.75 - 40
39		**TOTAL UNLIMITED SPARK ADV. REL TO TDC (MSB)**	
40		**TOTAL UNLIMITED SPARK ADV. REL TO TDC (LSB)**	

Double byte value in 2's complement representation

		If Bit 7 of MSB = 0 then result is positiv	
		Value = ([n41] x 256 + [n42])	
		If Bit 7 of MSB = 1 then result is negative	
		Value = 65536 - ([n41] x 256 + [n42])	
		Deg Spk = value x 90/256	
41		**UNLIMITED SPARK ADV. REL TO REF. PULSE (MSB)**	
		TOTAL UNLIMITED SPARK ADV. REL TO TDC (LSB)	
		Double byte value in 2's complement representation	
		If Bit 7 of MSB = 0 then result is positive	
42		**Value = ([n41] x 256 + [n42])**	
		If Bit 7 of MSB = 1 then result is negative	
		Value = 65536 - ([n41] x 256 + [n42])	
		Deg Spk = value x 90/256	
43		**ESC (KNOCK) SIGNAL INPUT**	COUNTS
44		**ESC (KNOCK RETARD)**	Deg = n x 45/256
45		**INJECTOR BASE PULSE WIDTH (MSB)**	
46		**OBPINJ + 1 INJECTOR BASE PULSE WIDTH (LSB)**	msec = ([n45] x 256 + [n46])/65.536
47		**TOTAL FUEL AIR VALUE (FINAL) (MSB)**	
48		**TOTAL FUEL AIR VALUE (FINAL) (LSB)**	A/F RATIO = 6553.6/([n47] x 256) + 6553.6/[n48]
49		**RUNNING TOTAL OF FUEL DELIVERED (MSB)**	
50		**RUNNING TOTAL OF FUEL DELIVERED (LSB)**	usec = ([n49] x 256 + [n50])*15.26
51		**RUNNING TOTAL OF DISTANCE TRAVELED**	Miles = n/2000
52		**ENGINE RUNNING TIME IN SECONDS (MSB)**	
53		**ENGINE RUNNING TIME IN SECONDS (LSB)**	Sec's = ([n52] x 256 + [n53])
54		**Mode Word 2**	
	b0	not used	
	b1	MALF 14 OR 15 THIS START UP	
	b2	DRP, (6.25 MSEC CHECK)	
	b3	1 = IN CCM MODE	
	b4	DIAGNOSTIC SWITCH IN DIAG. POSITION	
	b5	DIAGNOSTIC SWITCH IN ALDL POSITION	
	b6	HIGH BATTERY VOLTAGE-DISABLE MCU SOL.DIS.	
	b7	SHIFT LIGHT, 1 = ON	
55		**Torque Converter Clutch Mode Word**	
	b0	1 = TCC LOCKED	
	b1	1 = COAST RELEASE	

		TPS THRESHOLD IN USE (HI MPH)
	b2	1 = 4-3/4-2 DOWNSHIFT RELEASE IN PROGRESS
	b3	1 = STATUS OF FOURTH GEAR LAST PASS
	b4	1 = TCC LOCKED FOR PASS BY NOISE
	b5	not used
	b6	not used
	b7	not used
56		**Fuel Modeling Device Byte 1**
		GEMERIC ALDL BYTE = 44
	b0	PARK/NEUTRAL SWITCH (1 = DRIVE)
	b1	1 = IN 3RD OR 4TH GEAR
	b2	1 = IN 4TH GEAR
	b3	0 = POWER STEERING CRAMP - CHANGE FROM '89
	b4	not used
	b5	not used
	b6	0 = HIGH A/C HEAD PRESSURE INDICATED (IF N.O. SWITCH)
	b7	1 = A/C CLUTCH ENGAGED
57		**Mode Word 1**
	b0	ADVANCE FLAG, 0 = ADV., 1 = RTD
	b1	1 = HIGHWAY FUEL TIMER ENABLED
	b2	INTERRUPT SERVICE EXC. 6.25 msec
	b3	1 = ALL FAN 1 PID STEPS ADDED (FAN 1 ENABLED)
	b4	1 = 1st PASS WITH 1 ROAD SPEED PULSE
	b5	AIR COND. CLUTCH FLAG (0 = A/C CLUTCH ON)
	b6	BYPASS CHECK ENABLE
	b7	ENGINE RUNNING FLAG (1 = RUNNING)
58		**Non Volatile MW**
	b0	1 = 02 SENSOR READY
	b1	1 = CLOSED LOOP TIMER TIMED OUT
	b2	not used
	b3	1 = BAD SHUTDOWN
	b4	Not used
	b5	1 = IAC KICKDOWN ENABLED
	b6	1 = KICKDOWN ENABLED
	b7	1 = err 42 FAILED (EST monitor)
59		**Computer Aided Ratio Selection Mode Word**
	b0	1 = CARS DISABLED DUE TO LOW BAROMETRIC PRESSURE
	b1	1 = CARS ACTIVE

	b2	not used
	b3	not used
	b4	1 = TRANSMISSION IN 4th. GEAR
	b5	1 = TRANSMISSION IN 1st, GEAR
	b6	1 = WAIT FOR Vss RESET
	b7	not used
60		**Closed Loop CC Mode Word**
	b0	BOOKKEEPING FLIP FLOP
	b1	1 = Use F69 ALT TABLE
	b2	1 = IDLE
	b3	1 = UNDERSPEED IDLE SPARK, 0 = OVERSPEED
	b4	1 = Decel Fuel Cut Ooff STALL SAVER ENABLED
	b5	1 = USING KF93 MULT TRIM TO D-MAP A.E. ENABLE THRESHOLD
	b6	1 = Non Volatile. MEMORY BOMBED
	b7	1 = Has been in Closed Loop at least once since restart
61		**AIR MW**
		GENRERIC ALDL BYTE = 18
	b0	1 = 100 msec OLD CCP PURGE ON FLAG (0 = OFF)
	b1	1 = AIR CONTROLLED, 0 = AIR DIVERTED
	b2	1 = AIR SWITCHED TO PORT (If air is controlled)
	b3	1 = FAN 1 REQUESTED
	b4	1 = FAN 2 REQUESTED
	b5	1 = 'OLD' FAN 1 STATE WAS ON
	b6	1 = ALL FAN 2 PID STEPS ADDED (Fan 2 enabled)
	b7	1 = DECEL ENLEANMENT ACTIVE
62		**LCCP MW**
	b0	1 = CAN PURGE ACTIVE
	b1	1 = TIME 1st. REF TO ENG RUN
	b2	1 = MALFS HAVE OCCURRED
	b3	1 = IN 8192, Mode 4, Bypass fuel mode
	b4	2nd. TIME COOLANT
	b5	1 = err 43A (voltage presence check) Indicated
	b6	1 = KICKDOWN REQUEST
	b7	1 = TIME OUT FINISHED
63		**Mode Word Fuel/Air 1**
	b0	1 = IN SINGLE FIRE MODE
	b1	BLM ENABLE FLAG, 1 = ENABLE STORE
	b2	1 = DELIVER 0 FUEL (Single fire)
	b3	1 = ALLOW SINGLE FIRE DISABLE
	b4	1 = VEHICLE SPEED SENSOR FAILURE
	b5	1 = EECC SLOW 02 RICH, 0 = SLOW 02 LEAN

b6 RICH-LEAN FLAG (1 = RICH, 0 = LEAN)
b7 CLOSED LOOP FLAG, 1 = CLOSED LOOP

Appendix C – ECU Trouble Codes for ALL GM OBD-I systems [5]

12. No reference pulses to Electronic Control Module (ECM).
13. Oxygen sensor signal stays lean during warm engine cruise
14. High temperature indicated at engine coolant temp sensor
15. Low temperature indicated at engine coolant temp sensor
16. High battery voltage OR Direct ignition system open or shorted to ground
17. RPM signal problem

18-20. N/A

21. High voltage at throttle position sensor
22. Low voltage at throttle position sensor OR Fuel cutoff relay circuit open or shorted to ground
23. Low temperature at manifold air temperature sensor OR Throttle position sensor error
24. Circuit fault in vehicle speed sensor
25. High temperature at manifold air temperature sensor OR Vacuum switching valve circuit open or shorted to ground OR High voltage at ATS sensor
26. Fault in quad driver module
27. Fault in 2nd gear switch
28. Fault in 3rd gear switch
29. Fault in 4th gear switch
30. N/A
31. Low voltage at manifold absolute pressure sensor OR Fuel injector OR Park or neutral switch OR CAM diagnostic OR Governor malfunction OR Wastegate overboost OR Wastegate eletrical signal open or shorted to ground
32. Fault in barometric pressure sensor circuit OR Fault in exhaust gas recirculation valve diagnostic switch OR Fault in electronic vacuum regulator valve
33. High voltage (low vacuum) at mass air flow sensor (or MAP sensor)
34. Low voltage (high vacuum) at mass air flow sensor (or MAP sensor)
35. Idle speed can not be set to desired RPM
36. Burn off at mass air flow sensor OR Problem in transmission shift OR Fault in direct ignition system OR Missing pulses in electronic spark timing signal
37. N/A
38. Fault in torque converter clutch brake switch
39. Fault in torque converter clutch circuit
40. N/A
41. Fault at cam sensor OR Cylinder select error OR Tach input error
42. Fault at electronic spark timing circuit OR Fault at direct ignition system OR Fault at fuel cutoff relay circuit
43. Low voltage at electronic spark timing circuit
44. Oxygen sensor lean
45. Oxygen sensor rich
46. Fault at vehicle anti-theft sytem OR Fault at power steering switch
47. Problem at Electronic Control Module (ECM)
48. Misfire
49. Vacuum leak
50. N/A

51. PROM error

52. Problem at Electronic Control Module (ECM) - Missing fuel calpac missing OR Analog to digital converter error OR Fault at quad driver module OR Low voltage at oil temperature sensor

53. High voltage at battery OR High voltage at exhaust gas recirculation valve OR Voltage reference error OR Problem at vehicle anti-theft system

54. Low voltage at fuel pump OR Low voltage at Fuel pump relay OR Output failure at quad driver module

55. Problem at Electronic Control Module (ECM) - ECM failure OR Serial bus error OR Fuel lean malfunction

56. Low coolant or corrosivity or fault in port throttle system vacuum sensor

57. N/A

58. Problem at vehicle anti-theft system

59-60. N/A

61. Oxygen sensor degraded OR Port throttle system error OR Cruise control problems

62. Gear switch input diagnostics OR High voltage at oil temperature sensor OR Fault in cruise control- vacuum solenoid circuit

63. High voltage at manifold absolute pressure sensor OR Fault in exhaust gas recirculation valve OR Fault at right oxygen sensor

64. Low voltage at manifold absolute pressure sensor OR Fault in exhaust gas recirculation valve OR Right oxygen sensor lean

65. Failure at exhaust gas recirculation valve OR Failure at injector peak/hold diagnostic OR Right oxygen sensor rich OR Fault at cruise control position sensor

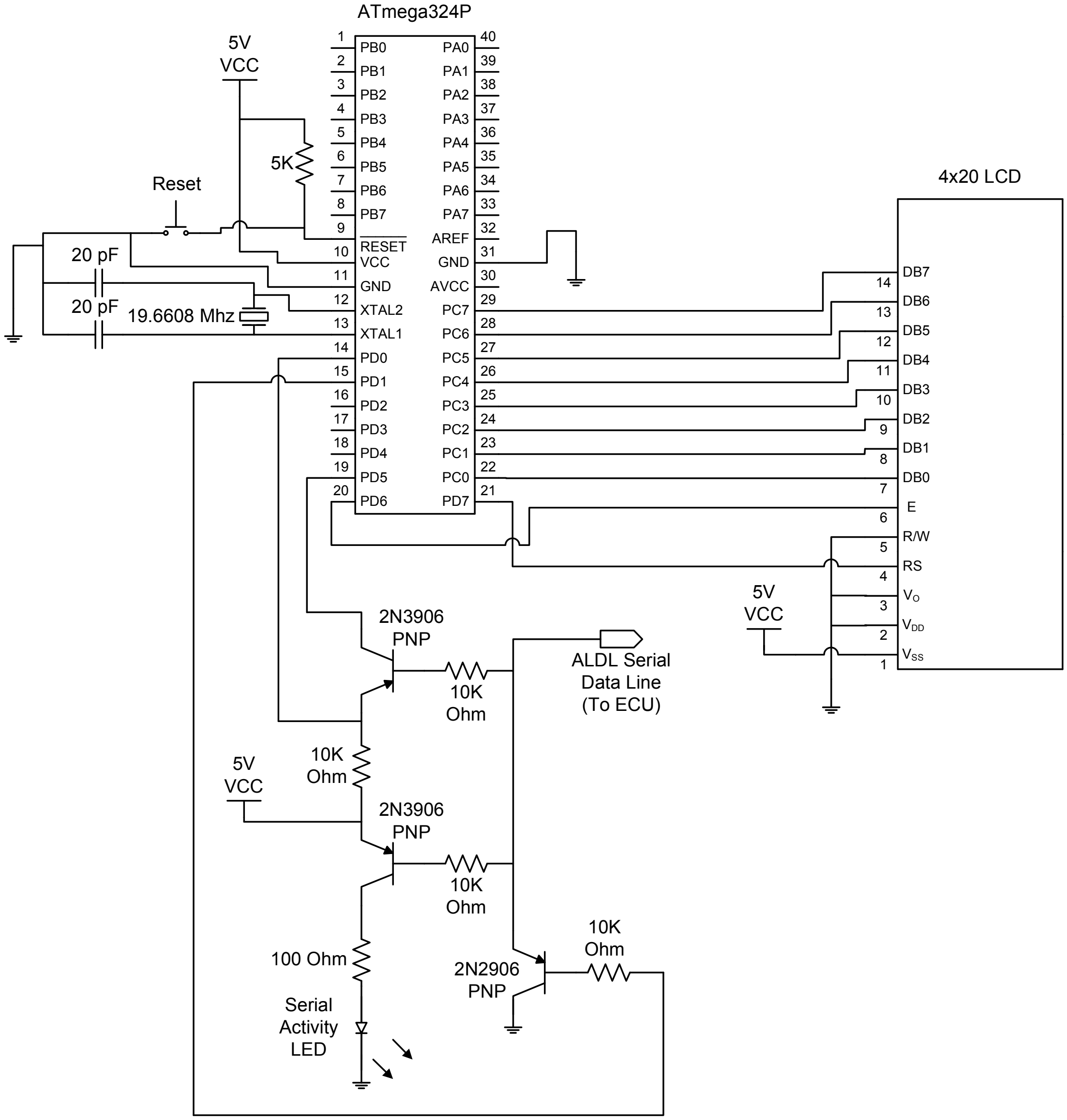
66. Internal reset of Electronic Control Module (ECM)

67. Fault at cruise control switch

68. Fault at cruise control switch

69. Fault at air conditioner pressure switch

Appendix D – Full Schematic of circuit



Appendix E – AVR Microcontroller Code

```
//************************************************************
// Auther: Luke Skaff
// EET 480 - Senior Project
// Spring 2007
// Automotive engine computer (ECU) diagnostic interface
// Compiled in WinAVR
//************************************************************


//***********************************************************
//                          Program description
// Sends Mode 1 command set to ECU
// Stores Mode 1 data
// Calculates desired values from stored data
// Outputs calculated data to LCD
//***********************************************************

// Dependent librarys
#include <avr/io.h>
#include <avr/pgmspace.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <util/delay.h>

#define F_CPU 19660800UL   // CPU crystal speed 19.6608Mhz
#define FOSC 19660800UL    // CPU crystal speed 19.6608Mhz
#define BAUD 8192UL        // 8192 Baud rate



// ------ PORT & PIN Definitions -------
#define   LCD_PORT    PORTC    //8-BIT LCD data lines
#define   RS_PORT     PORTD    //Port RS line in on
#define   RS          7        //Pin # of RS line
#define   E_PORT      PORTD    //Port Enable line in on
#define   E           6        //Pin # of Enable line

#define   RXcontrol   5        //Pin # RX enable transistor

#define   DataM      0xFF


#define Bit_Set(port,bit_num) (port = port | (0x01 << bit_num))
#define Bit_Clear(port,bit_num) (port = port & ~(0x01 << bit_num))

#define DataStreamSize  63    //Size of receiving data stream
#define CarDataSize     22    //Size of ASCII car data


unsigned char DataStream[DataStreamSize];
#define  Def_Offset     3
#define  DEF_ERROR1     3 + Def_Offset
#define  DEF_ERROR2     4 + Def_Offset
#define  DEF_ERROR3     5 + Def_Offset
#define  DEF_ERROR4     6 + Def_Offset
#define  DEF_ERROR5     7 + Def_Offset
#define  DEF_CTS        8 + Def_Offset
#define  DEF_RPM        11 + Def_Offset
#define  DEF_MPH        14 + Def_Offset
#define  DEF_BLM        20 + Def_Offset
#define  DEF_BLM_CELL   21 + Def_Offset
#define  DEF_STPS       24 + Def_Offset   // Scaled Throttle Position Sensor
#define  DEF_MAP        26 + Def_Offset   // volts = N x (5 / 255)
                                          // KPa = N * .369 + 10.354
#define  DEF_BATT       34 + Def_Offset   // volts = N / 10
```

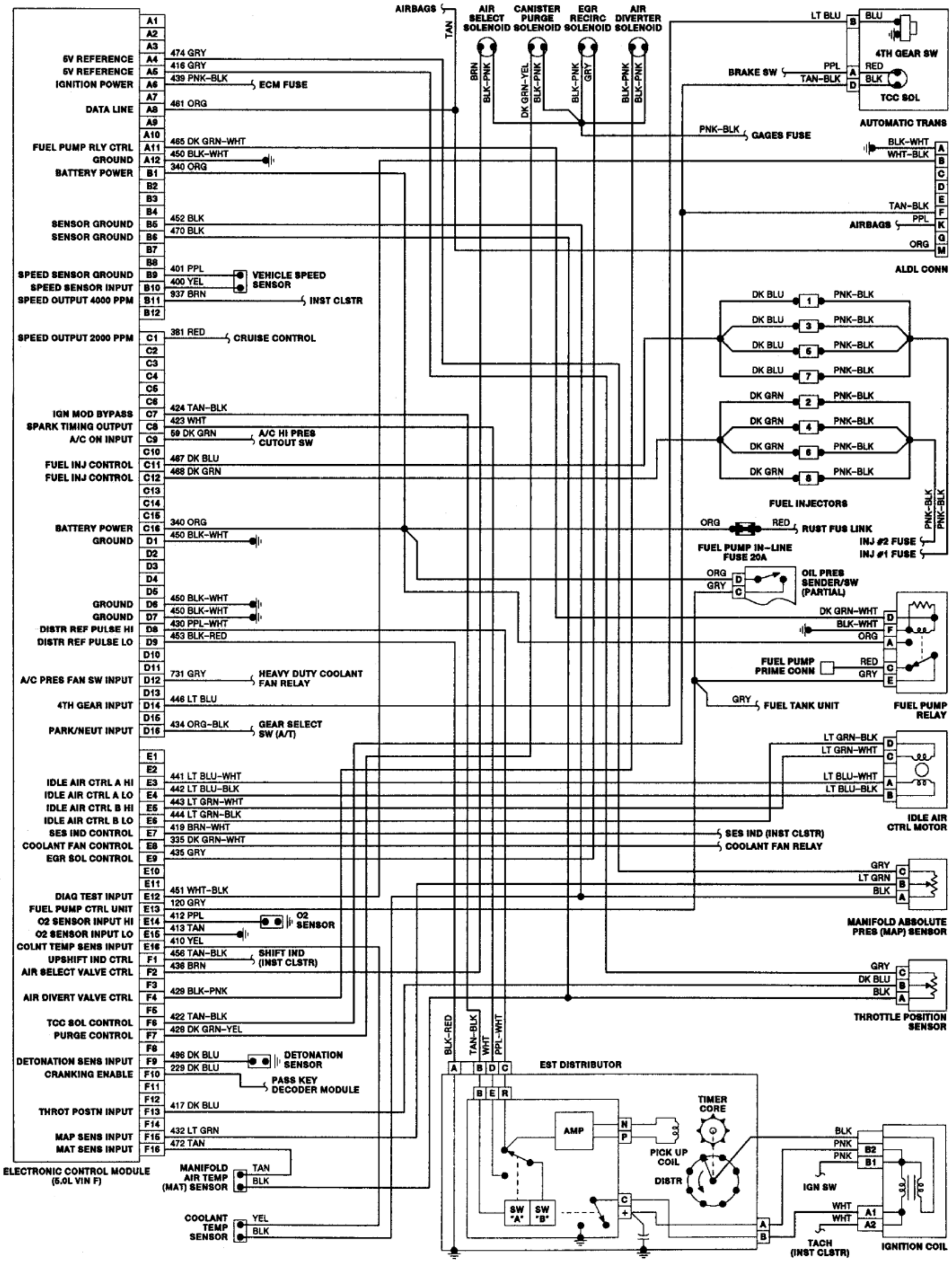
```
unsigned char CarData_ASCII[CarDataSize];
#define  DATA_CTS_2     0  //COOLANT TEMPERATURE Celsius
#define  DATA_CTS_1     1
#define  DATA_TPS_v     2  //TPS volts

#define  DATA_RPM_4     3
#define  DATA_RPM_3     4
#define  DATA_RPM_2     5
#define  DATA_RPM_1     6

#define  DATA_MPH_3     7
#define  DATA_MPH_2     8
#define  DATA_MPH_1     9

#define  DATA_BLM_3     10
#define  DATA_BLM_2     20
#define  DATA_BLM_1     21

#define  DATA_BLM_CELL  11

#define  DATA_TPS_3     12
#define  DATA_TPS_2     13
#define  DATA_TPS_1     14

#define  DATA_MAP_3     15
#define  DATA_MAP_2     16
#define  DATA_MAP_1     17

#define  DATA_BATT_2    18
#define  DATA_BATT_1    19




//ECU trouble codes lookup table
const unsigned char PROGMEM ErrorCode_Table[]=
{
   23,22,21,16,15,14,13,12,
   35,34,33,32,31,26,25,24,
   51,46,45,44,43,42,41,36,
   63,62,61,56,55,54,53,52,
   00,00,00,00,00,66,65,64
};


//Function prototypes
void LCD_Initialize(void);
void LCD_D_Write(unsigned char);
void LCD_I_Write(unsigned char);
void LCD_Delay(void);
void USART_init(void);
void Binary_Out(unsigned char);
void Get_Car_Data(void);
//void HEX_to_ASCII(unsigned int, unsigned char &, unsigned char &, unsigned char &, unsigned
char &);


// ------------------------------------------------------------------------------
// ------------------------------------------------------------------------------
// ------------------------ Program Code Starts Here ----------------------------
// ------------------------------------------------------------------------------
// ------------------------------------------------------------------------------


//******************************************************************
//* Function: LCD_Initialize                                       *
//* Initialize LCD                                                 *
//******************************************************************
void LCD_Initialize(void)
```

```
{
   Bit_Clear(E_PORT,E); // Bring enable pin low

   LCD_I_Write(0x38);   // 8-Bit data transfer mode
   LCD_I_Write(0x0C);   // Turn display on, No cursor, No blinking
   LCD_I_Write(0x01);   // Display clear, Restore to upper left position
   LCD_I_Write(0x06);   // Address counter increment after each display
   LCD_I_Write(0x02);   // Set address counter to zero, move cursor to home
}



//*************************************************************
//* Function: LCD_Write                                       *
//* Output data passed in, in data var. to LCD                *
//* - Commented out, not used anymore -                       *
//*************************************************************
/*void LCD_Write(unsigned char data, unsigned char select)
{
   Bit_Set(E_PORT,E);

   if(select==0xFF)
      Bit_Set(RS_PORT,RS);   //LCD Data mode
   else
      Bit_Clear(RS_PORT,RS); //LCD Instruction mode

   LCD_PORT=data;

   _delay_ms(1);
   Bit_Clear(E_PORT,E);
   _delay_ms(1);

}
*/

//*************************************************************
//* Function: LCD_D_Write                                     *
//* Output data passed in, in data var. to LCD                *
//*************************************************************
void LCD_D_Write(unsigned char data)
{
   Bit_Set(E_PORT,E);
   Bit_Set(RS_PORT,RS);   //LCD Data mode


   LCD_PORT=data;

   LCD_Delay();
   Bit_Clear(E_PORT,E);
   LCD_Delay();

}

//*************************************************************
//* Function: LCD_I_Write                                     *
//* Output data passed in, in data var. to LCD                *
//*************************************************************
void LCD_I_Write(unsigned char data)
{
   Bit_Set(E_PORT,E);
   Bit_Clear(RS_PORT,RS); //LCD Instruction mode

   LCD_PORT=data;

   LCD_Delay();
   Bit_Clear(E_PORT,E);
   LCD_Delay();

}
```

```
//*******************************************************************
//* Function: LCD_Delay                                             *
//* Relay function used for LCD communication                       *
//*******************************************************************
void LCD_Delay(void)
{
   _delay_ms(1);
}


//*******************************************************************
//* Function: USART_init                                            *
//* Enable and Initialize UART                                      *
//*******************************************************************
void USART_init(void)
{
    // Set the USART baudrate registers for 8192, UBRR=149
    UBRR0H = 0;
    UBRR0L = 149; // Set for 4800 for PC testing
                  //255 = 4800 baud
                  //149 = 8192 buad
                                //127 = 9600 baud


        //UCSRnA - USART Control and Status Register A
    // Enable 2x speed change, 0=16 divider 1=8 divider
    /*
          Bit 7 MSB
        R RXCn  -> USART Receive Complete
        0 TXCn  -> TXCn: USART Transmit Complete
        R UDREn -> USART Data Register Empty
        R FEn   -> Frame Error
        R DORn  -> Data OverRun
        R UPEn  -> USART Parity Error
        0 U2Xn  -> Double the USART Transmission Speed, 0=16 divider 1=8 divider
        0 MPCMn -> Multi-processor Communication Mode
          Bit 0 LSB
        */
        UCSR0A = (0<<U2X0);


    // UCSRnB - USART Control and Status Register n B
    // Enable receiver
    /*
         Bit 7 MSB
        RXCIE0 = 0 Receive Interrupt disabled
        TXCIE0 = 0 Transmit Interrupt disabled
        UDRIE0 = 0 Interupt disabled
        RXEN0  = 1 Receive enabled
        TXEN0  = 1 Transmit enabled
        UCSZ02 = 0 8-bit mode
        RXB80  = 0 9'th RX data bit when using 9-bit UART
        TXB80  = 0 9'th TX data bit when using 9-bit UART
         Bit 0 LSB
        */
        UCSR0B =(1<<RXEN0)|(1<<TXEN0);


        // UCSRnC - USART Control and Status Register n C
        // Set the USART to asynchronous at 8 bits no parity and 1 stop bits
    /*
          Bit 7 MSB
        0 UMSEL01 |
        0 UMSEL00 |-> Async
        0 UPM01  |
        0 UPM00  |-> No parity
        0 USBS0 -> 1 stop bits;
        1 UCSZ01  |
        1 UCSZ00  |-> 8-bit
        0 UCPOL0 -> Receiving data sampled on falling edge
          Bit 0 LSB
        */
        UCSR0C
=(0<<UMSEL01)|(0<<UMSEL00)|(0<<UPM01)|(0<<UPM00)|(0<<USBS0)|(1<<UCSZ01)|(1<<UCSZ00)|(0<<UCPOL
0);

}


//******************************************************************
//* Function: Binary_Out                                           *
//* Outputs any passed in variable to LCD in binary                *
//* - Used for debugging -                                         *
//* - Commented out, not used in final working code -              *
//******************************************************************
/*
void Out_Binary(unsigned char data)
{
        for(unsigned char bitn = 8 ; bitn > 0 ; bitn--)
        {
                if( data & ( 1 << (bitn-1) ) )
                        LCD_Write(0x31,0xFF); // Output 1 to LCD
                else
                        LCD_Write(0x30,0xFF); // Output 0 to LCD
        }
}
*/


//*******************************************************************
//* Function: USART_Receive                                         *
//* Waits for incoming data on USART then returns it to the calling *
//* function                                                        *
//* - Commented out, not used in final working code -               *
//*******************************************************************
/*
unsigned char USART_Receive(void)
{
        // Wait for data to be received
        while ( !(UCSR0A & (1<<RXC0)) );

        // Get and return received data from buffer
        return UDR0;
}
*/


//******************************************************************
//* Function: USART_Transmit                                       *
//* Transmit data                                                  *
//* Data stream array until array is full                          *
//******************************************************************
void USART_Transmit(unsigned char data)
{
        // Wait for empty transmit buffer
        while ( !( UCSR0A & (1<<UDRE0)) );

        // Put data into buffer, sends the data
        UDR0 = data;

        // Wait for empty transmit buffer
        while ( !( UCSR0A & (1<<UDRE0)) );
}


//******************************************************************
//* Function: Get_Car_Data                                         *
//* Waits for incoming data on USART then puts received char into  *
//* Data stream array until array is full                          *
//******************************************************************
void Get_Car_Data(void)
{
    unsigned int timeout;

    Bit_Set(PORTD,RXcontrol);  //Lockout recieve pin

    //Mode 1 command set
    USART_Transmit(0xF4);
        USART_Transmit(0x56);
        USART_Transmit(0x01);
        USART_Transmit(0xB5);

    Bit_Clear(PORTD,RXcontrol);  //Open up recieve pin

        for(unsigned char i = 0 ; i < DataStreamSize ; i++)
        {
       timeout=0;  // Reset timeout to zero

                // Wait for data to be received if data not recived in timout period
       // then exit loop
                while( !(UCSR0A & (1<<RXC0) )  && (timeout < 12000) )
       {
          timeout++;
       }

                // Get and return received data from buffer
       if(timeout == 12000)
          DataStream[i]=0;     // If timeout load 0 into datastream value
       else
                   DataStream[i]=UDR0;  // Otherwise load recieved value
        }

    Bit_Set(PORTD,RXcontrol);  //Lockout recieve pin
}


//******************************************************************
//* Function: Flash_LEDs                                           *
//* - Used for debugging -                                         *
//* - Commented out, not used in final working code -              *
//******************************************************************
/*
void Flash_LEDs()
{
    for(int i=0 ; i<20 ; i++ )
    {
       PORTA=0x00;
       _delay_ms(1000);
       PORTA=0xFF;
       _delay_ms(1000);
    }
}
*/

//******************************************************************
//* Function: HEX_to_ASCII                                         *
//* Convert inputted 8-bit HEX data to 4 ANSII digits for LCD      *
//******************************************************************
void HEXtoASCII(unsigned int Hex_Input,unsigned char *digit4, unsigned char *digit3, unsigned
char *digit2, unsigned char *digit1)
{

    // Clear varaibles
     *digit4 = 0x00;
     *digit3 = 0x00;
     *digit2 = 0x00;
     *digit1 = 0x00;

     while(Hex_Input >= 1000)
     {
         Hex_Input=Hex_Input - 1000;
         *digit4 = *digit4 + 0x01;
     }
     while(Hex_Input >= 100)
     {
         Hex_Input=Hex_Input - 100;
         *digit3 = *digit3 + 0x01;
     }

     while(Hex_Input >= 10)
     {
         Hex_Input=Hex_Input - 10;
         *digit2 = *digit2 + 0x01;
     }

     *digit1 = Hex_Input; //remainder

     // Convert to ASCII, OR with 0x30 produces ASCII
     *digit4 = *digit4 | 0x30;
     *digit3 = *digit3 | 0x30;
     *digit2 = *digit2 | 0x30;
     *digit1 = *digit1 | 0x30;
}



//*******************************************************************
//* Function: Calculate_CarData                                     *
//* Convert raw data from data stream to real numbers and put in   *
//* ASCII format                                                    *
//*******************************************************************
void Calculate_CarData()
{
    unsigned char ignore;
    unsigned int temp;

    // Calculate coolant tempature
    temp = DataStream[DEF_CTS] * 0.75 - 40;
    HEXtoASCII(temp,&ignore, &ignore, &CarData_ASCII[DATA_CTS_2], &CarData_ASCII[DATA_CTS_1]);

    // Calculate RPM
    temp = DataStream[DEF_RPM]*25;
    HEXtoASCII(temp,&CarData_ASCII[DATA_RPM_4], &CarData_ASCII[DATA_RPM_3],
&CarData_ASCII[DATA_RPM_2], &CarData_ASCII[DATA_RPM_1]);

    // Calculate MPH
    temp = DataStream[DEF_MPH] / 1;
    HEXtoASCII(temp,&ignore, &CarData_ASCII[DATA_MPH_3], &CarData_ASCII[DATA_MPH_2],
&CarData_ASCII[DATA_MPH_1]);

    // Calculate Block Learn Multiplier (BLM)
    temp = DataStream[DEF_BLM];
    HEXtoASCII(temp,&ignore, &CarData_ASCII[DATA_BLM_3], &CarData_ASCII[DATA_BLM_2],
&CarData_ASCII[DATA_BLM_1]);

    // Calculate Throttle Position Sensor (TPS) percent
    temp = DataStream[DEF_STPS] / 2.56;
    HEXtoASCII(temp,&ignore, &CarData_ASCII[DATA_TPS_3], &CarData_ASCII[DATA_TPS_2],
&CarData_ASCII[DATA_TPS_1]);

    // Calculate Manifold Air Pressure (MAP)
    temp = (DataStream[DEF_MAP] * 0.369) + 10.354;
    HEXtoASCII(temp,&ignore, &CarData_ASCII[DATA_MAP_3], &CarData_ASCII[DATA_MAP_2],
&CarData_ASCII[DATA_MAP_1]);

    // Calculate Battery voltage
    temp = DataStream[DEF_BATT] / 10;
    HEXtoASCII(temp,&ignore, &ignore, &CarData_ASCII[DATA_BATT_2],
&CarData_ASCII[DATA_BATT_1]);


}
```

```
//*******************************************************************
//* Function: Output_CarData()                                      *
//* Output calculated car data and trouble codes to LCD             *
//*******************************************************************
void Output_CarData()
{
   unsigned char ignore;
   unsigned int  ErrorCode;
   unsigned char bitn=0;
   unsigned char code_2, code_1;
   unsigned char error_bit, error_byte;

   LCD_I_Write(0x02);   // Set address counter to zero, move cursor to home

   LCD_D_Write('R');
   LCD_D_Write('P');
   LCD_D_Write('M');
   LCD_D_Write(' ');
   LCD_D_Write(CarData_ASCII[DATA_RPM_4]);
   LCD_D_Write(CarData_ASCII[DATA_RPM_3]);
   LCD_D_Write(CarData_ASCII[DATA_RPM_2]);
   LCD_D_Write(CarData_ASCII[DATA_RPM_1]);

   LCD_D_Write(' '); //Space
   LCD_D_Write(' '); //Space
   LCD_D_Write(' '); //Space

   LCD_D_Write('M');
   LCD_D_Write('P');
   LCD_D_Write('H');
   LCD_D_Write(' ');
   LCD_D_Write(CarData_ASCII[DATA_MPH_3]);
   LCD_D_Write(CarData_ASCII[DATA_MPH_2]);
   LCD_D_Write(CarData_ASCII[DATA_MPH_1]);

   LCD_D_Write(' '); //Space
   LCD_D_Write(' '); //Space


   LCD_D_Write('B');
   LCD_D_Write('L');
   LCD_D_Write('M');
   LCD_D_Write(' ');
   LCD_D_Write(CarData_ASCII[DATA_BLM_3]);
   LCD_D_Write(CarData_ASCII[DATA_BLM_2]);
   LCD_D_Write(CarData_ASCII[DATA_BLM_1]);

   LCD_D_Write(' '); //Space
   LCD_D_Write(' '); //Space
   LCD_D_Write(' '); //Space
   LCD_D_Write(' '); //Space

   LCD_D_Write('C');
   LCD_D_Write('T');
   LCD_D_Write('S');
   LCD_D_Write(' ');
   LCD_D_Write(CarData_ASCII[DATA_CTS_2]);
   LCD_D_Write(CarData_ASCII[DATA_CTS_1]);
   LCD_D_Write('C');
   LCD_D_Write(' ');


   LCD_D_Write(' '); //Space

   LCD_D_Write('M');
   LCD_D_Write('A');
   LCD_D_Write('P');
   LCD_D_Write(' ');

   LCD_D_Write(CarData_ASCII[DATA_MAP_3]);
   LCD_D_Write(CarData_ASCII[DATA_MAP_2]);
   LCD_D_Write(CarData_ASCII[DATA_MAP_1]);
   LCD_D_Write(' '); //Space
   LCD_D_Write(' '); //Space
   LCD_D_Write(' '); //Space
   LCD_D_Write(' '); //Space

   LCD_D_Write('T');
   LCD_D_Write('P');
   LCD_D_Write('S');
   LCD_D_Write(' ');
   LCD_D_Write(CarData_ASCII[DATA_TPS_3]);
   LCD_D_Write(CarData_ASCII[DATA_TPS_2]);
   LCD_D_Write(CarData_ASCII[DATA_TPS_1]);
   LCD_D_Write('%');

   LCD_D_Write(' '); //Space

   LCD_D_Write('B');
   LCD_D_Write('A');
   LCD_D_Write('T');
   LCD_D_Write('T');
   LCD_D_Write(' ');


   LCD_D_Write(CarData_ASCII[DATA_BATT_2]);
   LCD_D_Write(CarData_ASCII[DATA_BATT_1]);
   LCD_D_Write('V');
   LCD_D_Write(' ');

   LCD_D_Write(' ');
   LCD_D_Write(' ');

   error_bit=0;  //Reset error bit to zero

        for(error_byte = DEF_ERROR1 ; error_byte < DEF_ERROR5 ; error_byte++)
        {
        for(bitn = 0 ; bitn < 7 ; bitn++)
        {
         //If error bit in datastream is 1 then ouput trouble code
               if(DataStream[error_byte] & ( 1 << (bitn) ) )
         {
            ErrorCode = (unsigned char)pgm_read_byte(&ErrorCode_Table[error_bit]);

            if(ErrorCode != 0) //If error code is 0 ignore trouble code
            {
               ErrorCode = (unsigned char)pgm_read_byte(&ErrorCode_Table[error_bit]);

               HEXtoASCII(ErrorCode,&ignore, &ignore, &code_2, &code_1);
                      LCD_D_Write(code_2);
               LCD_D_Write(code_1);
               LCD_D_Write(' ');
            }
         }

         error_bit = error_bit + 1; //Increment error bit number

        }
    }

}


//******************************************************************
//* Function: main                                                 *
//* Main control function                                          *
//******************************************************************
int main (void)
{
   DDRA = 0xFF;                                 // Configure PORTA as output
   DDRC = 0xFF;                                 // Configure PORTC as output
   DDRD = (1<<PD1)|(1<<PD5)|(1<<PD6)|(1<<PD7);  // Configure PORTD

   USART_init();        //Initialize USART
   LCD_Initialize();    //Initialize LCD


   // Loop forever
   while(1)
   {
      Get_Car_Data();
      Calculate_CarData();
      Output_CarData();
   }

  while(1)
  ;

}// End of main
```

Appendix F – General Motors 1227730 ECU pin out [8]



Appendix G – Important Datasheet Pages