ECE-416: Senior Project

Automotive Diagnostic Tool

Adam Serafin
Jonathan Johnston
December 6, 2004
Senior Project: Final Report

Table of Contents

Introduction:

This report is documentation of our senior project from its beginning to the very end. Our project was first conceived in ECE-414 which is the project proposal component of the Senior Project class that is required for our Bachelors degree in Computer Engineering. Our project can be subdivided into three parts; proposal, design, and implementation. This report is organized similarly.

This project was fairly well rounded in the area of Computer Engineering. The design involved both hardware and software. Designing the required hardware demanded knowledge of computer architecture, digital logic, and integrated circuits (ICs). In addition, knowledge of basic circuit theory is a must. As for software design we programmed in both assembly language and high-level language. Because our project is based upon the Motorola 68000 Series microprocessor, we wrote our program using this assembly language. In addition, we used Visual Basic to write software that would allow our M68k single board computer to communicate with any PC.

Our project was challenging in both design and implementation. It took the entire semester to complete the basic functions as we proposed. As with most engineering projects, after successful completion we have noticed many areas that can be upgraded and improved.

Proposal: Abstract

The first automobiles relied on mechanical systems to operate the engine. This proved good for the times, but the evolution of the computer would make a big change. Modern automobile fuel and ignition systems are almost entirely computer controlled. The ECU(Electronic Control Unit) of an automobile is responsible for making the precise calculations that allow the proper amounts fuel to be injected at the proper time. The ECU also determines exactly when to fire each spark plug taking into account the ignition spark advance and retard under all engine operating conditions. The ECU obtains all of its data through various electrical sensors. When an engine component fails or one of the sensors is not working properly, the engine will not function efficiently or sometimes, not at all. It is difficult to determine which sensor or mechanical system is not functioning correctly because the vehicle operator does not have access to a unit that can display sensor output. Our proposal is to make this available.

Proposal: Background

Since our project emphasizes automobile engine diagnostics, we will monitor the sensors that allow the operator or technician to completely understand the common problems. Examples and information of the sensors we'd like to monitor are as follows:

Intake Air Temperature(IAT): Changes its resistance with respect to the ambient air temperature. The temperature of the air is related to how dense it is. As air temperature decreases, density and oxygen content increases. The output of this sensor is important for diagnosing trouble with the intake system, and a malfunctioning of this sensor would result in a less efficient operation of the engine. The output that we would like to obtain is ambient air temperature.

Throttle Position Sensor(TPS): The TPS essentially is a potentiometer that is controlled by the throttle (gas pedal). This sensor is located on the throttle body. Throttle position is one of the key elements of fuel injection control. A problematic TPS sensor can result in loss of engine power. The output that we would like to obtain is throttle position in degrees.

Manifold Absolute Pressure(MAP): The MAP sensor varies its voltage with respect to the air pressure inside the engines intake manifold. The MAP sensor accurately senses the amount of vacuum in the manifold. Manifold vacuum is a good indicator of engine load. A malfunctioning MAP sensor can result in erratic engine idling, loss of power, less efficient operation, etc. The type of output we would like to obtain is engine vacuum in inches mercury (unit of vacuum) the range is 30 inches mercury to 0.

EGR System: EGR which stands for Exhaust Gas Recirculation is a system that was implemented in the 1980's to produce cleaner vehicle emissions. The theory behind this is that engineers have noticed when inert gas is introduced into the combustion chamber of the engine, NOX emissions are reduced. However, there is one problem, introducing an inert gas from some outside source of the vehicle would be impractical. The solution to this problem was to use exhaust gas as the inert gas. This was an ideal solution because the oxygen and fuel of exhaust is almost entirely spent.

The effectiveness of the EGR system is achieved by carefully metering exhaust gas into the intake manifold through a variable valve. This valve is vacuum controlled by a diaphragm which in tern is controlled by a solenoid actuator. To ensure that the EGR system is working properly, modern manufacturers employ a EGR Valve Lift sensor. The EGR lift sensor is used to send an analog signal representing valve lift to the ECU. The ECU then compares this actual valve lift to theoretical valve lift. If the two values do not agree then the system is not working properly which can result in a "Check Engine" light, poor running conditions, and finally increased emissions. Because this system is so important to the clean operation of an engine, and common engine trouble, a read-out of this sensor information would be extremely valuable to a technician.

Coolant Temperature Sensors: The coolant temperature sensors used in modern vehicles are what are known as thermistors. This means that it varies its resistance with respect to temperature. Coolant temperature sensors are of great diagnostic value because they can detect cooling system problems. Coolant temperature sensors also allow the computer to adjust its fuel map according to the temperature. The output we would like to obtain is coolant temperature.

Proposal: Preliminary Design

In our project design we will use the 0808 Analog to Digital converter. This chip contains 8 converters which will allow us to convert the analog sensor outputs into the binary data we need to complete the calculations. We will also need the Motorola 68k Single Board Computer. The M68k SBC is a Single Board Computer that we have constructed together previously. The price to build one is around $150. We will get an LCD display (about $30) and connect it to the SBC to have a better and easier to read display screen. The M68k will provide sufficient processing power to provide a real-time display of sensor information. We will have to program the M68k in assembly language. The programming is crucial because it will contain the algorithms that will obtain the proper conversion data in a lookup table that will be stored in the SBC's ROM. Our interface will also allow the user to toggle through the different senor outputs. We are thinking of also making the interface toggle according to a set timer. Lastly, we want to take the serial port of our SBC and connect it to a PC. This will allow a user to do further, more powerful analysis of the sensors and create a more user friendly, GUI interface. This part of our project entails knowledge of the C++ programming language and how the serial port of the SBC transfers information to the PC.

As of now the SBC is dependant on a 120VAC source. We would like to create a power supply for it that will take advantage of the automobiles 12-13.5VDC system conveniently available through the cigarette lighter. This will make our project completely portable, unlike the expensive diagnostic equipment. This offers an advantage of diagnosing problems when the engine is in highway or city conditions.

The first step in designing our project is obtaining the desired senor outputs that we want through the engine's electrical system. To simulate analog sensor outputs we will use 100k ohm potentiometers that will connect to the 8 analog inputs on the analog to digital converter. The next phase of project design will be the configuration of the analog to digital converter within our M68k SBC. This configuration will be somewhat complex because we want the user to be able to toggle through all of the different sensor outputs. This design of this will probably use a 3-8 Decoder and a 4-1 Multiplexer. The next step of the design is writing the code in assembly language that will convert the binary data obtained from the analog to digital converter. The conversion of this binary data is needed in order to obtain values corresponding to what the sensor is sensing. We will use a lookup table in the ROM of the SBC that will correspond binary values with actual data. Once this data is obtained, we will output it to an LCD display. This will allow the user to know what the sensors are outputting.

The final phase of our project design would be to interface the SBC to a PC. This will involve programming in both C++ and assembly language.

Motorola 68k Single Board Computer: Why the M68k?

To implement our project, a microprocessor is necessary to receive, process, and output data in a carefully structured manner. Because we studied the design of the Motorola 68k, and built an SBC around one in two of our undergraduate courses; using the M68k SBC in our project was a logical step.

Our SBC has many desirable features for this project, they include: serial I/O, RAM, EEPROM, parallel I/O, room for expansion, and a basic operating system. Serial I/O is necessary for our project because of the need to transmit/receive data with a PC.

The RAM would allow us to store programs on the SBC that were downloaded via the serial port. This is very important for software testing; imagine re-burning an EEPROM every time you made a change to your software? The EEPROM would serve as the storage for the final version of our software. Parallel Input is necessary, and was used to implement the sensor scroll feature of our project. Parallel Output was used to output data to our Parallel LCD display and an LED bar display. Room for expansion was a major benefit since we would have to add additional I/O to our system. The use of a 3-8 Decoder for the addressing logic in the design of the SBC allows us to add up-to 4 additional I/O devices with relative ease. This will come in handy for the sensor scroll feature and addressing the Analog to Digital Converter.

The final and perhaps greatest asset of the M68k SBC was the monitor program designed by Professor Rosenstark. The monitor program when stored on the system’s EEPROM allows you to download programs to the SBC serially. This is a great way to debug software on the actual system which is necessary to accomplish things an emulator cannot possibly handle. An example of this is the need to run programs on the SBC to test and debug the LCD display, or the Analog to Digital Converter.

Motorola 68k Single Board Computer: Software Details

In a class called ECE 252 we studied the instruction set architecture of the Motorola 68000 series microprocessor. This class taught us to write and emulate assembly level programs using two programs called asm68k.exe and emu68k.exe. Both of these programs are written by the author of the textbook used in ECE 252: ***The 68000 Microprocessor: Hardware and Software Principles and Applications Fourth Edition,*** **James L. Antonakos.**

Asm68k.exe is an assembler, this means it is a program that converts assembly op-code into machine code. Machine code is the hexadecimal equivalent of assembly op-code that is stored into the memory system of the SBC and is executed by the processor. In addition, asm68k.exe had advanced features that can pick up syntax errors, and create a .lst file that show addressing and disassembly data of the assembly op-code. A .lst file can be opened with any text editor, namely notepad.exe which is available on every Windows PC.

Emu68k.exe is an emulator that allows the programmer to execute programs that were designed to run on the Motorola 68k on an everyday PC instead. Emu68k.exe was extremely valuable in our software design because it allowed us to work efficiently. This holds true because it is much quicker to modify and test programs using the emulator then it is to use the SBC. One could test and debug programs on any PC without having to carry the SBC everywhere, and it is quicker to test the program in an emulator using a DOS prompt then it is to download a program to the SBC with a serial communication program. While the emulator may be more convenient, it still cannot replace the hardware of the SBC because quite a few things needed to be tested with I/O that would be difficult or impossible to emulate in some cases.

Shown below is a simple program written in Motorola 68k Series Assembly language to give you a feel on how this is done. The semicolon ';' is used for comments.

```
            Org         $8000       ;start at first location in RAM
Prog        move.b      #$FF,d0     ;moves byte $FF into data register
            Trap        #9          ;interrupt request
            End         prog        ;ends program
```

This program should be written in notepad because it is free of any text formatting characters unseen to the user. This program must be saved with the file extension of '.asm' in order to be used by the assembler. The first step in running this program is converting the opcode into machine code using asm68k.exe. To do this you must obtain this program at http://www.sunybroome.edu/~antonakos_j/68ktoc.html . Place this program into a directory. Use a DOS prompt to access the directory and type "asm68k filename". The assembler will then assemble the file and create a .hex file that contains the hex code needed. In order to run the program using the assembler you must obtain emu68k.exe from the address above, put this file in a directory, access this directory in DOS, and type emu68k filename. The emulator will load, at this point type ‘g’. The program will then execute.

To run this program on the SBC itself you must use serial communication software such as “hyperterminal”. Open this program set these parameters: baud rate to 38400 bps, no stop bits, no parity bits, and no hardware. Once these parameters have been set the software will now be in communication with the SBC’s monitor program. Now reset the SBC using the reset button, type ‘L’ at the prompt, from the menu select “send text file,” select filename.hex of the program you want to run and click “send.” The hex code will transmit, after this is done press enter and type in ‘g’ to run the program. The program has now been executed

Now that the basics such as writing a program in notepad, assembling, and executing the program in an emulator and on the SBC have been covered, you will have an easier time understanding the assembly software information detailed in this report.

Motorola 68k Single Board Computer: Hardware Details

The ECE 252 and ECE 395 supplementary manuals at the URL: web.njit.edu/~rosensta provides in depth documentation of the hardware details of the Motorola 68k SBC. Below are the basic principles of the organization and architecture of the SBC that are required to understand our project design.

The addressing logic generated from the high order address pins of the CPU is the heart of understanding the function of the SBC. To simplify matters the SBC was designed using a 3-8 decoder to generate the addressing logic. The 3 high order address bits are input into the decoder which allow up to 8 unique outputs. In technical terms this is known as high-order interleaving. When studied, the schematic illustrates the principle of addressing, for example when the three high order address bits are 000 the first output of the decoder is selected, this output when enables the EEPROM chip. Therefore the first location of EEPROM is at $00000h. Now consider the second output on the decoder if you trace the schematic it leads to the Chip Enable CE pin of the RAM chip, this output is asserted when the 3 high order address bits are 001 now illustrated with the other address pins starting from A17-A0:

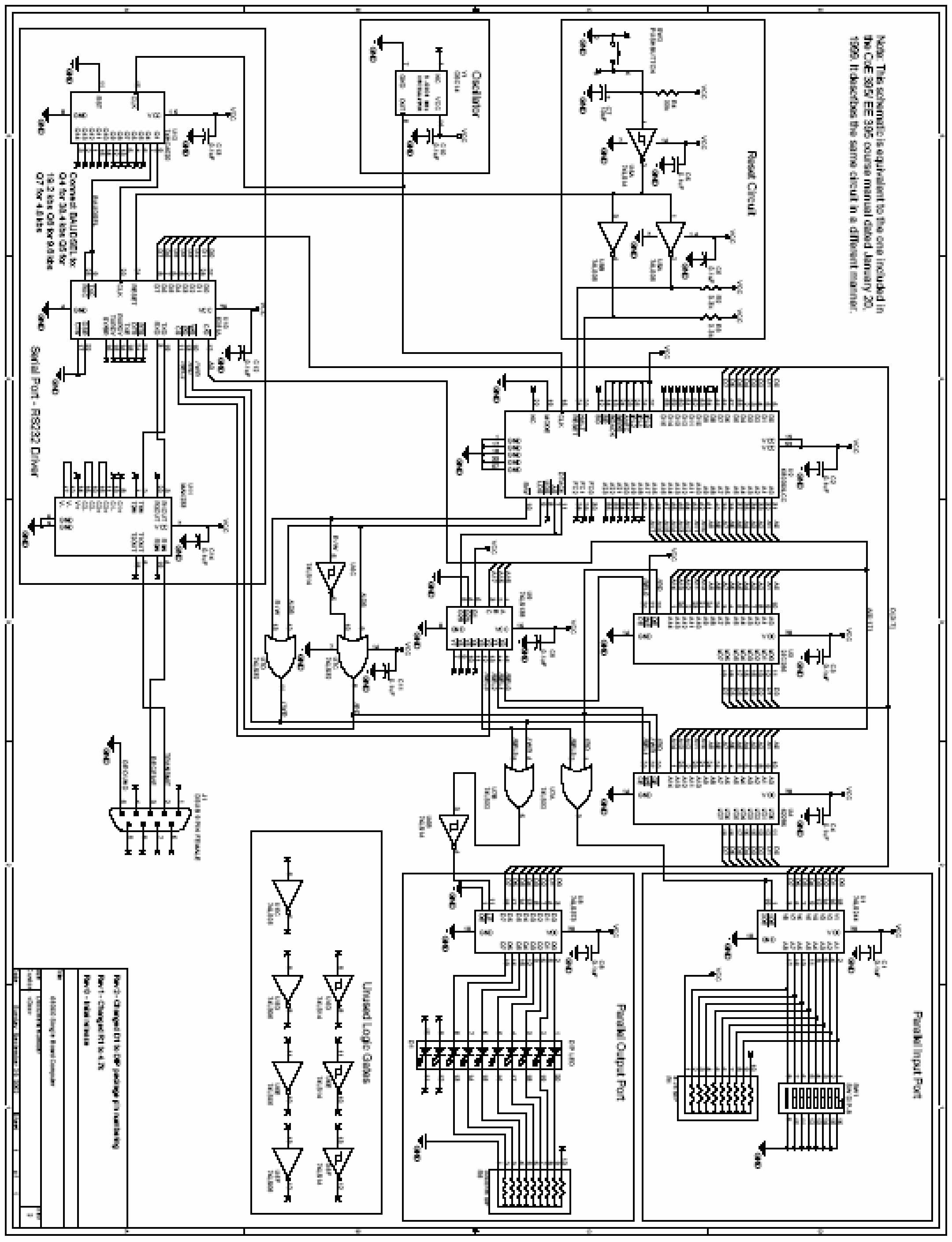
00(**1**000)(0000)(0000)(0000)
=$8000h

Therefore the starting address of the RAM is $8000h. Using this method you will arrive with $10000h as the starting address of the Serial Port, $18000h as the starting address of the Parallel Input/Output port. You will notice that sel4 and sel5 are not used but when considered, these addresses will be $20000h and $28000h respectively. This is important to note because these two addresses will be used for our project.

Other interesting hardware details of the M68k SBC are that it has an 8 bit data bus, a system clock speed of 9.8304 MHz, a binary ripple counter that divides the system clock into lower frequencies for the operation of the serial port interface, and several unused logic gates. Another point worth mentioning is the reset circuit design. This design uses a SPST momentary push-button switch de-bounced using a 330ms RC time constant and Schmitt-trigger inverter. By inherent design the Schmitt-trigger inverter ensures a cleaner more “square” pulse.

The aforementioned hardware features of the SBC are necessary to understand the implementation of our hardware design. Additional, more advanced documentation of the M68k SBC design can be found at Professor Rosenstark’s website (listed above)

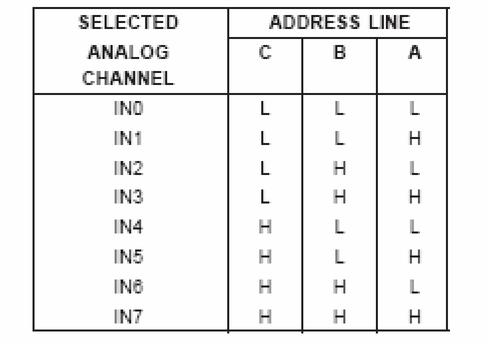
Motorola 68k SBC Schematics:



Hardware: Analog to Digital Conversion

Almost all physical occurrences are analog. This means that they are continuously variable, or measurable. Examples are length, width, voltage, pressure, temperature etc. This applies to an engine's operating conditions as well. The ECU, as explained earlier has many decisions to make in order to successfully fire the spark plugs and fuel injectors. The ECU bases these decisions on its inputs, much like humans base their decisions on their inputs, i.e. eyes, ears, taste, touch, and smell. In order for an ECU to make decisions based on the inputs it must process the information. For an ECU to process this analog data, it must first be converted to a binary value. Our project follows these same guidelines.

In order to make the conversion, we will use an 8 bit Analog to Digital converter, namely the ADC0808. The ADC0808 can access up to 8 different analog signals via three address pins and one address latch enable pin. Once the address has been set on the three address pins the address latch enable pin must be pulsed high in order to "latch in" the new address. It is this process that selects 1 of up to 8 analog inputs.



SELECTED ANALOG CHANNEL	ADDRESS LINE		
	C	B	A
IN0	L	L	L
IN1	L	L	H
IN2	L	H	L
IN3	L	H	H
IN4	H	L	L
IN5	H	L	H
IN6	H	H	L
IN7	H	H	H

To integrate the converter into the SBC's system architecture several steps must be taken. The first step is to wire the 8 data pins into the data bus on the SBC. Next, for continuous conversion START and EOC on the converter must be wired together. Vcc and Vref+ must be wired to +5V, by contrast Vref- and Gnd pins are wired to ground. The address latch enable is supplied by the output of the Schmitt-trigger inverter used by the SBC's reset circuit. This means that ALE will be pulsed whenever the reset button is depressed, the reason for this will be explained in the Sensor Scroll section of this report. The three address pins on the converter will be wired to the output of a latch we added to the system. The explanation for this is in the Sensor Scroll section as well.

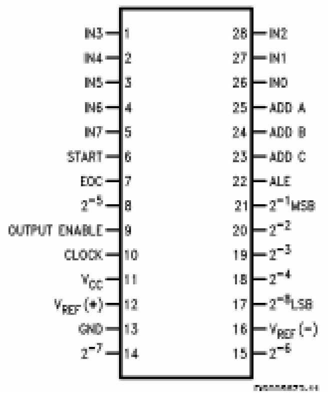
The ADC0808 requires a clock input in order to do conversions on analog data. We originally supplied this clock signal from the SBC's existing binary ripple counter. However, the frequency was much too high and resulted in data that seemed very unstable. To remedy this we obtained another binary ripple counter and used the MSB on the original counter to drive the CLK input on the next counter. By doing this we effectively obtained a slower, more reasonable frequency (2.4kHz) to supply our converter with.

Finally the last step of successfully integrating the ADC0808 into the SBC is generating the output enable signal. Because the system architecture of the SBC uses memory mapped I/O we must enable this chip using addressing logic. To do this we will use the fifth output of the 3-8 decoder located on the SBC. Because the outputs of the decoder are active low we must invert this signal using an un-used Schmitt-trigger inverter (74LS14). The output of the inverter can now be input to the "output enable" pin on the ADC0808. The address of the ADC0808 is now $20000h. To illustrate this below is a simple program that reads from the converter and writes to the output port ($18000h)

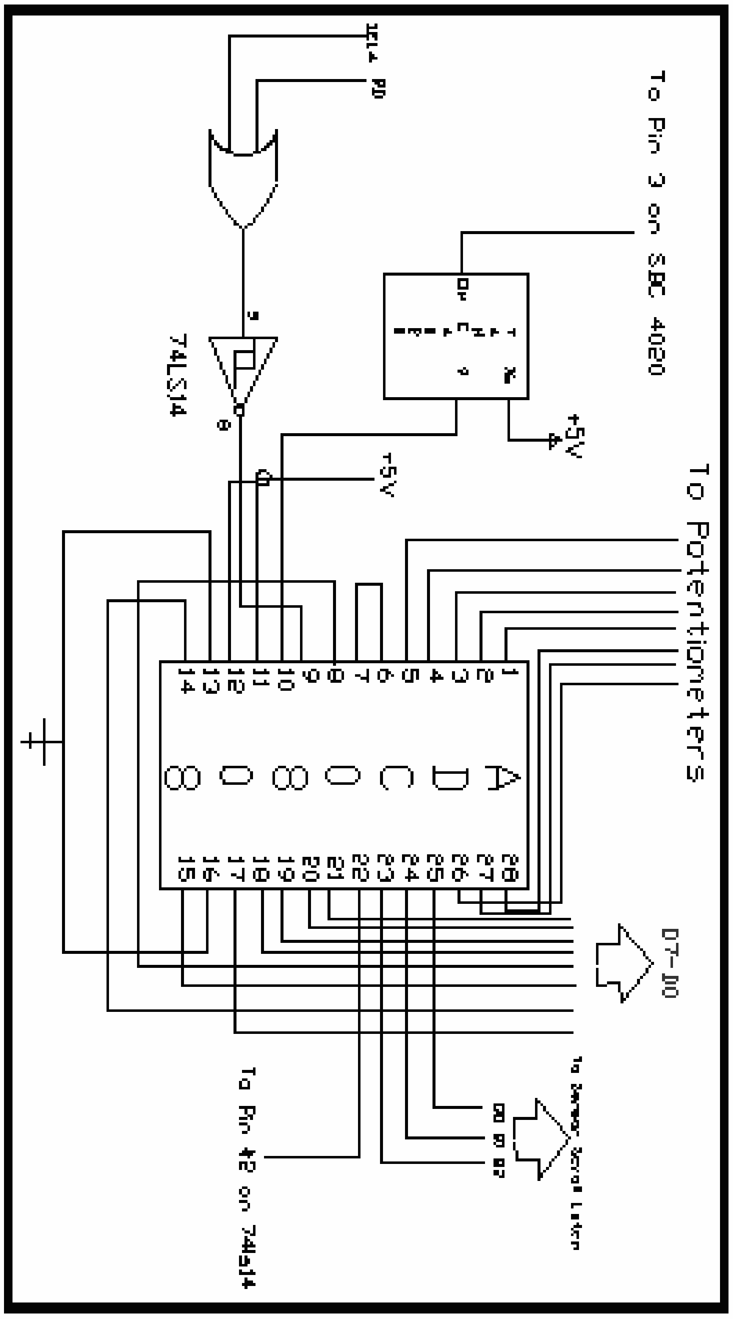
```
          Org       $8000
Prog      move.b    #$20000,d0 ;move from converter to d0
          move.b    d0,$18000 ;move from d0 to output port
          bra       prog      ;branch always to 'prog'
          trap      #9
          end       prog
```

The above program is an infinite loop that continuously reads from the converter and outputs to the output port.

ADC0808 Pinouts:



Analog to Digital Conversion: Schematics



Hardware: Sensor Scroll Feature

The sensor scroll feature allows the user to toggle through different sensor outputs by the push of a button. The button used to accomplish this is the reset button. As described in the hardware description section the reset button on the SBC is a SPST momentary pushbutton switch. The reason why we use the reset button to toggle through states is because its simple, shortens the amount of code need to be written, and acts as an interrupt.

To implement this design a storage device is needed. For this purpose we used a d-type latch. We wired three input pins of the latch to the data bus of the SBC, the three output pins are wired to the input port of the SBC, and ADD A, ADD B, and ADD C in parallel. The CLK signal of the latch is generated by the addressing logic of the SBC. We used the sixth output of the 3-8 decoder. As with the converter, we needed to invert this signal using an unused Schmitt-trigger inverter. The output of this inverter is wired to the CLK pin on the latch. The latch is addressed at $28000h. See the schematic for an illustrated view.

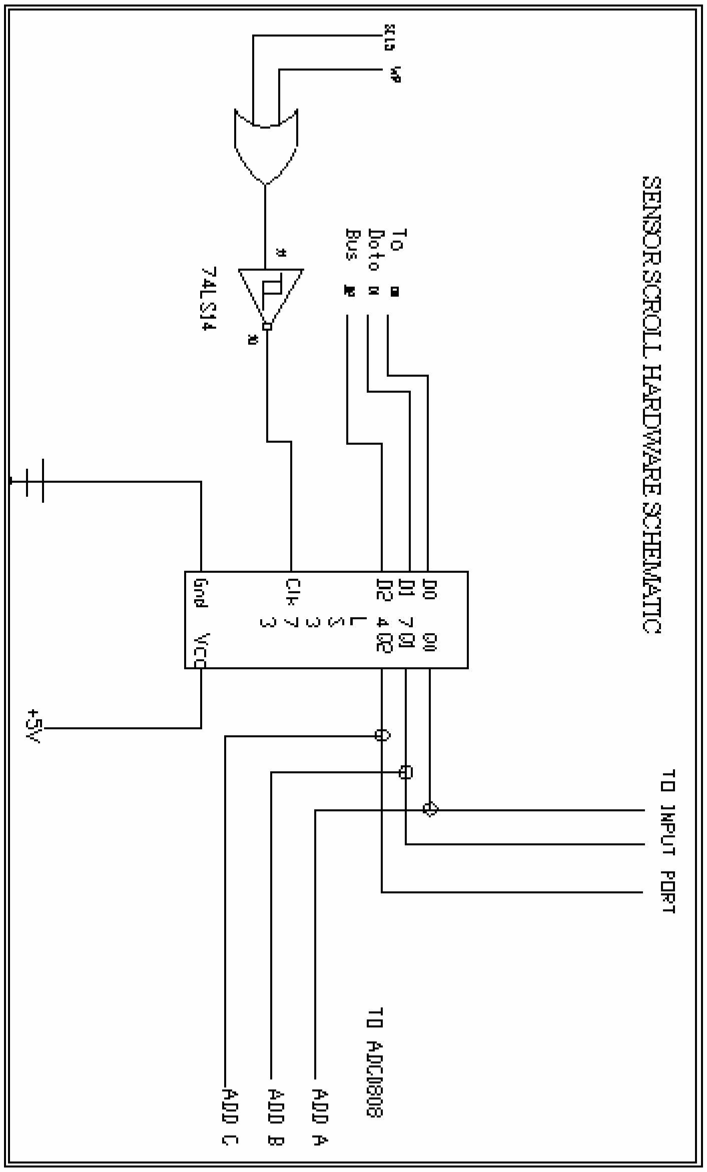
The reason why we used a latch is that once the project is powered up the latch will contain $00 which means that the converter will be reading from the 1st analog sensor input. In our main program we read from address $18000h because this address contains the value of the latch. We then store this value into a register, add 1 to this value and write the result back into the latch. The code to implement this is shown below:

```
Move.b    #$18000,d0     ;reads from input port
Addi.b    #1,d0          ;adds 1 to the value
Move.b    d0,$28000 ;writes value to latch
```

When the 3rd line of the above code is executed the latch contains the previous value+1 this means that the address pins on the converter also contain the same value. However, the present value of the address pins on the converter are not being used until the Address Latch Enable pin is pulsed. Therefore once the reset switch is depressed the present value will be latched into the converter. Now the process will repeat itself in a way that every time the reset switch is pressed, the next converter input will be read.

The sensor scroll feature is very important to the usefulness of our project and takes advantage of the converter's ability to read from 8 different sensors. While functional, it is also simple in that it is activated by the push of a single button.

Sensor Scroll Hardware Schematic:



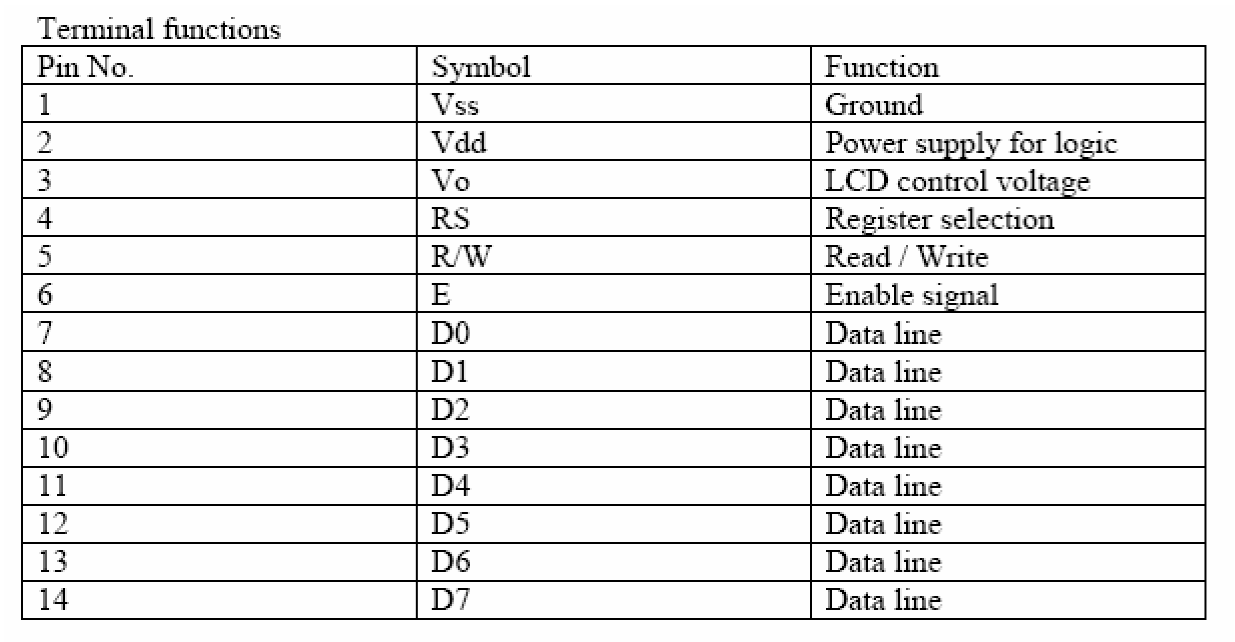
Hardware: DC Power Source

The SBC was initially powered by a 120VAC source that was stepped down, rectified, and cleaned to +5VDC. This was done using a standard 120VAC in +5VDC out power supply. Because the average car does not supply 120VAC and we want our project to be completely portable, we had to find a way to connect into an automobiles DC voltage system. To do this we simply used a ATA6405 voltage regulator IC. The reason why we chose this IC is because of its low power consumption, use for automotive applications, and thermal overload shutdown. This IC is vital to regulate the unstable 12-14 VDC supplied by an automobile's electrical system. For ease of use, our project can be plugged into a vehicle's cigarette lighter outlet.

This feature is extremely important because it allows the user to diagnose/monitor engine operating conditions on the road. This is advantageous because most problems do not reoccur in the confines of a garage bay. Furthermore this allows our design to be completely portable.

Hardware: 20x1 LCD Line Display

The LCD we have decided to use is a parallel 20x1 line LCD. There are only fourteen pins needed for the LCD. The Hitachi HD44780U controller chip is integrated onto the LCD chip. The pin numbers and functions are as follows:



Terminal functions

Pin No.	Symbol	Function
1	Vss	Ground
2	Vdd	Power supply for logic
3	Vo	LCD control voltage
4	RS	Register selection
5	R/W	Read / Write
6	E	Enable signal
7	D0	Data line
8	D1	Data line
9	D2	Data line
10	D3	Data line
11	D4	Data line
12	D5	Data line
13	D6	Data line
14	D7	Data line

We connected pin 3 to a potentiometer to create a contrast knob for the LCD screen. Pin 5 goes to ground because we are only writing to the screen, never reading. Therefore we constructed our assembly program so we only write to the LCD and never read from it. Pin 4 is connected to address line 2. This pin toggles between the write function of characters to the LCD and the operation functions of the LCD screen. If the pin is high, the LCD is in character writing mode and operation writing mode if low. The address of writing mode is $18002, the address of operation writing mode is $18000. Pin 6 is connected to the enable pin on the 74LS373 chip on the SBC. This pin is pulsed to enable the LCD.

The Hitachi chip on the LCD board allows 5x8 or 5x10 sized characters. It also allows an 8-bit operation, 8-digit x 1 line display, a 4-bit operation, 8-digit x 1 line display, and an 8-bit operation, 8-digit x 2 line display. Because the Hitachi has these choices, we needed to do a function set to tell the Hitachi chip what type of display we are using.

Instruction	Code RS	 R/$\overline{W}$	 DB7	 DB6	 DB5	 DB4	 DB3	 DB2	 DB1	 DB0	Description	Execution Time (max) (when f_{cp} or f_{OSC} is 270 kHz)
Write data to CG or DDRAM	1	0	Write data								Writes data into DDRAM or CGRAM.	37 µs t_{ADD} = 4 µs*
Read data from CG or DDRAM	1	1	Read data								Reads data from DDRAM or CGRAM.	37 µs t_{ADD} = 4 µs*
	I/D = 1: I/D = 0: S = 1: S/C = 1: S/C = 0: R/L = 1: R/L = 0: DL = 1: N = 1: F = 1: BF = 1: BF = 0:		Increment Decrement Accompanies display shift Display shift Cursor move Shift to the right Shift to the left 8 bits, DL = 0: 4 bits 2 lines, N = 0: 1 line 5 × 10 dots, F = 0: 5 × 8 dots Internally operating Instructions acceptable								DDRAM: Display data RAM CGRAM: Character generator RAM ACG: CGRAM address ADD: DDRAM address (corresponds to cursor address) AC: Address counter used for both DD and CGRAM addresses	Execution time changes when frequency changes Example: When f_{cp} or f_{OSC} is 250 kHz, $37\ \mu s \times \frac{270}{250} = 40\ \mu s$

Note: — indicates no effect.

* After execution of the CGRAM/DDRAM data write or read instruction, the RAM address counter is incremented or decremented by 1. The RAM address counter is updated after the busy flag turns off. In Figure 10, t_{ADD} is the time elapsed after the busy flag turns off until the address counter is updated.

Clear display

	RS	R/$\overline{W}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Code	0	0	0	0	0	0	0	0	0	1

Return home

	RS	R/$\overline{W}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Code	0	0	0	0	0	0	0	0	1	*

Note: * Don't care.

Entry mode set

	RS	R/$\overline{W}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Code	0	0	0	0	0	0	0	1	I/D	S

Display on/off control

	RS	R/$\overline{W}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Code	0	0	0	0	0	0	1	D	C	B

Cursor or display shift

	RS	R/$\overline{W}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Code	0	0	0	0	0	1	S/C	R/L	*	*

Note: * Don't care.

Function set

	RS	R/$\overline{W}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Code	0	0	0	0	1	DL	N	F	*	*

Set CGRAM address

	RS	R/$\overline{W}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Code	0	0	0	1	A	A	A	A	A	A

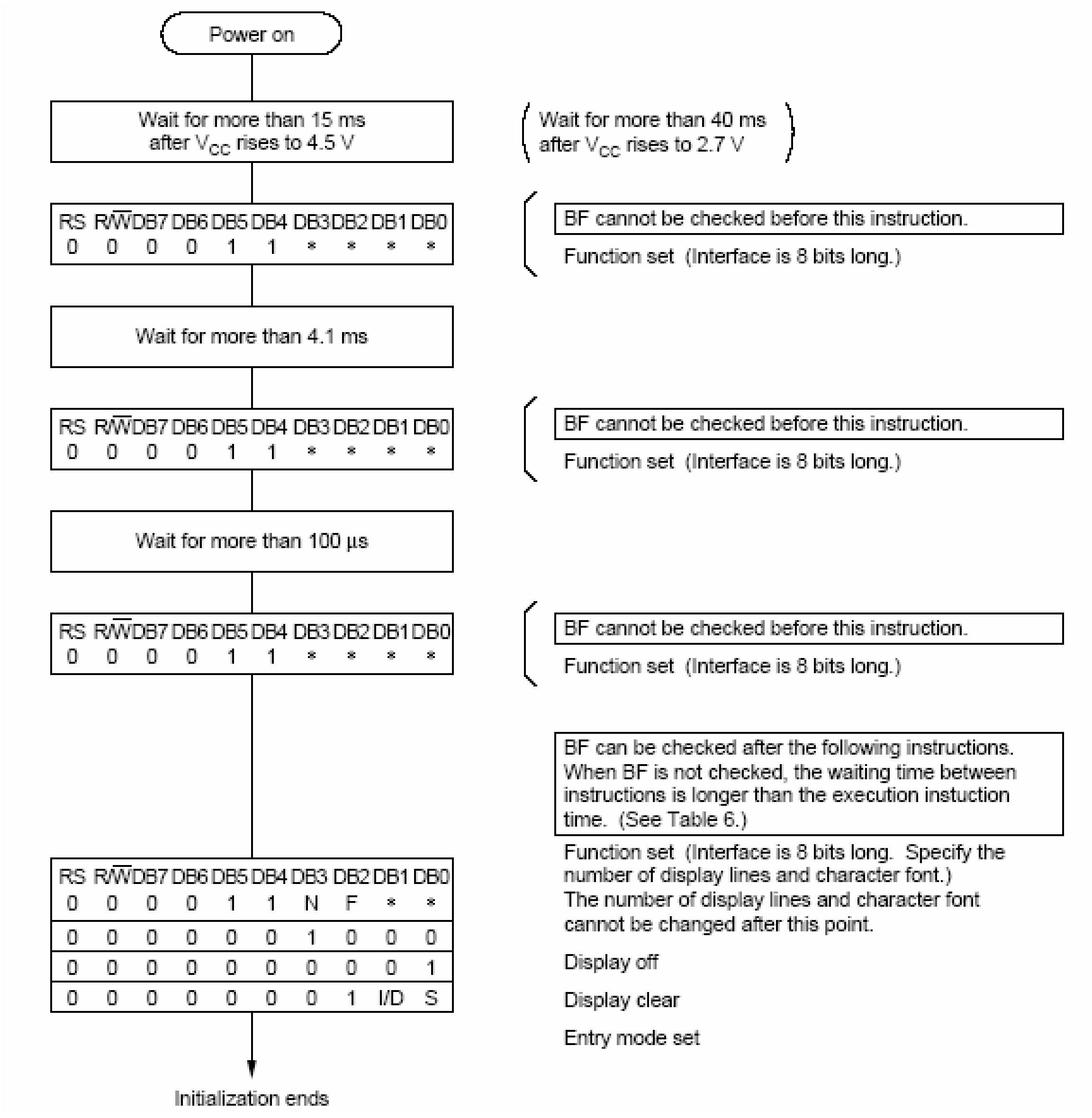
$\longleftarrow$ Higher order bit Lower order bit $\longrightarrow$

Once again the LCD is addressed to 18000 on the board. Since the RS pin is connected to address 2, at address 18002, the RS pin is high and at 18000, the RS pin is low. We decided to not display the cursor because of the rapid change in position. Since we decided to ground the R/W pin, the SBC cannot read the busy flag of the LCD. The SBC operates at a system clock many times higher then that of the LCD circuitry. This can cause problems in the character writing and operation performing because the SBC may go to the next instruction before the LCD is finished with the instruction. To fix this problem, we wrote a delay function in our program.

```
delay     move.b       #6,d1        ;This moves the number 6 to d1
delay1    move.b       d0,(a0)      ;This moves d0 to address a0
          subq.b       #1,d1        ;This subtracts 1 from d1
          bne          delay1       ;If d1 ≠ 0 then it goes to delay1
          rts                       ;This returns back to where it
jumped from
```

The LCD is initialized when the power is turned on; however, we are using the reset button as our toggle button so we need to manually initialize the LCD. To do this, we followed this flow chart and programmed accordingly. We also used this initialization sequence in the beginning of our program.

```
init      movea.l      #$18000,a0   ;sets the RS to 0, puts address in
a0
          movea.l      #$20000,a1   ;sets the potentiometer to a1
          move.b       #4,d2        ;moves number 4 to d2
init2     move.b       #$30,d0      ;moves HEX 30 to d0 (sets to 8-
bit,1 line,
                                    ;5x8 dot)
          bsr          delay        ;jumps to the delay
          subq.b       #1,d2        ;subtracts one from d2
          bne          init2        ;if d2 isn't 0 then it goes back
          move.b       #$01, d0     ;moves HEX 01 to d0 (clears the
screen)
          bsr          delay        ;jumps to the delay
          bsr          delay2       ;does an extra delay because
clearing takes
                                    ;longer
          move.b       #$0E,d0      ;moves HEX 0E to d0 (sets cursor
shift,
                                    ;decrement)
          bsr          delay        ;jumps to delay
          move.b       #$0C,d0      ;moves HEX 0C to d0 (turns cursor
off)
          bsr          delay        ;jumps to delay
          bra          read1        ;jumps to start of program
delay2    move.w       #$FFFF,d1    ;moves HEX word FFFF to d1
          move.w       #$FFFF,d2    ;moves HEX word FFFF to d2
delay3    subq.w       #1,d2        ;subtract 1 from d2
          bne          delay3       ;if d2 isn't 0 goes to delay3
delay4    subq.w       #1,d1        ;subtracts 1from d1
          bne          delay4       ;if d1 isn't 0 goes to delay4
          rts                       ;returns to jumped spot
```


Power on
Wait for more than 15 ms after V_{CC} rises to 4.5 V
Wait for more than 40 ms after V_{CC} rises to 2.7 V
RS R/W̄ DB7 DB6 DB5 DB4 DB3 DB2 DB1 DB0
0 0 0 0 1 1 * * * *
BF cannot be checked before this instruction.
Function set (Interface is 8 bits long.)
Wait for more than 4.1 ms
RS R/W̄ DB7 DB6 DB5 DB4 DB3 DB2 DB1 DB0
0 0 0 0 1 1 * * * *
BF cannot be checked before this instruction.
Function set (Interface is 8 bits long.)
Wait for more than 100 µs
RS R/W̄ DB7 DB6 DB5 DB4 DB3 DB2 DB1 DB0
0 0 0 0 1 1 * * * *
BF cannot be checked before this instruction.
Function set (Interface is 8 bits long.)
BF can be checked after the following instructions. When BF is not checked, the waiting time between instructions is longer than the execution instuction time. (See Table 6.)
RS R/W̄ DB7 DB6 DB5 DB4 DB3 DB2 DB1 DB0
0 0 0 0 1 1 N F * *
Function set (Interface is 8 bits long. Specify the number of display lines and character font.) The number of display lines and character font cannot be changed after this point.
0 0 0 0 0 0 1 0 0 0
Display off
0 0 0 0 0 0 0 0 0 1
Display clear
0 0 0 0 0 0 0 1 I/D S
Entry mode set
Initialization ends

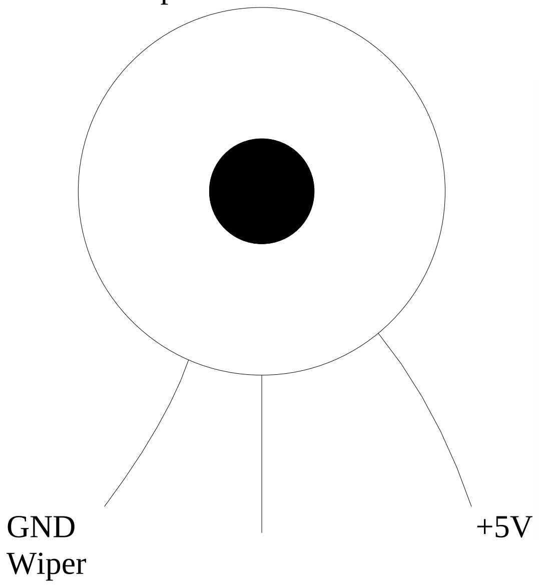
The character chart of the Hitachi is very similar to the ASCII chart. The HEX values for numbers and letters are exactly the same. The HEX for symbols may be different. Since the HEX values for numbers are the same for Hitachi and ASCII, we were able to use an algorithm to convert the 8-bit data into the HEX value needed to display a number on the screen. The Hitachi character chart is shown below:

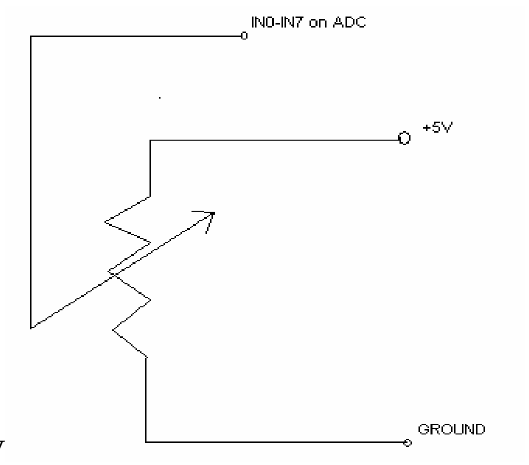
Upper 4 Bits / Lower 4 Bits	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
xxxx0000	CG RAM (1)			0	@	P	`	p				ー	タ	ミ	α	p
xxxx0001	(2)		!	1	A	Q	a	q			。	ア	チ	ム	ä	q
xxxx0010	(3)		"	2	B	R	b	r			「	イ	ツ	メ	β	θ
xxxx0011	(4)		#	3	C	S	c	s			」	ウ	テ	モ	ε	∞
xxxx0100	(5)		$	4	D	T	d	t			、	エ	ト	ヤ	μ	Ω
xxxx0101	(6)		%	5	E	U	e	u			・	オ	ナ	ユ	σ	ü
xxxx0110	(7)		&	6	F	V	f	v			ヲ	カ	ニ	ヨ	ρ	Σ
xxxx0111	(8)		'	7	G	W	g	w			ァ	キ	ヌ	ラ	g	π
xxxx1000	(1)		(	8	H	X	h	x			ィ	ク	ネ	リ	√	x̄
xxxx1001	(2)		)	9	I	Y	i	y			ゥ	ケ	ノ	ル	$^{-1}$	y
xxxx1010	(3)		*	:	J	Z	j	z			ェ	コ	ハ	レ	j	千
xxxx1011	(4)		+	;	K	[	k	{			ォ	サ	ヒ	ロ	^{x}	万
xxxx1100	(5)		,	<	L	¥	l	|			ャ	シ	フ	ワ	¢	円
xxxx1101	(6)		-	=	M	]	m	}			ュ	ス	ヘ	ン	£	÷
xxxx1110	(7)		.	>	N	^	n	→			ョ	セ	ホ	゛	ñ	
xxxx1111	(8)		/	?	O	_	o	←			ッ	ソ	マ	゜	ö	█

Hardware: Sensor Simulation

In our project, we needed a way to simulate the signals that the car sensors would send. We decided that using 8-100k ohm potentiometers are the best solution because a potentiometer is an analog signal that varies in voltage.

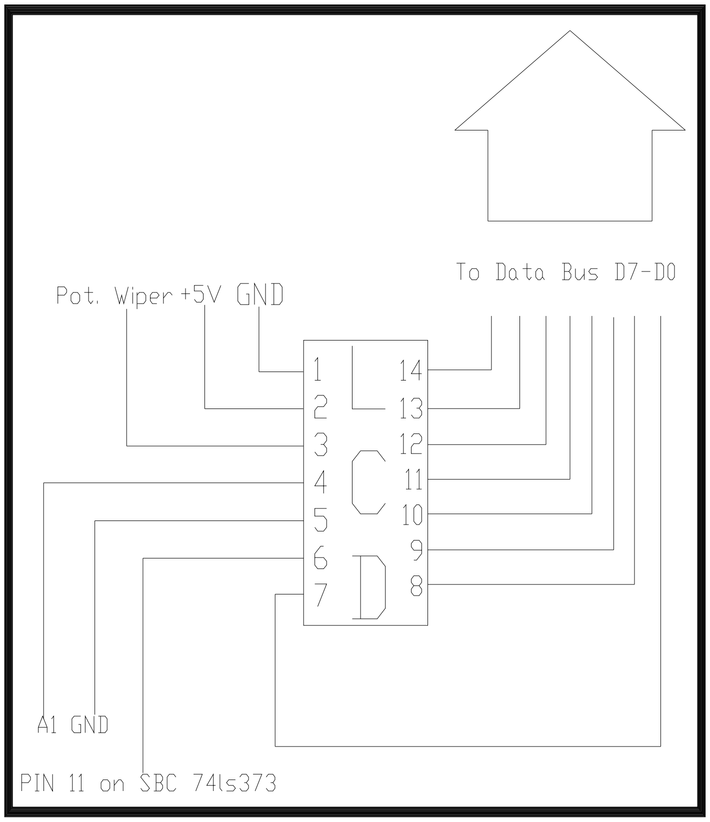
The potentiometers that we are using have three pins to be wired.





The wiper of each potentiometer is connected to the analog input pins on the ADC0808. The A/D converter chip can receive up to eight analog signals. Each Out pin of each potentiometer is wired to the In0-In7(pins 1-5, and 26-28) pins of the A/D converter.

LCD Schematics



Assembly Program: BCD Conversion

One of the most important subsystems of our project is to output data to an LCD screen. Suppose I want to send the number 255 to the LCD screen. In order for this to work I would have to send ‘2’ ‘5’ ‘5’ (when the characters are in quotes assume it’s the ASCII equivalent of that character). Simply stated, only one character can be sent to the screen at one time. In order for this to occur, some processing must take place, it is known as binary to BCD conversion. If we were to convert the number 255 the following steps would have to take place:

255÷100=**2** Remainder 55
55÷10=**5** Remainder 5
5÷1=**5** Remainder 0

As you can see from the bold type the numbers ‘2’ ‘5’ ‘5’ are isolated from the original 255. This is the only way characters can be sent to the LCD screen, one at a time. A final operation must be done on this data, because of the ASCII standard a bias of $30h must be added to each character being sent to the screen so
(2+$30)
(5+$30)
(5+$30)
will yield 255 written on the LCD screen.

The code to perform such an algorithm is described above with the addition of deleting most significant zeros from the 3 number string, and returning the cursor back to the start position (reset).

```
go          cmp.b       #99,d7            ;check if lower then 100
            bls         showdec     ;if so branch to showdec
            move.b      #1,d3       ;if not set d3 flag to 1
showdec     move.w      d7,d6       ;copy string into register D6
            move.w      #100,d5     ;store 100 devisor
            bsr         dodigit           ;branch to dividing algorithm
            move.w      #10,d5            ;store 10 devisor
            bsr         dodigit           ;branch to dividing algorithm
            move.b      d6,d1       ;copy d6 into d1
            addi.b      #$30,d1     ;add ASCII bias
            bsr         delay       ;send to screen with delay
reset       movea.l     #$18000,a0  ;set operation mode for LCD
            move.b      d2,d0       ;move LCD start add into LCD mem.
            bsr         delay       ;send to screen with delay
            bra         read        ;return to main function
dodigit     andi.l      #$ffff,d6   ;clear upper word of d6
            divu        d5,d6       ;divide string by devisor
            move.b      d6,d1       ;copy d6 into d1
            addi.b      #$30,d1     ;add ASCII bias of $30h
            cmp.b       #1,d3       ;check d3 flag
            beq         do          ;If equal branch to send char
            cmp.b       #$30,d1     ;if not equal send go to do
            bne         do
            move.b      #$80,d0     ;send blank space to screen
            bsr         delay       ;send with delay
            bra         do2         ;go to do2 procedure
do          move.b      d1,d0       ;send to screen
            bsr         delay       ;with appropriate delay
do2         swap        d6          ;get remainder
            rts                     ;return to original routine
```

If traced through properly you will find that the above algorithm takes up to any 3 digit number, separates each character to send to the screen separately, removes any most significant ‘0’ characters, and resets the cursor to a start position. This start position is different for every sensor string. For example, the start position for the following string: “The number is=xxx” is address 14. The string “The xxx is right” has a start position of 4. It is vital that the cursor return to the proper start position or the LCD display will not function properly.

Assembly Program: Reading Individual Sensors

The code to read from each sensor and output the proper data is vital to the operation of our project. Each sensor has its own data so to speak. To illustrate this consider this chart:

Sensor input	Data from $18000
1	000
2	001
3	010
4	011
5	100
6	101
7	110
8	111

Because such data exists it is possible to check the data from $18000h and perform an address jump that corresponds to a subroutine that contains the data processing for each sensor. The code for this is shown below:

```
read1   move.b      $18000,d7       ;store data from 18000 into register d7
        move.b      d7,d0           ;copy d7 into d0
        addi.b      #1,d0           ;adds one to d0 (for the next sensor)
        move.b      d0,$28000       ;writes this to the latch
        movea.l     #$18002,a0      ;changes to character write mode
        movea.l     #$8300,a6       ;address index into a6
        mulu.w      #$4,d7          ;multiply by 4 compensates for 4byte long instr.
        adda.l      d7,a6           ;adds address d7 to index
        jmp         (a6)            ;jumps to index

        org     $8300
        bra     tps     ;once the jump is made
        bra     map     ;the branch to separate subrountines
        bra     temp1   ;to handle processing/LCD info can be made
        bra     temp2
        bra     o2
        bra     iat
        bra     null
        bra     rand
```

Once the jump has been made the processing/retrieving characters to send to the screen is trivial. How this is done is shown in the main program.

Assembly Program: Processing Data

Raw data from the sensor output or potentiometers must be processed into information that can be comprehended by humans. In order to do this a sensor characteristic must be found, and the math processing must be done to approximate this characteristic. Below is an example taken from the code that processes the TPS sensor.

```
tps     movea.l     #tpsd,a3     ;sets address for TPS character info into a3
        movea.l     #tpsa,a4     ;sets address for TPS math processing
        move.b      #8,d3        ;sets the amount of characters initially written LCD
        move.b      #$84,d2      ;sets return address of cursor
        bra         loop         ;branches to character retrieving/send algorithm
tpsa    move.b      #$FF,d1      ;approximation of sensor char performed on d7
        sub.b       d7,d1
        mulu.w      #$6,d1
        divu        #17,d1
        move.b      d1,d7
        andi.l      #$ff,d7      ;clear everything but lower byte of d7
        bra         go           ;branches to BCD algorithm

loop    move.b      (a3)+,d0     ;fetches TPS characters
        bsr         delayl       ;branches to a delay routine
        subq.b      #1,d3        ;loop decriment
        bne         loop
        bra         reset        ;LCD address reset routine
error   move.b      #$45,d0      ;if there is an error
        bsr         delayl       ;this routine will be called
        move.b      #$72,d0      ;and type ‘Err’ to screen
        bsr         delayl
        move.b      #$72,d0
        bsr         delayl
        bra         reset        ;LCD address reset routine

        org     $8350
tpsd    dc.b    'TPS=',$80,$80,$80,$df        ;here is the location for TPS char info
```

The above subroutines are what handle character fetch, send, delay, reset, and processing of the signal into useable data. Subroutines “loop”, and “error” are universal for all the sensors, subroutine “loop” fetches the characters and sends them to the screen, “error” notifies the user of a sensor error. Subroutine “tpsa” handles the math, subroutine “tps” handles setting certain flags used for fetching characters and resetting the cursor.

Assembly Program: Program Flow

To someone that is unfamiliar with assembly language code, it would be difficult for them to understand the program flow of our final program. The program structure is as follows:

1. Initialization routine to clear LCD display
2. Read from the latch to determine which sensor is being accessed
3. Write to latch for the next sensor to be accessed after the reset press
4. Branch to subroutine that handles the sensor being accessed
5. Set flags such as character fetch data, and cursor reset information
6. Write characters for the current sensor to the screen i.e. “TPS=xxx Degrees”
7. Read information from analog-digital converter for the current sensor
8. Process information into useable data using sensor characteristics
9. Perform the BCD algorithm/MSB zero deletion algorithm
10. Go to step 7

Additional Rules:

- Any information/initialization data send to the LCD screen must be accompanied with a delay.
- The program, when executed, is infinite, it doesn’t stop until the reset button is pressed.
- When the reset button is pressed the program returns to step 1 of the basic program flow.

Assembly Program: Final Assembly Code

Below is the fully commented final assembly code used on the SBC through serial port communication.

```
        org         $8000           ;start at address $8000
init    lea         tps,a3          ;sets a3 address
        movea.l     #$18000,a0      ;RS=0
        movea.l     #$20000,a1      ;sets ADC to address register
        move.b      #4,d2           ;loop size
init2   move.b      #$30,d0         ;initialization
        bsr         delay           ;delay
        subq.b      #1,d2           ;loop decrementer
        bne         init2
        move.b      #$01, d0        ;initialization
        bsr         delay           ;delay
        bsr         delay2          ;added delay for clear
        move.b      #$0E,d0         ;initialization
        bsr         delay           ;delay
        move.b      #$0C,d0         ;initialization
        bsr         delay           ;delay
        bra         read1   ;       ;branch to next part of program
delay2  move.w      #$FFFF,d1       ;delay functions
        move.w      #$FFFF,d2
delay3  subq.w      #1,d2
        bne         delay3
delay4  subq.w      #1,d2
        bne         delay4
        rts
;***********************************
;***********************************
read1   move.b      $18000,d7       ;move from latch into d7
        move.b      d7,d0           ;copy into d0
        addi.b      #1,d0           ;add 1 to d0
        move.b      d0,$28000       ;send next sensor info to latch
        movea.l     #$18002,a0      ;character write mode
        movea.l     #$8300,a6       ;set address index
        mulu.w      #$4,d7          ;multiply to compensate for 4byte opcode size
        adda.l      d7,a6           ;adds address index and compensation
        jmp         (a6)            ;jump to address of sensor

tps     movea.l     #tpsd,a3        ;move data location into a3
        movea.l     #tpsa,a4        ;move algorithm info to a4
        move.b      #8,d3           ;character retrevial amount
        move.b      #$84,d2         ;LCD cursor reset
        bra         loop            ;branch to fetch/write char to screen
tpsa    move.b      #$FF,d1         ;math approximation code
        sub.b       d7,d1
        mulu.w      #$6,d1
        divu        #17,d1
        move.b      d1,d7
        andi.l      #$ff,d7         ;leaves lower byte intact
        bra         go

map     movea.l     #mapd,a3        ;move data location into a3
        movea.l     #mapa,a4        ;move algorithm location to a4
        move.b      #15,d3          ;character retreval amount
        move.b      #$84,d2         ;LCD cursor reset
        bra         loop            ;branch to fetch/write char to screen

mapa    cmp.b       #$99,d7         ;check for error
        bhi         error
        move.b      #$99,d1         ;math approximation code
        sub.b       d7,d1
        move.b      d1,d7
        divu        #$6,d7
        andi.l      #$ff,d7         ;leaves lower byte intact
        bra         go

temp1   movea.l     #temp1d,a3      ;move data location into a3
        movea.l     #temp1a,a4      ;move algorithm location to a4
        move.b      #19,d3          ;character retreval amount
        move.b      #$8e,d2         ;LCD cursor reset
        bra         loop            ;branch to fetch/write char to screen
temp1a  move.b      #$FF,d1         ;math approximation code
        sub.b       d7,d1
        move.b      d1,d7
        divu        #$2,d7
        andi.l      #$ff,d7         ;leaves lower byte intact
        bra         go

temp2   movea.l     #temp2d,a3      ;move data location into a3
        movea.l     #temp1a,a4      ;move algorithm location to a4
        move.b      #18,d3          ;character retreval amount
        move.b      #$8d,d2         ;LCD cursor reset
        bra         loop            ;branch to fetch/write char to screen

o2      movea.l     #o2d,a3         ;move data location into a3
        movea.l     #o2a,a4         ;move algorithm location to a4
        move.b      #19,d3          ;character retreval amount
        move.b      #$8f,d2         ;LCD cursor reset
        bra       loop          ;branch to fetch/write char to screen

o2a     mulu.w    #100,d7       ;math approximation code
        divu      #$FF,d7
        andi.l    #$ff,d7       ;leaves lower byte intact
        bra       go

iat     movea.l   #iatd,a3      ;move data location into a3
        movea.l   #temp1a,a4    ;move algorithm location to a4
        move.b    #20,d3        ;character retreval amount
        move.b    #$8f,d2       ;LCD cursor reset
        bra       loop          ;branch to fetch/write char to screen

null    movea.l   #nulld,a3     ;move data location into a3
        movea.l   #nulla,a4     ;move algorithm location to a4
        move.b    #19,d3        ;character retreval amount
        move.b    #$93,d2       ;LCD cursor reset
        bra       loop          ;branch to fetch/write char to screen
nulla   bra       stop          ;stop program because of no sensor

rand    movea.l   #randd,a3     ;move data location into a3
        movea.l   #randa,a4     ;move algorithm location to a4
        move.b    #20,d3        ;character retreval amount
        move.b    #$91,d2       ;LCD cursor reset
        bra       loop          ;branch to fetch/write char to screen
randa   bra       go

loop    move.b    (a3)+,d0      ;fetches next character from a3
        bsr       delayl        ;writes char to screen w/ delay
        subq.b    #1,d3         ;loop decrementer
        bne       loop
        bra       reset         ;cursor reset
error   move.b    #$45,d0       ;subroutine writes Err to screen
        bsr       delayl        ;with delay
        move.b    #$72,d0
        bsr       delayl
        move.b    #$72,d0
        bsr       delayl
        bra       reset
read    clr.b     d3            ;resets d3
        movea.l   #$18002,a0    ;character write mode
        move.b    (a1),d7       ;moves data from ADC to data register
        jmp       (a4)          ;jumps to current sensor data processing
go      cmp.b     #99,d7        ;check if lower then 100
        bls       showdec       ;if so branch to showdec
        move.b    #1,d3         ;if not set d3 flag to 1
```

```
showdecmove.w   d7,d6           ;copy string into register D6
        move.w  #100,d5         ;store 100 devisor
        bsr     dodigit         ;branch to dividing algorithm
        move.w  #10,d5          ;store 10 devisor
        bsr     dodigit         ;branch to dividing algorithm
        move.b  d6,d1           ;copy d6 into d1
        addi.b  #$30,d1         ;add ASCII bias
        bsr     delay           ;send to screen with delay
reset   movea.l #$18000,a0      ;set operation mode for LCD
        move.b  d2,d0           ;move LCD start add into LCD mem.
        bsr     delay           ;send to screen with delay
        bra     read            ;return to main function
dodigit andi.l  #$ffff,d6       ;clear upper word of d6
        divu    d5,d6           ;divide string by devisor
        move.b  d6,d1           ;copy d6 into d1
        addi.b  #$30,d1         ;add ASCII bias of $30h
        cmp.b   #1,d3           ;check d3 flag
        beq     do              ;If equal branch to send char
        cmp.b   #$30,d1         ;if not equal send go to do
        bne     do
        move.b  #$80,d0         ;send blank space to screen
        bsr     delay           ;send with delay
        bra     do2             ;go to do2 procedure
do      move.b  d1,d0           ;send to screen
        bsr     delay           ;with appropriate delay
do2     swap    d6              ;get remainder
        rts                     ;return to original routine

delay   move.b  #6,d1           ;delay rountines
delay1  move.b  d0,(a0)
        subq.b  #1,d1
        bne     delay1
        rts
delayl  move.b  #13,d1
        move.b  d0, (a0)
delayl2subq.b   #1,d1
        bne     delayl2
        rts
        org     $8300
        bra     map
        bra     temp1
        bra     temp2
        bra     o2
        bra     iat
        bra     null
        bra     rand
```

```
stop     trap       #9
         org        $8350
tpsd     dc.b       'TPS=',$80,$80,$80,$df          ;data
mapd     dc.b       'MAP=',$80,$80,$80,' in. Hg.'
temp1ddc.b          'Cyl.Head Temp=',$80,$80,$80,$df,'C'
temp2ddc.b          'Coolant Temp=',$80,$80,$80,$df,'C'
o2d      dc.b       'EGR Valve Lift=',$80,$80,$80,'%'
iatd     dc.b       'Intake AirTemp=',$80,$80,$80,$df,'C'
nulld    dc.b       'No Sensor Available'
randd    dc.b       'Test of 8bit ADC=',$80,$80,$80
         end        init
```

PC Program

This program communicates with the SBC through the serial port. In the assembly program we will declare a3 as the address for the serial port. With our SBC, the address is $10000. Then whenever we read from the potentiometer, we will send the data to the serial port as well as the LCD. We use this code:

```
        move.b      d7, a3      ;moves the data in d7 to the serial port
```

At this point, everything is happening at the PC end. First we needed to create a form and load in the MSComm32.OCX component. This component allows serial communication through com port 1. The program is written in Visual Basic 5. We picked an easy programmable language to get a good visual of what is happening from the sensor. In the SBC, the Intel 8251 UART is sending data at a 38400 baud rate. In the program, we set the Com Port to 8-bit, no parity, no handshaking, and 38400 baud rate, with a stop bit of 1.

```
MSComm1.Settings = "38400,N,8,1"      'sets the baud rate, parity bit, bit size and
                                      'stop bit
MSComm1.Handshaking = comNone         'no handshaking of port
```

Then we had to set the length of the buffer, and the mode at which to send. Since we are going to be receiving binary data, the mode has to be set to binary. As information is sent through the serial port, a buffer stores the data until the user of the program asks for it. by setting the length of the buffer to zero, the program will read all data in the buffer. These commands set these options:

```
MSComm1.InputLen = 0                        'sets length of buffer to 0
MSComm1.InputMode = comInputModeBinary      'sets to read binary
MSComm1.NullDiscard = False                 'accepts binary "0"
MSComm1.DTREnable = True                    'sets the data terminal ready to true
```

Once everything is initialized, the port can be opened. When the port is opened, the PC will now be receiving whatever data is being sent. This data may be coming to the port incorrectly due to non-synchronization. Once the reset button is hit, however, the data should be sending correctly. Once the data is received, the PC must convert the 8-bit data into useful information. We need the data converted to a number. This is the operations needed to do so:

```
Select Case MSComm1.CommEvent
        Case comEvReceive             'redefines the size of buffer then
        txtSensor.Text = ""
        ReDim InBuffer(MSComm1.InBufferCount)
        ReceiveBits                         'reads the bits from the port
        Case comEventRxOver                 'redefines the size of buffer if
        ReDim InBuffer(MSComm1.InBufferCount)        'the buffer starts to overflow
End Select

If MSComm1.InBufferCount = 0 Then           'checks for data in buffer and displays if not
        txtSensor.Text = "No Data in Buffer"
InBuffer = MSComm1.Input                    'pulls data from buffer
For i = 0 To UBound(InBuffer)               'takes upper bounds of buffer
        message = Chr$(InBuffer(i) + 30)    'and puts it in a string
Next i
txtSensor.Text = "Sensor Reading= " & message     'writes message to text box
```

The Chr$ is the line that takes the 8 bits and converts it to a character string. Since we need a number, we need to add $30 to the binary number to gets the correct ASCII character string we need. Below is an ASCII chart to see how it converts from binary to the character needed (taken from Professor Rosenstark's ECE 252 Supplemental Notes):

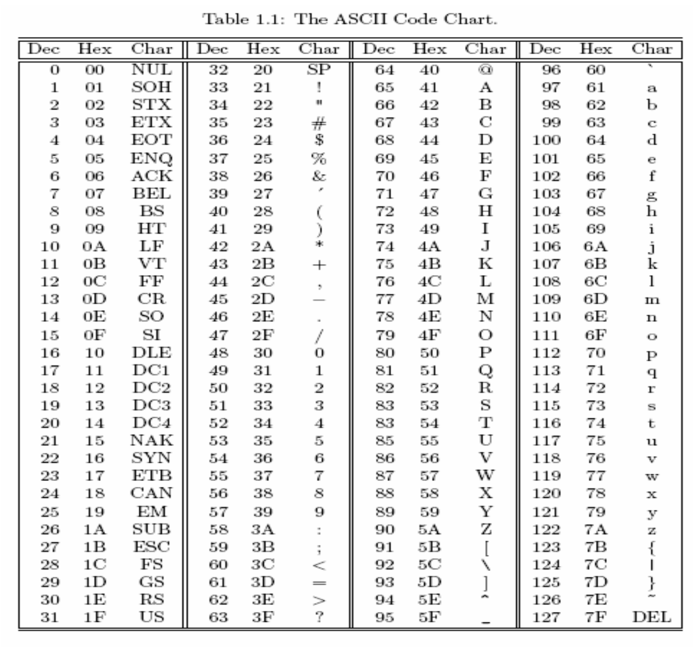


Table 1.1: The ASCII Code Chart.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	NUL	32	20	SP	64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	–	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	83	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

Once the PC has the correct character, which in this case is a number, we can generate a scope view. Below is the code to have the data received from the SBC transformed into graphical data:

```
iLow = iHigh + iDelta          'Low Scope Trace Value
LabelOffSet = Index * 35       'One Control Array For All Ports (offset accordingly)
picScope.Refresh               'Clear The Scope Display
hPic = shpScope.hDC            'Get The Handle of The Scope Display Picture Box
iBytes = UBound(nScope)        'No Of Bytes In The Data Array To Display
```

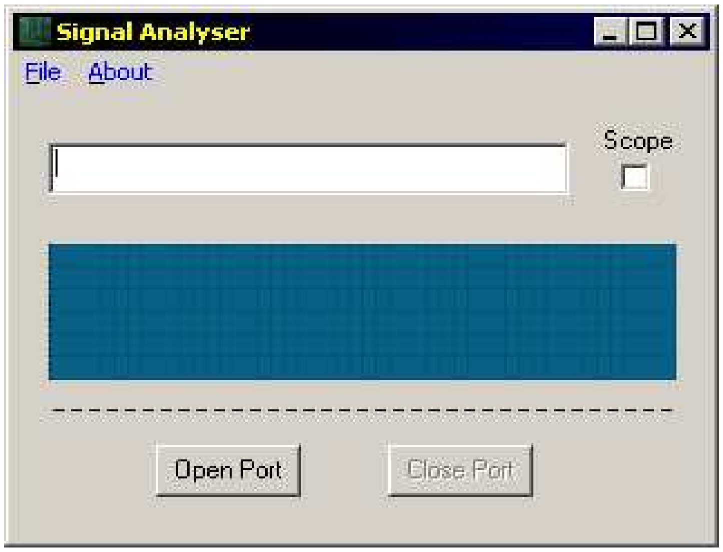
```
iZ = 0

With PortSet                    'Count The Additional Bits From The Ports Settings
        If .Parity <> "n" Then
              iZ = 1
              bChkParity = True
        End If
        iBits = CLng(.DataBits)              'Number Of Bits To Display From Port
                                             'Settings
        iStop = CLng(.StopBits)              'Number Of Stop Bits To Add
        iZ = iZ + iBits + iStop'Sum Stop,Parity, and Data Bits
End With

iBitCount = (iZ * (iBytes + 1)) + (iBytes + 1)
ReDim bTrace(iBitCount - 1)                  'Trace Array One less than Total BitCount
                                             ' (array index)
        iZ = 0
For iX = 0 To iBytes                         'Max iBytes is 3 (Four Byte Trace)
        bTrace(iZ) = True                    'Start Bit
        lblScope1(iZ + LabelOffSet).Caption = sStartBitLabel
        iZ = iZ + 1
        iOnBits = 0                          'Zero Count For On Bits To Check Parity
        For iY = 1 To iBits                  'Parse Each Data Bit
              bOn = BitOn(Data(iX), iY)
              iOnBits = Abs(bOn) + iOnBits
              bTrace(iZ) = Not bOn           'Invert as Negative Voltage on Scope is
                                             'Logical True
              lblScope1(iZ + LabelOffSet).Caption = CStr(iY - 1)
              iZ = iZ + 1
        Next
        If bChkParity Then                   'True For All But Parity Setting Of "None"
              Select Case PortSet.Parity
              Case "e"                       'Even Parity
              If iOnBits Mod 2 Then          'OnBits is Odd Parity Bit Is On
                bTrace(iZ) = False
              Else
                bTrace(iZ) = True            'OnBits is Even Parity Bit Is Off
              End If
              Case "o"                       'Odd Parity
              If iOnBits Mod 2 Then          'OnBits is Odd Parity Bit Is Off
                bTrace(iZ) = True
                Else
                bTrace(iZ) = False           'OnBits is Even Parity Bit Is On
                End If
              Case "m"                       'Mark Parity Bit is Always On
                bTrace(iZ) = False
                Case "s"
                  bTrace(iZ) = True                 'Space Parity Bit is Always Off
                End Select
                lblScope1(iZ + LabelOffSet).Caption = sParityBitLabel
                iZ = iZ + 1
        End If

'Set The Scope Trace Stop Bits
If iStop = 1 Then                   'One Stop Bit
        bTrace(iZ) = False      'Stop Bit
        lblScope1(iZ + LabelOffSet).Caption = sStopBitLabel
        iZ = iZ + 1
Else                                'Two Stop Bits
        bTrace(iZ) = False      'Stop Bit
        lblScope1(iZ + LabelOffSet).Caption = sStopBitLabel
        bTrace(iZ + 1) = False                  'Stop Bit
        lblScope1(iZ + LabelOffSet + 1).Caption = sStopBitLabel
        iZ = iZ + 2
End If
Next

'Get The Proper Scope Trace To Match The Background Color
iTraceColor = typDisplayColor.Trace
iBits = 0                           'Zero The Bit Count Index
bOn = bTrace(iBits)                 'Parse The Trace Array And Draw on Scope Display
For iX = 1 To picScope.ScaleWidth               'X Coordinate
        iY = iLow - (Abs(bOn) * iDelta)         'Y Coordinate
Do While iX Mod iPW                             'Draw Horizontal Line To The Pulse Width
        SetPixel hPic, iX, iY, iTraceColor
        iX = iX + 1
Loop
If iBits < UBound(bTrace) Then                  'Check For End Of Trace
        If bTrace(iBits) <> bTrace(iBits + 1) Then      'Check To See If Next Bit Has
                                                        'Changed State
                For iY = iHigh To iLow                  'Next Bit Change In State Draw
                                                        'Vertical Line
                                SetPixel hPic, iX, iY, iTraceColor
               Next
        End If
        iBits = iBits + 1                       'Get The Next Bit
        bOn = bTrace(iBits)
Else                                            'End Of Trace Data Set The Stop Bit
        iX = iX + 1                             'Skip a Bit for the Stop Bit
        iBits = iBits + 1 + LabelOffSet
        Exit For
End If
```

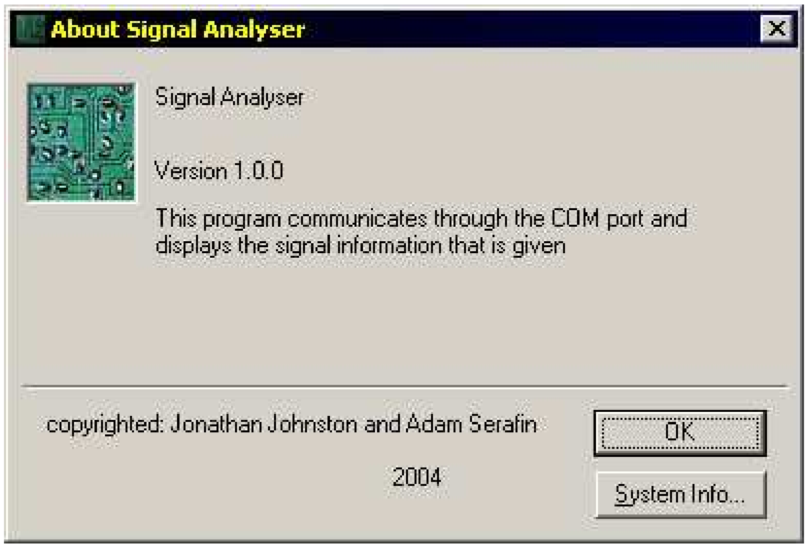
```
Next

Do While iX < picScope.ScaleWidth           'Run The Scope Trace Out
        SetPixel hPic, iX, iLow, iTraceColor
        iX = iX + 1
Loop
        For iX = iBits To LabelOffSet + 47   'Set The Remaining Scope Label Captions
              lblScope1(iX).Caption = "-"
        Next
```

These are the basic functions needed to have the SBC communicate with the PC. The scope is the feature in the PC program that further analyses the sensors from the vehicle. The whole program can be found in the programming section of the report.

Visual Basic Program: Screen Shots

Below are several preliminary screenshots of the visual basic program we created:





Signal Analyser

Version 1.0.0

This program communicates through the COM port and displays the signal information that is given

copyrighted: Jonathan Johnston and Adam Serafin

2004

Conclusion:

Designing and implementing our project was a very challenging process. Design methods that did not work had to be scrapped. Using new hardware such as LCD displays, and Analog to Digital converters involved a learning curve that required a lot of time to master. For example, the first LCD screen we ordered was defective. Unsure of our own understanding and methods to work the screen plagued us until enough testing had gone by when we finally had no doubt the LCD screen was faulty. Designing the sensor scroll feature was also met with difficulty. After several different attempts at using different hardware failed, the idea of using a latch to store next sensor states proved successful.

Even software design proved to be a difficult endeavor that required a good understanding of the Motorola 68k Series Instruction Set Architecture. In addition to the syntax, programming structures such as loops, conditions, branches, subroutines, and data flow all had to be mastered. Software design in VB is a little less tedious then that on the assembly level but still a major pain nonetheless, and there are still bugs in this software that would take months to massage out.

In addition to learning from the problems, we also learned a bit on structured design. By using structured design it is possible to tackle one design problem at a time rather then a host of design problems that would be difficult or impossible to debug effectively. Overall this project was a huge success because we had high goals, and met them through hard-work. The end result is a smoothly running system that is a valuable asset to any technician or car junkie wishing to diagnose a problem resulting in poor drivability, low fuel economy, and failed emissions tests.