Team America

Bill Hazel
Michael McSorley
Billy Parker
George Warner

Final Documentation
EE Senior Design 2010
Professor Schafer

Table of Contents

1. Introduction:

The search for sustainable energy in a world of depleting natural resources has led to the development of systems which allow automobiles to run on less, or no, gasoline. Hybrid cars have the capability of combining the power of internal combustion engines and electric motors to substantially increase the efficiency of the vehicle, and electric cars are able to run entirely off of batteries and eliminate the need for any other fuel. However, both systems involve batteries that have finite and fairly short life times. A hybrid electric vehicle that is capable of powering an electric motor using a bank of ultra-capacitors would avoid the problems of short lifetimes and recharging batteries.

Dr. Bauer, a professor at the University of Notre Dame, is working with a company to perform research on series hybrid vehicles that use ultra-capacitors to provide the power for the electric drive motors in place of batteries. However, the current state of capacitor development dictates that the bank of ultra-capacitors being used for power in the hybrid needs to be recharged after roughly two minutes of operation, depending on the driving conditions. Recharging the ultra-capacitors while still being able to drive is thus essential in order to travel any meaningful distance in a trip. To this end, the hybrid uses a diesel generator located in the bed of the truck to recharge the ultra-capacitors after they have been discharged.

Dr. Bauer quickly found out when doing testing with this capacitor-based hybrid setup that it is impractical to have a person monitoring ultra-capacitor voltage levels and turning the generator on and off. Therefore, an automated switching system that turns the generator on and off according to ultra-capacitor voltage levels is desirable, which is where we come into the picture.

1.1 Problem Statement:

Our primary objective in aiding Dr. Bauer was to design and implement a working circuit that acted as an automated switching system to turn the generator on and off according to the ultra-capacitor voltage levels. However, there were several other functionalities that we wanted to achieve in the system in order to aid research. It was desirable for the person using the hybrid to be able to turn it on and off from inside the cab of the truck instead of doing this from the truck bed. Because the purpose of a hybrid is greater energy efficiency than a conventional gas-powered car, it is important to be able to measure power flow going to the ultra-capacitor bank and power consumption by the DC drive motors. Finally, to aid engine longevity it is important to turn the engine on only as often as necessary, leading to a requirement for a dynamic lower ultra-capacitor voltage bound based off of driving conditions and anticipated DC motor load.

1.2 System Requirements:

The following system requirements were determined in order to fulfill the Problem Statement:

1) Ability to determine when the generator needs to be turned on and off.
Solution: Use voltage sensor on ultra-capacitor stack routed to A/D converter on micro-controller that then sends logic to switching circuits based on ultra-capacitor voltage values.
2) Turn generator on and off.
Solution: Use micro-controller to read voltage generator voltage levels and send logic to power MOSFET gates to switch the relevant solenoids on the diesel.
3) Measure and store current levels.
Solution: Current sensors on cable going from generator to ultra-capacitors and cable from ultra-capacitor to DC motors. Micro-controller reads current levels and writes values to SD card.
4) GPS to measure and store position, speed.
Solution: Purchase GPS and integrate it into main board.
5) Dynamic lower bound on ultra-capacitor voltage.
Solution: Use GPS to capture real-time position and speed data and enter into algorithm to calculate anticipated DC motor load conditions.
6) SD card to store current levels.
Solution: Integrate SD card into micro-controller circuit. Write current levels to SD card.
7) In-cab user interface / LCD display / toggling of relevant data.
Solution: LCD display and board in a plastic case that shows different pertinent information based out of cab. Use push-buttons on case to toggle between different types of information. A toggle switch on case to turn hybrid on and off.

1.3 High Level Solution Description

The requirements to track power usage and turn the hybrid on and off from inside the cab evolved simultaneously into an integrated human-to-charging-system interface. This human-to-charging-system interface is a plastic case with an LCD screen, a toggle switch, and push-button's. It allows the user to turn the ultra-capacitor/diesel generator circuit off from inside the cab of the truck as well as display different types of data relevant to the system, such as current draw.

The requirement of knowing power usage led us to include current sensors in the project so we can determine the currents going both to the ultra-capacitors when they are charging as well as the currents going to the driver motors. This power usage requirement also led us to include an SD card in the project as a data-storage device so power data can be stored and examined in detail after driving the hybrid. In order to display the instantaneous power data to the user some type of visual display device is needed, leading us to use an LCD screen in the cab to show different information to the user.

In order to aid engine longevity and use the engine only when necessary, Dr. Bauer envisioned a dynamic lower bound on the ultra-capacitor voltage. The ultra-capacitor voltage threshold can be adjusted lower when the motors are experiencing low load requirements (i.e. going downhill) as well as adjusted upward when the motors are about to or are experiencing high load requirements (accelerating from a stop). To implement this dynamic ultra-capacitor voltage range it is necessary to know and predict the conditions the drive motors are under. The only device that can accurately determine what driving conditions, and therefore current levels, the hybrid is experiencing / is about to experience is a global positioning system (GPS). To this end, we integrated a GPS chip and antenna into our board design and executed the software that allows the GPS to talk to the micro-controller and GPS data to be displayed on the LCD.

Our solution to control the switching on and off of the generator was to use power MOSFET's as switches controlled by the micro-controller. A single MOSFET would each be positioned in series with one of the the three devices on the generator that controls its turning on and off. These devices are the starter solenoid, to turn on and off the starter motor; the glow plug solenoid to control the temperature of the cylinder head; and the fuel valve solenoid which provides the engine with fuel. Power MOSFET and MOSFET driver circuitry are used to take a 3.3 V logic signal from the microcontroller and turn it into a 25 V signal on the MOSFET gate that can turn on and off the three different solenoids on the generator in order to start and stop charging cycles.

Thus, our overall solution to Dr. Bauer's project requirements saw us utilizing the following devices:

- Micro-controller
- Power MOSFET's and MOSFET drivers
- GPS
- SD card
- Current Sensors
- LCD
- Transformer

1.4 Design Expectations and Results

We feel that we have met about half of the design expectations determined for us by Dr. Bauer. In the success category, our automated switching system to turn the generator on and off works well and has proven to be reliable. The human-to-charging-system interface located in the cab of the truck not only turns the hybrid on and off but also is capable of displaying all relevant information. Both the GPS system and the current sensors are able to talk to the micro-controller and the LCD displays correct information from the two.

Unfortunately, the unfinished category also has several entries, primarily related to the SD card. We were unable to get the micro-controller to write to the SD card and were therefore unable to store the information we were gathering from our current sensors and GPS. This prevented us from computing the dynamic lower bound on the ultra-capacitor voltage and changing the turn-on time of the generator based on current and anticipated driving conditions. Additionally, our ability to sense the ultra-capacitor in order to determine when to turn the generator on and off only partially works. While the micro-controller and code we have is able to accurately determine when the ultra-capacitor has reached its lower voltage bound, it does not accurately determine when the ultra-capacitor voltage has reached its upper bound.

2 Detailed Project Description

2.1 Theory of Operation

Our project was focused around the control aspect of the series hybrid vehicle. As may be seen in Figure 2.1, the heart of our control system is the PIC18F4620 micro-controller; all other devices are based around it. It is used to receive data from the GPS and current sensors, write to the SD card, write to the LCD, sense the ultra-capacitor voltage levels, and send the signals to turn the generator on and off.

The automated switching system works by using the micro-controller to sense the ultra-capacitor voltage and uses this information to determine when to turn the generator on and off. The micro-controller sends control signals to turn the starter motor, fuel valve, and glow plug on and off based on time, ultra-capacitor voltage, and generator voltage.

The GPS, SD card, LCD, and current sensors all communicate with the micro-controller. The LCD is controlled by push-buttons on the plastic case, which the user can press to toggle between various pieces of information about the overall hybrid system. For example, current draw, GPS coordinates, ultra-capacitor voltage, and vehicle speed are all displayed on the LCD to the user depending on the order in which the push-buttons are pressed. A toggle switch located on the same plastic case as the LCD and push-buttons is used to turn the hybrid on and off.

The GPS is used to obtain longitude, latitude, altitude, and velocity information and then sends this information to the micro-controller. When asked for it, the micro-controller outputs this information to the LCD. It also uses the position and velocity information to determine the dynamic lower-bound on the ultra-capacitor voltage.

Saving all of the desired data so it can later be analyzed after the running of the truck is the SD card. This data-storage device accepts information from the PIC and is used to store the current draw of the DC drive motors and ultra-capacitors.

Once the ultra-capacitor voltage drops below the minimum threshold of 55 volts, the micro-controller sends signals turning on the generator's glow-plug, starter motor, and fuel-valve. The glow plug is turned off after remaining on for 10 seconds. The starter motor is turned off after the

2.2 System Block Diagram

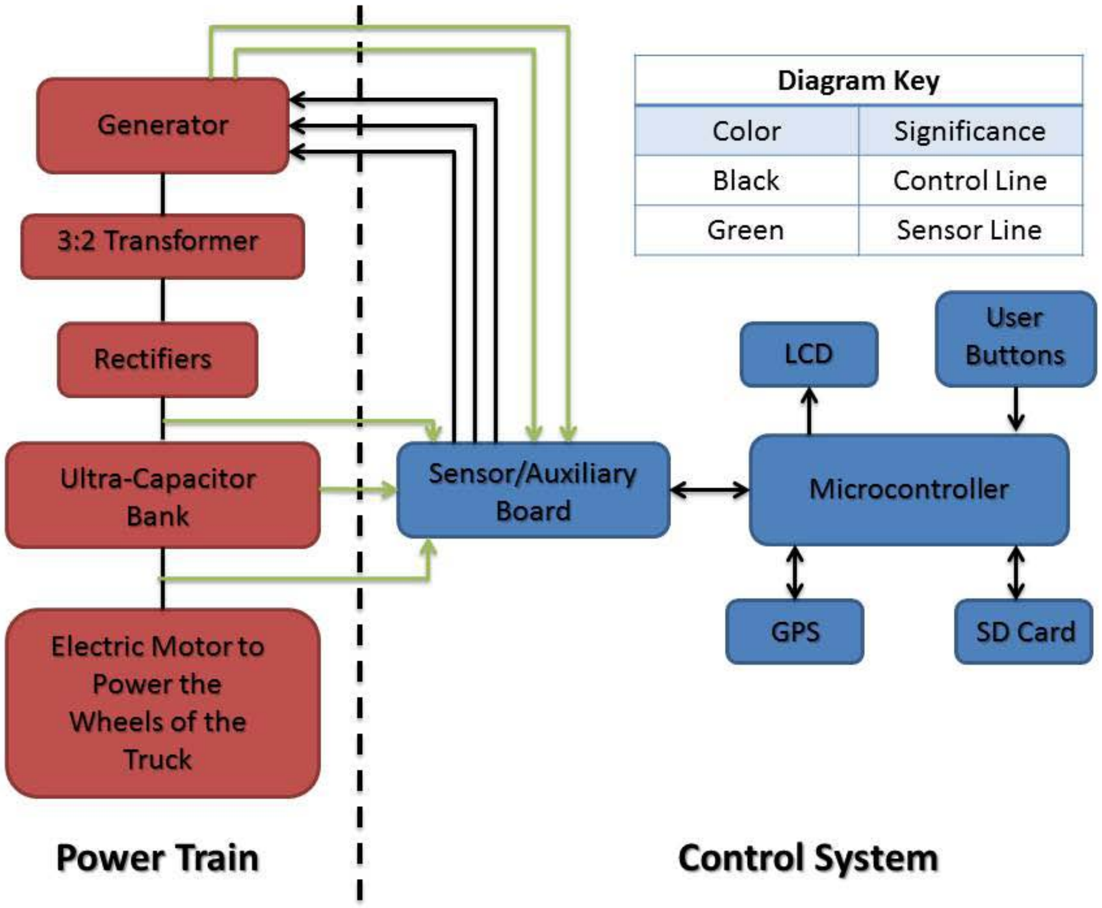


Figure 1 Overall System Block Diagram

2.3 Detailed Operation of Auxiliary Board Subsystem

The auxiliary board subsystem is designed to control the automated switching system and collect information about the power-train. As Figure 2 shows, this subsystem is controlled by the microcontroller on the main board and involves the MOSFET switching circuitry for the generator.

2.3.1 Auxiliary Board Subsystem Block Diagram

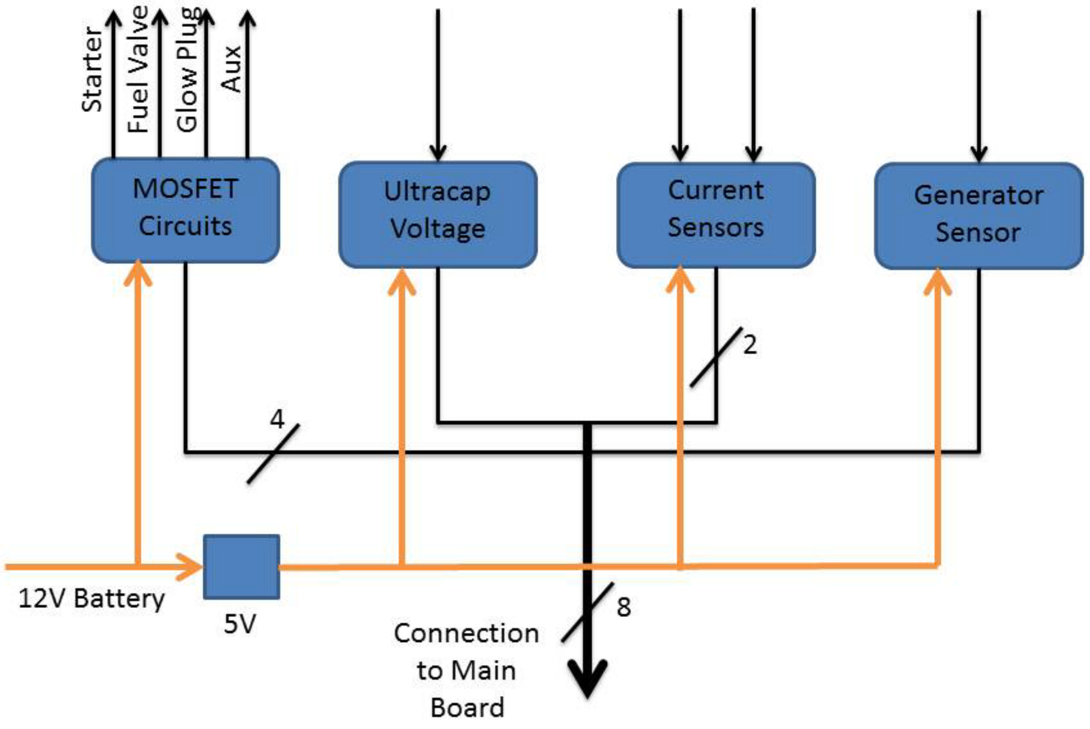


Figure 2 Auxiliary Board Block Diagram

2.3.2 MOSFET Circuit

Turning on and off the generator is accomplished by using n-channel power MOSFET’s as high-side switches to turn the starter, glow plug, and fuel valve on and off. The gate voltage of the power MOSFET’s is each set by a dedicated MOSFET driver. The drains of the MOSFET’s are connected to a common 12-volt lead-acid battery, and the MOSFET sources are connected to their respective devices on the generator. MOSFET 1 is connected to the solenoid that turns the starter motor on and off, MOSFET 2 is connected to the solenoid that turns the fuel valve on and off, and MOSFET 3 is connected to the solenoid that turns the glow plug on and off. Applying a gate voltage of 27 volts turns the MOSFET on. When the MOSFET is turned on current is allowed to pass from the drain to the respective solenoids.

As seen in Figure 3, when the MOSFET is turned off (i.e. has no gate voltage), no current can pass through the MOSFET's to the solenoids and the solenoids remain turned off. 3.3 volt or 0 volt logic signals to control the solenoids are sent by the PIC to MOSFET drivers. The MOSFET drivers control the gate voltage of the MOSFET's and are the devices that turn the power MOSFET's on and off. The MOSFET driver's turn the MOSFET's on and off by applying a 0 volt signal to the MOSFET gate when a 0 volt signal is sent from the PIC to the MOSFET driver, or a 27 volt signal to the MOSFET gate when a 3.3 volt signal is sent from the PIC to the MOSFET driver.

2.3.2.1 Schematic

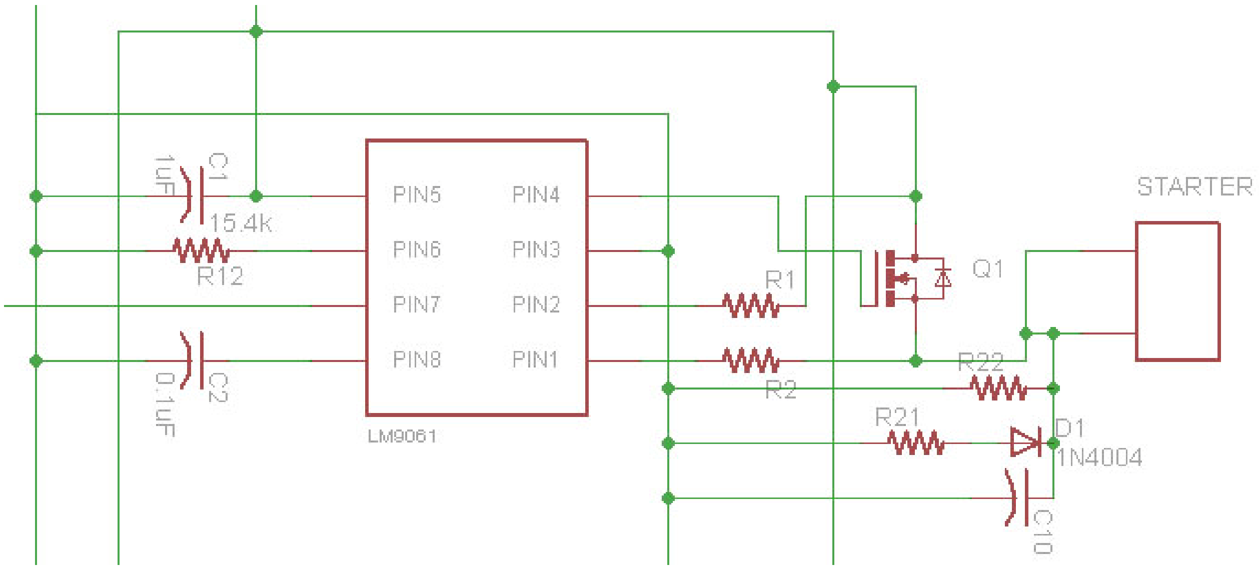


Figure 3 MOSFET Circuit Schematic

2.3.3 Ultra-capacitor Voltage Sensor

The MOSFET switching circuitry we designed is used to turn on and off the generator. The first piece of the puzzle in determining when to turn on and off the generator is sensing the voltage on the ultra-capacitor bank. This was done by running a wire from the ultra-capacitor bank to the auxiliary board. On the auxiliary board the ultra-capacitor voltage is multiplied by a 28:1 input:output ratio using a voltage divider. The signal is then sent to pin RA1 of the main board, where the A/D converter converts it to a digital form.

2.3.3.1 Schematic

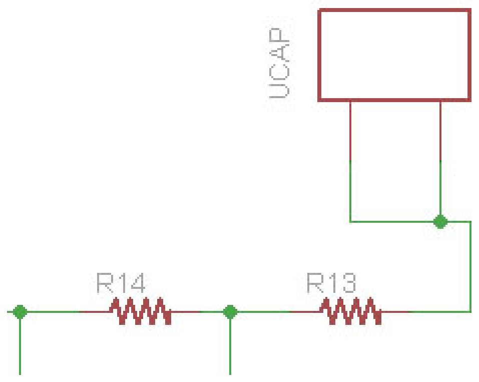


Figure 4 Ultra-capacitor Input Circuit

2.3.4 Current Sensors

The current sensors are through loop Hall Effect sensors. There are 2 LEM Hass 200-S sensors that each have a wire going through them. On the generator side of the ultracapacitor bank, the wire that power flows through to charge the bank has a current sensor to read the current into the ultracapacitor bank, and on the motor side, the wire coming out of the ultracapacitor bank has a current sensor to read the current out and ultimately the power being consumed by the vehicle. These sensors each take four wires, a supply votlage pin, a reference voltage pin, an output signal pin, and a ground pin. These wires run to the main board in the cab and the output signal is read by an A/D conversion in the PIC microcontroller on pins RA0 and RA2.

2.3.4.1 Schematic

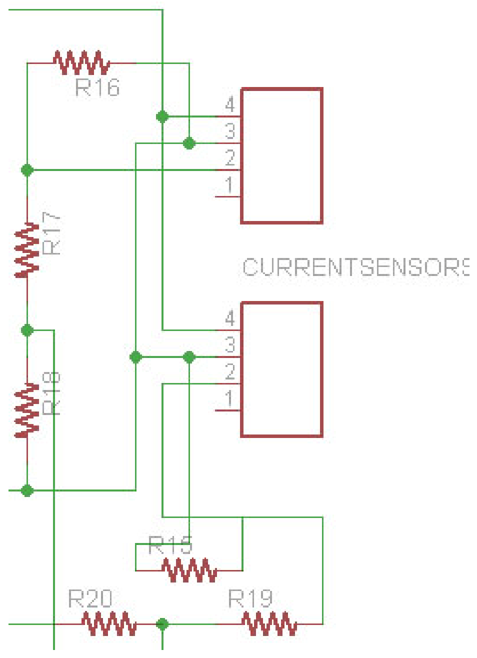


Figure 5 Current Sensor Schematic

2.3.5 Generator Sensor

The starter motor on the generator creates a large amount of torque for a short period and can cause some very serious vibrations on the entire generator system. In order to turn the generator on, both the fuel valve need to open and the starter has to turn the engine over to get the generator to start producing power. Because of the mechanical force and stress the starter introduces, it is desired to shut off the starter motor as soon as the generator is fully on. Therefore, a technique to determine that the generator was on was needed. The generator has an AC outlet on it that is used to send power to the ultracapacitors when charging it. By putting a splitter outlet on this outlet, we were able to insert our own three prong plug into the generator along with the plug that powers up the ultracapacitors. When a signal is sensed on this plug, the microcontroller knows to turn off the signal to the starter motor solenoid. More engineering was involved here though as will be described in the next section.

2.3.5.1 Schematic

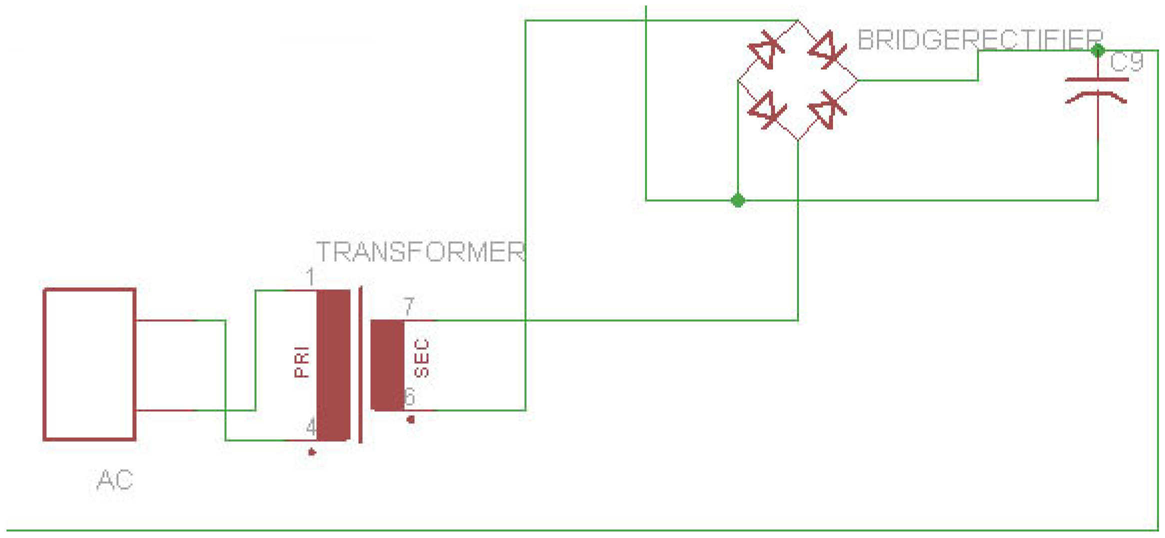


Figure 6 Generator Sensor Circuit Schematic

2.3.6 Op-Amp Circuits

Op-Amp circuits were initially placed on the auxiliary board in order to be used as unity gain buffers. However, the circuits only caused problems with our signals, possibly due to malfunctioning parts or, although unlikely because of careful design, the circuitry on the board. The unity gain buffers were left out and the signals that were supposed to have them (voltage and current sensors), were simply connected directly to the pins that led to the main board.

2.3.6.1 Schematic

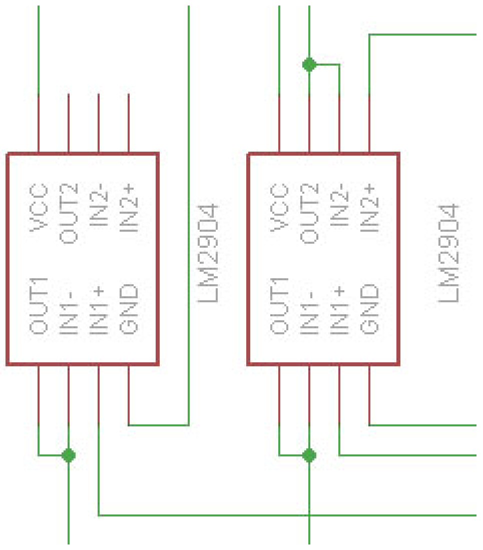


Figure 7 Op-Amp Schematic

2.3.7 Engineering Decisions and Auxiliary Board Testing

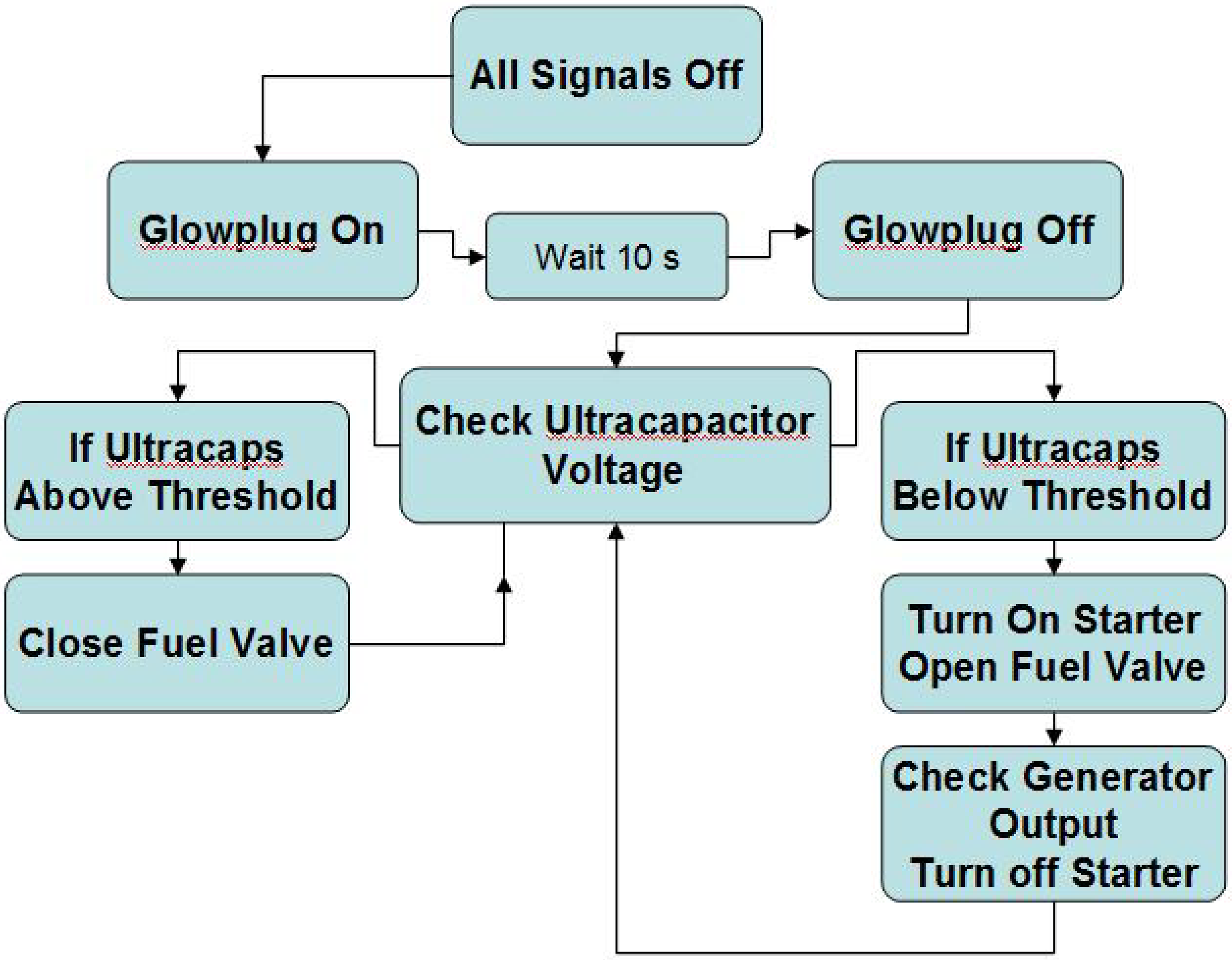
Relays were originally considered to switch the solenoids on and off, but mechanical reliability problems caused us to rule them out of the project. In their place, power MOSFET's were chosen to handle the switching because it was believed that they would be relatively simple to implement and more than capable of handling the current requirements.

One of the most challenging parts of the project was achieving a reliable method for turning the solenoids off after the MOSFET's had been used to turn them on. After realizing that the solenoids created large back voltages when we tried to turn them off and were breaking our power MOSFET's, we installed diodes in parallel with the solenoids. The thinking behind this approach is that when the MOSFET tried to turn off, the back voltage from the solenoid would turn the diode on and allow current to flow from ground to the solenoid before it broke the MOSFET.

This approach was only partially successful and our next step saw us installing a 470 micro-farad capacitor in parallel and a 100 ohm resistor in parallel with the solenoid and including a 47 ohm resistor in series with the diode in case the diode continued to break. We were hoping that the resistor and capacitor combination would help to lower the back voltage and give the diode time to turn on before the back voltage broke it. This circuit only worked once out of the three times that we tried it, so we turned to a larger power MOSFET. The last power MOSFET that we used was rated for 80 amps of current; well above the 30 amp maximum that the MOSFET would have to source in our system. Most importantly, the MOSFET was rated for a 100 volt maximum drain-to-source voltage. While lacking an explanation for why our original power MOSFET's continued to break even with the diode circuit, we experienced complete reliability with the 80A MOSFET.

For the engineering decisions in the generator sensing that was desired to turn off the starter motor, we chose to use the AC output signal from the outlets on the generator. By cutting a three prong outlet wire, we were found black, white, and green wires inside, corresponding to hot, neutral, and ground signals. The ground signal was covered with electrical tape, and only the hot and neutral wires of the 110 V AC signal were used. These two signals were attached to a screw terminal on the auxiliary board that connected to a 120:12 transformer. After the signal was transformed down to a safe level to be input to surface mount parts on our board, the signal was sent through a fullwave bridge rectifier with a 100 μF capacitor for DC smoothing. At this point, the signal was passed through the data cable that connected to the main board in the cab. This signal was around 16 V DC because of the RMS values and the DC smoothing occurring at the peaks of the signal. The options we considered at this point were to either use a voltage divider to divide the signal down to a safe input for the PIC microcontroller, or use a MOSFET switch with this signal applied to the gate. We were unsure of how much current was going to be coming from this signal when we implemented it on the generator system, and with a maximum current rating of 200mA on the PIC, we did not want to take any chances and went with the MOSFET design. The rectified DC signal was fed from the auxiliary board to the gate of a power MOSFET on the main board that had its drain held constant V_{DD}. When the transformed signal from the generator outlet was above about 4 V DC, the MOSFET would switch on at the source that was fed into pin RA3 and turn off the signal to the starter motor. This design worked well and effectively shut off the starter motor when the generator was up and running.

The hardware for the control system has been thoroughly explained, so it is time to take a look into the software that was controlling the turn on/off process. The main goal of our project was to be able to control the generator based on real-time inputs of voltage levels, to keep the ultracapacitor bank in a safe range. For this reason, the main function of the software was a continuous while loop that constantly checked the voltage on the ultracapacitor using an A/D conversion to determine if the generator needed to be switched on or off. The following diagram describes the flow of the programming in the control system subsystem of the project.


All Signals Off
Glowplug On
Wait 10 s
Glowplug Off
Check Ultracapacitor Voltage
If Ultracaps Above Threshold
Close Fuel Valve
If Ultracaps Below Threshold
Turn On Starter Open Fuel Valve
Check Generator Output Turn off Starter

Once the program starts, the engine is warmed by turning on the glowplug signal for 10 seconds, as was recommended by Dr. Bauer. After the engine is warmed, and the user has pressed the begin button, the program enters the main function and continually checks the ultracapacitor voltage. At a lower threshold of 57 volts, the generator will turn on, and when the A/D converter senses a voltage over 75 volts, the generator will shut off again. The code looks like the following.

```
while(1)
{
        //Check Voltage on Ultracaps and Charge if Necessary//
        ultracap_voltage = ultracap_voltage();
        delay_ms(10);
        checkbuttons();
        if (ultracap_voltage < 159)
        {
                start_charge();         // Turn on fuelvalve and starter signals
                check_starter();                // Turns off starter once generator is on;
                                        //  Will not continue until starter signal goes off
```

```
                        monitor_ultracaps(1);        // Monitor ultracap voltage; Turn off generator at full charge
                }//end if(uc<130)
        }
```

The while loop will never be left, and the voltage will be monitored at all times. The value of 159 is the value from the MSB of the A/D conversion that corresponds to 57 volts after the 57 volts is divided down in the resistor voltage divider on the auxillary board. The function start charge simply turns on the signals to the starter motor and fuel valve by setting the volatile bits assigned to LATB.3 and LATB.4. ‘Check_starter’ is a function that performs A/D conversion on port RA3. When the value of ADRESH, the MSB of the A/D conversion is over 200, the starter motor will be turned off. This value was chosen because the generator takes a while to start outputting the AC signal from the outlets, so lowering threshold was helpful in turning off the starter a little bit earlier than waiting for an input value of 1. After ‘check_starter’ is called, ‘monitor_ultracaps’ is called and the voltage on the ultracap bank is constantly checked until the value is over 208, the value corresponding to 76 volts. At this point, all signals are turned off and the function returns to the beginning of the while(1) loop. To see the functions that are called in this main loop, refer to the complete software listing later in the report.

2.4 Microcontroller and User Interface Subsystem

The second major subsystem included all of the aforementioned user interface. This section of the project revolved around the pertinent data being produced during the trucks performance. This data consisted of voltage and current levels, power consumption, global positioning system information, speed, amount of time spent charging ultracapacitors, and real time output signal levels. The hardware for this subsystem consisted of a variety of parts and functions interfacing with each other. By referring to the schematics and board layouts in this report, one can get a better understanding of the hardware involved.

On the in-cab display side, parts included a plastic mounting box that was bored out using a Dremmel drill. In the top, a rectangular hole was cut to mount the NHD-0420D3Z-FL-GBW LCD screen in. This cut was made so that the 20 character by 4 line display LCD would fit tightly without having to add screws to keep it in place. Also on the front of the mounting box are three circular holes of a 1cm diameter that were cut to mount the user buttons in. On the right side, a circular hole was cut to mount the on and off switch in place. As was done with the user buttons, the threaded rod was pushed through the bottom of the mounting box and from the top, a washer and a nut were screwed in place to keep the buttons and the switch solidly in place. Other features of the mounting box include a hole to feed the power cables through, and a hole to feed the input/output data cable through.

2.4.1 Subsystem Block Diagram

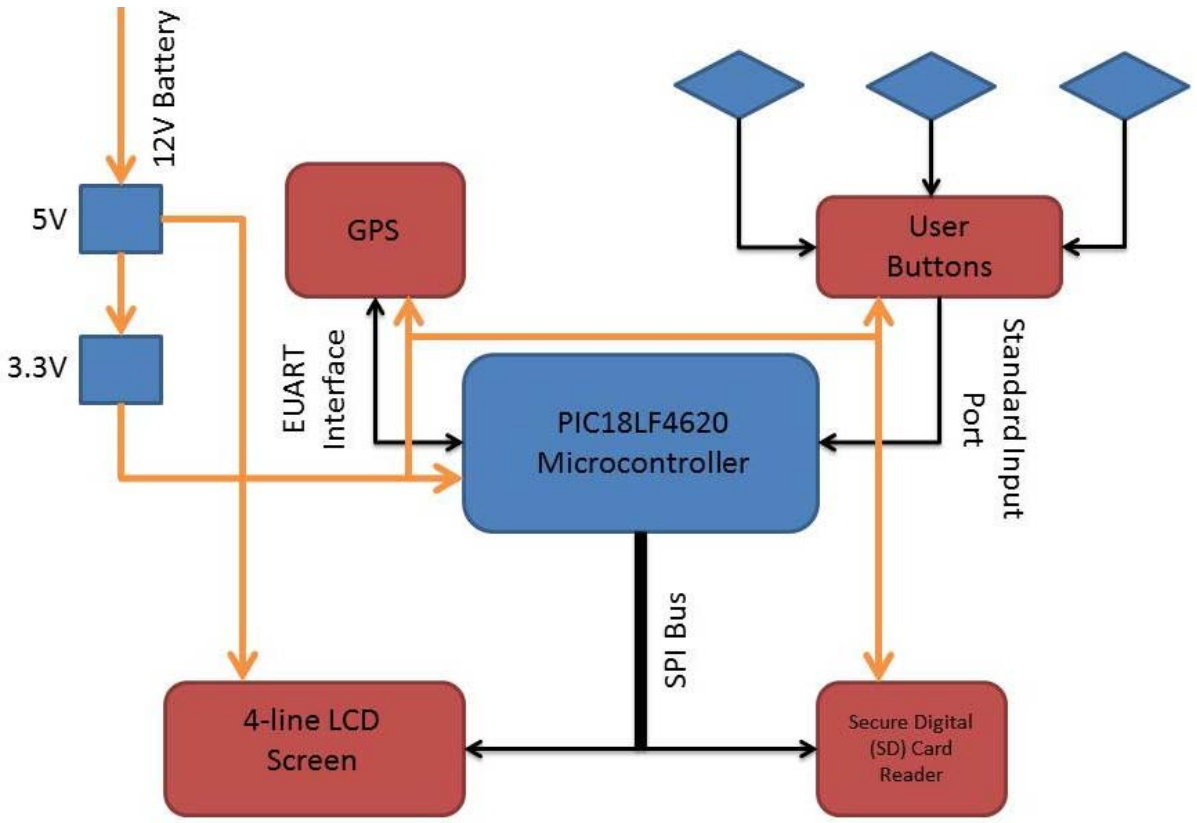


Figure 8 Main Board Block Diagram

2.4.2 Microcontroller

Inside of the mounting box is the main board that can be seen in the board layout. The user interface subsystem uses every part of this main board. First, the board was programmed with software using the programmer port. This communicates with the programming pins PGM, PGC, and PGD or RB4, RB5, and RB6 of the PIC18F4620 microcontroller. Once the program is installed the board can interact with its various parts, such as the GPS, LCD screen, I/O ports, switch, and buttons. In terms of system flow, the first important part of the user interface system is the on and off switch. Mounted on the right side of the box, the switch leaves an open circuit between the positive and negative leads of the battery in the cab when it is in the OFF position. We decided to use a separate battery with a common ground in the cab due to the large amounts of current that the generator often draws out of the battery when signals need to be turned on. Also, this eliminates the need to run a positive wire from the cab to the back of the truck. Another reason the battery from the generator was not chosen was because of the excessive power it supplies. When the low internal impedance, coupled with the 12 volt signal enters the main board, the 5 V voltage regulator gets very warm and can cause temperature issues with other parts on the board. Using a 9 volt battery in the cab, with a wire connecting the ground of the 9 volt to the system ground was the safer choice for our system.

To power up the main board, the switch is flipped to the ON position. At this point, the program that has been installed on the microcontroller will start to execute.

2.4.4 LCD Screen

Once the program begins to execute, a few initialization routines occur. First, the NHD-0420D3Z-FL-GBW LCD screen is configured with its appropriate SPI (Serial Peripheral Interface) commincation pins. On pin RC2/SCK., the program configures an acceptable clock rate for the LCD using the Timer2 module of the PIC18F4620. The MISO (master input slave output) pin on RC4/SDI was not necessary to wire or program because the LCD module is incapable of transmitting signals. However, it is initialized as an input pin for future use in SD card communication. The MOSI (master out serial in) pin on RC5/SDO is set as an output to send information and commands to the LCD. Also, a chip select is required. For the LCD screen, RC1 is made an output pin in order to send a chip select signal of 0 when communication with the LCD is required.

After the LCD is initialized, the GPS is initialized. This device uses EUSART (Universal Asynchronous Receive Transmit) communication to send and receive signals. The EUSART initilization requires a few steps. First the baud rate must be set. The baud rate for our Inventek Systems ISM300F2-C4 GPS module is 4800, so the program uses the specified formula in the PIC18F46209 manual to set this baud rate through the SPBRGH:SPBRG registers. After this is set, the EUSART transmit and receive registers are initialized. The pins for EUSART commincation are also on Port C. Pin RC6 is the microcontroller transmit pin which is connected to the GPS receive pin. This pin is cleared and set as an output by the microcontroller. Pin RC7 is set to an input, as it is the EUSART receive pin, connected to the GPS trasnmit pin. Once these registers and ports are confiured, the program sends information to the GPS. This information involves a series of three strings. The first string initializes the GPS to send and receive information. The second two strings are sent to turn off the unwanted data that the GPS will start to transmit upon beginning communication. The signals that are turned off are the GPGSV (Satellites in View) and GPGSA (Active Satellies) and the signals that are kept on are the GPGGA (Global Posistioning System Fixed Data) and GPRMC (Recommended Minimum Specific Data). By transmitting only the GPGGA and GPRMC information, less communication is done with the GPS module and only relevant information is sent.

2.4.6 User Buttons

2.4.6.1 Schematic

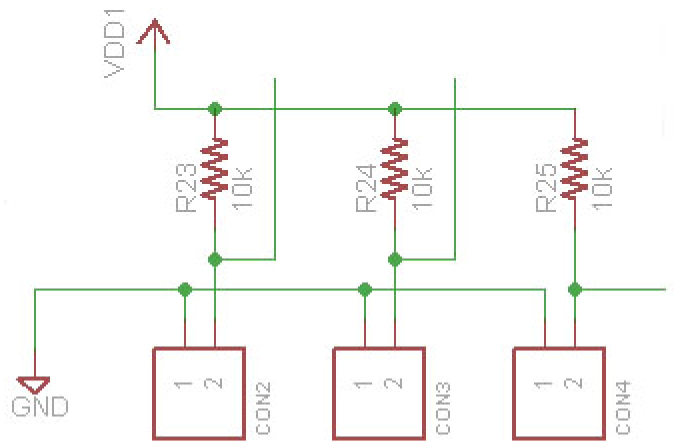


Figure 9 Button Schematic

The next initialization that occurs is for the buttons. Button pins RB0,RB1, and RB2 on the microcontroller are configured as input pins to receive button presses. These buttons are hardwired on the main PCB board to be normally at a low level of 0 V, and when pressed, the button completes the circuit and sends 3.3 V to the pin. Initially, the buttons all utilized the external INT0,INT1, and INT2 interrupts. The other interrupts in the program are all much more important, and it was decided that a button interrupt would not be used for the scrolling buttons. However, button number two, our 'Begin' button was left as an interrupt because it is solely for initializing the driving sequence, and does not need to be called upon after the user presses it for the first time. Buttons 0 and 1, the scrolling buttons, use a plain input signal, and are polled throughout the main loop to check their value and determine if they are being pressed or not.

After all of the initialization features occur, the program welcomes the user to the system with a greeting that reads, "Welcome to the New Hybrid Electric Vehicle Experience". This message holds for a delay of two seconds. Next, an instruction appears on the screen that asks the user to press the 'Begin' button to start the driving sequence. This feature was added to insure that the generator did not suddenly start without the users knowledge. When the program starts running, it can turn the generator on as soon as it enters the main loop. Requiring this button to be pressed to start the flow of the control subsystem makes the event of the loud generator turning on less alarming to a user who doesn't know what to expect.

After the 'Begin' button gets pressed, the program advances to where the Timer0 module gets enabled. This module is for the user interface solely. This timer causes an overflow in the 16 bit value of Timer0 every 1.667 seconds. When this interrupt occurs, the ISR (Interrupt Service Routine) is called and checks the value of the Timer0 interrupt flag, denoted as tmr0if in our program. Every time the Timer0 interrupt flag is set, a counting value, tmr0counter is incremented. For the first 15 seconds after the begin button is pressed, Timer0 goes through 9 iterations. The purpose of this is to allow time for the GPS to acquire satellites and to warm the engine using the glow plug. While this is occurring, the screen flashes messages that say precisely so. Using the modulo function, the value of tmr0counter is divided by two, if the remainder is 0, the screen displays “Please Wait, Engine Warming...” and if the remainder is 1, “Please Wait, Acquiring Satellites...” is displayed. Once this time is completed, the LCD displays “You may now begin your hybird driving experience”. At this point, the tmr0counter is noticed to be above 9 and the counter is reset to 0 and the variable ready is set to 1 to allow the program to finally advance into the control system main loop.

Once in the main loop, the control system starts to function. However, on the user interface side of the system, a function named ‘checkbuttons’ is called in each loop of the program. By calling this function at every loop where the program might get hung up while waiting for a signal to change thresholds, the ‘checkbuttons’ function can be nearly continuously called upon without worrying about cluttering up the ISR. The ‘checkbuttons’ function looks like the following.

```
void checkbuttons(void)
{
        if (button1 == 1)
        {
                delay_ms(40);     //wait to avoid multiple presses and debounce the button
                if (button1==1)
                {
                        count++;
                        display();
                }
        }
        if (button0 == 1)
        {
                delay_ms(40);
                if (button0==1)
                {
                        count--;
                        display();
                }
        }
        return;
}
```

As mentioned earlier, on the microcontroller side, pins RB0 and RB1 are always at a low level if the button is not pressed. On the button side, the 3.3 V signal from V_{DD} is connected to a resistor that goes to one of the leads of the button. If this button is pressed, the two leads of the button are connected and the microcontroller will have (3.3 - 300*I) go to the input of whichever button has been pressed. ‘checkbuttons’ uses volatile bits button1@PORTB.1 and button0@PORTB.0. If these bits are noticed to be at a high level when the function polls for them, a delay is implemented. This delay is a normal button debouncing delay to make sure the button doesn’t think it is being pressed more than once or to make sure the pin did not accidentally read a high level. After the delay, the button is checked again to verify that it is pressed and the count variable is either incremented or decremented depending on the button that had been pressed. After the count is altered, the program calls the function ‘display’ to give the user a new set of information on the screen. Depending on the value of the count, the user will see one of the following items in the bullets below.

-Ultra-capacitor Voltage

-Currents In and Out of the Ultra-capacitor Bank
-Power Consumption (Current out * Voltage)
-Output Signals
-Percentage of time spent charging the ultra-capacitors
-Speed
-Latitude
-Longitude

The display function will be described in further detail shortly.

2.4.3 Global Positioning System

2.4.3.1 Schematic

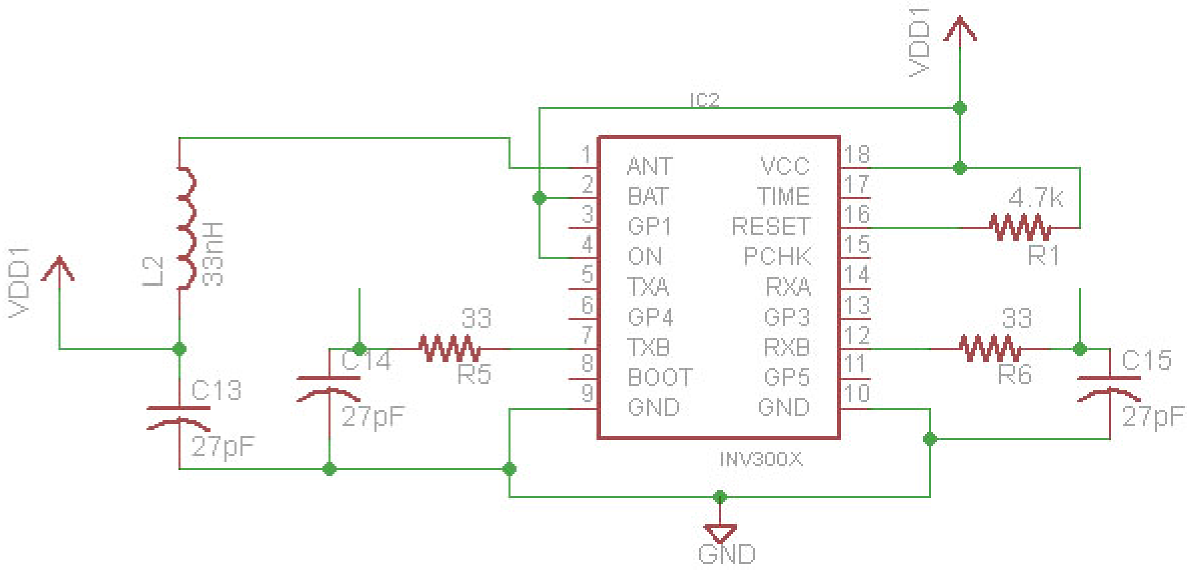
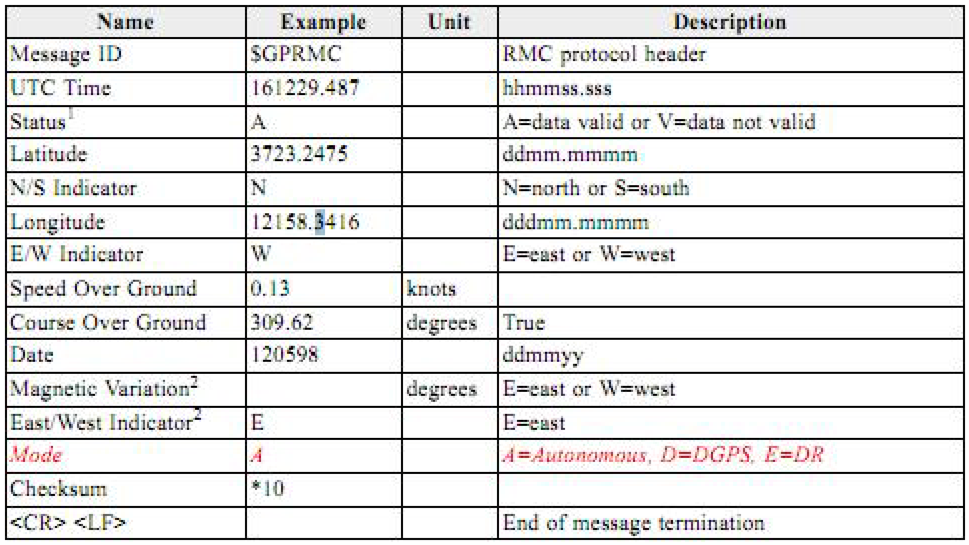


Figure 10 GPS Schematic

In order to display the speed, latitude, and longitude to the screen, the GPS module had to be used. As described earlier, this module is initialized by setting up the EUSART. After the Timer0 module was enabled in the code, the continuous receive signal for the EUSART module was also enabled, and the microcontroller starts to receive data from the GPS. For our GPS, we chose to communicate with the GPS via NMEA 0182 protocol. This communication technique was chosen because of the simplicity of the message. The GPS receives a string of characters from the GPS, and all that needs to be done is to parse the data, and store it in a string of variables. The other option was Binary SiRF communication which had less information when we researched the options, swaying us towards NMEA communication. An example of NMEA data is the following string that comes from the GPS and is received by the microcontroller.

$GPRMC,161229.487,A,3723.2475,N,12158.3416,W,0.13,309.62,120598,,N,*10

This string has different information between each set of commas. This particular string, the GPRMC string was actually the only string that was stored in our program because it gives all of the relevant information we need including speed, latitude, and longitude. The table below, from the NMEA Referrence Manual by SiRF Technologies shows the specific information that is between each comma in the string.



Name	Example	Unit	Description
Message ID	$GPRMC		RMC protocol header
UTC Time	161229.487		hhmmss.sss
Status[1]	A		A=data valid or V=data not valid
Latitude	3723.2475		ddmm.mmmm
N/S Indicator	N		N=north or S=south
Longitude	12158.3416		dddmm.mmmm
E/W Indicator	W		E=east or W=west
Speed Over Ground	0.13	knots	
Course Over Ground	309.62	degrees	True
Date	120598		ddmmyy
Magnetic Variation[2]		degrees	E=east or W=west
East/West Indicator[2]	E		E=east
Mode	*A*		*A=Autonomous, D=DGPS, E=DR*
Checksum	*10		
<CR> <LF>			End of message termination

As can be seen, the data mentioned above can be obtained by parsing this string. So, in order to obtain this information, the microcontroller had to receive these GPRMC strings from the Inventek Systems GPS module. To do so, the EUSART receive interrupt flag had to be enabled. Because the interrupt from the EUSART receive was not desired at first, it was not turned on until the drive sequence was ready. Once turned on, the strings coming from the GPS started causing the program to enter the interrupt service routine. If a byte was sent through the EUSART communication, the EUSART Receive interrupt flag, denoted as rcif in our software, was set. The ISR checks for this and if rcif is set, the function 'readGPS' is called. This function checks the ascii character that is coming through the communication port. If the character is a '$', then the EUSART Receiver knows that it is at the beginning of a new string. The function will continue to check the string until it gets to the first indicator that can separate the two strings that it is receiving (mentioned earlier, only GPGGA and GPRMC NMEA strings are being transmitted). If the fourth character in the string is 'G', the function will return to the ISR which will clear the interrupt flag. If the fourth character is a 'P', the function will continue to evaluate the string and store the relevant information about it. This relevant information includes, the position and the character at each position in the string and the position of the commas in each string. Once all this information is saved into a string, the function returns to the ISR, where it can clear the rcif and return to the main program. The code for the 'readGPS' function below shows what was just described above.

```
void readGPS(char read)
{
        gpsout = rcreg;
        if (gpsout == '$')
        {
                getstr = 1;
                commaR = 0;
                commaG = 0;
                num = 0;
                while (getstr==1)
                {
                        num++;
                        cren = 1;
                        while(!rcif);
                        gpsout = rcreg;
                        if (num==3)
                        {
                                if (gpsout =='G')
                                {
                                        GGA=0;
                                        return;
                                }
                                if (gpsout =='R')
                                        RMC=1;
                                getstr=0;
                        }
                }
        }
        while(RMC==1)
        {

                if (gpsout == ',')
                {
                        commaRpl[commaR] = num;
                        commaR++;
                }
                if (num==99 || gpsout == '*')
                {
                        gpsdataR[num] = ' ';
                        RMC = 0;
                        num = 0;
                        getspeed();
                        getLatR();
                        getLonR();
                }
                if (gpsout != '*')
                {
                        gpsdataR[num] = gpsout;
                        num++;
                }
                cren=1;
                while(!rcif);
                gpsout=rcreg;
        }
        return;
}
```

As can be seen above, if the received character is a ‘,’ the string ‘commaRpl[commaR]’ stores a value. This value, num, will help in parsing to note the position of commas in the RMC string. Also seen above is the if statement that checks if the string is too large or if the character received is a ‘*’. If the character is a ‘*’, this means that the NMEA output string is terminated.

The length restriction was put in for safety, as neither the GPGGA or GPRMC string should ever be more than 100 characters. At this point in the code, the string is obviously complete and the program can return to the main, i.e. the control system part of the programming. The final thing to comment on in this code is the functions that get called when the string is terminated. These functions parse the data in the string that has just been stored. In order to parse this data, we had to determine where in the string the data we were looking for was, and have a loop run through the values between the commas we desired and save this into a new string. Below is the parsing for latitude.

```
void getLatR(void)
{
        start = commaRpl[2]; start++;
        stop = commaRpl[4];
        for (start; start<stop; start++)
        {
                lat[latcount] = gpsdataR[start];
                latcount++;
        }
        for (latcount; latcount <21; latcount++)
                lat[latcount] = ' ';
        return;
}
```

From the table above, it can be seen that the latitude that is given is between the third and fourth commas of the GPRMC string, with the direction indicator between the fourth and fifth commas. In the 'readGPS' function, a number is assigned to each comma. So for example with the GPRMC string :

"$GPRMC,161229.487,A,3723.2475,N,12158.3416,W,0.13,309.62,120598,,N,*10"

the function 'readGPS' will only store:

"RMC,161229.487,A,3723.2475,N,12158.3416,W,0.13,309.62,120598,,N,"

with R corresponding to the variable 'num = 3'. So, the variable num is incremented everytime a character gets stored in the RMC string. Therefore, in the string above, the first comma will correspond to num = 6, the second to num =17 and the third to num = 19. So for the start and stop varaibles in 'getLatR', start = 6 and stop = 19. 'start' is incremented in order to avoid storing the comma in the latitude string, and a loop stores each of the characters betwee the commas into the string 'lat[latcount]'. Also here, because the satellites often takes time to acquire a signal, a second loop fils the string lat[] with ' ' characters if not enough information is stored in the overall string.

At this point, it is seen how the GPS data is parsed. Using this data, the user interface can now come back into play. Every time a GPRMC string is stored, the speed, latitude, and longitude are parsed, and ready to be displayed. Depending on what the 'count' variable is, based on the number of button presses, the user may now see speed, latitude, and longitude.

In order to decide what the user is going to display, the function 'display' seen below is called. This function takes the variable count, which gets altered by the button presses, and uses the modulo function on it to scroll between 8 different options of display information.

```
void display(void)
{
        disp = count % 8;
        switch (disp)
```

```
{
	case 0:
		LCD_clear();
		dispucvolts = ultracap_voltage4();
		LCD_printf(" Ultracap Voltage");
		LCD_cursor(0x46);
		LCD_voltage(dispucvolts);
		LCD_printf(" V");
		break;
	case 1:
		LCD_clear();
		LCD_printf("  Power Consumption");
		dispucvolts = ultracap_voltage2();
		dispcurrent2 = current_out();
		int volts;
		int current;
		volts = dispucvolts;
		dispucvolts = dispucvolts*15400;
		dispucvolts =dispucvolts/425;
		current = dispcurrent2;
		int power;
		power = current*volts;
		LCD_current(power);
		break;
	case 2:
		LCD_clear();
		dispcurrent1 = current_in();
		dispcurrent2 = current_out();
		LCD_printf("Current In: ");
		//LCD_current(dispcurrent1);
		LCD_printf("000.00 A");
		LCD_cursor(0x14);
		LCD_printf("Current Out: ");
		LCD_dec(dispcurrent2);
		LCD_printf("A");
		break;
	case 3:
		LCD_clear();
		LCD_printf("Speed");
		LCD_cursor(0x45);
		LCD_speed();
		break;
	case 4:
		LCD_clear();
		LCD_printLon();
		break;
	case 5:
		LCD_clear();
		LCD_printLat();
		break;
	case 6:
		LCD_clear();
		LCD_printf("   Output Signals");
		LCD_cursor(0x40);
```

```
                        if (latb.3==1) LCD_printf("     Starter On");
                        if (latb.3==0) LCD_printf("    Starter Off");
                        LCD_cursor(0x14);
                        if (latb.4==1) LCD_printf("   Fuel Valve On");
                        if (latb.4==0) LCD_printf("   Fuel Valve Off");
                        LCD_cursor(0x54);
                        if (late.0==1) LCD_printf("    Glow Plug On");
                        if (late.0==0) LCD_printf("   Glow Plug Off");
                        break;
                case 7:
                        LCD_clear();
                        LCD_printf("Charge Time:")
                        LCD_cursor(0x40);
                        char percentdisp;
                        percentdisp = chargetime_disp();
                        LCD_dec(percentdisp);
                        LCD_printf(" %");
                        break;
                default:
                        LCD_clear();
                        LCD_printf("DEEFAWLT");
                        break;
        }
        if (count == 0)
                count = 1000;
        return;
}
```

Based on the value of 'disp' the switch statement will perform the necessary functions to dislpay the desired messages to the screen. At this point, the extent of the user interface has been described. Further information about the user interface can be obtained from looking at the code files in the libraries titled "button_lib.c", "GPS_parse_lib.c", and "LCD_lib.c". A block diagram for the programming flow of this section can be seen below.

2.4.5 SD Card

In an effort to save GPS and energy data to the microcontroller for research purposes, we decided to try and write to an SD card. Unfortunately we were unable to secure functionality of the SD card in time, and it was thus relegated to being a future enhancement.

2.4.5.1 SD Card Hardware

The SD card is a non-volatile memory card format that provides a cheap and portable way of storing large amounts of data. The SD card interface is located on Port C of the microcontroller and utilizes its SPI functionality, where pins C3-C5 are coupled with one of its two Master Synchronous Serial Ports. These three pins handle communication with the microcontroller, with C2 connected to Pin 1 on the interface and holding the responsibility of enabling operation of the SD card. The other pins on the card handle power and binary operations. The presence of pull up resistors indicates that communication with the SD card occurs when the pins are in a low-voltage state.

2.4.5.2 Schematic

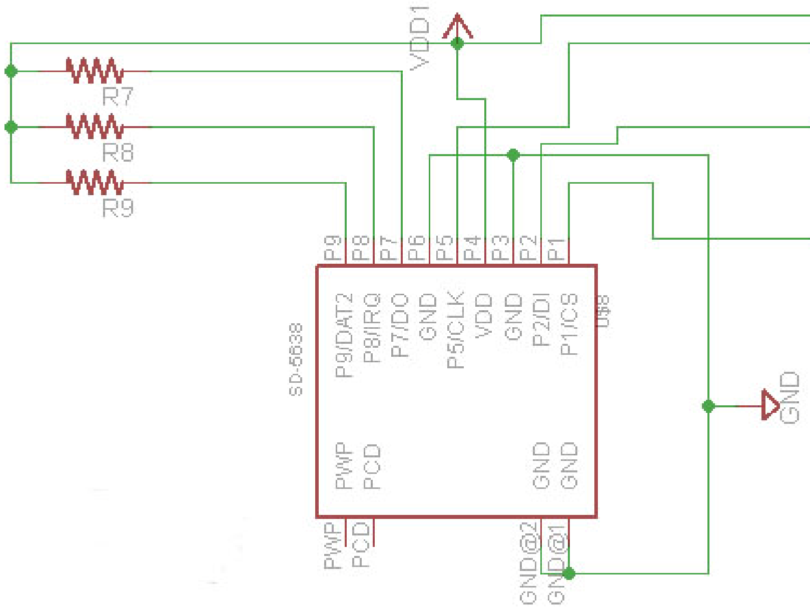


Figure 11 SD Card Reader Schematic

2.4.5.3 SD Card Software

Due to time constraints and difficulty in understanding the file system, the software for the SD card was modified with the permission of Rob Jones of 2009-2010 Team LED Zeppelin. The new code would, in addition to initializing the interface and reading from the card, write a file ‘test.txt’ each time the microcontroller was reset and store the information gathered during a drive cycle.

Two initialization functions, spi_init() and sd_init, establish the SPI interface on PORT C of the microcontroller and prepare the SD card to send and receive data. Also, the code sets a 512 byte block length, which amounts to how much data is sent to the SD card during a read or write command. Read and write commands send the appropriate commands to the SD card and then use a buffer to gauge the response of the card. More important are functions to traverse the FAT file system on the SD card. The basic storage unit in this file system is called a sector, which are 512 bytes each and combine to form clusters. Since memory is mapped in a way resembling a linked list, a cluster need not be made up of linear sectors in memory. Instead, each sector in a cluster points to the location of the next sector that stores part of the file. The sd_fs_init() function is used to calculate and store important offsets in the file system, including the number of sectors per cluster and the offset of the root directory in a partition. These variables clarify the issue of knowing where one is within the file system. Get_offset() provides the address in the SD card memory for a given folder and file.

In addition to modifying the aforementioned functions, new functions were developed in order to try and write a file to the SD card. First is the get_free() function, which searches the file system for an open cluster by going through the FAT table and checking whether a given cluster has 0x0000 in its entry. This value confirms an empty cluster, and the number of the cluster returns for further use in creating a file. Speaking of creating a file, the function create_file() passes in the number of the cluster to be written to and then traverses to that spot in the table. The put_block() function is utilized here to write information about the file in the FAT. An eight byte character array containing the name of the file is written to the FAT to provide file information, and subsequently the function traverses to the data section of the disk where the contents of the file can be written to the previously known cluster. Finally, a file_append() function is provided so that data from a drive cycle can be periodically added to the test file. This function traverses the FAT table and goes through the file one sector at a time until the end of the file is reached. At this point, the last sector with data is provided with a new value in the FAT table to point to so as to indicate a new end of file. After using the get_free() to learn where to append to the file, the function traverses to the appropriate data sector of the SD card and writes the new data to the file.

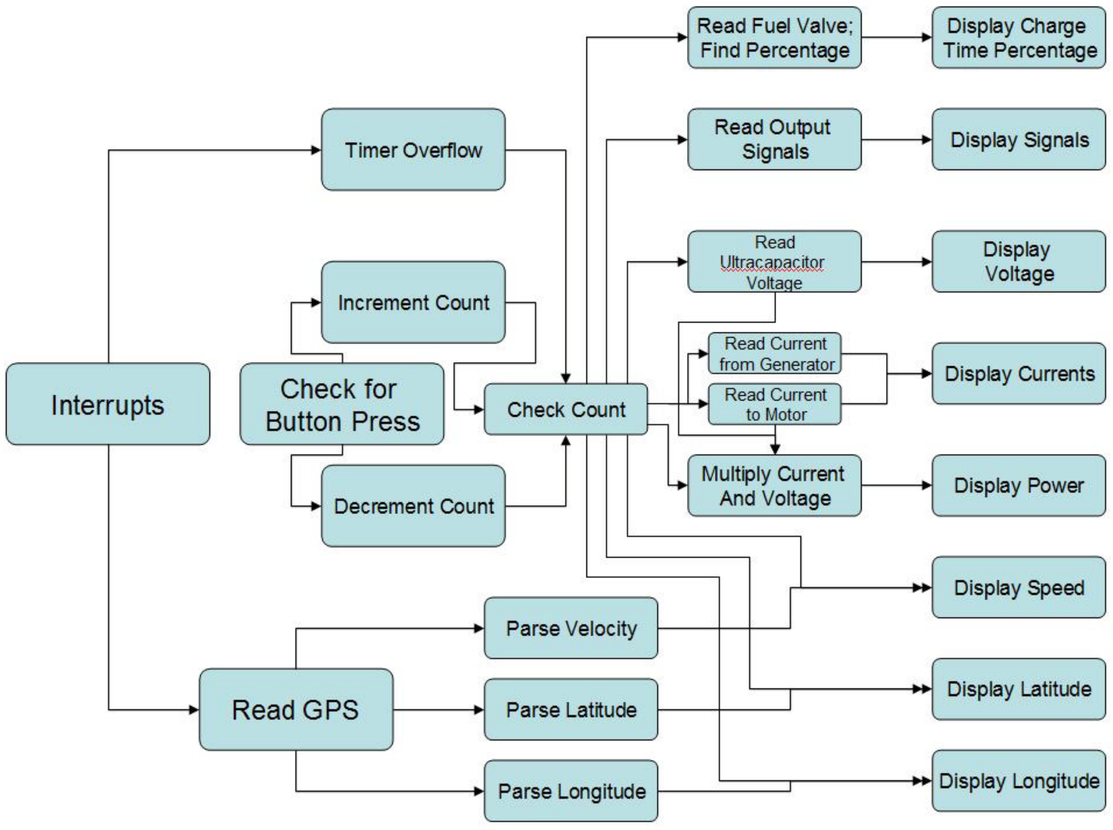


Figure 12 Interrupt Block Diagram

2.5 Interfaces and Sensors

2.5.1 Current Sensors

There are two HASS 200-S current sensors capable of detecting up to plus-minus 200 amps in our system. One HASS 200-S is located on the power-line between the ultra-capacitor bank and the rectifiers, and the other is located on the power-line between the ultra-capacitor bank and the DC driver motors. These current sensors output an analog voltage between 0 and 5 volts, with the voltage varying according to the amount of current going through the sensor. The analog voltage from the sensors is sent to pins A0 and A2 of the PIC and converted to a digital form by the PIC's A/D converter. Because the PIC can only accept input voltages between 0 and 3.3 volts, the current sensor output voltage is first multiplied by a 9.5:6.2 input:output ratio of its original value and then sent through a unity-gain buffer before arriving at the input pins x and y. The HASS 200-S current sensors were chosen because we knew that there would be up to 100 amps of inrush current passing through the power-lines when the capacitors were just being recharged, and as few as zero amps when the capacitors were not being recharged at all. The 200 amp sensors were chosen just because if any unexpected current spikes did occur in the subsystem we wanted to be able to capture them and not have our sensors peak at the normal 100 amps.

2.5.2 Voltage Sensing

The voltage sensing was done by simple voltage division on the main board with resistor dividers, then and analog to digital conversion by the PIC. This was done for the ultracapacitor voltage, and the voltage coming out of the MOSFET to signal that the starter can be shut off. This information is explained in detail in the control system subsystem mentioned earlier.

Other interfaces and sensing can be found explained in much detail in earlier subsystem operation descriptions.

3 System Integration Testing

The power control system was the first system tested. The first item that was tested was sending signals from the microcontroller. This was very simple, as all that was needed to do was set in software to make a pin an output, and apply a low or high signal. The next test involved the A/D converter. To test this, the code was written to perform an A/D conversion, a power supply was attached to the input pin desired for A/D conversion, and the value that the microncontroller read was displayed to the LED lights on the kit board. From here it was noticed that by using the ADRESH register, from 0 to 3.3 V, the value of the A/D conversion indirectly proportional to the input voltage. Now we knew how to perform A/D conversion and send signals, and it was time to test the switching circuitry. A lot of trouble was encountered with the MOSFETs that we were using, as we kept short circuiting the drain-source connection because of what was most likely the back emf of the inductive load. Careful considerations were taken to properly design the circuit so that surging currents or voltages from the back emf would not destroy the MOSFETs. Also the MOSFETs were increased in capacity from a rating of 40 A to a rating of 80 A and 100 V. These MOSFETs, once final testing was done proved very sufficieient in withstanding the wear and tear of the starter and fuel valve solenoids. After the circuitry was built succesfully, the software was loaded to the PIC. The software set lower and upper threshold for voltage which was simulated by a power supply. The software also accounted for the AC signal from the generator which was simulated by a simple wall plug outlet. When the voltage was low, the software was to turn on signals to the starter and fuel valve. When the plug was inserted into an outlet, the starter signal was to shut off, and when the voltage on the power supply was ramped up high enough, the fuel valve signal was to shut off. After this, the supply voltage would be pulled down to the lower threshold and the flow of the system was supposed to continue. By using the Timer0 interrupt on the PIC microcontroller, LED lights would check the level of the signals every 0.2 seconds. By using the lights, we were able to visually see when signals turned on and off rather than have to hold a multimeter on them at all times. Once the final circuitry was in place, a starter motor with only a voltage applied to the solenoid was attached to the source of the MOSFETs and it worked effectively during testing.

For the user interface, the testing involved three segments. First, the LCD screen, then the buttons, and finally the GPS. To test the LCD, the proper registers were found in the PIC18F4620 datasheet and New Haven Display LCD manual for configuration. Code was written to simply write messages to the screen using the library created by Dr. Schafer on the EESD wiki page. After this was accomplished, functions were written to adapt Dr. Schafers library to our SPI interfaced LCD screen. The functions were mostly commands designated by the New Haven Display LCD manual, which could clear the screen, move the cursor, dim the backlight, etc. To test the LCD incorporated with the A/D converter, a pin was configured with AD conversion capability, a voltage was applied to the pin, and the software was written to display the value of the conversion on the LEDs and the LCD screen using a function mimicked from Dr. Schafer's library for displaying a char type to the screen. The test worked effectively.

The next test was for the buttons. For these, we initially used the INT0 interrupt to detect an external interrupt on PORTB.0. The button was tied to the pin on one side, and to a resistor connected to 3.3 Von teh other side. The INT0 interrupt was configured to have the interrupt enabled, and trigger on a rising edge as the pin was normally held low. Every time the interrupt occurred, a counter was to be increased, and in the main, the value of the count was set to 'late', the port on the kit board microcontroller with the LEDs attached to it. This test proved to be fine. After this, the LCD screen and the button pressing was integrated to display different messages using a C switch statement. The A/D conversion was incoporated with this to display the voltage being input to a pin by a power supply, and a function was written to scale the voltage back to its actual value based on the 0-255 conversion value. This test was also successful and showed that we were able to obtain real time values, and cycle through the values with a button and display them to the LCD screen.

To test the GPS, the EUSART communication was initialized on the microcontroller. Using the techniques learned in the first half of the semester, we were able to use the multiple EUART pins on the PIC18F6722 to communicate with the GPS, and have the output of the GPS, the information on recreg1, display to the terminal of the IDE Sourceboost. By doing this, we were able to detect the NMEA strings coming from the GPS. After this, the next step was to integrate this into our main program by using the EUART receive interrupt. To test this, a function, 'readGPS' was created to be called every time an interrupt occurred. This function would read the NMEA strings and store them in an array of char values. Then, once the last char value was stored, the main program would receive a signal to display the entire string that was just stored to the GPS. This was done successfully. Next test involved parsing the GPS. This time, instead of sending a signal to main to print the NMEA string that was just stored, a signal was sent to store the ASCII char values in between the 7th and 8th commas. After this was done, the main was given the signal to print the NMEA string to the LCD screen, followed by the ASCII char values that had been stored in char string for speed. This test worked, as the LCD showed the NMEA string and followed immediately by the same values that were in between the specified commas in the string.

The final test involved running the control subsystem in main, and using the interrupts to detect GPS info. Also, a function to poll for button presses was strewn throughout main at every point where main may get caught in a loop. The screen was able to display different messages upon button presses, including strings from the GPS, while the LEDs showed which signals were on, whether they be the starter or the fuel valve or the glow plug.

4 Users Manual/Installation Manual

4.1 Control System Installation

In order to install the series hybrid control system, a vehicle equipped with an electric motor, ultra-capacitor bank, and generator must be supplied. As seen in the system block diagram, the control system is attached to the already existing power train. The control system is installed as a two part system. An auxiliary board and connecting sensors and cables must be installed first. Second is the main control unit that sits within the cabin of the vehicle. A set of cables must be connected from the auxiliary board to the control unit in the cabin.

Auxiliary Board Installation

In order to install the auxiliary board, a set of cables must be made that will connect to generator and various sensors. It is advised to install the auxiliary board as close the generator as possible. This will eliminate that need for extra long cables for the high current applications. The starter motor on the generator will likely require a high current input for starting, and thus a low gauge wire is recommended. The fuel valve and glow plug will not be sourcing as much current and thus can be made with a higher gauge wire. The board must be placed in a secure and dry location. A water tight box is recommended. Measure the length of each cable coming from the auxiliary board to each component. The cables that need to made: generator starter, fuel valve, glow plug, ultra capacitor back voltage, two current sensors, cables to the battery, a data cable to the main control unit in the cabin, and an AC cable to the output of the generator. The auxiliary board has screw terminals to securely fasten the cables to the board. Also, the board has screw holes to securely lock the board to its encasement.

Main Control Box Installation

The main control unit is placed in the cabin of the vehicle. It should be placed next to the driver because the unit will display relevant data for the driver. The necessary cables for the unit are the battery cable and data cable from the auxiliary board. Caution must be given to make sure the main control unit and auxiliary board are connected to the same ground. The whole system must be placed on the same ground.

4.2 Control System Setup

When the whole system has been connected to the power train and various sensors, then the control unit can be programmed. Using the code that comes with the product, a user can freely change parameters of the code to customize their setup. Simply use a compatible programmer to program the microcontroller with the code. Programming is a necessary step with setup as the program must be compatible with all attached sensors. Once the control system has been programmed once, it will retain the program for future use. When the control has been programmed, then the unit is functional. Simply turning the unit ON will activate the power train control system. The user is ready to begin driving.

4.3 Identifying Problems

Common problems that can arise while using the control system include: loose cable connections, errors in the code, and improper cable connections. The user should place an external multimeter on the ultra capacitor bank to measure the voltage during first use. This will check to see if the control unit is reading correct voltages. If the generator activates or does not activate according to the voltage readings, then there are possible cable connection issues. The user should scroll through the data on the LCD and check that each reading is correct. Current, voltages, and power can be measured with external meters. During first time use of the system all parts should be checked.

4.4 Troubleshooting

When the user finds issues arising with the generator not responding correctly to the voltages on the ultra-capacitor bank, the user should first check all cable connections. All connections must be secure and connected to the correct input or output. If the issues continue to arise, then the microcontroller code must be reviewed. Sometime errors can arise within code after alterations. The original code should never be modified to alter proper operation of the system. Only code that determines vehicle parameters should be modified by the user.

5. To-Market Design Changes

Although our project focused on data collection, information display, and the automated switching system, in this section we will talk about changes to the entire hybrid vehicle system, as it makes little sense to only sell part of a vehicle to customers.

While fairly satisfied with the functionality of the system as a whole, there are many individual changes that can be made to make the system more user-friendly and turn it from a research vehicle to a retail automobile. Instead of sitting in the bed of the truck, the entire power-train should be transferred back under the hood of the vehicle. Not only will this protect the power-train from rain, rust, and short-circuits, it will also free up the truck bed for use. To reduce vibrations from the generator to the vehicle the generator should have rubber mounts placed underneath it. Also, an alternator on the generator to recharge the 12-volt battery and an upgraded starter motor to reliably handle the large amount of starts and stops the generator will undergo are both necessary. Capacitance should be improved in order to lengthen the amount of time that the vehicle can run solely off of the ultra-capacitors and decrease the amount of times the engine has to undergo a start-stop cycle.

Several changes can be made to make the driving experience more user-friendly. Two changes would be to use a larger display screen and integrate both the display screen and buttons into the dashboard, where they are more accessible than sitting on the seat by the user. To allow the user to better understand their power consumption and fuel usage graphs and displays showing power and fuel use over time and different types of terrain can be view options on the LCD screen. To store the information about power and fuel use, solid-state memory accessible via a USB port is preferable to an SD card because of the USB's widespread prevalence.

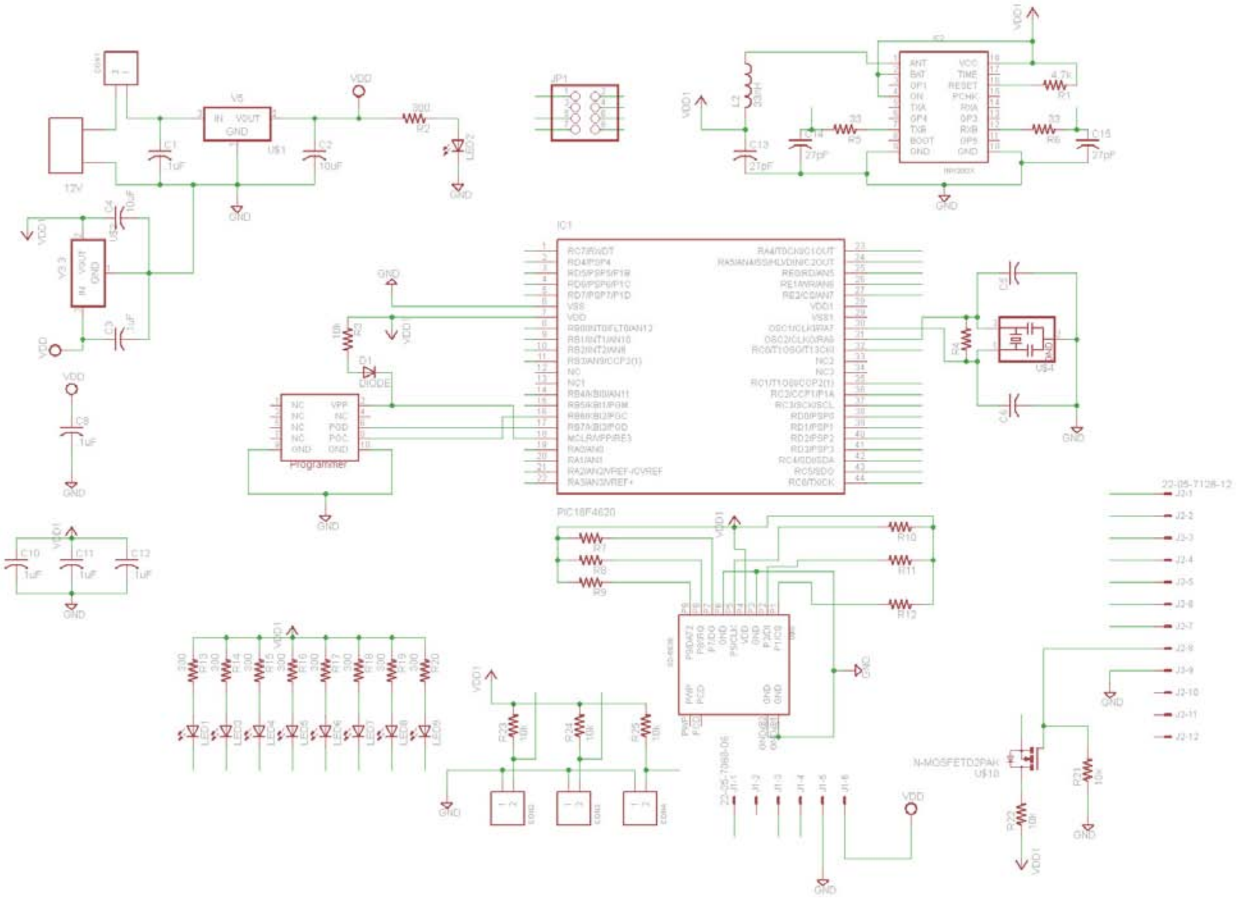
6 Conclusions

Hybrid vehicles continue to lead at the front line of automotive fuel-economy and Dr. Bauer is looking to make further improvements by investigating the series-hybrid setup using ultra-capacitors in place of the conventional batteries. Dr. Bauer assigned our team several tasks to accomplish to improve the research setup on his series hybrid. Our team attempted to accomplish all of these tasks and emerged mostly successful. We were able to automate the start-stop function of the generator based on the voltage of the ultra-capacitor bank. We also constructed an interface for the driver to control the hybrid system and view relevant information from an LCD display. Although we were unable to get it to work, we made great strides in integrating an external data-storage device into the system, an SD card. Fortunately, the SD card was always viewed in the context of a nice addition to the project that would aid research, but was not a necessary item to conduct the hybrid-vehicle research. So in the context that we were able to greatly aid Dr. Bauer in his research while not accomplishing every stretch goal, we still view ourselves as being overall successful with our senior design project.

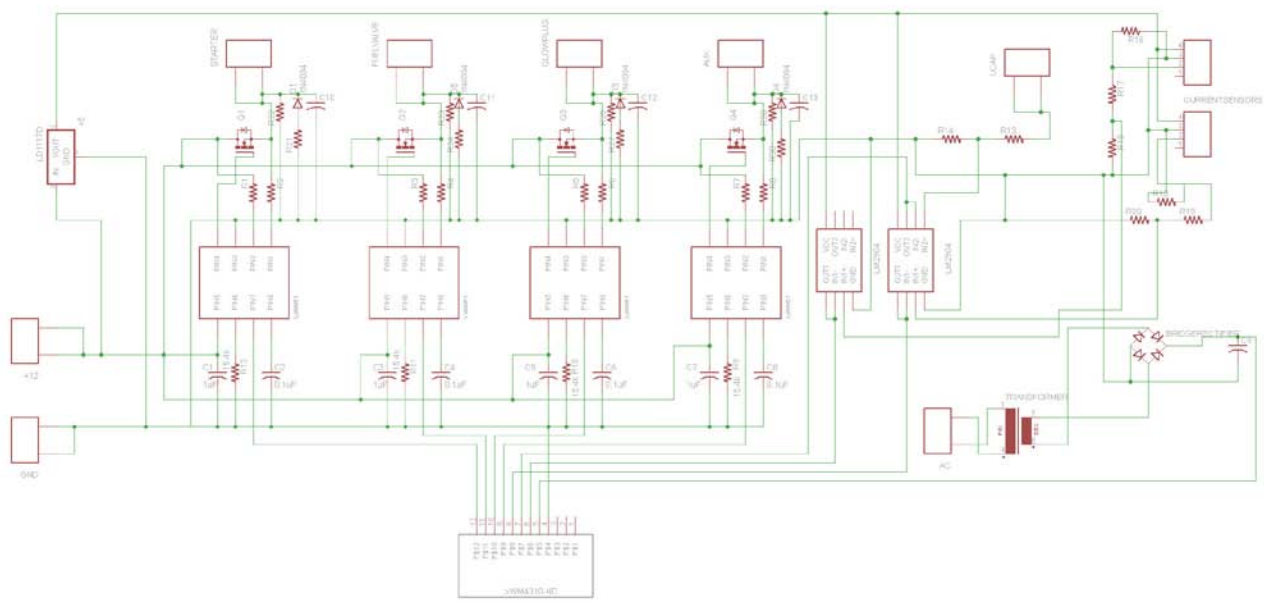
7 Appendices

Schematics:

Main Board:



Auxiliary Board:



Software Listing:

Microcontroller Code:

Main Code:

```
#include <system.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#pragma DATA _CONFIG1H, _OSC_HS_1H
#pragma DATA _CONFIG2H, _WDT_OFF_2H
#pragma DATA _CONFIG4L, _LVP_OFF_4L & _XINST_OFF_4L
#pragma DATA _CONFIG3H, _MCLRE_ON_3H
#pragma CLOCK_FREQ 20000000

#include "button_lib.h"
#include "LCD_lib.h"
#include "A2DConv.h"
#include "generator_commands.h"
#include "GPS_parse_lib.h"

//Global Interrupt Bits//
volatile bit peie@INTCON.6;
volatile bit gie@INTCON.7;
volatile bit ipen@RCON.7;
//Timer 0 Interrupt Signals//
volatile bit tmr0if@INTCON.2;
volatile bit tmr0ie@INTCON.5;
//Button Interrupt Signals//
volatile bit int2ie@INTCON3.4;  //B2
volatile bit int2if@INTCON3.1;
volatile bit intedg2@INTCON2.4;
//A2D Conversion Interrupt Signal//
volatile bit adif@PIR1.6;
volatile bit adie@PIE1.6;
//A2D Convert Variables//
unsigned char ultracap_voltage;
unsigned char gen_output;
int monitorucs;
unsigned char dispucvoltage;
//GPS Receive Interrupt Signals//
volatile bit rcif@PIR1.5;
volatile bit rcie@PIE1.5;
volatile bit rcip@IPR1.5;
//GPS Send/Receive Signals
volatile bit txen@TXSTA.5;
volatile bit trmt@TXSTA.1;
volatile bit cren@RCSTA.4;
//Generator Control Signals//
volatile bit starte@LATB.3;
volatile bit fuelvalv@LATB.4;
volatile bit glowp@LATE.0;
unsigned char monitor_ultracaps;
//Initialization Variables;
bool go;
bool ready;
int tmr0counter;
char interruptson;

void initialize(void)
{
	//Enable Global Interrupts//
	peie = 1;
	gie = 1;
	ipen = 1;
	//Set LED's For Output//
	trisd = 0;
	latd = 0;
	//Call Initialization Functions//
	button2_init();
	A2D_init();
	gen_commands_init();
	GPS_init();
	LCD_init();
	//Initalize Variables//
	go = 0;
	ready = 0;
	tmr0counter = 0;
	monitorucs = 0;
```

```
}

void interrupt(void)
{
	if (go == 0)
	{
		if (tmr2if)
		{
			go = debounce3();
			tmr2if = 0;
			button_init();
		}
	}
	if (tmr0if)
	{
		tmr0counter++;
		if (ready ==1)
		{
			display();
		}
		if (ready==0)
		{
			LCD_clear();
			LCD_printf("Please Wait,");
			LCD_cursor(0x40);
			if (tmr0counter%2==0)
				LCD_printf("Acquiring Satellites");
			if (tmr0counter%2==1)
				LCD_printf("Engine Warming......");
			if (tmr0counter>9)
			{
				LCD_clear();
				LCD_printf("You may now begin");
				LCD_curosr(0x40);
				LCD_printf("your hybrid driving");
				LCD_cursor(0x14);
				LCD_printf("sequence".);
				ready = 1;
				tmr0counter=1;
				button_init();
			}
		}
		tmr0if=0;
	}
	if (rcif)
	{
		redGPS(1);
		rcif = 0;
	}
	return;
}

void main()
{
	initialize();		//Initialize all libraries
```

```
        LCD_printf("Welcome to the New");//Display Message to user
        LCD_cursor(0x40);   LCD_printf("Hybrid Electric");//Second line of cursor is at
                                                //0x40, so need to account for this
        LCD_cursor(0x14);   LCD_printf("Vehicle Experience");
        delay_s(2);
                            LCD_printf("Please Press 'Begin'");
        LCD_cursor(0x40);   LCD_printf("Button to Start Your");
        LCD_cursor(0x40);   LCD_printf("Driving Sequence");

        while (!go);
        glowplug = 1;//Warm up the engine
        cren = 1;           //Accept receptions from the GPS
        //Bits for Timer 0 Control Reg for interrupt every 1.677 seconds
        //Bit7 = 1 : Timer0 Enable
        //Bit6 = 0 : Timer0 16-bit Counter
        //Bit5 = 0 : Timer 0 clock source select CLKO
        //Bit4 = 1 : Edge Select (high-to-low)
        //Bit3 = 0 : Prescalar Bit is assigned
        //Bit2:0 = 110 : Prescale at 1:128
        t0con = 10010110b;
        tmr0ie = 1;
        while(!ready);      //Wait for engine to warm and satellites to acquire
        rcie = 1;           //Turn on the EUSART interrupt flag bit
        while(1)
        {
                //Check Voltage on Ultracaps and Charge if Necessary//
                ultracap_voltage = ultracap_voltage3();
                delay_ms(10);
                checkbuttons();
                if (ultracap_voltage < 159)
                {
                        start_charge();     //Turn on fuelvalve and starter signals
                        check_starter();    //Turns off starter once generator is on; Will
                                                //not continue until starter signal
goes off
                        monitor_ultracaps(1); //Monitor ultracap stack value and turn off
                                            //generator at full charge
                }//end if(uc<130)
        }
        return;
}
```

A/D Conversion Library and Header:

```
#include <system.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#include "button_lib.h"
#include "LCD_lib.h"
#include "A2DConv.h"
#include "generator_commands.h"
#include "GPS_parse_lib.h"

//A_D Conversion Bits//
```

```
volatile bit ad_convert@ADCON0.1;       //GO_DONE status bit
volatile bit ad_enable@ADCON0.0;        //ADC Enable bit
volatile bit csb0@ADCON0.2;             //Channel Select Bit 0
volatile bit csb1@ADCON0.3;             //Channel Select Bit 1
volatile bit csb2@ADCON0.4;             //Channel Select Bit 2
volatile bit csb3@ADCON0.5;             //Channel Select Bit 3
//A_D Conversion Signals//
unsigned short current_1;        //Port A0
unsigned char uc_voltage;        //Port A1
unsigned char uc_voltage2;
unsigned short current_2;        //Port A2
unsigned char gen_out;                  //Port A3

//Initialize A2D Conversions//
void A2D_init(void)
{
       //adif = 0;
       //adie = 1;
       trisa.0 = 1;
       trisa.1 = 1;
       trisa.2 = 1;
       trisa.3 = 1;
       //ADCON0 Register Bits
              //Bit 7-6 Unimplemented
              //Bit 5-2 Voltaile Channel Select Bits
              //Bit 1 Volatile Status
              //Bit 0 ADC Enable Bit
                     adcon0= 00000000b;
       //ADCON1 Register Bits
              //Bit 7-6 Unimplemented
              //Bit 5-4 Voltage Reference(Chosen as Vref+=Vdd and Vref-=Vss)
              //Bit 3-0 Port Configuration Bits (1011 = AN0-AN3 Analog Inputs)
                     adcon1= 00001011b;
       //ADCON2 Register Bits
              //Bit 7 A/D Result Formal Select Bit (0=left justified)
              //Bit 6 Unimplemented
              //Bit 5-3 Acquisition Time Select Bit (111= 20TAD)
              //Bit 2-0 ADC Clock Select (101= Fosc/16)
                     adcon2= 00111101b;
       current_1 = 0;
       uc_voltage = 0;
       uc_voltage2 = 0;
       current_2 = 0;
       gen_out = 0;
       return;
}
//ADConversion for Current into Ultracaps
unsigned short current_in(void)
{
       //Channel Select Bits ADCON0 5:2 for Pin A0//
       csb3 = 0; csb2 = 0; csb1 = 0; csb0 = 0;

       ad_enable = 1;
       ad_convert = 1;
       delay_us(10);
```

```
	while (ad_convert);//Wait For Conversion to Finish
	ad_enable = 0;
	unsigned short temp = 0xffff;				//Need to be more 
specific for current A/D so use all 10 bits
	temp &= adresh;
	//Store MSB first and shift right 8 bits.
	current_1 = ((temp << 8) | (adresl&0x00ff));	//Store LSB and create short that 
is shifted left 6 bits.
	current_1 = ((current_1 >> 6) & 0x03ff);		//Shift back 6 bits to 
right so 10 bit A/D conversion can be passed.
	return current_1;

//ADConversion for Ultracap Voltage//
//Used to sample varying rectified signal when Ultracaps are charging//
unsigned char ultracap_voltage(void)
{
	csb3 = 0; csb2 = 0; csb1 = 0; csb0 = 1;
	short sample =0;
	char volts;
	for (int multi = 0; multi<10; multi++)
	{
		ad_enable = 1;
		delay_us(10);
		ad_convert = 1;
		while (ad_convert);//Wait For Conversion to Finish
		ad_enable = 0;
		volts = adresh;
		sample = sample + volts;
		adresh=0;
	}
	sample = sample/10;
	uc_voltage = sample;
	return uc_voltage;
}
//Used to not confuse main and interrupt calling same function.
unsigned char ultracap_voltage2(void)
{
	csb3 = 0; csb2 = 0; csb1 = 0; csb0 = 1;
	short sample2 =0;
	for (int multi = 0; multi<10; multi++)
	{
		ad_enable = 1;
		ad_convert = 1;
		delay_us(10);
		while (ad_convert);//Wait For Conversion to Finish
		ad_enable = 0;
		uc_voltage4 = adresh;
		sample2 += uc_voltage4;
		adresh=0;
	}
	sample2 = sample2/10;
	uc_voltage2 = sample2;
	return uc_voltage2;
}
//ADConversion for Current out of Ultracaps//
```

```
unsigned short current_out(void)
{
       //Channel Select Bits ADCON0 5:2 for Pin A2//
       csb3 = 0; csb2 = 0; csb1 = 1; csb0 = 0;

       ad_enable = 1;
       ad_convert = 1;
       delay_us(10);
       while (ad_convert);//Wait For Conversion to Finish
       ad_enable = 0;
       unsigned short temp = 0xffff;
       temp &= adresh;                                          //
       current_2 = ((temp << 8) | (adresl&0x00ff));
       current_2 = ((current_2 >> 6) & 0x03ff);
       return current_2;
}
//ADConversion for Generator Signal//
unsigned char gen_output(void)
{
       //Channel Select Bits ADCON0 5:2 for Pin A3//
       csb3 = 0; csb2 = 0; csb1 = 1; csb0 = 1;

       ad_enable = 1;
       ad_convert = 1;
       delay_us(10);
       while (ad_convert);//Wait For Conversion to Finish
       ad_enable = 0;
       gen_out = adresh;
       adresh=0;
       return gen_out;
}
#ifndef _A2DCONV_H_
#define _A2DCONV_H_
#include<system.h>


//A2D Converstion Functions//

void A2D_init(void);
unsigned char ultracap_voltage(void);
unsigned char ultracap_voltage2(void);
unsigned short current_in(void);
unsigned short current_out(void);
unsigned char gen_output(void);

#endif //_A2DCONV_H_
```

LCD Library and Header:

```
#include <system.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#include "button_lib.h"
#include "LCD_lib.h"
#include "A2DConv.h"
```

```
#include "generator_commands.h"
#include "GPS_parse_lib.h"
volatile bit LCD_cs@LATC.1;                          //low true chip selects
//initialize spi for lcd
void LCD_init(void)
{
	//Setting Timer 2//
		//Bit7 = 0 : Unimplemented
		//Bit6:3 = 0000 : Postscale Bits, 1:1 Postscale
		//Bit2 = 1 : TMR2 on
		//Bit1:0 = 11 : Prescalar 1:1 ratio
	t2con = 00000100b;
	//Initially Clear all//
	sspcon1 = 00000000b;
	//Set Bits for I/O Port C
		//Bit7 = 1 : EUSART Recieve Input
		//Bit6 = 0 : EUSART Transmit Output
		//Bit5 = 0 : SPI Data Output cleared (MOSI)
		//Bit4 = 1 : SPI Data Input Auto-Controlled by SSPEN (MISO)
		//Bit3 = 0 : SCK (Master Mode=0) (Slave Mode=1)
		//Bit2 = 0 : Irrelevant
		//Bit1 = 0 : Chip Select Bit for SD Card
		//Bit0 = 0 : Chip Select Bit for LCD
	trisc = 10010000b;
	//Set Bits for SSP1STAT Register
		//Bit7 = 0 : SPI Master Sample Time in Middle of output time
		//Bit6 = 0 : clock select bit:transmit on idle to active transition
		//Bit5:1 = 0 : Used for I2C mode
		//Bit0 = 0 : SSP1BUF transmit complete bit when = 1
	sspstat = 00000000b;

	//Set Bits For SSP1CON1 Register
		//Bit7 = 0 : Write Collision Detect
		//Bit6 = 0 : Receive Overflow Indicator
		//Bit5 = 1 : SPI enable Serial ports
		//Bit4 = 1 : Clock Parity Select (1=idle-high;0=idle-low)
		//Bit3:0 = 0011 : (Master Mode, TMR2 output/2)
	sspcon1 = 00110011b;

	LCD_cs = 1;
	LCD_clear();
	return;
}
//Print Character to SPI LCD Screen by Sending 1 Byte through SPI//
char LCD_putchar(char data)
{
	LCD_cs = 0;
	sspbuf = data;              //BUF register sends out data
	while (!sspstat.BF);              //Wait for IF to say transmission/receive
complete
	LCD_cs = 1;
	delay_ms(1);
	return sspbuf;
}
//Print a character to the screen//
```

```
void LCD_printf(const char* text )
{
	char i = 0;
	while( *text != 0 )
	{
	LCD_putchar( *text++ );
	}
	return;
}
//Print the Speed from the GPS parsing//
void LCD_speed(void)
{
	int data = speedtopass();
	int decimal = decimal_to_disp();
	int tens;
	int ones;
	int tenth;
	int hundth;
	tens  = data/1000;  //Tens Place
	data  = data%1000;  //Remainder for Tens Place
	ones  = data/100;   //Ones Place
	data  = data%100;
	tenth = data/10;    //Tenths Place
	hundth= data%10;    //Hundreths Place

	switch (decimal)//to know where to place the decimal//
	{
		case 0:
			LCD_putchar(ones + '0');
			LCD_putchar(0x2E);
			LCD_putchar(tenth + '0');
			LCD_putchar(hundth + '0');
			break;
		case 1:
			LCD_putchar(ones + '0');
			LCD_putchar(0x2E);
			LCD_putchar(tenth + '0');
			LCD_putchar(hundth + '0');
			break;
		case 2:
			LCD_putchar(tens +'0');
			LCD_putchar(ones + '0');
			LCD_putchar(0x2E);
			LCD_putchar(tenth + '0');
			LCD_putchar(hundth + '0');
			break;
	}
	return;
}
//Print unsigned long with decimal in correct place//
void LCD_voltage(unsigned char voltage)
{
	unsigned long data;
	data = voltage*15400;
	data = data/425;
	int tenk;
	int onek;
	int hund;
	int tens;
	int ones;

	tenk = data/10000;  //Ten Thousands Place
	data = data%10000;  //Remainder for Thousands Place
	onek = data/1000;   //Thousands Place
	data = data%1000;
	hund = data/100;    //Hundreds Place
	data = data%100;
	tens = data/10;		//Tens Place
	ones = data%10;		//Ones Place

	if (tenk != 0)
		LCD_putchar(tenk + '0');
	if (onek != 0)
		LCD_putchar(onek + '0');
	LCD_putchar(hund + '0');
	LCD_putchar(0x2E);
	LCD_putchar(tens + '0');
	LCD_putchar(ones + '0');
	return;
}
//Print unsigned long with decimal in correct place//
void LCD_current(int data)
{
	data = data*6902;
	data = data - 800000;
	unsigned short hunk;
	unsigned short tenk;
	unsigned short onek;
	unsigned short hund;
	unsigned short tens;
	unsigned short ones;
	hunk = data/100000;
	data = data%100000;
	tenk = data/10000;  //Ten Thousands Place
	data = data%10000;  //Remainder for Thousands Place
	onek = data/1000;   //Thousands Place
	data = data%1000;
	hund = data/100;    //Hundreds Place
	data = data%100;
	tens = data/10;		//Tens Place
	ones = data%10;		//Ones Place

	if (hunk != 0)
		LCD_putchar(hunk + '0');
	LCD_putchar(tenk + '0');
	LCD_putchar(onek + '0');
	LCD_putchar(0x2E);
	LCD_putchar(hund + '0');
	LCD_putchar(tens + '0');
	LCD_putchar(ones + '0');
	return;
}
//Print unsigned short as decimal number//
void LCD_dec(unsigned short data)
{
	unsigned short tenk;
	unsigned short onek;
	unsigned short hund;
	unsigned short tens;
	unsigned short ones;

	tenk = data/10000;  //Ten Thousands Place
	data = data%10000;  //Remainder for Thousands Place
	onek = data/1000;   //Thousands Place
	data = data%1000;
	hund = data/100;    //Hundreds Place
	data = data%100;
	tens = data/10;		//Tens Place
	ones = data%10;		//Ones Place

	if (tenk != 0)
		LCD_putchar(tenk + '0');
	if (tenk >= 1)
		LCD_putchar(onek + '0');
	if (hund != 0)
		LCD_putchar(hund + '0');
	if (tens != 0)
		LCD_putchar(tens + '0');
	LCD_putchar(ones + '0');
	return;
}

//Print char as a decimal number//
void LCD_dec(unsigned char data)
{
	unsigned char hund;
	unsigned char tens;
	unsigned char ones;

	hund = data/100;
	data = data%100;
	tens = data/10;
	ones = data%10;

	if (hund != 0)
		LCD_putchar(hund + '0');
	if (hund >= 1)
		LCD_putchar(tens + '0');
	if (hund == 0 && tens != 0)
		LCD_putchar(tens + '0');
	LCD_putchar(ones + '0');
	return;
}

//put the cursor in a specific position
```

```
///////////COLUMN 1//////COLUMN 20////////
//Line 1/// 0x00   ////// 0x13     ////////
//LINE 2/// 0x40   ////// 0x53    ////////
//LINE 3/// 0x14   ////// 0x27     ////////
//LINE 4/// 0x54   ////// 0x67    ////////
//////////////////////////////////////////
void LCD_cursor(char position)
{
       LCD_putchar(0xFE);
       LCD_putchar(0x45);
       LCD_putchar(position);
       delay_ms(10);
       return;
}
//Clear Display and Place cursor at line 1 column 1//
void LCD_clear(void)
{
       LCD_putchar(0xFE);
       LCD_putchar(0x51);
       delay_ms(10);
       return;
}
//Set Backlight Brightness:Brightness Value between 1 and 8//
void LCD_brightness(char brightness)
{
       if (brightness<=8 && brightness >=1)
       {
              LCD_putchar(0xFE);
              LCD_putchar(0x53);
              LCD_putchar(brightness);
              delay_ms(10);
       }
       else
       {
              LCD_putchar(0xFE);
              LCD_putchar(0x53);
              LCD_putchar(0x04);
              delay_ms(10);
       }
}
#ifndef _LCD_LIB_H_
#define _LCD_LIB_H_
#include<system.h>


//LCD Functions//

void LCD_init(void);
char LCD_putchar(char data);
void LCD_cursor(char position);
void LCD_clear(void);
void LCD_printf(const char* text );
void LCD_speed(void);
void LCD_dec(unsigned char data);
void LCD_dec(unsigned short data);
void LCD_voltage(unsigned char voltage);
void LCD_current(int data);
void LCD_brightness(char brightness);

#endif //_LCD_LIB_H_
```

Generator Commands Library and Header:

```
#include <system.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#include "button_lib.h"
#include "LCD_lib.h"
#include "A2DConv.h"
#include "generator_commands.h"
#include "GPS_parse_lib.h"

//Generator Signals//
volatile bit starter@LATB.3;
volatile bit fuelvalve@LATB.4;
volatile bit glowplug@LATE.0;

unsigned char ucvoltage;
unsigned char gen_signal;

void gen_commands_init(void)
{
	trisb.3 = 0;
	trisb.4 = 0;
	trise.0 = 0;
	starter = 0;
	fuelvalve = 0;
	glowplug = 0;
	return;
}
//Start the Charging Cycle//
void start_charge(void)
{
	fuelvalve = 1;
	starter =1;
	return;
}
//Check to See if Generator is On and Turn off Starter//
void check_starter(void)
{
	while (starter == 1)
	{
		checkbuttons();
		gen_signal = gen_output();	//check to see if generator is
outputting						//voltage
		delay_ms(10);
		if (gen_signal > 220)		//if generator is outputting voltage,
						//starter can be turned off
```

```
                    starter = 0;
        }
        return;
}
//Check Ultracaps and Turn off Generator if Charging is Finished//
void monitor_ultracaps(int monitor_ultracaps)
{
        while (monitor_ultracaps == 1)
        {
                ucvoltage = ultracap_voltage();
                if (ucvoltage > 208)            //if ultracaps above threshold, stop
charging
                {
                        fuelvalve = 0; starter = 0; //turn off all signals
                        monitor_ultracaps = 0;      //exit checking/charging loop
                }
        }
        return;
}
#ifndef _GENERATOR_COMMANDS_H_
#define _GENERATOR_COMMANDS_H_
#include<system.h>

//Generator Commmands Functions//

void gen_commands_init(void);
void start_charge(void);
void check_starter(void);
void monitor_ultracaps(int monitor_ultracaps);

#endif //_GENERATOR_COMMANDS_H_
```

Button Library and Header:

```
include <system.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#include "button_lib.h"
#include "LCD_lib.h"
#include "A2DConv.h"
#include "generator_commands.h"
#include "GPS_parse_lib.h"

//Button Press Bits//
volatile bit int2ie@INTCON3.4;
volatile bit int2if@INTCON3.1;
volatile bit intedg2@INTCON2.4;
volatile bit button1@PORTB.1;
volatile bit button0@PORTB.0;

//Button Press variables//
int count;
unsigned short debounce;
int disp;
int chargetime;
int totaltime;
int begin;
//Display variables//
unsigned short dispcurrent1;
unsigned short dispcurrent2;
unsigned char dispucvolts;
int dispspeed;
int disppower;
int dispdecimal;
int displayspeed;

void button_init(void)
{
       trisb.0 = 1;
       trisb.1 = 1;

       count = 1000;
       disp = 0;
       begin = 0;
       chargetime = 0;
       return;
}
void button2_init(void)
{
       trisb.2 = 1;
       int2ie = 1;
       intedg2 = 1;
       int2if = 0;
       return;
}
//What to display on LCD when button is pressed//
void display(void)
{latd=56;
       disp = count % 8;
       switch (disp)
       {
              case 0:
                     LCD_clear();
                     dispucvolts = ultracap_voltage2();
                     LCD_printf(" Ultracap Voltage");
                     LCD_cursor(0x46);
                     LCD_voltage(dispucvolts);
                     LCD_printf(" V");
                     break;
              case 1:
                     LCD_clear();
                     LCD_printf("  Power Consumption");
                     dispucvolts = ultracap_voltage2();
                     dispcurrent2 = current_out();
                     int volts;
                     int current;
                     volts = dispucvolts;
                     dispucvolts = dispucvolts*15400;
                     dispucvolts =dispucvolts/425;
```

```
                current = dispcurrent2;
                int power;
                power = current*volts;
                LCD_current(power);
                break;
        case 2:
                LCD_clear();
                dispcurrent1 = current_in();
                dispcurrent2 = current_out();
                LCD_printf("Current In: ");
                LCD_current(dispcurrent1);
                LCD_printf("000.00 A");
                LCD_cursor(0x14);
                LCD_printf("Current Out: ");
                LCD_current(dispcurrent2);
                LCD_printf("A");
                break;
        case 3:
                LCD_clear();
                LCD_printf("Speed");
                LCD_cursor(0x45);
                LCD_speed();
                break;
        case 4:
                LCD_clear();
                LCD_printLon();
                break;
        case 5:
                LCD_clear();
                LCD_printLat();
                break;
        case 6:
                LCD_clear();
                LCD_printf("   Output Signals");
                LCD_cursor(0x40);
                if (latb.3==1) LCD_printf("     Starter On");
                if (latb.3==0) LCD_printf("    Starter Off");
                LCD_cursor(0x14);
                if (latb.4==1) LCD_printf("   Fuel Valve On");
                if (latb.4==0) LCD_printf("   Fuel Valve Off");
                LCD_cursor(0x54);
                if (late.0==1) LCD_printf("    Glow Plug On");
                if (late.0==0) LCD_printf("   Glow Plug Off");
                break;
        case 7:
                LCD_clear();
                LCD_printf("Charge Time:")
                LCD_cursor(0x40);
                char percentdisp;
                percentdisp = chargetime_disp();
                LCD_dec(percentdisp);
                LCD_printf(" %");
                break;
        default:
                LCD_clear();
```

```
                    LCD_printf("DEEFAWLT");
                    break;
        }
        if (count == 0)
                count = 1000;
        return;
}
void charge_time(int tmr0countval)
{
        chargetime++;
        totaltime = tmr0countval;
        return;
}
char chargetime_disp(void)
{
        chargetime = chargetime*100;
        chargetime = chargetime/totaltime;
        char percent;
        percent = chargetime;
        return percent;
}
int debounce3(void)
{
        while(debounce <1000)
        {
                debounce++;
        }
                debounce = 0;
        begin = 1;
        return begin;
}

void checkbuttons(void)
{
        if (button1 == 1)
        {
                delay_ms(40);
                if (button1==1)
                {
                        count++;
                        display();
                }
        }
        if (button0 == 1)
        {
                delay_ms(40);
                if (button0==1)
                {
                        count--;
                        display();
                }
        }
        return;
}
```

```
#ifndef _BUTTON_LIB_H_
#define _BUTTON_LIB_H_
#include<system.h>


//Button Library Functions//

void button_init(void);
void display(void);
int debounce3(void);
void charge_time(int tmr0countval);
void button2_init(void);
void checkbuttons(void);
char chargetime_disp(void);

#endif //_A2DCONV_H_
```

GPS Library and Header:

```
#include <system.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#include "button_lib.h"
#include "LCD_lib.h"
#include "A2DConv.h"
#include "generator_commands.h"
#include "GPS_parse_lib.h"

//GPS Receive Interrupt Signals//
volatile bit rcif@PIR1.5;
volatile bit rcie@PIE1.5;
volatile bit rcip@IPR1.5;
//GPS Send/Receive Signals
volatile bit txen@TXSTA.5;
volatile bit trmt@TXSTA.1;
volatile bit cren@RCSTA.4;

char gpsout;
char gpsdataR[100];
char gpsdataG[100];
unsigned char num;
char dat;
int tempR;
int tempG;
int getstr;
volatile int RMC;
volatile int GGA;
int commaR;
int commaG;
int commaRpl[12];
int commaGpl[14];
int start;
int stop;
char speed[11];
int speedcount;
int decimal;
char lat[21];
int latcount;
char lon[22];
int loncount;
char time[15];
int timecount;
int endspeedcount;
char speedtopass[4];
int speedknots;
int speedmph;
bool oktoparse;


void init_uart(void)
{
	//Set Baud rate//Initialize PortC USART
	spbrg = 00010000b;
	spbrgh = 00000100b;

	txsta = 00100110b;
	baudcon = 01001000b;  //16-bit generator
	rcsta = 10000000b;
	return;
}
void putc(char value) //Send an ASCII character to Port C
{
	volatile bit txen@TXSTA.5;
	volatile bit trmt@TXSTA.1;
	txen = 1;
	while(true)
	{
		if(trmt)
		{
			txreg = value;
			return;
		}
	}
}
void GPS_init_strings(void)
{
	//Initialize GPS by sending it this string//
	char string1[] = "$PSRF100,1,4800,8,1,0*0E\r\n";
	for (int i=0; i<strlen(string1); i++)
	{
		char x = string1[i];
		putc(x);
	}
	//Turn off unwanted GSV and GSA Messages//
	char string3[] = "$PSRF103,2,0,0,1*26\r\n";
	for (int l=0; l<strlen(string3); l++)
	{
```

```
                char m = string3[l];
                putc(m);
        }
        char string4[] = "$PSRF103,3,0,0,1*27\r\n";
        for (int n=0; n<strlen(string4); n++)
        {
                char o = string4[n];
                putc(o);
        }

}
void GPS_init()
{
        init_uart();
        delay_s(1);
        GPS_init_strings();
        GPS_init_strings();
        GPS_init_strings();
        GPS_init_strings();
        GPS_init_strings();//Send strings multiple times to ensure GPS receives them
        //GPS Variables//
                num = 0;
                rcie = 0;
                rcip = 1;
                getstr = 0;
        //Parsing Variables//
                RMC = 0;
                tempR = 10;
                GGA = 0;
                commaR = 0;
                commaG = 0;
                start= 0;
                stop =0;
        //Get Speed Variables//
                speed[0] = 'S';speed[1] = 'p';speed[2] = 'e';speed[3] = 'e';speed[4] =
'd';
                speed[5] = ' ';speed[6] = '=';
                speedcount = 7;
                decimal = 0;
                speedknots = 0;
        //Get Latitude Variables//
                lat[0]='L';lat[1]='a';lat[2]='t';lat[3]='i';lat[4]='t';
                lat[5]='u';lat[6]='d';lat[7]='e';lat[8]=':';
                latcount = 9;
        //Get Longitude Variables//
                lon[0]='L';lon[1]='o';lon[2]='n';lon[3]='g';lon[4]='i';
                lon[5]='t';lon[6]='u';lon[7]='d';lon[8]='e';lon[9]=':';
                loncount = 10;
        //Get Time Variables//
                time[0]='T';time[1]='i';time[2]='m';time[3]='e';time[4]=' ';
                timecount = 5;
        cren = 0;
        oktoparse = 0;
        return;
}
```

```
//Get time by parsing between 0 and 1st commas//
void gettimeR(void) //store gps output between comma 0-1 in time[] string
{
	start = commaRpl[0]; start++;
	stop = commaRpl[1];
	for (start; start<stop; start++)
	{
		time[timecount] = gpsdataR[start];
		timecount++;
	}
	return;
}
//Get latitude by parsing between 2nd and 3rd (N/S between 3rd and 4th)//
void getLatR(void)
{
	start = commaRpl[2]; start++;
	stop = commaRpl[4];
	for (start; start<stop; start++)
	{
		lat[latcount] = gpsdataR[start];
		latcount++;
	}
	for (latcount; latcount <21; latcount++)
		lat[latcount] = ' ';
	return;
}
//Get longitude by parsing gps between 4th and 5th commas (E/W between 5th and 6th)
void getLonR(void)
{
	start = commaRpl[4]; start++;
	stop = commaRpl[6];
	for (start; start<stop; start++)
	{
		lon[loncount] = gpsdataR[start];
		loncount++;
	}
	for (loncount; loncount <22; loncount++)
		lon[loncount] = ' ';
	return;
}
//Read the speed from the gps by parsing between 6th and 7th commas//
void getspeed(void)
{
	start = commaRpl[6]; start++;
	stop = commaRpl[7];
	int decimalcount = 0;
	int speedpass = 0;
	decimal = decimalcount;
	for(start; start<stop; start++)
	{
		if (gpsdataR[start] == '.')
			decimal = decimalcount;    //to be used to scale speed
		decimalcount++;
		if (gpsdataR[start] != '.')     //store the numbers in the speed value
		{
```

```
                    speed[speedcount] = gpsdataR[start];
                    speedtopass[speedpass] = gpsdataR[start];
                    speedpass++;
                    speedcount++;
             }
      }
      for(speedcount; speedcount<11; speedcount++)
             speed[speedcount] = ' ';
      speedknots = atoi(speedtopass);
      return;
}
int speedtopass(void) //return speed in mph by scaling speed in knots without decimal
{
      speedmph = speedknots*115;
      speedmph = speedmph/100;
      return speedmph;
}
int decimal_to_disp(void)
{
      return decimal;
}
void readGPS(char read)
{
      gpsout = rcreg;
      if (gpsout == '$')
      {
             getstr = 1;
             commaR = 0;
             commaG = 0;
             num = 0;
             while (getstr==1)
             {
                    num++;
                    cren = 1;
                    while(!rcif);
                    gpsout = rcreg;
                    if (num==3)
                    {
                           if (gpsout =='G')
                           {
                                  GGA=0;
                                  return;
                           }
                           if (gpsout =='R')
                                  RMC=1;
                           getstr=0;
                    }
             }
      }
      while(RMC==1)
      {

             if (gpsout == ',')
             {
                    commaRpl[commaR] = num;
```

```
                        commaR++;
                }
                if (num==99 || gpsout == '*')
                {
                        gpsdataR[num] = ' ';
                        RMC = 0;
                        num = 0;
                        getspeed();
                        getLatR();
                        getLonR();
                }
                if (gpsout != '*')
                {
                        gpsdataR[num] = gpsout;
                        num++;
                }
                cren=1;
                while(!rcif);
                gpsout=rcreg;
        }
        return;
}
//Function for printing longitude//
void LCD_printLon(void)
{
        for (int printLon = 0; printLon<21; printLon++)
        {
                if(printLon == 10) LCD_cursor(0x43);
                LCD_putchar(lon[printLon]);
        }
        return;
}
//Function for printing latitude//
void LCD_printLat(void)
{
        for (int printLat = 0; printLat<21; printLat++)
        {
                if(printLat == 9) LCD_cursor(0x43);
                LCD_putchar(lat[printLat]);
        }
        return;
}
#ifndef _GPS_PARSE_LIB_H_
#define _GPS_PARSE_LIB_H_
#include<system.h>
//GPS Functions//
void init_uart(void);
void GPS_init();
void gettimeR(void);
void getspeed(void);
int speedtopass(void);
int decimal_to_disp(void);
void readGPS(char read);
void getLatR(void);
void getLonR(void);
void LCD_printLat(void);
void LCD_printLon(void);
void GPS_init_strings(void);

#endif //_GPS_PARSE_LIB_H_
```