Design of Control for Efficiency of AUV Power Systems

by

Laura M. Ware

Submitted to the
Department of Mechanical Engineering
in Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Mechanical Engineering

at the
Massachusetts Institute of Technology

June 2012

© 2012 Massachusetts Institute of Technology. All rights reserved.

Signature of Author: ____________________________

Department of Mechanical Engineering
May 21, 2012

Certified by: ____________________________

Douglas P. Hart
Professor of Mechanical Engineering
Thesis Supervisor

Accepted by: ____________________________

John H. Lienhard V
Samuel C. Collins Professor of Mechanical Engineering
Undergraduate Officer
Design of Control for Efficiency of AUV Power Systems

by

Laura M. Ware

Submitted to the Department of Mechanical Engineering on May 21, 2012 in Partial Fulfillment of the Requirements for the Degree of Bachelor of Science in Mechanical Engineering

ABSTRACT

The MIT Rapid Development Group designed and built an internal combustion hybrid recharging system for the REMUS 600 Autonomous Underwater Vehicle (AUV) in collaboration with the MIT Lincoln Laboratory. This power system will recharge the lithium ion battery pack of the REMUS 600 and allow the vehicle to travel for 40 consecutive 12-hour missions without returning to recharge.

This study analyzes the optimization of time and fuel efficiency in systems of this type. First, the battery charging scheme for optimal time efficiency was investigated through theoretical simulation of the REMUS battery recharging, based on typical curves for lithium ion battery charging.

Secondly, the optimal control system for optimizing fuel efficiency was found by examining behavior in several different engines and predicting behavior in MIT RDG hybrid system’s engine. A system was developed to control the throttle of the engine while sensing the voltage coming out of a synchronous rectification bridge. This scheme keeps the throttle above 50% unless the power requirement of the charger drops suddenly.

Finally, the control scheme was implemented in software, along with controls for engine starting and shutdown.

Thesis Supervisor: Douglas P. Hart
Title: Professor of Mechanical Engineering
Acknowledgements

The author would like to thank Douglas Hart for his mentorship and leadership of the 2.014 class. Thanks and congratulations to Brianna Woeckener, Yazan Alnahhas, Erich Brandeau, Brian Oldfield, Jessy Mwarage, Julian Merrick, Mariah Murray, and the rest of the 2.014 team for their incredible contributions to the project. Finally, thank you to Gary Ware, Kwami Williams, Kellie Courtney, Ron Perez, Dani Hicks, Liz Jensen, and Abby Van Hook for all their amazing support with writing this document.
# Table of Contents

Abstract

Acknowledgements

Table of Contents

List of Figures

List of Tables

1. Introduction to Project
   1.1 History and Objective of Project
   1.2 Motivation of Project
   1.3 System Architecture

2. Optimizing Conversion Efficiency
   2.1 Time Efficiency
      2.1.1 Charging Lithium Ion Batteries
      2.1.2 Background of Battery Charger
      2.1.3 Methods for Minimizing Charging Time
   2.2 Fuel Efficiency
      2.2.1 Background of Engine
      2.2.2 Engine Efficiency
      2.2.3 Background of Alternator
      2.2.4 Control System Topology
      2.2.5 Engine Dynamics
      2.2.6 Final Design of Throttle Control

3. Implementation
   3.1 Throttle Control Software
   3.2 Engine Starting and Shutdown Logic
   3.3 Stepper Motors and Mount
   3.4 Testing and Future Work

4. Appendix: Full Software

5. References
List of Figures

1.1 Location of the REMUS Power Electronics and Control System 12
1.2 Control System Design from Fall 2011 14
1.3 Engine and Electronics System Architecture 16
1.4 Flow Diagram of the Microcontroller and Control System 17
2.1 REMUS Battery Packs 19
2.2 Lithium Ion Battery Charging Curves 19
2.3 iCharger 3010B Battery Charger Ports 20
2.4 iCharger 3010B Battery Charger Specifications 21
2.5 Control Loop of iCharger 3010B Battery Charger 22
2.6 iCharger 3010B Maximum Output Power vs. Input Voltage 22
2.7 Voltage Curve for our system based on typical lithium ion battery charging curves 23
2.8 Current Curve for our system based on typical lithium ion battery charging curves 24
2.9 Power Curve for our system by multiplying the values from the current and voltage curves. 24
2.10 Total Energy put into the battery, calculated with a trapezoidal integration approximation of the power curve 25
2.11 Charging time after each mission vs. the energy already in the battery when the charging begins 26
2.12 Charging time after every two missions vs. the energy already in the battery when the charging begins 27
2.13 Charge curve and desired operating range for REMUS from Fall 2011 28
2.14 Honda GXH50 Engine 29
2.15 Honda GXH50 Engine Specifications 30
2.16 Wolverine Heavy Fuel Engine 31
2.17 Control Loop Design for Throttle from Fall 2011 35
2.18 System Architecture with Control and Sensing Points 36
2.19 System Architecture with Control Loop between throttle, PWM, and voltage going into the battery charger 38
2.20 Power, bmep, and Specific Fuel Consumption curves vs. Engine speed 40
2.21 BHP and SFC vs. rpm for varying throttle positions 41
2.22 Manipulation of Fig. 2.21 –Specific Fuel Consumption vs. Engine Speed for constant loads at varying throttle settings 43
2.23 Fuel Consumption Map 45
2.24 Honda GXH50 Net torque and Net power vs. Engine speed 45
3.1 ROB-10551 Small Stepper Motor 53
3.2 Pololu A4988 Stepper Motor Driver Carriers 53
3.3 Our pin setup for the Pololu A4988 Stepper Motor Driver Carrier 54
3.4 Connections to PIC24H Microcontroller 55
3.5 3D-printed mount for stepper motors, throttle, and choke 56
List of Tables

Table 2.1 Fuel consumption of Honda GXH50 under full and no load conditions 30
Table 2.2 Control Points and Sense Points in the System 37
1 Introduction to Project

1.1 History and Objective of UUV Power System Project

In collaboration with the MIT Lincoln Laboratory and Woods Hole Oceanographic Institute, the MIT Rapid Development Group completed a design effort to develop a new hybrid power system that will allow a REMUS 600 Autonomous Underwater Vehicle (AUV) to travel for 40 consecutive 12-hour missions without returning. During Fall 2011, the MIT team focused on exploring significant design decisions. In determining the optimal system to power the REMUS batteries, considerations included safety, size constraints of the REMUS hull, and feasibility with available time and funding. The team selected an internal combustion hybrid recharging system to pursue as a design. This hybrid power system can be organized into four different subsystems: the engine system, the fuel system, the snorkel system, and the control system.

*Engine*

The engine selected for the power system was the Honda GXH50 engine, taken from the EU1000i generator, which produces 1000 W. The engine is specified to be 16% efficient, and although it is designed to be manually started and adjusted, along with air cooled, we have configured it to operate automatically with a water cooling system.

*Fuel*

The fuel system comprises a sealed tank, in which fuel is stored in bladders. As the engine uses fuel, the vehicle is able to take on water and air to replace the lost mass, allowing the vehicle to have zero moment and neutral buoyancy.

*Snorkel System*

The snorkel system is in use only when the REMUS has surfaced to recharge its batteries. The snorkel brings air inside to create the desired buoyancy, and it lets in water to cool the engine. In addition, it has a variable ballast air reservoir that empties at the surface to allow the REMUS to float higher on the waves.
Control System

The control system manages power generation and controls components in the power system to allow the battery to be recharged and ensure system safety during recharging. The power regulation board filters the three-phase AC power from the generator, allowing it to be usable for battery charging. The control board interfaces with the REMUS computer. Figure 1.1 shows the location of the control system within the REMUS power system.

![Location of the REMUS Power Electronics and Control System](image)

Figure 1.1: Location of the REMUS Power Electronics and Control System

In a standard cycle, the REMUS surfaces upon receiving a signal from the computer that it needs to recharge. The system first empties the water from the snorkel plenum by opening the high-pressure and low-pressure valves and activating the water pumps. Water conductivity sensors are used to detect when the water level has emptied sufficiently. At this time, the valves will open to allow air into the engine. After a series of steps to control the throttle, choke, and ignition kill switch, the engine is started by back-driving the generator. Once the engine is running stably and generating power, the synchronous machine is turned on and the battery begins to charge. In a series of cycles, valves are activated to allow fuel and air to be fed in during charging to offset their loss and keep the mass of the vehicle constant. During these cycles a series of safety checks are conducted as well. The power is filtered and used to charge the batteries. When the
control system detects a sufficiently charged battery, the engine turns off. The REMUS then dives from the surface and returns to its mission.

Figure 1.2 portrays the above steps in a control diagram. The top section of the diagram shows preparation for charging and starting the engine. The middle section represents the main process of charging, in which the fuel and air tank valves are activated, and safety interrupts are conducted. Finally, the bottom section shows the engine shutdown process.
Figure 1.2: Control System Design from Fall 2011
1.2 Motivation of Project

The following work focuses on the control of the engine and optimization of efficiency. Both size constraints and charging time were considerations in design decisions for the power system, such that optimizing efficiency is critical to the feasibility of the system. In this particular underwater vehicle, conserving space and weight is especially vital. As a result, the volume of fuel needs to be kept at a minimum, and the system needs to use fuel as efficiently as possible. The purpose of this work is to explore the rationale behind creating efficiency in power systems of this kind, and to explain the methods chosen and implemented for the REMUS power system.
1.3 System Architecture

The heart of the hybrid internal combustion recharging system is shown in Figure 1.3. In order to charge the lithium ion batteries of the REMUS, the internal combustion engine takes in fuel and air to convert chemical energy to mechanical energy. The throttle controls the flow of air to the engine, which combines the air with fuel from the fuel tank and spins an alternator. The alternator produces three-phase AC current using DC field excitation. A synchronous rectifier takes the output from the generator and rectifies it, such that the input to the battery charger is DC voltage. The battery charger then uses a buck-boost converter to convert voltage and current into the requirements of the charging process. The system controller controls both the engine and the rectifier. The engine control sets the throttle and choke positions, along with controlling the behavior of the ignition. The alternator can also be set to back-drive the engine in order to start it.

![Figure 1.3: Engine and Electronics System Architecture](image)

Figure 1.3 shows a schematic of the interaction of the PIC24H microcontroller with the system. During this project, we have not had access to the REMUS computer,
but our control system is designed such that the REMUS will communicate information including when the battery needs to recharge, and the microcontroller will send data back to the REMUS computer. The microcontroller will take in data from each of the sensors and in turn control valves, electronics, and actuators appropriately.

Figure 1.4: Flow Diagram of the Microcontroller and Control System [1]
2. Optimizing Conversion Efficiency

The main goal of my work with the MIT RDG during Spring 2012 was to analyze and optimize overall efficiency of the power system. Efficiency for this type of system can be defined as the extent to which time and fuel are well used to charge the batteries. Time needs to be conserved because time spent recharging at the surface of the water is time taken away from the vehicle’s mission. Time on the surface is also more time that the REMUS is vulnerable to being detected. Fuel needs to be conserved mainly because of the size and weight constraints of the REMUS hull. I will examine each of these types of efficiencies, but as size and weight constraints are more critical to the feasibility of this project, they will be the main focus of this work. I seek to answer the following question: given the limited volume available for fuel in the REMUS hull and fuel bladders, how can we optimize system efficiency to produce sufficient energy during each charging cycles and make this project feasible?

2.1 Time Efficiency

2.1.1 Charging Lithium Ion Batteries

The REMUS is able to hold up to three battery packs as shown in Figure 2.1. Each of these packs contains ten modules of seven 4.2-V lithium-ion cells each. Lithium batteries are ideal for this system for many reasons. They are rechargeable batteries designed for high discharge rates and frequent use, often used in high-performance systems because of their high specific energy, which is typically close to 0.15kWh/kg. Lithium ion batteries also have a cycle life of up to 1000 discharge/charge cycles, making them a good choice for highly efficient systems that will be frequently recharged, such as the REMUS power system. Finally, the charging energy efficiency is 95%, significantly higher than lead-acid, Ni-Cd, or Ni-mH batteries. [2]
Figure 2.1: REMUS Battery Packs [1]

Figure 2.2 shows the current and voltage over time during the charging process for a single cell. In order to achieve proper charging, the current should be held constant in stage one, and the voltage should be held constant in stage two. As either voltage or current are always changing, the recommended power level is changing continually during charging. If the power input to the battery is less than this recommended level, charging will require more time. If the power input is more, the batteries could potentially be damaged. In particular, the lifetime of the batteries may decrease as a result of excessive heat dissipation. The maximum power requirement during charging occurs at the end of Stage 1, when power is nearly 1000 W.

Figure 2.2: Lithium Ion Battery Charging Curves [3]
2.1.2 Background of Battery Charger

In order to simplify the battery charging process and control the battery charging algorithm, we decided to use a battery charger off the shelf. The iCharger 3010B (pictured in Figure 2.3) takes the DC voltage coming out of the synchronous rectifier and inputs it into the battery.

![iCharger 3010B Battery Charger Ports](image)

Figure 2.3: iCharger 3010B Battery Charger Ports [4]

The charger has a number of features beneficial for the efficiency and safety of the system. A built-in fan monitors against overheating. If the temperature rises above 55°C, the power output will be reduced by 25%. If the temperature rises above 60°C, charging will be terminated. For lithium batteries, the battery charger also has a number of settings for various needs. For example, it can be set to balance charging, normal charging, fast charging, storage, discharging, ext-discharging, charge/discharge cycling, and battery monitoring.

Both the normal charge and fast charge mode use the two phases of charging explained above: a constant current phase and then a constant voltage phase after the voltage reaches its peak. In the normal charging mode, the charging will be terminated when current is 1/10 of the charge current (set by the user). In the fast charge mode, the
charging will be terminated when the current is 1/5 of the configured value. The battery will be less than 100% of the charge, but the charging will be done faster. In our design, the batteries should not reach a current this low, so it does not matter which of these modes is chosen.

The discharge capability is not needed in this application. As the mode setting is manual, it must be preset. Our system controller needs to track the charging progress and stage. The battery charger, however, does not communicate with our system controller, such that we must create an indirect way of tracking the progress. By monitoring the current going into the battery, we can track the constant current stage (stage 1), then detect the decreasing current (stage 2).

As shown in the specifications (Figure 2.4), the iCharger 3010B takes input voltages between 4.5V and 38.0V, and it requires a charge/discharge current rate of 0.05A to 30.0A. As long as the current and voltage are within these specifications, the battery should be charging safely.

<table>
<thead>
<tr>
<th>Specifications</th>
<th>3010B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input voltage range</td>
<td>4.50 – 38.0VDC</td>
</tr>
<tr>
<td>Charge current range</td>
<td>0.05 – 30.0A</td>
</tr>
<tr>
<td>Discharge current range</td>
<td>0.06 – 30.0A</td>
</tr>
<tr>
<td>Maximum charge power capacity</td>
<td>1000W @ input voltage 23V (500W @ input voltage 12V)</td>
</tr>
<tr>
<td>Maximum discharge power capacity</td>
<td>80W</td>
</tr>
<tr>
<td>Maximum regenerative discharge power capacity</td>
<td>1000W</td>
</tr>
<tr>
<td>Maximum external discharge power capacity</td>
<td>1200W @ 40V/33A</td>
</tr>
<tr>
<td>Current drain for balancing</td>
<td>&lt;500mA</td>
</tr>
<tr>
<td>Balance accuracy</td>
<td>&lt;10mV</td>
</tr>
<tr>
<td>Lithium (LiPo/Li/Fe) battery cell count:</td>
<td>1 – 10 series (in non-balance mode, expand LiFe to 12s)</td>
</tr>
<tr>
<td>NiCd/NiMH battery cell count:</td>
<td>1 – 25 series</td>
</tr>
<tr>
<td>Pb battery cell count:</td>
<td>1 – 18 series (2 – 36V)</td>
</tr>
<tr>
<td>Log File storage:</td>
<td>16Mbit (31 hours)</td>
</tr>
<tr>
<td>Battery setup memories:</td>
<td>Yes</td>
</tr>
<tr>
<td>Intelligent temperature control:</td>
<td>USB port</td>
</tr>
<tr>
<td>PC Connect:</td>
<td></td>
</tr>
<tr>
<td>Weight:</td>
<td>750g</td>
</tr>
<tr>
<td>Dimensions (L X W X D):</td>
<td>143X123X46mm</td>
</tr>
<tr>
<td></td>
<td>5.63&quot;X4.84&quot;X1.81&quot;</td>
</tr>
</tbody>
</table>

Figure 2.4: iCharger 3010B Battery Charger Specifications [4]

Similar to the buck converter we were originally planning on using, the charger uses a Synchronous Buck-Boost DC/DC converter with an output conversion efficiency of over 90%. As shown in Figure 2.5, the charger has its own controls that sense the voltage and current going into the battery. The converter steps up or steps down the
voltage depending on the stage and progress of charging. Therefore, the charger removes
the necessity for us to set a constant voltage or current with our own buck converter and
complex electrical circuitry. We must only monitor the voltage and current going into
the battery charger to ensure it is within the charger’s specifications.

Finally, Figure 2.6 shows the maximum output power of the battery charger vs.
input voltage. This information is important to our charging strategy because the power
will remain constant at 1000W once the voltage rises above 23 V. This does not imply
that the power at 23V will always be 1000W, but an input voltage of 23V will ensure the
battery charger is capable of maximum power and thus, that the charging is progressing
at the maximum rate that is safe and possible.

Figure 2.5: Control Loop of iCharger 3010B Battery Charger

Figure 2.6: iCharger 3010B Maximum Output Power vs. Input Voltage [4]
2.1.3 Methods to Minimize Charging Time

As previously noted, one REMUS battery tray has 10 modules of 7 cells each. Each cell is 4.2 V, thus indicating a voltage of 29.4V per module with a peak rating of 3.3A per module. The cells in each module are in series, and the modules in each tray are in parallel, such that one tray has a peak current of 33A. The maximum charge rate should be 32V at 33A.

If we assume no losses, a current of 33A, and a voltage of 29.4V, we can estimate a power requirement of 1056 W to charge one battery pack at peak consumption. One battery tray stores 5.2kW-hr of energy, such that it requires more than 5.2kW-hr of energy to charge the battery. As a result, if it were using maximum power to charge the battery pack, it would take an estimated 6 hours to charge one tray or 18 hours to charge three trays sequentially.

In our power system design, however, the REMUS only needs to use one battery tray, and the power does not remain at 1 kW. In order to calculate the total time that would be necessary if we charged the battery tray from depletion to full charge, it is possible to duplicate the Lithium ion charging curve from Figure 2.2 and modify it using our values to produce the curves shown in Figures 2.7 through 2.10.

![Voltage Curve](image)

Figure 2.7 Voltage Curve for our system based on typical lithium ion battery charging curves
Figure 2.8: Current Curve for our system based on typical lithium ion battery charging curves.

Figure 2.9: Power Curve for our system by multiplying the values from the current and voltage curves.
Figure 2.10: Total Energy put into the battery, calculated with a trapezoidal integration approximation of the power curve

The power curve was created from the product of the voltage and current curve, and the total energy into the battery was found from a trapezoidal integral approximation. According to this approximation, in order to achieve a total energy input of 5.2 kWh, each battery tray needs to be charged for 9.13 hours. Charging all three battery trays completely, therefore, would require 27.4 hours. These values assume no losses and should be considered low.

The REMUS has two different modes: high speed (mission mode) and low speed. The high-speed mode is used only 5% of the time. For one mission, the REMUS needs 0.24 kW-hr for mission mode, 1.38 kW-hr for low speed mode, and 0.144 kW-hr for idle processes. In total, it is estimated that the REMUS requires 1.77 kW-hr of energy for one 12-hour mission. Since one full battery charge provides 5.2 kW-hr of energy, this would theoretically allow 2.9 missions per each full charge.

Given this analysis, we can conduct charging in two different ways. We can charge the battery between each 12-hour mission after approximately 1.77 kW-hr of energy use, or we can charge after every 2 missions and approximately 3.54 kW-hr of energy use. That is, we can charge 34% or 68% of the battery during each cycle.
First assuming that we charge after every mission, we also have a choice about the energy state at which we recharge the battery. One option is to charge the battery fully to 5.2 kW-hr at first and then recharge the battery when the capacity reached 3.42 kW-hr (66%). Another option is to oscillate between a two partially charged states, such as 3.54 kW-hr (68%) state and a 1.77 kW-hr state (34%). Figure 2.11 allows us to analyze these different situations by calculating the charging time to charge 34% of the battery, given the starting state of the battery. The first option requires a charging time of approximately 4 hours and the second option requires a charging time of approximately 2 hours. The charging time will be least when the average power during the charging window is highest. This analysis shows that the minimum charging time is 1.8 hours and occurs when the battery has a capacity of 1.16 kWh. This implies that the most time efficient charge occurs between a 22% state and a 56% state.

![Charging Time (Every Mission)](image)

Figure 2.11: Charging time after each mission vs. the energy already in the battery when the charging begins

The second scenario involves charging after every 2 missions. Again we can consider different charging windows in which to operate. Figure 2.12 shows that the minimum charging time would be 4.33 hours, or 2.16 hours per mission. Therefore, this
charging scheme would theoretically be less time efficient. This makes sense because charging a smaller amount every time allows the charging window to be closer to the peak power and have a higher average power input. Charging every mission, however, requires the time and energy to empty the plenum, start the engine, and more. The time for these activities was not taken into account for this approximation.

![Charging Time (Every 2 Missions)](image)

Figure 2.12: Charging time after every two missions vs. the energy already in the battery when the charging begins

The following figure shows the charge curve at 0.36A. The original design specifications for the power system during Fall 2011 showed that the desired operating range for the REMUS is between 48% charge and 83% charge (Fig. 2.13). This analysis shows, however, that the charging time for this scheme would be approximately 2.47 hours, or 37% longer than the minimum charge time.
Because there is no way for the system controller to know the state of charge of the battery, the difficult part is actually ensuring that the charging occurs between the two suggested states. One way we can do this indirectly is by stopping charging once the current in stage 2 drops below a fixed value. At a 56% charge stage, the current can be estimated to be 27.7A.

2.2 Fuel Efficiency

2.2.1 Background of Engine

As mentioned, the MIT RDG chose an internal combustion hybrid system to conduct this recharging. In particular, the heart of this system is the Honda GXH50 engine (Fig. 2.14), which was taken from the Honda EU 1000i generator. Using gasoline...
stored in the fuel tank, this engine will power a custom AC generator to produce electricity filtered by a synchronous machine. Figure 2.15 shows the Honda GXH50 specifications, which are especially important for discussion of efficiency.

Figure 2.14 Honda GXH50 Engine [5]
Figure 2.15 Honda GXH50 Engine Specifications [5]

Through tests conducted in Fall 2011 (Table 1.1), the fuel consumption of our engine was discovered to be 0.6 gallons in 3.8 hours at 100% load, or 0.12 g/s at 100% output. The air consumption is 1.8 g/s.

Table 2.1: Fuel consumption of Honda GXH50 under full and no load conditions [1]

<table>
<thead>
<tr>
<th>Load Condition</th>
<th>Mass Burned</th>
<th>Time to Burn Fuel</th>
<th>Mass Consumption</th>
<th>Fuel Density</th>
<th>Volume Consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Load</td>
<td>1.0 g</td>
<td>8.2 s</td>
<td>0.12 g/s</td>
<td>0.651 g/mL</td>
<td>0.18 mL/s</td>
</tr>
<tr>
<td>No Load</td>
<td>1.0 g</td>
<td>10.5 s</td>
<td>0.095 g/s</td>
<td>0.651 g/mL</td>
<td>0.15 mL/s</td>
</tr>
</tbody>
</table>

The cited efficiency of the engine according to the specifications is 16%. According to fuel tests conducted in Fall 2011, the efficiency at full throttle was 17%,
calculated using a volumetric consumption rate of 0.18mL/s at full load (960W) and a rate of 0.15mL/s with no load.

The operating point refers to point along the torque-speed curve at which the engine runs. The operating point can be adjusted with the gear train of the engine, the field current, and the load applied to the engine. This will determine speed and torque of the engine.

2.2.2 Engine Efficiency

There are several ways of increasing the efficiency of the engine specifically. First, one can optimize fuel type. Mixed fuels such as including methane mixes and high-octane gasoline mixes have high power densities. The safety risks with these fuels, however, are increased because the risk of explosion is higher.

Secondly, one can optimize engine type. The alternative engine choice was the Wolverine Heavy Fuel Engine (Figure 2.16). This two-stroke engine generates more power, with a higher efficiency of 20%. It can use any type of fuel and be set to a low throttle position to decrease fuel consumption. The prohibitive cost and lead time for custom development, however, made this choice impossible for the project this year.

![Figure 2.16 Wolverine Heavy Fuel Engine](image)

Figure 2.16 Wolverine Heavy Fuel Engine [1]

At the start of Spring 2012, the type of engine and fuel had been decided. With the capabilities of these components known, we could then determine the fuel efficiency necessary to make the project feasible.
The minimum conversion efficiency of the system was calculated during the design process in Fall 2011. The Honda GXH50 engine has a fuel requirement of unleaded 86 octane or higher. We can assume an energy density of 34 MJ/L for this type of gasoline [6]; we can also estimate an availability of 45 L of fuel. In our energy calculations, we determined that the total energy necessary for forty missions is 70kWh, or 252MJ. This takes into account the hotel load, sensor load, and propulsive power according to the specified mission profile. With 45L, we estimate the availability of 1530MJ of chemical energy. This implies a total conversion efficiency of 16.5% efficiency.

During Spring 2012, the available volume for fuel decreased as a result of the limitations of the fuel bladders chosen for the system. The available volume is now estimated to be 33L. This may present a serious threat to the feasibility of the system. With 33L, we can estimate an availability of 1122MJ of chemical energy, suggesting a necessary conversion efficiency of 22.5%. The cited efficiency of the Honda engine is 16%, meaning that 16% of the chemical energy from fuel is converted to mechanical energy. This does not take into account the conversion efficiency of mechanical energy from the engine to electrical energy going into the battery, which includes the efficiency of the synchronous machine and battery charger. In addition, it does not account for the waste of fuel that occurs by providing a higher power input than necessary to the battery. The insufficient estimated efficiency for our original specifications, therefore, makes this work even more important.

We can also estimate the required efficiency using our previous calculations for the charging time. If fuel consumption is close to 0.18 mL/s at full load, and we need to charge the battery for 1.8 hours, each mission will require an estimated 1.17L of fuel. This implies that a capacity of 33L will provide enough fuel for 28 missions rather than the original goal of 40 missions. If we were to charge every two missions and take 4.33 hours to charge every cycle, we would need 2.8L of fuel each cycle and be able to have fuel for 22 missions. Conserving fuel and determining a scheme to run the engine at parameters of highest efficiency could allow the REMUS to run for 40 missions as planned.
2.2.3 Background of Alternator

A generator uses an operating principle that is the reverse of a motor. According to Faraday's law of magnetic induction, a changing magnetic field through a loop of wire will induce a current in that wire. The alternator that was designed and fabricated for the REMUS power system uses a brushless design, such that the rotor (rotating part) is a field winding rather than a permanent magnet. This rotor is turned by the Honda engine, and the current supplied to the rotor windings comes from a DC source. The brushless nature of this solution minimizes losses. The current is known as the field current. The stator (stationary part) consists of three coils of wire offset by 120 degrees. Thus, the rotating magnetic field induces AC current in these coils, producing three sine waves offset by 1/3 phase each. The synchronous rectifier then takes this current and converts it back into a DC energy source to charge the battery. Considering the complexity introduced by power regulation, one might wonder why a DC generator is not used instead of a three-phase AC generator. AC generators are more common for high current applications, such as this one. [7]

A voltage regulator can ensure that the voltage output of the generator is constant. When the load increases, the output voltage drops, such that more field current is supplied and the voltage is restored. The field current is typically much less than the output current of the generator.

The voltage output by the generator depends on the speed, the load from charging, and the DC field excitation. If the speed of the generator is held constant, and the DC field excitation increases, the output voltage will increase proportionally. After some value, the flux will be saturated and the increase in voltage will no longer be proportional.

The output voltage depends on the magnetic flux in the air-gap, which is only determined by field current if there is no load on the generator. When the battery is charging however, the current draw creates an opposing force in this air gap as a result of the back EMF. When a load is applied, the current in the stator windings increases, and a magnetic field surrounding the coil of wire produces a force opposite in direction as that of the rotor windings. As a result, it increases the torque on the engine and the speed decreases. [8]
When the load used by charging the battery is more than the power output of the generator, the speed of the engine will decrease. This is because the magnetic field of the armature increases. [9]

We can change the load point by increasing or decreasing the field current. Increasing the DC field excitation will cause the magnetic field of the rotor windings to be stronger, thus increasing the rate of change of magnetic flux in the stator windings. With no load (and no back EMF), this will not change the speed but will change the output voltage. With a load, the effect of the back EMF will be stronger, and increasing the field excitation will decrease the speed and increase the torque.

The induced voltage (EMF) is a function of speed (rpm) and field excitation according to Faraday’s law of induction. Increasing the speed decreases the time over which the magnetic flux is changing, and increasing the field current increases the change in magnetic flux over a given time interval.

**2.2.4 Control Loop Topology**

The main effort of this work was analysis of the design of appropriate control loops to optimize efficiency. Before reaching complete understanding of the system, our team explored many different design options to determine what parameters to sense and what parameters to control. Overall, the purpose of the control is to ensure both safe and efficient charging procedures.

When I began working with the MIT RDG in February 2012, the aim of this project was to control the battery charging current with the throttle. The controls team was investigating using a Proportional Integral (PI) Controller in firmware to set the throttle to an appropriate setting for the range of currents we desired. Fig. 2.17 shows the control loop that outlined the design planned for the software. The reference or desired current is compared with the current coming out of the generator, as detected by the microcontroller. The error is then put into a PI controller to find the setting for the stepper driver. The number of pulses from the stepper driver determines the angular position of the throttle. This angular position has an effect on current by changing the speed of the engine. This new current is put back into the control loop and the cycle repeats.
This projected plan, however, was changed significantly over the course of the term, as a result of increased information and understanding of efficiency and understanding of the desired behavior of the system. In particular, a PI controller for the throttle was more complex than necessary. Because the throttle is a mechanical component controlling the engine, it takes significantly longer to alter than an electrical component, and its feedback loop is relatively slow. The desired behavior for the throttle, therefore, is minimal oscillation. That is, we wanted the throttle to open or close gradually rather than switching directions many times, especially because there is some error associated with each pulse sent to the stepper motors.

Instead of assuming this control system design was the best path to optimize efficiency, our team examined a number of design decisions to fully understand how they would affect the performance of our system.

**Decision One: Buck-Boost Converter or Battery Charger**

The first decision we explored was whether to have a buck-boost converter as the control element, an off-the-shelf battery charger, or both. Having both elements was ruled out because the battery charger itself is essentially a buck-boost converter, such that having both is unnecessary and adds inefficiency. It is possible to use only a buck-boost converter rather than a battery charger, but the design work associated with this solution would not be a wise use of time. A buck-boost converter would convert the
voltage output from the synchronous rectifier into the voltage necessary for the battery charger. In order to do this, we would need to design the buck-boost converter to have two stages: a constant current phase and a constant voltage phase, according to the battery charging process explained previously. Rather than complicating the project in this way, we decided to use an off-the-shelf battery charger that takes care of the phases associated with charging lithium ion batteries.

**Decision Two: Points of Control and Sensing**

The second decision that had to be made regarding design of the synchronous machine and control was what would be controlled and what would be sensed. The first step was to look at the system architecture (Fig. 2.18) and outline all of the possible points of control and sensing (Table 2.2.). The stepper driver is the first point of control. Giving the stepper driver a certain number of pulses determines the angle of the stepper motor opening and closing the throttle and choke. The choke is only needed while starting the engine, so it was ruled out as a control point while the engine was running. The mode of the rectifier is also only important in engine starting, so the three controls able to control charging are only the throttle, the field current going into the generator, and the PWM of the synchronous rectifier.

![Figure 2.18 System Architecture with Control and Sensing Points](image-url)
Table 2.2 Control Points and Sense Points in the System

<table>
<thead>
<tr>
<th>Control Points</th>
<th>Sense Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throttle position</td>
<td>Engine RPM</td>
</tr>
<tr>
<td>Choke position</td>
<td>Engine Torque</td>
</tr>
<tr>
<td>Field Current</td>
<td>Voltage/Current out of generator</td>
</tr>
<tr>
<td>PWM of Synchronous Rectifier</td>
<td>Voltage/Current into Battery</td>
</tr>
<tr>
<td>Mode: Rectifier/Starter</td>
<td>Charger</td>
</tr>
<tr>
<td>Voltage/Current out of Battery Charger (into Battery)</td>
<td></td>
</tr>
</tbody>
</table>

Control Point:

Out of the three possible points of control, the throttle was chosen. A major advantage to controlling the throttle is that it already needs to be controlled during the starting and shutdown of the engine, such that it is much simpler to add control to the throttle. In contrast, the field current is set to a constant and does not otherwise need to be controlled. Using the field current or PWM to control the output of the synchronous bridge, however, could be explored in future work.

The field current should be chosen such that the appropriate dynamic range of power is produced. From Figure 2.9 it was determined that we need approximately 800W – 1kW. The field current must be adequate to achieve 1kW at full throttle with some design margin. Figure 2.21 will show that fuel efficiency is poor at low throttle settings, therefore we do not want to set the field current higher than necessary, which would require a low throttle setting to achieve the 800W.

Sense Point:

A number of parameters could be used to control the throttle. We could have a control loop, for example that sensed the speed of the engine and increased or decreased the throttle position to achieve the appropriate rpm. The engine speed is directly related to the voltage output of the generator, such that this is similar to using the throttle to
control the voltage. We could find the engine speeds that correspond with the voltages coming out of the generator.

Ultimately, once we decided to use a battery charger, it was clear that the simplest and most practical sense point was the voltage going into the charger, as it needed to be within a certain specification range. Controlling the throttle to change the rpm of the engine would only be indirectly ensuring the safety of the flow of electricity into the charger.

The diagram in Fig. 2.19 shows the points of control and sensing in the system architecture, along with the control loop that was eventually selected. Voltage #2, the voltage coming out of the synchronous rectifier and into the battery charger, is sensed and fed back to the controller, which gives the stepper driver pulses to change the position of the throttle. In return, the angle of the throttle affects the speed, RPM, torque, voltages, and currents throughout the system. Because of the complex relationships between these parameters, we still needed to investigate how the voltage readings should be used to control the throttle.

Figure 2.19: System Architecture with Control Loop between throttle and voltage going into the battery charger
2.2.5 Engine Dynamics

There are a number of parameters related to engine dynamics that can be analyzed in order to determine appropriate methods of achieving the highest system efficiency possible. The main two controllable factors are throttle setting and field current. In turn, these affect engine speed, torque, and power output of the engine. The load on the engine affects these parameters as well.

Our main goal in controlling engine dynamics is to optimize fuel efficiency while staying within the specifications of the battery charger and, thus, not damaging the battery or charger. Optimizing fuel efficiency can be thought of as optimizing the total mass of fuel used for the total energy required to charge the battery. A relevant metric is Brake Specific Fuel Consumption (BSFC)—or simply Specific Fuel Consumption (SFC), the rate of fuel consumption divided by the power produced. This metric is especially helpful in comparing the fuel efficiencies of different engines and at different power outputs. Since the power requirements change throughout the battery charging process in our system, we wish to achieve a given power input with the lowest fuel consumption possible. Our highest priority is minimizing the waste of fuel.

The purple curve in Figure 2.20 below shows specific fuel consumption (SFC) in g/kWh for full throttle and varying speed. It is clear that, with this constant throttle position, SFC is lowest at low to medium speeds for this particular engine model (different from the one in our system). In addition, maximum torque is produced at medium speeds, but it is a myth that SFC is necessarily highest at maximum torque, or at the maximum power point. This data suggests that if we were using this engine model and we kept the throttle at 100%, we would run the engine between 2000rpm and 3000rpm for optimal efficiency. This assumes that the power output corresponding to that engine speed is appropriate. For this engine, the point at which SFC is the lowest corresponds with approximately 100 kW, much higher than the range of our battery charger.
Figure 2.21 shows two different relationships between engine parameters. The top graph in Figure 2.21 shows brake horsepower, B.H.P., vs. engine speed for various throttle settings. The bottom graph shows SFC for different throttle settings. This bottom graph offers important information about fuel efficiency. Firstly, it shows us that for this particular engine model, the differences in SFC between half throttle and full throttle are not very significant for most speeds. All values of SFC are between 0.45 lb/B.H.P./hr and 0.55 lb/B.H.P./hr. The SFC values for quarter throttle, however, increase drastically above 2000rpm.

This means that if our engine is similar, and if it is possible given our power requirements, we should spend most effort in disallowing the throttle from going below 50%. Estimations can help determine whether quarter throttle would ever need to be used. Our engine has a maximum power output of 1.6kW. Assuming an efficiency of 85% for the generator and synchronous rectifier, a maximum load would be a power input of 1.36kW into the battery charger. For the range of charging suggested by the discussion above, the initial energy stored in the battery would be 1.16kW-hr, or 22% capacity. According to Figure 2.21, the power output at this charge is 862 W, or 63% load. The engine from Figure 2.21 is about a 62 BHP engine, such that 63% load would
occur at 39 BHP. According to the figure, both full throttle and half throttle should provide enough power for this load, such that the engine never has to use a quarter throttle.

Following the original recommendation for the REMUS battery charging, we would begin charging at 48% capacity, at which power input to the battery would be near 1000 kW, or 74% load. This would be the equivalent of 46 BHP in Figure 2.21. Half throttle provides potentially enough power, but its peak is very close to 46 BHP. In this case, it would be necessary to keep the throttle close to 100%.

![Figure 2.21 BHP and SFC vs. rpm for varying throttle positions][1]

Figure 2.21 also shows important relationships between the throttle setting and power. Firstly, it shows that for a given engine speed, a higher throttle setting will lead to more power. This is because more air is allowed into the engine, and combustion takes place at a higher rate. Secondly, the figure shows that for a given power output, a lower throttle setting will generally lead to a lower engine speed.

Practically, this has important implications. At any given point, the battery charger has a certain power requirement that it will supply to the battery, such that
changing the throttle setting will not change the load on the engine. The battery charger is essentially constant power in short range, and variable power in long range. Therefore, changing the throttle setting will cause the engine parameters to shift to the point of corresponding power output on the curve of the new throttle setting. In most cases of medium to low engine speeds, decreasing the throttle setting will actually cause the engine to spin faster in order to produce the same output. A faster speed on the generator leads to a higher voltage because the magnetic flux is changing more rapidly. The voltage will increase and the current will decrease.

The opposite affect may occur in some cases, however. For example, if the throttle is set to 25%, and the power requirement of the battery charger is in the range of the quarter throttle, switching to half throttle may cause the engine speed to jump to an incredibly high rpm value, where the power is much lower. Therefore, setting the throttle too high could cause the voltage to rise higher than the battery charger can handle.

Because the battery charger has an upper limit of 38V, and we wish to provide a small buffer, we should not allow a voltage into the battery charger of more than 34V. If the voltage reaches this value, the throttle is most likely set too high, and we should therefore decrease the throttle setting in order to lower the voltage.

We have discussed the relationship between throttle setting and engine speed, but what is the relationship between throttle setting and specific fuel consumption, and how does this affect our methods of optimizing efficiency? Using both the top and bottom graphs, we can manipulate the data to determine the effect SFC when power remains constant and the throttle is opened or closed.

Figure 2.22 shows the manipulation of these graphs—for constant power lines, it plots specific fuel consumption vs. engine speed. Each point is also labeled with the throttle position. For two cases, SFC decreases (gets better) for higher throttle settings
and lower rpms. In one case, it stays the same, and in one case, it gets worse for a higher throttle setting and lower speed. There is not a clear relationship between this behavior and the percent load. Furthermore, assuming that we are only changing between half throttle and full throttle, the difference is small; increasing the throttle from half position to full position generally only changes the specific fuel consumption by 10% or less.

Our main conclusion from these graphs, then, is that if it is not imperative for the throttle to be less than 50%, it should not be, as the specific fuel consumption is generally significantly higher than higher throttle positions, especially at higher engine speeds. As is clear in the bottom graph of Figure 2.21, there is only a small overlap in the power ranges of quarter throttle and half throttle. If the load on the engine was in the range of quarter throttle, we would need to start the throttle on 25% and increase to half throttle only when the power requirement of the battery charger increased. As previously mentioned, if the power requirement is in the range of quarter throttle and we kept the throttle at 50%, the engine will jump to the point on the half throttle curve of lowest power, farther right than is shown on the graph. As a result, if we leave the throttle
setting at half or above, we risk the voltage increasing too high if the power requirement is momentarily low. This is reason to add control such that the throttle will decrease if the voltage goes above the range of the battery charger.

The above discussion of specific fuel consumption is helpful in determining a theoretical approach to optimizing conversion efficiency, but the data differs significantly depending on the engine model. As a result, empirical data is necessary to determine the power outputs at various throttle settings and loads.

In an ideal world, if we had not yet decided on an engine model, we would consider specific fuel consumption before purchasing an engine. For the engine from Figure 2.21, for example, it is clear that the best specific fuel consumption occurs at half throttle when the speed is between 2500 rpm and 3000 rpm. This corresponds to 35-40 BHP, which is equivalent to a range of 58% load to 67% load. To maximize conversion efficiency, it is best to match this horsepower to a value of power that is in the range of the power the battery charger will be outputting to the battery.

In our case, with the range suggested previously, the load during charging should go between 63% load and 74% load. This suggests that the fuel efficiency should be close to optimum values during charging.

Finally, we examine a map of specific fuel consumption that combines measurements of torque (measured in bmep), engine speed, power output, and SFC.
The data in Figure 2.23 was taken from a 100 HP engine. It is interesting to note that the lowest fuel consumption occurs close to 50% load and a low engine speed. This is similar to what we found in Figure 2.21, where the lowest specific fuel consumption occurred at half throttle, a low to medium engine speed (approximately 2750 rpm) and close to 60% load.

Figure 2.24 shows the only data available to us for the Honda GXH50 engine we are using. It shows power and torque as functions of speed. It is interesting to note that the recommended operating speed range is between about 4200rpm and 7700rpm, corresponding with 1.2 kW to 1.6kW, or 75% load to 100% load for full throttle. These engine speeds are much higher than the values we have seen for minimal specific fuel consumption. It is also interesting, for example, that at 1.5kW, the engine is a bi-stable system. That is, it could run at either 5700 RPM or at 8000 RPM, and the latter is above the recommended speed range—this could be concerning. Assuming that the curve actually extends beyond 8000 RPM, at lower power levels (e.g. 1kW), the system could
be stable at even higher, potentially damaging speeds. At lighter loads, reducing the throttle setting will prevent this as suggested by Figure 2.21.

![Graph showing net torque and net power vs. engine speed for Honda GXH50](image)

Figure 2.24: Honda GXH50 Net torque and Net power vs. Engine speed [5]
2.2.6 Final Design of Throttle Control

The analysis of control loop topology and engine parameter relationships informed our final design of engine control. We decided to control the throttle and sense the voltage going into the battery charger. While the input voltage specification for the battery charger is 4.5V – 38V, we need to build in some margin because of practical matters related to the stability of the engine. Since the battery charger at any given point is delivering a close-loop controlled power into the battery, the input current to the charger module is inversely proportional to the input voltage. Dropping the input voltage too low will cause the current to go very high, overloading the generator; this should be avoided. Therefore, a reasonable voltage range is 23V-35V. The voltage should not go below 23V, as the maximum output is less than 1000W for voltages below 23V. Having the voltage above 23V ensures that the battery charger can always supply maximum power if necessary. A voltage above 35V is dangerously close to the upper range of the battery charger’s specifications. The throttle should be controlled to stay within this voltage range.

Since the charging cycle begins with constant current and a low voltage into the battery, the power begins low, so before the control loop has a chance to take control, the first throttle setting should be relatively low—between 25% and 50%. Once an initial reading is taken, the setting will be adjusted accordingly. This initial setting will only last a few seconds, so specific fuel consumption should not be a concern during this time.

The throttle control loop will continue to maintain the input voltage within the specified range (e.g. 23V – 35V). At the same time we’re controlling the throttle to maintain the voltage, we also want to choose the most fuel-efficient operating point. As was seen in Figure 2.20 and Figure 2.21, the most fuel efficient point is at throttle settings above 50%, therefore we should set the throttle to at least 50% as long as this does not exceed the maximum voltage.

As the charging algorithm continues, the charger will autonomously transition to the constant voltage phase and the power will begin to decrease. When the current drops below the specified threshold, the charger will determine that the cycle is finished and shut off. This will be noted as a jump in input voltage because the load went to zero.
3 Implementation

3.1 Throttle Control Software

After initializing the throttle to between 25% and 50%, the throttle check will ensure that the input voltage to the battery charger remains within the specifications. If the voltage is less than 23V, pulses will be sent to the stepper motor to increase the throttle setting. If the voltage is greater than 35V, pulses will be sent to the stepper to decrease the throttle position. This software code assumes that under normal operation, a 50% throttle, and between 63% and 74% load, the voltage should be between 23V and 35V. This logic will ensure optimal fuel efficiency according to the analysis presented here.

```c
void throttleCheck()
{
    int stateCharging;
    set_adc_channel(5);
    float V=read_adc();

    if (V<23) //voltage control to reach stable threshold
        output_high(stepDir_throttle);
        fprintf(PORT1,"step driver throttle on high\n");
        output_high(stepper_throttle);
        delay_ms(10);
        output_low(stepper_throttle);
        fprintf(PORT1,"step motor throttle on high for 10ms\n");
    } else if (V>35) //safety measure for over-voltage
        output_low(stepDir_throttle);
        fprintf(PORT1,"step driver throttle on low\n");
        output_high(stepper_throttle);
        delay_ms(10);
        output_low(stepper_throttle);
        fprintf(PORT1,"step motor throttle on low for 10ms\n");
    return;
}
```

3.2 Engine Starting and Shutdown Logic

The implementation of an automatic start and shutdown of the engine and charging process required collaboration between the engine team, electronics team, and software team. The challenge with this task was that the engine is designed to have a
manual start, yet we needed to have it start automatically. This required dual functionality of the generator such that it could be used to back-drive the motor. In addition, the ignition was configured to allow the engine to start automatically.

1. The first step is to assume that both the throttle and the choke are completely closed. In order to ensure the proper fuel-air mixture, the choke should be completely closed before the engine is turned on. When we start the engine, we want the richest fuel mixture possible. In addition, the throttle should be completely open, as there is no need to restrict the amount of air and fuel entering the engine. We do not need to change the choke position since it should already be closed, but we need to change the throttle position from completely closed to completely open.

```c
//open throttle
output_high(stepDir_throttle);
fprintf(PORT1,"stepdir throttle high\n");

int i=0;
for(i=0; i<16;i++){
    output_high(stepper_throttle);
    delay_ms(10);
    output_low(stepper_throttle);
    delay_ms(10);
}
fprintf(PORT1,"throttle given 16 steps to fully open it\n");

//choke control to check that it is closed
output_low(stepDir_choke); //backward motion
fprintf(PORT1,"choke stepdir on low\n");
for(i=0; i<16;i++){  //give pulses until closed
    output_high(stepper_choke);
    delay_ms(10);
    output_low(stepper_choke);
    delay_ms(10);
}
fprintf(PORT1,"choke 16 steps step\n");
```
2. After ensuring proper position of the throttle and choke, we are ready to start the engine. We turn on the ignition by opening the ignition “kill switch.” It should be closed since we closed it to turn off the engine at the end of the previous run.

```c
if (!input(PINC1)){ //engine kill
    output_low(PINC1); //assuming normally open
    fprintf(PORT1,"pin c1 low\n");
}
```

3. Next we back-drive the engine using the generator/motor. It should need only about 50 rpm. After this step, it should only take a couple seconds to start. Our program gets a signal when the engine is started by monitoring the output voltage of the alternator. Once the generator stops driving the engine, whether the engine is running is determined with a simple rectifier (diode, cap & resistor) on the output of the alternator, which is monitored by an A/D input on the system controller. A voltage will signal that the engine is running. If it receives this signal, it will open-circuit the synchronous rectifier. This is because we are not yet ready to load the engine, as the engine needs to stabilize.

```c
//send signal to start engine
output_high(startEngine_signal);
fprintf(PORT1,"start signal given to arduino\n");
//check if engine started
output_high(receive_engineStart);
while (!input(receive_engineStart)){ //if low engine has not started
    //wait and do nothing
}
//engine has started
```

4. After two seconds, the generator stops driving the engine. The choke is opened about ⅓ of the way or 30 degrees. After a few more seconds it is opened completely to 90 degrees.
delay_ms(2000);// ideally 2 sec
output_high(stepDir_choke); // forward motion
fprintf(PORT1,"delay 2 sec and choke stepdir high\n");
for(i=0; i<5;i++){
    output_high(stepper_choke);
    delay_ms(10);
    output_low(stepper_choke);
    delay_ms(10);
}
fprintf(PORT1,"choke moved 5 steps\n");
delay_ms(2000);
for(i=0; i<11;i++){
    output_high(stepper_choke);
    delay_ms(10);
    output_low(stepper_choke);
    delay_ms(10);
}
fprintf(PORT1,"choke moved remaining 11 steps\n");

5. After six seconds the throttle is closed to 50% or about 45 degrees. After 1 second, we are finally ready to start the synchronous rectifier and begin the charging process. We let the system settle for a little while before allowing the control loop to take control.

delay_ms(6000);
output_low(stepDir_throttle);
fprintf(PORT1,"6 sec delay and throttle stepdir low\n");
for(i=0; i<8;i++)// closes throttle til
    output_high(stepper_throttle);
    delay_ms(10);
    output_low(stepper_throttle);
    delay_ms(10);
}
fprintf(PORT1,"throttle moved 8 steps\n");
// send signal to start charging

6. To shut down the engine, the system will get a signal that the charging current out of the battery charger has reached below a threshold. This will cause the ignition kill switch to be closed, thus closing the circuit and causing the engine to shut down. After the engine is killed, we will close the throttle completely in order to prepare it for the next cycle and close off the flow of air and fuel to the engine. Finally, we will close the choke
completely such that nothing can enter into the engine while the vehicle is submerged, and the choke is in the proper position for the next charging cycle.

```c
shutdown:
if(remus_sig=="e" || input(fuel_level)){
    // kill engine
    output_high(engine_short); // check once this part is built

    // close choke
    output_low(stepDir_choke); // backward motion
    fprintf(PORT1,"choke low\n");
    for(i=0; i<15;i++){
        output_high(step_choke);
        delay_ms(10);
        output_low(step_choke);
        delay_ms(10);
    }
    fprintf(PORT1,"choke closed 15 steps\n");

    // close throttle
    output_low(stepDir_throttle);
    fprintf(PORT1,"throttle low");
    for(i=0; i<15;i++){
        output_high(step_choke);
        delay_ms(10);
        output_low(step_choke);
        delay_ms(10);
    }
    fprintf(PORT1,"throttle closed 15 steps\n");

    // close engine valve
    output_low(low_p);
```

3.3 Stepper Motors and Mounts

In order to control the angle of the throttle and choke of the engine, we used two stepper motors. Stepper motors are brushless DC motors that split a full rotation into an equal amount of steps and can thus be set to a specified angle. For the choke, we are using the ROB-10551 Small Stepper Motor (shown in Fig. 3.1) rated for a voltage of 12V and 400 mA. The stepper for the throttle was taken from the Honda GXH50 engine.
Figure 3.1: ROB-10551 Small Stepper Motor [12]

To control both stepper motors we are using the Pololu A4988 Stepper Motor Driver Carriers. These chips are small in size (see Fig. 3.2) and have a high power density and monolithic construction. The carriers have 16 pins, of which four connect to the stepper motor. The stepper driver also connects to digital output pins of the controller. One of these pins specifies the step direction and the other one specifies the logical input (i.e. receives pulses to signal the steps). Three of the others pins (MS1, MS2, and MS3) determine the microstep resolution of the stepper driver. For example, setting all three pins to high splits each step from the stepper motor into 16 steps. Setting all three pins to low will keep the full steps of the stepper. Fig. 3.3 shows the connections we used for each pin of the carrier.

Figure 3.2: Pololu A4988 Stepper Motor Driver Carriers [13]
Testing revealed that the small stepper motor for the choke had 48 steps for every rotation. Because the angle between the closed and open positions is 90 degrees, this could be achieved with 12 pulses in our code. To account for error, we added three extra pulses for an entire open-close transition for a total of 15 pulses. Similar testing showed that the stepper motor for the throttle had 144 steps for every rotation.

Figure 3.3: Our pin setup for the Pololu A4988 Stepper Motor Driver Carrier

Fig. 3.4 shows how the carriers are connected to the PIC24H microcontroller.
Figure 3.4: Connections to PIC24H Microcontroller [1]
Finally, a mount was constructed by the engine team using 3D printing to hold both of the stepper motors and connect them to the choke and throttle.

Figure 3.5: 3D-printed mount for stepper motors, throttle, and choke

3.4 Testing and Future Work

Testing our software began with wiring the stepper motors with the choke, throttle, and microcontroller. These tests ensured that the code was functional and determined the step size of each of the stepper motors.

Completion of the engine-generator system was delayed beyond the writing of this document, such that fuel efficiency tests could not be analyzed here. These tests should include measuring fuel consumption by weight. Data should be taken at both varying throttle settings and different loads. Ideally, the battery charger and lithium ion batteries should be used as a load because they act as a constant power load, whereas a resistive load acts as a constant resistance load.
Appendix: Full Software

//skeleton for the main REMUS operating code!
#include <24HJ128GP306.h>
device ICD=TRUE
#fuses HS,NOWDT,PR
#use delay (clock=20000000)
#use rs232(baud=9600, UART1, stream=PORT1)
//use rs232(baud=9600, UART2, stream=PORT2)
#include <math.h>
//pressure sensor and snorkel variables
//#define water1 PIN_F0 //top
//#define water2 PIN_F1 //bottom of tank
#define p_sense PIN_B1 //from pressure sensor analog
#define high_p1 PIN_D0
#define pump PIN_D3
#define high_p2 PIN_D1
#define low_p PIN_D2
#define gyro PIN_B6

//fuel valves and sensors
#define fuel_valve1 PIN_D4
#define fuel_valve2 PIN_D5
#define water_valve1 PIN_D6//b
#define water_valve2 PIN_D7//f
#define air_valve1 PIN_D8
#define air_valve2 PIN_D9
#define flowSensor PIN_B2
#define fuel_level PIN_F2

//engine variables
#define engine_short PIN_F3
#define thermo PIN_B3
#define back_emf PIN_B4 //tells if engine on or off
#define stepper_throttle PIN_G0
#define stepper_choke PIN_G3
#define stepDir_throttle PIN_G1
#define stepDir_choke PIN_G2
#define current PIN_B5
#define home_switch_throttle PIN_G6
#define home_switch_choke PIN_G7
#define startEngine_signal PIN_G8
#define receive_engineStart PIN_G9

//interrupts
#INT_EXT2
#INT_EXT1

char remus_sig='z'; //=''s'';//need to set up keyboard input
//double fuel_flow_rate= exp((( double flow_meter-5)/0.729)-1)/3)
//converted rate from input voltage of flow_meter
float thermoR= thermo;
float temp_gain= 0.0000516131;
float temperature= temp_gain * thermo;
float engine_gain= 0.0000516131;
float engine_bemf;
int stateCharging = 0;
float p_senseR = p_sense;
float num_reads = 100;
float totalTime = 1000; //values will change later
float normTotal = 2.5;
float maxDeviation = 1;
float totalSampleTime = 2000;
float totalDisplaced = 0;
float flowRates[2][100]; // [num_samples, 1] --- also change the one in
sampleFlowRates
int num_samples = 0;
float sampleTime = (1.0 / num_reads) / 60000.0;
float totalFlow = 0.0;
float time = 0.0;
float displaced_fuel;
float fuelDisplaced; // used to store fuel tank function return value
float totalDisplaced_fuel = 0.0; // out of tanks
float totalDisplaced_air = 0.0; // into tanks
float totalDisplaced_water = 0.0;
float Overall_Displaced_fuel_FrontTank = 0.0;
float Overall_Displaced_fuel_BackTank = 0.0;
float TotalFuel_Fronttank = 0.0; // change later
float TotalFuel_Backtank = 0.0; // change later

/******************************************************/
int tempcheck() { // returns true if above safe temp threshold
    set_adc_channel(3);
    thermoR = read_adc();
    temperature = temp_gain * thermoR;
    temperature = 1; // testing only

    if(temperature > 2.8) {
        fprintf(PORT1, "Temperature too high\r\n");
        output_high(PIN_D9);
        return 1;
    }
    fprintf(PORT1, "temp ok\n");
    return 0;
}
/******************************************************/
void engine_check(){
    set_adc_channel(4);
    engine_bemf = read_adc();
    float engine_status = engine_gain * engine_bemf;

    if(tempcheck()){ // temp not ok ==> Need to check if conditional actually works!
        if(engine_status > .1){ // engine on
            fprintf(PORT1, "turned off engine\n");
            output_low(engine_short);
        }
        else{
            fprintf(PORT1, "temp high engine off\n");
        }
    }
else {
    if (engine_status > 0.1) { // engine on
        fprintf(PORT1, "temp low engine running\n");
    } else {
        fprintf(PORT1, "temp ok turn engine on\n"); // engine on
    }
}
return;
}

/*********************************************************
throttleInit() {
    int i = 0;
    output_high(stepDirthrottle);
    fprintf(PORT1, "step driver for the throttle on high\n");
    for (i = 0; i < 2; i++) {
        output_high(stepper_throttle);
        delay_ms(10);
        output_low(stepper_throttle);
        delay_ms(10);
    }
    fprintf(PORT1, "step driver for the throttle moved 2 steps\n");
    // Pulse 2 times to move the throttle to position 10.
    return 1;
}

/***********************************************************/
void throttleCheck() {
    // int stateCharging;
    set_adc_channel(5);
    float V = read_adc();
    V = 20; // testing only

    if (V < 23) { // current control to reach a stable threshold
        output_high(stepDir_throttle);
        fprintf(PORT1, "step driver throttle on high\n");
        output_high(stepper_throttle);
        delay_ms(10);
        output_low(stepper_throttle);
        fprintf(PORT1, "step motor throttle on high for 10ms\n");
    } else if (V > 35) { // safety measure for current overdraw
        output_low(stepDir_throttle);
        fprintf(PORT1, "step driver throttle on low\n");
        output_high(stepper_throttle);
        delay_ms(10);
        output_low(stepper_throttle);
        fprintf(PORT1, "step motor throttle on low for 10ms\n");
    }
    return;
}

/***********************************************************/
void waterCheck() {
    set_adc_channel(6);
    float gyroState = read_adc();
gyroState=3;//testing only
if (gyroState > 2.5) {//if upright
  if(input(PIN_F0)){//upright and water1 activated
    output_high(pump);
    fprintf(PORT1,"pump on, gyro state interrupt triggered\n");
    output_high(PIN_D8);///interrupt
  }
  else if (!input(PIN_F0)) {//upright and water1 not activated
    output_low(pump);
    output_low(low_p);// turn low pressure pump back on
  }
  fprintf(PORT1,"water check done. ok.\n");
}
else {//if upside down keep pump off
  output_low(pump);
  output_low(low_p);
  fprintf(PORT1,"choke 16 steps step\n");
  fprintf(PORT1,"Remus upside down. low pressure valve closed. pump
off.\n");
}
return ;
}

/****************************fuel tank
functions****************************/

float flow_rate (float num_reads, float totalTime, float normTotal,float
maxDeviation){
  /* From sensor code converted to C:
    num_reads: number of samples per second
    TotalTime: total time for a reads of the sensor
    NormTotal: norm value ?
    MaxDevation: max deviation we'll allows the sensor to read...
    My addition: Returns average sensor value and fluidtype -- I need to
    figure out how using one sensor for all the fluids and having water/air
    flowing at the same time as fuel works... I'll talk to Tess
   */

  //assign flow sensor pin to ADC port:
  set_adc_channel(5);

  float sampleTime = (1.0/ num_reads) / 60000.0;
  float totalFlow = 0.0;
  float time = 0.0;

  while (time < totalSampleTime){
    float V_out = read_adc();
    float Flow_Rate = 0.0;
    if (V_out > 5.0){ //means positive flow
      Flow_Rate = (float)(exp((V_out - 5.0)/.729) - 1.0) / 3.0;
    }
    if (V_out < 5.0){
      Flow_Rate = (float)-(exp((V_out - 5.0)/.729) - 1.0) / 3.0;
    }
    totalFlow += Flow_Rate * sampleTime; //maybe change
    time += sampleTime;
  }
if ((totalFlow - normTotal)/ maxDeviation >= 2){
    fprintf(PORT1," error! above 2 standard deviations\n");
    return -1; //means there is an error
}
return totalFlow / (totalSampleTime * sampleTime); //return average value the sensor read in this total sampling time
}

void Switch_Fluid(char fluid_on, char fluid_off, char tank){
    /*
    fluid_on and fluid_off: A or W, where A = air, W = water
    tank: char F or B, where F is front tank and B is back tank
    returns None
    switches between air and water in one of the tanks...
    Tess thinks we should do this every minute or so to keep everything at equilib */

    if (fluid_on == fluid_off){
        return;
    }
    if (fluid_on == "A"){
        if (tank == "F"){
            output_high(air_valve2);
            output_low(water_valve2);
        }
        else{
            output_high(air_valve1);
            output_low(water_valve1);
        }
    }

    else/* fluid_on == "W */{
        if (tank == "F"){
            output_high(water_valve2);
            output_low(air_valve2);
        }
        else{
            output_high(water_valve1);
            output_low(air_valve1);
        }
    }
    return;
}

float sampleFlowRates(totalTime, sampleTime, num_samples){
    /*TotalTime: int total time for sampling
    SampleTime: int duration of one sample
    this function calculates the average flow rate
    */
num_samples: int number of samples

initializes a 2-D array, samples flow rate values, and adds flow rate values and SampleTimes to the array*/

float flowRates[2][100]; // [num_samples, 1]
int time = 0;
int count = 0;

while (time < totalTime){
    float rate = flow_rate (num_reads, totalSampleTime, normTotal, maxDeviation);
    flowRates[0][count] = rate;
    flowRates[1][count] = totalSampleTime;
    count += 1;
    totalTime += totalSampleTime;
}

return flowRates;
}
float calc_displaced(){
    /*
    FlowRates: array of (average flow rate, sample time)
    returns volume of flow that went through the sensor in time
    calculates volume by integrating flow rate over time
    */

    int i;
    float totalDisplaced = 0;
    for (i=0; i < 100; i++){ // cycles through all entries in FlowRates array
        float flowRate = flowRates[i][0];
        float TIME = flowRates[i][1];
        totalDisplaced += flowRate*TIME;
        // flowRate != array flowRates /* in this round of calculating*/
    }

    return totalDisplaced;
}
void Switch_Tank(char tank_off, char tank_on){
    /*
    tank_off and tank_on: char F or B, where F is front tank and B is back tank
    returns None
    switches a fluid's valve it's partner valve in opposite tank */

    if (tank_off == tank_on){
        return;
    }

    if (tank_on == "F"){
        // turn off valves to back tank
output_low(fuel_valve1);
output_low(air_valve1);
output_low(water_valve1);

// turn on fuel valve to front tank
output_high(fuel_valve2);
output_low(air_valve2);
output_low(water_valve2);
}
else/* tank_on == "B" */{
  // turn off valves to front tank
  output_low(fuel_valve2);
  output_low(air_valve2);
  output_low(water_valve2);
  // turn on fuel valve to back tank
  output_high(fuel_valve1);
  output_low(air_valve2);
  output_low(water_valve2);
}
return;

/*********************** interrupts
***************************/

void EXT1_isr() {
  if(input(PIN_C1)) {
    fprintf(PORT1,"interrupt snorkel\n\n");
    output_low(low_p);
    // delay_ms(50);
    fprintf(PORT1,"low pressure valve closed\n\n");
  }  
clear_interrupt(PIN_D8);
}

void EXT2_isr() {
  if(tempcheck()) {
    output_high(engine_short);// engine out
    fprintf(PORT1,"engine off\n\n");
  }  
clear_interrupt(PIN_D9); // interrupt pin
}

/******************************main
function******************************/

void main (void){
  #use STANDARD_10 (G)
  #use STANDARD_10 (F)
  #use STANDARD_10 (C)
  #use STANDARD_10 (D)
  setup_adc_ports(ALL_ANALOG);
  setup_adc(ADC_CLOCK_INTERNAL);
  // later we will need to write code to read the values of every
  sensor, valve, pump
  // also need to set up a timer
while (remus_sig=='z'){
    if(kbhit()){
        remus_sig=getc();
        delay_ms(100);
       putc(remus_sig);
    }
}

while (remus_sig=='s'){
    output_toggle(PIN_B4);//testing
    enable_interrupts(INT_EXT2);
    int pressure_count=0;
    //output_high(flowSensor);//testing only

    start:
    
    /*********************************************************
    start up
    **********************************************************/
    //pressure sensor code
    p_senseR= 1;//set to 1 for testing only

    if(p_senseR < 2 && p_senseR >= 0) { //if pressure in range
        output_high(high_pl);//open high pressure valves and pump
        fprintf(PORT1,"pressure 1 high\n\n");
        output_high(pump);
        fprintf(PORT1,"pump high\n\n");
        output_high(high_p2);
        fprintf(PORT1,"pressure 2 high\n\n");
    }
    else { //exit program
        if(pressure_count >= 100,000){
            fprintf(PORT1,"exiting program\n");
            remus_sig='e';
            goto end;
        }
        pressure_count++;
    }
    //output_high(water1);//testing only!
    //output_high(water2);//testing only

    firstWC:
    if (input(PIN_F0) && input(PINF1)){
        output_high(pump);
        fprintf(PORT1,"pump high\n\n");
    }
    else{
        goto firstWC;
    }

    output_high(low_p);
    //output_low(PIN_C1);//testing
    //output_high(PIN_B4);
    fprintf(PORT1,"low pressure valve high, air to engine ok\n\n");
enable_interrupts(INT_EXT1);

//engine starting!!!
if (!input(PINC1)){ //engine kill //if its not open, open it. close to kill engine
    output_low(PINC1); //assuming normally open
    fprintf(PORT1,"pin c1 low\n");
}

output_high(stepDir_throttle);
fprintf(PORT1,"stepdir throttle high\n");

int i=0;
for(i=0; i<16;i++){
    output_high(stepper_throttle);
    delay_ms(10);
    output_low(stepper_throttle);
    delay_ms(10);
}
    fprintf(PORT1,"throttle given 16 steps to fully open it\n");

    //then give 16 pulses in the other direction until open
    output_high(stepDir_throttle);
    fprintf(PORT1,"throttle stepdir high\n");
for(i=0; i<16;i++){
    output_high(stepper_throttle);
    delay_ms(10);
    output_low(stepper_throttle);
    delay_ms(10);
}
    fprintf(PORT1,"throttle 15 steps\n");

    //choke control to make sure it is closed
    output_low(stepDir_choke); //backward motion
    fprintf(PORT1,"choke stepdir on low\n");
for(i=0; i<16;i++){
    //give pulses til closed
    output_high(stepper_choke);
    delay_ms(10);
    output_low(stepper_choke);
    delay_ms(10);
}
    fprintf(PORT1,"choke 16 steps step\n");

    //send signal to start engine
    output_high(startEngine_signal);
    fprintf(PORT1,"start signal given to arduino\n");
    //check if engine started
    output_high(receive_engineStart); //testing only!
while (!input(receive_engineStart)){ //if low engine has not started
    //wait and do nothing
}

//engine has started

delay_ms(2000);//ideally 2 seconds
output_high(stepDir_choke); //forward motion
fprintf(PORT1,"delay 2 sec and choke stepdir high\n")
for(i=0; i<5;i++){
    output_high(stepper_choke);
    delay_ms(10);
    output_low(stepper_choke);
    delay_ms(10);
}
fprintf(PORT1,"chokemoved 5 steps\n");

delay_ms(2000);
for(i=0; i<11;i++){
    output_high(stepper_choke);
    delay_ms(10);
    output_low(stepper_choke);
    delay_ms(10);
}
fprintf(PORT1,"choke moved remaining 11 steps\n"
)

delay_ms(6000);
output_low(stepDir_throttle);
fprintf(PORT1,"6 sec delay and throttle stepdir low\n"
for(i=0; i<8;i++) //closes throttle til
    output_high(stepper_throttle);
    delay_ms(10);
    output_low(stepper_throttle);
    delay_ms(10);
}
fprintf(PORT1,"throttle moved 13 steps\n"
)

//may need to send signal to start charging
throttleInit(); //put throttle to 5

fprintf(PORT1,"engine starting complete\n"
)

/******************* Engine fuel tank on *************/
engine_bemf=2;//testing only
while(engine_bemf>1){ //engine on
    //safety checks
    waterCheck();
    engine_check();
    //tempcheck();
    throttleCheck();

/*******************Front Tank On**********************/
switch1:
//fill front tank with 7ml air and 3ml water and supply engine 10ml of fuel

totaldisplaced_fuel = 0.0; // out of tanks
totaldisplaced_air = 0.0; // into tanks
totaldisplaced_water = 0.0;

fprintf(PORT1,"start switching\n");

//turn on front tank
Switch_Tank(/*from*/'B', /*to*/ 'F');

totaldisplaced_fuel=0;//testing only
Overall_Displaced_fuel_FrontTank=1;//testing only
TotalFuel_FrontTank=3;//testing only
fprintf(PORT1,"front tank done switching\n");

while (totaldisplaced_fuel < 10.0){ // this loop should take about 2 minutes or so real time

    //make sure fuel in front tank
    //switch to back if front runs out of fuel
    if(Overall_Displaced_fuel_FrontTank >= TotalFuel_FrontTank){
        goto switch2;
    }

    //make sure the right valve is open:
    if (totaldisplaced_water < 3.0) {
        //open water valve in front tank
        Switch_Fluid('W' /*on*/, 'A' /*off*/, 'F');
    } else /*totaldisplaced_water > 3.0 && totaldisplaced_air < 7.0*/{

        //turn on air valve:
        Switch_Fluid('A', 'W', 'F');
    }

    //safety checks:
    waterCheck();
    engine_check();
    //tempcheck();
    throttleCheck();

    //read flow sensor
    sampleFlowRates(totalTime, sampleTime, numsamples);

    //calculate volume fuel displaced
    displaced_fuel = calc_displaced();
    totaldisplaced_fuel += displaced_fuel;
    Overall_Displaced_fuel_FrontTank += displaced_fuel;

    //keep track of total water or air displaced this round of the loop
    if (totaldisplaced_water < 3.0){
        totaldisplaced_water += /*displaced_water*/ displaced_fuel;
    }
else /*totaldisplaced_water > 3.0 && totaldisplaced_air < 7.0*/{
    totaldisplaced_air += /*displaced_air*/ displaced_fuel;
}

if (totaldisplaced_fuel >= 10.0) { //necessary?
    goto switch2;
}

//switch tanks if front tank runs out of fuel
if (Overall_Displaced_fuel_FrontTank >= TotalFuel_FrontTank) {
    goto switch2;
}
} //end of front tank cycle

/*****************Back Tank On*******************/

switch2:
//now fill back tank with 7ml air and 3ml water and supply engine 10ml of fuel

    totaldisplaced_fuel = 0.0; // out of tanks
    totaldisplaced_air = 0.0; // into tanks
    totaldisplaced_water = 0.0;

//turn on back tank and turn front tank off
Switch_Tank(/*from*/ 'F', /*to*/ 'B');

while (totaldisplaced_fuel < 10.0) { // this loop should take about 2 minutes or so real time

    //make sure fuel in back tank
    //switch tanks if back tank runs out of fuel
    if (Overall_Displaced_fuel_BackTank >= TotalFuel_BackTank) {
        goto switch1;
    }

    //make sure the right valve is open:
    if (totaldisplaced_water < 3.0) {
        //open water valve in front tank
        Switch_Fluid('W' /*on*/, 'A' /*off*/, 'B');
    } else /*totaldisplaced_water > 3.0 && totaldisplaced_air < 7.0*/{
        //turn on air valve:
        Switch_Fluid('A', 'W', 'B');
    }

    //safety checks
    engine_check();
    tempcheck();
    throttleCheck();

    //read flow sensor & calculate volume fuel displaced
    sampleFlowRates(totalTime, sampleTime, num_samples);
displaced_fuel = calc_displaced();
total_displaced_fuel += displaced_fuel;
Overall_displaced_fuel_Ba ckTank += displaced_fuel;

// Keep track of total water or air displaced this round of the loop
if (total_displaced_water < 3.0){
    total_displaced_water += /*displaced_water*/ displaced_fuel;
} else /*total_displaced_water > 3.0 && total_displaced_air < 7.0*/{
    total_displaced_air += /*displaced_water*/ displaced_fuel;
}

if (total_displaced_fuel >= 10.0){ //necessary? -- WANTED?
    goto switch1;
}

} // End of back tank cycle

shutdown:
if(remus_sig="e" || !input(fuel_level)){

    // Kill engine
    output_high(engine_short); // check once this part is built

    // Close choke
    output_low(stepDir_choke); // backward motion
    for(i=0; i<15;i++){
        output_high(stepper_choke);
        delay_ms(10);
        output_low(stepper_choke);
        delay_ms(10);
    }
    fprintf(PORT1,"choke closed 15 steps\n");

    // Close throttle
    output_low(stepDir_throttle);
    fprintf(PORT1,"throttle low\n");
    for(i=0; i<15;i++){
        output_high(stepper_choke);
        delay_ms(10);
        output_low(stepper_choke);
        delay_ms(10);
    }
    fprintf(PORT1,"throttle closed 15 steps\n");

    // Close engine valve
    output_low(low_p);

    // Close all fuel tank valves
    output_low(fuel_valve1);
    output_low(fuel_valve2);
    output_low(water_valve1);
    output_low(water_valve2);
    output_low(air_valve1);
output_low(air_valve2);

//wait while water is going into the plenum
while(!input(PIN_C1) && !input(PIN_C2)){
    output_high(high_pl); //ask snorkel about this
    output_high(high_p2);
}
output_low(high_pl);//close high pressure valves
output_low(high_p2);

fprintf(PORT1,"system shut down complete");

//****************************************************************************
end:
break;
}
}
}
References