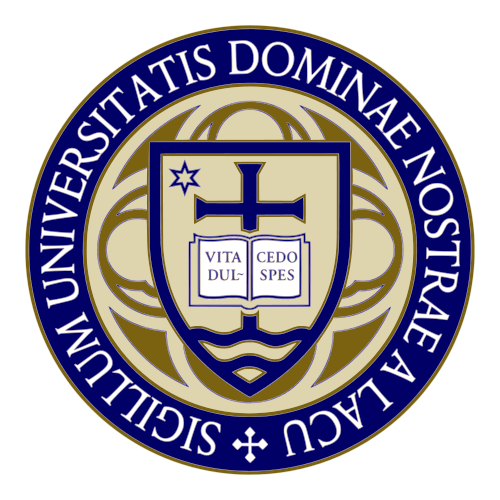
Department of Electrical Engineering

University of Notre Dame



Light Bike

Battery Charger, Peripheral Controls, and Motor Controller for an Electric Motorcycle

Pat Bowlds

Ben Coffey

Mike Mellitt

Jake Thordahl

Alex Toombs

May 9th, 2013

# 

# Table of Contents

**1. Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . . . . .4

1.1 High Level Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .6

1.2 Expectation and Reality . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . …. . . . . . . .7

**2. Detailed System Requirements** . . . . . . . . . . . . . . . . . . . . . . . . .. . . . . . . . . . . . . . .8

2.1 Overall System . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ... . . . . . . . . . . . . . 8

2.2 Battery Charger . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . . . . . . . . . . . . . 9

2.3 Motor Controller . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . . . . . . . . . . .10

2.4 User Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

2.5 Regenerative Braking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ... . . . . . . . . . 11

**3. Detailed Project Description** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .... . . . . . . . . . .12

3.1 System Theory of Operation . . . . . . . . . . . . . . . . . . . . . . . . . . ... . . . . . . . . .12

3.2 System Block Diagram . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . … . . . . . . . .13

3.3 Subsystem Testing. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . . . . . . . .14

3.4 Detailed Design/Operation of Subsystem 1 . . . . . . . . . . . . . ….. . . . . . . . . .14

3.4.1 **Battery Stack** . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. .. . . . . . . . . . .14

3.4.2 **Wall AC to DC Converter** . . . . . . . . . . . . . . . . . . . . …. . . . . . . . . 15

3.4.3 **Voltage Divider and Temp Sensor**. . . . . . . . . . . . . …... . . . . . . . 15

3.4.4 **Voltage Regulators** . . . . . . . . . . . . . . . . . . . . . . . . . . …. . . . . . . . 16

3.4.5 **Charging Circuit Controller** . . . . . . . . . . . . . . . . . . . ….. . . . . . . 16

3.5 Detailed Design/Operation of Subsystem 2. . . . . . . . . . . . . . . . ….. . . . . . . 18

3.5.1 **Peripheral Controller Circuit** . . . . . . . . . . . . . . . . . . ….. . . . . . . .18

3.5.2 **User Interface** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ... . . . . . 18

3.6 Detailed Design/Operation of Subsystem 3. . . . . . . . . . . . . . . . . ….. . . . . .19

3.6.1 **Motor Controller** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .... . . . . .19

3.6.2 **Motor**. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . . . . . . 20

3.6.3 **Regenerative Braking** . . . . . . . . . . . . . . . .. . . . . . . .. . . . …. . . . . 20

3.7 Interfaces . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . . . . . . . 21

**4. System Integration Testing** . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . . . . . . . . . . . . 21

**5. Users Manual/Installation Manual**. . . . . . . . . . . . . . . . . . . . . … . . . . . . . . . . . . . . 23

5.1 Proper Charging Procedure . . . . . . . . . . . . . . . . … . . . . . . . . . . . . . . . . . . 24

5.2 LightBike Products . . . . . . . . . . . . . . . . . . . . . . . ... . . . . . . . . . . . . . . . . . . . 25

5.3 Product Choice Guide . . . . . . . . . . . . . . . . . . . . . . ... . . . . . . . . . . . . . . . . . 25

5.4 Installation Guide . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .26

5.5 DIY bundle information . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .28

5.6 Troubleshooting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ... . . . . . . . . . . . . . . . 29

**6. To-Market Design Changes** . . . . . . . . . . . . . . . . . . . . . . . . . ….. . . . . . . . . . . . . . . 31

**7. Recommendations for Project Continuation** . . . . . . . . . ……. . . . . . . . . . . . . . . .33

**8. Conclusions** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ...... . . . . . . . . . . . . . 34

**9 Appendices** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . … . . . . . . . . . . . . . . . .35

A: Charging Circuit Code . . . . . . . . . . . . . . . . . .. . . . . . . . …. .. . . . . . . . . . . . . .35

B: Peripheral Board Code . . . . . . . . . . . . . . . . .. . . . . . . . …. . . . . . . . . . . . . . .53

C: Charging Board Schematic. . . . . . . . . . . . . . . . . . …. . . .. .. . . . . . . . . . . . . . 65

D: Charging Board File . . . . . . . . . . . .. . . . . . . . . . … . . . . . .. . . . . . . . . . . . . . 68

E: Peripheral Board Schematic. . . . . .. . . . . . . . . . . …. . . . . .. . . . . . . . . . . . . . 69

F: Peripheral Board FIle. . . . . . . . . . . . . . . . . . .. . . . ... . . . . . .. . . . . . . . . . . . . .69

G: Motorcontroller Schematic . . . . . . . . . . . . . .. . . . .... . . . . . . . . . . . . . . . . . . 70

H: Component Datasheet Links. . . . . . . . .. . . . . . . . . . …. . . . . . . . . . . . . . . . . 70

# 

# 1. Introduction

Gasoline power vehicles have been the norm since before even Henry Ford’s Model T began rolling off assembly lines in the early 20th century. Refined gasoline, sourced from fossil fuels, has remained inarguably the best transportation fuel source due to its high power and energy densities per weight and volume compared to any other fuel source, meaning that a tank full of gasoline can take you much further than an equivalent weight or size of almost any other material.

However, gasoline has become an unsustainable fuel source over the past few decades as fossil fuels have been found to be nonrenewable at a pace that the world can keep up with. As development continues and demand for petroleum-based products continues to rise, pollution levels in developing nations become a concern thanks to the byproducts of petroleum refinement and consumption. That demand cannot be met with the reserves of oil we now have, meaning that soon an alternative must be found.

As long as transportation remains important to us, the energy we use for it must be at the forefront of thought. Now, like never before, is the right time to consider hybrid and electric vehicle systems. Vehicles that are partially or wholly electric, including the Toyota Prius, Chevy Volt, and Tesla Model S are seeing unprecedented commercial success. The materials for high energy and power density batteries are being developed right now, with new and improved technologies coming out annually while gasoline engines stagnate. The goal to prove that both cars and even largely recreational vehicles like motorcycles can be powered by batteries led to the genesis of the electric motorcycle project between the Department of Engineering and Department of Energy at the University of Notre Dame over half a decade ago. As the public becomes more environmentally conscientious and a desire for alternative fuels grows (with less pollution) so too does the desire for a battery based electric vehicle.

The main goal of this project is to enhance the charging system for the existing electric bike. The former group who worked on the bike had a less than ideal charging system that rectified wall voltage at 120 V AC 60 Hz three-phase power down to a suitable voltage for their project. Eventually, this charging led to the damaging of the Optima Yellow Top batteries that we used, meaning that we had to spend a lot of money (approx $1200) just to replace the batteries the motorcycle used before beginning to test our project. Taking the previous groups’ frame, mounting system, batteries, and motor controller, our goal was to improve the charging and peripheral display subsystems to a point where the vehicle could be used more frequently.

Our new charger had the goal of following the specced charging profiles as per Optima’s website while allowing for a shorter charge time and a reconditioning period that should extend the lifetime of the batteries. At an average cost of $160 per battery while using a total of six batteries in the stack, preserving lifetime is of the utmost importance. Well-maintained, a stack of batteries in a Tesla Model S can last up to 10 years. A great deal of engineering goes into allowing those batteries to safely charge and discharge in a way that provides a stable ride while allowing the batteries to live indefinitely.

While electric vehicles are still far from the norm, sound and safe charging practices can go a long way toward making batteries more feasible. Our exercise in controls has shown the need for more reliable and accurate charging systems. A motorcycle provides a unique opportunity to prototype an electric vehicle due to its small size, short usage intervals, and low cost to maintain. As the world turns more toward electrical systems to meet its energy needs, well-designed and precisely engineered battery systems will be necessary to supply the power that the world demands.

*1.1 High Level Description*

Overall, the project is comprised of several important subsystems. The systems are the power supply, charging circuit, motor controller and user interface. The power supply steps down voltage from a standard wall outlet and rectifies it to DC. The supply output is 96V and limited to 10A.

This charger is designed to charge the full stack of six Optima Yellow Top batteries according to specifications. The motorcycle has six deep cycle 12V lead acid batteries connected in series for a total of 72V and 38 Ah. Optimal charging for the batteries in this configuration is to apply a constant voltage of 88.2V to the stack until the current drops below 1A. At this point the charger switches to a current controlled source at 2A for 1 hour. If the temperature of any of the batteries goes over 51.7 Celsius then the charger needs to cut off until the temperature is back within a safe range.

The third subsystem of our project is the motor controller. Our original plan was to use the Kelly motor controller that was purchased previously by the original bike group. The motor controller is a commercially available product that handles the full battery stack and regulates the power given to the motor. The controller gets input from the user through two potentiometers, throttle and braking. Unfortunately, this motor controller broke. A current source fed from a single 12V batteries was used in replacement. This system only takes input from the throttle so the bike’s mechanical brake has to be used.

Our last subsystem dealt with how the user would be able to interface with the bike. This board takes inputs from the user via switches and buttons and displays information on an LCD display. This subsystem communicates with both the charging board and motor controller in order to display pertinent information such as the bikes current state (charging or running) and battery voltage. The board disables the motor controller if the bike is connected to a wall outlet or if the bike is off. It also controls the bike’s tail and plate lights.

*1.2 Expectations and Reality*

The power supply creates the expected 96V and when the charging circuit is in full use has only 2V of ripple. The battery charger works as anticipated and transitions between the different charging protocols correctly. We were unable to test a complete charge with all of the batteries at empty because there was not an efficient way to discharge all of them. With all of the batteries slightly discharged the charger went through both the normal and trickle charge states without temperature or current issues and returned the stack to full charge.

The motor controller is able to spin the wheel when the motorcycle is elevated. The mosfets, however, cannot handle the current spike when the bike is ridden and blow out. The motor draws a large current when transitioning from stationary to rotating and this transition is prolonged when weight is added to the bike. This can be improved by using more mosfets in parallel and ones rated for higher current. The best solution would be to construct a controller that adopts the known technologies of mosfet H bridge and pulse width modulation, however, we obviously did not have the time or the parts available to implement this into our prototype.

The user interface works as expected and accurately displays information to the LCD screen. The battery voltage displayed is usually within 2V of the actual and so could be further refined through trial and error within the logic of the peripheral board.

# 2. Detailed System Requirements

*2.1 Overall System*

Having acquired new batteries, our first and foremost goal as well as the greatest portion of our project will be to provide a safe and efficient wall socket based charging circuit that stays true to the recommended charging profiles of the 12V Optima Batteries. Charging will be accomplished with all 6 batteries in series, which is seen as “one battery” in electricity perspective, and will involve both a constant voltage phase as well as a constant current “trickle charge” phase.

An easy to read and intuitive user interface will be installed, displaying the capacity of the battery (as % remaining) and be a functional odometer and speedometer. A light will turn on to indicate when the batteries are getting low and the batteries will gradually cut off as a safety restraint if the voltage of the battery drops too low. The battery is capable of 72 Volts. Finally, any mechanical changes must be optimized; including the removal of the frame from the rear of the motorcycle originally used to mount a gas generator.

The promise of future street legality must be kept in mind and the beginnings of code to allow for such a design accounted for in the programs. It will also be important to note that the motor controller will need to be designed/purchased in future groups as the one provided has finally reached the end of its lifetime. This means enough room for mounting future motor controllers must be left open on the bike’s frame.

*2.2 Battery Charger*

A battery charger will need to be implemented to quickly and safely charge the batteries. The batteries needs to charged in a way that is safe for both health of the battery and user . The battery charger must remain true to the recommended Optima Battery charging profiles.

The charging profile for Optima Batteries is a two phase charge and reconditioning. The first phase requires constant voltage at 88 V DC. This requires a step-down transformer, capacitor bank, and regulator for a “clean” DC source with very little voltage fluctuation. This phase will take as long as necessary and current will be measured at the battery stack to ensure dangerous levels are not reached. If current breaks a certain point, or if temperature rises too high, the battery controller must respond and shut off power. The second phase requires constant current for an hour. This phase prolongs battery life and serves as a conditioner to prevent battery damage. The controller will turn off automatically after this stage. For this setup we will require a way to accurately measure both the current and voltage and adjust their values as needed. There will be multiple conversions from analog to digital and vice versa in order for a microcontroller to interact with this system. The microcontroller needs to act as a state machine. Additionally there will need to be a way to accurately read the temperature of each battery. All logic should favor safety, with both current levels and temperature, over a fast charge.

*2.3 Motor Controller*

Although this was not a focal point for the project, Team Light Bike wanted a basic motor controller operational both for demonstration and for testing the effectiveness of the designed battery charger and user interface. Once the original Kelly Controller died, a new one must be designed so as to allow for acceleration as well as maintaining speed based off of a handlebar potentiometer. This design must handle fairly high currents constantly running at approximately 40A to 60A as well as handle spikes of current between 150A to 180A. Additionally, there needs to be a safe way to cut power to the system in case of failure. If possible, this controller must be able to handle the load of at least the tire itself so as to demonstrate a basic understanding of how motor controllers work and how to effectively design one.

*2.4 User Interface*

For our product to be both lucrative and easy to use (maximum target audience) we will need an intuitive and simple user interface. This will require both a microcontroller to read voltages and currents but also the ability to output and display values to a GUI on an LCD screen for the user to use. We have to ensure that the data sent is sent accurately and displayed correctly on the LCD screen: we can’t have the display showing batteries at 50% when, in reality, they are running dry. This is both a safety and a convenience feature as running these batteries dry is dangerous and fatal for the device. It will require a large number of output ports to read the state of switches and buttons as well as logic in order to respond to the user. The microcontroller needs a way to communicate to both the charge and motor controller to ensure that the motor is never operational when the bike is switched off or charging. There will need to be a way to drive the 12V tail and plate light from the board. Finally, using a mechanical reader on the wheel, we can calculate rotations and, from rotations/s, calculate both speed and the distance traveled. Finally, due to the naturally inclement weather during the winter and spring seasons, the LCD screen must be waterproof.

It is also important to make sure the HUD is easy to read. Since the bike can operate at speeds upwards of 50 miles per hour, it will need to display relevant information in a easy to comprehend screen. Making the user stare at an LCD while driving the motorcycle would cause serious safety concerns.

*2.5 Regenerative Braking*

Finally, to maximize efficiency and preserve the batteries, we plan on incorporating regenerative braking as part of our optimization process. This can add an incredible amount of energy back into the system, increasing efficiency by up to 20% for high quality systems. This was not planned to be a major feature of our vehicle, but is one that was taken seriously and a design opportunity to add serious value. Many motor controllers have this feature “built in” and simply need to be calibrated via simple software solution. However, we believe that an advanced motor controller with regenerative braking is a great opportunity for future senior design groups.

# 

# 3. Detailed Project Description

## 3.1 System Theory of Operation

Our system operates through the combination of charging, peripheral, and motor controller subsystems. The general theory of the design is to charge the on bike stack of batteries according to the Optima. The charging system is connected to a wall outlet, and charges the stack of batteries using mosfets as a voltage controlled current source. Through the use of an I2C line in conjunction with an 8-bit DAC (Datasheet link available in appendix H) on the charging board the voltage across the gate of the mosfets can be varied, allowing us to charge the batteries at the current we desire. The DAC can output a maximum of 5V and so is scaled up using a op-amp with a gain of 2. Feedback changes the output to the DAC for precision control.

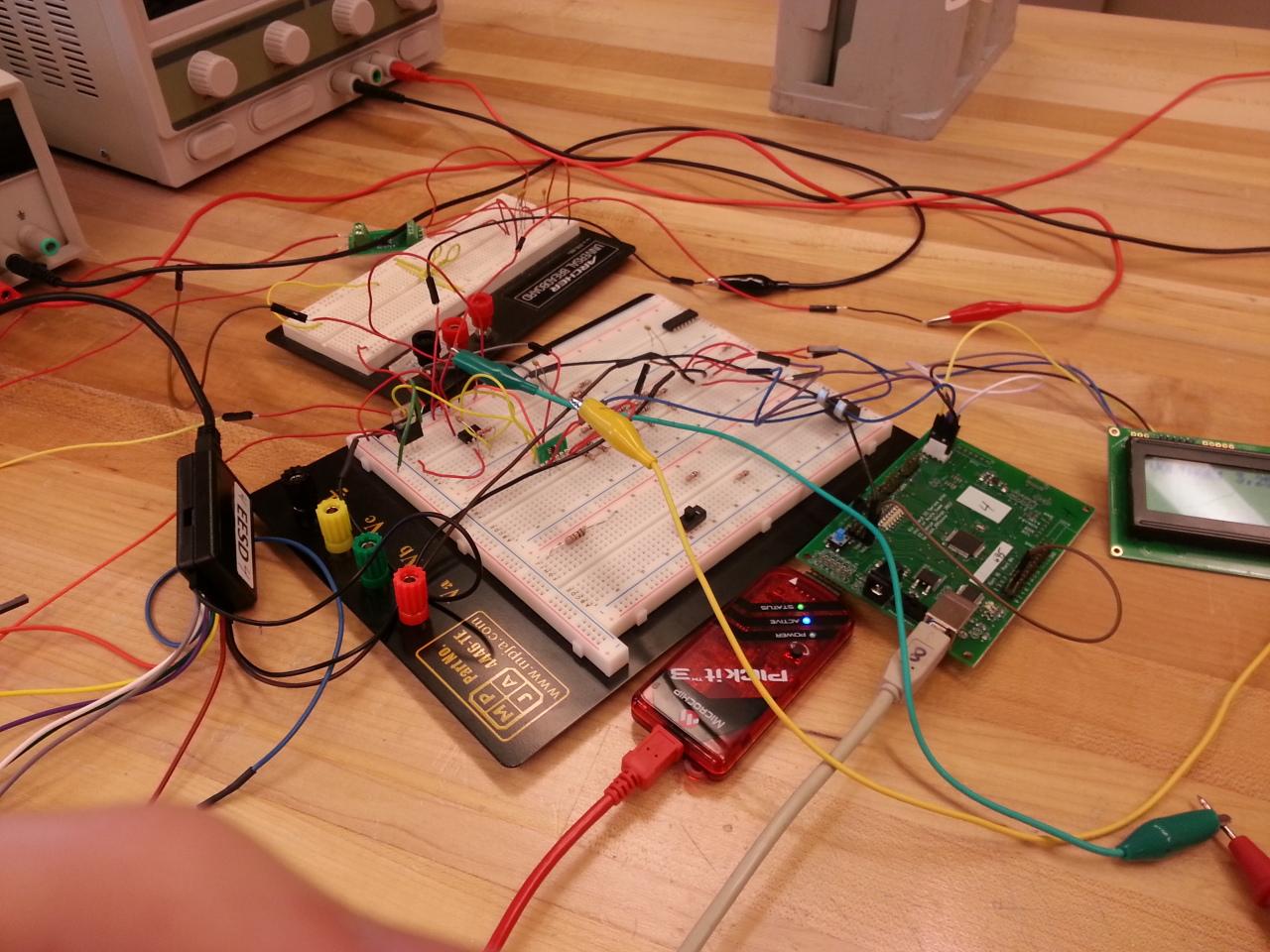


Figure 1 - Prototype Charging Circuit

Readings are taken from the battery stack and charging board via temperature sensors and volt meters, and are used with the peripheral board to display important information through another I2C line to the LCD heads up display. The motor controller drives the motor based on the position of the handlebar throttle allowing for typical motorcycle operation, and the peripheral board allows the on bike switches to be used to turn the bike on, as well as turn on the different indicators found on a typical motorcycle; brake and warning lights, turn signals, and license plate spotlight.

## 3.2 System Block Diagram

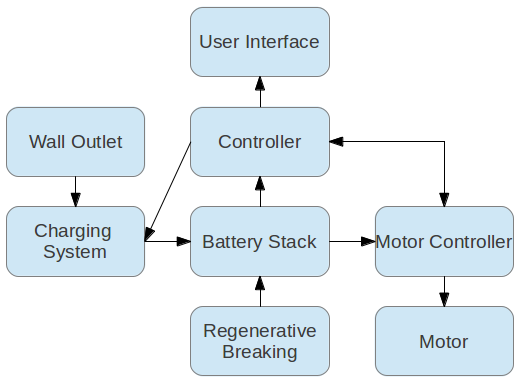


Figure 2 - Control System Diagram Overview

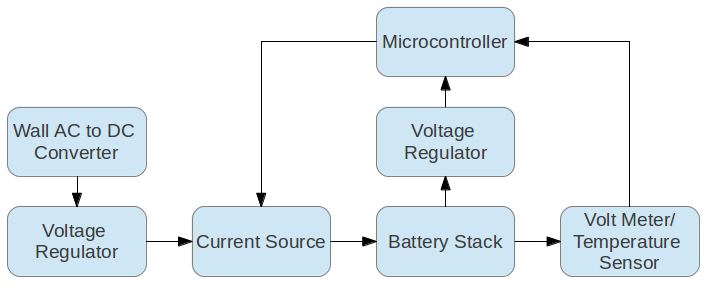


Figure 3 - Charging Circuit Subsystem Overview

*3.3 Subsystem Testing*

Testing the subsystems was primarily limited to using a motor to sink the current while the charging board was being tested. It gave the opportunity to test the individual pieces of the board while giving us a simulated battery.

The motor and original motor controller were tested once the old dead batteries were replaced with the new ones.

*3.4 Detailed Operation of Subsystem 1- Charging*

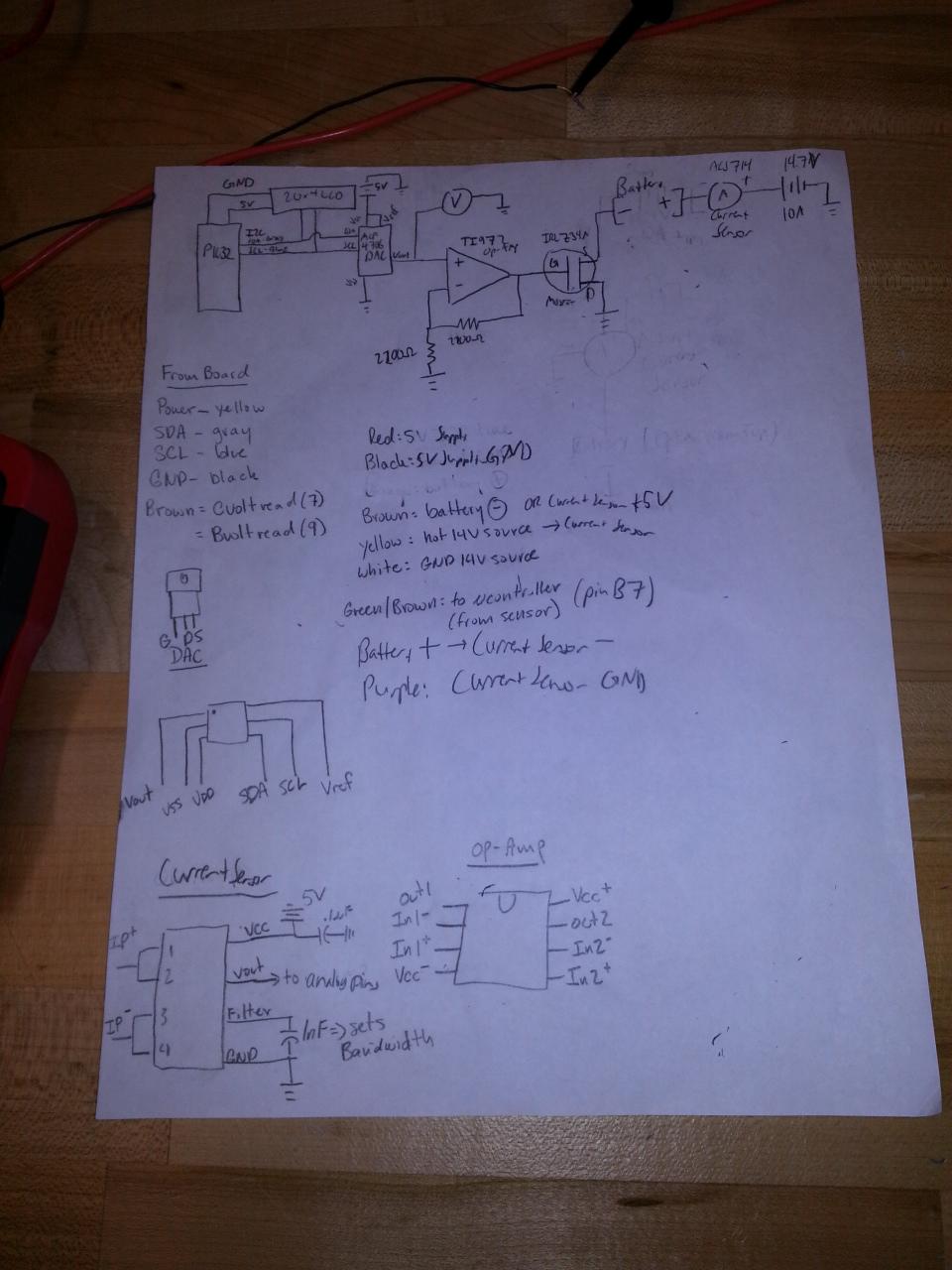


Figure 4 - Early Subsytem 1 Schematic

**3.4.1 Battery Stack**

The battery stack consists of six 12V Optima Yellow Top D51 (Datasheet link available in appendix H) deep-cycle lead acid batteries connected in series with a full charge capacity of ~87V. These batteries are rated at 37Ah (amp hours). They are oriented on the bike to allow easy connections between them, minimizing danger while hooking them up.

**3.4.2 Wall AC to DC Converter**

The conversion of the wall socket voltage of 120V AC at 60Hz is rectified to our required voltage of 68V rms which is rectified to 96V DC handled by our on bike transformer, a full bridge rectifier and a 47 mF capacitor bank. This is necessary to provide the batteries with a suitable DC voltage for charging. We discovered that when the bike was not plugged into the wall the batteries would charge the capacitors at an amperage greater than 10A, the amperage rating of our fuse, and this was why we were blowing fuses. In order to prevent this a power diode should be added between the capacitor bank and the battery stack.

**3.4.3 Voltage Divider and Temperature Sensor**

Battery stack voltage is read using a resistor divider with a total value of 103.2 kilo Ohms. This divides down the stack voltage and current from the batteries to a maximum of 3V which is tolerant for the 3.3 V analog I/O pins on the PIC32. As the first phase of our project is constant voltage, ensuring that this value scaled down and the scaled up again in software is constant is crucial. A similar divider measured the voltage across the transistor terminals to determine how “on” each was. Normally closed solid state relays turned these dividers off when charging was not in progress, ensuring batteries did not drain.

Additionally, our microcontroller reads in the temperatures of all of the batteries using LM35 (datasheet available in appendix H) temperature sensors in TO220 packages heat epoxied to the frame of the battery. When temperature of any battery goes above the specified limit of 50.0 C, the DAC closes all of the charging MOSFETs while the microcontroller continues to analyze the temperature on the batteries. As soon as the temperature goes below the limit, charging resumes.

**3.4.4 Voltage Regulators**

Our project used 5.0 V with pull down resistor for powering of the I2C line peripherals and 3.3 V for operation of the PIC32 and many other ICs on the board. We used two LM1117 regulators, one rated for 5.0 V and one for 3.3 V, in order to supply this current, fed from the transformer using a resistor and 12V zener diode.

**3.4.5 Microcontroller (Charging Circuit Controller)**

The charging controller uses nine of the analog pins on the controller as input to determine states and outputs using a 5V I2C line. These analog pins run to a temperature sensor on each battery, the current sensor, and the resistor dividers on both the battery stack and mosfets.

The charging circuit controller runs on the theory of a finite state machine. This means it enters a state, and stays in that state, until a certain set of condition or breaks force it into another state. Looking at Figure 5 below, you can easily see the transition states for the FSM.

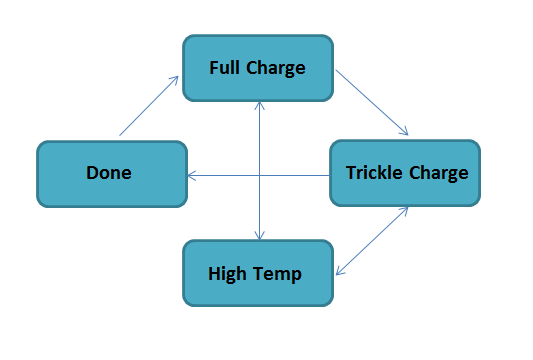


Figure 5 - State Transitions for Charging Program

When the microcontroller first gets power it will remove power from the motor. It scans the batteries and reads the voltage levels of the stack. Assuming the batteries need to be charged, it will enter the charging state. If temperatures rise too high too quickly, it will enter the “high temp” state and stay passive as the batteries cool off. It will automatically enter the charging state once a safe cut off has been reached. Once the batteries are fully charged (current drops below 1A) the charger switches to the “trickle charge” state where it provides a constant 2A and variable voltage over a period of one hour. When the hour is completed, the microcontroller enters the “done” state until it is unplugged. When unplugged, a reset occurs and the motor will become active again. Plugging in the motor controller again will push it to the monitoring stage and it will decided what state to enter based on the value obtained from measuring the battery stack.

*3.5 Detailed Operation of Subsystem 2-Peripheral*

**3.5.1 Peripheral Controller Circuit**

A custom-designed board for the peripheral control system also used a PIC32MX microcontroller. That board has the capability to read in voltage from the stack in order to display a charging percentage to the user. A 2x20 Newhaven LCD Display (datasheet available in appendix H) displays information to the user when the bike is turned on or in operation. Additionally, the peripheral controller board reads the voltages off of the charging status LEDs located on the charging circuit board in order to display the current status of the charging protocol, making the whole process much more user friendly.

**3.5.2 User Interface**

The goal of the user interface is to display real time readings of the important information for the rider. The LightBike is capable of displaying the current charge still remaining in the batteries as well as the current state of the motorcycle. It does this by taking data from the charging circuit so that the information is up to date. It will also display any warning that may exist.

The switches on this board include: a switch to power the board and turn on the LCD screen, a switch to turn on the motor controller, a switch to turn on hazard lights and a switch to turn on the plate light. There is also a button to change from displaying battery voltage to estimated distance before empty. The board also gets three inputs from the charging board in order to tell if the bike is pulled into the wall and the current charging state. Since the charging board and peripheral board are on separate grounds a opto-isolator was used. The current charging state would be displayed on the LCD as well as estimated time remaining. When the bike is plugged into the wall the board will not allow the motor controller to turn on for safety using a large solid state relay on the power line to the controller.

For operation of the tail and plate light two power mosfets were used to drive the 12V with the microcontroller inputs on the gates.

All of the information is presented on a LCD screen mounted on the handlebars so that the rider will not have to be too distracted while driving the bike. In addition to the LCD screen, there also are turn signal indicators, a warning LED, and a neutral LED. This offer the user some ability to see if there bike is in the correct state. The user interface is made to be simple so that the driver of the motorcycle can focus on the road rather than having to worry about the data being displayed.

*3.6 Detailed Operation of Subsystem 3-Motor Controller*

**3.6.1 Motor Controller**

The motor controller is a component from the former electric motorcycle groups. It is a Kelly motor controller. The part number is KD72301, has a voltage rating of 24-72 V and rated for a current of 300 A. (Datasheet link available in appendix H) It broke and could not be repaired which led to the creation of a custom made motor controller. This custom motor controller, built using scavenged parts could only handle working one battery but could not handle having the extra load of a rider on the bike. This new motor controller was built using a repurposed charging board with the mosfet array at the bottom of the motor which is hooked to power. The gate of the mosfets are driven by a resistor divider that uses the throttle’s 5kohm potentiometer. The potentiometer connects the gate of the mosfet to ground and so in the default 0ohm position the mosfets are off and no current is supplied to the motor. When at the full 5k the mosfets are fully biased and supply as much current as the motor will take. This ultimately caused the system to fail as the initial current to begin turning the motor was too great for the mosfet array to handle and caused them to explode. This system can be improved by using a mosfet H bridge with pulse width modulation which would decrease the current spike time so the mosfets would be less likely to pop.

**3.6.2 Motor**

The motor being used is that which the previous senior design groups had implemented. It has a top speed of 50 MPH. Though there was not much testing done with the motor, the previous groups had recorded that with 72 V, the motor would be able to handle 100 A at 50 MPH. It is connected through the motor controller to the batteries.

**3.6.3 Regenerative Braking**

Regenerative braking is a technology that has been developed to extend the charge on electric vehicles. The basic idea behind it is when the breaks to the vehicle are applied, the motor controller takes the kinetic energy used in slowing the vehicle down and instead uses this energy and feeds its back into the batteries, thus using the energy that would be lost in braking due to friction and heat as a way to charge the batteries without having to stop and charge using a wall outlet. On our project, the existing Kelly motor controller had the hardware already implemented for regenerative braking. It was hoped that it would extend the ride time of the bike, especially given how heavy our bike turned out to be. However, as is know, the motor controller broke with three days until presentation. This being the case, our motorcycle does not have the functionality of regenerative braking. Because of the broken motor controller, the only braking we had on our motorcycle was simply the mechanical brake that was apart of the original bike. Yet, this technology can still be implemented in future years with either purchasing a new motor controller or by designing a custom made controller that can implement regenerative braking.

## 3.7 Interfaces

Because the bike carries high current, requires quick and reliable communication between subsystems, and the subsystems are in close proximity everything on our bike has been hardwired together. The battery charging circuit and peripheral board contain embedded programs to control the logic of charging states and conversions of voltages measured by the microcontroller onboard to useful and easy to interpret information on the HUD.

# 4. System Integration Testing

## Due to the nature of our project, careful consideration was required on how to test each subsystem individually, and how to test them as an integrated whole. The majority of the required testing was done on the charging board, as this board provided the logic on how to charge the batteries at any given time, the safety factors to determine if the bike needed to be shut down, and the raw data to the peripheral board to determine what information needed to be displayed. Because of the high current needed to operate the motor, and the high voltages supplied by the battery stack, it was impractical to test the charging board with our projected full setup. Instead we tested using a high voltage supply to mimic the battery, a single battery to charge instead of the full stack, as well as a smaller motor to sink current in place of the final motor. A great deal of logic and calculations were required to get our charging board to operate according to our design specifications in each distinct charging state as dictated by Optima’s yellow top battery charging profile. This involved reprogramming the board many times while changing values until each charging state operated according to our specifications and expectations.

Our impromptu motor controller required little testing, as it operates in a very simple manner, and integrated easily into our charging board subsystem. The peripheral board was also straightforward to test, as we had already programmed our charging board to use I2C in order to easily read important values and change our coding accordingly.

Once each subsystem functioned according to our expectations, we were able to test the system as a whole. The integrated system was no different than our prototyped charging subsystem, it was only on a larger scale using the actual battery stack, capacitor bank, transformer, and motor.

Using our integrated subsystems, we successfully charged our full stack of batteries through both our full and trickle charge states. The motor controller successfully varies the speed of the motor by varying the position of the throttle on the bike handlebars. The peripheral board displays the charging state of the bike, as well as pertinent information about speed and current charge on the batteries. With these subsystems integrated and functioning according to our specifications, we demonstrated that we have met a large majority of our design requirements. The goals that were not met were mostly due to the Kelly motor controller breaking, and the decreased functionality of our last minute motor controller substitute. The temperature sensors on the batteries were also never integrated into our subsystems, but all necessary hardware and software to allow them to stop charging is completed and functional, they are simply not yet hooked up into the charging board subsystem. As we were not able to actually ride the bike, and wanted to get our charging system functioning perfectly, we decided that the temperature sensors were most important for actual riding conditions and left them off until the bike is ridable.

# 5. Users Manual/Installation Manual

This is the most important portion of the user manual. The following describes the proper procedure, step by step, to set up, use, and operate the LightBike battery charging system. It is important that these steps be followed exactly or the fuse will blow. Following these few steps will be our “user and installation manual” for a completed LightBike design and what Team LightBike would like the completely finished product to represent. Some creative licensing was taken to accurately describe what we desire for the future of LightBike.

## 5.1 Proper Charging Procedure for LightBike Prototype

1. Plug the extension cord into the wall.
2. Connect the common ground to the bare terminal on the battery. This should be the only terminal left bare. PLEASE USE HAND PROTECTION AND AN INSULATED TOOL LIKE PLIERS/WRENCHES TO CONNECT TO THIS TERMINAL. Also, be aware that SPARKS ARE TO BE EXPECTED. Do not be alarmed, this is simply the current beginning to flow through the completed circuit.
3. Take the 12V DC supply and plug it into the wall. This MUST be completed in this order or the fuse will be under risk of blowing. Once this supply is plugged in, the “ON” state LED should light up and charging will begin. It will reach a conditionally phase for one hour and shut off accordingly

When charging is complete, please follow these steps to disconnect the LightBike from the wall socket

1. Unplug the 12V DC Supply.
2. Disconnect the common ground terminal of the battery. Same precautions apply. Place connector in a safe place so it will not come loose and swing, causing a short/damaging the batteries.
3. Unplug the extension cord from the wall. Be very careful as the capacitors will take some time to discharge.

*5.2 What products does LightBike provide consumers?*

LightBike Inc. has two kits available for end use consumers. These are in the form of our best selling predesigned, complete, ready to use fully electric motorcycle as well as the second package: a do it yourself (DIY) kit. LightBike Inc. is dedicated to motorcycle enthusiasts providing around the clock customer service as well as a dedicated forums for the DIY community, which comprises a large portion of the electric vehicles market.

*5.3 Which package is best for me?*

The most stand out reason to stick with the first package is the ability to have a fully working motorcycle for their daily commute and recreational use. This vehicle “works out of the box” and is available for immediate shipment. This is a great opportunity for those new to electric vehicles and want to get into the DIY crowd, but have yet to gain experience in constructing one themselves. The end user will be able to see first hand how electric vehicles operate with our bundled pamphlet titled “How Electric Vehicles Operate”.

The DIY package is the perfect opportunity for beginner and advanced users to construct and operate an electric vehicle. This unique package contains the LightBike Battery Charger, the LightBike Handlebar Digital Braking and accelerator kit, the LightBike motor controller as well as a heads up display. Batteries can be optionally purchased in a special bundled price through our online store. If a bike frame is already owned or an electric motorcycle is just begging for an upgrade, this kit is perfect for you. The motor controller can be varied in current capacity and load capacity based on the end user needs and size/weight of their desired vehicle. The next few commonly asked questions will be answered in regards to each package. What each kit for the DIY bundle includes is explained below.

## 5.4 How to install your product

Package one customers need not worry about installation! LightBike is designed to be plugged into a standard wall outlet thanks to the onboard transformer and capacitor bank. The retractable extension cord reaches up to 10 feet in length, perfect for getting to those hard to reach sockets. Optima Yellowtop Batteries promise at least 350 full discharge/recharge cycles before they reach 90% of original capability. LightBike battery charger has been carefully optimized to match the optimum charging profiles for the yellowtop batteries.

With our charging circuit, visible underneath the plexiglass cover, a series of LEDs will notify the user of what state the bike is in. If the first LED is on, (the LED furthest away from the dashboard) the charging circuit is powered on and ready to charge your LightBike! The heads up display will notify you that it is ready for charging. Once you plug in LightBike, the processor automatically disables the motor from operating to prevent injury to the user or the bike. Once the microcontroller has completed reading the current voltage and charge state of the batteries, it will automatically begin phase one of charging and light up LED 2 alongside LED 1. When the bike enters the conditioning phase 2, LED 3 will light up and LED 2 will shut off. When LED 4 is lit up, the bike has finished charging and is ready to go. Once the motorcycle is unplugged from the wall, the heads up display will reboot, check the system and display “driving mode”.

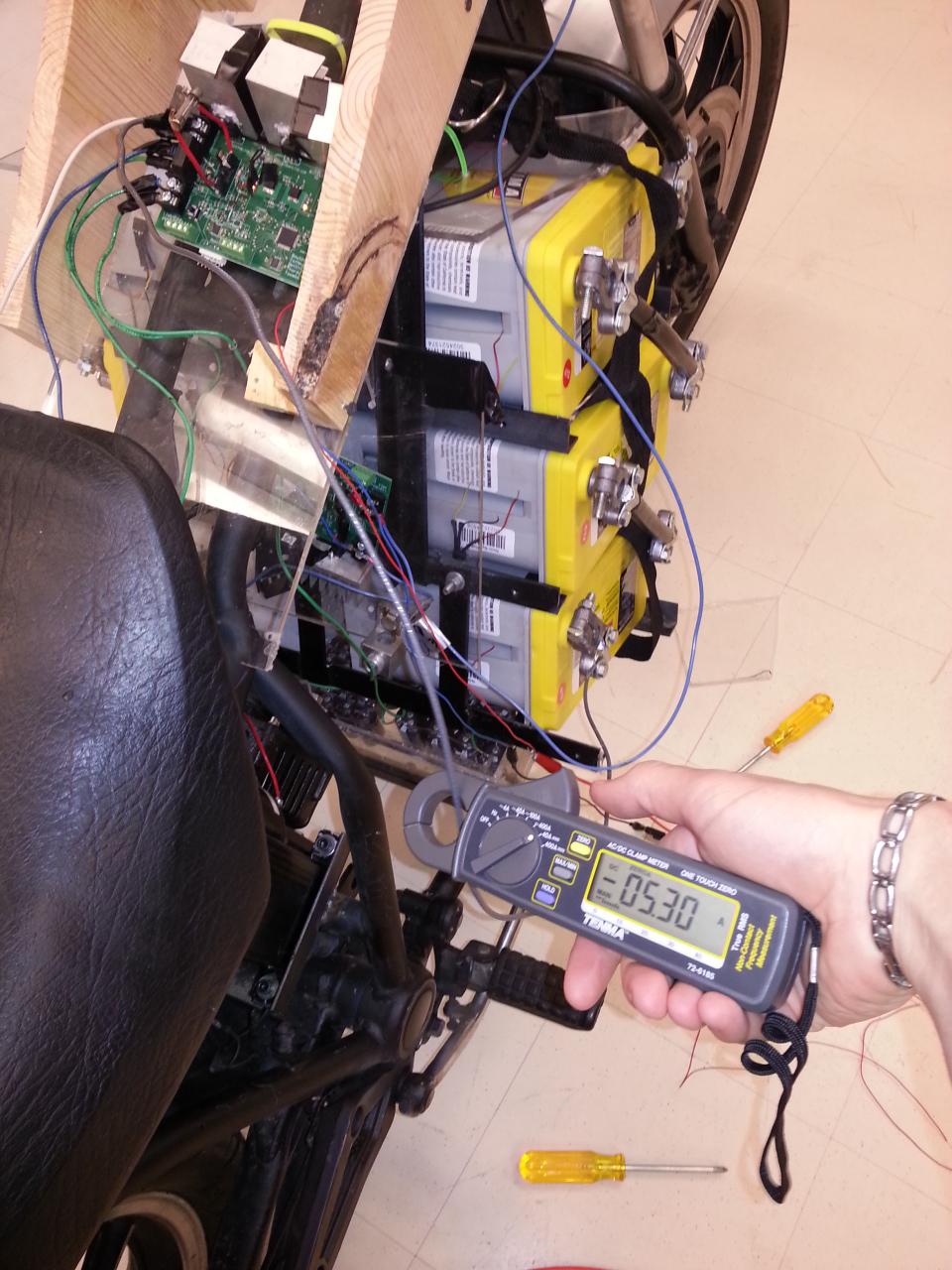


Figure 6 - Verifying that Current is Flowing

To operate LightBike, basic motorcycle skills apply. Since this is an electric motor, there will not be any gear shifting necessary. Simply use the twist throttle to accelerate and the thumb trigger to brake. If, for whatever reason, the thumb trigger should fail an emergency mechanical brake is easily accessible at the toes of your right foot. Should the motor become uncontrollable, a quick pull safety feature is implemented on the heads up display. Simply pull out the key and power will be removed from the system and the bike will gradually coast to a stop.

*LightBike is as simple as it gets for plug and ride vehicles.*

Package two customers will have, obviously, a little bit more to do and understand before attempting to ride their LightBike. First and foremost, they must acquire an empty bike frame. At LightBike we recommend that the mechanical braking as well as the mechanical system for speed/distance be intact and operational. Advanced DIY customers may also use their own batteries for the job, however this will require that the battery charging profiles be altered in the processor to ensure that it is being properly charged based on its unique requirements. However, if the optional D51 Yellowtop batteries are purchased, the kits come with pre-installed and programmed software ready to use once properly assembled.

*5.5 What is the breakdown of the DIY Bundle?*

As seen above, the DIY bundle is a collection of various kits that LightBike has fine tuned for customer satisfaction.

**Charger Kit**

1. Includes charging board, complete with fuse for protection
2. Transformer
3. Capacitor Bank
4. Retractable Extension Cord
5. (OPTIONAL) Cables to connect to batteries
6. (OPTIONAL) 6 Yellowtop Batteries

**Motor Controller Kit**

1. Motor Controller Board
2. Software to Program Motor Controller
3. Cables and Mounting Equipment

**Handlebar Kit**

1. Twist Throttle
2. Thumb Trigger Brake
3. Leads to the Motor Controller

**Heads Up Display Kit**

1. NewHaven LCD Screen
2. Various Leads and Connections (To connect to battery charger)
3. Cables and Mounting Equipment
4. (OPTIONAL) Display Box
5. Note: Contains Software to correctly use brake and turn signal lights

The DIY customer can purchase the entire bundle or each of the individual kits above. Mounting each piece and ensuring a proper and stable connection is the first step in using LightBike. Custom programming can be done using a Pickit Connection on the motor controller as well as the charging circuit.

## 5.6 How the user can troubleshoot the product

If the product is not working as described, your first priority should be to check your connections. Ensure that the capacitor bank, battery stack, and transformer output are all connected with the correct polarity. Failure to do so may have caused obvious damage to the board. Additionally, check voltages across the transformer and capacitor terminals. Finally, verify continuity along microcontroller pins and status operation LEDs.

Identifying and connecting a Newhaven I2C LCD display to the correct pins (see board in appendix) in order to verify operation of the I2C and other lines. Correct connection of this can be seen below in Figure 7. The display can show vital information from the microcontroller regarding the current, voltage, and status of the charging circuit.

If problems persist after these debugging steps, reprogramming may be necessary. Please contact the manufacturer in order to obtain details about obtaining open source code for charging circuit and a Pickit3 in order to compile code and reprogram board manually.



Figure 7 - Debugging your Motor Controller Board over I2C

# 6. To-Market Design Changes

The LightBike has gone through three senior design groups and still has much to be done before it would be market ready. That being said, the LightBike is very far along in the design process and would be ready for market relatively soon.

It is already very user friendly in terms of the charging process with its plug in anywhere design. Also, the heads up display shows relevant data with regards to the charging of the batteries. The simplicity of the motorcycle is an important component for the potential marketing of the LightBike.

That being said, there still are some things that need to be addressed before the bike could be taken to market. One fix that is of the utmost importance is to add a power diode between the capacitor bank and the charging circuit to stop the large current spike upon final battery connection. This will solve our previous problem of blowing the fuse on our charging board, but there are a variety of other improvements required before the product is market ready. One is the motor controller. Redesigning the motor controller so that it will provide regenerative braking and a more failsafe construction. If the bike’s motor controller broke for the customer, it would reflect poorly on the product, no matter how well the batteries charge. Regenerative braking is should be a necessary piece of technology on any electric vehicle and the LightBike is no exception. Having a motor controller that can handle the above would be necessary before the LightBike could be sold to consumers.

Also, with regards to the rest of the bike, the aesthetics and user interface needs to be improved in order for the LightBike to be a viable consumer product. The HUD needs to modified in order to have more features that would appeal to a larger number of consumers. It can also be modified for the LED to display more information and have more features. Also, making the bike street legal is also an important aspect that would obviously need to be completed before selling the motorcycle. Aesthetically, the motorcycle is lacking. With the addition of a body shell and making the bike more streamlined, it would appeal to a larger number of consumers who are looking for more than just an electric motorcycle.

Finally, the safety measures would need to be tested further and other measures would need to be added. Since safety is paramount for consumers, having an extensive number of safety measures would be necessary. Already, the bike has temperature sensors. These would need to be tested further during actual use of the bike to find an appropriate range of operating temperatures. Additionally, further safety measures would need to be added. A more secure shell would need to be added to protect the user from touching the batteries while operating or charging. The best thing that can be done in terms of safety for the operator is to test the motorcycle extensively and make sure that the bike operates according to expectations and that if things do fail,there are measures that can protect the user. Safety measures are the most important aspect that would need to be addressed before the motorcycle moved on to the consumer.

While the bike’s core functionality is there, there are several things that would need to be improved, modified, and tested before the bike could be sold to customers.

**7. Recommendations for Project Continuation**

With the unfortunate blow-out of the Kelly motor controller, the most obvious continuation of the project would be to get the bike to run as expected. For this to happen, a new motor controller must either be purchased or designed. On the current LightBike, the custom built motor controller can only handle the current of a few batteries and if a person tries to ride it, the increased weight causes the mosfets on the controller to burn out due to the high current necessary to move the motor.

Also, because the full 6 battery charger has been completed successfully this year, for continuation, future groups could be charged with designing and building a motor controller. While the basic circuit design is not that complex, in order to be able to handle the high currents and voltages that are needed to power the motor, there would have to be some serious design considerations. Additionally, having future teams implement regenerative braking would be a lofty challenge but prove to enhance the motorcycle overall.

Finally, given the hectic design considerations with the motor controller breaking with so few days left in the build, many of the peripherals were not completed fully. These include the heads up display along with the controls of the bike, such as turn signals and warning lights. These were completed with the current build of the LightBike but further improvement and additions could be included to improve the riding experience. Additionally, finally making the bike street legal would be an obvious next step for the designers. Much of the required parts exist on the motorcycle currently, but the addition of a horn, headlight, and turn signals for example would be easy add ons. Also, improving the aesthetics of the bike remain a secondary but important component if the LightBike would be sold to consumers.

# 8. Conclusions

Overall, the project was a success. The largest task was completing the battery charger. This involved extensive coding and testing of the circuit board with and without the battery. Because of the very high voltages that were being used, any errors in the code could lead to parts on the board being physically broken. This lead to major headaches when it came to debugging the code. Once periodic check on the pieces of the board were implemented, this issue was resolved. The code was successful at charging both a single battery and all six batteries in series. This was the ultimate goal of our project and the group succeeded in this task.

Despite the charging circuitry working, the bike’s motor controller broke with only days before the due date. This was a major setback in terms of actually making a motorcycle that could drive. However, we were still able to design a basic controller which would allow for the wheel to spin. The basic motor controller that was designed could not support the load of a rider before the mosfets would blow out, but having a motorcycle that could at least rotate the wheel was something that showed going beyond the initial requirements.

Finally, the peripherals on the bike such as the display which showed charge and the state the bike was in all worked successfully. While the packaging for the peripherals was not ideal and some objectives such as making the LightBike street legal were not bet, this was a secondary objective of the product. The charging of the six batteries was the main goal of the LightBike and that has been proven to work successfully.

# 9. Appendices

*Appendix A: Charging Circuit Code*

/\*

\* File: main.c

\* Author: Mike Mellitt

\* Ben Coffey

\* Jake Thordahl

\* Pat Bowlds

\* Alex Toombs

\*

\* Main file for program to read voltage from current sensor and adjust current

\* sourced to battery accordingly, using I2C to communicate with MCP

\* 4706DAC.

\* Notre Dame Senior Design 2013

\* Lightbike Group

\*

\* LED Status Codes:

\* (excludes LED on RE0, status to peripheral board)

\* (1 is high) RE3 RE1 RE2 RE0

\* State Top LED Bottom LED

\* 10 Amp charge 0 0 0 1

\* High Temp 0 0 1 1

\* Trickle charge 0 1 0 1

\* Charge done 1 0 0 1

\*

\* Analog pinout on board:

\* A12: Current Sensor

\* A7: Battery Stack Voltage (through divider)

\* A6: Power MOSFET Voltage

\* A2,A3,A4,A5,A8,A9: Temperature Sensors

\*

\* Created on November 26, 2012, 8:26 PM

\*

\* Last Modified: May 3, 2013

\*/

#include <stdio.h>

#include <stdlib.h>

#include <sys/attribs.h>

#include "configbits.h"

#include <xc.h>

#include <plib.h>

#include <math.h>

#include "UART.h"

#include "Other.h"

#define A 0x41

#define B 0x42

#define C 0x43

#define D 0x44

#define E 0x45

#define F 0x46

#define G 0x47

#define H 0x48

#define I 0x49

#define J 0x4A

#define K 0x4B

#define L 0x4C

#define M 0x4D

#define N 0x4E

#define O 0x4F

#define P 0x50

#define Q 0x51

#define R 0x52

#define S 0x53

#define T 0x54

#define U 0x55

#define V 0x56

#define W 0x57

#define X 0x58

#define Y 0x59

#define Z 0x5A

#define LED 0x50

#define LEDREG 0xFE

#define DAC 0xC0

#define DACOUT 0x00

#define LEDCLR 0x51

#define LEDRIGHT 0x4A

#define ZERO 0x30

#define ONE 0x31

#define TWO 0x32

#define THREE 0x33

#define FOUR 0x34

#define FIVE 0x35

#define SIX 0x36

#define SEVEN 0x37

#define EIGHT 0x38

#define NINE 0x39

#define COLON 0x3A

#define EQUAL 0x3D

#define DECIMAL 0x2E

#define true 1

#define false 0

// boolean to set debug (UART out or not)

int debug = false;

// boolean to set temp control

int tempControlOn = false;

// voltage reading from current sensor; global variable

double Csensor\_volt=0;

double Battery\_volt=0;

// current through stack

double current = 0;

// real stack voltage converted up

double realStack = 0;

// MOSFET voltage

double mosVolt = 0;

// real-world mosfet voltage

double realMos = 0;

// should hopefully be real battery voltage

double realBatt = 0;

// max current allowed into battery, Amps. Critical limit to stay under

double maxCurrent = 8.0;

// max voltage across stack, Volts, if 15 V per battery limit

double maxVoltage = 90;

// offset to adjust the DAC

// "shift" is the value written to the DAC, which is off the gate of the

// array of power MOSFETs. Lowering "shift" decreases the current allowed

// through the MOSFET and thus the battery stack that the FET is in

// series with.

int shift = 0;

// read-in voltages from temperature sensors, in Volts

double vt1 = 0;

double vt2 = 0;

double vt3 = 0;

double vt4 = 0;

double vt5 = 0;

double vt6 = 0;

// converted temp sensor readings, in degrees C

double temp1;

double temp2;

double temp3;

double temp4;

double temp5;

double temp6;

// battery temp limit, in degrees C

static double tempLimit = 50.0;

// int timer to control 1 hour @ 2 Amp charging routine

int trickleTimer = 0;

// timer to control delay at beginning of charge before entering full charge state

int delayTimer = 0;

// boolean to determine if full charge (to current < 1.0 A) is done

int isHighCurrentDone = false;

// boolean to determine if charge is totally done

int isTrickleDone = false;

// boolean to determine if batteries are too hot

int isTooHot = false;

// Loops continuously to adjust current source output

int main(int argc, char\*\* argv) {

// Uart Config 57600 (serial baud rate)

UartConfig();

// Timer Interrupts

ConfigTime();

// Analog Config pin B7

ConfigAnalog();

// I2C Config

ConfigI2C();

TRISEbits.TRISE3=0;

LATEbits.LATE3=0;

TRISEbits.TRISE2=0;

LATEbits.LATE2=0;

TRISEbits.TRISE1=0;

LATEbits.LATE1=0;

TRISEbits.TRISE0=0;

LATEbits.LATE0=1;

// run until charge routine is done

while(1)

{

}

return (EXIT\_SUCCESS);

}

// Interrupt fires every half second

void \_\_ISR(8, IPL3AUTO) Timer2Hand(void)

{

INTClearFlag(INT\_T2);

// Display shift value on LCD

SendI2C3(LED,LEDREG,LEDCLR);

SendI2C2(LED,S);

SendI2C2(LED,H);

SendI2C2(LED,I);

SendI2C2(LED,F);

SendI2C2(LED,T);

SendI2C2(LED,COLON);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C2(LED,ParseFirstShift());

SendI2C2(LED,ParseSecondShift());

SendI2C2(LED,ParseThirdShift());

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

// Display current sensor reading on line two

SendI2C2(LED,C);

SendI2C2(LED,U);

SendI2C2(LED,R);

SendI2C2(LED,R);

SendI2C2(LED,COLON);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C2(LED,ParseTens(current));

SendI2C2(LED,ParseFirst(current));

SendI2C2(LED,DECIMAL);

SendI2C2(LED,ParseSecond(current));

SendI2C2(LED,ParseThird(current));

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C2(LED,A);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

// Display current sensor reading on line two

SendI2C2(LED,B);

SendI2C2(LED,R);

SendI2C2(LED,E);

SendI2C2(LED,A);

SendI2C2(LED,L);

SendI2C2(LED,COLON);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C2(LED,ParseTens(realBatt));

SendI2C2(LED,ParseFirst(realBatt));

SendI2C2(LED,DECIMAL);

SendI2C2(LED,ParseSecond(realBatt));

SendI2C2(LED,ParseThird(realBatt));

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C2(LED,V);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

// Display battery stack reading on line three

SendI2C2(LED,R);

SendI2C2(LED,M);

SendI2C2(LED,O);

SendI2C2(LED,S);

SendI2C2(LED,COLON);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C2(LED,ParseFirst(realMos));

SendI2C2(LED,DECIMAL);

SendI2C2(LED,ParseSecond(realMos));

SendI2C2(LED,ParseThird(realMos));

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C2(LED,V);

// populate sensor readings, update values

getAnalog();

// update temperature sensor values from battery

updateTemps();

updateValues();

// delay timer to control delay period

delayTimer = delayTimer + 1;

if(debug==true) {

// show values (estimated)

printf("\nDivided Stack Voltage Reading: %5.2f V\n", Battery\_volt);

printf("Corresponds to real voltage: %5.2f V\n", realStack);

printf("Estimated current is: %5.2f A\n", current);

printf("MOSFET voltage is: %5.2f V\n", mosVolt);

}

// All charging logic follows below

// Shuts down charger entirely if temperatures get too hot.

// Switches between isTooHot false/true states

if(tempControlOn == true) {

printf("\nTemperature control is on...\n");

if(temp1 >= tempLimit || temp2 >= tempLimit || temp3 >= tempLimit ||

temp4 >= tempLimit || temp5 >= tempLimit || temp6 >= tempLimit) {

shift = 0;

isTooHot = true;

writeToDAC();

// RE3 on as warning light for high temp

LATEbits.LATE3=1;

LATEbits.LATE2=0;

LATEbits.LATE1=0;

LATEbits.LATE0=1;

}

// goes back into charging state if temp limit goes low enough

else if(temp1 >= tempLimit - 10 || temp2 >= tempLimit - 10 || temp3 >= tempLimit - 10||

temp4 >= tempLimit - 10 || temp5 >= tempLimit - 10 || temp6 >= tempLimit - 10) {

// resets delay timer to avoid early transition to trickle state

delayTimer = 0;

isTooHot = false;

}

}

else {

if(debug == true)

printf("\nWARNING: Temperature control is OFF!\n");

}

// when stack voltage is low but current isn't low, keep doing full charge

if(!isHighCurrentDone && !isTooHot && !isTrickleDone) {

fullCharge();

if(debug == true)

printf("\nDoing full charge routine");

LATEbits.LATE3=0;

LATEbits.LATE2=0;

LATEbits.LATE1=0;

LATEbits.LATE0=1;

}

// when stack is high AND current is low, it's time to trickle charge

else if(isHighCurrentDone && delayTimer >= 62 && !isTooHot && !isTrickleDone) {

trickleTimer = trickleTimer + 1;

trickleCharge();

if(debug == true)

printf("\nTrickle charging");

LATEbits.LATE3=0;

LATEbits.LATE2=0;

LATEbits.LATE1=1;

LATEbits.LATE0=1;

}

// if trickleCharge runs for 1 hour (currently ~7200 ISRs), charge is done.

// adjust trickle charge state done

if(trickleTimer >= 3605 && isTrickleDone) {

shift = 0;

writeToDAC();

isTrickleDone = true;

LATEbits.LATE3=1;

LATEbits.LATE2=0;

LATEbits.LATE1=0;

LATEbits.LATE0=1;

if(debug == true) {

printf("\nCharging routine is done!\n");

}

}

// when current drops below 1.0 amp, switch to conditioning trickle state

if(current < 1.0 && delayTimer >= 62) {

isHighCurrentDone = true;

}

}

// Configure bits for Timer operation. Timer fires every half-second.

void ConfigTime()

{

// Stops Timer and Clears register

T2CON = 0x0;

TMR2 = 0xFD8E;

// Set PR to 65535 originally 16000 with 3E80

PR2 = 0x0FA0;

// Set prescaler at 1:256

T2CONSET = 0x0070;

// Start Timer

T2CONSET = 0x8000;

INTConfigureSystem(INT\_SYSTEM\_CONFIG\_MULT\_VECTOR);

// Enables Global Interrupts

INTEnableInterrupts();

// Enables Timer2 Interrupts

INTEnable(INT\_T2, INT\_ENABLED);

// Clears timer2 flag

INTClearFlag(INT\_T2);

// Timer2 has priority 3

INTSetVectorPriority(INT\_T2,3);

}

// Configure analog registers to read value from sensor

void ConfigAnalog()

{

// enable analog pins 8, 9, 12

DDPCONbits.JTAGEN = 0;

IFS1CLR = 2; //clear ADC conversion interrupt

IEC1SET = 2; //enable ADC interrupt

AD1PCFG = 0x0000; //Configure relevant pins to analog

AD1CON1 = 0b00000000000000001000000011100110; //Configure Sample clock source

AD1CON2 = 0b0000010000100000; //Configure ADC voltage reference

AD1CON3 = 0x0000; //Configure ADC conversion clock

AD1CON3bits.SAMC = 0b00001; //auto sample at 2TAD

AD1CON3bits.ADCS = 0b00000001; //TAD = 4TPB

AD1CHS = 0x00000000; //Configure input channels- CH0+ input,

AD1CON2bits.CSCNA=1;

AD1CSSL = 0b0001001111111100;

AD1CON1SET = 0x8000; //Turn on the ADC module

}

// Get all analog values

void getAnalog() {

while( ! IFS1bits.AD1IF); //wait until buffers contain new samples

AD1CON1bits.ASAM = 0; //stop automatic sampling (shut down ADC basically)

vt1 = ADC1BUF0\*.003185;

vt2 = ADC1BUF1\*.003185;

vt3 = ADC1BUF2\*.003185;

vt4 = ADC1BUF3\*.003185;

mosVolt = ADC1BUF4\*.003185;

Battery\_volt = ADC1BUF5\*.003185;

vt5 = ADC1BUF6\*.003185;

vt6 = ADC1BUF7\*.003185;

Csensor\_volt = ADC1BUF8\*.003185;

IFS1bits.AD1IF = 0;

AD1CON1bits.ASAM = 1; //restart ADC and sampling

}

// Configure I2C registers

void ConfigI2C()

{

//1 USES RD10 AS SCL1 AND RD 9 AS SDA1

/// I2CxCON I2CxSTAT I2CxADD I2CxMSK I2CxTRN I2CxRCV

I2C1BRG=0x030; //390 for 80MHz to 100KHz

I2C1CONbits.A10M=0; //Use 7-bit addresses

I2C1CONbits.DISSLW=1; //disable slew control for standard

I2C1CONbits.ACKDT=0; //Use and ACK not NACK

I2C1ADD=22; //Sets slave address for PIC32

TRISD=1; //Sets Port D to input

I2C1CONbits.ON=1; //turn on I2C

}

// Start I2C

void I2C\_start(void)

{

I2C1CONbits.SEN=1; //send start

while(I2C1CONbits.SEN){} //waits till start bit detected

}

// Restart I2C

void I2C\_restart(void)

{

I2C1CONbits.RSEN=1; //send restart

while(I2C1CONbits.RSEN){} //waits till start bit detected

}

// Stop I2C

void I2C\_stop(void)

{

I2C1CONbits.PEN=1; //send stop

while(I2C1CONbits.PEN){} //waits till stop bit detected

}

// Write char of data to I2C line

char I2C\_write(char data)

{

I2C1TRN=data; //sends data to transmit register

while(I2C1STATbits.TRSTAT==1){} //waits to finsh transmission

return(I2C1STATbits.ACKSTAT); //returns 0 for ack received

}

// Write int of data to I2C line

int I2C\_writeDAC(int data)

{

I2C1TRN=data; //sends data to transmit register

while(I2C1STATbits.TRSTAT==1){} //waits to finsh transmission

return(I2C1STATbits.ACKSTAT); //returns 0 for ack received

}

// Check for acknowledgement

void mAckI2C1(void)

{

I2C1CONbits.ACKDT=0;

I2C1CONbits.ACKEN=1;

while(I2C1CONbits.ACKEN){}

}

// Check for lack of acknowledgement

void mNAckI2C1(void)

{

I2C1CONbits.ACKDT=1;

I2C1CONbits.ACKEN=1;

while(I2C1CONbits.ACKEN){}

}

// Read data back from I2C line

char I2C\_read(char ack)

{

I2C1CONbits.RCEN=1;

while(I2C1CONbits.RCEN){}

//Reception is started, send ack/nack after read

if(ack==0)

{mNAckI2C1();}

else

{mAckI2C1();}

//Reception should be complete - pull out data

return(I2C1RCV);

}

// Make I2C line wait for registers to clear

void I2C\_idle()

{

while((I2C1CON&0x001F)!=0){}

// Wait for Acken, Rcen, Pen, Rsen and Sen to clear

}

// Send data to I2C line at given address

void SendI2C3(char addrs,char regis, char data)

{

char ack;

I2C\_start();

ack=I2C\_write(addrs); //Address for LED is 0x50

ack=I2C\_write(regis); //0xFE for LED

ack=I2C\_write(data); //0x20to0x7F standard

I2C\_stop();

}

// General I2C call for DAC (just checks for ACK)

void SendI2CGen(char regis) {

char ack;

I2C\_start();

ack=I2C\_write(regis);

I2C\_stop();

}

// Writes to standard registers

void SendI2C2(char addrs, char data)

{

char ack;

I2C\_start();

ack=I2C\_write(addrs); //Address for LED is 0x50

ack=I2C\_write(data); //0x20to0x7F standard

I2C\_stop();

}

// Charge full stack of batteries from zero until they can be used.

// Constant voltage phase

// Operating conditions: 82.8V to 90V whole stack, current < 10A (fuse)

// For full charge, voltage must be controlled at ~88V

void fullCharge()

{

if(current >= maxCurrent - 1.5) {

// constrict MOSFET output by 1 if current gets too high

shift = shift - 1;

writeToDAC();

}

if(realStack < (maxVoltage)) {

// increment DAC out value by 1 if stack voltage is too low

shift = shift + 1;

// Output to UART in text

if(debug == true) {

printf("\nShifting up...");

}

writeToDAC();

}

else if(realStack >= (maxVoltage)) {

// decrease DAC out value by 1 if stack voltage is too high

shift = shift - 1;

// Output to UART in text

if(debug == true) {

printf("\nShifting down...");

}

writeToDAC();

}

}

// Last hour of charge should control current @ 2A

// Constant current phase

void trickleCharge() {

double trickleCurrent = maxCurrent / 2.0;

// increase DAC out value by 1 to push it closer to required current

if(current < trickleCurrent) {

shift = shift + 1;

writeToDAC();

}

// decrease DAC out value by 1 if current gets too high

else if(current > trickleCurrent) {

shift = shift - 1;

writeToDAC();

}

}

// Constrict value of shift to 8 bits, 0 through 255

void shiftSafety() {

if(shift > 255)

shift = 255;

else if(shift < 0)

shift = 0;

}

// update variables that store current/voltage

void updateValues() {

// linear fit conversion from voltage to current

// experientially derived

double fakeCVolt = Csensor\_volt \* 2.50 / 1.30 - 2.50;

current = 15.132 \* fakeCVolt - 0.0537;

// conversion of mosVolt to read

realMos = mosVolt / .18;

// gives real stack voltage

realBatt = Battery\_volt / 0.03101 - realMos;

}

// updates temperatures of battery stack

void updateTemps() {

temp1 = vt1 \* 100.0;

temp2 = vt2 \* 100.0;

temp3 = vt3 \* 100.0;

temp4 = vt4 \* 100.0;

temp5 = vt5 \* 100.0;

temp6 = vt6 \* 100.0;

// print temps to UART

if(debug == true) {

printf("\nTemp1: %3.2f C\n", temp1);

printf("Temp2: %3.2f C\n", temp2);

printf("Temp3: %3.2f C\n", temp3);

printf("Temp4: %3.2f C\n", temp4);

printf("Temp5: %3.2f C\n", temp5);

printf("Temp6: %3.2f C\n", temp6);

}

}

// Write value of shift to DAC output register, controlling shift value as 8 bits

void writeToDAC() {

shiftSafety();

if(debug == true) {

printf("Writing shift value of: %3d\n", shift);

printf("Calc. Current is: %3.2f A\n", current);

printf("Estimated DAC Output voltage is: %5.3f V\n", 5.00/255.0 \* shift);

printf("Current Sensor Reading is: %5.2f V\n", Csensor\_volt);

printf("Stack Voltage Reading is: %5.2f V\n", Battery\_volt);

}

// write value to DAC Vout register

SendI2C3(DAC,DACOUT,shift);

}

// parse tens place of input double for printing to LCD

char ParseTens(double in) {

if(fmod(in/10,10.0)<1)

return ZERO;

else if(fmod(in/10,10.0)<2)

return ONE;

else if(fmod(in/10,10.0)<3)

return TWO;

else if(fmod(in/10,10.0)<4)

return THREE;

else if(fmod(in/10,10.0)<5)

return FOUR;

else if(fmod(in/10,10.0)<6)

return FIVE;

else if(fmod(in/10,10.0)<7)

return SIX;

else if(fmod(in/10,10.0)<8)

return SEVEN;

else if(fmod(in/10,10.0)<9)

return EIGHT;

else if(fmod(in/10,10.0)<10)

return NINE;

}

// Parse first digit (ones place) of input double for printing to LCD

char ParseFirst(double in)

{

if(fmod(in,10.0)<1)

return ZERO;

else if(fmod(in,10.0)<2)

return ONE;

else if(fmod(in,10.0)<3)

return TWO;

else if(fmod(in,10.0)<4)

return THREE;

else if(fmod(in,10.0)<5)

return FOUR;

else if(fmod(in,10.0)<6)

return FIVE;

else if(fmod(in,10.0)<7)

return SIX;

else if(fmod(in,10.0)<8)

return SEVEN;

else if(fmod(in,10.0)<9)

return EIGHT;

else if(fmod(in,10.0)<10)

return NINE;

}

// Parse second digit (tenth place) of input double for printing to LCD

char ParseSecond(double in)

{

if(fmod(in\*10,10.0)<1)

return ZERO;

else if(fmod(in\*10,10.0)<2)

return ONE;

else if(fmod(in\*10,10.0)<3)

return TWO;

else if(fmod(in\*10,10.0)<4)

return THREE;

else if(fmod(in\*10,10.0)<5)

return FOUR;

else if(fmod(in\*10,10.0)<6)

return FIVE;

else if(fmod(in\*10,10.0)<7)

return SIX;

else if(fmod(in\*10,10.0)<8)

return SEVEN;

else if(fmod(in\*10,10.0)<9)

return EIGHT;

else if(fmod(in\*10,10.0)<10)

return NINE;

}

// Parse third digit (hundredths place) of input double for LCD outputting

char ParseThird(double in)

{

if(fmod(in\*100,10.0)<1)

return ZERO;

else if(fmod(in\*100,10.0)<2)

return ONE;

else if(fmod(in\*100,10.0)<3)

return TWO;

else if(fmod(in\*100,10.0)<4)

return THREE;

else if(fmod(in\*100,10.0)<5)

return FOUR;

else if(fmod(in\*100,10.0)<6)

return FIVE;

else if(fmod(in\*100,10.0)<7)

return SIX;

else if(fmod(in\*100,10.0)<8)

return SEVEN;

else if(fmod(in\*100,10.0)<9)

return EIGHT;

else if(fmod(in\*100,10.0)<10)

return NINE;

}

// Parse the hundreds place of the integer shift for display

char ParseFirstShift() {

if(fmod(shift/100,10)<1)

return ZERO;

else if(fmod(shift/100,10)<2)

return ONE;

else if(fmod(shift/100,10)<3)

return TWO;

else if(fmod(shift/100,10)<4)

return THREE;

else if(fmod(shift/100,10)<5)

return FOUR;

else if(fmod(shift/100,10)<6)

return FIVE;

else if(fmod(shift/100,10)<7)

return SIX;

else if(fmod(shift/100,10)<8)

return SEVEN;

else if(fmod(shift/100,10)<9)

return EIGHT;

else

return NINE;

}

// Parse the tens place of the integer shift for display

char ParseSecondShift() {

if(fmod(shift/10,10)<1)

return ZERO;

else if(fmod(shift/10,10)<2)

return ONE;

else if(fmod(shift/10,10)<3)

return TWO;

else if(fmod(shift/10,10)<4)

return THREE;

else if(fmod(shift/10,10)<5)

return FOUR;

else if(fmod(shift/10,10)<6)

return FIVE;

else if(fmod(shift/10,10)<7)

return SIX;

else if(fmod(shift/10,10)<8)

return SEVEN;

else if(fmod(shift/10,10)<9)

return EIGHT;

else

return NINE;

}

// Parse the ones place of the integer shift for display

char ParseThirdShift() {

if(fmod(shift,10)<1)

return ZERO;

else if(fmod(shift,10)<2)

return ONE;

else if(fmod(shift,10)<3)

return TWO;

else if(fmod(shift,10)<4)

return THREE;

else if(fmod(shift,10)<5)

return FOUR;

else if(fmod(shift,10)<6)

return FIVE;

else if(fmod(shift,10)<7)

return SIX;

else if(fmod(shift,10)<8)

return SEVEN;

else if(fmod(shift,10)<9)

return EIGHT;

else

return NINE;

}

/\*

\* File: configbits.h

\* Author: Mike

\*

\* Created on October 9, 2012, 1:50 PM

\*/

#ifndef CONFIGBITS\_H

#define CONFIGBITS\_H

/\* 20 MHz crystal run at 80 mhz

peripher clock = at 10 MHz (80 MHz/8)

\*/

#pragma config FNOSC = PRIPLL // Oscillator selection

#pragma config POSCMOD = HS // Primary oscillator mode

#pragma config FPLLIDIV = DIV\_5 // PLL input divider (20 -> 4)

#pragma config FPLLMUL = MUL\_20 // PLL multiplier ( 4x20 = 80)

#pragma config FPLLODIV = DIV\_1 // PLL output divider

#pragma config FPBDIV = DIV\_8 // Peripheral bus clock divider 10 mhz

#pragma config FSOSCEN = OFF // Secondary oscillator enable

/\* Clock control settings

\*/

#pragma config IESO = ON // Internal/external clock switchover

#pragma config FCKSM = CSDCMD // Clock switching (CSx)/Clock monitor (CMx)

#pragma config OSCIOFNC = OFF // Clock output on OSCO pin enable

/\* Other Peripheral Device settings

\*/

#pragma config FWDTEN = OFF // Watchdog timer enable

#pragma config WDTPS = PS1024 // Watchdog timer post-scaler

#pragma config FSRSSEL = PRIORITY\_7 // SRS interrupt priority

#pragma config ICESEL = ICS\_PGx1 // ICE pin selection

#endif /\* CONFIGBITS\_H \*/

/\*

\* File: Other.h

\* Author: mmellitt

\*

\* Configures I2C on board and calls other configure functions

\*

\* Created on February 18, 2013, 3:04 PM

\*/

#ifndef OTHER\_H

#define OTHER\_H

#ifdef \_\_cplusplus

extern "C" {

#endif

void ConfigTime();

void ConfigAnalog();

void getAnalog();

void ConfigI2C();

void I2C\_start(void);

void I2C\_restart(void);

void I2C\_stop(void);

char I2C\_write(char data);

int I2C\_writeDAC(int data);

void mAckI2C1(void);

void mNAckI2C1(void);

char I2C\_read(char ack);

void I2C\_idle();

void SendI2C3(char addrs, char regis, char data);

void SendI2C2(char addrs, char data);

void SendI2CGen(char regis);

void fullCharge();

void trickleCharge();

void shiftSafety();

void updateValues();

void updateTemps();

void writeToDAC();

char ParseTens(double in);

char ParseFirst(double in);

char ParseSecond(double in);

char ParseThird(double in);

char ParseFirstShift();

char ParseSecondShift();

char ParseThirdShift();

#ifdef \_\_cplusplus

}

#endif

#endif /\* OTHER\_H \*/

#include <xc.h>

#include "UART.h"

void UartConfig()

{

U6MODEbits.ON=1;

U6STAbits.UTXEN=1;

U6STAbits.URXEN=1;

U6STAbits.ADM\_EN=1;

U6MODEbits.PDSEL=0x00;

U6MODEbits.STSEL=0;

U6MODEbits.BRGH=1;

U6BRG=42;

}

void UARTrepeat()

{

send(get());

}

char get(void)

{

if(U6STAbits.URXDA)

{

return(U6RXREG);

}

}

void \_mon\_putc(char a)

{

send(a);

}

void send(char a)

{

while(U6STAbits.UTXBF){}

if(!U6STAbits.UTXBF)

{

U6TXREG = a;

}

}

/\*

\* File: UART.h

\* Author: mmellitt

\*

\* Created on February 11, 2013, 7:10 PM

\*/

#ifndef UART\_H

#define UART\_H

#ifdef \_\_cplusplus

extern "C" {

#endif

void UartConfig();

void UARTrepeat();

char get();

void \_mon\_putc(char a);

void send(char a);

#ifdef \_\_cplusplus

}

#endif

#endif /\* UART\_H \*/

Detailed charging circuit source code available on [Github](https://github.com/alextoombs/Lightbike/)

*Appendix B: Peripheral Board Code*

/\*

\* File: main.c

\* Author: Mike Mellitt

\* Ben Coffey

\* Jake Thordahl

\* Pat Bowlds

\* Alex Toombs

\*

\* Main file for program to read voltage from battery resistor divider to

\* monitor charge, MPH, other peripherals

\*

\* Analog pinout on board:

\* A7: Battery Stack Voltage (through divider)

\* A6: Break sensors

\*

\* Created on April 23, 2013, 4:35 PM

\*

\* Last Modified: May 3, 2013

\*/

#include <stdio.h>

#include <stdlib.h>

#include <sys/attribs.h>

#include "configbits.h"

#include <xc.h>

#include <plib.h>

#include <math.h>

#include "Other.h"

#define A 0x41

#define B 0x42

#define C 0x43

#define D 0x44

#define E 0x45

#define F 0x46

#define G 0x47

#define H 0x48

#define I 0x49

#define J 0x4A

#define K 0x4B

#define L 0x4C

#define M 0x4D

#define N 0x4E

#define O 0x4F

#define P 0x50

#define Q 0x51

#define R 0x52

#define S 0x53

#define T 0x54

#define U 0x55

#define V 0x56

#define W 0x57

#define X 0x58

#define Y 0x59

#define Z 0x5A

#define LED 0x50

#define LEDREG 0xFE

#define DAC 0xC0

#define LEDCLR 0x51

#define LEDRIGHT 0x4A

#define ZERO 0x30

#define ONE 0x31

#define TWO 0x32

#define THREE 0x33

#define FOUR 0x34

#define FIVE 0x35

#define SIX 0x36

#define SEVEN 0x37

#define EIGHT 0x38

#define NINE 0x39

#define COLON 0x3A

#define EQUAL 0x3D

#define DECIMAL 0x2E

#define CON 0x52

#define true 1

#define false 0

// voltage reading from battery sensor analog pin

double Battery\_volt=0;

// voltage reading from break sensor through divider

double Break\_volt=0;

int ChargeOn,Phase2,ChargeDone;

int On2,Haz,Night;

int Right,Left;

int But1,But2;

int breaking=0;

int flash=1,error=0;

int startcount=0;

// Loops continuously to adjust current source output

int main(int argc, char\*\* argv) {

// Timer Interrupts

ConfigTime();

// Analog Config pin B7

ConfigAnalog();

// I2C Config

ConfigI2C();

// disable JTAG so pin 12-15 can be used

DDPCONbits.JTAGEN = 0;

AD1PCFGbits.PCFG12 = 1;

AD1PCFGbits.PCFG13 = 1;

AD1PCFGbits.PCFG14 = 1;

AD1PCFGbits.PCFG15 = 1;

// Configure input ports------------------------------------------------

//Status

TRISEbits.TRISE0 = 1; //charger connected //Logic Done

TRISEbits.TRISE1 = 1; //phase2

TRISEbits.TRISE2 = 1; //charger done

//Switches

TRISBbits.TRISB12 = 1; //night

TRISBbits.TRISB13 = 1; //hazards

TRISBbits.TRISB14 = 1; //on2

TRISDbits.TRISD5 = 1; //right

TRISDbits.TRISD4 = 1; //left

//Buttons

TRISBbits.TRISB10 = 1; //but2

TRISBbits.TRISB11 = 1; //but1

// Configure output ports-----------------------------------------------

TRISEbits.TRISE3 = 0; //voltage div on

LATEbits.LATE3 = 1; //Turn on voltage divider

TRISDbits.TRISD1 = 0; //Motor Controller on

LATDbits.LATD1= 0; //Motor START off

TRISBbits.TRISB5 = 0; //LeftLED

LATBbits.LATB5 = 0; //Start off

TRISBbits.TRISB4 = 0; //RightLED

LATBbits.LATB4 = 0; //Start off

TRISBbits.TRISB3 = 0; //WarnLED

LATBbits.LATB3 = 0; //Start off

TRISBbits.TRISB2 = 0; //NEULED

LATBbits.LATB2 = 0; //Start off

TRISBbits.TRISB8 = 0; //Plate light

LATBbits.LATB8 = 0; //Start off

TRISBbits.TRISB9 = 0; //Tail light

LATBbits.LATB9 = 0; //Start off

// Create Variables

SendI2C4(LED,LEDREG,CON,0x1E);

// run continously to keep program running

while(1)

{

//User Inputs

ChargeOn=PORTEbits.RE0;

Phase2=PORTEbits.RE1;

ChargeDone=PORTEbits.RE2;

On2=PORTBbits.RB14;

Haz=PORTBbits.RB13;

Night=PORTBbits.RB12;

Right=PORTDbits.RD5;

Left=PORTDbits.RD4;

But1=PORTBbits.RB11;

But2=PORTBbits.RB10;

//Outputs

if(On2&!ChargeOn){

LATDbits.LATD1= 1; //Turn on motor

error=0;

}

else if(On2&ChargeOn){

LATDbits.LATD1= 0; //Turn off motor

error=1;

}

else{

LATDbits.LATD1= 0; //Turn off motor

error=0;

}

if(Night){

LATBbits.LATB8 = 1; //Turn on plate light

}

else

LATBbits.LATB8 = 0; //Turn off plate light

}

return (EXIT\_SUCCESS);

}

//-------------------------------------------------------------------------------

// TO-DO: Configure timer and present message over I2C to display

void \_\_ISR(8, IPL3AUTO) Timer2Hand(void)

{

INTClearFlag(INT\_T2);

if(((Haz|error)|(Right|Left))&flash){ //Causes hazards to flash on and off

if(Haz|error)

LATBbits.LATB3 = 1; //Warning display On

else if(Right)

LATBbits.LATB4 = 1; //Right Light On

else

LATBbits.LATB5 = 1; //Left Light On

if(!error)

LATBbits.LATB9 = 1; //Tail Light On

flash=0;

}

else{

LATBbits.LATB3 = 0; //Warning display OFF

LATBbits.LATB5 = 0; //Left Light OFF

LATBbits.LATB4 = 0; //Right Light OFF

if(!breaking)

LATBbits.LATB9 = 0; //Tail Light OFF if not breaking

flash=1;

}

// write logic to communicate values to peripherals

// Display Status on I2C display

if(startcount<25){

SendI2C3(LED,LEDREG,LEDCLR);

SendI2C2(LED,L);

SendI2C2(LED,I);

SendI2C2(LED,G);

SendI2C2(LED,H);

SendI2C2(LED,T);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C2(LED,B);

SendI2C2(LED,I);

SendI2C2(LED,K);

SendI2C2(LED,E);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C2(LED,S);

SendI2C2(LED,T);

SendI2C2(LED,A);

SendI2C2(LED,R);

SendI2C2(LED,T);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C2(LED,U);

SendI2C2(LED,P);

startcount++;

}

else if(ChargeDone){

SendI2C3(LED,LEDREG,LEDCLR);

SendI2C2(LED,D);

SendI2C2(LED,O);

SendI2C2(LED,N);

SendI2C2(LED,E);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C2(LED,C);

SendI2C2(LED,H);

SendI2C2(LED,A);

SendI2C2(LED,R);

SendI2C2(LED,G);

SendI2C2(LED,I);

SendI2C2(LED,N);

SendI2C2(LED,G);

}

else if(Phase2){

SendI2C3(LED,LEDREG,LEDCLR);

SendI2C2(LED,T);

SendI2C2(LED,R);

SendI2C2(LED,I);

SendI2C2(LED,C);

SendI2C2(LED,K);

SendI2C2(LED,L);

SendI2C2(LED,E);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C2(LED,C);

SendI2C2(LED,H);

SendI2C2(LED,A);

SendI2C2(LED,R);

SendI2C2(LED,G);

SendI2C2(LED,I);

SendI2C2(LED,N);

SendI2C2(LED,G);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C2(LED,E);

SendI2C2(LED,S);

SendI2C2(LED,T);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C2(LED,T);

SendI2C2(LED,I);

SendI2C2(LED,M);

SendI2C2(LED,E);

SendI2C2(LED,COLON);

SendI2C3(LED,LEDREG,LEDRIGHT);

}

else if(ChargeOn){

SendI2C3(LED,LEDREG,LEDCLR);

SendI2C2(LED,C);

SendI2C2(LED,H);

SendI2C2(LED,A);

SendI2C2(LED,R);

SendI2C2(LED,G);

SendI2C2(LED,I);

SendI2C2(LED,N);

SendI2C2(LED,G);

SendI2C3(LED,LEDREG,LEDRIGHT);

}

else if (!On2){

SendI2C3(LED,LEDREG,LEDCLR);

SendI2C2(LED,R);

SendI2C2(LED,E);

SendI2C2(LED,A);

SendI2C2(LED,D);

SendI2C2(LED,Y);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C2(LED,T);

SendI2C2(LED,O);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C2(LED,T);

SendI2C2(LED,U);

SendI2C2(LED,R);

SendI2C2(LED,N);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C2(LED,O);

SendI2C2(LED,N);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C2(LED,B);

SendI2C2(LED,A);

SendI2C2(LED,T);

SendI2C2(LED,T);

SendI2C2(LED,E);

SendI2C2(LED,R);

SendI2C2(LED,Y);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C2(LED,V);

SendI2C2(LED,O);

SendI2C2(LED,L);

SendI2C2(LED,T);

SendI2C2(LED,A);

SendI2C2(LED,G);

SendI2C2(LED,E);

SendI2C2(LED,COLON);

SendI2C3(LED,LEDREG,LEDRIGHT);

}

else{

SendI2C3(LED,LEDREG,LEDCLR);

SendI2C2(LED,D);

SendI2C2(LED,R);

SendI2C2(LED,I);

SendI2C2(LED,V);

SendI2C2(LED,I);

SendI2C2(LED,N);

SendI2C2(LED,G);

SendI2C3(LED,LEDREG,LEDRIGHT);

SendI2C2(LED,M);

SendI2C2(LED,O);

SendI2C2(LED,D);

SendI2C2(LED,E);

SendI2C3(LED,LEDREG,LEDRIGHT);

}

}

// Configure bits for Timer operation

void ConfigTime()

{

// Stops Timer and Clears register

T2CON = 0x0;

TMR2 = 0x0;

// Set PR to 65535 originally 16000 with 3E80

PR2 = 0xAFFF;

// Set prescaler at 1:256

T2CONSET = 0x0070;

// Start Timer

T2CONSET = 0x8000;

INTConfigureSystem(INT\_SYSTEM\_CONFIG\_MULT\_VECTOR);

// Enables Global Interrupts

INTEnableInterrupts();

// Enables Timer2 Interrupts

INTEnable(INT\_T2, INT\_ENABLED);

// Clears timer2 flag

INTClearFlag(INT\_T2);

// Timer2 has priority 3

INTSetVectorPriority(INT\_T2,3);

}

//----------------------------------------------------------------------------------------------------------------------------

// Configure analog registers to read value from sensors

void ConfigAnalog()

{

// ensure the ADC is off before setting the configuration

CloseADC10();

// Turn module on |ouput in integer| trigger mode auto | enable autosample

#define PARAM1 ADC\_MODULE\_ON | ADC\_FORMAT\_INTG | ADC\_CLK\_AUTO | ADC\_AUTO\_SAMPLING\_ON

// ADC ref external | disable offset test | disable scan mode

// | perform 8 samples | use dual buffers | use alternate mode

#define PARAM2 ADC\_VREF\_AVDD\_AVSS | ADC\_OFFSET\_CAL\_DISABLE | ADC\_SCAN\_OFF | ADC\_SAMPLES\_PER\_INT\_2 | ADC\_ALT\_BUF\_ON | ADC\_ALT\_INPUT\_ON

// use ADC PB clock| set sample time | auto

#define PARAM3 ADC\_CONV\_CLK\_INTERNAL\_RC | ADC\_SAMPLE\_TIME\_15

// AN7 as analog inputs

#define PARAM4 ENABLE\_AN6\_ANA | ENABLE\_AN7\_ANA

// do not assign channels to scan

#define PARAM5 SKIP\_SCAN\_ALL

// configure to sample AN7 B7 and AN8

SetChanADC10( ADC\_CH0\_NEG\_SAMPLEA\_NVREF | ADC\_CH0\_POS\_SAMPLEA\_AN6| ADC\_CH0\_NEG\_SAMPLEB\_NVREF | ADC\_CH0\_POS\_SAMPLEB\_AN7);

// configure ADC using the parameters defined above

OpenADC10( PARAM1, PARAM2, PARAM3, PARAM4, PARAM5 );

// Note the 65 NS minimum TAD from datasheet, don't use FRM

//AD1CON3bits.ADCS=0x01;

EnableADC10();

}

// Read analog pin values for battery stack voltage and break voltage

void getAnalog()

{

while ( ! mAD1GetIntFlag() )

{

// wait for the first conversion to complete so there

// will be vaild data in ADC result registers

}

Battery\_volt = ReadADC10(7)\*.003185; // pin 7 stack voltage

Break\_volt = ReadADC10(6)\*.003185; // pin 6 break voltage

mAD1ClearIntFlag();

// Clear ADC interrupt flag

}

//------------------------------------------------------------------------------------------------------------------------------------

// Configure I2C registers

void ConfigI2C()

{

//1 USES RD10 AS SCL1 AND RD 9 AS SDA1

/// I2CxCON I2CxSTAT I2CxADD I2CxMSK I2CxTRN I2CxRCV

I2C1BRG=0x030; //390 for 80MHz to 100KHz

I2C1CONbits.A10M=0; //Use 7-bit addresses

I2C1CONbits.DISSLW=1; //disable slew control for standard

I2C1CONbits.ACKDT=0; //Use and ACK not NACK

I2C1ADD=22; //Sets slave address for PIC32

TRISD=1; //Sets Port D to input

I2C1CONbits.ON=1; //turn on I2C

}

// Start I2C

void I2C\_start(void)

{

I2C1CONbits.SEN=1; //send start

while(I2C1CONbits.SEN){} //waits till start bit detected

}

// Restart I2C

void I2C\_restart(void)

{

I2C1CONbits.RSEN=1; //send restart

while(I2C1CONbits.RSEN){} //waits till start bit detected

}

// Stop I2C

void I2C\_stop(void)

{

I2C1CONbits.PEN=1; //send stop

while(I2C1CONbits.PEN){} //waits till stop bit detected

}

// Write char of data to I2C line

char I2C\_write(char data)

{

I2C1TRN=data; //sends data to transmit register

while(I2C1STATbits.TRSTAT==1){} //waits to finsh transmission

return(I2C1STATbits.ACKSTAT); //returns 0 for ack received

}

// Check for acknowledgement

void mAckI2C1(void)

{

I2C1CONbits.ACKDT=0;

I2C1CONbits.ACKEN=1;

while(I2C1CONbits.ACKEN){}

}

// Check for lack of acknowledgement

void mNAckI2C1(void)

{

I2C1CONbits.ACKDT=1;

I2C1CONbits.ACKEN=1;

while(I2C1CONbits.ACKEN){}

}

// Read data back from I2C line

char I2C\_read(char ack)

{

I2C1CONbits.RCEN=1;

while(I2C1CONbits.RCEN){}

//Reception is started, send ack/nack after read

if(ack==0)

{mNAckI2C1();}

else

{mAckI2C1();}

//Reception should be complete - pull out data

return(I2C1RCV);

}

// Make I2C line wait for registers to clear

void I2C\_idle()

{

while((I2C1CON&0x001F)!=0){}

// Wait for Acken, Rcen, Pen, Rsen and Sen to clear

}

// Send data to I2C line at given address

void SendI2C3(char addrs,char regis, char data)

{

char ack;

I2C\_start();

ack=I2C\_write(addrs); //Address for LED is 0x50

ack=I2C\_write(regis); //0xFE for LED

ack=I2C\_write(data); //0x20to0x7F standard

I2C\_stop();

}

void SendI2C4(char addrs,char regis, char data, char con)

{

char ack;

I2C\_start();

ack=I2C\_write(addrs); //Address for LED is 0x50

ack=I2C\_write(regis); //0xFE for LED

ack=I2C\_write(data); //0x20to0x7F standard

ack=I2C\_write(con); //0x20to0x7F standard

I2C\_stop();

}

// Writes to standard registers

void SendI2C2(char addrs, char data)

{

char ack;

I2C\_start();

ack=I2C\_write(addrs); //Address for LED is 0x50

ack=I2C\_write(data); //0x20to0x7F standard

I2C\_stop();

}

/\*

\* File: configbits.h

\* Author: Mike

\*

\* Created on October 9, 2012, 1:50 PM

\*/

#ifndef CONFIGBITS\_H

#define CONFIGBITS\_H

/\* 20 MHz crystal run at 80 mhz

peripher clock = at 10 MHz (80 MHz/8)

\*/

#pragma config FNOSC = PRIPLL // Oscillator selection

#pragma config POSCMOD = HS // Primary oscillator mode

#pragma config FPLLIDIV = DIV\_5 // PLL input divider (20 -> 4)

#pragma config FPLLMUL = MUL\_20 // PLL multiplier ( 4x20 = 80)

#pragma config FPLLODIV = DIV\_1 // PLL output divider

#pragma config FPBDIV = DIV\_8 // Peripheral bus clock divider 10 mhz

#pragma config FSOSCEN = OFF // Secondary oscillator enable

/\* Clock control settings

\*/

#pragma config IESO = ON // Internal/external clock switchover

#pragma config FCKSM = CSDCMD // Clock switching (CSx)/Clock monitor (CMx)

#pragma config OSCIOFNC = OFF // Clock output on OSCO pin enable

/\* USB Settings

\*/

/\* Other Peripheral Device settings

\*/

#pragma config FWDTEN = OFF // Watchdog timer enable

#pragma config WDTPS = PS1024 // Watchdog timer post-scaler

#pragma config FSRSSEL = PRIORITY\_7 // SRS interrupt priority

#pragma config ICESEL = ICS\_PGx1 // ICE pin selection

#endif /\* CONFIGBITS\_H \*/

/\*

\* File: Other.h

\* Author: mmellitt

\*

\* Configures I2C on board and calls other configure functions

\*

\* Created on February 18, 2013, 3:04 PM

\*/

#ifndef OTHER\_H

#define OTHER\_H

#ifdef \_\_cplusplus

extern "C" {

#endif

void ConfigTime();

void ConfigAnalog();

void getAnalog();

void ConfigI2C();

void I2C\_start(void);

void I2C\_restart(void);

void I2C\_stop(void);

char I2C\_write(char data);

void mAckI2C1(void);

void mNAckI2C1(void);

char I2C\_read(char ack);

void I2C\_idle();

void SendI2C3(char addrs, char regis, char data);

void SendI2C2(char addrs, char data);

void SendI2C4(char addrs,char regis, char data, char con);

#ifdef \_\_cplusplus

}

#endif

#endif /\* OTHER\_H \*/

*Appendix C: Charging Board Schematic*

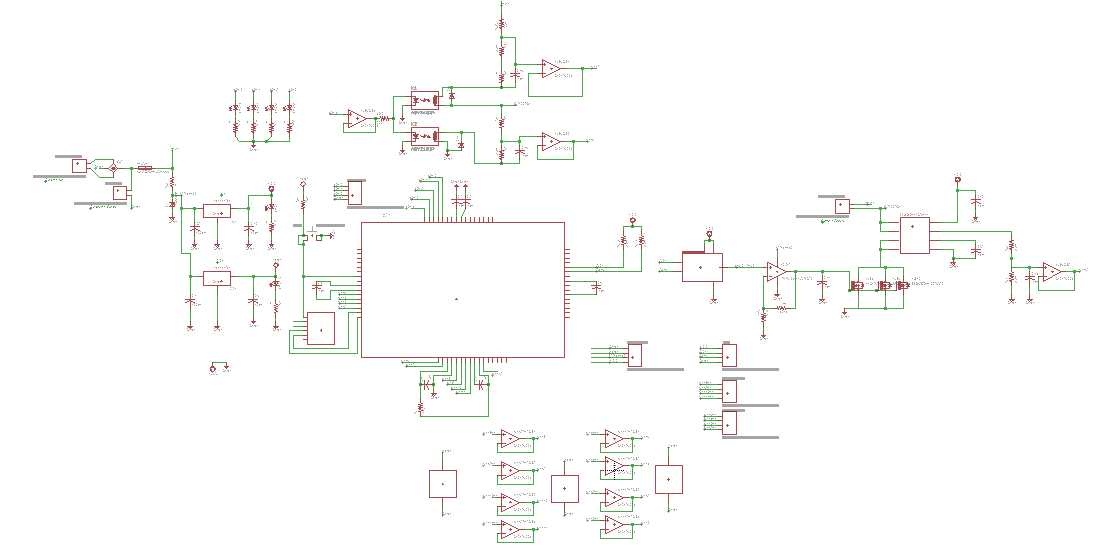


Figure 8- Charging Subsystem

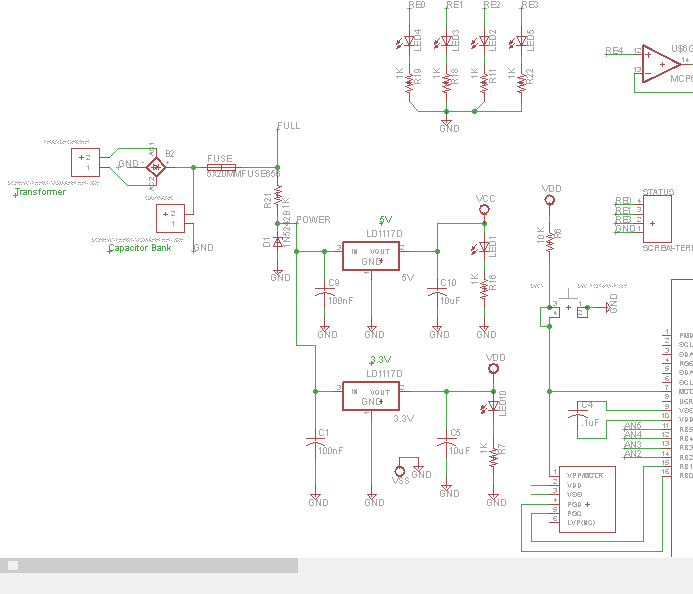


Figure 9a- Left Side of Charging Schematic

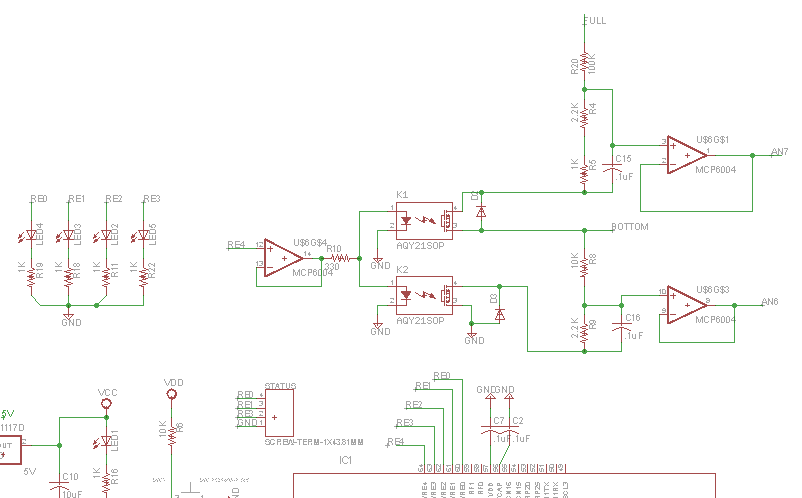


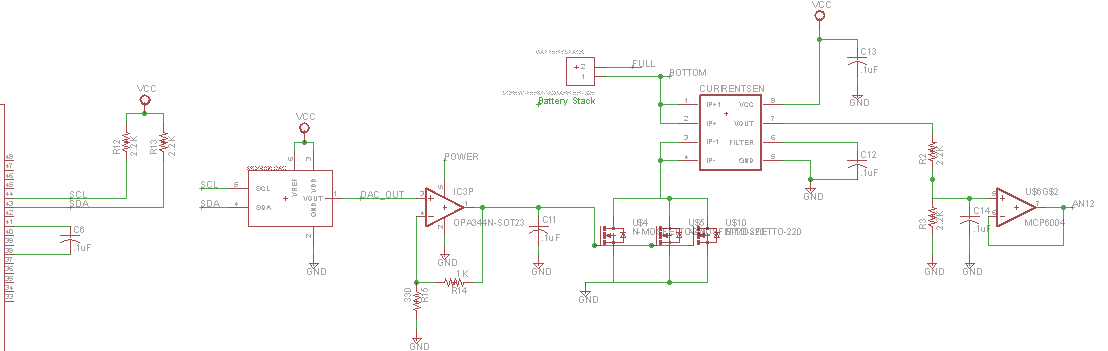
Figure 9b- Top of Charging Schematic

Figure 9c- Right Side of Charging Schematic

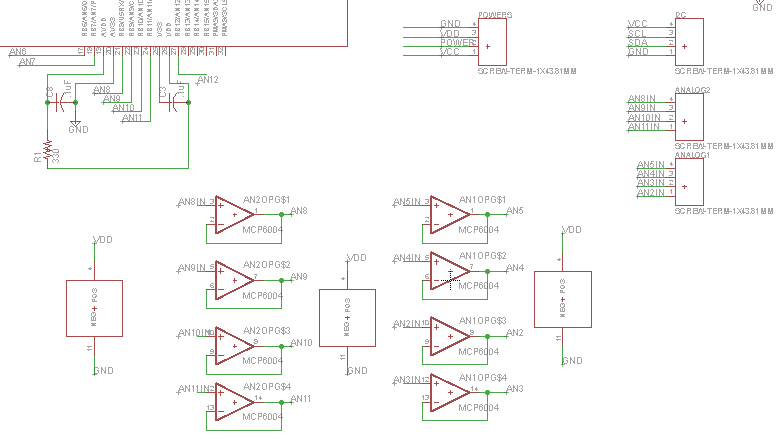


Figure 9d- Bottom of Charging Schematic

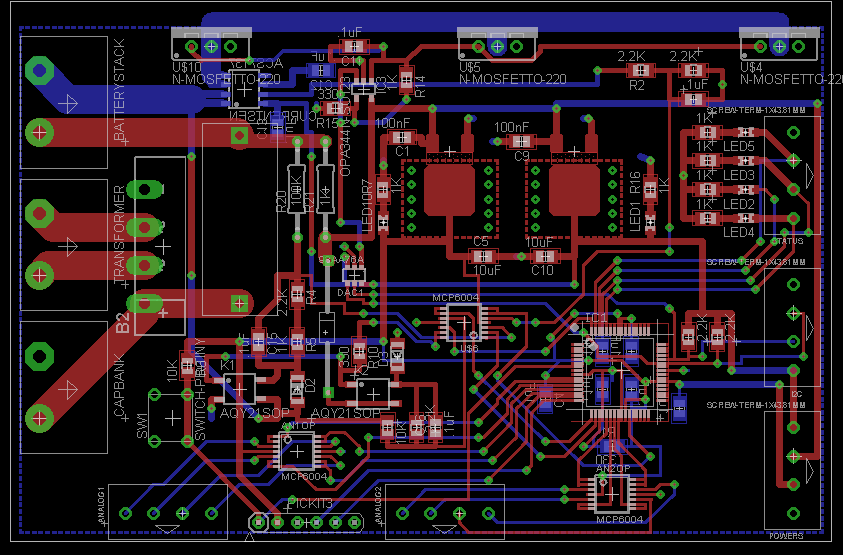
*Appendix D: Charging Board File*

Figure 10- Charging Board Layout

*Appendix E: Peripheral Board Schematic*

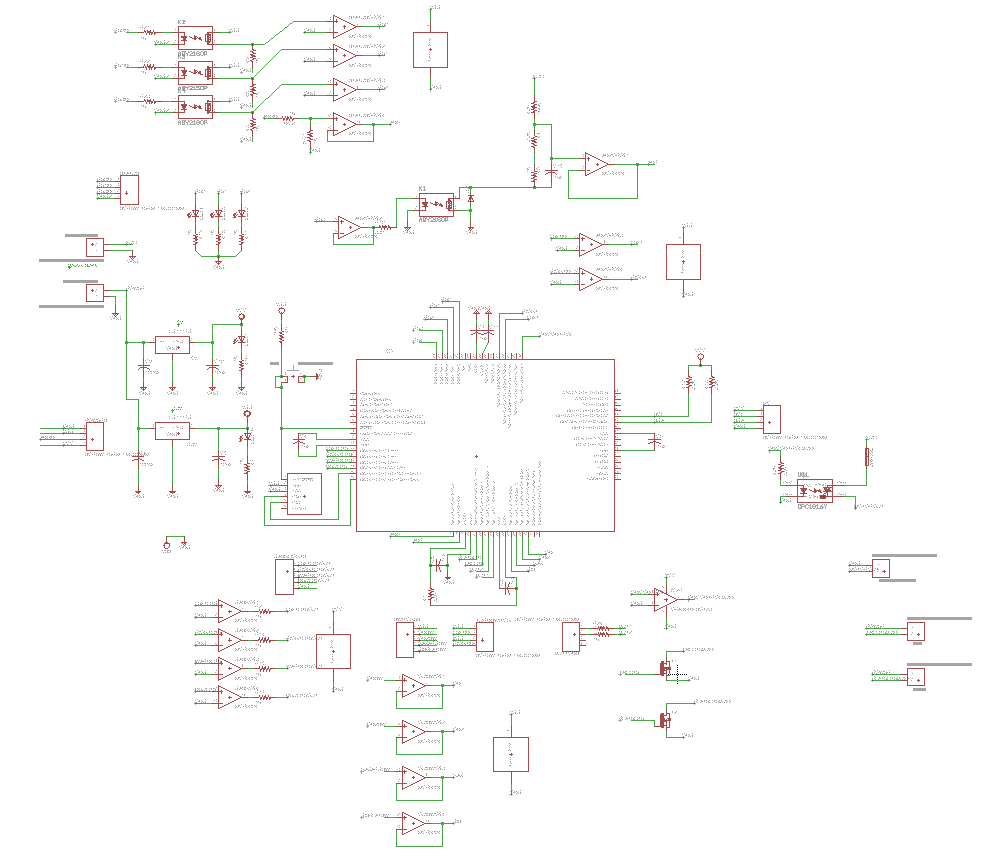


Figure 11- Peripheral Board Schematic

*Appendix F: Peripheral Board File*

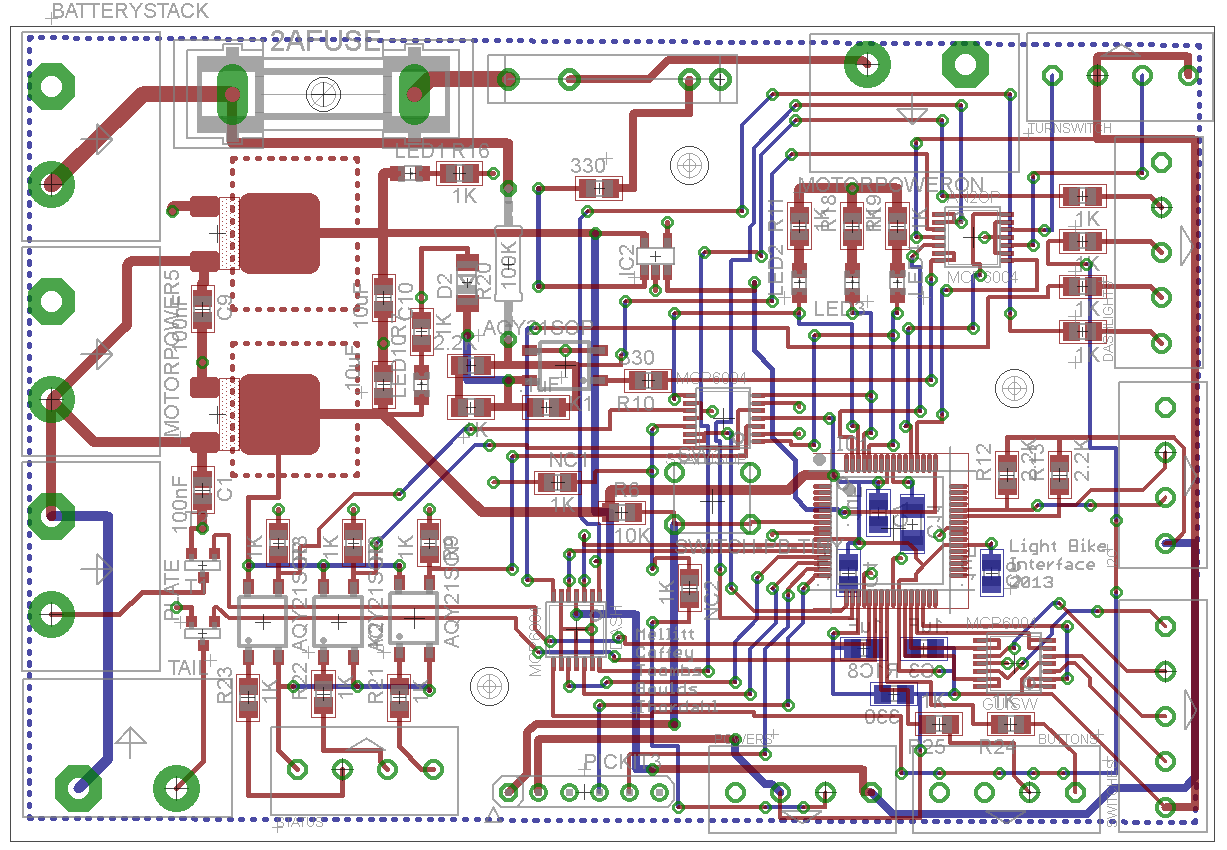


Figure 12- Peripheral Board Layout

*Appendix G: Motorcontroller Schematic*

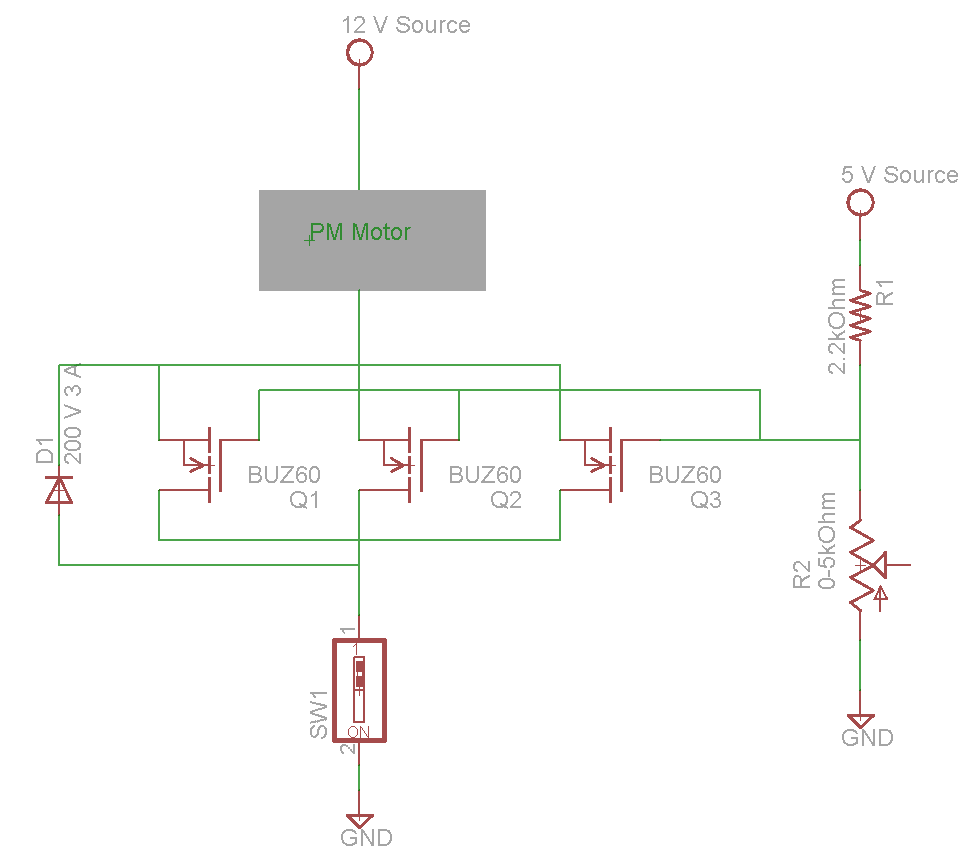


Figure 13- Mototrcontroller Schematic

*Appendix H: Component Datasheet Links*

Table 1 - Component Descriptions and Datasheet Links

|  |  |  |
| --- | --- | --- |
| **Part Number** | **Description** | **Datasheet Link** |
| ACS713ELCTR-30A-T | Hall-Effect current sensor | <http://www.allegromicro.com/en/Products/Current-Sensor-ICs/Zero-To-Fifty-Amp-Integrated-Conductor-Sensor-ICs/ACS713.aspx> |
| PIC32MX695F512H | Embedded microcontroller | <http://www.microchip.com/pagehandler/en-us/family/32bit/> |
| LM35 | Digital temperature sensor | <http://www.ti.com/product/lm35> |
| MCP6004 | Quad op-amp for unity gain buffers on analog inputs | <http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en010435> |
| SUP57N20 | Power MOSFET for battery charging | <http://www.vishay.com/mosfets/list/product-72100/> |
| TS321 | Op amp voltage buffer for DAC output | <http://www.ti.com/product/ts321> |
| Optima Yellow Top D51 | Batteries for bike | <http://www.optimabatteries.com/us/en/products/yellowtop/> |
| MCP4706 | 8-bit DAC for I2C-based control of MOSFET gate voltage | <http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en552631> |
| NHD-0220D3Z-FL-GBW-V3 | LCD display for peripherals | <http://www.newhavendisplay.com/nhd0220d3zflgbwv3-p-5742.html?zenid=6651soskceahlceagoett9o5p5> |
| Kelly Motor Controller KD72301 | Former motor controller for bike; now obsolete. | <http://kellycontroller.com/mot/downloads/KellyKDUserManual.pdf> |