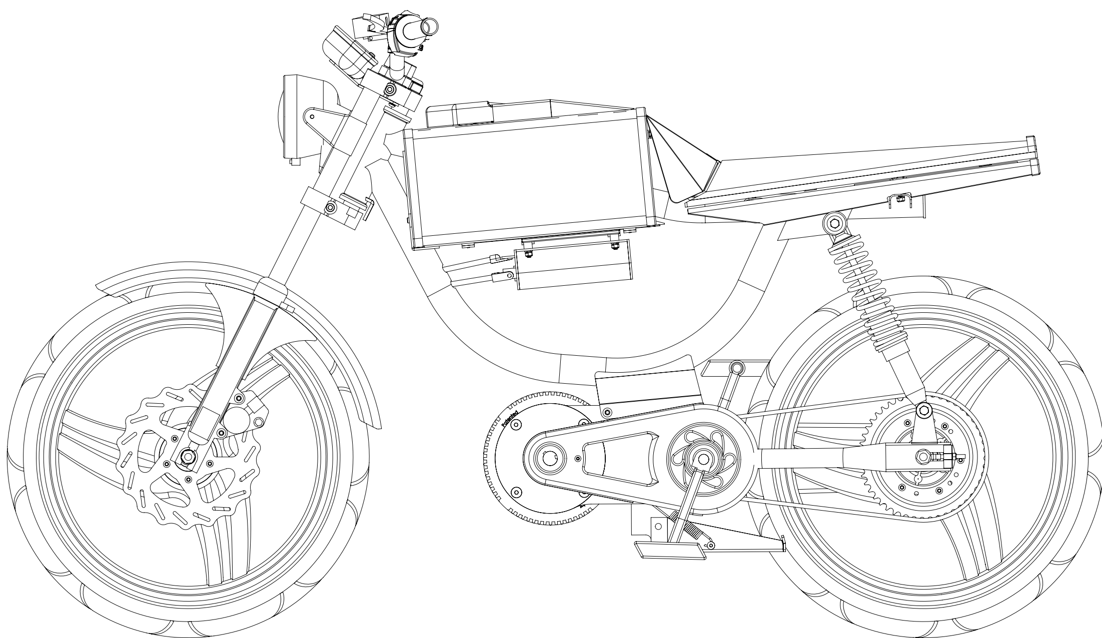
Bolt Motorbikes Mid-Project Report
Senior Design Project 2015-2016

Alexis Bartels, Bruce Gordon, Garrett Deguchi,
Francisco Alvarado & Zachary Levenberg

April 5, 2016



BOLT ENGINEERING UCSC

Contents

1 Project Description Overview **5**

2 Bike on a Board **6**
2.1 Purpose . . . 6
2.2 Overview . . . 6
2.3 Significant Findings . . . 6

3 Microcontroller Hardware **7**
3.1 Overview . . . 7
3.2 Microcontroller Selection . . . 7
3.3 Minimum Specs/Dev kits/Experimentation . . . 8

4 Power Distribution **9**
4.1 Overview . . . 9
4.2 Power Budget . . . 10
4.3 Switching Regulators . . . 11

5 Battery Management System **12**
5.1 Overview . . . 12
5.2 Algorithm . . . 12
5.3 Theory of Operation . . . 13

6 Internet of Things **14**
6.1 Overview . . . 14
6.2 System Description . . . 14
6.3 RN4020 Command Interface . . . 14
6.4 Bluetooth Services and Characteristics . . . 16
6.5 Android App Development . . . 17

7 Microcontroller Network **18**
7.1 Dashboard . . . 18
7.1.1 Input/Output . . . 18
7.1.2 State Machine . . . 19
7.1.3 Dash Circuitry/PCB . . . 20
7.2 Main Control Unit . . . 21
7.2.1 Input/Output . . . 21
7.2.2 StateMachine . . . 22
7.2.3 Main Control Unit Circuitry/PCB . . . 23

8 Software **24**
8.1 Peripheral Modules . . . 24
8.1.1 Oscillator . . . 24
8.1.2 Analog to Digital . . . 25
8.1.3 LCD Screen . . . 25
8.2 Framework . . . 28

9 Conclusion **28**

Team Member Contact Information

Alexis Bartels
Robotics Engineering/ EE minor
abartels@ucsc.edu
alexis.bartels@gmail.com
661-369-6233

Bruce Gordon
Computer Engineering / Physics minor
brdgordo@ucsc.edu
619-609-6530

Garrett Deguchi
Bioengineering/CE Minor
gdeguchi@ucsc.edu
707-548-6266

Francisco Alvarado
Electrical Engineering
fjalvara@ucsc.edu
francisco.alvarado22@gmail.com
707-228-6385

Zachary Levenberg
Robotics Engineering
zlevenbe@ucsc.edu
zach.levenberg@gmail.com
415-812-8554

Abstract

Bolt Motorbikes is a Bay Area start-up company co-founded by team member Zachary Levenberg in 2014. The existing Bolt Motorbikes team has created a completely electrical, yet street legal, Bolt M1 Motorbike. The goal of Bolt motorbikes is to create an energy efficient, stylish and environmentally conscious alternative form of urban transportation. By blending a bicycle with the look and feel of a motorcycle, the Bolt M1 Motorbike is changing transportation as we know it. In order to make the Bolt M1 a scalable transportation solution, the internals of the Bolt Motorbike needs to be retrofitted in order to optimize cost, manufacturability and engineering design. Team Bolt at UCSC has been asked to recreate the internal systems of the existing bike and to deliver electrical, connectivity and sensor solutions for the second major revision of the bike to be manufactured in late 2016. Team Bolt will focus on redoing the electrical and battery management system, sensor network and controller framework while adding Internet of Things (IoT) connectivity. The following mid-project report describes our engineering design procedures and technical discoveries.

Bolt Motorbike System Overview

The Bolt M-1 is an all-electric motorbike, that is classified as an electric bicycle. It has pedals, but is intended to be used as a motorbike. The M-1 features a 1.68kWhour battery pack, a 5kW continuous motor with a 300 Amp peak current. The bike features an embedded system with a user-friendly dashboard and an LCD screen. The M-1 has headlights, taillights, a horn, a USB accessory charger, and a sensor network that monitors to bike. The M-1 is driven by a twist throttle. There are no gears to shift, just twist and go! The front brake is a traditional hydraulic disc brake, and the rear is a drum. The rear brake also consists of a regenerative braking system, which allows the user to extend battery life by riding smarter. The bike can go for 50 miles on a single charge on flat ground. Typical mileage depends on the riding style. The bike has three modes: Sport, Econ, and Demo. Sport is the fully unlocked 38mph ”off-road-only” setting. Econ is the legal bicycle mode where the top speed is limited to 20mph and power output to 2kW. Demo is for indoor using when demonstrating the functionality of the bike.

1 Project Description Overview

The Bolt team at UCSC will focus on five major systems: battery management system, power distribution system, sensor system, main control unit and software framework, and Internet of Things connectivity. The battery management system will monitor, charge, and balance the batteries on the Bolt motorbike. The power distribution system will efficiently regulate and distribute power to the motorbike. The sensor system will use a network of digital and analog sensors to collect and display data about the motorbike. The main control unit is the software harness which regulates the state-machine of the motorbike. In addition, the main control unit will regulate the dashboard display to the motorbike. Internet of Things will be adds bluetooth connectivity to the motorbike and an accompanying phone app to display and store motorbike data. Each module will communicate to each other via controller area network bus (CANbus). CANbus is an industry standard communication protocol in the automotive industry. In order to facilitate the testing of each motorbike component, a desktop ”bike on a board” was constructed. An overview of the system can be seen in Figure 1 which demonstrates each component of the system working together.

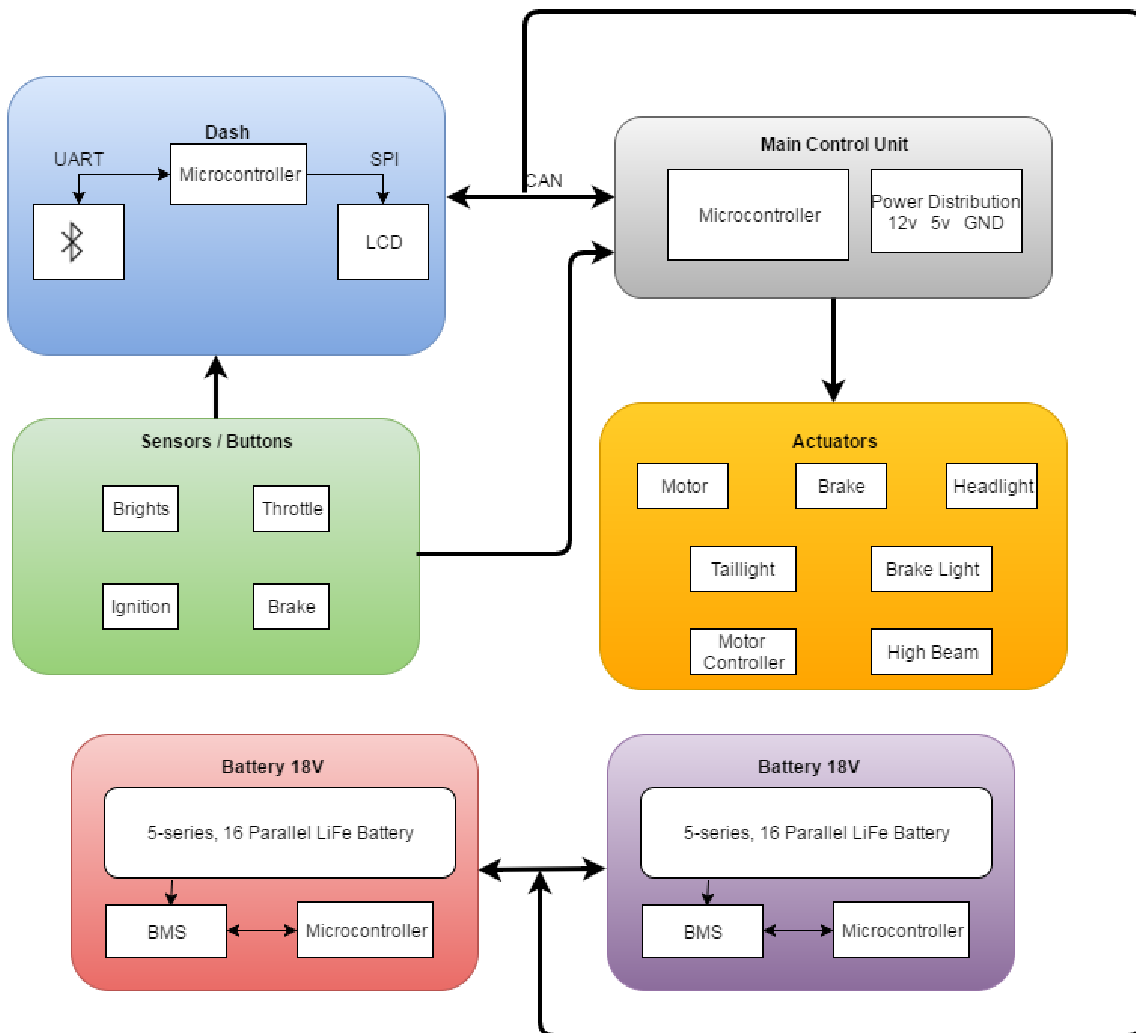


Figure 1: An overview of the Bolt motorbike system

2 Bike on a Board

2.1 Purpose

The purpose of the Bike on a Board is to provide a platform to test the integration of hardware and software. The idea is to have each component and sensor readily available for testing new software changes. The Bike on a Board allows the team to see the physical results a specific component or on the whole motorbike system. The Bike on a Board serves as a hardware-accessible alternative to for the development stages of the project as opposed to using an existing Bolt M1 Motorbike.

2.2 Overview

The Bike on a Board consists of each component present on a real Bolt M1 motorbike, except mounted on a flat desktop board to facilitate accessibility and component testing. The Bike on a Board was conceptualized during a team meeting and designed in CAD as seen in Figure 2. The idea was to make the Bike on a Board a functional testing piece, but also an interactive display to demonstrate how the Bolt M1 Motorbike works. A handlebar is mounted along with a functioning headlight, taillight, motor, motor controller, power supply and wheel. The buttons and switches on the handlebar are able to control the bike, just as if the user were driving a real Bolt M1. The wheel is held in tension and mounted upon two rollers such that it is actually able to be driven by the motor. Along with parts from the Bolt M1, various breadboards are used to electrically wire and connect the components while testing.

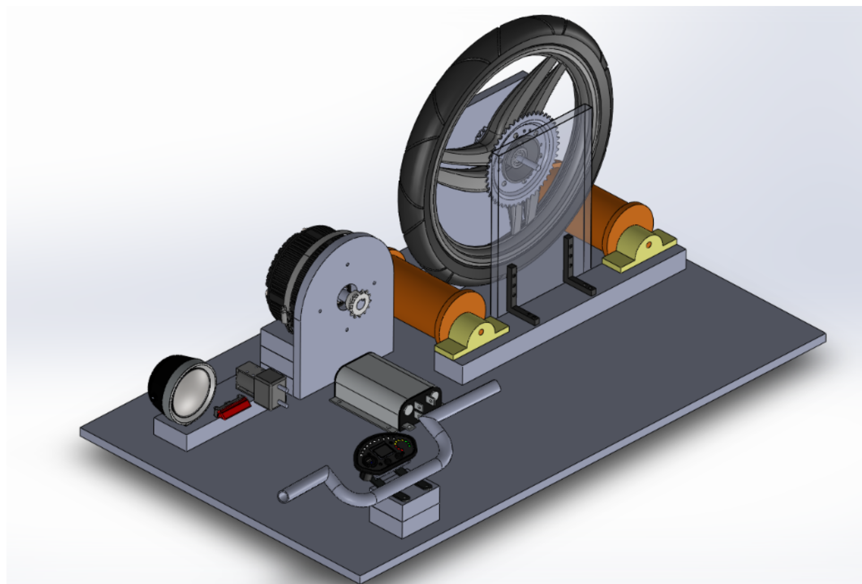


Figure 2: Bike on a Board CAD design

2.3 Significant Findings

The Bike on Board has proven invaluable for testing the logic of our state machine and the communication between our sensors, microcontroller, and actuators. We have been able to set up a main control state machine which allows the user to control headlights, brake lights, and motor through the appropriate inputs and via messages, which are currently sent from a computer via UART, but which will eventually come from the dash via CANbus. Figure 3 shows the physical build up of the Bike on a Board.

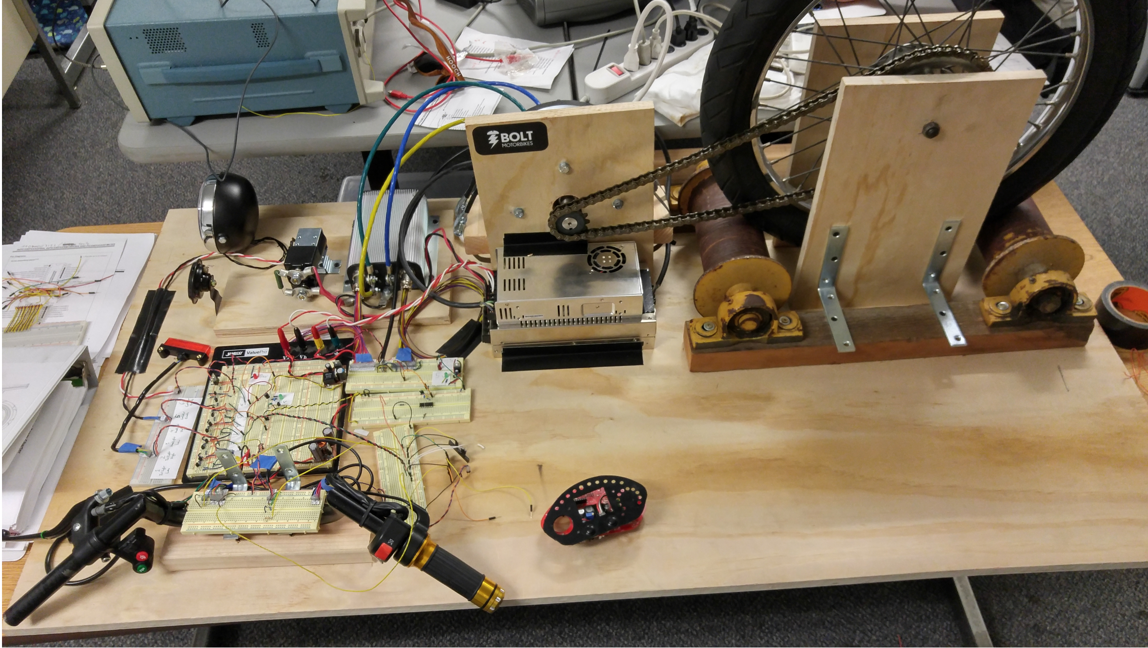


Figure 3: The Bike on a Board system after use in the lab

3 Microcontroller Hardware

3.1 Overview

A Microcontroller is needed to serve as the "brains" of the project; microcontrollers store the state of the motorbike, process sensor information and control actuators. All system inputs will be processed by the microcontroller and outputs are generated from the microcontroller accordingly. For example, changing the throttle of the will send a signal to the microcontroller which will accelerate or decelerate the motorbike. Selecting and then interfacing with a microcontroller was one of the earliest goals of our project. Selecting a microcontroller which did not have the correct features could cost our project crucial time. Additionally, in order to save on cost and isolate only desired features we interfaced with the microcontrollers using custom development kits.

3.2 Microcontroller Selection

We researched several microcontroller options for use in the Bolt Motorbike project. Our main concerns and criteria included cost, availability, and cost of development kits or Dual in-line packages (DIP), CANbus functionality, and pin count. We looked at various controllers from several different brands, including the Microchip dsPIC33EP32GP502, the MicroChip dsPIC33EP256MC502, the MicroChip dsPIC33EP512MC502, the Texas Instruments Hercules TMS57004 LaunchPad, the Atmel AT90CAN32, and the Atmel AT90CAN32. The microcontroller that we decided to use going into the development stage of this project is the Microchip dsPIC33EP256MC502. We selected this controller for several reasons. Of large influence in this decision is the fact that it is among the cheapest chips that we found that met all our other requirements, and could be obtained in relatively small quantities. The prices for the Atmel chips were not outrageous, but they could only be ordered in quantities of 1000, a number that would have put us well over budget and in quantities that were not necessary for our project. The TI chips were generally very expensive. The MicroChip chips were by far the most reasonable, economically speaking which was a big factor in our decision to select a MicroChip microcontroller.

3.3 Minimum Specs/Dev kits/Experimentation

In order to successfully develop code on our Microchip dsPIC33 microcontrollers, we needed functional development boards. Rather than purchase the more expensive pre-made development boards sold by chip manufacturers, we purchased dual-inline package versions of our controller and created our own development boards on perforated prototype boards. These development boards include the microcontroller, a voltage regulation circuit that includes current fusing and reverse polarity protection, various decoupling capacitors other minimum operation circuitry for the controller, as well as header pin input/output ports and power and ground rails. The schematic for the boards is in Figure 4.

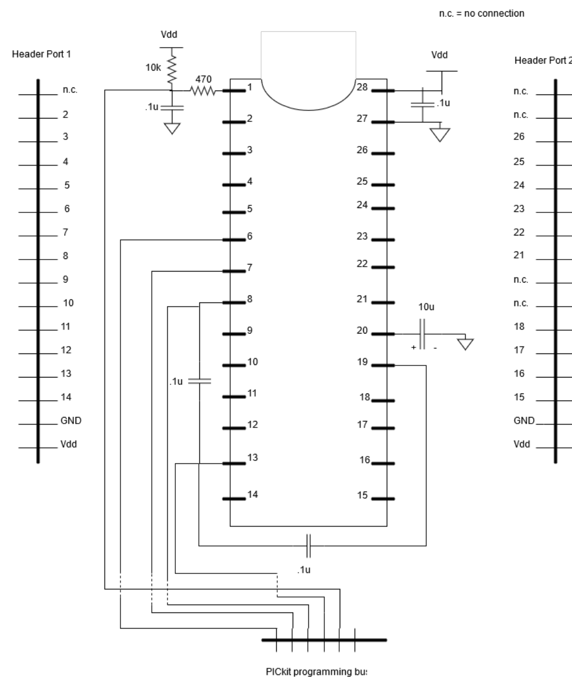


Figure 4: Minimum Operation Schematic

Each development kit was brought up by hand by team members. The development boards were soldered and tested for functionality. Figure 5 shows one of the five development kits our team created. Although creating custom development boards was more time consuming than an off the shelf board or kit, the money saved and the ability to have the minimum amount of features is crucial for our project. After the prototyping stage, printed circuit boards (PCB's) were created in order to fit within the space confinement of the motorbike. Knowing exactly how to use our microcontroller for the PCBs were a big benefit of creating our own development boards.

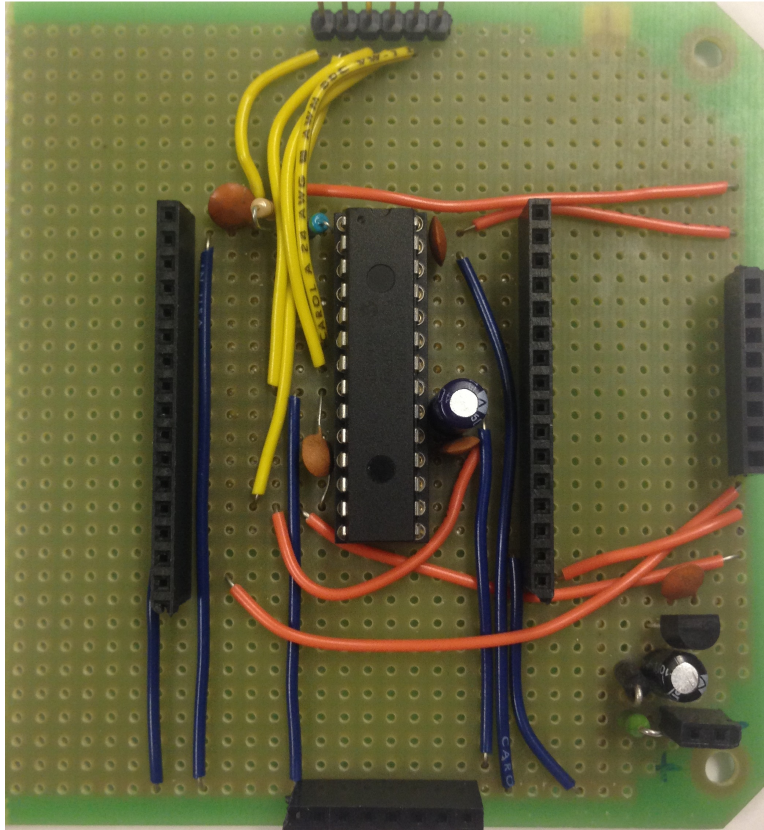


Figure 5: Development Board

4 Power Distribution

4.1 Overview

Almost every component of the motorbike requires electrical power. The power distribution board is where the supply power is stepped down to interface with all of our components (see Figure 6). The bike is powered from a 36V Lithium Iron Phosphate (LiFe) battery. At a high level, the bike has three main power levels: 36V, 12V, and 5V. The 36V rail supplies power for the main contactor relay, the motor controller, and the motor itself. The 12V rail is for peripheral power and is the standard operating voltage for most vehicles. The Headlight, taillight, and horn are all 12V devices. There is also an auxiliary output to power optional devices. The 5V rail is for the MCU, sensor network, and other integrated circuits. Because there are multiple nodes on this system, each node will have its own linear regulated supply voltage of 3.3V and is not represented in the power distribution system.

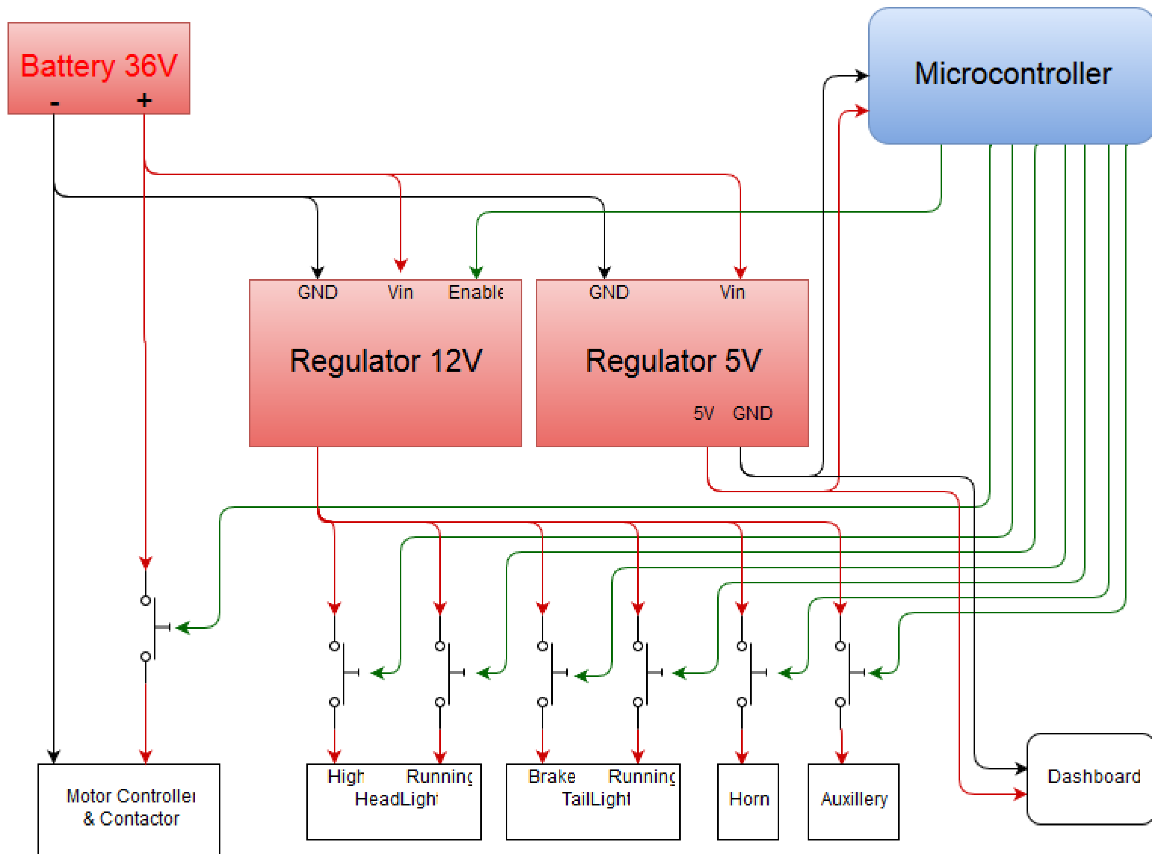


Figure 6: Power Distribution Diagram

4.2 Power Budget

In order to ensure the motorbike would be electrically sound, we created a power budget with the minimum power requirements and ensured our power distribution system was capable of delivering at least the minimum amount of power. Both the 12V system and the 5V system use switching regulators by Texas Instruments. The 12V regulator is an LM2679. This switching regulator is part of TIs simple switcher family and was designed using the WeBench software provided by TI. The LM2679 has a maximum input voltage of 40V and output current of 5Amps (see minimum specifications in Figure 7).

Item	Current (mA)	Total mWatts
36V Rail		
Contactor	350	
Controller	65	
Total	**415**	**14940**
12V Rail		
HeadLight (HB)	290	
HeadLight (LB)	67	
TailLight	4	
BrakeLight	64	
Horn	1500	
Turn Signals	250	
Battery Latch	800	
Total	**2975**	**35700**
5V Rail		
Dash LEDs	360	
Dash LCD	80	
USB charger	500	
Dash uC	10	
Brain uC	10	
Total	**940**	**4700**
Total System		**55340**

Figure 7: Power Budget for the Bolt M1

4.3 Switching Regulators

Selection of this component was determined by minimum power specifications and ease of design. Compared to other switchers, particularly the Linear Technology LT3976, the LM2679 uses the minimum number of components while allowing for maximum design flexibility and free design software. The 12V system was prototyped using the M60 design process where the circuit board is routed on a CNC router in Figure 8. This allowed the team to quickly prototype the complete power distribution board and interface with the Bike on a Board.

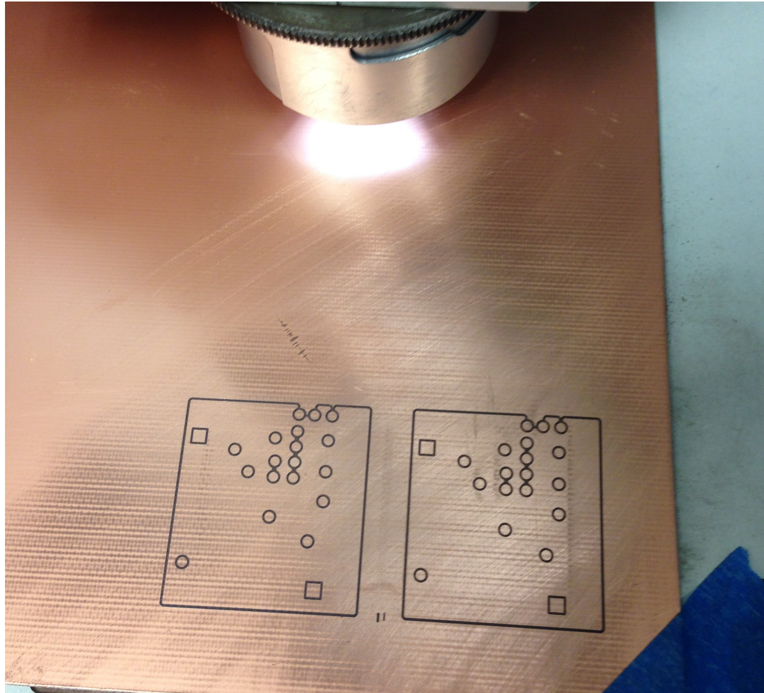


Figure 8: Circuit boards being routed using the M60 CNC machine.

5 Battery Management System

5.1 Overview

Since the Bolt M1 motorbike is all electrical, the charging and maintenance of batteries is a key point to the product. The current bolt motorbike uses an off-the-shelf Junsi smart charger that requires a power rail, and a separate 11 output connection to monitor 10 individual battery cells. The goal for the battery management system was to have one power connection, which would connect into two 18V battery packs that are in series. Additionally, it needed to balance the cells inside each individual pack, and capable of communicating with the main brain. This will improve the user experience of charging the batteries of the Bolt M1 motorbike.

In addition to charging the batteries, monitoring and balancing the batteries when they are in use is another critical part of our project. Proper battery management can extend the life and power output of the batteries, making the motorbike able to ride longer and ultimately improve the user experience. The goal for the quarter was to have one battery management system balancing a pack of 5 cells in series. This was accomplished by using an LTC6802-2 which is a complete battery monitoring circuit that includes a 12-bit ADC, a precision voltage reference, a high voltage input multiplexer and a serial interface. Since the LTC6802-2 is capable of monitoring 12 input channels, and the current battery only requires 5 inputs it fulfills one of the main concerns for the battery management system that was scalability of the battery voltage.

5.2 Algorithm

The first state monitors the cells and temperature readings and if a low voltage or temperature flag appears it sends a notification to the user. If the charger is connected it will go into balancing mode.

The second state waits for the charger to be plugged in and will remain in an idle condition unless the batteries are charging. This prevents the user from damaging the batteries.

The last state uses passive cell balancing which dissipates the highest cell batteries at the correct time

to get all the batteries balanced, and reads the temperature to prevent damages to the cells. Once this state is done it goes back to the first state.

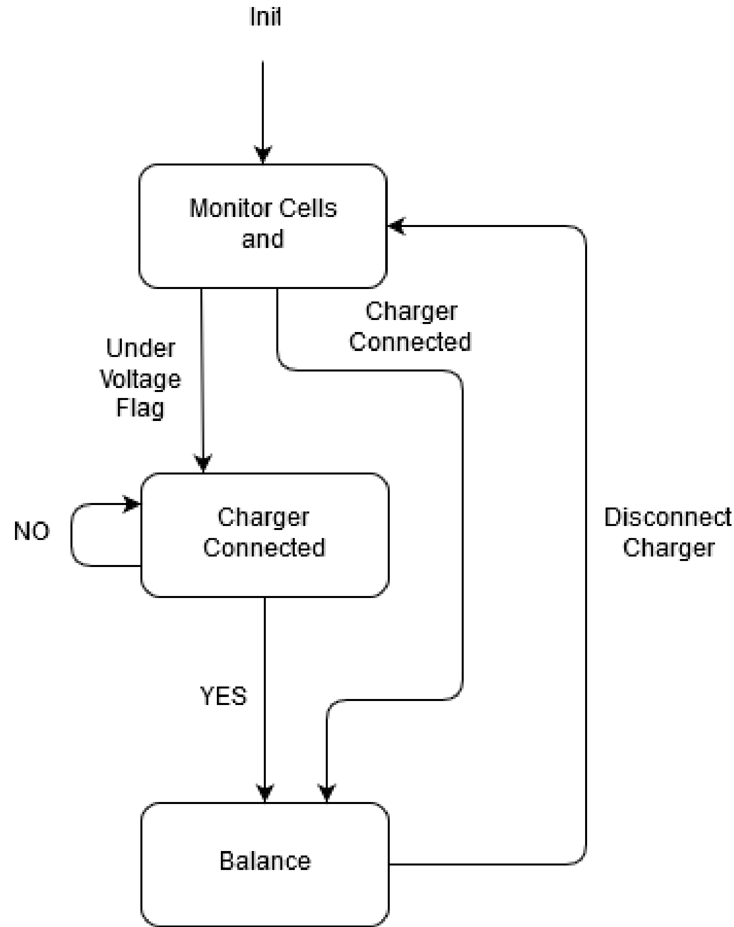


Figure 9: Flow Chart for BMS Charging

5.3 Theory of Operation

The LTC6802-2 uses Passive cell balancing which will dissipate excess energy through a bypass resistor until the voltage matches the voltage of the weaker cells. This was done with a P-channel MOSFETs controlled through the LTC S pins, and 10Ω, 2W flameproof resistors. The S pin had to be connected in series with a 3.3kΩ resistor to dissipate heat outside of the LTC6802-2.

The pic33EP256MC502 was used to control the LTC6802-2, which used an opto-isolator that allows two separate battery packs with different ground references, 18V and 0V, to communicate in the same CAN bus. Part of the challenge was to create an algorithm which would balance the cells quickly, and monitor the temperature. Figure 10 shows the unbalanced battery pack being charged and balanced. Since each of the five cells start at different voltages and charge up to the same 3.6V reference voltage, the pack of 5 cells are being balanced properly.

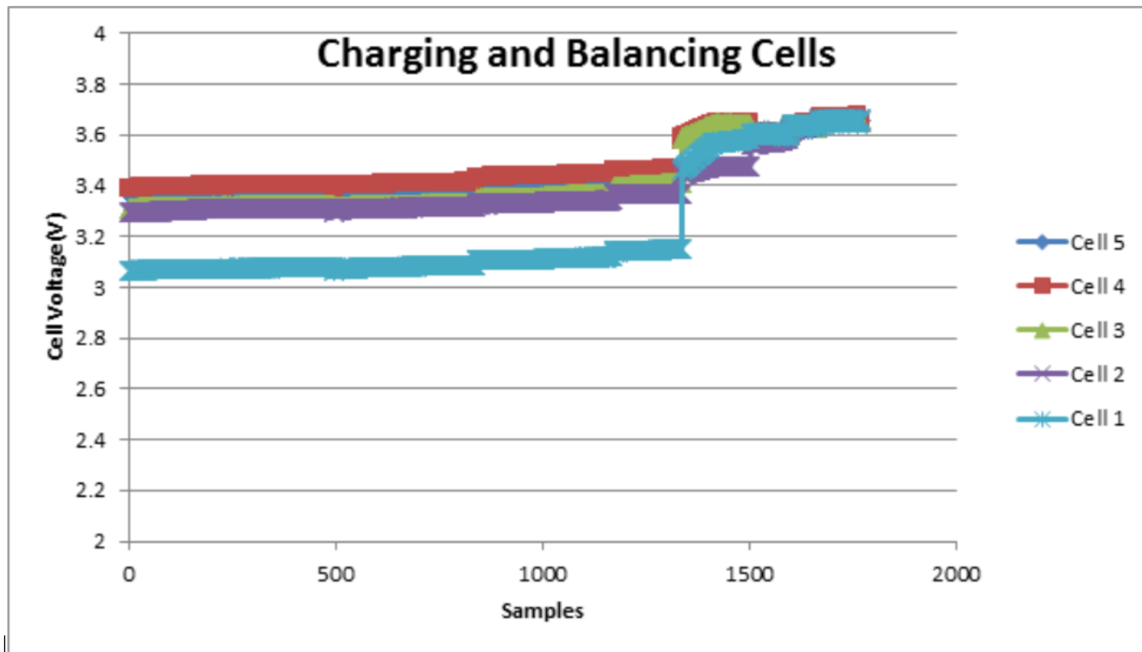


Figure 10: 5 Cells in Series Being Charged at 1Amp. There are 200 missing samples between times 1400 and 1401 causing the sharp jump in the graph.

6 Internet of Things

6.1 Overview

One of the requirements of the revised Bolt M1 is to add Internet of Thing (IoT) connectivity by adding Bluetooth Low Energy (BLE) and connection to an internet-enabled smart phone. Adding Bluetooth connection to the motorbike allows for user experience improvements to the Bolt motorbike experience. For example, after adding Bluetooth connectivity, an app can be paired with the Bolt motorbike which allows the user to unlock the bike or view data collected during rides such as battery level or speed.

6.2 System Description

For this task, the microchip RN4020 V1.23 was selected as a fully FCC certified Bluetooth Version 4.1 low energy module. The RN4020 is a surface mount chip which is interfaced via ASCII commands over UART. This module is controlled by a microcontroller and used Bluetooth SIG profiles with Generic Attribute Profile (GATT) services to transmit data over the air. The Bolt team interfaces with the RN4020 via Microchips Pictail developer board. In addition to the hardware, an Android application was created in Java to receive and analyze data as well as change settings of the motorbike.

6.3 RN4020 Command Interface

The RN4020 uses an ASCII command interface sent via UART to communicate. Commands are used to configure the board, set attributes, perform an action or read and write to GATT services. In order to initially test the configurations of the RN4020 module, a serial interface was established using PuTTY to manually send commands to configure the module. Using PuTTY, connection between the phone (or central device) and RN4020 module (or peripheral device) was achieved by commanding the RN4020 to advertise. After a serial connection between the RN4020 is established, the board can be configured. For example, the name characteristic of the module was changed to BoltBLE using the the SN command. A sample command sequence and explanation of the commands are listed in Figure 11 with the corresponding explanations in Table 1.

```
COM23 - PuTTY
Echo On
BTA=001EC01D3E6E
Name=BoltBLE
Connected=no
Bonded=no
Server Service=D0000000
Features=20000000
TxPower=4
180A
  2A25,000B,V
  2A27,000D,V
  2A26,000F,V
  2A28,0011,V
  2A29,0013,V
  2A24,0015,V
1809
  2A1C,0018,V
  2A1C,0019,C
  2A1D,001B,V
180F
  2A19,001E,V
  2A19,001F,C
END
Connected
Connection End
Connected
BTA=001EC01D3E6E
Name=BoltBLE
Connected=4911F187B526,1
Bonded=no
Server Service=D0000000
Features=20000000
TxPower=4
WC,001F,0100.
```

Figure 11: Sample command Sequence

Command	Result
+	Echo on (turns on type-echoing in terminal).
D	Dump of current configuration status, including device MAC address, Device name, connection roll, active services and features and TX power level. Note the change in the connection characteristic after a connection is established.
LS	List of Client GATT services.
A	Advertise.

Table 1: Selected Commands

After configuring the RN4020 to advertise via serial, it is detectable by other Bluetooth enabled devices when scanning from the Bluetooth menu. A connection can be established through the phone menu and will connect to the module. A connection light on the module indicates when the RN4020 is connected to another device. Without a custom app however, there is no way to send or receive data.

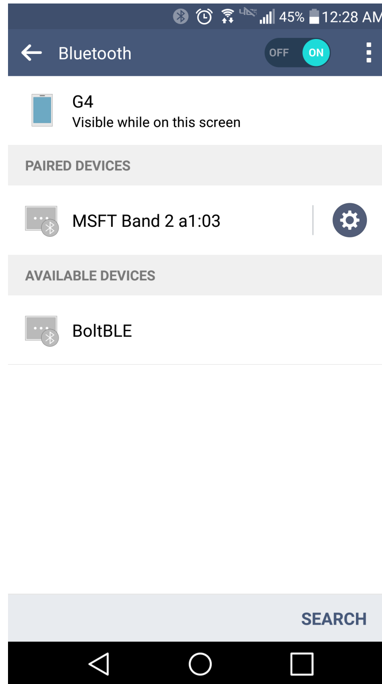


Figure 12: Bluetooth Detection on Android Device

6.4 Bluetooth Services and Characteristics

Bluetooth communication is based on the Attribute Protocol (ATT) which assigns attributes based on device roll. A module can be a client, a server, or both. For the our project, the RN4020 will serve as a server and the Android application serves as both a client and a server. To organize attributes, each attribute is assigned a universally unique identifier (UUID) which can be used to access the attribute value. The Generic Attribute Profile (GATT) builds upon the ATT to add hierarchy and data abstraction. GATT is the basis for BLE data transfer because of the unification of data organization and exchange. GATT mandates that data is encapsulated in a service which consists of one or more characteristics. Each characteristic in a service contains a shortened UUID handle, a type, permissions, a value and a value length.

Bluetooth SIG has created a set of existing services with characteristics which can be adapted or used in different application. The RN4020 supports 17 public services registered with Bluetooth SIG. In order to facilitate simple data transfer, two services were selected: the battery service and the health thermometer service. The services were set on the RN4020 using the SS command. The battery service is know by SIG UUID 0x180F and has one characteristic, the battery level, known by the handle 0x2A19. The battery level characteristic is a mandatory uint8 with a range between 0 and 100. The battery service was selected because of its simplicity and its ranged input value. The health thermometer is marked by SIG UUID 0x1809 and has two characteristics: temperature type (0x2A1D) and temperature measurement (0x2A1C). The temperature type is a mandatory 1 byte flag value corresponding to descriptions of different locations of the sensor. For the purposes of Bolt, this field is ignored. The temperature measurement is also an 5 byte mandatory value which has a flag and a variable length data stream (assuming the data fits within 5 bytes). The health thermometer service was selected because of its flexibility with the variable length floating point data value.

In order to send commands to the RN4020, a microcontroller communicates via UART to send ASCII commands. A physical button and a potentiometer were used to produce data values to send with services via bluetooth. They system setup between the Bluetooth module and the microcontroller is shown in Figure 13. The microcontroller is responsible for processing the button press and then changing the value of the battery level characteristic to indicate the button press. Pressing down the button will send the command SUW,21A9,32 which writes the value 0x32 (or 50) in the 0x2A19 characteristic. A release of the button will

send a similar command SUW,21A9,64 or 0x64 (100) to the battery level characteristic. The microcontroller acts as a data transmitter and thus the serial communication between the two values is simple.

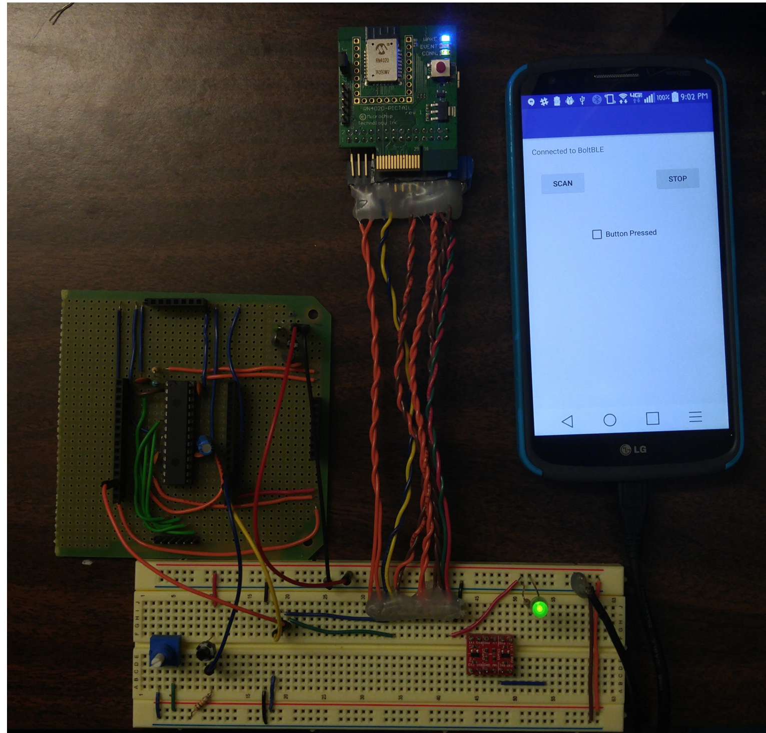


Figure 13: The full Bluetooth system

6.5 Android App Development

To receive and process the Bluetooth data, an Android application in Java was created using Android Studio. Bluetooth functionality was added using the Bluetooth Application Accelerator class. The Bluetooth Application Accelerator class provides a wrapper and functions for the development of Bluetooth applications. The Bluetooth services run on one thread of the program. Additionally, a second thread of the program is used to update the user interface. Without multi-threading, the display freezes because bluetooh processes running in the background constantly seek data and lock the program flow to update the display. After the user selects a button to begin the scan, the application automatically scans and connects to the RN4020 module by seeking the known device name (BoltBLE). After connecting, if the physical button is pressed, a check box will light up and text will display to indicate the button status. Screen shots of the application are displayed in Figure 14. The button press is achieved by setting a notification for each service present on the RN4020. A snapshot from the debugger can be seen in Figure 15 showing the enumeration of the services. The characteristic of each service is read in, via UUID, and the value is checked for the desired value. In the case of the button, a button press is indicated by a 0x32 and a button not pressed is indicated by a 0x64.

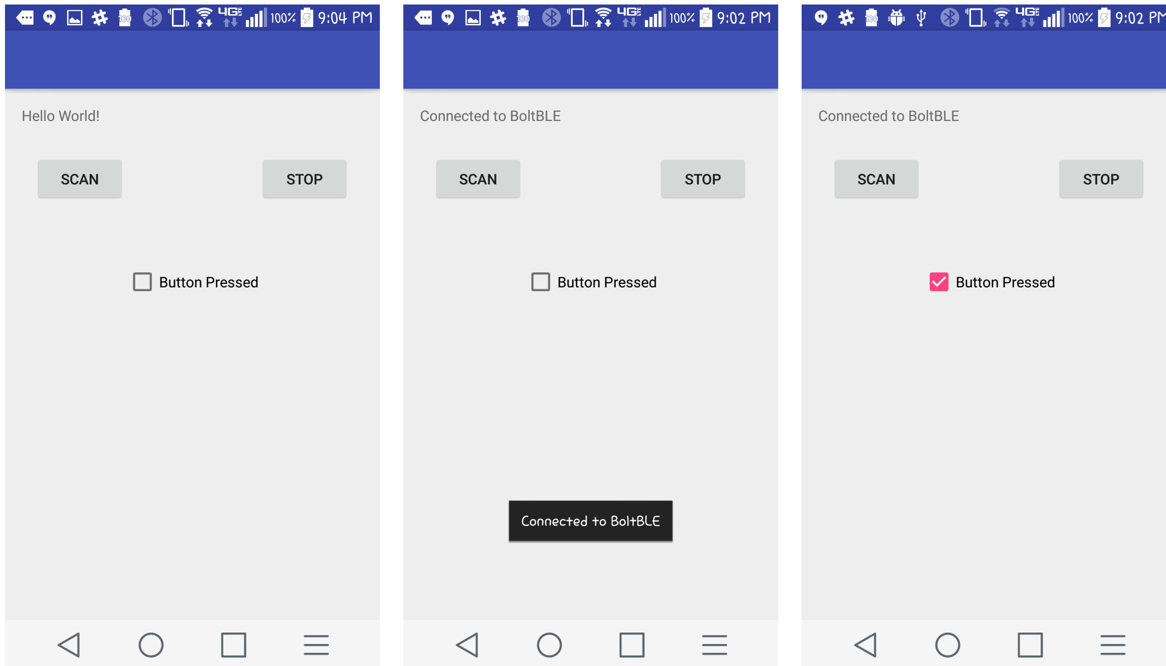


Figure 14: App screenshots

```
02-25 22:18:31.015 2256-2273/com.example.abartels.bluetoothtest D/BluetoothLeScanner: onScanResult() - ScanResult{mDevice=FB:6E:79:7A:8A:D4, mScanRecord=ScanRecord [mAdvertiseFlags=
02-25 22:18:31.016 2256-2256/com.example.abartels.bluetoothtest D/DEBUG: uiDeviceFound: uiDeviceFound: Adafruit Bluefruit LE, -82
02-25 22:18:31.172 2256-2806/com.example.abartels.bluetoothtest D/BluetoothGatt: onSearchComplete() = Device=00:1E:C0:1D:3E:6E Status=0
02-25 22:18:31.182 2256-2806/com.example.abartels.bluetoothtest D/DEBUG: Service: Generic Access[00001800-0000-1000-8000-00805f9b34fb]
02-25 22:18:31.182 2256-2806/com.example.abartels.bluetoothtest D/DEBUG: Service: Generic Attribute[00001801-0000-1000-8000-00805f9b34fb]
02-25 22:18:31.182 2256-2806/com.example.abartels.bluetoothtest D/DEBUG: Service: Device Information[0000180a-0000-1000-8000-00805f9b34fb]
02-25 22:18:31.182 2256-2806/com.example.abartels.bluetoothtest D/DEBUG: Service: Health Thermometer[00001809-0000-1000-8000-00805f9b34fb]
02-25 22:18:31.182 2256-2806/com.example.abartels.bluetoothtest D/DEBUG: Service: Battery Service[0000180f-0000-1000-8000-00805f9b34fb]
02-25 22:18:31.182 2256-2806/com.example.abartels.bluetoothtest D/DEBUG: If happened
02-25 22:18:31.182 2256-2806/com.example.abartels.bluetoothtest D/BluetoothGatt: setCharacteristicNotification() - uuid: 00002a19-0000-1000-8000-00805f9b34fb enable: true
02-25 22:18:31.186 2256-2806/com.example.abartels.bluetoothtest D/DEBUG: for
```

Figure 15: Debugger Output

7 Microcontroller Network

7.1 Dashboard

On the motorbike, a dashboard serves as the main point for user interaction. Featuring an LCD display, buttons, and LED indicators, the dashboard is a critical component to connect the user to the motorbike. One microcontroller is used to generate the display for the LCD screen and keep track of the state of the dash based on user input. Additionally, the parts we deliver to the Bolt company must fit within a specified size footprint for the dashboard.

7.1.1 Input/Output

The table in Figure 16 lists all the inputs and outputs that control the state machine and the hardware peripherals.

Inputs	Outputs
Bluetooth_RX	Bluetooth_Wake
CAN_RX	Bluetooth_Enable
Light_Sensor	Bluetooth_TX
Speed_Sensor	CAN_TX
Kickstand	CAN_SDBY
Mode_Button	LCD_CLK
Select_Button	LCD_DCIN
Kill_Switch	LCD_CE
Ignition_Switch	LCD_RST
	LCD_DC
	LCD_BACKLIGHT
	LED_CLK
	LED_DIN
	LED_LAT
	LED_PWM

Figure 16: Dashboard Input/Ouput Table

7.1.2 State Machine

The dash handles the necessary outputs to the LCD screen as well as inputs such as the the safety kill switch, ignition, ambient light sensor, speed sensor, password, and button presses. It will tell the user the necessary information (through the LCD screen) depending on what state the dash. The state machine diagram is shown in Figure 17. A note to consider before reviewing the state machine is that the kill switch must be on in all states except Sleep and Init since the user could need to cut the power at any point in an emergency. If the kill switch is turned off at any point, the state will immediately switch to the Sleep state and all other systems will go idle.

Init State This state initializes the inputs/outputs for the microcontroller. It also initializes the necessary software modules for microcontroller setup. The motorbike will enter this state immediately once there is power for the bike, meaning once the batteries are inserted. When initialization processes are complete, the state immediately moves into the Sleep state.

Sleep State The Sleep state sits and waits for someone to "turn" on the bike by waiting for an ignition press. At this point, the motorbike appears off. Once the ignition is pressed, state will transition into the Welcome state.

Welcome State The Welcome state will simply give the user a welcome message for a certain period of time and then move onto the Locked state.

Locked State The Locked state asks the user to put in the password. If the user puts in the wrong password or takes too long the bike will return to the Sleep state. If the password is correct it will move into the Safe Mode state.

Safe Mode State This state will allow the user to change different throttle modes such as sport, economy, and demo. It also allows the user to look at information on the bike such as lights, battery power level, and last trip distance, among other things. Once the user has decided on your throttle mode and held the ignition switch, a message will be sent to the main control unit (MCU) about what throttle mode you choose and turn on the bike to start running (which will also change the state to Running).

Running State The Running state displays speed, lights, brake, trip distance, etc. while the user is riding on the bike. It will also communicate to the MCU on how fast the bike is going so the bike does not go too fast in a specific throttle mode. If the ignition is held while in this state it will go back to the Safe Mode state where the user can change ride settings.

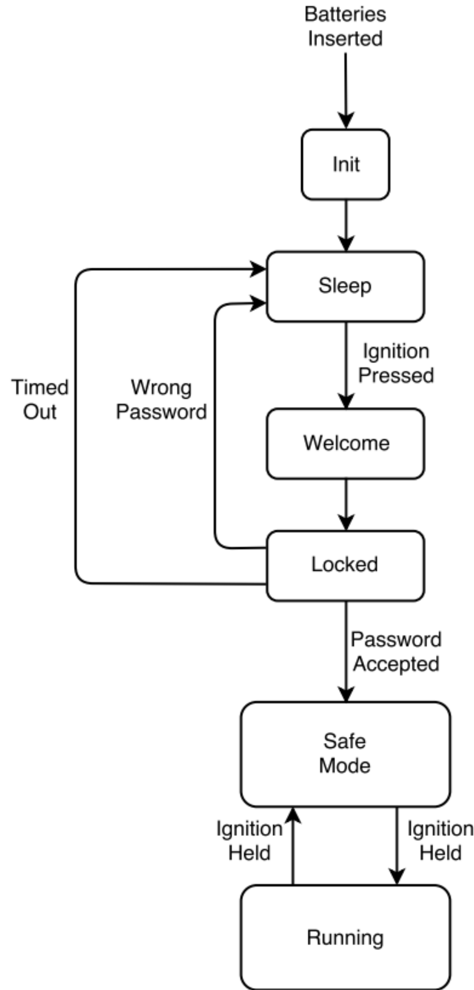
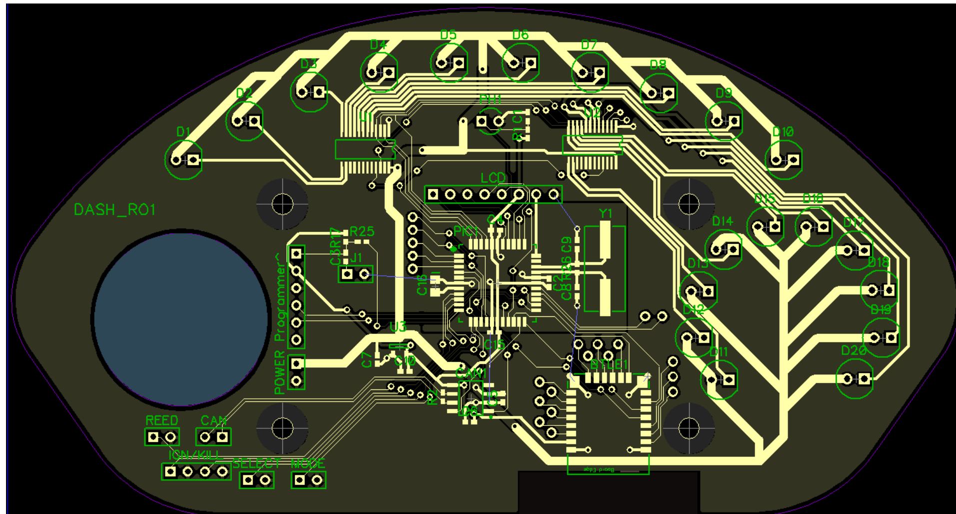


Figure 17: Dashboard State Machine

7.1.3 Dash Circuitry/PCB

The PCB was designed with 5V and 3.3V power rails, and a single ground plane. The area directly under the microcontroller has it's own local ground plane connected at only one point, and is fully decoupled using ceramic capacitors at each V_{dd} input. The crystal oscillator is also within this local plane. All power and ground traces are at least 18 mils wide, and most signal traces are 6 mils. The board also has pads for every unused pin on the microcontroller and bluetooth module in case we want to add or test extra functionality. The board also has multiple cut-outs to accommodate the bluetooth antennae and other hardware. Figure 18 shows the digital PCB footprint for the dashboard.



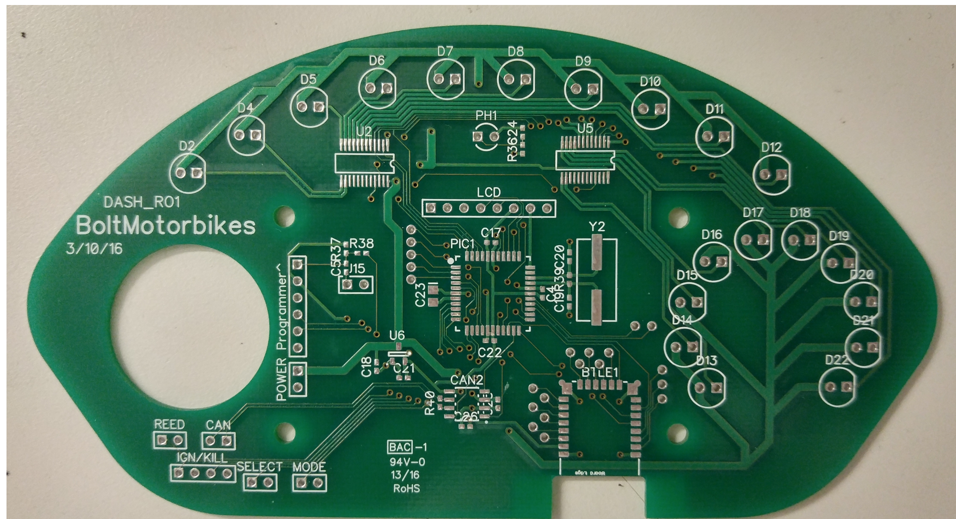


Figure 18: Dashboard PCB Digital footprint compared its physical cutout

7.2 Main Control Unit

The main control unit or MCU of the motorbike is in charge of controlling the overall state of the bike for all systems besides the dashboard. This includes driving actuators for user input such as the throttle, brake or horn. Additionally, the MCU handles more behind the scenes necessities like controlling the CANbus communication protocol. The dashboard requires a PCB to fit within a specified area on the Bolt motorbike.

7.2.1 Input/Output

The table in Figure 19 shows all the inputs and outputs that control the state machine and the hardware peripherals.

Inputs	Outputs
Throttle	Throttle_PWM
Brake	Lights
MotorVoltage	12V_Enable
BatteryVoltage	BrakeLight
MotorCurrent	CAN_TX
CAN_RX	CAN_STBY
HighBeam	MotorController_Enable
Horn	Contactor_Enable

Figure 19: Main Control Unit Input/Ouput Table

7.2.2 StateMachine

The state machine is driven by events that come from both sensor inputs and messages sent from the dash. The main control unit is tested using the Bike on a Board, since inputs from sensors on Bike on a board are collected and then outputs are sent via UART the the LCD display or computer. The logic for the state machine can be seen in Figure 20

The state machine begins by simply waiting to receive a ready signal from the CANbus. Since we are still using UART as a placeholder for CAN, this is simply an input that we currently trip high with a wire to begin state machine operation. From here, the MCU sits in an idle state until receiving a message from the dash to tell it that the ignition button has been pressed for the required amount of time. The MCU then waits for a message, which should be sent from the dash immediately after the ignition message, that tells the MCU what drive mode to enter. Demo mode is meant as a demonstration mode where motor power is limited to extremely low speeds for safety. Econ mode enforces charge efficiency by keeping the motor speed at a slower but more constant level. Sport mode allows the user to bring the bike to a greater speed. For each of these modes, we will experimentally develop mappings from the throttle input ADC to the motor output PWM to achieve the appropriate ratios between throttle and speed for each mode. Each drive mode has its own associated break state, so that the state machine knows what drive state to return to when the brake is released. If at any time in any of these drive or brake states, the MCU receives a message from the dash the the ignition button has been pressed and held again, the MCU will kill the motor and return to the idle state.

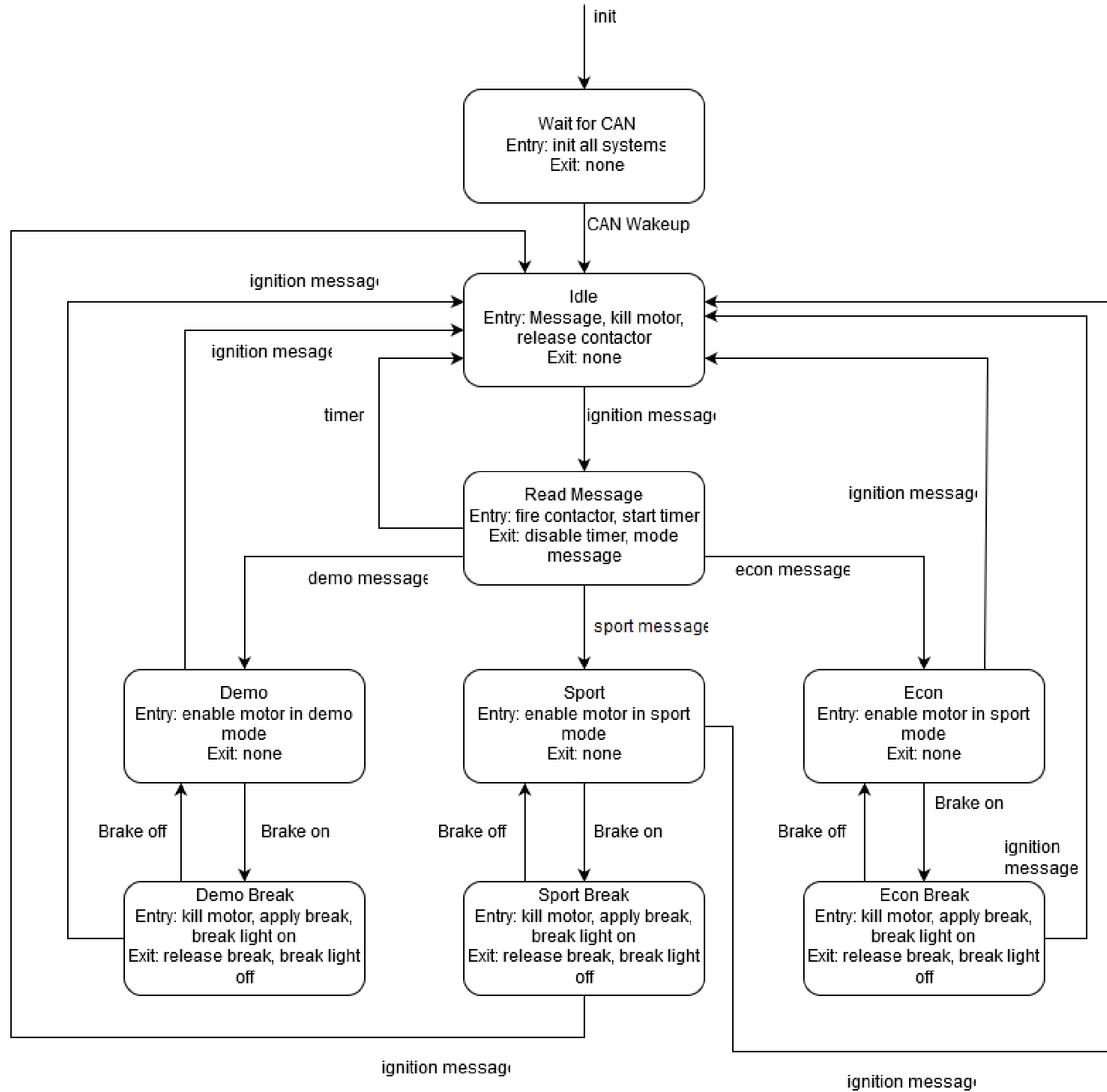


Figure 20: MCU State Machine Diagram

7.2.3 Main Control Unit Circuitry/PCB

The MCU and Power Distribution PCB is based on the prototype PCB created using the M60 CNC process. Even though this design is technically complete, it needs to be re-designed. However, it will serve its purpose and be ordered along with our first article Dash PCB.

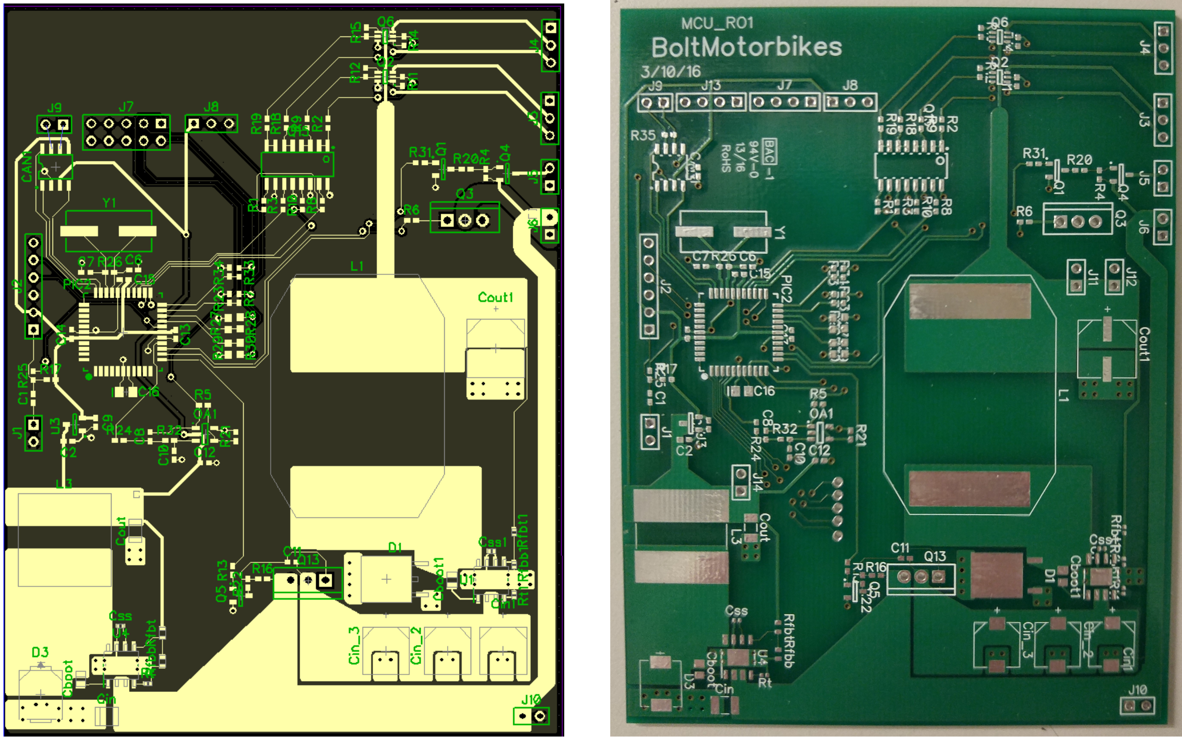


Figure 21: Power Distribution and MCU digital and physical PCB designs

8 Software

8.1 Peripheral Modules

In order to interface with hardware devices, software has to be written in the microcontroller for each peripheral device. All code was written in C using MISRA-C standards. MISRA-C is a guideline for the C language for mission critical systems and is a standard in the automotive industry. Describing every peripheral on our project would prove to be redundant, so highlighted below are the modules which proved most troubling to develop. The oscillator is needed to control the clock for the system which is necessary to keep data transmission constant and at the same speed. The Analog to Digital (ADC) module is needed to read sensor voltages and other non-digital values. The LCD screen, which is a part of the dashboard, is needed to display messages and information to the user.

8.1.1 Oscillator

The oscillators main function is to serve as a clock; each microcontroller is fitted with an external crystal oscillator. Since we are using asynchronous protocols like UART, SPI, and CANbus, it is important that the microcontrollers can sync together with the same time to communicate properly.

Oscillator Circuit

The oscillator circuit was designed using the manufacturers guidelines for finding the stray capacitance, Equation 1,

$$\frac{C1 * C2}{C1 + C2} + C_{stray} = C_{load} \tag{1}$$

where $C1 = C2, C_{stray} = 15pF$ (from microchip data sheet) and $C_{load} = 20pF$ (from crystal data sheet), we find that C1 and C2 are 10pF, and the closest standard value is 10pF.

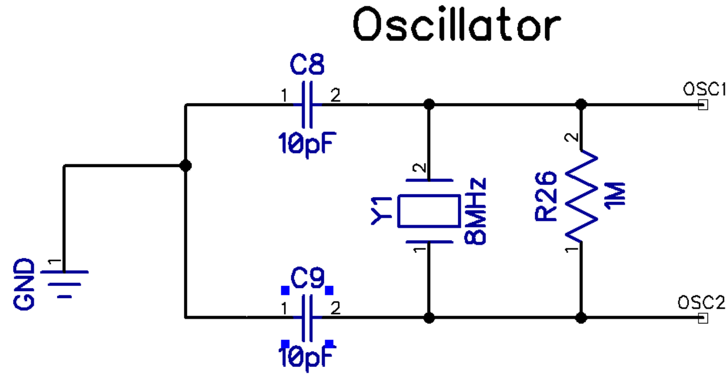


Figure 22: 8.000MHz Oscillator circuit

Oscillator Software Module A module called "freq.h" is designed to set the system clock using either the internal or external oscillator for frequencies between 32kHz and 70MHz. This is needed so that the Dash and the MCU can both enter sleep mode, a low powered state. There are other modules that depend on freq.h, for example, the UART module needs to know the clock speed so that communications can be maintained even during a low-powered state. SPI, and CANbus also depend on the frequency so it is important that switches between clock sources are done correctly. High frequencies require the clock to enter a phase-locked-loop, which can take some time to stabilize. The function does not complete the switch until the phase-locked-loop is enabled. The user may call the function clockInit with the arguments of clock speed and clock source. All of our boards will have an external crystal oscillator at 8MHz, but the internal oscillator is not as accurate and oscillates at about 7.37MHz. At any time, the user may call the function clockSpeed to return the speed of the clock.

8.1.2 Analog to Digital

The Analog to Digital (ADC) module has eight different control registers which need certain bits to be manipulated in code. The control registers are AD1CON1, AD1CON2, AD1CON3, AD1CON4, AD1CHS123, AD1CHS0, AD1CSSH, and AD1CSSL. How we made ours is that it will initialize AN0, AN1, and AN2 it be analog input pins on our microcontroller. Some of the settings we decided to choose where to make the inputs return a 10-bit max integer form, use external Vdd and Vss, fill the buffer from the start address, among many other things. Basically this function sets AN0, AN1, and AN2 to be analog input pins and return an integer with a max value of 1024 since it is a 10-bit integer. This means that for a 0V input we get return value of 0, 1.65V a return value of 512, and 3.3V a return value of 1024.

8.1.3 LCD Screen

The LCD screen Bolt uses is the Nokia 5110. The screen requires the user to either write commands or data. Commands set certain configurations of the screen such as contrast, temperature coefficients, etc. It also requires the Nokia's 5110 DC value to be low (0). When data is written, it turns on or off certain pixels on the screen. This requires the DC value to be high (1). They way we write to the screen is using the SPI module. Basically the SPI will write hex values to the screen using non-blocking code.

LCDInit

This function initializes the LCD and the SPI at the same time so they will work together. The pins we use for the LCD are RB15 for the reset pin, RB14 for the count enable pin, RB13 for the DC pin, and RB12 for the backlight pin. The SPI uses pins RB7 for its clock and RB8 for the data out. To get the Nokia 5110 to start you need to write a high and low signal within 300ms of each other coming out from the reset pin. Once

this is done you have to write certain commands to configure the screen. Using the LCDWriteCmd function (write command to the screen) we are able to write an array of hex values that act as certain commands. The array is:

commandBuffer[] = { 0x21, 0xBF, 0x04, 0x10, 0x20, 0x0C }

The following table will explain what each hex value does.

Hex Value	Description
0x21	The LCD will be set in extended commands mode.
0xBF	This sets the LCD Vop contrast values for the screen. This hex value can be played with until an appropriate contrast has been found.
0x04	This sets the temperature coefficient of the screen.
0x10	This sets the bias system to a mix rate of 1:100.
0x20	The LCD will now be set to basic commands. (i.e. it can only change the cursor of the screen/where the data will start writing to on the screen).
0x0C	This sets the screen to be in normal mode. The other modes turn all segments on the screen on or off. Normal mode will allow the user to manipulate the screen however they choose.

Once these commands have been written the Nokia 5110 screen will be ready to write data to the screen.

LCD Functions

The important functions in the LCD_SPI module are the shown next, with the exception of LCDWriteCmd and LCDWriteData. These two function are essentially the same thing but the DC value is low for commands and high for data. Everything that is written through these functions also go through a Queue which is explained in the Queue section. Essentially though, these functions will tell the screen a command or write data to the screen.

LCD Function:	Description:
LCDCursor(uint8_t xCursor, uint8_t yCursor)	This function will write to a certain spot on the screen. The xCursor can go from 0 - 84 and the yCursor can be 0 - 6.
LCDWriteString(char characters)	This takes each character from a string array and writes each one to the screen. To do this it references an ASCII table to write the specific letter to the screen.
LCDWriteBitmap(void)	This will take a hex array and write it to the screen which creates a picture.

Queue

The Queue allows the LCD_SPI to be non-blocking and do other processes while the SPI is writing to the LCD screen. When a LCDWriteCmd or LCDWriteData function is called it stores the information passed into those functions, into the Queue. Once the data from the SPI has finished writing it looks for the next item in the Queue to be written if the Queue is not empty. The Queue is tracked using a currentIndex and an emptyIndex.

The important functions for the Queue are shown in the following:

Queue Function:	Description:
addToQueue(uint8_t data, uint16_t length, uint8_t DC)	This function adds the data, length, and DC values into arrays in the Queue. It will then increment the emptyIndex to the next slot in the Queue.
deleteFromQueue(void)	This will delete the items at the currentIndex of the Queue and then increment the currentIndex.
resetEmptyIndexQueue(void)	If the emptyIndex is at the size of the Queue and needs to increment, it is set to the beginning of the Queue (0).
resetCurrentIndexQueue(void)	If the currentIndex is at the size of the Queue and needs to increment, it is set to the beginning of the Queue (0).
queueWrite(uint16_t theIndex)	This set the DC value to be high or low depending on if it is a command or data write. It then writes the CE pin low to the LCD and lastly writes the data from theIndex from the array stored in the Queue.
getQueueLength(void)	This returns the length value stored in the Queue at the currentIndex.
emptyIndexNext(void)	This checks to see if the emptyIndex is right before the currentIndex. If it is right before the currentIndex then the Queue is essentially full.

How the Queue works with the LCD_SPI (example)

CODE:

```
LCDWriteCmd((uint8_t)commandArray, 6); //Step One
LCDWriteData((uint8_t)dataArrayOne, 4); //Step Two
LCDWriteData((uint8_t)dataArrayTwo, 3); //Step Three
```

QUEUE:

Queue			
Data	Length	DC	Indexes
commandArray	6	0	currentIndex
dataArrayOne	4	1	
dataArrayTwo	3	1	
			emptyIndex

As this point the Queue will begin writing what is in the commandArray through the SPI to the LCD screen. Once commandArray has been written the currentIndex will increment and we will move onto the next indice to do an SPI write.

Queue			
Data	Length	DC	Indexes
dataArrayOne	4	1	currentIndex
dataArrayTwo	3	1	
			emptyIndex

The Queue will continue to write everything in the Queue until it has reached emptyIndex. So if we do two more SPI writes with dataArrayOne and dataArrayTwo we will get the Queue in the following position:

Queue			
Data	Length	DC	Indexes
			currentIndex emptyIndex

Once currentIndex is at the same spot as emptyIndex we know that we have nothing more in the Queue. If the Queue becomes full for some reason it will simply not add anything into the Queue until an open spot becomes available. The Queue becomes full if emptyIndex = currentIndex - 1. This means there should always be an empty slot in the Queue so that the emptyIndex will not overlap the currentIndex.

8.2 Framework

To facilitate intuitive, readable state machine code, we developed an events framework that made it simple to incorporate event checking functions into the state machine code. The framework operates by posting service functions to a queue to be run by the processor. The framework constantly loops through checking this queue for functions to be run, and when there are no functions to be run, the framework runs the event checkers in order to check for more event that will call more services. These services are the initialization and run functions for the state machines and sub-state machines that run our various software modules. The framework includes a configuration header file that makes it simple for the programmer to define the various functions, event names, state names, and lists of event checking functions necessary to run a state machine through the framework.

9 Conclusion

The Bolt Team at UCSC met all deliverable goals set at the beginning of the quarter. We were able to create a Bike on a Board test bench and start development for nearly every component of our system, including battery management, power distribution, system framework and Bluetooth Low Energy integration. With over 1000 aggregate hours put in by all team members over the quarter, the Bolt team has consistently met and exceeded weekly goals. Figure 23 displays a graph of hours worked by each member over the course of Winter Quarter. Next quarter we plan to focus on CANbus development, system integration, and user experience in order to deliver the newly designed Bolt Motorbike.

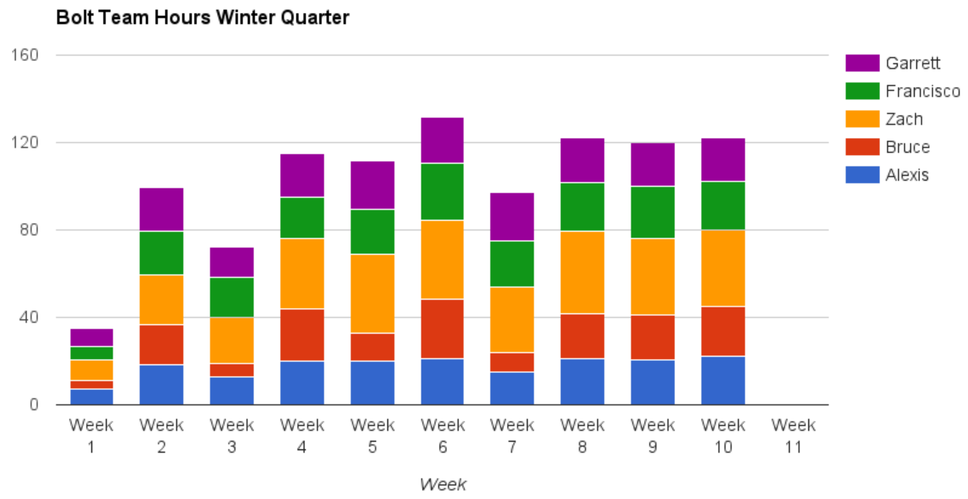


Figure 23: Bolt Team Total Hours for Winter Quarter