Spring 2016

# Motorcycle Helmet Crash Detection/Prevention System

David Witsaman
dmw51@zips.uakron.edu

Motorcycle Helmet Crash Detection/Prevention System

David Witsaman

Department of Computer Engineering

**Honors Research Project**

Submitted to

*The Honors College*

Approved:

Bahrami Date 5/6/16
Honors Project Sponsor (signed)

HAMID BAHRAMI
Honors Project Sponsor (printed)

_____ Date 5/10/16
Reader (signed)

_____
Reader (printed)

Robb J Veillette Date 16 May 2016
Reader (signed)

ROBERT J. VEILLETTE
Reader (printed)

Accepted:

Joan E. Carletta Date 5/16/2016
Department Head (signed)

Joan E. Carletta
Department Head (printed)

Robb J Veillette Date 16 May 2016
Honors Faculty Advisor (signed)

ROBERT J. VEILLETTE
Honors Faculty Advisor (printed)

_____ Date _____
Dean, Honors College

Originally, my responsibilities with this project were to include the development of the mobile application that runs on the user's cell phone. This application would allow for two way communication between the cell phone and microprocessor. The application would send speed and turn-by-turn directions to the microprocessor and the microprocessor would notify the application if a crash had been detected. Due to my very limited experience with writing Android applications, I traded this responsibility with one of my teammates for his responsibility in writing a majority of the Arduino code.

In addition to writing a majority of the final code running on each of the three TinyDuino microprocessors, one of my big responsibilities was to get all three of these subsystems to work together. This involved countless hours in the lab writing and re-writing the code while using all of the senior design lab's hook clip test cables to connect all of the subsystems together (I believe at one point I was using 28 of them).

Another one of my original responsibilities was to write the Arduino code that allowed information from the cell phone application to be displayed on the heads up display. As this closely tied in with the development of the mobile application, this was also passed off to another teammate so that we could better segment our work.

I was also responsible for providing the helmet that would eventually become our prototype. As I had an old motorcycle helmet that had previously taken an impact and needed to be disposed of, this was an easy task.

Lastly, since I was the 'Archivist' for the group, I was tasked with keeping track of the group's progress via a website so that our Faculty Advisor could easily check in on how our project was going. I was also in charge of submitting all of our assignments and deliverables on time to the senior design instructor.

# Motorcycle Helmet Crash Detection/Prevention System

## Final Design Report

Design Team H

Joseph Douglas

Brandon Flint

Wesley Waters

David Witsaman

Dr. Hamid Bahrami, Faculty Advisor

April 22nd, 2016

**Table of Contents**

**List of Figures**

**List of Tables**

**Abstract**

This project proposes a motorcycle safety system that increases safety by actively helping to prevent crashes while also helping in the case that an accident does occur. The system actively helps prevent accidents by keeping the user's eyes on the road with three additions to the typical helmet. The helmet has a heads-up-display (HUD) containing the speed of the motorcycle and turn-by-turn directions. Instead of tilting their head down the user can see their speed and directions by moving their eyes which will keep the road in their field of view. Blind-spot detection increases the user's overall awareness of their surroundings. The helmet increases the speed of response when an emergency situation occurs. After a crash has been detected, emergency response will be notified via call and text from the user's phone. In order to increase attention to the accident, external LEDs flash and an external speaker sounds. Keeping the driver's eyes on the road makes for a safer driving experience which decreases the number of crashes. Having emergency response arrive quicker helps to improve the chances of a speedy recovery or even could save a person's life.

- Crash detection
- Contacting emergency response in the event of a crash
- LEDs will flash in the event of a crash
- Speaker will sound in event of a crash
- Blind-spot detection
- Turn-by-turn directions displayed on HUD
- Speed displayed on HUD

[JD,BF]

# 1. Problem Statement

## 1.1 Need

Motorcyclists are at a higher risk of being involved in a life-threatening motor vehicle accident. Motorcycles do not have many safety features in order to protect the operator during a crash. The operators tend to wear leather clothing and a helmet as their only safety precautions. Being thrown from a motorcycle at any speed with these precautions can result in life-threatening injuries. Motorcyclists are less likely to be seen by others after an accident. Increasing the safety for motorcyclists through accident prevention, as well as accident detection, could decrease motorcyclist injuries/fatalities.

Preventing motorcyclists from having an accident is the first step to increasing the safety of the driver. In some new cars they have blind-spot detection to prevent accidents from occurring. Implementing this safety feature for motorcyclists would help to increase the driver's awareness and therefore increase their safety. Car drivers have the convenience of having driver-assistance tools in safer locations. Motorcyclists have to take their eyes off the road more to be able to use tools like turn-by-turn directions and their speedometer.

Accidents are inevitable and motorcyclists are less visible than cars. When a motorcycle accident occurs, that results in an incapacitated motorcyclist, they may not be seen quickly enough to call emergency response in a timely manner. Automatically calling emergency response is beneficial. Making the accident more obvious to other drivers would help the motorcyclist's safety.

[JD,BF]

## 1.2 Objective

Increasing the safety of motorcyclists is attainable through preventing crashes, detecting crashes, and notifying emergency services. In order to help prevent crashes, safety features can be added that help maintain the driver's focus on the road. When motorcyclists have accidents, having a timely emergency response could mean the difference between life and death.

The helmet crash detection/prevention system will help prevent crashes from occurring through blind-spot detection and a better display of pertinent information. Adding blind-spot detection that does not require the driver to take their eyes off the road would increase the likelihood of a safe drive. For a motorcyclists to check their speedometer, they may have to take their eyes off the road. Likewise, adding this information in a more visible location, that can keep the driver more focused on the road, would result in a safer driving experience. Drivers often need directions to the location that they are traveling. It is not realistic for a motorcyclist to look at turn-by-turn directions on their phone and still drive safely. Adding turn-by-turn directions to the safety helmet is another feature for preventing accidents.

The helmet crash detection/prevention system will increase the response speed. It will detect the crash after the crash has been detected, a call will automatically be made, alerting emergency response of the situation with an automated message. After a crash has been detected, a text conveying there has been a crash will be sent to emergency response. The text

will also include the GPS coordinates from the user's phone. After the crash, the system will emit lights and sound to increase the likelihood that the driver will be located quickly.

[JD,BF]

## 1.3 Background (Research Survey)

Upon detecting a crash, the system will send a signal to the phone letting it know that a crash has occurred. Once this happens the phone will call 911 where they will be able to track the GPS location even if the GPS location is turned off. The helmet will also help anyone in the vicinity find the motorcyclist by flashing lights and sounding an alarm.

The system will also help to prevent crashes. It will have LED's inside of the helmet on both sides of the driver's face. The lights will alert the driver if a car is entering their lane or if they are merging into a lane with a vehicle already in it. It will also have a HUD that gives the driver information that normally they would have to take their eyes off of the road to obtain such as speed and GPS information. The speed will be obtained from the GPS located on the phone. The GPS data will be sent over from the phone. The settings for the HUD will be set through an app on the phone as well as having an on/off button on the helmet.

There are products on the market that have some of these features individually, but none that have all of them, at least none that are a reasonable price. A couple of examples include the Skully AR-1 helmet and the ICEdot crash sensor. The Skully AR-1 has GPS navigation, a blind spot camera, and a HUD. This helmet is not yet available, but is taking pre-orders for $1499. The ICEdot crash sensor is a mountable device that is attached to a helmet (more so a bicycle helmet) that alerts your emergency contacts in the event of a crash. It notifies first responders of your name, medications, allergies, etc..

The Crash Prevention/Detection system combines the best of these features and improve on them. The Crash Prevention/Detection system includes blind spot detection, GPS directions, and a HUD like the Skully AR-1 has, but it also includes crash detection and emergency services notification like the ICEdot system has. Also, when the Crash Prevention/Detection system detects a crash, it will immediately notify emergency services with your location, improving response time, unlike the ICEdot which only alerts your emergency contacts.

[WW,DW]

## 1.4 Marketing Requirements

The helmet crash detection/prevention system must wirelessly communicate with the driver's phone in order to relay turn-by-turn directions, as well as relaying the phone's GPS speed to be used as a speedometer. The display must be located within the helmet in an easily visible region. The turn-by-turn directions are defined as an arrow being displayed indicating the next change in direction and distance to next change of direction.

The system must detect if a vehicle is in the driver's blind-spot and indicate to the driver of the vehicle. The information should be relayed using a light on either side of the user's peripherals.

The system must detect impact to the helmet that may be deemed as a crash. The helmet will give the user a grace period in order to prevent a call to emergency response. If the grace period expires, the helmet will communicate to the driver's phone to contact emergency response with the GPS coordinates of the phone.

In the event the system detects a crash, lights and sound will emit from the helmet in order to draw attention to the driver.

The system must sustain power for at least six hours on a full charge. The system should be waterproof. The system will work with Android 4.1 and newer phones. The weight should not increase by more than 10% of helmet weight.

[JD,BF]

## 1.5 Objective Tree

Figure 1, the Objective Tree, is shown below.

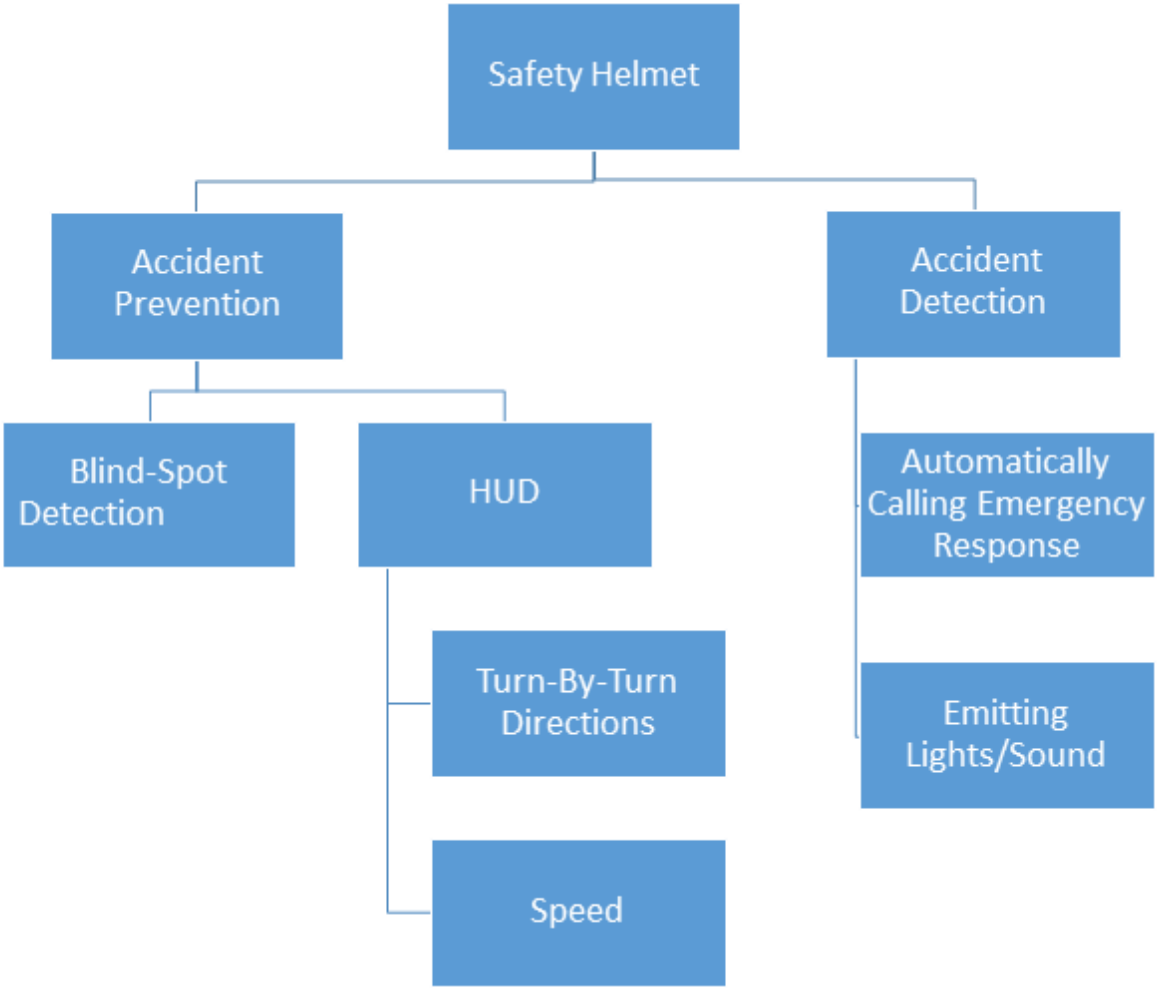



Figure 1: Objective Tree [JD,BF]

## 2. Design Requirements Specification

### 2.1 The Requirements

The marketing requirements for this project are as follows:

1. The system must detect impact classifiable as a crash.
2. On detection of a crash, the user will be afforded a thirty second grace period in which they can terminate further action by pressing the on/off button, which will power off the system before further action is taken.
3. During the thirty second grace period the driver will be notified.
4. On detection of a crash, after the grace period, the system will communicate with the user's phone in order to initiate a communication to emergency response conveying a crash has occurred, and providing the GPS coordinates of the user's phone to emergency response.
5. On detection of a crash, after the grace period, the system will emit lights and sounds separate from the grace period sounds.
6. The system must detect a vehicle in the user's "blind spot" and communicate through the microprocessor, emitting a light on the side of the user in which the vehicle is located.
7. The system's microprocessor will communicate to the HUD the speed that the user's phone is traveling, based upon the phone's GPS movement.
8. The system's microprocessor will communicate with the user's phone and will relay turn-by-turn directions supplied by an application from the user's phone to the HUD, showing an arrow to next change in direction as well as distance to the change.
9. The system should sustain power for at least six hours on a full charge.
10. The system should be usable on Android 4.1 and newer phones.
11. The system will be waterproof.
12. Weight should not increase by more than 10% of helmet weight.

[JD,BF,WW,DW]

**Table 1: Engineering Requirements with Justification**

| Marketing Requirements | Engineering Requirements | Validation Rationale |
|---|---|---|
| 1 | Use multiple impact sensors to detect when the impact is above a threshold that can be described as a crash. The impact sensor does not need to be accurate at small impacts. For redundancy the accelerometer from the user's phone will also detect a crash. | The impact sensor needs to detect when a crash has occurred. A crash is defined by a high impact. The impact sensor does not need to have precise readings at low impact. In a case of a substantial impact not near enough to the impact sensor to characterize it as a crash, the phone's accelerometer can also determine a crash. |
| 2 | Utilize the on/off button, which powers off the system. | It is not necessary to contact emergency response if the user feels it is not required. On detection of a crash the user will be afforded the opportunity to cancel further action from the system. The on/off button is an easy way to receive user feedback. |
| 3 | A HUD will display a countdown, and blind-spot LEDs will flash | This HUD countdown and flashing LEDs let the user know that a crash has been detected by the system and the system will contact emergency response when the countdown is complete. |
| 1,4 | The microprocessor will communicate using Bluetooth with the phone's application, which will initiate a call and text to emergency. | The system must contact emergency response with the crash coordinates. The text message ensures emergency response receives the information. If the user is able to speak, the call allows the user to give further details to responders. Bluetooth is a convenient way to communicate wirelessly. |
| 3,5 | The system will use LED's and an external speaker. The LED's | The system must draw attention to any nearby people. The system should allow |

| | | |
|---|---|---|
| | should be sufficiently bright to draw attention. The speaker should be sufficiently loud to draw attention. | emergency response to locate the user quickly. LED's are efficient which will conserve power, and they are bright. Speakers are necessary to draw attention in the daylight. |
| 6 | The system will utilize an ultrasonic distance sensor which will communicate with the microprocessor. The microprocessor will turn an LED on, inside the helmet. | The ultrasonic distance sensors are accurate enough for inexpensive units. On detection of an object within the allotted range, an LED on the respective side will turn on. |
| 7 | The phone's application will utilize the phone's GPS in order to calculate the user's speed. The application will relay the speed to the microprocessor, which will display the speed on the HUD. | The phone has GPS as well as the computing power readily available to transmit. The user will have a more convenient location for verifying speed. |
| 7, 8 | The Google Maps API will be accessed in order to send the directions and distance to the next change in direction to the microprocessor. | The Google Maps API is readily available and contains the turn-by-turn directions needed to be sent to the HUD. |
| 9 | Utilizing a Li-Ion battery the system will sustain itself for at least six hours. | Li-Ion batteries are a good solution for a light-weight, low cost, and high capacity. The discharge capabilities will be sufficient to power the components. |
| 10 | Android version 4.1 and greater share common libraries. | Over 92% of Android phones are running Android 4.1+, so rather than increasing research cost, we can appeal to this growing percentage of users. |
| 11 | | The internal components are protected |

| | | |
|---|---|---|
| | The external components will be waterproof, and the inside components are already protected. | from the elements. There are external components that are waterproof. |
| 12 | A Li-Ion battery will be used and the HUD will be small. | Li-Ion is a lightweight, high capacity battery. The HUD also needs to be lightweight. |

[JD,BF]

## 2.2 Constraints

### 2.2.1 Economic

The prototype should cost less than $400.

### 2.2.2 Environmental

The system should be able to communicate with any Android 4.1 or newer phone.

### 2.2.3 Sustainability

The system must be able to operate for six hours. The system should maintain its integrity under normal use.

### 2.2.4 Manufacturability

The system must be modular for different levels of prevention and/or detection.

### 2.2.5 Social

An FCC qualified Bluetooth module will be used within the system.

[JD,BF,WW,DW]

## 2.3 Standards

### 2.3.1 Testing

The system must meet all marketing requirements.

### 2.3.2 Communications

The system should be user-friendly which should only require the user to power the system on, open the application on their phone and turn the phone's Bluetooth on.

[JD,BF]

## 3. Design

Figure 2 shows the Level 0 Diagram of the Prevention/Detection System. The system will have five inputs power, impact sensor, ultrasonic range sensor, 9-axis IMU, and communication from the phone. The system will have four outputs HUD, a speaker, LEDs, and communication to the phone.


**Figure 2: Level 0 Diagram**

[JD,BF]

Figure 3 displays the Level 1 Diagram for this system. The battery supplies power to the HUD as well as the Microprocessor. The power for the Impact Sensors, Ultrasonic Range Sensors, Bluetooth Module, 9-Axis IMU, Blind-Spot LEDs, Crash Locating LEDs, and External Speaker will be from the Microprocessor. The Impact Sensors communicate with the Microprocessor. Then, the Microprocessor communicates to the HUD to countdown and to the blind-spot LEDs to begin flashing. After the countdown completes, the Microprocessor communicates with the phone through the Bluetooth Module. When the phone receives this message it will initiate a call and text to emergency response with the user's coordinates. Along with this, the Microprocessor powers the Crash Locating LEDs and External Speaker. The On/Off Switch can be used and this will turn off the system, preventing the system from external communication. The Ultrasonic Range Sensors communicate with the Microprocessor. When the Microprocessor reads a value indicating that a vehicle is in the user's blind-spot it checks the value of the IMU to make sure the reading is not coming from the sensor reading the road, if not it then powers the blind-spot LED for that respective side. The phone's application calculates speed based on the GPS within the phone and relays this information with the Microprocessor via the Bluetooth Module. The Microprocessor then communicates to the HUD which displays the speed the user is traveling. The phone's application relays turn-by-turn directions with the Microprocessor via the Bluetooth Module. The Microprocessor then communicates to the HUD which displays turn-by-turn directions using the HUD.

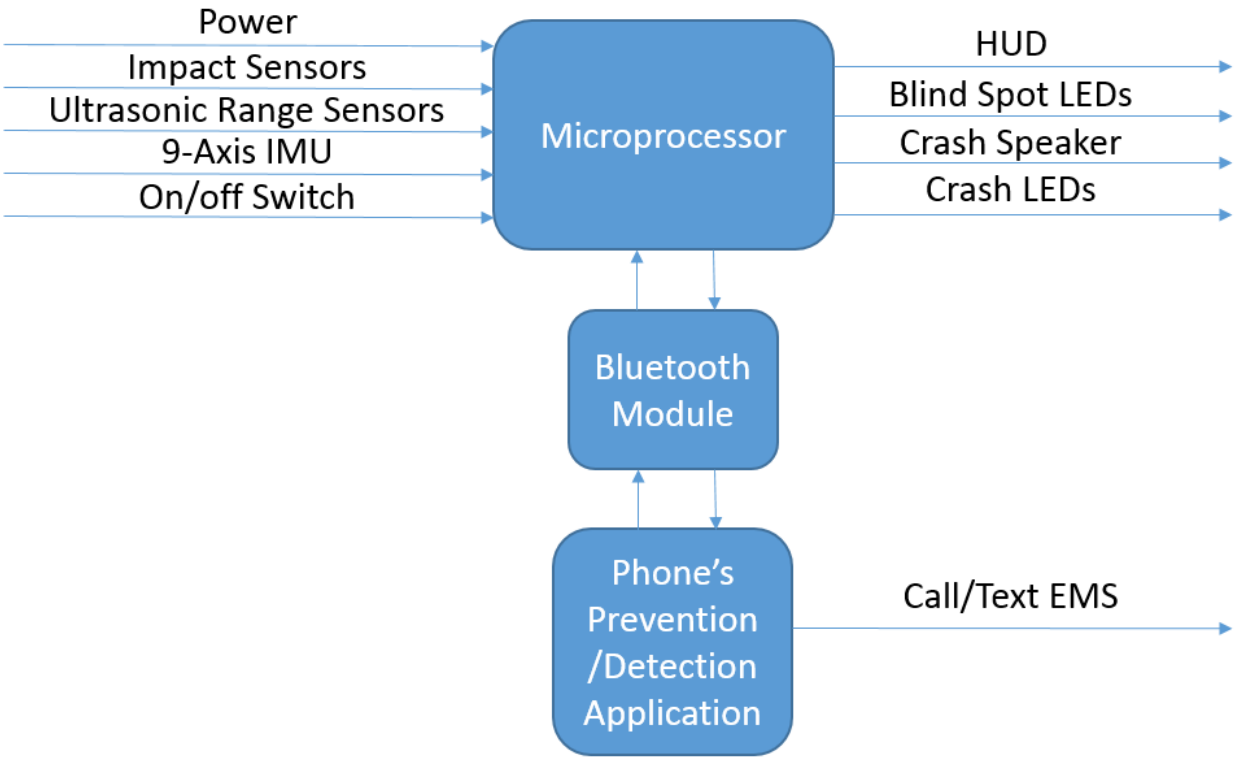



**Figure 3: Level 1 Diagram**

[JD,BF]

The hardware diagram including all major components is shown in Figure 4. The Li-Ion Battery utilizes a voltage regulator to supply voltage to all three microprocessors. An on/off switch is used between the voltage regulator and the battery. All three processors are connected to their respective proto boards in order to easily connect inputs and outputs.

One proto board connects to all and only components related to blind spot detection. The IMU works in conjunction with the range sensors. From that information, the microprocessor determines when to emit either or none of the blind spot LEDs.

The second processor is used for the external speaker. The microprocessor is connected to an audio support circuit. The audio support is connected to a waterproof speaker that will emit sound after the crash countdown is complete.

The third microprocessor is used for all other functions. A Bluetooth module is connected to the processor in order to connect to an Android cell phone running Android 4.1+. The microprocessor is connected to two impact sensors. Two external LED strips, to be emitted after the crash countdown is complete, are connected to the microprocessor. The heads-up display will show the current speed of the user, miles until the next turn, and the direction of the next turn. The cell phone will send this information over Bluetooth, through the microprocessor

and to the heads-up display. The microprocessor will also communicate to the heads-up display to initiate a countdown after a crash has been detected. The on/off switch will be used at any time to turn the system on or off.

[JD]

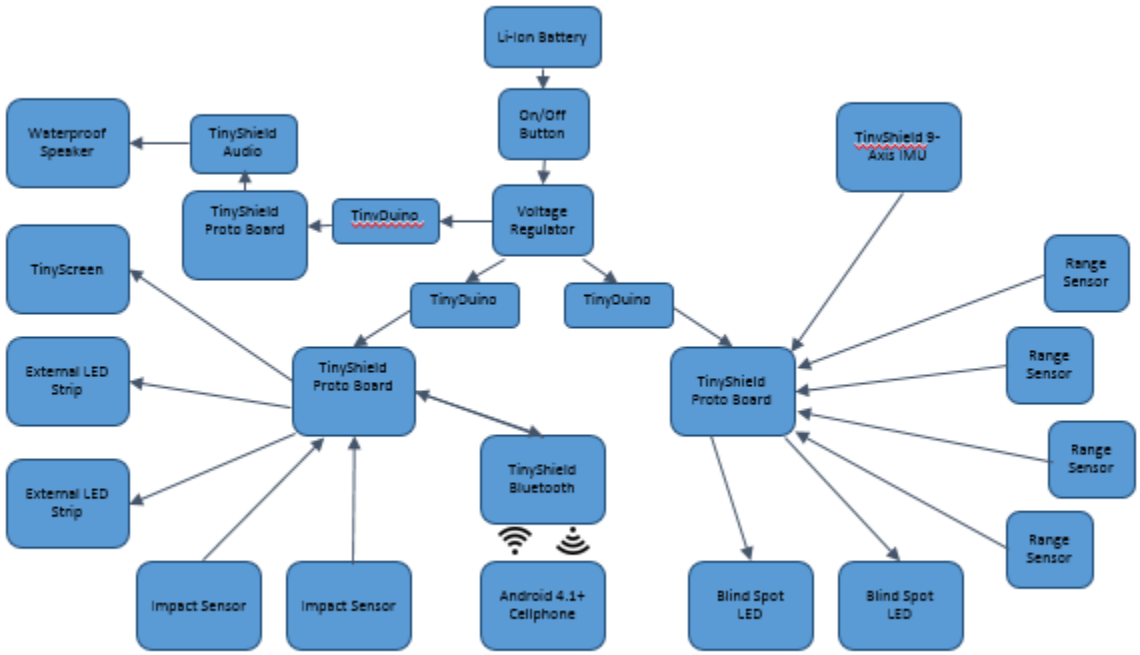



**Figure 4: Hardware Diagram**

[JD]

**Table 2. Functional Requirements of Power**

| Module | Power |
|---|---|
| Input | +7.4 V Li-Ion Battery |
| Outputs | +5 V DC |
| Functionality | Powers the system's components for six hours of use. |

**Table 3. Functional Requirements of Blind Spot Detection**

| Module | Blind Spot Detection |
|---|---|
| Input | +5 V DC<br>IMU Data |

| | |
|---|---|
| | Distance Sensor Data |
| Outputs | Blind Spot LEDs |
| Functionality | Sends distance data in conjunction with positional data of the range sensors in order to determine if an object is in the user's blind spot. If an object is in the blind spot the Blind Spot LED on the corresponding side of the object, is illuminated. |

**Table 4. Functional Requirements of Heads-Up Display**

| | |
|---|---|
| Module | Heads-Up Display |
| Input | +5 V DC<br>Android 4.1+ Cell Phone |
| Outputs | Turn-by-turn directions, speed, and countdown after crash detection. |
| Functionality | Displays turn-by-turn directions and speed.<br>Displays countdown after a crash has been detected. |

**Table 5. Functional Requirements of Crash Detection**

| | |
|---|---|
| Module | Crash Detection |
| Input | +5 V DC<br>Impact Sensors<br>Android 4.1+ Cell Phone Accelerometer |
| Outputs | Speaker, LED Strips, Android 4.1+ Cell Phone, Heads-Up Display |
| Functionality | Detects a crash based on readings from the impact sensors.<br>Detects a crash based on readings from the Android 4.1+ Cell Phone Accelerometer.<br>Starts a countdown on the heads-up display after a crash has been detected.<br>Emits sound after the crash countdown has ended.<br>Emits blinking lights after the crash countdown has ended.<br>Texts 911 with GPS coordinate after the crash countdown has ended.<br>Calls 911 and plays an audio file after the crash countdown has ended. |

Each component of Figure 4: Hardware Diagram will be detailed in the following discussions.

[JD, BF, DW, WW]

### 3.1 Power

#### 3.1.1 Voltage and Current Requirements

The Power Module is shown in Figure 5. A +7.4 V Li-Ion Battery is input to the Power Module. A +5 V is output from the Power Module. The +5 V is supplied to both microprocessors and voltage is distributed to the other components from the microprocessors. The following list describes the voltage requirements of each component.

1) The TinyDuino Microprocessor operates between +2.7 V to +5.5 V and pulls 1.2 mA.
2) The LV-MaxSonar – EZ Sonar Range Finder operates between +2.5 to +5.5 V and pulls 2 mA.
3) The Tinyshield Bluetooth Module operates between +3.3 V to + 5V and operates at 30 mA.
4) The Tiny Screen - OLED Tinyshield operates between +3 V to +5.5 V and pulls a maximum 45 mA.
5) The Tinyshield 9-Axis IMU operates between +2.4 V to +5 V and pulls 6.1 mA.
6) The Tinyshield Audio operates between +2.7 V to +5.5 V and pulls .175 mA.
7) The Knowles 2403 263 00077 waterproof speaker operates at +5 V and pulls 625 mA.
8) The Waterproof LED Strip operates at +5 V and pulls 500 mA.
9) The Panasonic Electronic Components LN48YPX LED operates at +5 V and pulls 20 mA.
10) The Force Sensitive Resistor 0.5” (impact sensor) operates at +5 V and pulls a maximum of 0.185 mA.
11) The C&K Components V70102SS05Q switch is rated up to +30 V and 4 A.


**Figure 5: Power Module**

**Table 6. Functional Requirements for Power**

| Module | Power |
|---|---|
| Input | +7.4 V Li-Ion Battery |
| Outputs | +5 V |
| Functionality | Step down the voltage from +7.4 V to +5 V.<br>Powers the system’s components for six hours of use. |

The Power Module must provide +5 V output. A +7.4V Li-Ion Battery will be input to the power module. The Power Module will regulate the voltage to +5 V. The Power Module makeup is entirely the Step-Down Voltage Regulator.

### 3.1.2 Step-Down Voltage Regulator

The Step-Down Regulator must receive a +7.4 V input and output + 5 V to the system. The functional requirements are shown below.

**Table 7. Functional Requirements for Voltage Regulator**

| | |
|---|---|
| Module | Texas Instruments LM1085ISX-5.0/NOPB Voltage Regulator |
| Input | +7.4 V |
| Outputs | +5 V, 1.64 A maximum |
| Functionality | Step down the voltage from +7.4 V to +5 V.<br>Powers the system's components for six hours of use. |

**Table 8. Functional Requirements for +7.4 Li-Ion Battery**

| | |
|---|---|
| Module | +7.4 V Li-Ion Battery |
| Input | None |
| Outputs | +7.4 V |
| Functionality | Powers the system's components for six hours of use. |

The maximum current draw if all components are active at maximums is 1.74 A. Under normal driving conditions the external LED strips and speaker are inactive. The maximum current draw under normal driving conditions is 112 mA. The maximum current draw after a crash is 1.64A.

The voltage regulator was chosen so it would step down the voltage to +5 V and sustain at least 1.64 A. The voltage regulator chosen can sustain a maximum of 25 V and 3 A. The voltage regulator circuit utilizes two capacitors with values given below. The resistor is chosen to be 363 ohms so that there is a +5 V output.

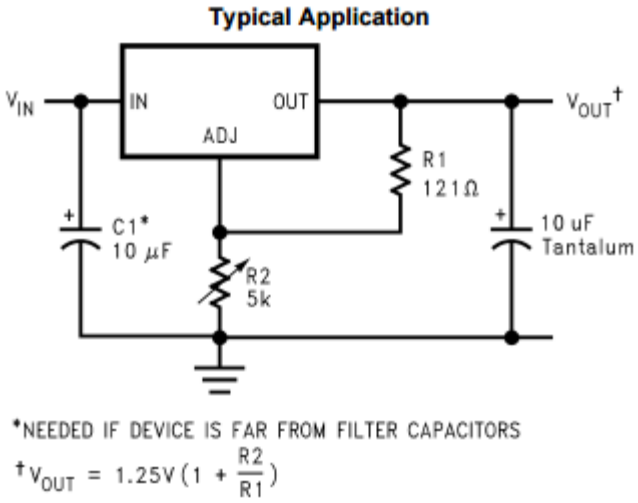



**Figure 6. Voltage Regulator Circuit**

The battery is chosen so that it could power the system for six hours. Under six hours of normal driving, the capacity needed is 672 mAh. Under crash conditions the system needs to be powered for at least thirty minutes to maintain visibility for responders. The capacity needed for crash condition is 820mAh. Total capacity needed is 1492mAh. The weight of the system must be less than 200g. The weight of the system before the battery weight is added is 90g. With 110g left the Tenergy Li-Ion 7.4 V 2200 mAh battery was chosen, as it is 99g. The battery is rated to 4 A, so it can supply the necessary current.

[JD, BF]

### 3.2 Blind Spot Detection

#### 3.2.1 Level 1 Design

The Blind Spot detection system will consist of four ultrasonic Range Sensors a 9-Axis IMU and two LEDs. It will take input from the range sensors and IMU and output to the LEDs as described in the Functional Requirements Table in Table 9.

**Table 9. Functional Requirements of Blind Spot Detection System**

| | |
|---|---|
| Module | Blind Spot Detection |
| Input | Power: 5V DC |
| Outputs | Illuminated LED light based on side that object is detected |
| Functionality | This system lights an LED on the respective side that an object is detected using Ultrasonic Range Sensors. An IMU is used to determine which sensors to use based on lean angle. |

### 3.2.2 Level 2 Design

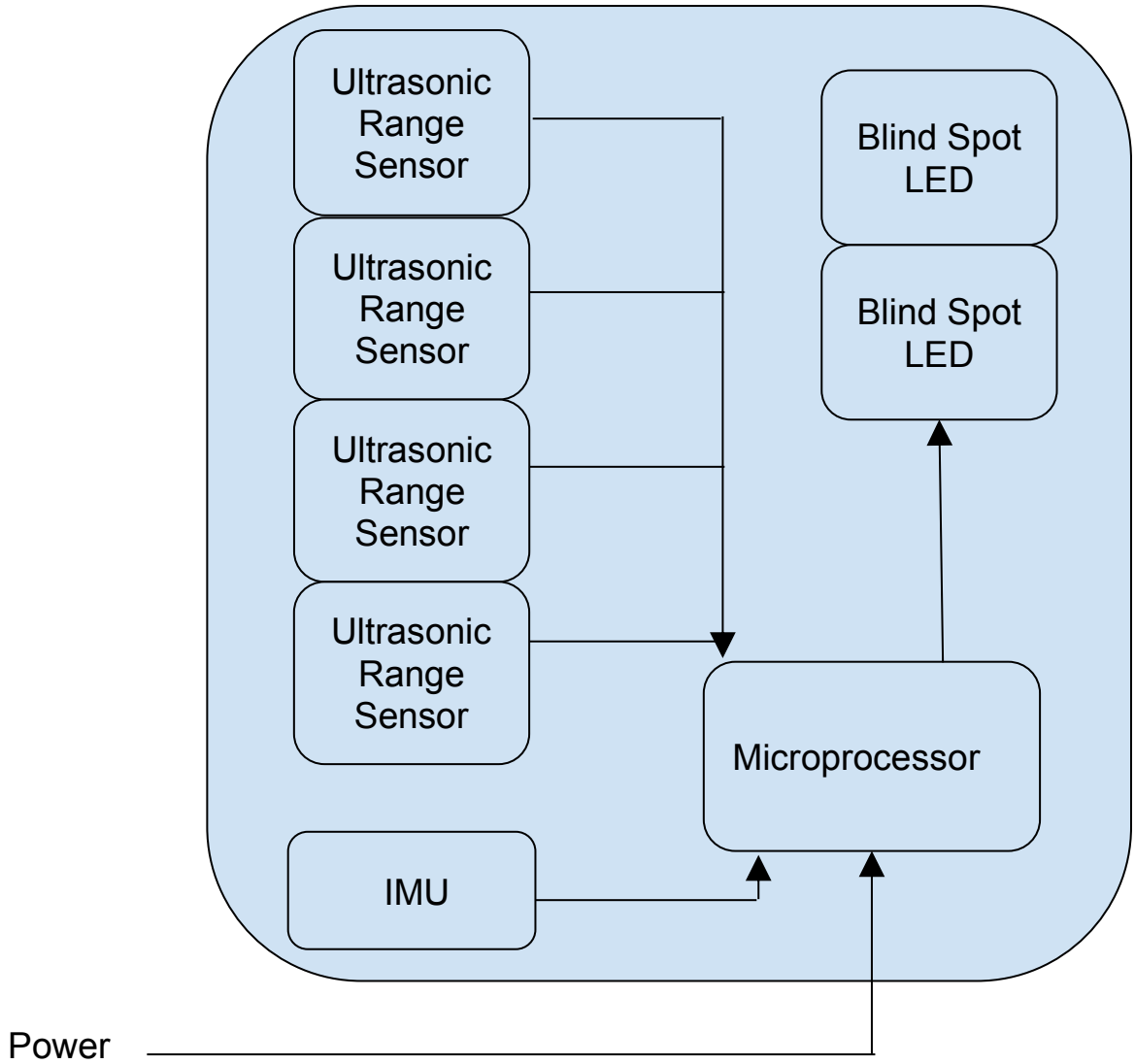



**Figure 7. Blind Spot Level 2 Block Diagram**

### 3.2.3 Ultrasonic Range Sensors

The ultrasonic range sensors are going to detect the distance to the nearest object in their line of sight. There are going to be two sensors on both sides of the helmet. They will be stacked one on top of each other on each side to provide more vertical range. We chose these range sensors because they have the necessary range, are small in size, and are cost effective.

**Table 10. Functional Requirements of Ultrasonic Range Sensors**

| | |
|---|---|
| Module | LV-MaxSonar – EZ Sonar Range Finder |
| Input | Power from microprocessor |
| Outputs | Signal to microprocessor |
| Functionality | Sends out ultrasonic pulse and calculates the round trip time of it in microseconds. It sends this information to the microprocessor. |

### 3.2.4 IMU

The IMU will be used to determine which of the ultrasonic sensors are giving relevant data. When a sensor is pointing at the ground the data that is being received is irrelevant. This IMU is produced by TinyCircuits

**Table 11. Functional Requirements of IMU**

| | |
|---|---|
| Module | 9-Axis IMU TinyShield |
| Input | Power from microprocessor |
| Outputs | Signal to microprocessor |
| Functionality | Determines an amount of change in direction on three axes. Sends this information to the microprocessor. |

### 3.2.5 Blind Spot LEDs

An LED will be on each side of the face of the user, within his/her peripheral vision but not obstructing view of the road. The color has been determined to be amber so that it is not overly bright for the user, but it is still noticeable. This is also to maintain the same color used in other vehicles' blind spot applications.

**Table 12. Functional Requirements of Blind Spot Indicating LEDs**

| | |
|---|---|
| Module | Panasonic LN48YPX LEDs |
| Input | +5 V DC from microprocessor limited with a 220 Ω resistor in series. |
| Outputs | Emit light |
| Functionality | Main Processor turns the LED on corresponding to the side in which an object has been detected in the blind spot. |

[DW, WW]

### 3.3 Heads-Up Display

The Heads-Up-Display will display the user's speed and turn-by-turn directions under normal driving conditions. The cell phone application will send speed and turn-by-turn directions to the microprocessor via Bluetooth. After the microprocessor receives this data it will forward it to the Heads-Up-Display to be displayed to the user.

Immediately following detection of a crash the Heads-Up-Display will display a countdown starting at thirty seconds. Once the microprocessor detects that a crash has occurred it will trigger the countdown on the Heads-Up-Display

The following describes the components used for the Heads-Up-Display system.

**Table 13. Functional Requirements of Tiny Screen - OLED Tinyshield**

| | |
|---|---|
| Module | TinyScreen - OLED Tinyshield |
| Input | +5 V<br>Data from microprocessor |
| Outputs | Turn-by-turn directions<br>Speed<br>Countdown |
| Functionality | Displays turn-by-turn directions and speed.<br>Displays countdown after a crash has been detected. |

The Tiny Screen - OLED Tinyshield was chosen because of the small size and weight, Arduino-compatibility, and low power consumption.

**Table 14. Functional Requirements of TinyDuino Microprocessor**

| | |
|---|---|
| Module | TinyDuino Microprocessor |
| Input | +5 V<br>Bluetooth module |
| Outputs | Data to Tiny Screen |
| Functionality | Sends turn-by-turn directions and speed.<br>Starts the Tiny Screen's countdown after a crash has been detected. |

The TinyDuino Microprocessor was used because of the size, Arduino compatibility, and low power consumption.

**Table 15. Functional Requirements of Tinyshield Bluetooth Module**

| | |
|---|---|
| Module | Tinyshield Bluetooth Module |
| Input | +5 V<br>Data from cell phone |
| Outputs | Data to microprocessor |
| Functionality | Sends turn-by-turn directions and speed to the microprocessor.<br>If the phone detects a crash, it sends an alert to the microprocessor over Bluetooth. |

The Tinyshield Bluetooth Module was used because wireless communication from the phone to the microprocessor is needed. Additionally the small size, Arduino compatibility, and low power consumption were necessary.

**Table 16. Functional Requirements of Cell Phone Application**

| | |
|---|---|
| Module | Cell Phone Application |
| Input | User |
| Outputs | Data to microprocessor |
| Functionality | Sends turn-by-turn directions and speed to the Bluetooth module.<br>If the phone detects a crash, it sends an alert to the microprocessor over Bluetooth. |

The application was needed to access the phone's accelerometer data to measure g-force. It is also needed to calculate speed based on GPS movement. The application must send turn-by-turn directions to the microprocessor through the Bluetooth module.

[JD, BF]

### 3.4 Crash Detection

The crash detection system will detect when a crash has occurred and start the crash detected procedure. The system can detect a crash from the phone or the microprocessor. The phone's application uses accelerometer data to measure g-force. If there is excessive g-force then the phone will indicate to the processor via Bluetooth that a crash has occurred. The microprocessor reads the impact sensor data and can determine that a crash has occurred.

Once a crash has been detected a countdown begins on the Heads-Up-Display and the Blind Spot LEDs blink. After the countdown reaches zero, the microprocessor instructs the phone via Bluetooth that the crash countdown has reached zero. Then, the phone will text 911 with GPS coordinates and call 911, playing an audio file from the application. The microprocessor flashes the external LED strips continuously. The microprocessor also plays an audio file through the speaker via the TinyShield Audio.

The user can turn off the system at any point during the countdown using the power switch and no further actions will be taken.

The following describes the components used for the Crash Detection system.

**Table 17. Functional Requirements of Impact Sensors**

| | |
|---|---|
| Module | Force Sensitive Resistor 0.5” |
| Input | +5 V |
| Outputs | Data to microprocessor |
| Functionality | Sends data to microprocessor for determination of a crash |

The impact sensor must be able to detect an impact above a threshold that may be categorized as a crash. An average crash is equal to about thirty-three pounds of force. The threshold for the impact sensor should be at least half of the average crash in order to detect even small crashes.

The Force Sensitive Resistor 0.5” is used because it can measure up to twenty-two pounds of force which exceeds the seventeen pounds of force needed. The sensor is lightweight, and inexpensive.

**Table 18. Functional Requirements of Android 4.1+ Cell Phone**

| | |
|---|---|
| Module | Cell Phone Application |
| Input | User<br>Microprocessor |
| Outputs | Data to microprocessor<br>Call to 911<br>Text to 911 |
| Functionality | Calculates it’s accelerometer to determine if a crash has occurred, so that it can inform the microprocessor over Bluetooth.<br>When the microprocessor communicates to the application over Bluetooth that a crash countdown sequence has reached zero, the application will text 911 with GPS coordinate and call 911, playing a recorded audio file. |

The application on the cellphone needs to be able to communicate wirelessly with the system. The application needs to access the phone's accelerometer data and calculate g-force for crash detection. The application needs to call and text 911 automatically on reception of a command by the microprocessor.

Because there is no application on the market that can perform all of these tasks, the custom Cell Phone Application is developed.

**Table 19. Functional Requirements of Microprocessor**

| | |
|---|---|
| Module | TinyDuino Microprocessor |
| Input | Force Sensitive Resistor 0.5"<br>Cell Phone Application |
| Outputs | Data to Cell Phone Application<br>TinyScreen - OLED TinyShield<br>Blind Spot LEDs<br>LED Strips<br>Speaker |
| Functionality | The microprocessor reads impact sensor data and determines if a crash has occurred. The application may communicate that a crash has occurred. Display countdown on TinyScreen. During countdown, the microprocessor will flash the Blind Spot LEDs. After countdown ends, the microprocessor will flash the LED Strips and play an audio file on the speaker through the TinyShield Audio. The microprocessor sends information to the application over Bluetooth conveying the crash countdown has reached zero. |

The microprocessor needs to be able to communicate with the TinyScreen - OLED TinyShield, TinyShield Audio, Blind Spot LEDs, Force Sensitive Resistor 0.5", and the Cell Phone Application.

The TinyDuino was chosen for this because it is able to communicate simply with TinyCircuits, as well as a wide range of Arduino compatible sensors.

**Table 20. Functional Requirements of TinyScreen - OLED TinyShield**

| | |
|---|---|
| Module | TinyScreen - OLED TinyShield |
| Input | Microprocessor |
| Outputs | Countdown |
| Functionality | The microprocessor initiates the countdown and it is displayed on the TinyScreen. |

The Heads-Up-Display must be able to display the countdown through the microprocessor.

The TinyScreen - OLED TinyShield is used as it can communicate with the other TinyCircuits, and is a reasonable size in order for the user to see the countdown.

**Table 21. Functional Requirements of Blind Spot LEDs**

| | |
|---|---|
| Module | Blind Spot LEDs |
| Input | Microprocessor |
| Outputs | Flashing |
| Functionality | The microprocessor flashes the Blind Spot LEDs during the countdown to indicate to the user that a crash has been detected. |

The LEDs must blink in order to indicate to the user that a crash has occurred.

The LEDs chosen are the Panasonic Electronic Components LN48YPX LEDs because they are inexpensive and emit light.

**Table 22. Functional Requirements of LED Strips**

| | |
|---|---|
| Module | Seagull RGB LEDs |
| Input | Microprocessor |
| Outputs | Flashing |
| Functionality | The microprocessor flashes the LED Strips after the crash countdown reaches zero. |

The LED Strips must operate on + 5 V and flash. The LED Strips chosen arrive in 8.3 cm strips with 5 LEDs. They can operate at +5 V.

**Table 23. Functional Requirements of TinyShield Audio**

| | |
|---|---|
| Module | TinyShield Audio |
| Input | Microprocessor |
| Outputs | Speaker |
| Functionality | The microprocessor plays the audio file through the TinyShield Audio to the Speaker. |

It is necessary to play audio through the external speaker.

The TinyShield Audio circuit is compatible with the TinyDuino microprocessor and can be used to play audio through a speaker.

**Table 24. Functional Requirements of Knowles Waterproof Speaker**

| | |
|---|---|
| Module | Knowles 2403 263 00077 Waterproof Speaker |
| Input | TinyShield Audio |
| Outputs | Sound |
| Functionality | The speaker emits the sounds from the TinyShield Audio |

A waterproof speaker is needed as it will be in the elements. The speaker should be small and lightweight.

The Knowles 2403 263 00077 Waterproof Speaker is chosen because it is small, lightweight, and waterproof. The Speaker utilizes a 3.5 mm jack that connects to the TinyShield Audio.

[JD, BF]

### 3.5 Software Design

There will be four programs running at all times. The first is the main crash detection program running on the main processor. The second is the blind spot program running on the secondary processor. The third is the audio program running on the third processor. The fourth is an Android 4.1 based app running on an Android 4.1+ based phone for communicating speed, directions, and accelerometer data to the main processor.

#### 3.5.1 Main Processor

This program will handle the crash detection, displaying of data on the HUD, and listening for incoming Bluetooth transmissions from the Android 4.1+ phone. In the event of a crash, it will send Bluetooth transmissions to the phone, initiating emergency response contact and notification to the secondary processor. The main processor, secondary processor, and third processor are all TinyDuinos which be programmed using the Arduino programming language which is a mix of C/C++. Figure 8 below shows the Arduino code for the Main Processor.

```
#include "Adafruit_NeoPixel.h"
#include "WS2812_Definitions.h"
#include "TinyScreen.h"
#include <SPI.h>
#include <Wire.h>

#define integrationPin 6
#define integrationPin2 7
#define FORCE1 0
#define FORCE2 1

TinyScreen display = TinyScreen(0);
```

```
int updated = 0;
int incomingByte;
int turning = 1;
int distance = 0;
int drivingSpeed = 0;
int sort = 0;
unsigned char test;
int count = 0;

Adafruit_NeoPixel leds = Adafruit_NeoPixel(5, 5, NEO_GRB + NEO_KHZ800);
float value1 = 0;
float resistance1 = 0;
float value2 = 0;
float resistance2 = 0;
float threshold = 3000;

//in root directory of SD card


void setup()
{

 Wire.begin();
 display.begin();

 pinMode(integrationPin, OUTPUT);
 pinMode(integrationPin2, OUTPUT);

   Serial.begin(9600);

 leds.begin();  // Call this to start up the LED strip.
 clearLEDs();   // This function, defined below, turns all LEDs off...
 leds.show();   // ...but the LEDs don't actually update until you call this.

 display.setFont(liberationSans_14ptFontInfo);
 display.fontColor(0x1F,0x00);
 drivingSpeed = 80;
 distance = 5;
 turning = 2;
 display.clearWindow(0,0,96,40);
 display.setCursor(20,00);
 display.print(drivingSpeed);
 display.print(" MPH");
 display.setCursor(20,20);
 display.print(distance);
 display.print(" Miles");
 display.setCursor(20,40);
 display.print("======>");
}

void loop()
{
 value1 = analogRead(FORCE1);
 resistance1 = 1/(((26.4 * value1)/(1-(value1/1023.0)))) *10000000;
 value2 = analogRead(FORCE2);
 resistance2 = 1/(((26.4 * value2)/(1-(value2/1023.0)))) *10000000;
 //Serial.print(resistance1,DEC);
```

```
 //Serial.print("         ");
 //Serial.println(resistance2,DEC);

 if((resistance1 > threshold) || (resistance2 > threshold))
 {
   //delay(1000);
     countdown(10);
 }

 bool needToUpdate = false;
   if (Serial.available())
 {

     incomingByte = Serial.read();
     if(incomingByte >= 0){
       if(sort != 0){
         count ++;
         if (count > 30){
           sort = 0;
           count = 0;
         }
       }
         if(sort == 0){
           sort = incomingByte;
         }
         else if (sort == 1){
           if(incomingByte != drivingSpeed){
             drivingSpeed = incomingByte;
             needToUpdate = true;
           }

           sort = 0;
           //Serial.print(drivingSpeed);
         }
         else if (sort == 2){
           if(incomingByte != distance){
             distance = incomingByte;
             needToUpdate = true;
           }
           sort = 0;
           //Serial.print(distance);
         }
         else if(sort == 3){
           if(incomingByte != turning){
             turning = incomingByte;
             needToUpdate = true;
           }
           sort = 0;
         }
     }
 }

if(needToUpdate){
 updateScreen();
}

 delay(50);
}
```

```
void updateScreen(){
 display.clearWindow(0,0,96,40);
  display.setCursor(20,00);
  display.print(drivingSpeed);
  display.print(" MPH");
  display.setCursor(20,20);
  display.print(distance);
  display.print(" Miles");
  display.setCursor(20,40);
  if(turning == 1){
    display.print("<======");
  }
  if(turning == 2){
    display.print("======>");
  }
}
void countdown(int seconds)
{
 while((seconds > 0))
 {
   display.setFont(liberationSans_22ptFontInfo);
   display.clearWindow(0,0,96,64);
   display.setCursor(20,20);
   display.print(seconds);
   //Serial.println(seconds);
   delay(1000);
   seconds--;
 }
 display.setFont(liberationSans_22ptFontInfo);
   display.clearWindow(0,0,96,64);
   display.setCursor(20,20);
   display.print(seconds);
 crashDetected();
}
void crashDetected()
{
 digitalWrite(integrationPin, HIGH);
 digitalWrite(integrationPin2, HIGH);
 Serial.print(3);
 while(true){

   for (int j=0; j<5; j++)
   {
   // First parameter is the color, second is direction, third is ms between falls
   for (int i=0; i<5; i++)
   {
     clearLEDs();  // Turn off all LEDs
     leds.setPixelColor(i, MEDIUMSPRINGGREEN);  // Set just this one
     leds.show();
     delay(60);
   }
   }
 }
}

void clearLEDs()
{
 for (int i=0; i<5; i++)
```

```
 {
   leds.setPixelColor(i, 0);
 }
}
// this handy function will return the number of bytes currently free in RAM, great for
debugging!
int freeRam(void)
{
 extern int  __bss_end;
 extern int  *__brkval;
 int free_memory;
 if((int)__brkval == 0) {
   free_memory = ((int)&free_memory) - ((int)&__bss_end);
 }
 else {
   free_memory = ((int)&free_memory) - ((int)__brkval);
 }
 return free_memory;
}
```

**Figure 8. Main Processor Code**

### 3.5.2 Blind Spot Processor

This program handles the detection of objects within proximity of the helmet as well as notification to the rider of which side the object is on. This program is also listening for notification of a crash from the main processor to start a repetitive flashing of the LEDs to notify the wearer that a crash has been detected and help is on the way. Figure 9 below shows the Arduino code for the Blind Spot System.

```
#include <Wire.h>
#include "I2Cdev.h"
#include "RTIMUSettings.h"
#include "RTIMU.h"
#include "RTFusionRTQF.h"
#include "CalLib.h"
#include <EEPROM.h>



RTIMU *imu;                                           // the IMU object
RTFusionRTQF fusion;                                  // the fusion object
RTIMUSettings settings;                               // the settings object

//  DISPLAY_INTERVAL sets the rate at which results are displayed

#define DISPLAY_INTERVAL  300                         // interval between pose displays

//  SERIAL_PORT_SPEED defines the speed to use for the debug serial port
```

```
#define  SERIAL_PORT_SPEED  115200

#define Power1 6
#define Power2 7
#define Power3 8
#define Power4 9
#define Data1 0
#define Data2 1
#define Data3 2
#define Data4 3
#define LEDleft 4
#define LEDright 5
#define integrationPin 3

#define leftThreshold  33
#define rightThreshold  -7



unsigned long lastDisplay;
unsigned long lastRate;
int sampleCount;

void setup()
{
   pinMode(3, INPUT);
   pinMode(4, OUTPUT);
   pinMode(5, OUTPUT);
   pinMode(6, OUTPUT);
   pinMode(7, OUTPUT);
   pinMode(8, OUTPUT);
   pinMode(9, OUTPUT);


   int errcode;

   Serial.begin(SERIAL_PORT_SPEED);
   Wire.begin();
   delay(100);

   imu = RTIMU::createIMU(&settings);                        // create the imu object

   Serial.print(F("ArduinoIMU starting using device ")); Serial.println(imu->IMUName());
   if ((errcode = imu->IMUInit()) < 0) {
       Serial.print("Failed to init IMU: "); Serial.println(errcode);
   }

   if (imu->getCalibrationValid())
       Serial.println("Using compass calibration");
   else
       Serial.println("No valid compass calibration data");

   lastDisplay = lastRate = millis();
   sampleCount = 0;
```

```
   // Slerp power controls the fusion and can be between 0 and 1
   // 0 means that only gyros are used, 1 means that only accels/compass are used
   // In-between gives the fusion mix.

   fusion.setSlerpPower(0.02);

   // use of sensors in the fusion algorithm can be controlled here
   // change any of these to false to disable that sensor

   fusion.setGyroEnable(true);
   fusion.setAccelEnable(true);
   fusion.setCompassEnable(true);


}

void loop() {
 if(digitalRead(integrationPin) == LOW){
   unsigned long now = millis();
   unsigned long delta;
   int loopCount = 1;

   while (imu->IMURead()) {                                // get the latest data if ready
yet
       // this flushes remaining data in case we are falling behind
       if (++loopCount >= 10)
           continue;
       fusion.newIMUData(imu->getGyro(), imu->getAccel(), imu->getCompass(), imu-
>getTimestamp());
       sampleCount++;
       if ((delta = now - lastRate) >= 1000) {
           Serial.print("Sample rate: "); Serial.print(sampleCount);
           if (imu->IMUGyroBiasValid())
               Serial.println(", gyro bias valid");
           else
               Serial.println(", calculating gyro bias");

           sampleCount = 0;
           lastRate = now;
       }
       if ((now - lastDisplay) >= DISPLAY_INTERVAL) {
           lastDisplay = now;
//          RTMath::display("Gyro:", (RTVector3&)imu->getGyro());                // gyro
data
//          RTMath::display("Accel:", (RTVector3&)imu->getAccel());              // accel
data
//          RTMath::display("Mag:", (RTVector3&)imu->getCompass());              // compass
data
           RTVector3& vec = (RTVector3&)fusion.getFusionPose();
//           RTMath::displayRollPitchYaw("Pose:", vec); // fused output
           Serial.print("roll              ");

           Serial.print(vec.y() * RTMATH_RAD_TO_DEGREE);
          Serial.println();
          if(vec.y() * RTMATH_RAD_TO_DEGREE > leftThreshold){//turning left
           digitalWrite(Power1, HIGH);
           digitalWrite(Power2, LOW);
           digitalWrite(Power3, LOW);
```

```
          digitalWrite(Power4, HIGH);
          delay(25);
          float value1 = analogRead(Data1);
          Serial.print("BS1: ");
          Serial.println(value1);
          float value4 = analogRead(Data4);
          Serial.print("BS4: ");
          Serial.println(value4);
          if((value1 < 20) && (value1 != 0)){
            digitalWrite(LEDleft, HIGH);
            delay(2000);
          }
          else{
            digitalWrite(LEDleft, LOW);
          }
          if((value4 < 20) && (value4 != 0)){
            digitalWrite(LEDright, HIGH);
            delay(2000);

          }
          else{
            digitalWrite(LEDright, LOW);
          }
        }
        else if(vec.y() * RTMATH_RAD_TO_DEGREE < rightThreshold){//turning right
          digitalWrite(Power1, LOW);
          digitalWrite(Power2, HIGH);
          digitalWrite(Power3, HIGH);
          digitalWrite(Power4, LOW);
          delay(25);
          float value2 = analogRead(Data2);
        Serial.print("BS2: ");
          Serial.println(value2);
          float value3 = analogRead(Data3);
          Serial.print("BS3: ");
          Serial.println(value3);
          if((value2 < 20) && (value2 != 0)){
            digitalWrite(LEDleft, HIGH);
            delay(2000);
          }
          else{
            digitalWrite(LEDleft, LOW);
          }
          if((value3 < 20) && (value3 != 0)){
            digitalWrite(LEDright, HIGH);
            delay(2000);
          }
          else{
            digitalWrite(LEDright, LOW);
          }
        }
        else{
          digitalWrite(Power1, LOW);
          digitalWrite(Power2, HIGH);
          digitalWrite(Power3, LOW);
          digitalWrite(Power4, HIGH);
          delay(25);
          float value2 = analogRead(Data2);
          Serial.print("BS2: ");
```

```
            Serial.println(value2);
            float value4 = analogRead(Data4);
            Serial.print("BS4: ");
            Serial.println(value4);
            if((value2 < 20) && (value2 != 0)){
              digitalWrite(LEDleft, HIGH);
              delay(2000);
            }
            else{
              digitalWrite(LEDleft, LOW);
            }
            if((value4 < 20) && (value4 != 0)){
              digitalWrite(LEDright, HIGH);
              delay(2000);
            }
            else{
              digitalWrite(LEDright, LOW);
            }
           }
        }
    }
  }
  else{//crash has been detected
    digitalWrite(Power1, LOW);
    digitalWrite(Power2, LOW);
    digitalWrite(Power3, LOW);
    digitalWrite(Power4, LOW);
    while(true){

      digitalWrite(LEDleft, HIGH);
      digitalWrite(LEDright, HIGH);
      delay(1000);
      digitalWrite(LEDleft, LOW);
      digitalWrite(LEDright, LOW);
      delay(1000);
    }
  }
}
```

**Figure 10. Blind Spot Processor Code**

### 3.5.3 Audio Processor

This program handles the playing of audio when a crash is detected. This had to be broken off of the main program due to the RAM limitations of the TinyDuino and the large size of the WaveHC library. This program is listening for notification of a crash from the main processor to start a repetitive playing of a siren through the external processor. Figure 10 below show the Arduino code for the Audio System.

```
#include "WaveUtil.h"
#include "WaveHC.h"

#define integrationPin 5
char fileName[]="song1.wav";

SdReader card;    // This object holds the information for the card
FatVolume vol;    // This holds the information for the partition on the card
FatReader root;   // This holds the information for the filesystem on the card
FatReader f;      // This holds the information for the file we're play

WaveHC wave;      // This is the only wave (audio) object, since we will only play one at a
time




void setup() {
 // put your setup code here, to run once:
byte i;
Serial.begin(9600);

 pinMode(integrationPin, INPUT);
 // Set the output pins for the DAC control. This pins are defined in the library
 pinMode(2, OUTPUT);
 pinMode(3, OUTPUT);
 pinMode(4, OUTPUT);
 // pin13 LED
 pinMode(13, OUTPUT);
 if (!card.init()) {         //play with 8 MHz spi (default faster!)
   putstring_nl("Card init. failed!");  // Something went wrong, lets print out why
   sdErrorCheck();
   while(1);                           // then 'halt' - do nothing!
 }

 // enable optimize read - some cards may timeout. Disable if you're having problems
 card.partialBlockRead(true);
// Now we will look for a FAT partition!
 uint8_t part;
 for (part = 0; part < 5; part++) {     // we have up to 5 slots to look in
   if (vol.init(card, part))
     break;                            // we found one, lets bail
 }
 if (part == 5) {                      // if we ended up not finding one  :(
   putstring_nl("No valid FAT partition!");
   sdErrorCheck();      // Something went wrong, lets print out why
   while(1);                           // then 'halt' - do nothing!
 }

 // Lets tell the user about what we found
 putstring("Using partition ");
 Serial.print(part, DEC);
 putstring(", type is FAT");
 Serial.println(vol.fatType(),DEC);     // FAT16 or FAT32?

 // Try to open the root directory
```

```
 if (!root.openRoot(vol)) {
   putstring_nl("Can't open root dir!"); // Something went wrong,
   while(1);                             // then 'halt' - do nothing!
 }

 // Whew! We got past the tough parts.
 putstring_nl("Ready!");
}

void loop() {
 if(digitalRead(integrationPin) == HIGH){

 while(true){
 // put your main code here, to run repeatedly:
Serial.print("Playing ");
 Serial.println(fileName);
 playfile(fileName);
 Serial.println("crash");
 while(wave.isplaying){
  Serial.println("crash");
  delay(50);
 }
 }
 }
 delay(100);
}
void playfile(char *name) {
 // see if the wave object is currently doing something
 if (wave.isplaying) {// already playing something, so stop it!
   wave.stop(); // stop it
 }
 // look in the root directory and open the file
 if (!f.open(root, name)) {
   putstring("Couldn't open file "); Serial.print(name); return;
 }
 // OK read the file and turn it into a wave object
 if (!wave.create(f)) {
   putstring_nl("Not a valid WAV"); return;
 }

 // ok time to play! start playback
 wave.play();
}

void sdErrorCheck(void)
{
 if (!card.errorCode()) return;
 putstring("\n\rSD I/O error: ");
 Serial.print(card.errorCode(), HEX);
 putstring(", ");
 Serial.println(card.errorData(), HEX);
 while(1);
}
```

**Figure 10. Audio Processor Code**

### 3.5.4 Crash Detection/Prevention System Android Application

This program runs on the Android 4.1+ phone and collects GPS speed data, Google Maps data, and accelerometer data. It sends this data to the main processor via Bluetooth. The phone app will be programmed in Java. Figures 11, 12, 13, and 14 below show the Java code for the Android Application.

```
package com.example.student.mhcdps;

import android.app.Activity;
import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothDevice;
import android.bluetooth.BluetoothSocket;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.location.Location;
import android.location.LocationListener;
import android.location.LocationManager;
import android.net.Uri;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.telephony.PhoneStateListener;
import android.telephony.SmsManager;
import android.telephony.TelephonyManager;
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.view.ViewDebug;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ListView;
import android.widget.TextView;
import android.widget.Toast;

import java.io.IOException;
import java.nio.ByteBuffer;
import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import java.util.UUID;

public class MainActivity extends Activity {
   private NotificationReceiver nReceiver;



   private static final int REQUEST_ENABLE_BT = 1;
   Button btnConnectDevice, btnSend1, btnSend0, btnSendText, btnSendSpeed, btnSendDistance;
   TextView txtViewTitle;
   EditText editTxtToSend;
```

```
   ListView lstDevices;

   private ArrayAdapter<String> mNewDevicesArrayAdapter;
   private static final UUID MY_UUID = UUID.fromString("00001101-0000-1000-8000-
00805F9B34FB");
   private String MACAddress = "00:00:00:00:00:10";//Fake made up default MAC Address...
   private BluetoothAdapter btAdapter = null;
   private BluetoothSocket btSocket = null;
   private StringBuilder sb = new StringBuilder();
   private ConnectedThread mConnectedThread;
   List<String> pairedDevicesList = new ArrayList<String>();
   List<String> newDevicesList = new ArrayList<String>();
   List<String> deviceList = new ArrayList<String>();

   String lat = "00";
   String lon = "00";
   String speed = "00";
   String GPSinfo = "00";
   String directions = "";

   final Handler periodicHandler = new Handler(){
   };

   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);
       nReceiver = new NotificationReceiver();
       IntentFilter filter = new IntentFilter();
       filter.addAction("notification_event");
       registerReceiver(nReceiver, filter);

       //buttons
       btnConnectDevice = (Button) findViewById(R.id.btnConnectDevice);
       btnSendSpeed = (Button) findViewById(R.id.btnSendSpeed);
       btnSendDistance = (Button) findViewById(R.id.btnSendDistance);
       btnSendText = (Button) findViewById(R.id.btnSendText);
       txtViewTitle = (TextView) findViewById(R.id.txtViewTitle);
       editTxtToSend = (EditText) findViewById(R.id.editTxtToSend);
       lstDevices = (ListView) findViewById(R.id.lstDevices);

       LocationManager locationManager = (LocationManager) this
               .getSystemService(Context.LOCATION_SERVICE);

       // Define a listener that responds to location updates
       LocationListener locationListener = new LocationListener() {
           @Override
           public void onLocationChanged(Location location) {
               lat = Double.toString(location.getLatitude()) ;
               lon = Double.toString(location.getLongitude()) ;
               speed = Float.toString(location.getSpeed());
               GPSinfo = lat +" "+ lon + "\n";
               //Intent i = new Intent("notification_event");
               //i.putExtra("notification_event", GPSinfo);
               //sendBroadcast(i);
               //Log.i(TAG, GPSinfo);
           }
```

```
            @Override
            public void onStatusChanged(String provider, int status, Bundle extras) {
            }

            @Override
            public void onProviderEnabled(String provider) {
            }

            @Override
            public void onProviderDisabled(String provider) {
            }
        };

        locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0, 0,
locationListener);



        //phone call
        PhoneStateListener phoneListener = new PhoneStateListener();
        TelephonyManager telephonyManager = (TelephonyManager) this
                .getSystemService(Context.TELEPHONY_SERVICE);
        telephonyManager.listen(phoneListener, PhoneStateListener.LISTEN_CALL_STATE);

        //text
        SmsManager smsManager = SmsManager.getDefault();
        if(!GPSinfo.equals("00")){ smsManager.sendTextMessage("3303220802", null, "Crash
occurred at: " + GPSinfo, null, null);}

        btAdapter = BluetoothAdapter.getDefaultAdapter();

        if (!btAdapter.isEnabled()) {
            Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
            startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
        }

        mNewDevicesArrayAdapter = new ArrayAdapter<String>(this,R.layout.activity_main,
pairedDevicesList);
        lstDevices.setAdapter(mNewDevicesArrayAdapter);

        btnConnectDevice.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {

                Set<BluetoothDevice> pairedDevices = btAdapter.getBondedDevices();
                pairedDevicesList.clear();
                int numDevices = 0;
                if (pairedDevices.size() > 0) {
                    for (BluetoothDevice device : pairedDevices) {
                        String name = device.getName();
                        if (name != null) {
                            pairedDevicesList.add(name);
                            //Toast.makeText(getBaseContext(),"Found Device: " + name + "\n"
+ device.getAddress() ,Toast.LENGTH_SHORT).show();
```

```
                            numDevices++;
                        }
                    }
                    if (numDevices == 0) {
                        Toast.makeText(getBaseContext(), "No Paired Devices",
Toast.LENGTH_SHORT).show();
                    }
                }
                //Toast.makeText(getBaseContext(),"Number of Devices: " +
pairedDevicesList.size() ,Toast.LENGTH_SHORT).show();
                lstDevices.setAdapter(new ArrayAdapter<String>(getBaseContext(),
                        android.R.layout.simple_list_item_1, pairedDevicesList));

            }
        });

        final Handler mHandler = new Handler() {
            @Override
            public void handleMessage(Message msg) {
                switch (msg.what) {
                    case 1:
                        byte[] readBuf = (byte[]) msg.obj;
                        // construct a string from the valid bytes in the buffer
                        String readMessage = new String(readBuf, 0, msg.arg1);
                        Toast.makeText(getBaseContext(), "Message Received" + readMessage,
Toast.LENGTH_SHORT).show();

                        break;
                }
            }
        };



        btnSendSpeed.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                if(mConnectedThread != null){
                    if(editTxtToSend.toString().length()>0) {
                        int speed = Integer.parseInt(editTxtToSend.getText().toString());
                        byte[] intToBytes = ByteBuffer.allocate(4).putInt(speed).array();
                        byte[] sendSpeed = new byte[]{(byte) 0x01, intToBytes[3]};
                        mConnectedThread.write(sendSpeed);
                    }
                }

            }
        });

        btnSendDistance.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                if(mConnectedThread != null){
                    if(editTxtToSend.toString().length()>0) {
                        int distance = Integer.parseInt(editTxtToSend.getText().toString());
                        byte[] intToBytes = ByteBuffer.allocate(4).putInt(distance).array();
```

```
                        byte[] sendDistance = new byte[]{(byte) 0x02, intToBytes[3]};
                        mConnectedThread.write(sendDistance);
                    }
                }
            }
        });


        lstDevices.setOnItemClickListener(new AdapterView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> adapterView, View view, int position, long
l) {
                String selected = lstDevices.getItemAtPosition(position).toString();
                //String selected = ((TextView)
view.findViewById(R.id.list_item)).getText().toString();

                Set<BluetoothDevice> pairedDevices = btAdapter.getBondedDevices();
                if (pairedDevices.size() > 0) {
                    for (BluetoothDevice device : pairedDevices) {
                        String name = device.getName();
                        if(name.equals(selected)){
                            //Toast.makeText(getBaseContext(), "FOUND DEVICE:" + name + "|" +
selected, Toast.LENGTH_SHORT).show();
                            try {
                                btSocket = device.createRfcommSocketToServiceRecord(MY_UUID);
                            } catch (IOException e) {
                                Toast.makeText(getBaseContext(), "Create RF : Failed",
Toast.LENGTH_SHORT).show();
                                e.printStackTrace();
                            }

                            try {
                                btSocket.connect();
                                Toast.makeText(getBaseContext(), "Connection to :" + name + "
Success", Toast.LENGTH_SHORT).show();
                            } catch (IOException e) {
                                try {
                                    Toast.makeText(getBaseContext(), "Connection to :" + name
+ " Failed", Toast.LENGTH_SHORT).show();
                                    btSocket.close();
                                } catch (IOException e1) {
                                    return;
                                }
                            }

                            mConnectedThread = new ConnectedThread(btSocket, mHandler);
                            mConnectedThread.start();

                            periodicHandler.postDelayed(sendDataPeriodic, 10000);
                            //ScheduledExecutorService executor =
Executors.newScheduledThreadPool(1);
                            //executor.scheduleAtFixedRate(sendDataPeriodic, 0, 3,
TimeUnit.SECONDS);

                        }
                    }
                }
```

```
        }
    });
}

@Override
protected void onDestroy() {
    super.onDestroy();
    unregisterReceiver(nReceiver);
    periodicHandler.removeCallbacks(sendDataPeriodic);
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.menu_main, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();

    //noinspection SimplifiableIfStatement
    if (id == R.id.action_settings) {
        return true;
    }

    return super.onOptionsItemSelected(item);
}

class NotificationReceiver extends BroadcastReceiver {
    private String TAG = this.getClass().getSimpleName();

    @Override
    public void onReceive(Context context, Intent intent) {
        directions = intent.getStringExtra("notification_event");
        if (directions.length() > 7) {
            Log.i(TAG, directions);
        }
    }
}

Runnable sendDataPeriodic = new Runnable() {
    public void run() {
        if(mConnectedThread != null){
            if(editTxtToSend.toString().length()>0) {
                int speed = Integer.parseInt(editTxtToSend.getText().toString());
                byte[] intToBytes = ByteBuffer.allocate(4).putInt(speed).array();
                byte[] sendSpeed = new byte[]{(byte) 0x01, intToBytes[3]};
                mConnectedThread.write(sendSpeed);
            }
        }
        if(mConnectedThread != null){
```

```
                if(editTxtToSend.toString().length()>0) {
                    int distance = Integer.parseInt(editTxtToSend.getText().toString());
                    byte[] intToBytes = ByteBuffer.allocate(4).putInt(distance).array();
                    byte[] sendDistance = new byte[]{(byte) 0x02, intToBytes[3]};
                    mConnectedThread.write(sendDistance);
                }
            }
            periodicHandler.postDelayed(this, 3000);
        }
    };
    public void crashDetected(){
        Intent callIntent = new Intent(Intent.ACTION_CALL);
        //phone call
        callIntent.setData(Uri.parse("tel:3303220802"));
        startActivity(callIntent);

        //text
        SmsManager smsManager = SmsManager.getDefault();
        smsManager.sendTextMessage("3303220802", null, "Crash located at longitude: " + lon +
", latitude: " + lat, null, null);
    }

}
```

**Figure 11. MainActivity.java**

```
package com.example.student.mhcdps;

import android.app.Activity;
import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothDevice;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.Bundle;
import android.widget.ArrayAdapter;

import java.util.Set;


/**
* Created by Student on 2/18/2016.
*/
public class BTConnection extends Activity {
    BluetoothAdapter mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
    int REQUEST_ENABLE_BT;
    ArrayAdapter mArrayAdapter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        if (mBluetoothAdapter == null){
```

```
            //Device does nto support Bluetooth
        }

        if(!mBluetoothAdapter.isEnabled()){
            Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
            startActivityForResult(enableBtIntent,REQUEST_ENABLE_BT);
        }

    }

    public void getPaired(){
        Set<BluetoothDevice> pairedDevices = mBluetoothAdapter.getBondedDevices();
        if(pairedDevices.size() > 0){
            for(BluetoothDevice device : pairedDevices){
                mArrayAdapter.add(device.getName() + "\n" + device.getAddress());
            }
        }
    }

    public void deviceDiscovery()
    {
        BroadcastReceiver mReceiver = new BroadcastReceiver() {
            public void onReceive(Context context, Intent intent){
                String action = intent.getAction();

                if(BluetoothDevice.ACTION_FOUND.equals(action)){
                    BluetoothDevice device =
intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
                    mArrayAdapter.add(device.getName() + "\n" + device.getAddress());
                }
            }

        };

        IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
        registerReceiver(mReceiver, filter);
    }

    public void enableDiscoverability(){
        Intent discoverableIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);
        discoverableIntent.putExtra(BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION,300);
        startActivity(discoverableIntent);
    }




}
```

**Figure 12. BTConnection.java**

```
package com.example.student.mhcdps;

import android.bluetooth.BluetoothSocket;
import android.os.Handler;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

/**
* Created by Student on 2/25/2016.
*/

class ConnectedThread extends Thread {
   private static final int MESSAGE_READ = 1;
   private final BluetoothSocket mmSocket;
   private final InputStream mmInStream;
   private final OutputStream mmOutStream;
   private Handler mHandler = new Handler();

   public ConnectedThread(BluetoothSocket socket, Handler handler) {
       mmSocket = socket;
       InputStream tmpIn = null;
       OutputStream tmpOut = null;
       mHandler = handler;

       // Get the input and output streams, using temp objects because
       // member streams are final
       try {
           tmpIn = socket.getInputStream();
           tmpOut = socket.getOutputStream();
       } catch (IOException e) { }

       mmInStream = tmpIn;
       mmOutStream = tmpOut;
   }

   public void run() {
       byte[] buffer = new byte[1024];  // buffer store for the stream
       int bytes; // bytes returned from read()

       // Keep listening to the InputStream until an exception occurs
       while (true) {
           try {
               // Read from the InputStream
               bytes = mmInStream.read(buffer);
               // Send the obtained bytes to the UI activity
               mHandler.obtainMessage(MESSAGE_READ, bytes, -1, buffer).sendToTarget();
           } catch (IOException e) {
               break;
           }
       }
   }

   /* Call this from the main activity to send data to the remote device */
   public void write(byte[] bytes) {
```

```
        try {
            mmOutStream.write(bytes);
        } catch (IOException e) { }
    }

    public void write(String message){
        //Log.d(TAG, "...Data to send: " + message + "...");
        byte[] msgBuffer = message.getBytes();
        try{
            mmOutStream.write(msgBuffer);
        }catch(IOException e){
            //Log.d(TAG, "...Error data send: " + e.getMessage() + "...");
        }
    }

    /* Call this from the main activity to shutdown the connection */
    public void cancel() {
        try {
            mmSocket.close();
        } catch (IOException e) { }
    }
}
```

**Figure 13. ConnectedThread.java**

```
package com.example.student.mhcdps;

/**
* Created by Flint on 4/3/2016.
*/

import android.app.Notification;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.Bundle;
import android.service.notification.NotificationListenerService;
import android.service.notification.StatusBarNotification;

public class NLService extends NotificationListenerService {

    private String TAG = this.getClass().getSimpleName();
    private NLServiceReceiver nlservicereciver;
    @Override
    public void onCreate() {
        super.onCreate();
        nlservicereciver = new NLServiceReceiver();
        IntentFilter filter = new IntentFilter();
        filter.addAction("notification_event");
        registerReceiver(nlservicereciver,filter);
    }
```

```
    @Override
    public void onDestroy() {
        super.onDestroy();
        unregisterReceiver(nlservicereciver);
    }

    @Override
    public void onNotificationPosted(StatusBarNotification sbn) {
        CharSequence maps = "com.mapquest.android.ace";
        //CharSequence maps = "com.google.android.apps.maps";
        Notification mNotification = sbn.getNotification();

        if(sbn.getPackageName().contains(maps)) {
            Bundle extras = mNotification.extras;
            //String a = (extras.toString());
            String direction = (extras.getCharSequence(Notification.EXTRA_TITLE)).toString();
            String distance = (extras.getCharSequence(Notification.EXTRA_TEXT)).toString();
            String ddsend = distance + "\n" + direction;
            Intent i = new Intent("notification_event");
            i.putExtra("notification_event", ddsend);
            sendBroadcast(i);
        }
    }

    @Override
    public void onNotificationRemoved(StatusBarNotification sbn) {
    }

    class NLServiceReceiver extends BroadcastReceiver {
        @Override
        public void onReceive(Context context, Intent intent) {

        }

    }

}
```

**Figure 14. NLService.java**

[DW]

## 3.6 Schematics

### 3.6.1 LV-MaxSonar – EZ Sonar Range Finder

#### Pin Out Description

**Pin 1-BW-***Leave open or hold low for serial output on the TX output. When BW pin is held high the TX output sends a pulse (instead of serial data), suitable for low noise chaining.

**Pin 2-PW**- This pin outputs a pulse width representation of range. The distance can be calculated using the scale factor of 147uS per inch.

**Pin 3-AN-** Outputs analog voltage with a scaling factor of (Vcc/512) per inch. A supply of 5V yields ~9.8mV/in. and 3.3V yields ~6.4mV/in. The output is buffered and corresponds to the most recent range data.

**Pin 4-RX–** This pin is internally pulled high. The LV-MaxSonar-EZ will continually measure range and output if RX data is left unconnected or held high. If held low the sensor will stop ranging. Bring high for 20uS or more to command a range reading.

**Pin 5-TX-** When the *BW is open or held low, the TX output delivers asynchronous serial with an RS232 format, except voltages are 0-Vcc. The output is an ASCII capital "R", followed by three ASCII character digits representing the range in inches up to a maximum of 255, followed by a carriage return (ASCII 13). The baud rate is 9600, 8 bits, no parity, with one stop bit. Although the voltage of 0-Vcc is outside the RS232 standard, most RS232 devices have sufficient margin to read 0-Vcc serial data. If standard voltage level RS232 is desired, invert, and connect an RS232 converter such as a MAX232. When BW pin is held high the TX output sends a single pulse, suitable for low noise chaining. (no serial data)

**Pin 6-+5V**- Vcc – Operates on 2.5V - 5.5V. Recommended current capability of 3mA for 5V, and 2mA for 3V.

**Pin 7-GND**- Return for the DC power supply. GND (& Vcc) must be ripple and noise free for best operation.

**Figure 15. LV-MaxSonar – EZ Sonar Range Finder Pin Out**

### 3.6.2 TinyDuino

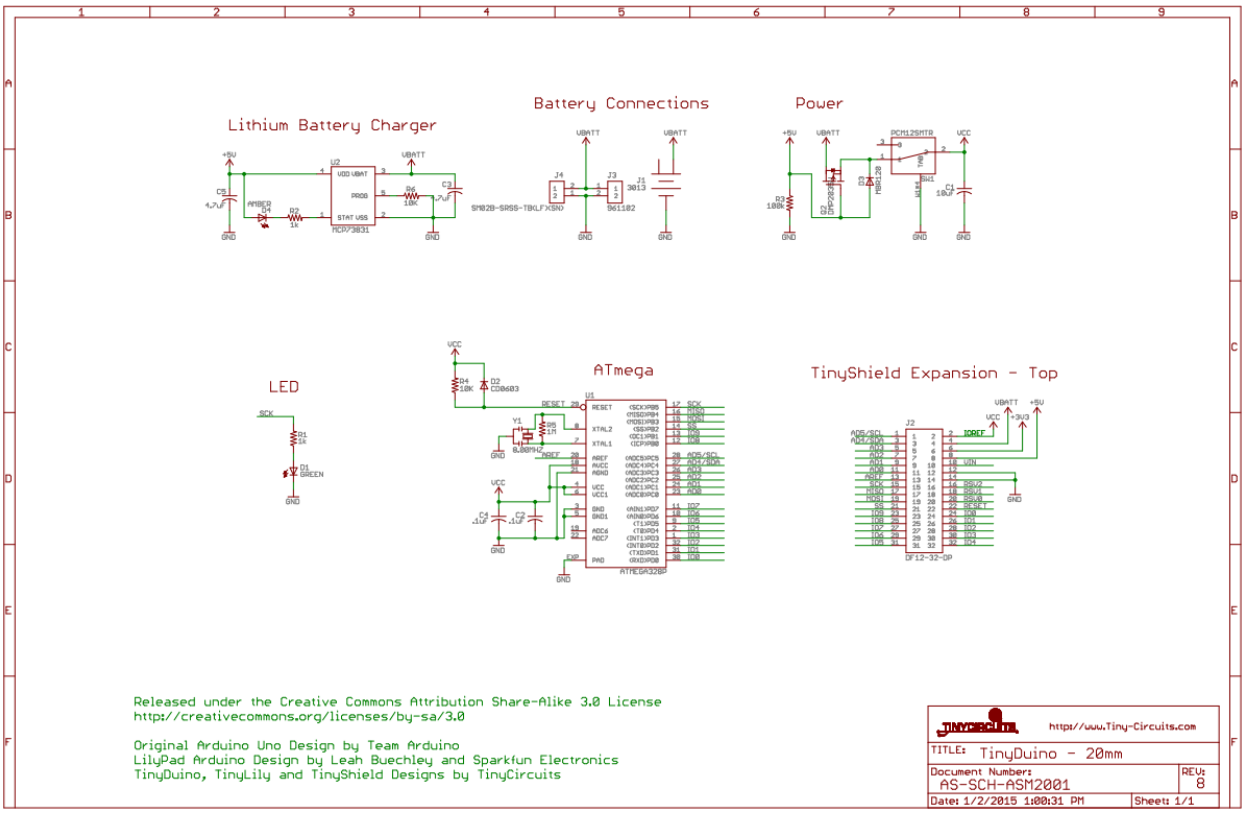



**Figure 16. TinyDuino Schematic**

### 3.6.3 Bluetooth Module

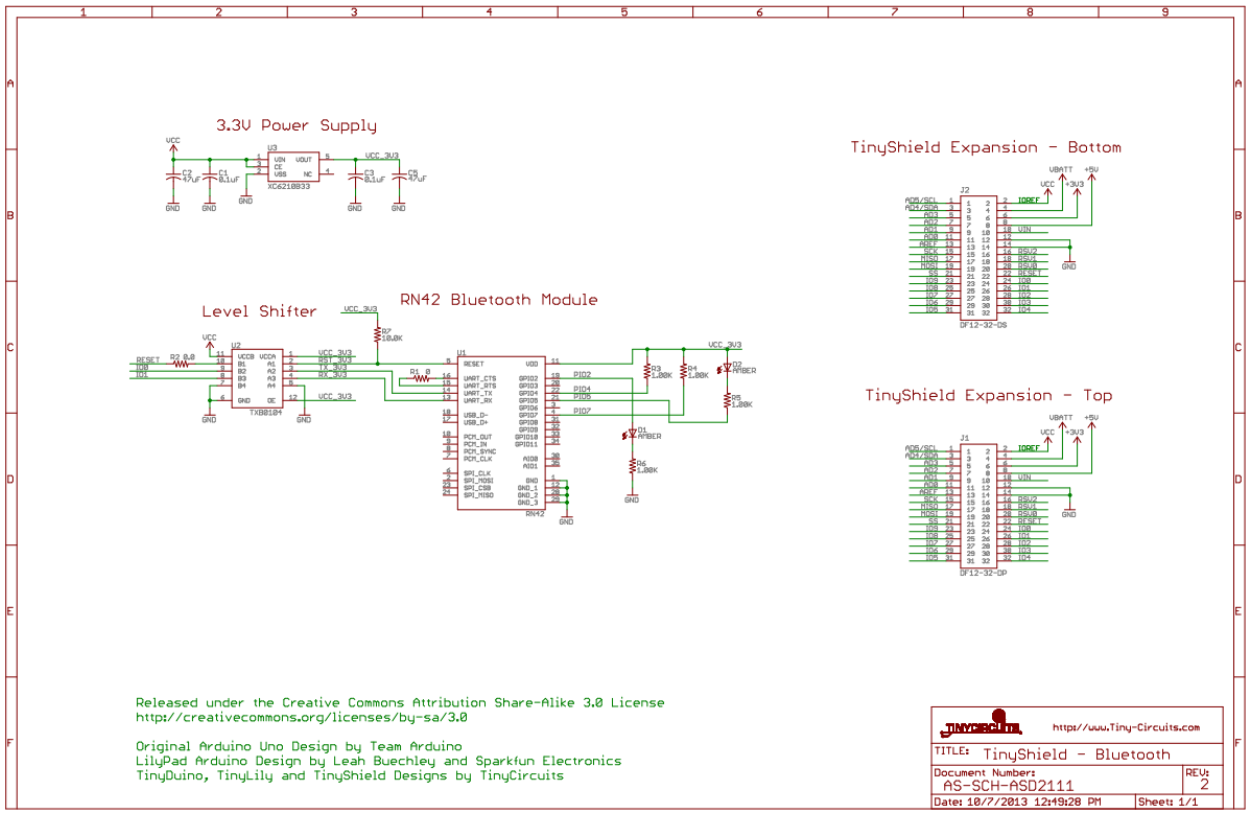



**Figure 17. Bluetooth Module Schematic**

### 3.6.4 TinyScreen

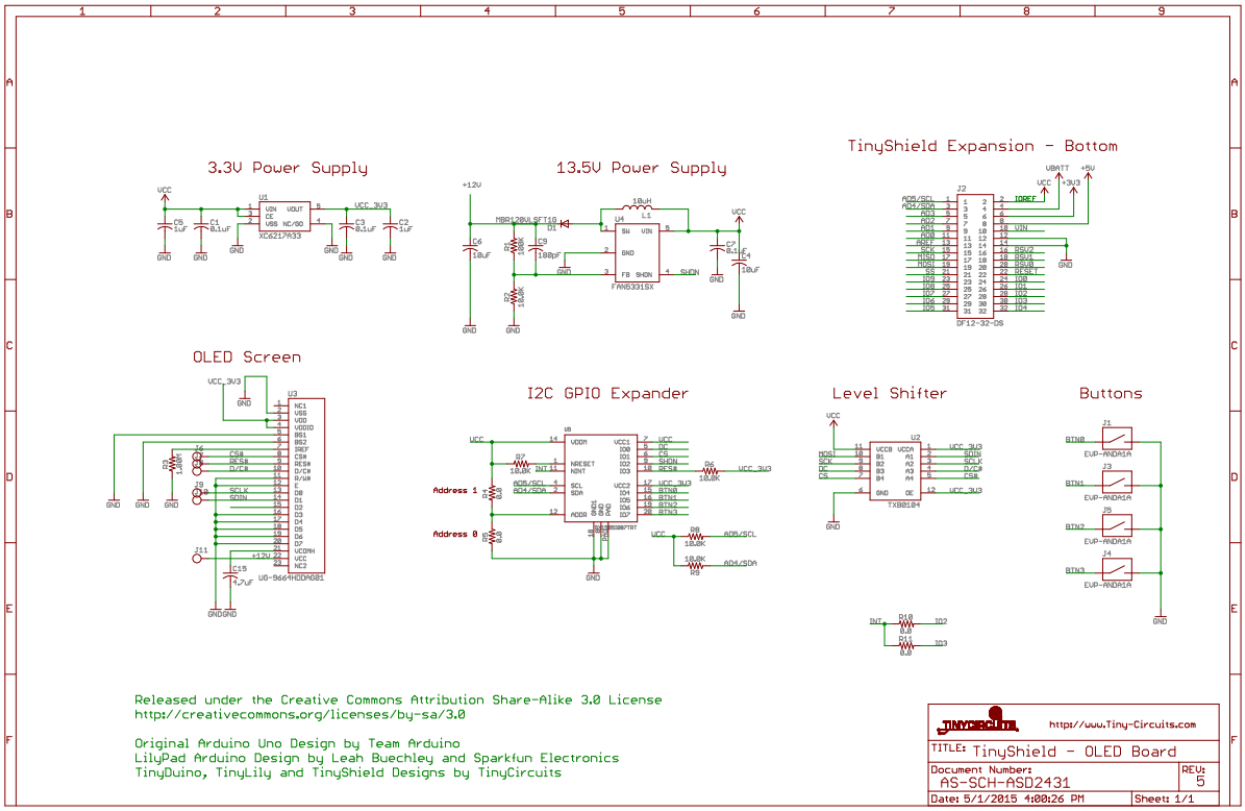



**Figure 18. TinyScreen Schematic**

### 3.6.5 TinyGyro

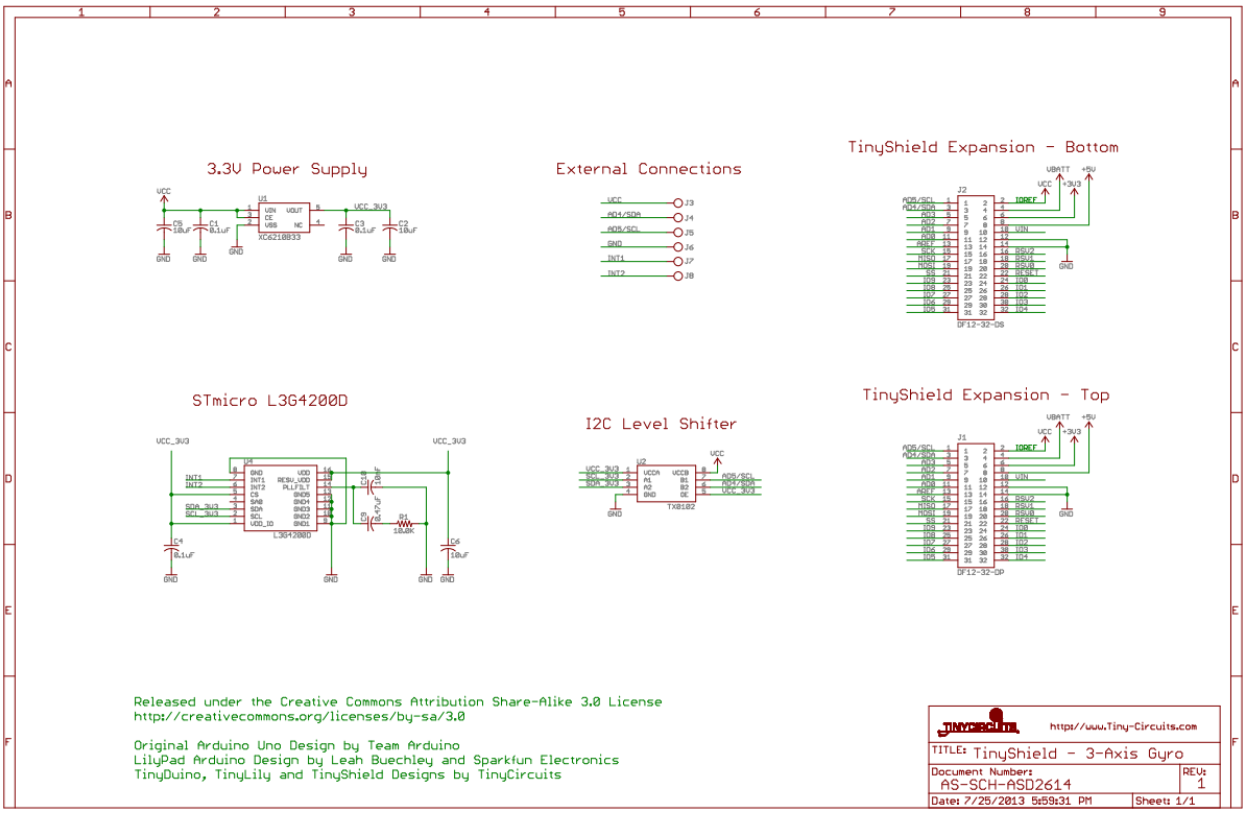



**Figure 19. TinyGyro Schematic**

### 3.6.6 TinyShield Protoboard

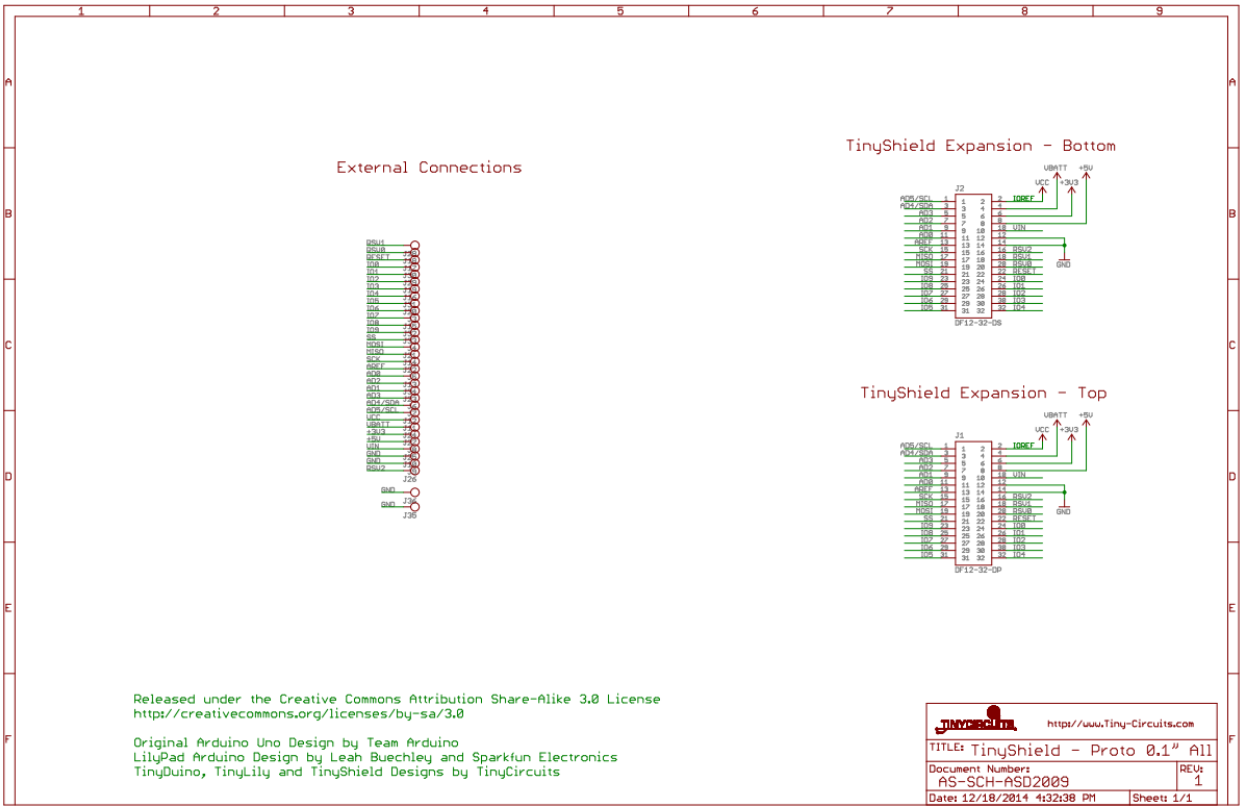



**Figure 20. TinyShield Protoboard Schematic**

### 3.6.7 TinyShield Audio

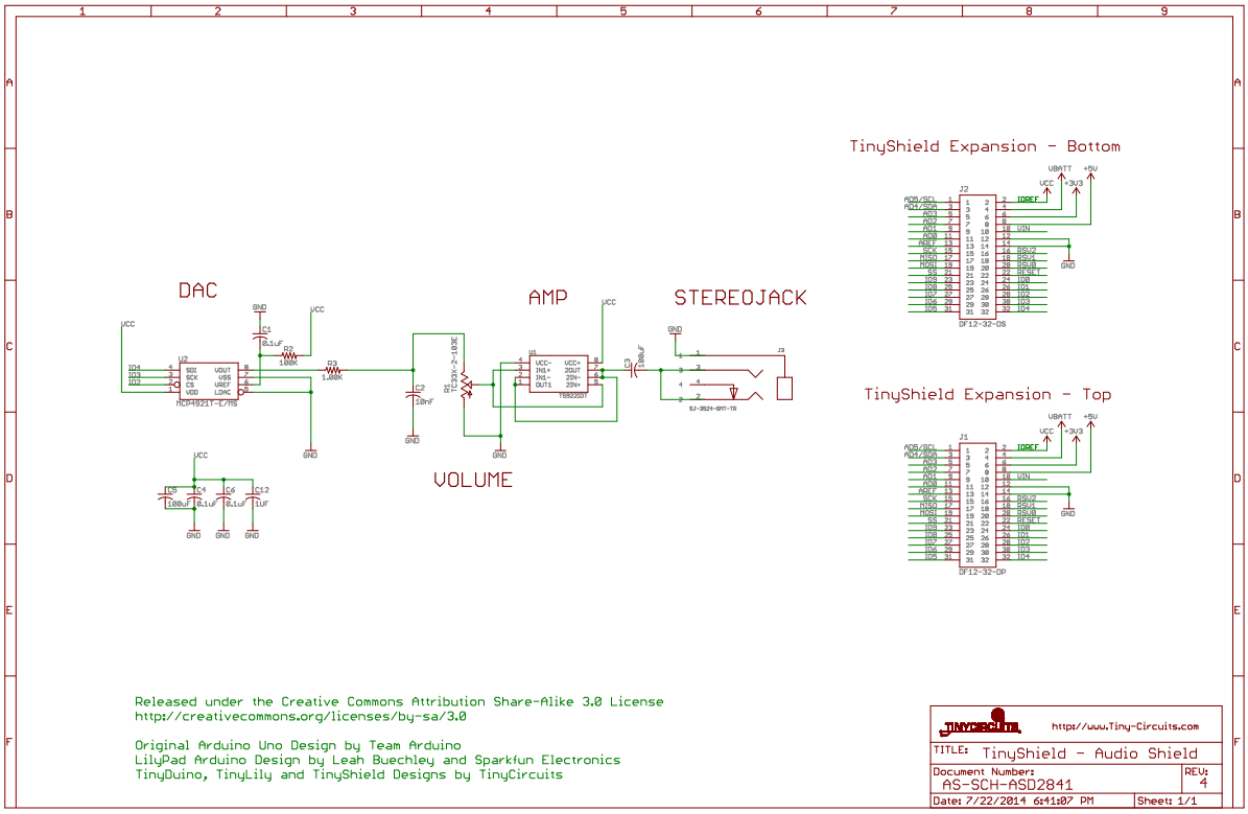



**Figure 21. TinyShield Audio Schematic**

### 3.6.8 External Speaker

2403 263 00077

**16x4.7 mm MFD16** **SPEAKER**

**Features:**

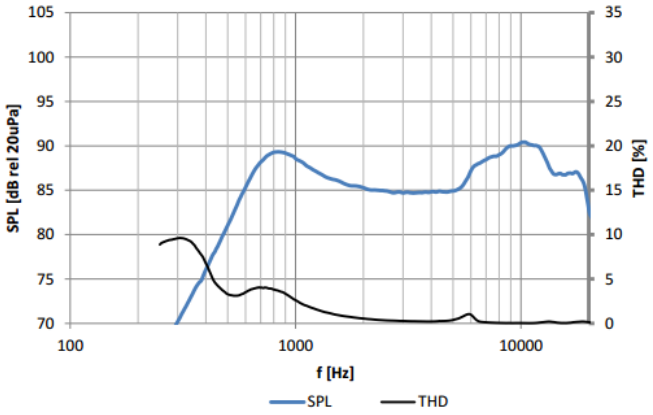
- Built-in Vibrator saves cost for separate vibrator component
- Usage as speaker/receiver/vibrator or speaker/vibrator
- Spring contacts for firm connection

**Specifications:**

| | | |
|---|---|---|
| Impedance | 8 | Ohms |
| Sensitivity * | 74 | dB (1W/m) |
| Air pumping capacity | 90 | mm³ |
| Typical backvolume | 2 | cm³ |
| Total volume | 2.93 | cm³ |
| f0 (typical backvolume) | 740 | Hz |
| Nominal power ** | 500 | mW |
| SPL Max *** | 91 | dB |
| Max sine power | 500 | mW |

Footnotes:
*) average value measured in baffle
**) using shaped noise signal according to product data sheet
***) average value measured in baffle in 0,1 meter distance at max sine power, in typical backvolume

**Frequency Response**




measured in IEC baffle at 125mW in 10cm distance with 2ccm back cavity

**Figure 22. External Speaker Data Sheet**

### 3.6.9 Voltage Regulator

## Features

- input voltage: 2.5 V – VOUT
- fixed 5 V output with 4% accuracy
- 1.4 A switch allows for input currents up to 1.4 A
- 2 mA typical no-load quiescent current
- integrated over-temperature shutoff
- small size: 0.515″ × 0.32″ × 0.1″ (13 × 8 × 3 mm)

## Typical Efficiency and Output Current

The efficiency of a voltage regulator, defined as (Power out)/(Power in), is an important measure of its performance, especially when battery life or heat are concerns. As shown in the graphs below, this switching regulator typically has an efficiency of 80 to 90%.

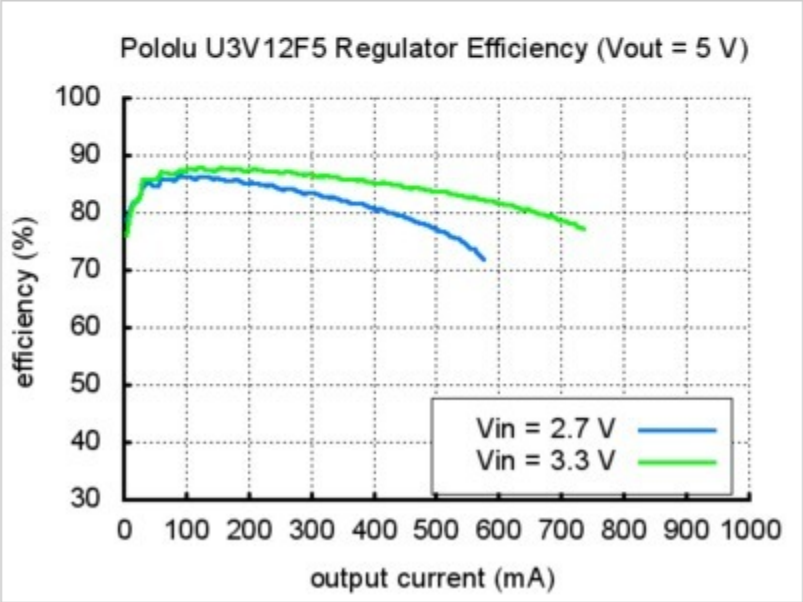



**Figure 23. Voltage Regulator Data Sheet**

[DW, WW]

## 4. Operation Instructions

Step 1. The user must charge the battery using a compatible Li-Ion battery charger.

Step 2. Turn on the Crash Detection/Prevention System using the switch on the back of the helmet.

Step 3. Turn on the Bluetooth on the user's phone.

Step 4. Open the Crash Detection/Prevention System mobile application.

Step 5. Connect to the system using the connect button within the application.

The application will now send data from Mapquest on the phone to the heads-up display of the system. The application will also send speed to the display. If there is an object in the user's blind spot, an LED on the corresponding side will emit light.

Step 6. In the event of a crash, the display will show a countdown. The user may terminate further action in this period by turning the system off with the switch on the rear of the helmet.

In the event of a crash, if the user does not prevent further action, when the countdown is over the helmet will emit light and sound. The user's phone will call and text 911. The text will include the GPS coordinates of the user.

Step 7. Turn off the system by moving the switch on the back of the helmet to the off position.

[JD]

## 5. Testing Procedure

The testing procedures discussed are broken down into the following subsystems audio, blind spot detection, power and miscellaneous.

First we tested the speaker by plugging the aux cable attached to the speaker into our phone. After hearing that this worked we decided that we needed to test it while connected to a microprocessor. We then loaded an arduino audio library onto one of the microprocessors and stored an audio file on the micro SD card. After this we powered the microprocessor and played the audio file through the speaker successfully.

For the power system we connected the regulator to the battery and read the output voltage with a multimeter. The output voltage of the regulator was 5V which is what the microprocessor requires. Since we were successfully outputting 5V we connected the regulator to a microprocessor and it turned on with no problems. We further tested this by powering three microprocessors at once and then adding individual subsystems and power issues never occurred. Finally, we added a button to the power subsystem and it still worked as intended.

The blind spot subsystem was probably the most difficult for testing purposes. We started out by testing the individual components of the subsystem. First we began with the gyroscope, it was able to record the data we needed (roll), but it was too slow. Since the gyroscope was reading slowly, the microprocessor would have incorrect values for the helmet's orientation. So, we switched to an IMU which has very fast roll readings and this fixed our problem. Next, the ultrasonic range sensors were tested individually by being connected to a microprocessor. Our first set of range sensors we tested were working except that they would freeze randomly at times. The next set of range sensors we tried worked much better. We loaded the arduino ping library onto the processor and set the trigger distance to 100cm. After this, we had one person hold the sensor at another person and we measured the actual trigger distance for the range sensor to activate. All of the sensors were very accurate within 1-2cm of the desired 100cm. Then, we tried two range sensors at once because that is the most that should be turned on. Both range sensors worked correctly except when changing to another set of two sensors there was a negligible delay. Finally, we attached all four of the range sensors and the IMU to a processor and began testing. When the helmet was rolled left past our threshold it would activate two range sensors and like so for the right side. After fixing a few errors in our code the blind spot system was working and the corresponding LEDs for the blind spot detection were lighting up when they should.

For the TinyScreen testing we began with loading the example code provided with the screen. This code provided a small game that could be played on the screen. After achieving this, we attempted sending simple string messages from our phone to the TinyScreen via Bluetooth and the microprocessor. Our first attempt did not work and it was because we were not able to establish a connection between the Bluetooth module and our phone. After learning the Java code for our android application we were able to send and receive data to the Bluetooth module. Finally, the string was being displayed on the TinyScreen from our phone. After further

development with the application we were able to send directions and the user's speed from Mapquest.

The last component to test was the impact sensors. After connecting one of them to the microprocessor we were able to record the amount of pressure applied to the impact sensor by pressing our finger on it against a hard surface. We then set a low threshold in the code that would activate our external LED strips when the impact sensors read above this threshold. Finally, we pressed on the impact sensor and the reading rose above our threshold and the external LED strips illuminated. We learned from testing that our sensors had to be pressed for them to work. This meant that for our demonstration we had to move the impact sensors to the outside of the helmet so that we could press them against the helmet. If we did not do this then someone's head would be required to be in the helmet for the impact sensor to register correctly.

Since we tested our subsystems on three different microprocessors our final test was quite easy. We connected all three of the microprocessors with their corresponding subsystem's components to the power supply. Then, we confirmed that each of these subsystems maintained desired functionality. Which they all correctly because they had very minor communication with each other.

[BF]

## 6. Budget Information

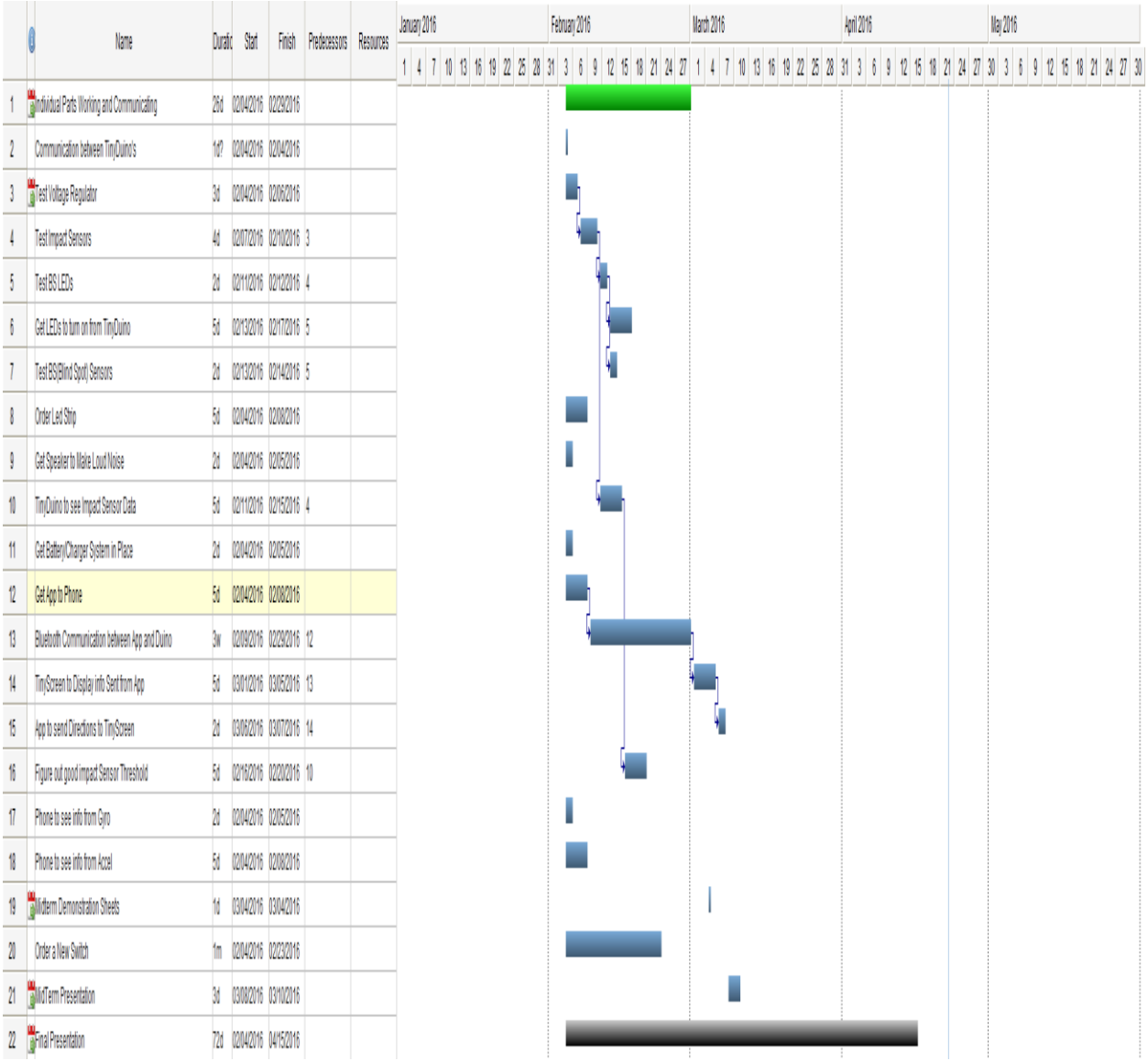
### Table 25. Budget Breakdown

| Qty. | Description | Cost Per Item | Cost | Qty. | Description | Cost Per Item | Cost |
|---|---|---|---|---|---|---|---|
| 1 | Arduino Processor | 15.95 | 15.95 | 2 | 50 CM LED Strip | 8.69 | 17.38 |
| 1 | Arduino I/O Board | 2.95 | 2.95 | 1 | External Speaker | 1.57 | 1.57 |
| 1 | Arduino Audio Board | 19.95 | 19.95 | 1 | Speaker wire to 3.5 mm | 2.8 | 2.8 |
| 1 | System Battery | 13.99 | 13.99 | 4 | Range Sensors | 3 | 12 |
| 1 | Power Switch | 3.06 | 3.06 | 1 | Arduino Processor | 15.95 | $15.95 |
| 2 | Impact Sensor | 6.95 | 13.9 | 1 | Storage reader for system | 14.95 | 14.95 |
| 2 | Resistors for Impact Sensors | 0.1 | 0.2 | 1 | Storage for system | 9.95 | 9.95 |
| 2 | Blind Spot LED | 0.58 | 1.16 | 1 | System Battery | 13.99 | 13.99 |
| 2 | Resistors for Blind Spot LEDs | 0.14 | 0.28 | 1 | Programmer for TinyDuino | 14.95 | 14.95 |
| 1 | Voltage Regulator | 2.16 | 2.16 | 1 | Battery Charger | 19.99 | 19.99 |
| 1 | 121 Ohm Resistor for Regulator | 0.15 | 0.15 | 1 | speaker | 6.13 | 6.13 |
| 1 | 365 Ohm Resistor for Regulator | 0.2 | 0.2 | 4 | Range Sensor | 26.95 | 107.8 |
| 2 | Capacitor for Regulator | 0.55 | 1.1 | 3 | ABS Plastic for Enclosure | 3.85 | 11.55 |
| | | | | | | | 324.06 |

The project was given a budget of $400.00 and it came in at $324.06 which was under budget. The biggest helper for this project coming in under budget was Tiny Circuits donating many parts to the project.

[WW]

## 7. Project Schedule



| | Name | Duration | Start | Finish | Predecessors | Resources |
|---|---|---|---|---|---|---|
| 1 | Individual Parts Working and Communicating | 26d | 02/04/2016 | 02/29/2016 | | |
| 2 | Communication between TinyDuino's | 1d? | 02/04/2016 | 02/04/2016 | | |
| 3 | Test Voltage Regulator | 3d | 02/04/2016 | 02/08/2016 | | |
| 4 | Test Impact Sensors | 4d | 02/07/2016 | 02/10/2016 | 3 | |
| 5 | Test BS LEDs | 2d | 02/11/2016 | 02/12/2016 | 4 | |
| 6 | Get LEDs to turn on from TinyDuino | 5d | 02/13/2016 | 02/17/2016 | 5 | |
| 7 | Test BS/Blind Spot Sensors | 2d | 02/13/2016 | 02/14/2016 | 5 | |
| 8 | Order Led Strip | 5d | 02/04/2016 | 02/08/2016 | | |
| 9 | Get Speaker to Make Loud Noise | 2d | 02/04/2016 | 02/05/2016 | | |
| 10 | TinyDuino to see Impact Sensor Data | 5d | 02/11/2016 | 02/15/2016 | 4 | |
| 11 | Get Battery/Charger System in Place | 2d | 02/04/2016 | 02/05/2016 | | |
| 12 | Get App to Phone | 5d | 02/04/2016 | 02/08/2016 | | |
| 13 | Bluetooth Communication between App and Duino | 3w | 02/09/2016 | 02/29/2016 | 12 | |
| 14 | TinyScreen to Display info Sent from App | 5d | 03/01/2016 | 03/05/2016 | 13 | |
| 15 | App to send Directions to TinyScreen | 2d | 03/06/2016 | 03/07/2016 | 14 | |
| 16 | Figure out good impact Sensor Threshold | 5d | 02/16/2016 | 02/20/2016 | 10 | |
| 17 | Phone to see info from Gyro | 2d | 02/04/2016 | 02/05/2016 | | |
| 18 | Phone to see info from Accel | 5d | 02/04/2016 | 02/08/2016 | | |
| 19 | Midterm Demonstration Sheets | 1d | 03/04/2016 | 03/04/2016 | | |
| 20 | Order a New Switch | 1m | 02/04/2016 | 02/23/2016 | | |
| 21 | MidTerm Presentation | 3d | 03/08/2016 | 03/10/2016 | | |
| 22 | Final Presentation | 72d | 02/04/2016 | 04/15/2016 | | |



**Figure 24. Gantt Chart**

[WW]

## 8. Design Team Information

Joseph Douglas - EE, Hardware Manager

Brandon Flint - CpE, Software Manager

Wesley Waters - EE, Team Manager

David Witsaman - CpE, Archivist

[JD,BF]

## 9. Conclusions and Recommendations

Throughout the process of building the project some aspects of the project were found to have better solutions. Some of these changes were made, but some were not made because of time/money constraints. Some examples of things that were changed was the addition of a 3rd processor from the original design of only having two, the addition of an IMU, upgraded Ultrasonic Sensors, and a different LED strip than originally spec'd out. A couple of additions or upgrades that would be made if the project were to be made into an actual product for the market would be custom circuit boards instead of TinyCircuits, the addition of a fail safe in the case of a loss of connection between the circuit and the phone, and another fail safe using the accelerometer on the phone, and more time into a better amp for the speaker.

During the integration process some issues came up with playing the sound file through the speaker. The serial out to the screen was showing that we were getting to the line where it would execute the playing of the file, but it was not playing. It was then discovered that our board only had 24 bytes of dynamic ram left while executing the program. This was ultimately found to be because the library for audio compatibility with the tinyDuino took up 71% of the available ram. There was no way other than completely stripping the library of all unnecessary functions which we did not have enough time to do. This led to the decision to add in a third processor. The third processor has one input that told it to play the audio file. This could have been avoided if custom boards were designed.

During the beginning of integration issues were coming up with using a gyro to determine the angle of the helmet, which was being used to determine which ultrasonic sensors were active. The issue was the processing power of the tinyDuino. To keep track of the angle of the helmet the processor was performing integration on the data from the gyro which worked when that was the only thing that the processor had to do, but when it was integrated with other systems the processor could not integrate the data fast enough to keep an accurate angle. To remedy this situation an IMU(a 9-axis sensor) was added as a replacement for the gyro. Once this was added the processor was able to work with the other subsystems while keeping track of the angle of the helmet.

The ultrasonic sensors that were initially used had issues with consistency. They would work most of the time, but would inevitable start sending corrupt data that either would not make sense, or they would just randomly stop sending data. It was researched and determined that better ultrasonic sensors and would remedy the situation. The upgraded sensors also had a smaller profile than the old sensors making the final product look better.

One week before the presentation Google changed the way that they send directions to the notifications bar which were originally being scraped to get directions to the user's destination. Because of this mapQuest was determined to be a good alternative. It used similar notifications as Google Maps before the update, and it is still a commonly used directions app.

Some ways that the Motorcycle Helmet Crash Detection/Prevention System could be improved in the future would be the addition of some safety fail safes. In the case of a crash that causes the phone to disconnect from the system a phone call to 911 should still be made. The best solution that has been determined to date is having the phone monitor it's accelerometer data and in the case of a disconnection to check the last few second of data from the accelerometer for a large

deceleration. If those two events occur together then have the phone determine a crash and call 911. It was also determined that the use of the accelerometer data from the phone would be a good check to help determine the event of a crash even when no disconnection occurred. This was not implemented because of a time constraint. If the project were to continue then the phone's accelerometer data would be checked against the data from the sensors on the system to help in the detection of a crash.

The speaker on the back of the helmet has never been able to make as loud of a sound as would be desireable. Originally thought to be a speaker issue, another speaker was ordered. This did not solve the problem. The speaker however was about 4x as loud, still not as loud as desired, if the plug was only plugged in halfway. This however left the plug very vulnerable to falling out. The plug on the circuit was then taken apart and the speaker was just soldered to the board.

The force sensors that were used were determined to be ineffective. A better alternative would have been to use a vibration sensor or if the helmet could be custom built with a backing for the force sensors. The issue with the sensors was with placement. We didn't want to destroy the foam on the inside of the helmet because that would make the helmet ineffective, but to properly use the sensors that we had a consistent pressure from both sides would be needed. If the sensors could have been placed between the foam that protects the wearers head and the plastic on the outside of the helmet these sensors may have worked better. However this was impractical with the tools, time, and money that we had to have a helmet made this way.

The biggest change that would be made going forward with the project would be the use of a better processor instead of using three separate processors and custom designed boards. The use of a custom designed board would have alleviated many of the issues that we came across such as the lack of enough memory to run our program. The use of a stronger processor could have eliminated the need for an IMU as well as allowing easy use of the IMU in multiple sub systems. Currently the IMU is only used for determining the angle of the helmet for the blind spot detection. If one processor was being used it would have been much easier to use the data from the IMU in conjunction with the other crash detection sensors to have more accurate crash detection.

[WW]

### 9.1 Satisfying the Design Requirements

**Table 26. Marketing Requirements/Implementation**

| Marketing Requirement | Implementation |
|---|---|
| The system must detect impact classifiable as a crash. | The system has impact sensors, but does not detect a crash based on the phone's accelerometer due to time constraints. |
| On detection of a crash, the user will be afforded a thirty second grace period in which they can terminate further action by pressing the on/off button, which will power off the system before further action is taken. | The user can use the switch to turn the system off and cancel the call. The microprocessor will countdown the time until further action is taken. |

| | |
|---|---|
| During the thirty second grace period the driver will be notified. | The HUD displays the countdown to notify the user of the time left before further action is taken. The blind spot LEDs will also blink to alert them that a crash has been detected. |
| On detection of a crash, after the grace period, the system will communicate with the user's phone in order to initiate a communication to emergency response conveying a crash has occurred, and providing the GPS coordinates of the user's phone to emergency response. | The microprocessor communicates to the phone application through Bluetooth informing the application that the countdown has completed and further action should be taken. |
| On detection of a crash, after the grace period, the system will emit lights and sounds separate from the grace period sounds. | After the microprocessor finishes the countdown it will then sound the speaker alarm and flash the LED strips on top of the helmet. |
| The system must detect a vehicle in the user's "blind spot" and communicate through the microprocessor, emitting a light on the side of the user in which the vehicle is located. | When an object is within 4 feet of an active sensor based on the IMU data the microprocessor tells the corresponding LED to turn on. |
| The system's microprocessor will communicate to the HUD the speed that the user's phone is traveling, based upon the phone's GPS movement. | The phone's application calculates speed based on the phone's GPS and sends it to the microprocessor through the Bluetooth module. |
| The system's microprocessor will communicate with the user's phone and will relay turn-by-turn directions supplied by an application from the user's phone to the HUD, showing an arrow to next change in direction as well as distance to the change. | The phone's application will send turn by turn directions to the microprocessor through the Bluetooth module. The HUD will then display the information sent from the microprocessor. |
| The system should sustain power for at least six hours on a full charge. | Based on the power needs of the system the battery was chosen to meet the 6 hour need. |
| The system should be usable on Android 4.1 and newer phones. | The application will be programmed using commands that work with all devices with a minimum requirement of 4.1+ |
| The system will be waterproof. | Many parts were specifically chosen to be waterproof and those that are not will be contained within a waterproof housing. Due to time constraints as well as cost constraints, the system is not waterproof. |
| Weight should not increase by more than 10% of helmet weight. | The parts were chosen with weight in mind. The weight came out to 10% of a slightly above average helmet. |

[JD, BF, DW, WW]

## 10. References

"Edge-based Lane Change Detection and Its Application to Suspicious Driving Behavior Analysis." *IEEE Xplore*. N.p., n.d. Web. 22 Oct. 2015.

"Skully AR-1." *Skully*. N.p., n.d. Web. 22 Oct. 2015. <http://www.skully.com/#smartest-helmet>.

"ICEdot Crash Sensor." *ICEdot*. N.p., n.d. Web. 22 Oct. 2015. <http://site.icedot.org/site/crash-sensor/>.

"A Novel Algorithm for Crash Detection Under General Road Scenes Using Crash Probabilities and an Interactive Multiple Model Particle Filter." *IEEE Xplore*. N.p., n.d. Web. 22 Oct. 2015.

"Patent US20100039216 - Crash Detection System and Method." *Google Books*. N.p., n.d. Web. 22 Oct. 2015.

"Patent US7348895 - Advanced Automobile Accident Detection, Data Recordation and Reporting System." *Google Books*. N.p., n.d. Web. 22 Oct. 2015.

"Wi-Fi Direct vs. Bluetooth 4.0: A Battle for Supremacy." *PCWorld*. N.p., n.d. Web. 22 Oct. 2015.

[WW,DW]