BIKE  TRACKER

Genevieve Heidkamp, Alison O'Connor, and Marisa Thompson

Final Design Document

April 29, 2016

BikeTracker: Table of Contents

1 Introduction

2 Detailed System Requirements

3 Detailed project description

- 3.1 System theory of operation
- 3.2 System Block diagram
- 3.3 Detailed Design and Operation of Wifi Capability
- 3.4 Detailed Design and Operation of GPS Tracking
- 3.5 Detailed Design and Operation of User App Interface
- 3.6 Detailed Design and Operation of Battery Power
- 3.6 Detailed Design and Operation of Accelerometer
- 3.7 Interfaces

4 System Integration Testing

5 Users Manual/Installation manual

6 To-Market Design Changes

7 Conclusions

8 Appendices

1 Introduction

In the United States, it is estimated that between 800,000 and two million bicycles are stolen each year. Many of these thefts occur because cyclists fail to lock their bike or use bike locks that are broken easily. In addition, once a bike is stolen, it is generally difficult to recover, especially in a city or on a college campus, because bikes can be transported so quickly and/or can easily blend in with other bicycles on a bike rack. Only about 5% of bicycles that are stolen are returned to their owners. [1]

On college campuses across the nation, these issues lead students to either be constantly preoccupied with the location of their bikes or be hesitant to buy a bike in the first place. Why? Because of the high likelihood that their bikes could be stolen, even while using bike locks or storing their bikes in inconspicuous locations. In order to eradicate these fears, it would be beneficial to many college students if a GPS device could attach to their bike to help them locate their bike in case of theft. Not only would this help them find their stolen bike, but it could also alert them at the time of the robbery so they could take action if desired. This concern led to the creation of BikeTracker.

In order to prevent bicycle theft and help more bikes be returned to their owners, we are proposing to make a smart bicycle finder. This product we are proposing is specifically targeted towards college campuses where wifi is prevalent and will be able to utilize a GPS signal.

[1] http://www.bicyclelaw.com/p.cfm/bicycle-safety/about-bike-theft

The following is a final document which outlines the project design of this BikeTracker including system requirements, design specifications, and detailed product description. It outlines the problem statement, our proposed solution, the overall system requirements, detailed subsystem explanations, integrated system testing, a user's manual, our eventual design changes for a to-market application, and our conclusions.

First, the bike finder includes a GPS device that, when connected with a smartphone via a mobile application, can inform a user where their bike is located at any given time.The user application is interfaced with the device in order to connect with the bike. The device itself appears to be a reflector installed on the back of the bike seat so as to remain inconspicuous. Also included in the design is an accelerometer that can track movement. When the device is set in “Tracker On” mode, meaning the user is anticipating the bike to be parked and not in use, and there is a movement detected, the application will be triggered to notify the user. Finally, a battery will be used to power the device. A battery will be used in order to keep the tracker in a discrete location on the bike. The battery will power the device with specific considerations taken so that battery need not be replaced too often.

As the design process for this BikeTracker went on, we were able to meet our planned requirements and use hardware planned in order to get a functioning prototype. At the current stage of development, we have created a prototype that can be seen below.



(a) (b) (c)

Pictures: (a) Full prototype with reflector masking, (b) Board with battery set up in casing, (c) Prototype installed underneath bike seat

The prototype we created is fully functional and able to communicate with a mobile user application in order to provide GPS location to the user when requested as well as notify the user when an unexpected movement occurs. At the onset of this project, we anticipated creating a working prototype and were successful. It gives a largely accurate location of the board when necessary and functions well with a very user-friendly mobile application. While we did have trouble in the final development stages as we integrated all of the subsystems together, we were able to work as a team to generate solutions and overall had a positive experience. The main source of testing was trial and error in order to work out any random trips in the coding, but most issues were solved without too much manipulation. From reaching this stage in our prototype,

we noticed improvements to be made in the future which are highlighted in section 6 of this document.

2 Detailed System Requirements

The development of the BikeTracker required a large amount of system requirements in order to have the desired full functionality .

The primary requirement of the system was an ability to interpret GPS data through NMEA sentences acquired from the GPS coordinate device. These sentences then needed to be parsed so as to extract latitude and longitude. This was done through the use of the GPS RMC sentence. This data then needed to be sent through the ESP12 device by the way of a WiFi network which a mobile application also subscribes to and can access. This was done using the MQTTlens application. Along with this data, the system also needed to be powered by a rechargeable battery so as to have full range on a bicycle which does not have access to power at all times.

The device needed to support GPS, battery, and ESP12 components together in order to function and communicate properly. The range of this device with all these components needed to be the approximate size of a large college campus while communicating with large campus WiFi so that it can be used among the targeted customer. This was achieved via access points in the WiFi networks such as ND-guest.

The device required to have a user interface in a mobile application format which shows the GPS location of the BikeTracker at the last time a measurement was taken. This application must contain modes so that if the location of the bike changes while in “parked” mode, an alert is sent through the application to notify the user. This was done through “Tracker On” and

“Tracker Off” modes. When Tracker on, any activity will trigger the device to get a GPS location as well as a set timed interval location. This activity is triggered through an ADXL345 accelerometer. The second interrupt from this device triggers based upon activity and signals a OneShot to send 1 high clock cycle to an n-MOSFET. This pulls the CH_PD pin low for a moment to reset the device and start getting and GPS location device. Tracker Off mode stops these signals from being transmitted in order to save battery power.

In order to connect the specific BikeTracker to the user in the mobile application, the device need to utilize an MQTT protocol which can communicate with the user and the hardware for the BikeTracker on the bicycle. MQTTbike/+ was the channel used to subscribe and publish information between the hardware device and mobile application.

The device, being that it is a tracker and needs to stay inconspicuous, needed to be installed under the bike seat and disguised as a reflector light. Once the battery is connected to the outer casing, the device functions as both a reflector and a tracker. In form with this, the device needed to be an approximate size and weight that would be able to hide behind panel which was found to be 3 in by 2 in with depth of about 1 inch.

3 Detailed Project Description

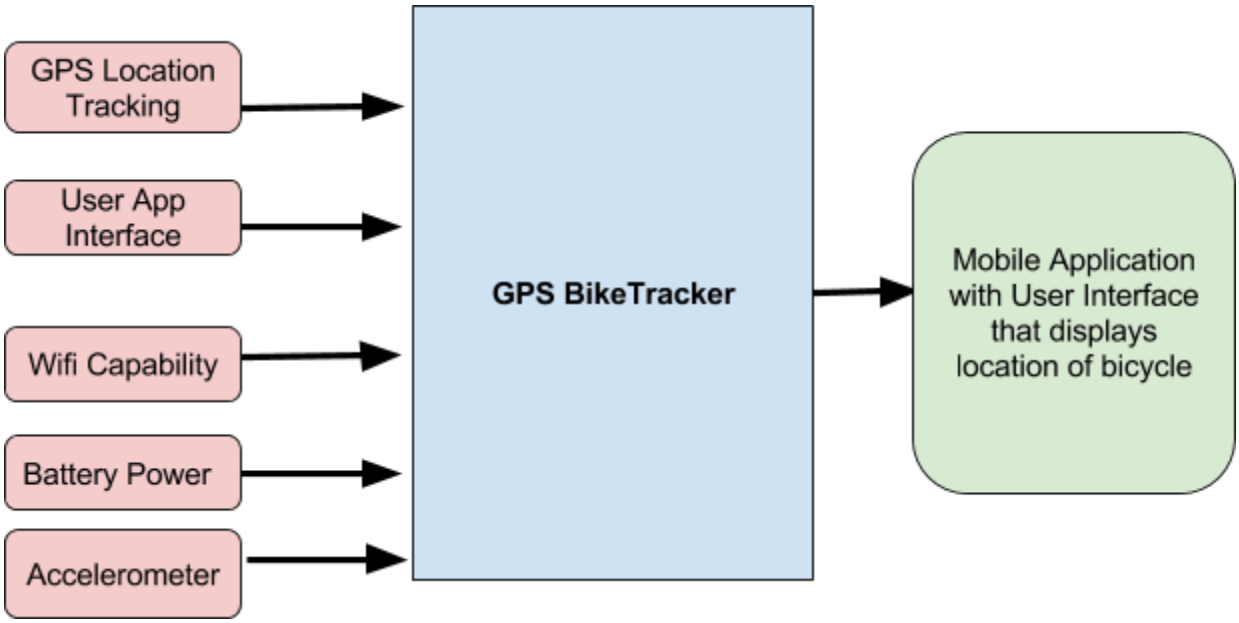
3.1 System theory of operation

The entire system works as a whole to provide a user with the location of their bicycle when unknown. This starts with a GPS hardware device that can pick up satellite info in order to give exact latitude and longitude of the board. This data is then sent through an ESP12 WiFi device which communicates with an MQTT protocol server to transmit the NMEA sentence data. This data is then obtained by an iOS user application that subscribes to the same MQTT topic.

This application has modes that can request the location of the bicycle at any given time and can communicate with the user when there is undesired movement.

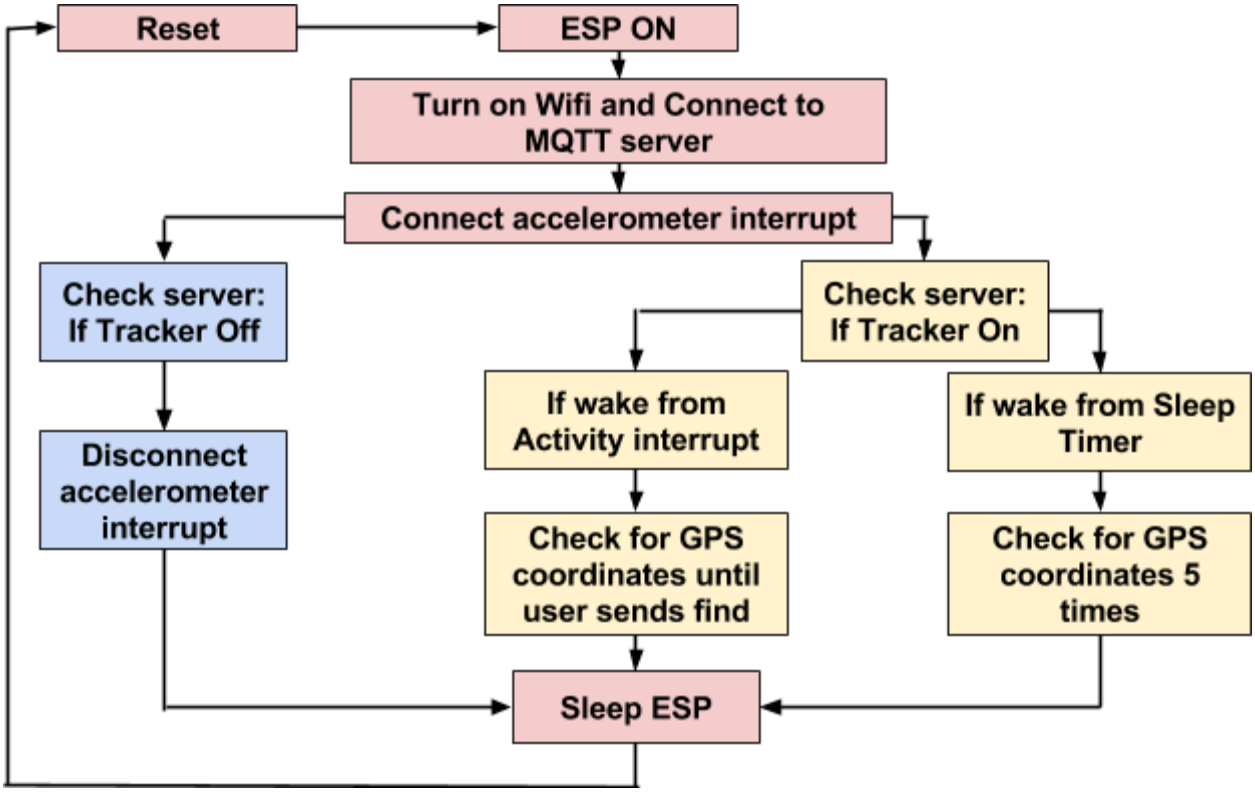
3.2 System Block diagram

The system requirements must be met by the system as a whole. These requirements are met through 5 subsystems: GPS location tracking, user app interface, WiFi capability,battery power, and accelerometer.

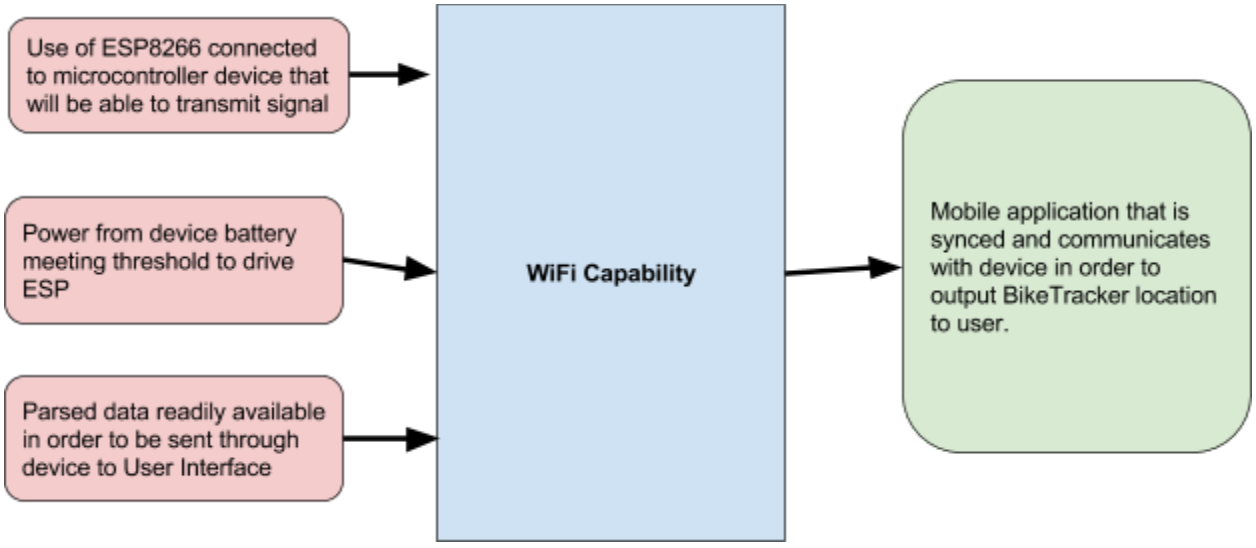


These five subsystems divide the project into manageable segments that allow for a gradual development of the project with consideration to testing phases until the product is fully functional. The interfaces between these are the User application with the MQTT protocol and WiFi network which allows the communication of the device.

The following diagram outlines the system modes that are controlled through the User Application. This flow demonstrates the functionalities through the device and user application.



3.3 Detailed Design and Operation of Wifi Capability



The WiFi Capabilities of this device rely upon the ESP12 to be connect to the microprocessor while being able to utilize the MQTT protocol developed on Amazon server in order to communicate with the User Application. Power from the battery drives the device and

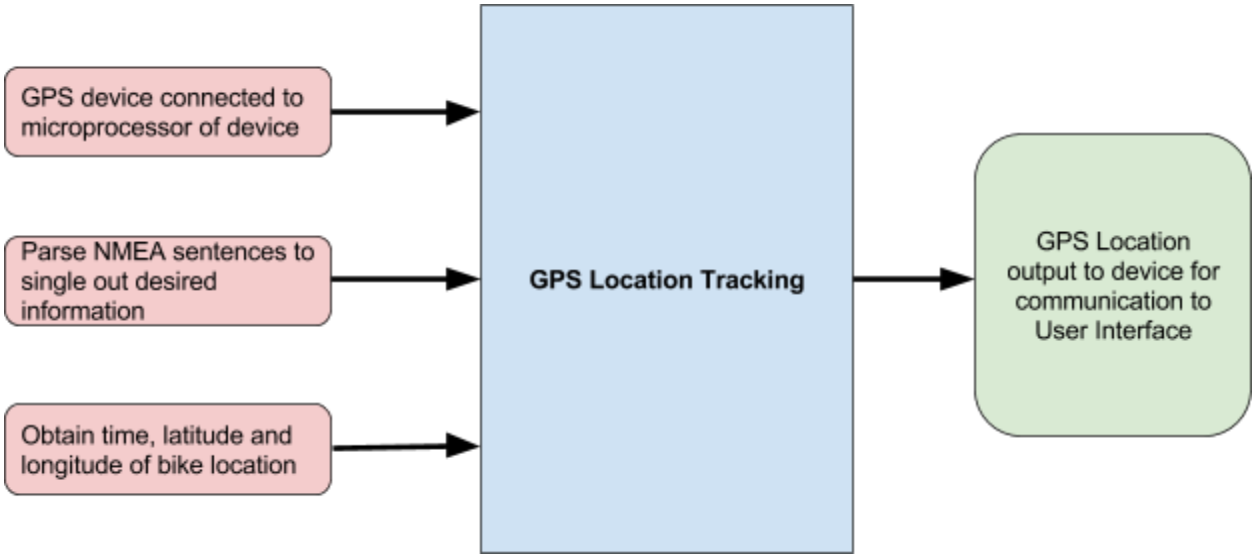
allows for the sleep/wake modes to occur on specified intervals. The GPS RMC sentence is sent through this capability for full functionality of the application.

The ESP12 device was chosen based on its specifications being close to what was needed on a campus WiFi network such as that of Notre Dame. We were provided with ESP12 data as well as code to help integrate the system together within the MQTT application through the Arduino environment. The network was set up through a router within the Senior Design classroom and Amazon server and we relied upon provided instructions to connect to it. Using the MQTTlens application, we set up a topic called bikeMQTT/+ which can be subscribed and published to by the ESP12 as well as the user application.

The code works by first setting the baud rate and connecting to the WiFi access point. When first run, a user must select a WiFi network for the device to use. When connected to the WiFi, the ESP12 subscribes to the MQTT protocol until successful. This allows the functionality of both subscribing and publishing to the MQTT topic. Messages can be sent by the user and allow communication through the user application interface. For more detailed explanation of the MQTT interface with the user application, refer to section 3.5.

In order to test the full functionality of this subsystem, we programmed our hardware and software and used bikeMQTT/+ to see the GPS RMC sentence transmitted and published to the window. From here, the application was able to access the latitude and longitude for GPS location, showing WiFi functionality.

3.4 Detailed Design and Operation of GPS Tracking



The GPS location tracking utilizes the Mediatech3329 GTPA010 device to connect with the microprocessor in order to transmit the NMEA sentences to be converted and used in the User Application. The NMEA sentences, specifically the RMC sentence, is parsed in the user application to obtain latitude and longitude. The device acquires data from satellites and sorts the data into sentences. We chose to utilize the RMC sentence due to its straightforward application of only supplying time, latitude, and longitude.

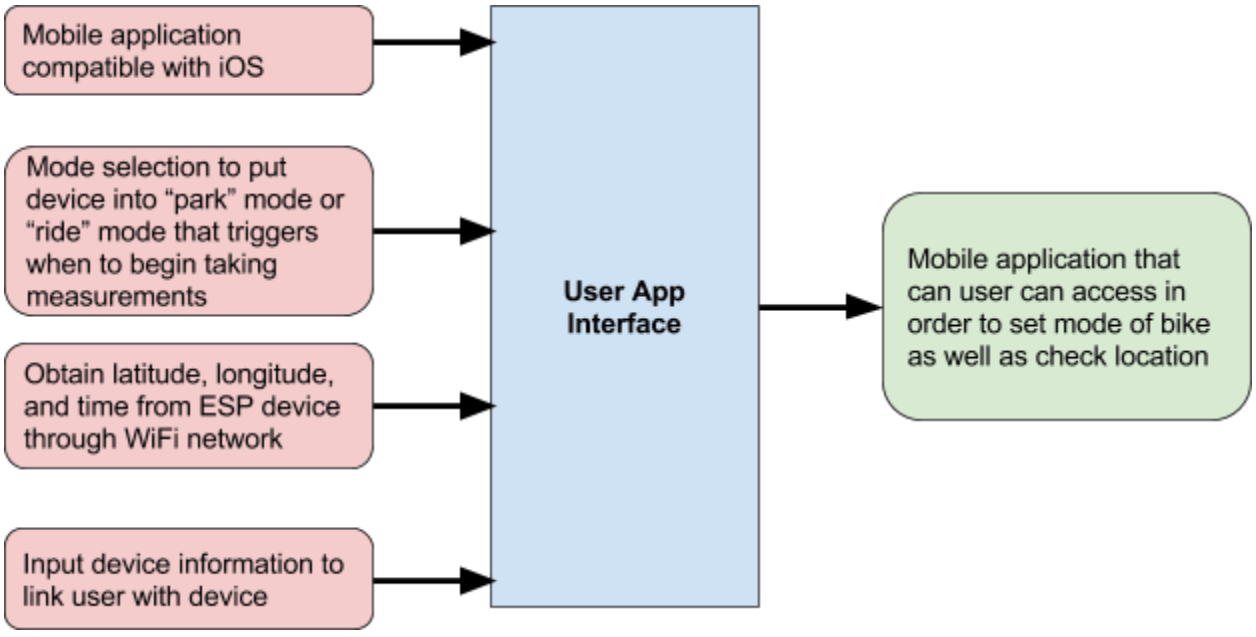
Since serial is the most straightforward way to communicate the data, the serial port is set up with the baud rate at 9600 (as set by the hardware). A PMTK220 sentence is then used to set the frequency at which to collect data, making the serial port ready to transmit data. Due to issues with serial port timing, it became challenging to print the NMEA sentences into the serial monitor. Therefore, a design decision was made to incorporate the GPS code with the MQTT protocol code for direct transmission to the user application.

Using serial.read command, the RMC sentence is obtained and sent into strings which are then published into bikeMQTT/+ topic for transmission to the user application. A more

detailed explanation of how the user application utilizes this data can be seen below in section 3.5.

In order to test the full functionality of this subsystem, we programmed our hardware and software and used bikeMQTT/+ to see the GPS RMC sentence transmitted and published to the window. From here, the application is able to access the latitude and longitude for GPS location. By checking this location with the exact location when entered into Google maps, we were able to confirm a correct location and prove functionality.

3.5 Detailed Design and Operation of User App Interface



The mobile app is an iOS app. It has two different modes: “park” mode and “ride” mode. If the bicycle moves during “park” mode, the app will send an alert to the user. The app will not send an update while in the ride mode. It will have the ability to display the location of the device and the time at which the location information was taken. Finally, the app has the ability to send

a message to the device that the bike has been found. The elements will comprise the user mobile interface.

The app was programmed in XCode, as one of our group members has experience working with the programming, as well as the fact that iPhones are very prevalent on college campuses. The app consists of one view controllers. When first opened, the app connects to the MQTT server and sets up all the MQTT protocols. Specifically, it subscribes to bikeMQTT/+ topic and sets up publishing to the bikeMQTT/inTopic topic and subscribes to the bikeMQTT/outTopic.

On the view controller, there is a map and a Tracker ON/OFF toggle button. A user has the option to press “Tracker On” or “Tracker Off”. When Tracker Off is pressed, the map will not be updated and “Tracker Off” will be published to the MQTT server to alert the device that it should not look for GPS data. Underneath the Tracker ON/OFF toggle button is a Show Last Location ON/OFF toggle button. When ON is selected, the last location the bike has received will be displayed on the map.

When Tracker On is pressed, the app will look for sentences published to the bikeMQTT/+ topic with the prefix “$GPRMC”. This will signal to the app that the GPS has published an RMC sentence. The RMC string is then converted to an array so the information in the string, such as longitude and latitude data, can be interpreted more easily. First, the app checks if the GPS has gotten a satellite fix by checking to see if the 19th character in the array is a “V” or an “A”. If it is a “V”, the GPS has not gotten a fix yet, and a text field will display over the map alerting the user that the GPS is looking for satellites. If it is an “A”, the GPS has gotten a fix and the app will proceed to parse the data.

One tricky part about the parsing is that the GPS data is given in Degrees and Minutes, while the map component of the app requires a decimal longitude and latitude. So, the Degrees

and Minutes are each called out of the array and converted into double form for both the longitude and latitude. Then to determine the decimal form, the equation (decimal = degrees + minutes/60) is used and this information is passed to the map component of the view controller.

As mentioned several times above, another major component of the second view controller is the map. The map has two goals, to show the current location of the user and the location of the bike. First, to determine the user's location, the CLLocationManager manager was used. When incorporated into a function, this manager will help the app determine the location of the user and display it on the map. Next, to determine the location of the GPS device, an annotation was added to the map using addAnnotion options for the MKMapView.

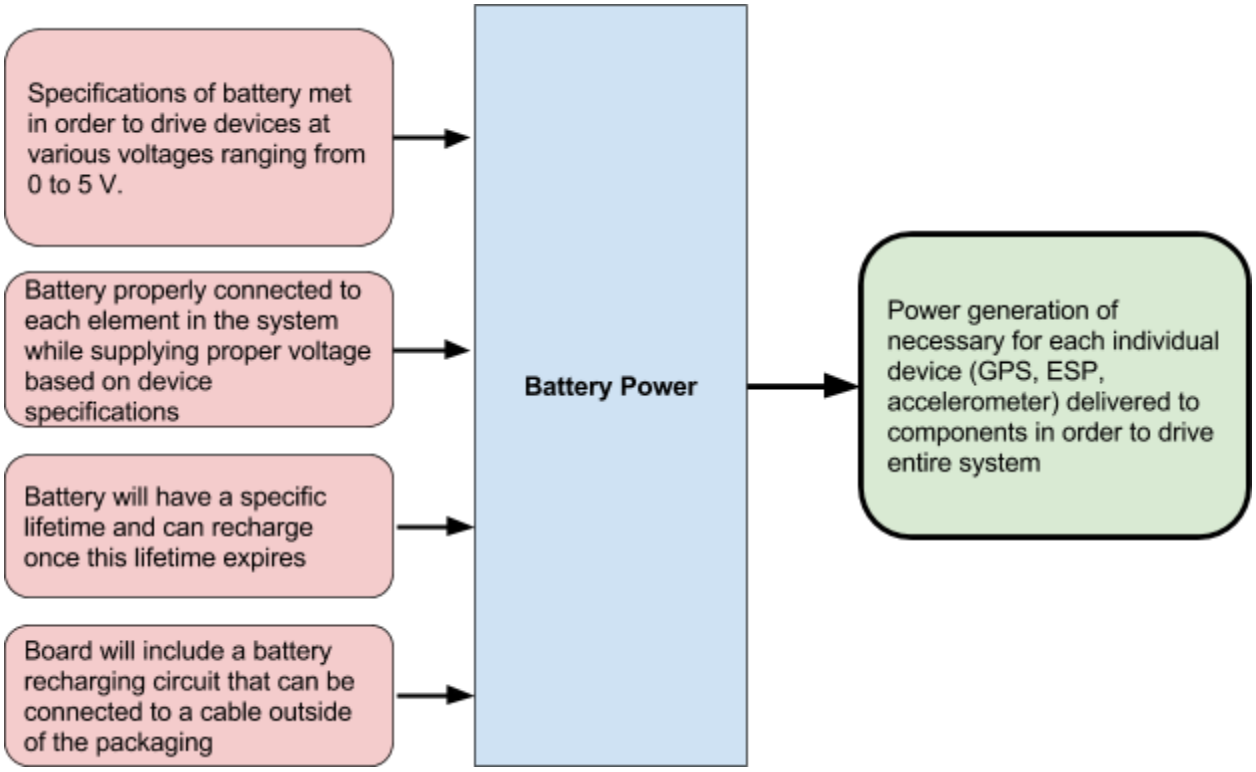
The final aspect of the app is the notification system. When Tracker On is pressed, the app will also look for sentences published to the bikeMQTT/+ topic with the prefix "Activity!". If that sentence is received, the app uses the UIAlertController to alert the user that an activity message sent from the accelerometer is sent to the MQTT server. This alert will appear on the user's lock screen or directly on the app interface. In addition, if an activity alert is received, a sentence is received, a new button will appear on the top of the screen. When the button is pressed, it will publish a message to the MQTT server that will tell the device to stop search for the bike and to restart.

In order to test the full functionality of this subsystem, we programmed our hardware and software and used bikeMQTT/+ to see the GPS RMC sentences transmitted and published to the window. From here, the application was able to access the latitude and longitude for GPS location. By observing the bikeMQTT/+ topic to see the subscriptions and publishings to it, we were able to confirm that the data was being properly transmitted to the user application and the application was able to show GPS location while also changing modes at the user's prompting.

Finally, we were able to check that Activity notifications were received properly by both checking the bikeMQTT/+ topic and seeing that alerts were properly activited.

The full code listing can be found in Appendix B and C.

3.6 Detailed Design and Operation of Battery Power



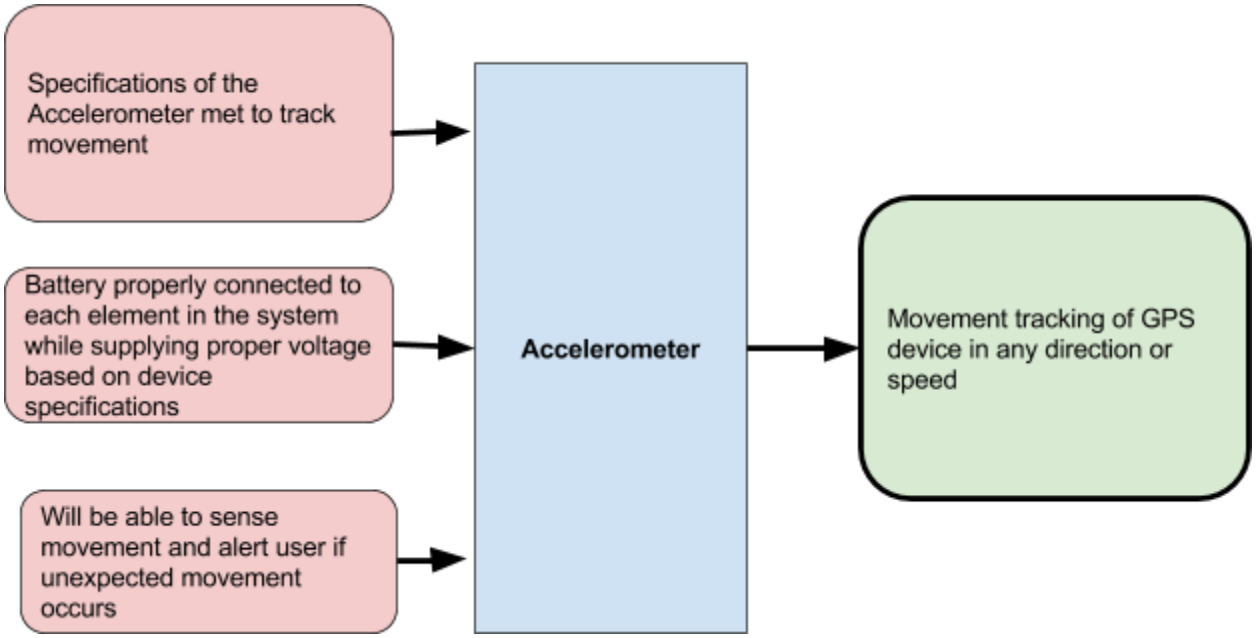
The system uses a 3.7V Lithium-Ion rechargeable battery. The decision to use a rechargeable battery was based on accessibility to USB recharging and cost of battery replacement. The Li-ion battery capacity is larger than conventional batteries, and the dimensions of the the rechargeable battery can more easily fit inside the device packaging along with the other device hardware. Since Li-ion batteries vary in size depending on power capacity, the power capacity used was the largest capacity available that fits in the packaging. The battery connects to the board via a 2-pin molex connector.

The components on the board require a 3.3V output. In order to step down the voltage of the battery to this required power, a DC/DC converter was used. A DC/DC converter was chosen because it is more efficient than a comparable voltage regulator. The footprint of the DC/DC converter on the final board required various stabilizing capacitors and planes connecting different pins in order to allow it to convert the voltage properly.

One aspect that was important to include in the design of the project was a backup battery for the GPS. When the GPS sleeps, it has two different options for restart: hot restart and cold restart. If the device has to undergo a cold restart every time it wakes from sleep, the device could require a prolonged period to get a GPS fix. If, however, there is a backup battery connected to the GPS, the device can save its last location and take less time to find another fix when it wakes up. For this reason, a coin cell battery and holder were incorporated into the board design and are included on the back of the board.

In order to increase the ease of charging the battery, a recharging circuit was incorporated on the board that could be accessed from outside of the packaging. The main component of this circuit is a Li-Polymer charge management controller. The battery can charge with this circuit, regardless of whether or not the rest of the system (GPS and ESP12) is turned on. The circuit includes an LED to indicate if the battery is charging, and it uses a USB-mini connector.

3.6 Detailed Design and Operation of Accelerometer



The accelerometer is used to track the movement of the BikeTracker and alert user if movement occurs unexpectedly. In order to code the functionality of the accelerometer, we chose to use I2C because we wanted to use a minimal amount of pins in order to reduce overlap with other hardware on the ESP12.

With the provided accelerometer, we used the wire.h and ADXL345.h libraries to communicate in I2C with some additional help from online resources to help with position acquisition. The software is setup to initialize the accelerometer device, set the baud rate, and then uses the functions outlined in the ADXL345.h library to set the activity interrupt flag.The final design utilizes the interrupt flag when motion occurs with the device when Tracker On mode is activated. This will wake the device from sleep and start the program in order to get a location. The interrupt flag on the accelerometer is connected to the B pin of a OneShot which will send one clock cycle of a high signal. This signal is sent through an n-MOSFET which pulls the chip enable pin low for a clock cycle. When it returns to high, the ESP will reset from sleep

and begin running the code to get a GPS location. This occurs in the Tracker On mode and will send a notification to the user via the phone application that the bike is in motion.

In order to test the full functionality of this subsystem, we attached the accelerometer to the ESP12 and programmed the x,y, and z coordinates to display in the serial monitor. From here, we could see the movement of the device. When a large movement was made, the device indicated an “Activity!” message showing that the interrupt flag was triggered and therefore that the accelerometer was functional. When implementing the functionality onto our own board, we made the “activity” message pop up and a serial message indicated that the entire ESP had woken from sleep in order to get a location.

3.7 Interfaces

The main interfaces of the BikeTracker are the user mobile application on iOS and the MQTT protocol which allows for the subscription of both the ESP and GPS hardware along with the mobile application. MQTTlens was the main source of testing and communication between these elements.

4 System Integration Testing

In order to test the functionality of the systems, we first used breakout boards from the Sparkfun DevThing in order to see if we could get each subsystem correctly functioning.

To test the WiFi Capabilities, a code was developed using the templates of Basic MQTT from Professor Schafer along with the information from various data sheets about the hardware of the ESP8266. The code listed in Appendix F shows the functioning MQTT communication with WiFi capabilities that was able to be achieved via the testing boards. This functionality was tested along with the GPS data which was parsed via the RMC NMEA sentence. The GPS

location tracking utilizes the Mediatech3329 GTPA010 device to connect with the microprocessor in order to transmit the NMEA sentences to be converted and used in the User Application. In order to test the full functionality of this subsystem, we programmed our hardware and software and used MQTTbike+ to see the GPS RMC sentence transmitted and published to the window. From here, the application was able to access the latitude and longitude for GPS location, showing WiFi functionality.

To test the accelerometer functionality, the ADXL345 was attached to the DevThing and used the wire.h library to communicate in I2C with some additional help from online resources to help with position acquisition. We attached the accelerometer to the ESP12 and programmed the x,y, and z coordinates to display in the serial monitor. From here, we could see the movement of the device. When a large movement was made, the device indicated an “Activity!” message showing that the interrupt flag was triggered and therefore that the accelerometer was functional. Appendix G lists the code that was used to achieve this functionality.

After achieving all these separate functionalities, we designed the board using Eagle, creating a schematic that included all the hardware we had determined necessary through the above testing. There was also a good amount of datasheet research necessary in order to make sure that all extra components necessary such as resistors/capacitors were placed onto the board. The final schematic can be seen in Appendix D. Once the schematic was complete with all hardware, we generated a Eagle Board file in order to send in to be created. The board was completed, checked for errors and then generated. The final Eagle Board file can be seen in Appendix E.

The finalized board was soldered with all necessary components attached as described by the Bills of Materials (Appendix H). This allowed for integrated testing to take place. Using

the codes from our individual subsystem testing, we combined these codes into one final testing code. The final Arduino code is listed in Appendix A with the accompanying Application Xcode in Appendices B and C. In order to test how we were communicating between all the subsystems, we utilized the serial monitor to display each state of the code that was being run at any given time and varying the data that was sent from the application in order to see that each state was being received and dealt with correctly.

We encountered several errors in the final stages of implementation that we were able to fix. First of all, we failed to ground GPIO15 in our final board design. Therefore, we replaced an incorrect 10k resistor with a 0k resistor and then added a 10k resistor to ground. This alleviated the problem. Also, we originally had an incorrectly positioned n-MOSFET that we switched the direction of in order to gain functionality. Finally, we had to connect pin 16 with the reset pin in order to get a properly functioning wake from sleep.The hardest integration we found was flagging the interrupt with the accelerometer in order to trigger the ESP to wake up. This was difficult because the interrupt needed to trigger a signal sent to the OneShot which sent a pulse to trigger the chip enable pin for only one clock cycle. After reading and writing to the registers relevant on the ADXL345, we were able to find a library on Adafruit that utilized the interrupt flag for activity in order to reset the device.

Once the code compiled, we systematically went through each scenario in which a specific response would be necessary and then checked to see that the response was correct. The following table demonstrates the scenarios and the responses.

Scenario	**Desired Response**	**Functional? (Y/N)**
Tracker Off, No Activity	-If asleep, stays asleep -Wakes up then goes straight back to sleep based on time check	Y Y

Tracker Off, Activity	-If Asleep, stays asleep -Wakes up then goes straight back to sleep based on time check	Y Y
Tracker On, No Activity	-If Asleep, stays asleep until next timed wake up -Wakes up and gets a location	Y Y
Tracker On, Activity	-If Asleep, wakes up to get a location and notifies user via app -If awake, gets a location -Will continue until user declares “Bike Found”	Y Y Y

Using this table, it is easy to trace the functionality assessed via testing. From the table, it is clear that full functionality was achieved with the BikeTracker and it is compliant to the original goal of creating a smart bicycle finder that is utilized via mobile application.

5 User’s Manual and Installation Information

Volume

1

BIKE TRACKER

Your Bike Security Solution

User Manual

Table of Contents

Section 1: Device Installation.....1

Section 2: Application Installation.....2

Section 3: Wifi Setup 4

Section 4: Bike Tracker in Action.....5

Section 1

Device Installation

Connecting the Bike Tracker to your bike

The Bike Tracker device will be installed underneath the user's bike seat on the bike post. While the device will be in plain sight, the reflective casing will disguise it from thieves by looking like a regular bike reflector.

1. Remove the screw closest to the circular clamp.
2. Place the clamp around the bike post. Place the clamp as close to the top of the bike as possible. Make sure that the hole for charging is at the bottom

3. Screw the clamp until tight.

Section

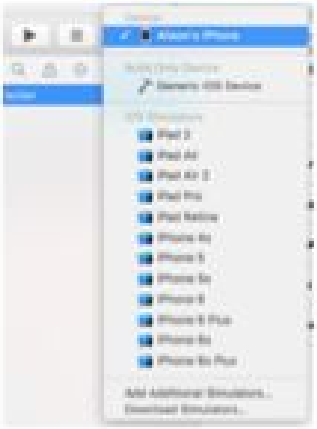
2

Bike Tracker App Installation

Installing the Bike Tracker App to your cellular device

In order to use the BikeTracker, a user must have access to the both an iPhone and a Mac computer. The user will also have to download a program to connect the app to his/her phone, if he/she doesn't have the program already.

1. Download XCode from App Store on Mac
2. Download BikeTracker App File - Go to http://seniordesign.ee.nd.edu/2016/Design%20Teams/biketra/documents.html and download the file entitled "BikeTrackerApp"
3. Extract the Zip File and open the file entitled "Bike Tracker. xcodeproj
4. In the upper left hand corner you will see the following buttons. Click on the device menu and select the name of your iPhone
5. Press the Run button (the top left button)
6. Follow the pop-up



instructions to verify you app development certificate.

7. A new application will appear on your iPhone with the logo on the left

Section

3

Wifi Setup

Connecting the Bike Tracker to wifi

The Bike Tracker uses access points to establish a wireless connection. The wifi connection needs to be established the first time the Bike Tracker is turned on, and it will then continue to connect to the selected wifi until that connection has been dropped.

1. On iPhone, select Settings -> Wifi
2. Turn on Wifi and select "BikeTracker"
3. A new window will appear on your phone (see below). Select "Configure Wifi". A list of available wireless connections will now appear.
4. A Wifi Manager will now appear with available wifi networks and their respective signal strengths. Select the wifi you would like to connect to use. Type in password (if necessary) and select save.

Bike Tracker in Action

Using the Bike Tracker to keep your bike safe!

The Bike Tracker App has a simple interface that interacts with the tracker device. The app tells the device to run in either Park Mode or Riding Mode, and will also alert the user if motion has been detected by the tracker.

Tracker On/Off –

The Tracker On/Off toggle will place the BikeTracker in Park Mode (Tracker On) or Riding Mode (Tracker Off). The toggle is initially set to Tracker Off and will only begin to track when the app is opened and Tracker On is selected.

> When in Park Mode, the tracker will update periodically update the location of the bike, which will subsequently be displayed on the Track Map. A notification will appear on the Track Map if the tracker is on, but there is no GPS fix.
>
> When in Riding Mode, the tracker will not update look for GPS locations or alert user when bike is in motion.

Last Location On/Off –

When Last Location On is selected, the last known location of the bike will display on the Track Map. If the tracker is on and receiving GPS locations, the map will display both the current and most recent bike locations. If both points are not visible, they are likely overlapped. The color should then toggle between purple and green, thus showing the last two reads have been in the same location.

Motion Detected -

When the bike is in Park Mode, the user may be alerted that their bike is moving and a new "Found Bike" button will appear on the map. This alert is created due to activity detected by an accelerometer on the tracker device. The tracker device will then repeatedly update its location and display this location on the app interface. When the user has found its bike, he/she should select the "Found Bike" button. This will alert the tracker device that it should return to its periodic tracking mode.

MQTT Server Connection -

If connection between the MQTT Server and the phone application is lost, the user will be alerted accordingly. The server will attempt to reconnect by itself. If issues continue, try to connect your phone to a different Wifi network or restart the application.

Charging -

Occasionally the Bike Tracker will need to be charged. This will be noticed when GPS location cannot be found for an extended period of time or the user opens the tracker and no lights are on. To charge the device, place your bike in a secure location and remove the tracker from the bike. Charge the tracker using the provided cord.

Reset -

If the Bike Tracker does not appear to be functioning properly, press the reset button using the User Application.

6 To-Market Design Changes

While our prototype gained full functionality, there were budget and time limitations which inhibited the further development of the project. With the knowledge from completing a year's worth of work developing the BikeTracker along with the ability to further develop the project to market, there are a few changes and adaptations that could be made.

First of all, it would be best to utilize a cellular module along with the WiFi capability in order to make the GPS functional anywhere as opposed to only in zones with an accessible Wifi network. This would be done using cellular hardware and could be incorporated into the board design to bring it to fruition.

Next, the lack of a developer's license that we had inhibited the ability for our app to be available to all phones via the AppStore. Additionally, with a license, it would be possible for the application to run in the background of the phone so the bike location would be able to track regardless of whether the user has the application directly open or not. This would give the BikeTracker a more relevant use for the user and they would not need to prompt looking for the bike every time. To sell commercially, the application would be expanded in these ways to be more versatile.

While we were able to sleep the ESP12 device in between GPS data readings, the way the hardware was set up did not allow for the GPS itself to be put to sleep along with the ESP. With more time to troubleshoot or adapt the board, it would be ideal to save battery life by making the GPS sleep. We were able to test a modified GPS mode in which the data was always available, but this setting did not fit into the larger scheme of our project prototype. For a future version, the saved battery life of this action would be very beneficial for the user. This change is already included partially in the board design of the product, since there is a backup battery for the GPS incorporated into the design. It mounts in a battery holder on the opposite side of the board.

Finally, after fully grasping the size of our board and all necessary components, the prototype casing ended up being a little larger than necessary. With this information, it would be ideal for the consumer if we reduced the size of the casing so the reflector was not quite as large. While it does fit on a standard bike seat, a slightly smaller design would look sleek and more inconspicuous. In a commercial selling state, the exact box would be manufactured to

perfectly encase the board and components while having the reflective gear on the outside fit seamlessly around it.

7 Conclusions

After the long development of the BikeTracker, a functional prototype was created. Through the design process, the subsystems were individually tested and developed in order to understand how we could integrate all functionalities together.

At the conclusion of the project, a great deal was learnt about the capabilities of WiFi utilizing an ESP12 with connection to the MQTT protocol. Additionally, the GPS module was used with NMEA sentences that were parsed within a mobile user application in order to get a GPS location for our hardware device. An accelerometer (ADXL345) was used which triggered a monostable multivibrator into an n-MOSFET which allowed the ESP12 to restart via chip enable. This interrupt allowed for a user notification to be sent when there was unexpected movement on the bicycle. The hardware utilized was mostly part which are accessible from an electronics distributor which allows for this product to have a large amount of possibility to be brought to market. In total, our budget fell under $200 including all development boards, parts, and production of the final board. With a design to keep the hardware asleep for approximately 80% of the time and a long battery life, it is likely that there would not need to be a recharge for at least 2 months when utilizing the BikeTracker.

Overall, our group thoroughly enjoyed the chance to expand our electrical engineering knowledge through hardware and software utilization. There were undoubtedly difficulties in working to combine each aspect of the project, but with a functional prototype at the end, it was well worth the struggle.

8 Appendices

Appendix A:

Final Main Xcode Listing

```
/*
Final BikeTracker Code
-connects to MQTT server
-tracks GPS data
-communicates with application in order to display
*/
#include <FS.h>                   //this needs to be first, or it all crashes and burns...

#include <ESP8266WiFi.h>

#include <DNSServer.h>
#include <ESP8266WebServer.h>
#include <WiFiManager.h>

#include <PubSubClient.h>

/************************* WiFi Access Point *********************************/

#define WLAN_SSID       "ND-guest"
#define WLAN_PASS       "CapstoneProject"

/************************* Adafruit.io Setup *********************************/

#define SERVER_ADDRESS      "senior-mqtt.esc.nd.edu"  // server in 213 SR senior-mqtt.esc.nd.edu 10.176.58.5
#define SERVER_PORT         1883        // standard port

//GPS Variables
String inputString= "";


//Accelerometer Variables
#include <Wire.h>
#include <ADXL345.h>
ADXL345 adxl; //variable adxl is an instance of the ADXL345 library

//Other Variables
int counter = 0;
int activity = 0;
int found = 1;


void callback(const MQTT::Publish& pub) {
  // handle message arrived
  //Serial.swap(); //ADD
  //delay(1000);
  Serial.print("Message arrived [");
  Serial.print(pub.topic());
```

```
  Serial.print("] ");

  Serial.println(pub.payload_string());

  Serial.println();

  if (pub.payload_string() == "Tracker Off"){
    adxl.setActivityX(0);
    adxl.setActivityY(0);
    adxl.setActivityZ(0);

    delay(1000);

   ESP.deepSleep(30000000); //30000000 30 sec

  }

  if (pub.payload_string() == "Bike Found"){

   ESP.deepSleep(30000000);
  }



  //Serial.swap(); //ADD
  delay(1000);
}// end of callback function


// Create an ESP8266 WiFiClient class to connect to the MQTT server.
WiFiClient wf_client; // instantiate wifi client
PubSubClient client(wf_client, SERVER_ADDRESS); // pass to pubsub

void setup() {

  // Setup console
  Serial.begin(9600);
  delay(10);
  Serial.println();
  Serial.println();
  Serial.println(("Modified pubsub client basic code using modified pubsub software"));
  client.set_callback(callback);


   // Connect to WiFi access point.
  Serial.println(); Serial.println();
  //Serial.print("Connecting to ");

  //Local intialization. Once its business is done, there is no need to keep it around
  WiFiManager wifiManager;

  //exit after config instead of connecting
  wifiManager.setBreakAfterConfig(true);

  //reset settings - for testing
  //wifiManager.resetSettings();
```

```
 //tries to connect to last known settings
 //if it does not connect it starts an access point with the specified name
 //here  "AutoConnectAP" with password "password"
 //and goes into a blocking loop awaiting configuration
 if (!wifiManager.autoConnect("BikeTracker", "glenna")) {
   Serial.println("failed to connect, we should reset as see if it connects");
   delay(3000);
   ESP.reset();
   delay(5000);
 }

 //if you get here you have connected to the WiFi
 Serial.println("connected :)");


 Serial.println("local ip");
 Serial.println(WiFi.localIP());

 //Accelerometer Setup
  adxl.powerOn();
  //set activity/ inactivity thresholds (0-255)
  adxl.setActivityThreshold(75); //62.5mg per increment
  //look of activity movement on this axes - 1 == on; 0 == off
 adxl.setActivityX(1);
 adxl.setActivityY(1);
 adxl.setActivityZ(1);
 //setting all interupts to take place on int pin 1
 //I had issues with int pin 2, was unable to reset it
 adxl.setInterruptMapping( ADXL345_INT_ACTIVITY_BIT, ADXL345_INT2_PIN );
 //register interupt actions - 1 == on; 0 == off
 adxl.setInterrupt( ADXL345_INT_ACTIVITY_BIT, 1);

 byte interrupts = adxl.getInterruptSource();

//activity
if(adxl.triggered(interrupts, ADXL345_ACTIVITY)){
   activity = 1;
   found = 0;
   Serial.print("activity");
//add code here to do when activity is sensed
}


 //GPS Setup
 delay(1000);
 Serial.swap(); //ADD
 delay(1000);

 Serial.print("$PMTK220,3000*1D\r\n");   //ADD
 Serial.print("$PMTK314,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0*29\r\n"); //ADD
 Serial.print("$PMTK225,0*2B\r\n"); //Normal mode
 //Serial.print("$PMTK225,8*23\r\n"); //Always locate mode

}

void loop() {
```

```
  byte interrupts = adxl.getInterruptSource();

  if (counter == 5) {
       ESP.deepSleep(30000000); //30000000 30 sec
  }

  if (WiFi.status() == WL_CONNECTED) {
    if (!client.connected()) {
      if (client.connect("mydevice")){
      //if (client.connect(MQTT::Connect("mydevice").unset_clean_session().set_auth("BikeTracker","glenna").set_keepalive(30))){ //
        //Serial.swap(); //ADD
        //delay(1000);
        //Serial.println("Connected to MQTT server");

      client.publish("bikeMQTT2/outTopic","hello world");
      //client.publish(MQTT::Publish("bikeMQTT/outTopic", "hello world qos=2").set_qos(0));

        //Serial.println("Just published hello world to outTopic");

      client.subscribe("bikeMQTT2/inTopic");
      //client.subscribe(MQTT::Subscribe().add_topic("bikeMQTT/inTopic",0));

      client.loop();

      }
    }

/* wait for incoming messages */

    if (client.connected()){
      client.loop();

      readGPS();
      counter++;
    }

  }


  if(activity == 1){
      adxl.setActivityX(0);
      adxl.setActivityY(0);
      adxl.setActivityZ(0);
      client.publish("bikeMQTT2/outTopic", "Activity!");
      //client.publish(MQTT::Publish("bikeMQTT/outTopic", "Activity!").set_qos(0));
  }

  while(activity == 1){
      readGPS();
      client.loop();
      /*if (found == 1){
        ESP.deepSleep(30000000);
      }*/

  }
```

```
}

void readGPS(){
    while(!Serial.available()){}

    delay(1000);

    while (Serial.available()){
          delay(1);
          char incomingByte = Serial.read();
          inputString += incomingByte;
       }



    //delay(3000);
    Serial.swap(); //ADD
    delay(1000);
    Serial.println(inputString);
    client.publish("bikeMQTT2/outTopic",inputString);
    delay(1000);
    Serial.swap(); //ADD
    delay(1000);


     //client.publish(MQTT::Publish("bikeMQTT/outTopic", inputString).set_qos(0));
     inputString = "";

}
```

Appendix B:

Final Main Xcode Listing

```
//
//  ViewController.swift
//  MapApp2
//
//  Created by Alison O'Connor on 9/16/15.
//  Copyright © 2015 Alison O'Connor. All rights reserved.
//
// $GPRMC,064951.000,A,4141.1456,N,08614.4338,E,0.03,165.48,260406,3.05,W,A*2C

// Done: save/show last location, customize annotations
// To do: notifications


import UIKit
import MapKit
import CoreLocation
import CoreMotion
import CocoaMQTT
import Foundation

class MapViewController: UIViewController, MKMapViewDelegate,CLLocationManagerDelegate{
```

```
 var onOff: String! = "Tracker Off"
 var lastOnOff:String! = "Last Off"
 var window: UIWindow?
 var timeOfDay: String!
 var timeHourDouble: Int!

 var info1 = Annotation()
 var info2 = Annotation()

 var found:Bool = true



@IBOutlet weak var foundButton: UIButton!

@IBAction func foundButton(sender: AnyObject) {
   foundWarn()
}


 @IBOutlet weak var resetButton: UIButton!

 @IBAction func resetButton(sender: AnyObject) {
    self.mqtt!.publish("bikeMQTT2/inTopic", withString: "Bike Found", qos: .QOS1)
 }


 @IBOutlet weak var foundBackground: UITextField!

 @IBOutlet weak var timeText: UITextView!

 @IBOutlet weak var segmentedControl2: UISegmentedControl!

 @IBOutlet weak var mapView: MKMapView!
 @IBOutlet weak var waitBox: UITextView!
 @IBOutlet weak var segmentedControl: UISegmentedControl!
 @IBAction func segmentedControlAction(sender: AnyObject) {
    switch segmentedControl.selectedSegmentIndex
    {
    case 0:
        onOff = "Tracker On"
        self.waitBox.hidden = false

    case 1:
        onOff = "Tracker Off"
        waitBox.hidden = true

    default:
        break;
    }
    //mqtt!.publish("bikeMQTT/inTopic", withString: onOff, qos: .QOS1)
 }

 @IBAction func segmentedControl2Action(sender: AnyObject) {

    switch segmentedControl2.selectedSegmentIndex{
```

```
    case 0:

        lastOnOff = "Last On"

        let defaults = NSUserDefaults.standardUserDefaults()

        let lastTime = defaults.stringForKey("StartTime")

        self.timeText.text = "Last Updated: " + lastTime!

        timeText.hidden = false

        print(lastTime)

        let lastLongitude = defaults.stringForKey("StartLongitude")
        let lastLatitude = defaults.stringForKey("StartLatitude")

        let lastLongDoub = Double(lastLongitude!)
        let lastLatDoub = Double(lastLatitude!)

        print(lastLongDoub)
        print(lastLatDoub)

        if (lastLongDoub != nil){

          let location = CLLocationCoordinate2D(
              latitude: lastLatDoub!,
              longitude: lastLongDoub!
          )

        info2.coordinate = location
        info2.color = .Purple

        mapView.addAnnotation(info2)

    }

    case 1:
        lastOnOff = "Last Off"
        mapView.removeAnnotation(info2)

    default:
    break
    }

}


var mqtt: CocoaMQTT?

let locationManager = CLLocationManager()


override func viewDidLoad() {
    super.viewDidLoad()
```

```
    foundBackground.hidden = true
    foundButton.hidden = true

    waitBox.hidden = true

    timeText.hidden = true

    mapView.mapType = MKMapType.Hybrid
    let mapCenter = CLLocationCoordinate2D(
        latitude: 41.701564,
        longitude: -86.237426
    )
    let span = MKCoordinateSpanMake(0.03, 0.03)
    let region = MKCoordinateRegion(center:  mapCenter, span: span)
    self.mapView.setRegion(region, animated: true)


    navigationController?.interactivePopGestureRecognizer?.enabled = false
    mqttSetting()
    mqtt!.connect()

    self.mapView.delegate = self
    //self.locationManager.delegate = self


}

override func didReceiveMemoryWarning() {
    super.didReceiveMemoryWarning()
    // Dispose of any resources that can be recreated.
}

func mqttSetting() {
    mqtt = CocoaMQTT(clientId: "BikeTracker", host: "senior-mqtt.esc.nd.edu", port:1883)
    //senior-mqtt.esc.nd.edu    10.176.58.5
    if let mqtt = mqtt {
        mqtt.username = "test"
        mqtt.password = "public"
        mqtt.willMessage = CocoaMQTTWill(topic: "/will", message: "dieout")
        mqtt.keepAlive = 90
        mqtt.delegate = self
    }
}

//Get current location
func locationManager(manager: CLLocationManager, didUpdateLocations locations: [CLLocation]) {

    let location = locations.last
    let currentCords = CLLocationCoordinate2D(latitude: location!.coordinate.latitude, longitude: location!.coordinate.longitude)
    let currentLocation = MKPointAnnotation()
    currentLocation.coordinate = currentCords

    self.locationManager.stopUpdatingLocation()

}

func locationManager(manager: CLLocationManager, didFailWithError error: NSError) {
```

```
    print("Error: " + error.localizedDescription)
}

//Stop updating location/motion when closed
override func viewDidDisappear(animated: Bool) {
    super.viewDidDisappear(animated)

    if CLLocationManager.locationServicesEnabled() {
        locationManager.stopUpdatingLocation()
    }

}


func moveWarn() {
    var notification = UILocalNotification()
    notification.alertBody = "Your Bike is Moving" // text that will be displayed in the notification
    notification.alertAction = "open" // text that is displayed after "slide to..." on the lock screen - defaults to "slide to view"
    notification.soundName = UILocalNotificationDefaultSoundName // play default sound
    notification.fireDate = NSDate(timeIntervalSinceNow: 5)
    UIApplication.sharedApplication().applicationIconBadgeNumber = 0
    UIApplication.sharedApplication().scheduleLocalNotification(notification)

    let cheerText = ("Your Bike is Moving")
    let messager = ("The Track Map will update shortly")
    let alert = UIAlertController(title: cheerText, message: messager, preferredStyle: UIAlertControllerStyle.Alert)
    alert.addAction(UIAlertAction(title: "Close", style: UIAlertActionStyle.Default, handler: nil))
    self.presentViewController(alert, animated: true, completion: nil)

}

func foundWarn() {
  let cheerText = ("Hooray!")
  let messager = ("You Found Your Bike")
  let alert = UIAlertController(title: cheerText, message: messager, preferredStyle: UIAlertControllerStyle.Alert)
  alert.addAction(UIAlertAction(title: "Oops! My Bike is Still Lost!", style: .Default, handler: nil))
  alert.addAction(UIAlertAction(title: "Return to Normal Track Mode", style: .Cancel, handler: { action in
    switch action.style{
    case .Default:
      print("default")

    case .Cancel:
      let found = true
      self.foundBackground.hidden = true
      self.foundButton.hidden = true
      self.mqtt!.publish("bikeMQTT2/inTopic", withString: "Bike Found", qos: .QOS1)

    case .Destructive:
      print("destructive")

    }
  }))


  self.presentViewController(alert, animated: true, completion: nil)

}
```

```
    func mapView(mapView: MKMapView, viewForAnnotation annotation: MKAnnotation) -> MKAnnotationView? {
        if (annotation is MKUserLocation) {
            return nil
        }

        let reuseId = "pin"
        var anView = mapView.dequeueReusableAnnotationViewWithIdentifier(reuseId)
        if anView == nil {
            if let anAnnotation = annotation as? Annotation {
                    let pinView = MKPinAnnotationView(annotation: annotation, reuseIdentifier: reuseId)
                    pinView.pinColor = anAnnotation.color
                    anView = pinView
            }
            anView!.canShowCallout = false
        }
        else {
            anView!.annotation = annotation
        }

        return anView
    }

}

class Annotation: NSObject, MKAnnotation
{
    var coordinate: CLLocationCoordinate2D = CLLocationCoordinate2D(latitude: 0.0, longitude: 0.0)
    var color: MKPinAnnotationColor = .Red
}


extension Double {
    func format(f: String) -> String {
        return String(format: "%\(f)f", self)
    }
    func roundToPlaces(places:Int) -> Double {
      let divisor = pow(10.0, Double(places))
      return round(self * divisor) / divisor
    }
}

extension MapViewController: CocoaMQTTDelegate {

    func mqtt(mqtt: CocoaMQTT, didConnect host: String, port: Int) {
        print("didConnect \(host):\(port)")

        if (onOff == "Tracker On"){
            self.waitBox.hidden = false
            self.waitBox.text = "Looking for GPS location..."
            self.waitBox.backgroundColor = .whiteColor()
            self.waitBox.textColor = .blackColor()
            self.waitBox.font = UIFont (name: "Helvetica Neue", size: 16.0)
            self.waitBox.textAlignment = .Center
      }
```

```
}

func mqtt(mqtt: CocoaMQTT, didConnectAck ack: CocoaMQTTConnAck) {
    if ack == .ACCEPT {
        mqtt.subscribe("bikeMQTT2/+", qos: CocoaMQTTQOS.QOS1)
        mqtt.ping()

    }

}

func mqtt(mqtt: CocoaMQTT, didPublishMessage message: CocoaMQTTMessage, id:UInt16) {
    print("didPublishMessage with message: \(message.string)")
}

func mqtt(mqtt: CocoaMQTT, didPublishAck id: UInt16) {
    print("didPublishAck with id: \(id)")
}

func mqtt(mqtt: CocoaMQTT, didReceiveMessage message: CocoaMQTTMessage, id: UInt16 ) {

    print("didReceiveMessage: \(message.string) with id \(message.topic)")

  if (message.string != nil){


    if (lastOnOff == "Last On"){

        mapView.removeAnnotation(info2)

        let defaults = NSUserDefaults.standardUserDefaults()
        let lastLongitude = defaults.stringForKey("StartLongitude")
        let lastLatitude = defaults.stringForKey("StartLatitude")

        let lastLongDoub = Double(lastLongitude!)
        let lastLatDoub = Double(lastLatitude!)

        print(lastLongDoub!)
        print(lastLatDoub!)

        if (lastLongDoub != nil){

          let location = CLLocationCoordinate2D(
            latitude: lastLatDoub!,
            longitude: lastLongDoub!
          )

          info2.coordinate = location
          info2.color = .Purple

          mapView.addAnnotation(info2)
        }

    }


  if (message.string!.hasPrefix("hello world")){
```

```
    mqtt.publish("bikeMQTT2/inTopic", withString: onOff, qos: .QOS2)
}


 if (onOff == "Tracker On"){

  if (message.string != nil){

     if (message.string!.hasPrefix("Activity!")){
            self.moveWarn()
            let found = false;
            foundBackground.hidden = false
            foundButton.hidden = false

        }


     if (message.string!.hasPrefix("$GPRMC")){

            self.mapView.removeAnnotation(info1)

            let gpsArray = [Character](message.string!.characters)

            if (gpsArray[18] == "A"){

               let latitudeDeg = gpsArray[20...21]
               let latitudeMin = gpsArray[22...28]

               let latDegString = String(latitudeDeg)
               let latMinString = String(latitudeMin)

               let latDegDouble = Double(latDegString)
               let latMinDouble = Double(latMinString)

               let latMinDoubleMod = Double(latMinDouble! / 60)

               let newLatitude = latDegDouble! + latMinDoubleMod

               print(latDegDouble!)
               print(latMinDoubleMod)

               let longitudeDeg = gpsArray[32...34]
               let longitudeMin = gpsArray[35...41]

               let longDegString = String(longitudeDeg)
               let longMinString = String(longitudeMin)

               let longDegDouble = Double(longDegString)
               let longMinDouble = Double(longMinString)

               let longMinDoubleMod = Double(longMinDouble! / 60)

               print(longDegString)
               print(longMinDoubleMod)

               let newLongitude = -(longDegDouble! + longMinDoubleMod)
```

```
print(newLatitude)
print(newLongitude)


let defaults = NSUserDefaults.standardUserDefaults()
defaults.setObject(newLongitude, forKey: "StartLongitude")
defaults.synchronize()
defaults.setObject(newLatitude, forKey: "StartLatitude")
defaults.synchronize()

waitBox.hidden = true

// Get Current Location
self.locationManager.delegate = self
self.locationManager.desiredAccuracy = kCLLocationAccuracyBest
self.locationManager.requestWhenInUseAuthorization()
self.locationManager.startUpdatingLocation()
self.mapView.showsUserLocation = true

// Set Bike Location
let location = CLLocationCoordinate2D(
    latitude: newLatitude,
    longitude: newLongitude
)

self.mapView.showsUserLocation = true


info1.coordinate = location
info1.color = .Green

mapView.addAnnotation(info1)

let timeHour = gpsArray[7...8]
let timeMinute = gpsArray[9...10]
let timeSecond = gpsArray[11...12]

let timeHourString = String(timeHour)
let timeMinString = String(timeMinute)
let timeSecString = String(timeSecond)
//print(timeString)

let timeHourDoubleOrig = Int(timeHourString)
timeHourDouble = timeHourDoubleOrig! - 4
if timeHourDouble < 12 {
   timeOfDay = "AM"
} else {
   timeOfDay = "PM"
   if (timeHourDouble != 12) {
   timeHourDouble = timeHourDouble! - 12
   }
}

let timeHourStringMod = String(timeHourDouble)
```

```
                self.timeText.text = "Last Updated: " + timeHourStringMod + ":" + timeMinString + ":" + timeSecString + " " +
timeOfDay

                let timeString = timeHourStringMod + ":" + timeMinString + ":" + timeSecString + " " + timeOfDay

                print(timeString)
                timeText.hidden = false

                defaults.setObject(timeString, forKey: "StartTime")
                defaults.synchronize()

                } else {

                  self.waitBox.hidden = false
                  self.waitBox.text = "Waiting for GPS Location..."
                  self.waitBox.font = UIFont (name: "Helvetica Neue", size: 16.0)
                  self.waitBox.textAlignment = .Center

                  self.waitBox.backgroundColor = UIColor.whiteColor()
                  self.waitBox.textColor = UIColor.blackColor()
                  self.mapView.showsUserLocation = false


                }
            }


        } else if (onOff == "Tracker Off"){

            self.mapView.showsUserLocation = false
            self.mapView.removeAnnotation(info1)
            waitBox.hidden = true
        }
    }
}
}



 func mqtt(mqtt: CocoaMQTT, didSubscribeTopic topic: String) {
    print("didSubscribeTopic to \(topic)")
 }

 func mqtt(mqtt: CocoaMQTT, didUnsubscribeTopic topic: String) {
    print("didUnsubscribeTopic to \(topic)")
 }

 func mqttDidPing(mqtt: CocoaMQTT) {
    print("didPing")
 }

 func mqttDidReceivePong(mqtt: CocoaMQTT) {
    _console("didReceivePong")
 }

 func mqttDidDisconnect(mqtt: CocoaMQTT, withError err: NSError?) {
    _console("mqttDidDisconnect")
```

```
        if (onOff == "Tracker On"){
            self.waitBox.hidden = false
            self.waitBox.text = "Connecting to Server..."
            self.waitBox.font = UIFont (name: "Helvetica Neue", size: 16.0)
            self.waitBox.textAlignment = .Center
            self.waitBox.backgroundColor = UIColor(red: 242/255, green: 38/255, blue:19/255, alpha: 1.0)
            self.waitBox.textColor = .whiteColor()
        }
            self.mqtt!.connect()

    }

    func _console(info: String) {
        print("Delegate: \(info)")
    }

}
```

Appendix C:

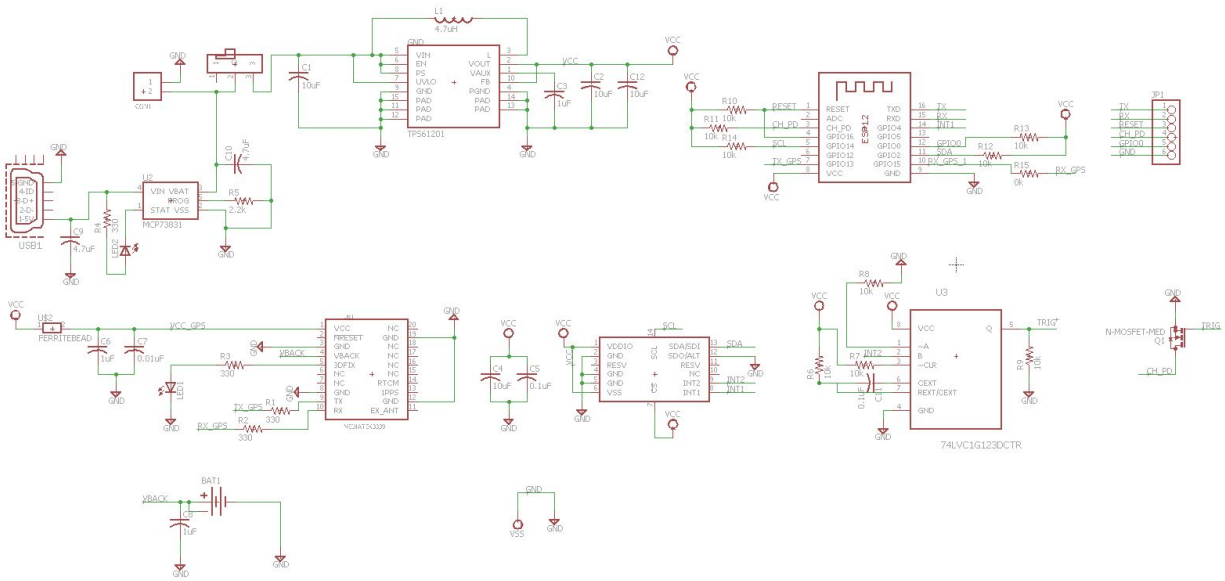
Final AppDelegate Xcode Listing

```
//
//  AppDelegate.swift
//  Example
//
//  Created by CrazyWisdom on 15/12/14.
//  Copyright © 2015年 emqtt.io. All rights reserved.
//

import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?


    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {

        let notificationTypes: UIUserNotificationType = [UIUserNotificationType.Alert,UIUserNotificationType.Badge,
UIUserNotificationType.Sound]
        let pushNotificationSettings = UIUserNotificationSettings(forTypes: notificationTypes, categories: nil)

        application.registerUserNotificationSettings(pushNotificationSettings)
        application.registerForRemoteNotifications()
        application.beginBackgroundTaskWithName("showNotification", expirationHandler:nil)

        let defaults = NSUserDefaults.standardUserDefaults()
        let defaultValueLong = ["StartLongitude" : ""]
        let defaultValueLat = ["StartLatitude" : ""]
        let defaultValueTime = ["StartTime" : ""]
        defaults.registerDefaults(defaultValueLong)
        defaults.registerDefaults(defaultValueLat)
        defaults.registerDefaults(defaultValueTime)
```

```
        return true

    }


    func application(application: UIApplication, didRegisterForRemoteNotificationsWithDeviceToken deviceToken: NSData) {
        print("DEVICE TOKEN = \(deviceToken)")
    }

    func application(application: UIApplication, didFailToRegisterForRemoteNotificationsWithError error: NSError) {
        print(error)
    }

    func application(application: UIApplication, didReceiveRemoteNotification userInfo: [NSObject : AnyObject]) {
        print(userInfo)
    }

    func applicationWillResignActive(application: UIApplication) {
        // Sent when the application is about to move from active to inactive state. This can occur for certain types of temporary interruptions (such as an incoming phone call or SMS message) or when the user quits the application and it begins the transition to the background state.
        // Use this method to pause ongoing tasks, disable timers, and throttle down OpenGL ES frame rates. Games should use this method to pause the game.
    }

    func applicationDidEnterBackground(application: UIApplication) {
        // Use this method to release shared resources, save user data, invalidate timers, and store enough application state information to restore your application to its current state in case it is terminated later.
        // If your application supports background execution, this method is called instead of applicationWillTerminate: when the user quits.
    }

    func applicationWillEnterForeground(application: UIApplication) {
        // Called as part of the transition from the background to the inactive state; here you can undo many of the changes made on entering the background.
    }

    func applicationDidBecomeActive(application: UIApplication) {
        // Restart any tasks that were paused (or not yet started) while the application was inactive. If the application was previously in the background, optionally refresh the user interface.
    }

    func applicationWillTerminate(application: UIApplication) {
        // Called when the application is about to terminate. Save data if appropriate. See also applicationDidEnterBackground:.
    }

}
```

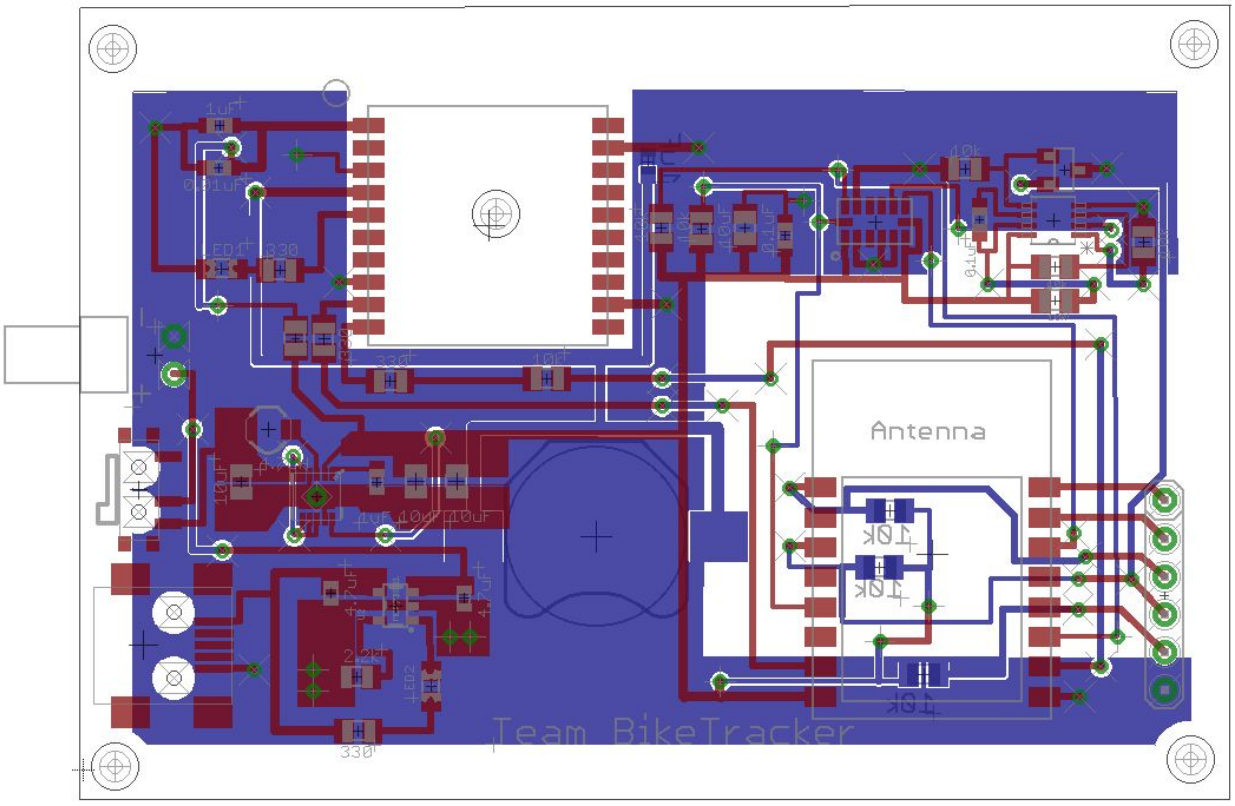
Appendix D:

Final Board Schematic



Appendix E:

Final Board Eagle File



Appendix F:

MQTT/WiFi Capability Testing Code:

```
/*
 Basic MQTT

*/

#include <ESP8266WiFi.h>
#include <PubSubClient.h>

/************************* WiFi Access Point *********************************/

#define WLAN_SSID       "ND-guest"
#define WLAN_PASS       "CapstoneProject"

/************************* Adafruit.io Setup *********************************/

#define SERVER_ADDRESS      "senior-mqtt.esc.nd.edu"  // server in 213 SR

#define SERVER_PORT         1883        // standard port

#define LED 5

void callback(const MQTT::Publish& pub) {
  // handle message arrived
  //Serial.swap();
  //Serial.print("Message arrived [");
  //Serial.print(pub.topic());
  //Serial.print("] ");

  //Serial.println(pub.payload_string());

  //Serial.println();
  //Serial.swap();

  if (pub.payload_string() == "Tracker Off"){
    digitalWrite(LED, 0);
    delay(1000);
    digitalWrite(LED, 1);
  }

}// end of callback function


// Create an ESP8266 WiFiClient class to connect to the MQTT server.
WiFiClient wf_client; // instantiate wifi client
PubSubClient client(wf_client, SERVER_ADDRESS); // pass to pubsub

String inputString= "";

//long lastMsg = 0;
//char msg[50];
//int value = 0;

void setup() {
```

```
  // Setup console
  Serial.begin(38400);  // opens serial port, sets data rate to 38400 bps
  delay(10);
  // Connect to WiFi access point.
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(WLAN_SSID);

  WiFi.begin(WLAN_SSID, WLAN_PASS);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println();

  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());

  client.set_callback(callback);

  pinMode(LED, OUTPUT);

  Serial.swap(); //ADD
}

void reconnect() {
  // Loop until we're reconnected
  Serial.swap();
  delay(5);
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    // Attempt to connect
    if (client.connect("ESP8266Client")) {
      Serial.println("connected");
      // Once connected, publish an announcement...
      client.publish("bikeMQTT/outTopic", "hello world");
      // ... and resubscribe
      client.subscribe("bikeMQTT/inTopic");
    } else {
      Serial.println(" try again in 5 seconds");
      // Wait 5 seconds before retrying
      delay(5000);
    }
  }
  Serial.swap();
  delay(1000);

}

void loop() {

  digitalWrite(LED, 1);

  if (WiFi.status() == WL_CONNECTED) {
      if (!client.connected()) {
          reconnect();
```

```
    }
    client.loop();

    //long now = millis();
    //if (now - lastMsg > 3000) {
        // lastMsg = now;
        // ++value;
        // snprintf (msg, 75, "Longitude #%ld", value);

         //Start ADD
    while(!Serial.available()){}

    Serial.println("$PMTK220,3000*1D\r\n");   //ADD
    Serial.println("$PMTK314,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0*29\r\n"); //ADD

    while (Serial.available()){
          delay(1);
          //inputArray[a] = Serial.read();
          //a++;
          char incomingByte = Serial.read();
          inputString += incomingByte;
       }
       //a=0;

     //End ADD

     //Serial.print("Publish message: ");
     //Serial.println(msg);
     client.publish("bikeMQTT/outTopic", inputString);
     //Serial.swap();
     inputString = "";
    //}
  }
}
```

Appendix G:

Accelerometer Testing Code:

```
// Cabling for i2c using Sparkfun breakout with an Arduino Uno / Duemilanove:
// Arduino <-> Breakout board
// Gnd       -  GND
// 3.3v      -  VCC
// 3.3v      -  CS
// Analog 2  -  SDA
// Analog 14 -  SCL
// Analog 0  - Int1
// Gnd       -  SDO

#include <Wire.h>

#define DEVICE (0x53) // Device address as specified in data sheet
#define LED 5
```

```
volatile int activity = 0;

byte _buff[6];
int Xold = 0;
int Yold = 0;
int Zold = 0;

char POWER_CTL = 0x2D;  //Power Control Register
char DATA_FORMAT = 0x31;
char DATAX0 = 0x32; //X-Axis Data 0
char DATAX1 = 0x33; //X-Axis Data 1
char DATAY0 = 0x34; //Y-Axis Data 0
char DATAY1 = 0x35; //Y-Axis Data 1
char DATAZ0 = 0x36; //Z-Axis Data 0
char DATAZ1 = 0x37; //Z-Axis Data 1

void setup()
{
  Wire.begin();        // join i2c bus (address optional for master)
  Serial.begin(38400);  // start serial for output. Make sure you set your Serial Monitor to the same!
  Serial.print("init");

  //Put the ADXL345 into +/- 4G range by writing the value 0x01 to the DATA_FORMAT register.
  writeTo(DATA_FORMAT, 0x01);
  //Put the ADXL345 into Measurement Mode by writing 0x08 to the POWER_CTL register.
  writeTo(POWER_CTL, 0x08);

  //attachInterrupt(0, toggle, RISING);
  pinMode(LED, OUTPUT);
}

void loop()
{
  readAccel(); // read the x/y/z tilt
  delay(3000); // only read every 0,5 seconds
}

void readAccel() {

  uint8_t howManyBytesToRead = 6;
  readFrom( DATAX0, howManyBytesToRead, _buff); //read the acceleration data from the ADXL345

  // each axis reading comes in 10 bit resolution, ie 2 bytes.  Least Significat Byte first!!
  // thus we are converting both bytes in to one int
  int Xnew = (((int)_buff[1]) << 8) | _buff[0];
  int Ynew = (((int)_buff[3]) << 8) | _buff[2];
  int Znew = (((int)_buff[5]) << 8) | _buff[4];
  Serial.print("x: ");
  Serial.print( Xnew );
  Serial.print(" y: ");
  Serial.print( Ynew );
  Serial.print(" z: ");
  Serial.println( Znew );

  int deltaX = Xnew-Xold;
  int deltaY = Ynew-Yold;
```

```
  int deltaZ = Znew-Zold;

  if (deltaX > 100 || deltaX < -100 || deltaY > 100 || deltaY < -100 || deltaZ > 100 || deltaZ < -100){
    attachInterrupt(0, toggle, RISING);
    activity = 1;
  }

  if(activity == 1)
  {
     Serial.println("Activity!");
     digitalWrite(LED, 0);
     delay(500);
  }
  activity = 0;
  Xold=Xnew;
  Yold=Ynew;
  Zold=Znew;
  digitalWrite(LED, 1);
}

void writeTo(byte address, byte val) {
  Wire.beginTransmission(DEVICE); // start transmission to device
  Wire.write(address);             // send register address
  Wire.write(val);                 // send value to write
  Wire.endTransmission();         // end transmission
}

// Reads num bytes starting from address register on device in to _buff array
void readFrom(byte address, int num, byte _buff[]) {
  Wire.beginTransmission(DEVICE); // start transmission to device
  Wire.write(address);             // sends address to read from
  Wire.endTransmission();         // end transmission

  Wire.beginTransmission(DEVICE); // start transmission to device
  Wire.requestFrom(DEVICE, num);    // request 6 bytes from device

  int i = 0;
  while(Wire.available())         // device may send less than requested (abnormal)
  {
    _buff[i] = Wire.read();    // receive a byte
    i++;
  }
  Wire.endTransmission();         // end transmission
}

void toggle(){
  activity = 1;
}
```

Appendix H:

Bill of Materials

Team Name	Part Description	Source/Supplier	Part Number	Packaging	Quantity
BikeTracker	10uF capacitor		C-USC0805	C0805	4
	1uF capacitor		C-USC0603	C0603	3
	0.1uF capacitor		C-USC0603	C0603	2
	0.01uF capacitor		C-USC0603	C0603	1
	4.7uF capacitor		C-USC0603	C0603	2
	2-pin Molex connector		CON-MOLEX-42XX-2RA	WM-4300	1
	DC-to-DC booster		TPS61201	QFN-10_PAD	1
	Accelerometer		ADXL345	LGA14	1
	6-pin pin header		PINHD-1X6	1X06N	1
	4.7uH inductor		INDUCTORCDRH	CRDH	1
	LED's		LEDCHIPLED_0805	0805	2
	N-Channel MOSFET		N-MOSFET-MED	SOT23	1
	330 ohm resistor		R-US_R0805	R0805	4
	2.2k-ohm resistor		R-US_R0805	R0805	1
	10k-ohm resistor		R-US_R0805	R0805	10
	GPS Receiver with antenna		MEDIATEK3339	MEDIATEK3339	1
	Ferrite Bead		FERRITEBEAD	C0805	1
	Sliding switch		S12B SLIDE-SPDTCUS12B	CUS12B	1
	LI-polymer charge management controller		MCP73831	SOT23-5	1
	Single retriggerable monstable multivibrator (OneShot)		TR 74LVC1G123DCTR	SOP65P400X130-8N	1
	Micro-USB connector		USB-MINI-MICRO-BMI	NI USB-MINI-FCI10033527	1
	ESP Wifi Device		ESP-12	ESP-12	1
	Coin Cell Battery Holder		BATTERY12MM	BATTCON_12MM	1

Appendix I:

Relevant Hardware Datasheets and Information

ADXL345 Datasheet and Information:
http://www.analog.com/en/products/mems/mems-accelerometers/adxl345.html#product-overview

DC to DC Converter Datasheet and Information:
http://www.alldatasheet.com/datasheet-pdf/pdf/94087/TI/TPS62101.html

ESP12 Datasheet and Information:
https://cdn-shop.adafruit.com/product-files/2471/0A-ESP8266__Datasheet__EN_v4.3.pdf

GPS MediaTek3339 Datasheet and Information:
http://www.datasheet-pdf.com/PDF/MT3339-Datasheet-MEDIATEK-900724

MCP73831 Datasheet and Information:
https://www.sparkfun.com/datasheets/Prototyping/Batteries/MCP73831T.pdf

Monostable Multivibrator (OneShot) Datasheet and Information:
http://www.nxp.com/documents/data_sheet/74LVC1G123.pdf