BusTracker

Senior Design Final Report

Grant Weber

Christine Joseph

SeungGoo Kang

Racine Hansen

## Table of Contents

1. Introduction

2. Detailed System Requirements

3. Detailed Project Description

3.1 System theory of operation (how the whole thing works)

3.2 System Block diagram

3.3 GPS Communications Subsystem

3.4 HSPA+ Communications Subsystem

3.5 MQTT Server Subsystem

3.6 Mobile Application Subsystem

3.7 Microcontroller Subsystem

3.8 Power Regulation Subsystem

4. System Integration Testing

4.1 Subsystem Testing

4.2 How Testing Meets System Requirements

5. Installation Guide

6. To-Market Design Changes

7. Conclusions

8. Appendices

## **1. Introduction**

South Bend Transpo does not always arrive on time at the bus stop. The riders stand outside waiting for a bus for extended periods of time. South Bend Transpo only has a timetable that shows when the bus is scheduled to arrive at certain bus stops. Though buses will inevitably fall behind schedule due to unforeseeable circumstances, the uncertainty around bus locations and arrival time can be combated with a bus tracking system. Other major cities such as New York and Chicago already have an application that shows the location of the bus and the estimated arrival time. South Bend needs a bus tracking system for the riders to see where their bus is. Adding a device that tracks the location of a bus and displays that GPS data for bus riders allows for better communication with bus riders and provides transit authorities with an effective, simple way to manage bus fleets.

The BusTracker is a device that tracks the location of South Bend Transpo and informs the riders about the estimated arrival time of the bus. The riders can easily check the location of the bus with their smartphones. The device is placed inside buses to get GPS data of the bus. While the bus moves along the route, it keeps updating the location of the bus to the server. The mobile application is subscribed to the server so that the users can access real-time data about the bus location. Google Direction API is employed in the application to display the routes from the bus to the bus stop. The application displays the estimated arrival time, reducing the uncertainty of bus schedules.

The BusTracker device is powered with 12V car cigarette lighter plug. Switching power regulator drops down 12V to 3.3V to power both the microcontroller and the Telit HE910 module. The microcontroller is programmed to send AT commands to the Telit HE910 module. A USB to serial chip was used to interface the microcontroller and the Telit HE910 module. A GPS antenna is connected to the Telit HE910 module to get the GPS data of the device. A cellular antenna is connected to the Telit HE910 to update the GPS data to the MQTT server. The mobile application gets the latitude and the longitude of the bus and updates the bus location into the map. When the user chooses the bus stop he is currently at, the application updates the routes from the bus to the bus stop and estimated arrival time.

The final prototype of the BusTracker met almost all of the requirements. The switching power regulator dropped down 12V to 3.3V to power both the microcontroller and the Telit HE910. The BusTracker device was able to obtain current GPS location and publish GPS data to the server. The mobile application could get the GPS data from the server and indicated the location of the bus and updated the route to the bus stop. However, the device did not have the capability to update the bus location on its own due to the team not receiving a firmware upgrade from Telit in time for demonstration day. The microcontroller was not able to send AT commands to the Telit HE910 because we could not interface the microcontroller and the Telit HE910 module with the FTDI chip.

## 2. Detailed System Requirements

* ● The system will be powered from a 12V line from the bus or from the car cigarette lighter receptacle for testing purposes. Power regulation will be needed to provide 3.3V to the Telit HE910 cellular module and 3.3V to the microcontroller.
* ● The system is installed by setting up a SIM card with a service provider. Once power is provided to the system, the microcontroller will turn on and begin sending AT commands to the Telit cellular module. The cellular chip will begin transmitting GPS coordinates to the web server. The system is used by the user pulling up our web application and seeing the bus location in real time. The user will be able to see the location of multiple buses as well as the current bus routes.
* ● All circuitry will be contained inside our box. The box itself will have to be installed on the bus without distracting the driver, so it should have the smallest possible form factor. The box should be placed where it is consistently able to get a GPS fix.
* ● None of the components are heavy so our box will be lightweight with antennas coming through the box for reception. The contents inside the box will include our final board with power regulator, PIC32MX250128B microcontroller, Telit HE910 mini-PCIE module, FT230x USB bridge, SIM card, and status LED’s. The system includes a main cellular antenna and GPS antenna. A small hole in the box will be required to allow for the 12V line to go to a connector on our board.
  + The reason using the Telit mini-PCIE module is a requirement is due to the fact that we are soldering the components of the PCB ourselves. We will not be able to solder one of the normal Telit chips, so we will provide a mini PCIE connector on our PCB and the Telit cellular chip will connect there.

## 3. Detailed project description

### 3.1 System theory of operation

This system works by sending AT commands to a Telit cellular module. The module has a SIM card associated with it set up with an AT&T data plan. The module has both GPS and cellular capabilities. The module sends GPS information to an MQTT server and publishes latitude and longitude data. An iOS application subscribes to these topics and updates the buses locations in the app. The app also shows where on the route the bus is, and how far away the bus is from your stop using the Google Map API.

### 3.2 System Block diagram

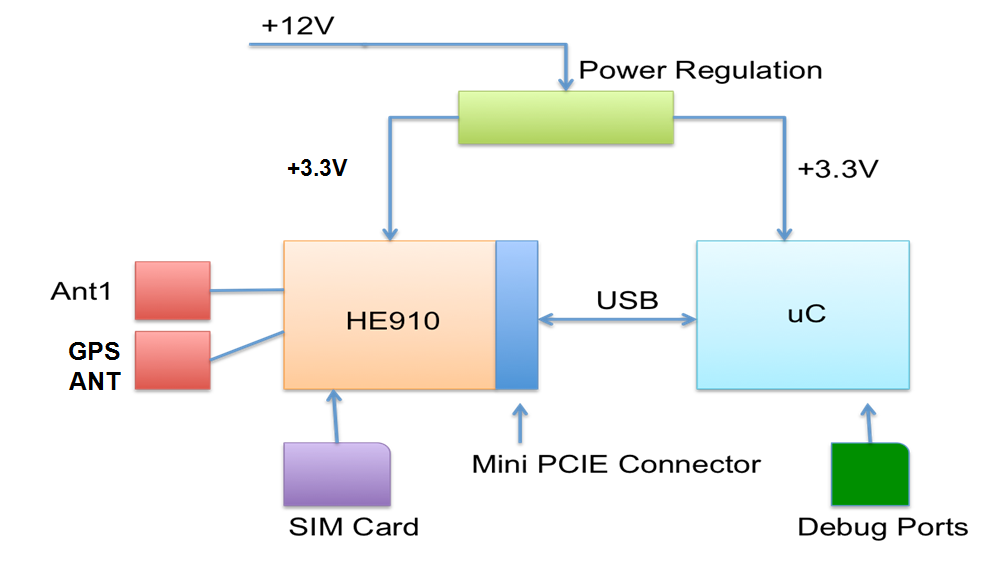


Figure 1. System Block Diagram

### 3.3 GPS Communications Subsystem

Maintaining accurate and consistent GPS data is critical to tracking the bus. The GPS communications subsystems involves obtaining relevant GPS data about the position of the bus. This subsystem includes the GPS/GNSS receiver within the Telit HE910 module, an active GPS antenna, an external radio frequency bias tee to enable the active antenna and the GPS NMEA sentences transmitted from GPS satellites.

GPS devices receive data from GPS satellites in the form of NMEA sentences. Different sentences can contain ranges of information, from position, to quality to fix to ground speed. Of the NMEA sentences the HE910’s GPS receiver is capable of receiving, it was determined that the focus would be on GLL sentences which contain information about Latitude and Longitude. The HE910’s GPS receiver was programmed to continuously return $GPGLL data. As measured during testing, new data was received about every two seconds. This data is then transmitted to the microcontroller. After the microcontroller receives the $GPGLL sentence, it should parse the sentence for latitude and longitude.Figure 2 below demonstrates the flow of GPS data to the microcontroller.

GPS Satellite

GPS Satellite

GPS Satellite

HE910 GPS Receiver

GPS Antenna

Microcontroller

GPS NMEA Sentences

GLL NMEA Sentence

Figure 2**.** Flow of GPS data to microcontroller

GPS devices usually receive data through the use of passive or active antennas. While passive antennas consume less power, they are limited by shorter RF cables. Depending on the configuration of the bus, installing the Bus Tracker on a bus could require placing the GPS antenna farther than 10cm from the actual module so that the antenna could be near a window. Therefore, an active antenna with a longer cable is needed. Active GPS antennas require a Low Noise Amplifier (LNA) to provide sufficient gain to overcome cable losses over the longer length. The active GPS antenna already has an integrated LNA. However, the LNA requires a DC bias to power it. The connections available and nature of the transmission routing to the U.FL GPS antenna connector on the mini PCIe adapter precludes simply routing 3.3V volts into the RF line to power the LNA. Therefore a bias tee is needed to inject the DC bias.

A bias tee is a three port network of components used to set the DC bias of an electrical component without disturbing other components. In this case the bias tee is used to provide a DC bias to the LNA of the active antenna, without disturbing the GPS receiver within the HE910 module. The ZFBT-4R2G part from Mini-Circuits was chosen as the bias tee. While lower cost surface mount bias tee components are available, the type of PCB being utilized and soldering processes available do not allow for correctly routing RF transmission lines. Therefore the external ZFBT-4R2G bias tee was chosen. A schematic of the equivalent electrical circuit of the bias tee is shown in Figure 3 below.

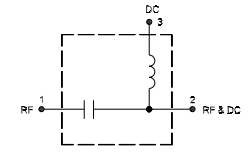


Figure 3

The capacitor allows AC RF signals to pass to and from the RF connector of the HE910 mini PCIe adapter, but blocks DC signals from affecting the HE910. The inductor allows the DC bias signal to pass, but blocks AC signals. Pin 1 was connected to HE910 U.FL GPS connector and pin 3 was connected to a 3.3V line on the main board to provide the DC bias. Pin 2 was connected to the external antenna, carrying both the GPS RF signals and the DC bias to power the LNA.

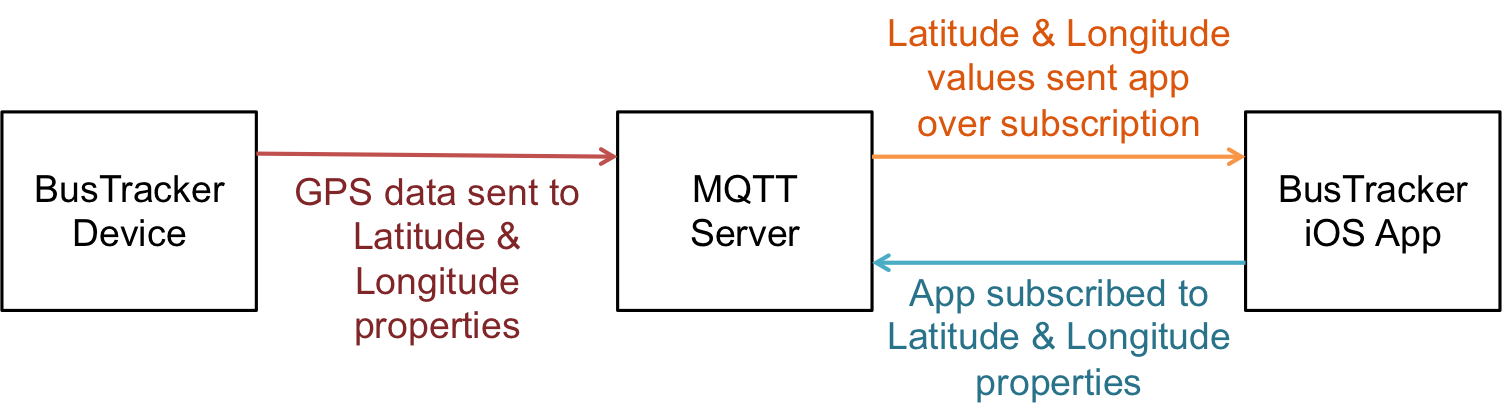
### 3.4 HSPA+ Communications Subsystem

This subsystem was responsible for transmitting the GPS data from our Telit module to our MQTT server. Although a very important and seemingly complex subsystem, it was made extremely simple due to our component selection. Our reasoning for using the HSPA+ mobile protocol is because that is what our Telit module uses to communicate with the cellular network. All we had to do was send a series of AT commands and the Telit module would do the rest of the work in the background. This was one of the many advantages of using the Telit module as opposed to a cellular breakout from Adafruit. The helpful documentation and the simplicity in the steps it took to receive GPS data and publish it to an MQTT server were extremely beneficial. The first AT command was AT+CGDCONT=1,"IP","<your apn here>". In the brackets we would send the access point name provided by AT&T for our mobile chip. You would then establish an internet connection by sending the command, AT#SGACT=1,1. At this point our Telit module was connected to the Internet. We then needed to connect it to the Telit cloud API. We would then configure the cloud configuration with the AT command AT#DWCFG=<MQTT server >,0,<application token>. The application token and the address of the MQTT server were both provided on the cloud API. The last step was to connect the module to the cloud with, AT#DWCONN=1. This documentation is all provided on the Telit website with the link in the appendix. When the device was connected to the MQTT server, the last step was to send the latitude and longitude data. This meant using the AT command AT#DWSEND=0,property.publish,key,latitude,value,40.6584. In this example, a value of 40.6584 was being published to the latitude property on the server. A similar command could be sent to the longitude property. This subsystem was tested using the EVK2 evaluation kit provided by Telit. Before we built our board, we verified that we could connect the evaluation kit to the cloud using the previous AT commands. We then posted dummy data, showing that we could get GPS information from our cellular chip to the server.

### 3.5 MQTT Server Subsystem

Telit provides an MQTT server with their HE910 device. In order to send data to the Telit MQTT server, the HE910 on the board is listed as a “thing” on the server with a key, and the properties “latitude” and “longitude” were created for the specific thing. To save the latitude and longitude of the BusTracker device, AT commands are used to publish to the defined properties latitude and longitude. Once the new GPS data is saved to the server in the properties, the iOS app receives the latitude and longitude values because it is subscribed to the topics “thing/357164040554068/property/latitude” and “thing/357164040554068/property/longitude.” From there, the location of the bus is updated on the map view in the app, and the user gets a new estimated arrival time to their stop.

In order to subscribe to the topics mentioned above, the iOS app needed a way to connect to the MQTT server. To connect to the MQTT, the app essentially becomes an MQTT client, and this was accomplished by utilizing the CocoaMQTT module for iOS. CocoaMQTT was installed using CocoaPods, and it allowed the application to connect to the server, subscribe to topics, and receive updates from the topics as the location of the bus changed. The basic interactions between the BusTracker device, the MQTT server, and the app are shown in the figure below.



### 3.6 Mobile Application Subsystem

An iOS application was developed using Swift in XCode. This application shows the map of route 1 of South Bend Transpo and the current bus location. The user can choose which bus stop he is currently at. The app then updates the routes from the bus location to the bus stop where the user is currently at. The estimated arrival time to the bus stop is calculated and displayed for the user.

To include a map of South Bend that shows routes, Google Directions API SDK for IOS is downloaded into the application. The latitude and longitude of all the bus stops in Route 1 of South Bend Transpo are used to indicate the bus stops in the map. A pickerview, a spinning-wheel that shows sets of values, is added beneath the map for the users to choose which bus stop they are currently at. The estimated arrival time is displayed on the top of the screen.

The application works as follows. The application connects to the MQTT server via CocoaMQTT which is an MQTT library for iOS. It is then subscribed to the MQTT server and obtains latitude and longitude published by Telit HE910. The application locates the bus location in the map. Google direction API calculates the bus routes from the bus location to the destination. It also takes into account the bus stops that is in between the bus location and the destination. The direction is displayed in the map and the estimated arrival time is updated. The screenshot of the mobile application is shown in the Figure below. The bus stops are indicated with red markers and the bus is indicated with a bus figure. The rider is at McKinley & Ironwood waiting for the bus going inbound. The estimated arrival time at McKinley & Ironwood is at 1:50 pm.

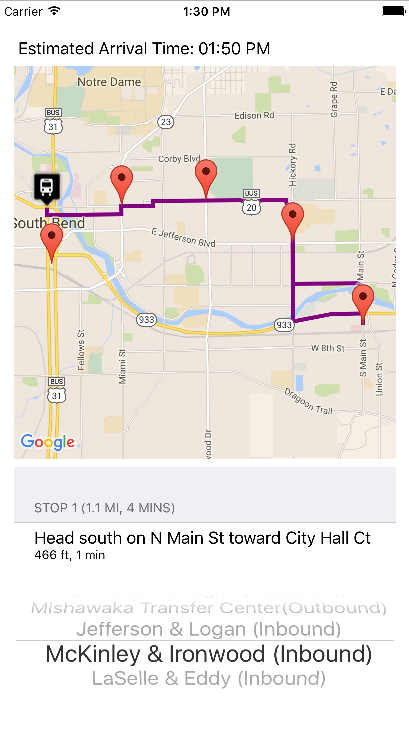


Figure 4. Screenshot of iOS application

### 3.7 Microcontroller Subsystem

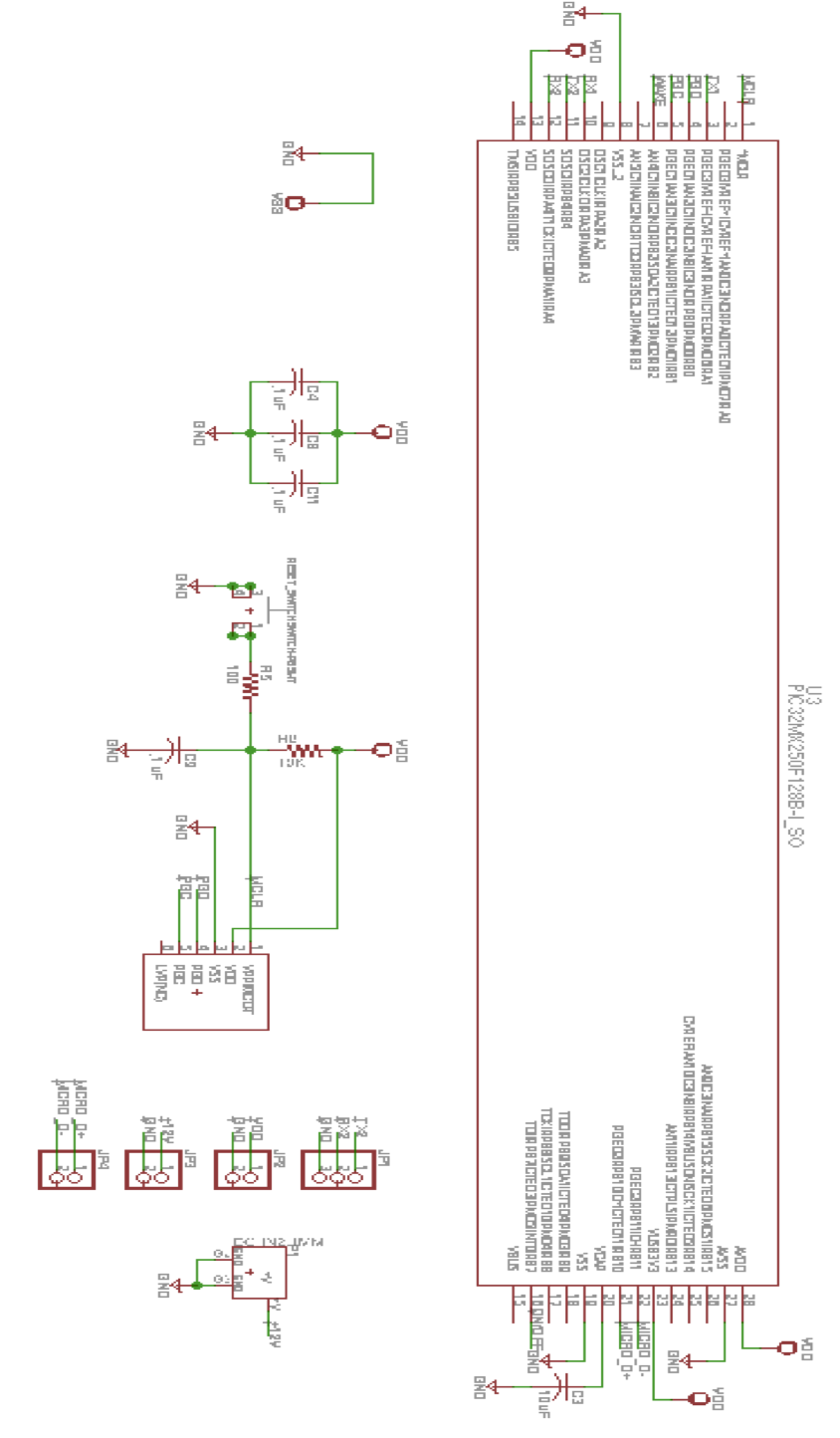
The purpose of our microcontroller was to send AT commands to the Telit module. Because we were familiar with PIC microcontrollers, we decided to include a PIC32MX250F128B on our board. This particular microcontroller was selected for a few reasons. It has 28 pins which is what we decided was optimal for our application. We wanted a component that had two UARTS as well as the ability to implement USB, and this model fulfilled both those requirements. It also included 19 remapable peripherals, meaning we could tailor the microcontroller specifically for our application. We used a PICKIT3 to program our microcontroller and decided to include a reset button for testing purposes. The programming circuitry is shown in the microcontroller schematic below. As can be seen in the figure, the reset switch is connected to the programmer MCLR line, meaning that when the button was pressed our loaded program would run again. This was useful when sending AT commands to the FTDI part, as we were able to see what data was being sent and received with a logic analyzer. The FTDI part will be discussed later in this section. We configured our microcontroller based on the minimum connections, seen on page 28 of the PIC datasheet. This meant adding decoupling capacitors where necessary. For debugging purposes we broke out one of the two UART’s to a header. The second UART was routed to the TX and RX lines of the FTDI part. We also broke out the PIC USB D+ and D- lines to a header in case we would have to implement USB with software. 

Figure 5. Microcontroller Schematic

We used the FTDI USB-to-serial chip part FT230XS to convert our data being sent from our microcontroller to USB to be received by the Telit module. We needed to do this because the only way to communicate with the Telit mini PCIE data card was with USB. This part could have been avoided if we were simply using a Telit cellular chip, but since we could not solder one of those parts onto our board by hand, we needed to use a data card that we could plug in. This part was chosen because we were familiar with how it worked since we used it during the fall semester. Our FTDI part configuration is show below.

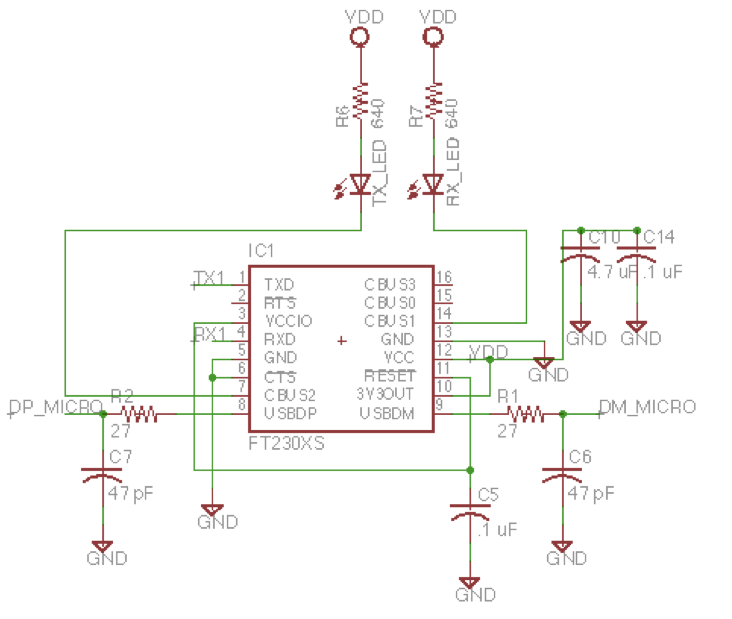


Figure 6. FT230XS Schematic

We were using the device in the self-powered configuration, as per page 23 in the datasheet. This meant that our chip was being powered from our 3.3V traces on our board, as opposed to the 5V line of the USB connector. The TX and RX leds were configured to light up when data was being transmitted or received. One thing we overlooked was that the FTDI part needs to be connected to a USB host, it cannot act as a USB host. We needed our microcontroller to be the host communicating with the Telit module. Because we overlooked this, we were not able to use the FTDI part as originally intended. One solution we tried was implementing USB with software. One of the reasons we chose this PIC version was that there is the ability to implement USB 2.0 to make your microcontroller act as a host or device. However, we had to quickly rule out this option because to use the PIC USB functionality you must use an external oscillator and the microcontroller must be operating at 48MHz. We used an internal oscillator so this clock speed wasn’t possible. This meant that to communicate with the Telit module on our final board, we had to send AT commands from a terminal program through a USB mini-B connector we added to our board. This solution is discussed in the System Integration Testing section. We have included the microcontroller code that we would have used in our final board design had we been able to communicate with USB using our microcontroller. This code sends a string of AT commands to the Telit module connecting it to our MQTT server and begins receiving GPS data. We then parse the NMEA sentences we get back and send the latitude and longitude to our server. This process happens about every two seconds the board is powered on. Our mobile application is subscribing to these topics and updates the buses location in the app.

### 3.8 Power Regulation Subsystem

The power regulator drops down 12V from the car cigarette lighter in the vehicle to 3.3V to power the microcontroller and the Telit module. L5973D switching regulator is used in our design. The input voltage should be between 4V and 36V. The regulator is able to handle higher input voltage of the car cigarette lighter. A switching power regulator is chosen because it has a smaller inductor and faster transient response than a linear power regulator. This allows the switching power regulator to respond quickly to the current peaks absorption without large voltage drops. For proper operation, a protection diode is inserted to the power input in order to save the HE910 from power polarity inversion and clean the supply from spikes.

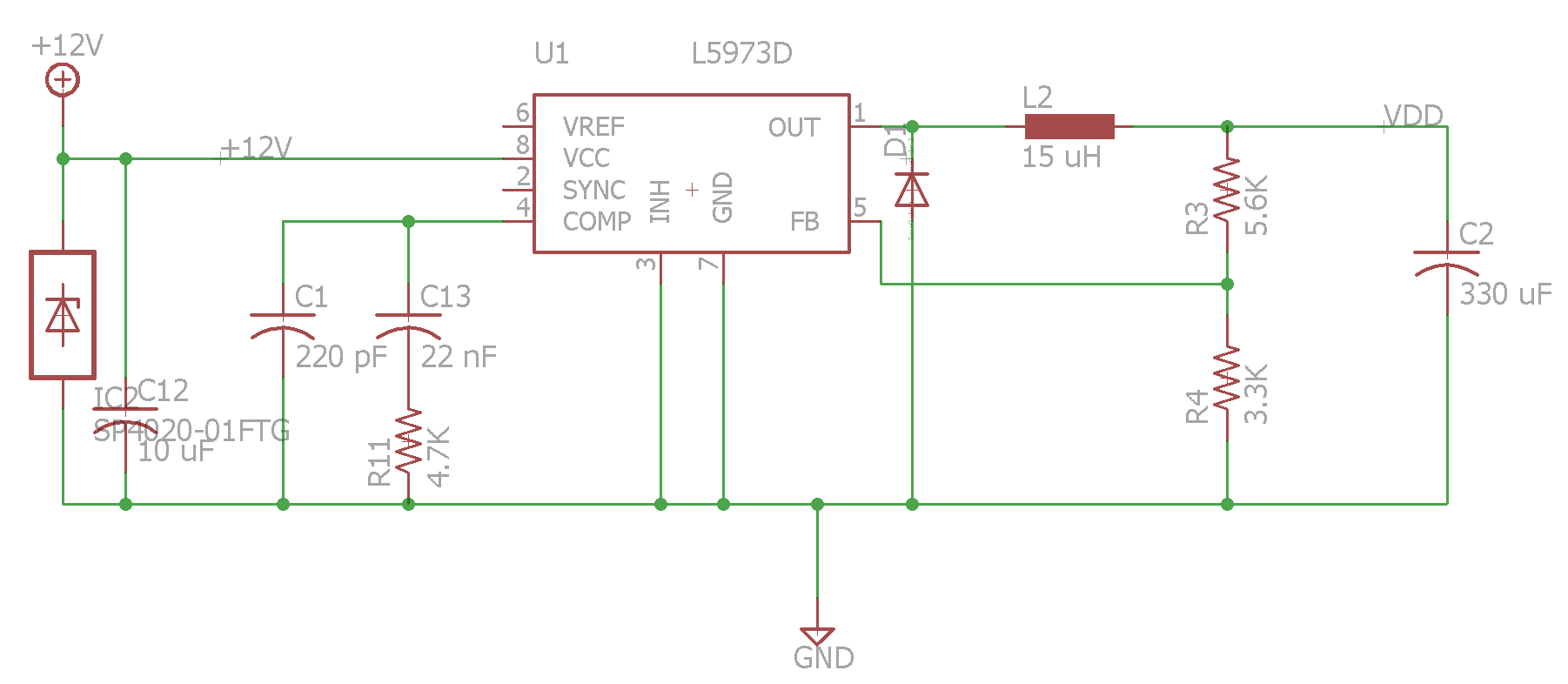


Figure 7. Voltage Regulation Schematic

## 4 System Integration Testing

### 4.1 Subsystem Testing

To test our subsystems, we first had to make sure our power regulation was working properly. We soldered all of our components onto the board, but the first time we tested our voltage regulation we did not place our Telit module into the PCIE card slot. We started testing by using 4V as the input to the voltage regulator and checking that all the VDD points were at 3.3V. This meant checking that the microcontroller, FTDI part, and mini PCIE connector pins were all correctly getting 3.3V. We then gradually increased the voltage to 12V and verified that the correct pins were still getting 3.3V.

Our next step was to integrate the microcontroller with the GPS communications. We verified that we could program the microcontroller and loaded our program that would send AT commands to the Telit module. Due to issues discussed previously, we could not implement USB-to-serial communications through the FTDI part, so we had to find workarounds to integrate the rest of our subsystems.

We added a micro-USB connector to our board to debug issues in case we had problems with the FTDI part, and this proved to be very useful in integrating the rest of the subsystems. When we first powered on the Telit module we verified that the device was receiving AT commands and that we could flash the status LED. We then sent the sequence of AT commands to start receiving GPS data. We went outside to get a fix and were able to get one in under a minute. This verified that our bias tee was correctly adding a DC bias to the active antenna.

To test the integration between the GPS communications, HSPA+ communications, and MQTT server we had to show that the GPS information was correctly being published to the Telit cloud API. We sent the sequence of AT commands to connect our module to the MQTT server and saw on the server that our device was connected. We then parsed the GPS information and used an AT command, AT$DWSEND, to individually publish latitude and longitude information. We saw these values being published on the server.

The next step to show complete system integration was to verify that our mobile application was correctly subscribing to the latitude and longitude topics and updating the bus location in the app when these values were changed. When we first tested this our app was not yet downloaded to our phones, so we showed this by running the simulator in XCode. When we published updated latitude and longitude information the buses location updated in the app as well as the updated time of arrival. We demonstrated this for multiple bus locations.

The final step was to test all the subsystems together, so we drove a bus route in our car and showed that the app was updating with the buses location as we drove the route. We powered our circuitry with the 12V cigarette lighter jack in the car. Because we could not implement USB-to-serial communication, we used a terminal program to send AT commands to the Telit module. We placed the GPS antenna in the window and once we started getting GPS data, the app updated with our location as we drove.



Figure 8. Final Product Configuration

### 4.2 How Testing Meets System Requirements

To show that our system met the design requirements we checked off each requirement as we were doing our tests. We demonstrated that our system was able to be powered from a 12V line and that voltage was properly regulated down to 3.3V for the microcontroller and Telit module. We showed that the SIM card on the Telit module could connect to the cellular network and track the device’s location. We did not completely fulfill the requirements as our microcontroller was not sending the AT commands as originally anticipated, but with a second board design we would have been able to implement USB on our microcontroller and set it up to be the USB host sending information to our cellular chip. We also showed that our app updated the buses location in real time. In our final design, all the circuitry was contained in a small box that allowed for the GPS and cellular antennas to come through the top. This allows for the box to be mounted out of sight with only the GPS antenna needing a clear view of a window to get a fix.

## 5 Installation Guide

For the Installation Guide to the Bus Tracker, refer to Appendix A in the Appendices document.

## 6 To-Market Design Changes

The goal of this project was to create a working bus tracker that we could take to the South Bend Public Transportation Company. We were very happy with our working prototype, but there would need to be quite a few changes before we could take this product to market. The main thing that we would need to accomplish would be to reduce the cost of our hardware. Due to soldering constraints we had to use the Telit mini PCIE data card with a cost of $120, but with better processes we could have used just the cellular chip itself, running about $60. We also could have avoided using the $50 bias tee, and added transmission lines to our board with an inductor and capacitor to allow for the use of an active antenna. This would have also saved on the costs of all the SMA connectors and cables we had to buy to hook up the bias tee. Another change we would have to work on is the ability for our app to learn the routes the buses drive, instead of manually entering all the routes. This would allow for our app and circuitry to work for a transportation system in any city. This would allow the buses to drive the route, and our app would be able to figure out where the stops are that the buses make, and the typical times in a day that the bus is at that stop. One of the advantages of using the Telit chip would be the ability to scale our application to an entire fleet of buses. Their cloud API makes it very easy to keep track of where all your connected devices are, and it would make it simple for whatever company was managing their fleet of vehicles with our product. Another change we would have to make is that our app could not just simply subscribe to our one board, but it would have to look for whatever buses were currently driving on that route and subscribe to those buses. This would require more logic in our mobile application code.

## 7 Conclusion

After working on the BusTracker for over a semester, we were able to create a prototype that successfully demonstrated a proof of concept for an actual bus tracking product. We learned many things such as GPS and HSPA+ communications, antenna differences, board and iOS design, and how to debug board issues when things didn’t go as planned. We estimate that with reduced hardware costs we would be able to have a working product that would be around $100 base cost with a small fee monthly for the data plan. One of the most exciting things about our product and our design is that it would be easily scalable for an entire fleet of vehicles. With a few changes to our app we could have a working bus tracker system for the city of South Bend.

## 8 Appendices

See appendices document on website.