

District Shared Parking in Seattle

EE 475: Embedded Systems Capstone



Team Park N' Go!

Rajdeep Singh

EE Class of 2017

Alexander Huttunen

EE Class of 2016

Jisoo Jung

EE Class of 2016

Yaning Yang

EE Class of 2016



Abstract

Due to limited availability of parking spaces in the Seattle area this project aims to find low-cost, low-power, and high-accuracy sensing solution to detect cars' availability in a shared parking lot. Our solution proposes placing a sensor in each individual parking space to determine the presence of a vehicle. The main sensor used in car detection is a HMC5883L, a triple-axis magnetometer, that detects large metal objects. The data from the magnetometer is then transmitted to a Si4010 microcontroller, a small microcontroller with integrated radio that is placed in each parking space. In the Si4010, a car detection algorithm is run to determine the car's presence and then information is packaged to be wirelessly transmitted to a Raspberry Pi or similar computer which functions as a base station. The base station is responsible for both receiving data from sensors and running a backend server for the parking information to be available via the web. A custom communication system was built to manage communication between the sensors and the base station. After the system was verified to work properly, a printed circuit board (PCB) was designed to be equipped with a road reflector for prototype. With this approach, the system achieves inexpensive, low-power, and accurate detection of many different types of vehicles.

What problem are you solving; goals

Each magnetometer is capable of detecting an occupancy of a single car. One base station can handle receiving data from multiple parking spaces and our system utilizes Si4010's low-power mode to collect samples in every N minutes and it accurately detects the car's presence. Therefore, the system focuses on solving long term parking garage-scale occupancy.

System Design

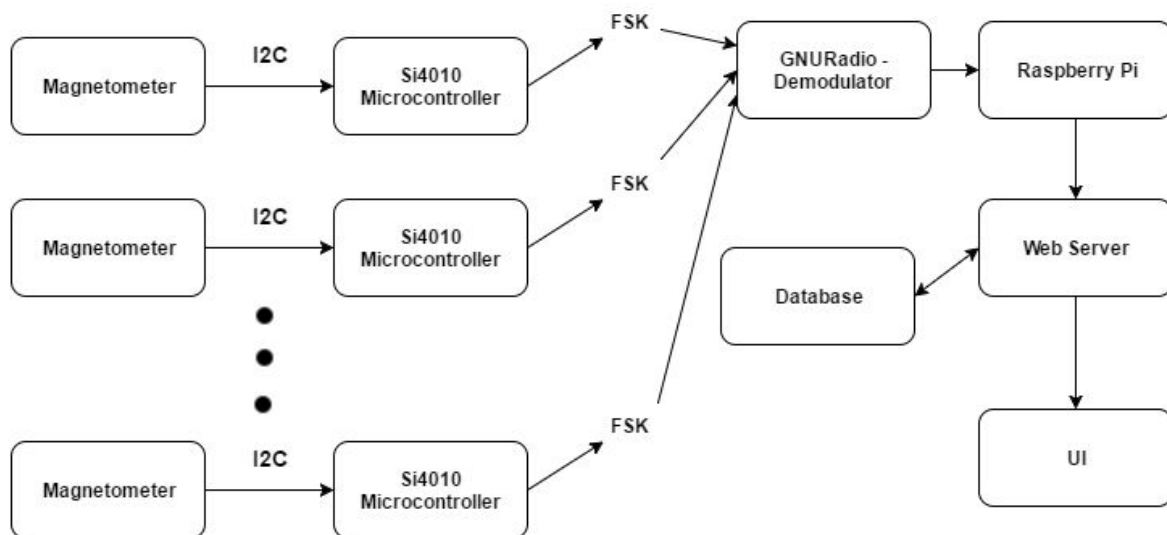


Figure 1. System architecture from sensor level to web user interface

Each magnetometer is connected to an Si4010 that includes a Frequency Shift Keying (FSK) radio and 8051 compatible microcontroller and is placed near the center bottom of each parking spot. The sensed data from the magnetometer is sent to Si4010 via the I²C protocol. The magnetometer will then collect data based on the car driving over the sensor for a set number of samples then average those samples. Note that the magnetometer can be placed in front or top of a car since it provides sensor data in all x,y and z-axis.

The sensed data from the magnetometer will be evaluated based on an baseline threshold value computed on initial power up and is then converted to a binary value. If the car is over the magnetometer the return value will be 1 and will be 0 if a car is not detected. Additionally, data such as the current temperature of the sensor, battery voltage, and information to help with error and collision prevention will be sent from the Si4010.

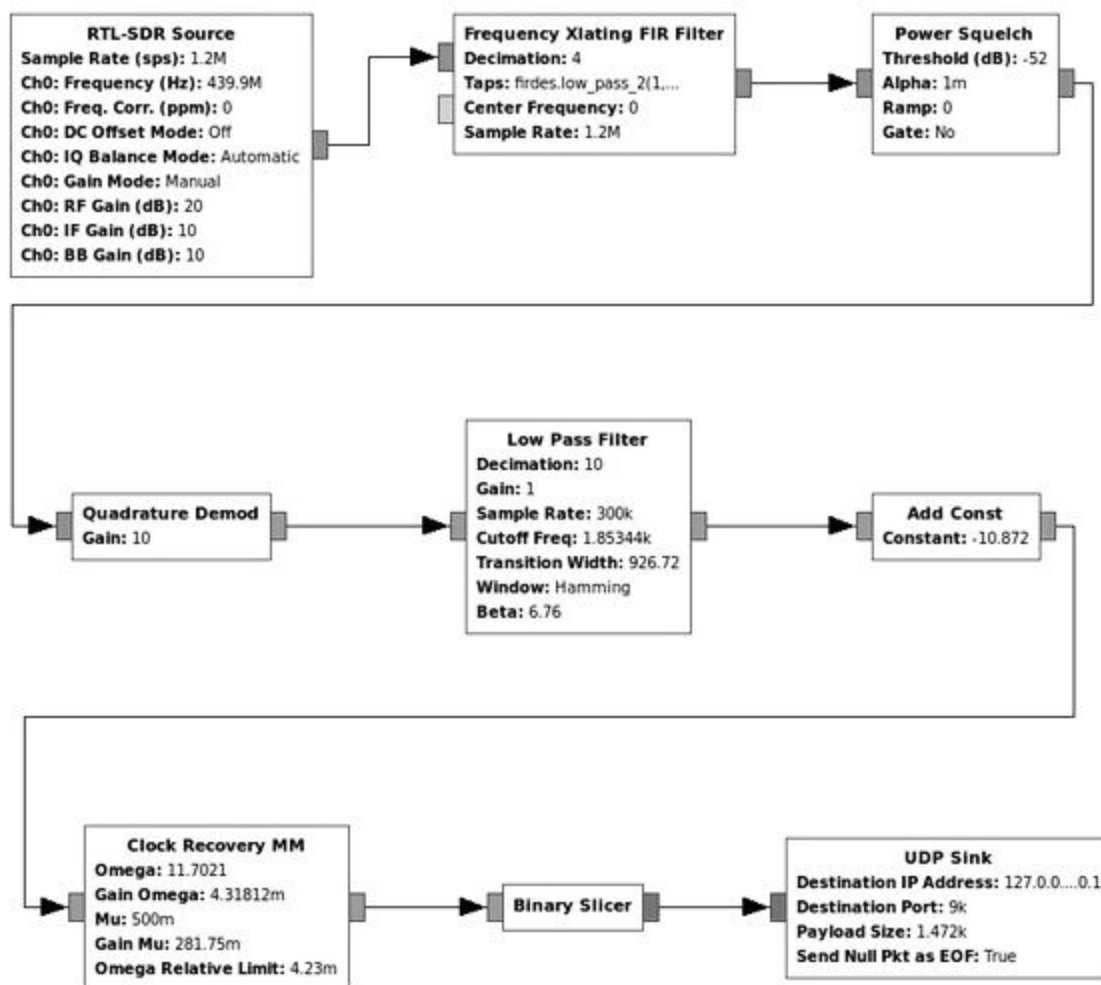


Figure 2. Schematics for demodulator in GNURadio

The Si4010 will send the data out over FSK at a frequency of around 440 MHz. This frequency can be changed later by changing a variable in the software that gets burned on to the Si4010. We will read these frequency by using GNURadio on the Raspberry Pi to receive, demodulate, filter, recover these data as 0's and 1's, and then interpret the data as a packet. A diagram from GNURadio is depicted in the figure above that shows the path that raw signal samples take when they are received at the base station.

The decoded data values will be passed to a web application and then stored in a database to know the occupancy of the whole parking lot. The data is presented on a webpage that can be viewed on devices ranging from smartphones to desktops, which will be made for the user's convenience. The backend performs actions such as receiving data from GNURadio, writing/reading to/from database and providing data to frontend will be handled by a web server. The database could also be queried by additional applications to find historical information and trends in the data and integrate it with other systems.

Strengths and Features

The main areas that our system is optimized for include low power consumption, low cost, and high accuracy.

1. Low Power Usage

In order to measure power usage over time currents during the standby mode (a.k.a. Low-power mode), wakeup mode and wakeup mode while transmitting packets. In this analysis the following assumptions were made: Si4010 wakes up every 5 minutes to send packets to the base station and Si4010 is powered by two AAA batteries.

During the wake up and transmitting packets we measured 0.017A for ~0.5 seconds and 0.031A for ~0.5 seconds (NOTE: the wakeup and the transmit occurred very quickly and 0.5 seconds is an approximate time measurement). This means the system requires 0.024A = 24mA for 1 second with 3.0 V voltage source during transmitting packets. During standby mode Si4010 used 1.33uA. 5 minutes is equivalent to 300 seconds. Therefore the system is in standby mode for 299 seconds and transmit mode for 1 second. The average power consumption of the system in 5 minutes is calculated as:

$$\text{Avg Amps} = (299 / 300) 1.33\mu\text{A} + (1 / 300) 24\text{mA} = 81.3256\mu\text{A for 5 minute wake up}$$

Since the AAA battery capacity = 1000 mAh, the system lasts

$$1000\text{e-3 Ah} / 81.3256\text{e-6 A} = 12296.25 \text{ hours} = 1.40368 \text{ years.}$$

If the system is always in low-power mode then it lasts

$$(1000\text{mAh}) / (1.33 \text{ uA}) = 751879.699248 \text{ hr} = 85.83102 \text{ years}$$

2. Low Cost

A Raspberry Pi costs \$35, but could be substituted with something like an OrangePi for \$15 with no drop in performance. The SD card and power adapter would most likely be around \$10. Additionally, a Realtek RTL2832U can be found for as little as \$8.66. The approximate cost of the system is ~\$12 per parking spot. The total cost could most likely be reduced drastically if the parking sensors were bought and built in bulk (perhaps around \$5.00).

Component	Price Available June 2016 for Quantity of 5
PCB	\$0.51
Road Reflector	\$2.39
Si4010	\$2.34
HMC5883L	\$3.00
Surface Mount Passives	\$0.20
Vishay 443MHz Antenna	\$3.28
TOTAL	\$11.72

Table 1: Cost of components for one road reflector assuming purchase of 5.

As seen in the table above, assuming that one orders the components from the most inexpensive source that sends in quantities of one, the cost comes out to be \$11.72 if ordering 5 sensors at once. Most of the cost saving compared to the prototype is in the PCB, as we would not need a very fast turn around time. It would also be fairly easy to reduce the cost a lot more. For example, the 433MHz antenna could be integrated into the circuit board instead of being a separate part. This would make the extra antenna part unnecessary. Additionally, when ordering in sufficient quantities, the price of the Si4010 would come down by about 50 cents.

A first run of sensors for a parking garage with 30 parking spaces and one base station should cost around \$400 before installation. However installation can be done by almost anyone, as all it takes to set up is to insert two batteries, type a unique identifier into a web interface, and glue the sensor down.

3. High Accuracy

A magnetometer is well-suited for detecting metal objects, which is very useful in detecting vehicles. The magnetometer consistently generates significantly differing data values when the car is over the sensor vs. when it is not over the sensor. Through a series of data collections we

have concluded that the car detection algorithm does not require complicated math, but just need a number of samples to be observed. In essence, the Si4010 collects N samples in the first wake up and stores the average of those samples to be the calibrated value, which is treated as the value when a car is not present in the parking spot. Then every time the system wakes up Si4010 computed the average of the next N samples to compare it with the calibrated value. If this new average is close enough to the calibrated value then we know that the car is present. The system with the averaging algorithm was tested with real car and it accurately detected the car. More details about the car detection algorithm are explained in the later section of the report.

Prototyping

Si4010 Development

To compile and run programs so Si4010 Keil compiler was used. We also found git repository of David Imhoff who already programmed with Si4010. From his repository we got C2 programmer [5]. The process of compiling and running goes like this (only works in Windows OS right now):

1. PC side - Run 'make uploadrun' command
2. Raspberry Pi side - Run setup.sh
3. Raspberry Pi side - Run progit.sh

I2C

The Si4010 needs to receive triple axis sensor data from the HMC5883L by communicating via the I²C protocol. Because boards like the Arduino Uno already abstracted away some things from this procedure for the user's convenience, we used an Arduino Uno to observe what an I²C signal should look like when it is sending and receiving data from the HMC5883L magnetometer. Signals on SCL and SDA pins were observed in a oscilloscope to see the expected behavior of data transmission. Setting up the magnetometer in Arduino Uno is very simple. VCC is connected to 3.3V and GND, SDA and SCL have corresponding pinout in Arduino Uno. Figure 3 shows these pinouts for the magnetometer.

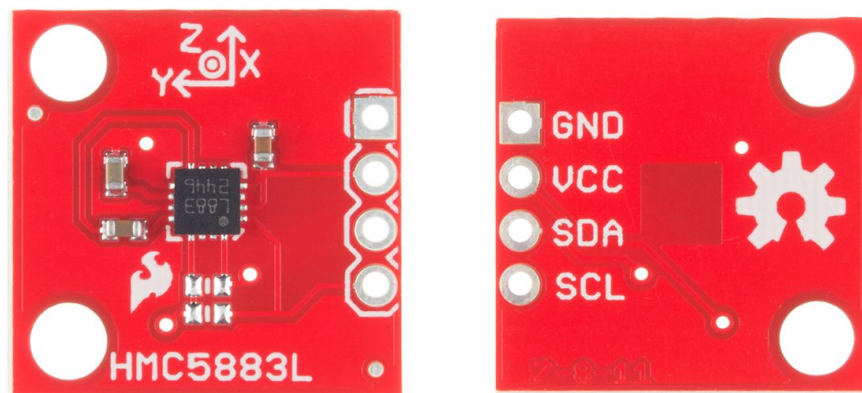
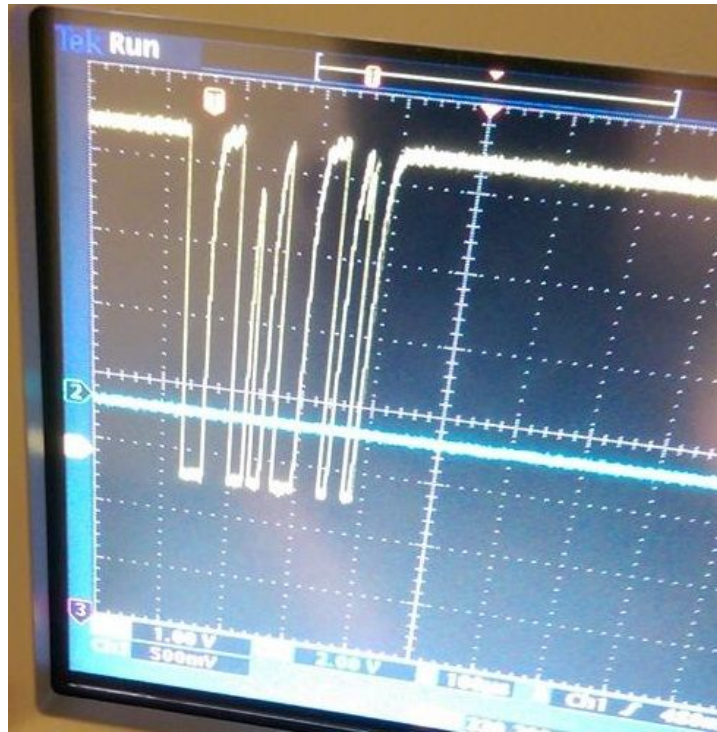


Figure 3. Triple Axis Magnetometer Breakout - HMC5883L [6]

Then the next step was to get the I²C working with Si4010. We used an I²C library from a website called Rickey's World [2]. Initial observation of SDA and SCL signals are shown in Figure 4.



**Figure 4. Testing I²C with Si4010. Signals SCL (yellow) and SDA (blue).
I²C not receiving response from the magnetometer.**

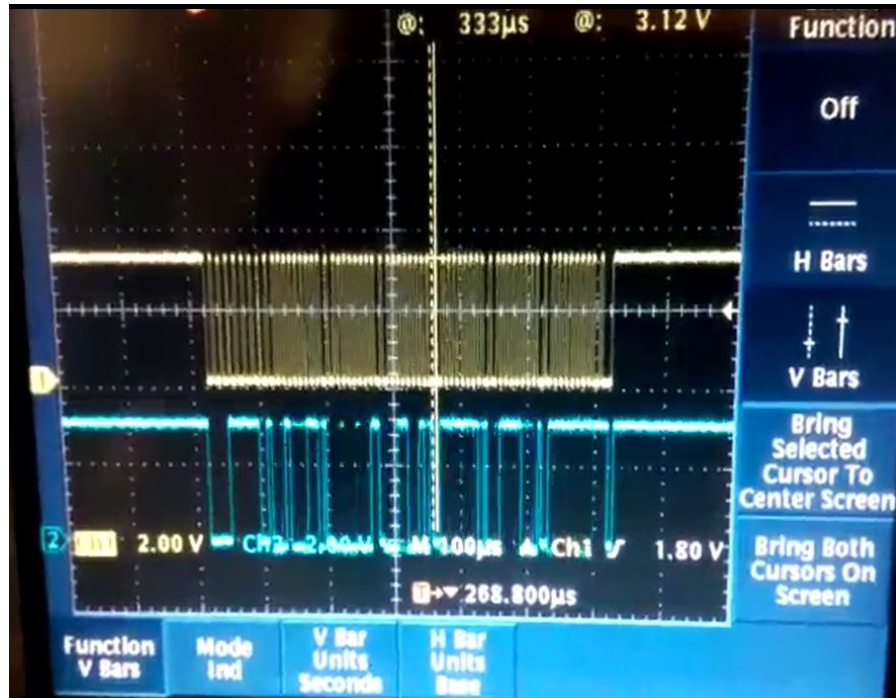


Figure 5. Fully working I2C on Si4010.

The signal for SCL was hard to interpret without proper delay in between transition. Also, SDA was not even responding. We have added delays in between data transition SCL looked like the yellow channel in the Figure 5. Then after reading through the I²C documentation we added 2.2kOhm pull-up resistor, which fixed non-responsive SDA signal.

Packet

The Si4010 sends information wirelessly to the Raspberry Pi. This information is contained in what we call a packet, which contains useful data such as car's presence in the parking spot, packet sequence number, temperature, battery life and a unique ID of the Si4010. The following code shows how the packets are defined and how they are constructed. NOTE: A single packet is simply an array of bytes and the total packet length is 10 in current implementation.

```
/*
PACKET DEFINITION
- preamble: 2 bytes for 0xFF3D
- header: 4 bytes for ID
- payload: {
    1 byte { car status (parked or not) 1bit, packet number 7bits}
    1 byte temperature in celcius,
    2 bytes battery voltage unit (mV)
}
*/
void makePacket(BYTE xdata * packet) {
    unsigned int battery_mV;
    battery_mV = (unsigned int) iMVdd_Measure(0);
```



```

// Preamble
packet[0] = 0xFF;      //Preamble 0101010101010101
packet[1] = 0x3D;      //Preamble 0110010101011011

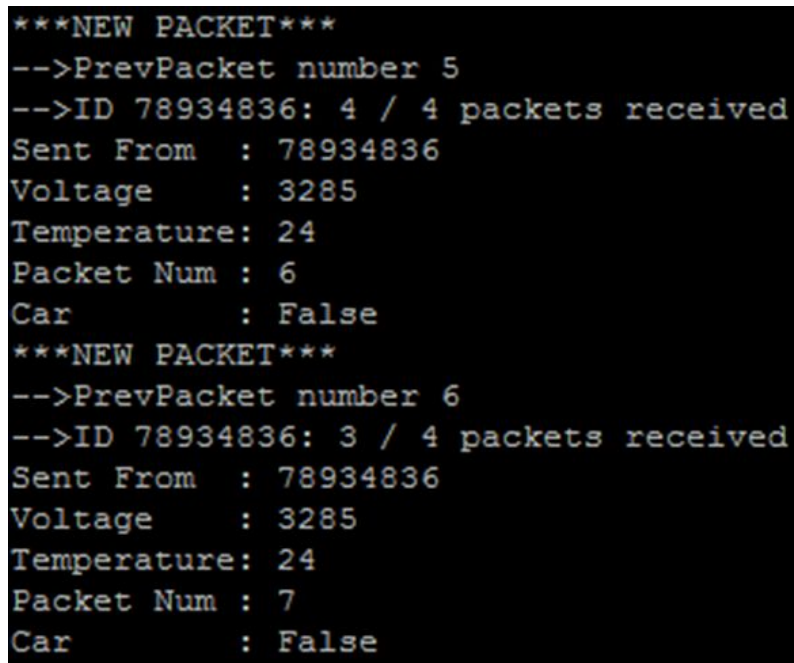
// Header: 4 byte ID which is unsigned long int
packet[2] = (BYTE) ((prodID >> 24) & 0xFF);
packet[3] = (BYTE) ((prodID >> 16) & 0xFF);
packet[4] = (BYTE) ((prodID >> 8) & 0xFF);
packet[5] = (BYTE) (prodID & 0xFF);

// Payload: car status, packet number, temp, battery voltage
packet[6] = ((found << 7) | packetNumber);
vDmdTs_RunForTemp( 3 ); // Skip 3 samples.
while ( 0 == bDmdTs_GetSamplesTaken() ) {}
packet[7] = iDmdTs_GetLatestTemp() / 220 + 25;
packet[8] = (BYTE) ((battery_mV & 0xFF00) >> 8);
packet[9] = (BYTE) (battery_mV & 0xFF);

// packeNumber range: [0, 127]
packetNumber = (packetNumber + 1) % 0x7F;
}

```

Here is how the packet looks like in the receiver side.



```

***NEW PACKET***
-->PrevPacket number 5
-->ID 78934836: 4 / 4 packets received
Sent From   : 78934836
Voltage      : 3285
Temperature: 24
Packet Num  : 6
Car         : False
***NEW PACKET***
-->PrevPacket number 6
-->ID 78934836: 3 / 4 packets received
Sent From   : 78934836
Voltage      : 3285
Temperature: 24
Packet Num  : 7
Car         : False

```

Figure 6. Printed packet information from GNURadio Python script

HVRAM

Si4010 has high-voltage RAM (HVRAM) which “provides 8 bytes of memory that retains its state as long as the battery voltage is applied and above 1.8 V” [3]. As long as the power is

connected to the chip, the HVRAM keeps its content [4]. It's very important to use HVRAM in this system because the system goes through low-power mode and wake up mode cycle. Any data stored in regular RAM is gone once the system goes to sleep so Si4010 needs a means to retain calibration data it calculated when it was first turned on.

Data types related issues in Si4010 (Reforming int from BYTE. Cause = casting issue)

HVRAM stores a BYTE in each memory address. The calibration data is of type int, which consists of 2 bytes in Si4010. An integer must be broken down into BYTE to be stored in two memory addresses. The current implementation uses total of 5 bytes to store x and z-axis magnetometer data and packet number (packetNumber is of type BYTE and ranges from 0 to 127 since it only uses first 7 bytes). The memory is mapped in big endian format so the most significant byte (MSB) is stored in the first memory address and the least significant byte (LSB) is stored in the last memory address. The following code snippet shows the memory layout in HVRAM:

```
// write to HVRAM. Data written in big endian
// Memory layout:
// addr: [0x00, 0x01, 0x02, 0x03, 0x04]
// data: [MSB_X, LSB_X, MSB_Z, LSB_Z, packetNumber]
```

In the process of putting bytes together in the next wake up mode, we have noticed that reconstructing the calibration data from HVRAM was not working properly. We were not writing and reading the same value from HVRAM. It turned out that when an int was reconstructed from multiple BYTE, the value must be casted to an int first and then shifted left by 8 bits which then fill lower 8 bits with lower significant byte. Without casting the value would not have enough bits to shift to the left direction, hence resulting an unexpected value. So here is how the calibration data are formed after waking up from low-power mode:

```
signalAverage_X = (int) bHvram_Read(calibAddrX) << 8;
signalAverage_X |= bHvram_Read(calibAddrX + 1);
signalAverage_Z = (int) bHvram_Read(calibAddrZ) << 8;
signalAverage_Z |= bHvram_Read(calibAddrZ + 1);
```

This bug is a very easy fix once a programmer knows what is causing the bug, but it is worth discussing since we have spent hours to fix the bug and can happen in the later .

Resolving packet conflict by randomDelay() and sending multiple packets

During the test of data transmission via FSK there were definitely missing packets. For more reliable communication we added a random amount of delay after each packet to reduce packet conflicts when multiple sensors are sending packets simultaneously. Also, the packets are sent 4 times to ensure there is a higher chance of receiving some packets in the backend server. Also, through a number of testing we figured out that Manchester Encoding significantly reduced the number of missing packet in the receiver side. The encoding is done in Si4010 simply by adding this line of code in SenseUnit.c :

```
vStl_EncodeSetup( bEnc_Manchester_c, NULL );
```

PCB

We decided to proceed with making a Printed Circuit Board(PCB). We first started making a schematic of the PCB layout using Altium software. The most challenging part was that the components we used were not in the Altium Vault, hence all schematics and footprints were individually drawn with Altium Software. The schematic was structured so there are five communication port at the end of the board that connects to all the pins we will use to program and run the chip. These five ports include ground (GND), voltages source (VCC), GPIO0, G17 and G18. GPIO0, G17 and G18 all connect to the Si4010 microcontroller and were used for programming the board. After the schematics we started to design the PCB layout. The goal was to fit everything in as compact as possible so the final product will fit under a road reflector as shown in Figure 7 below.

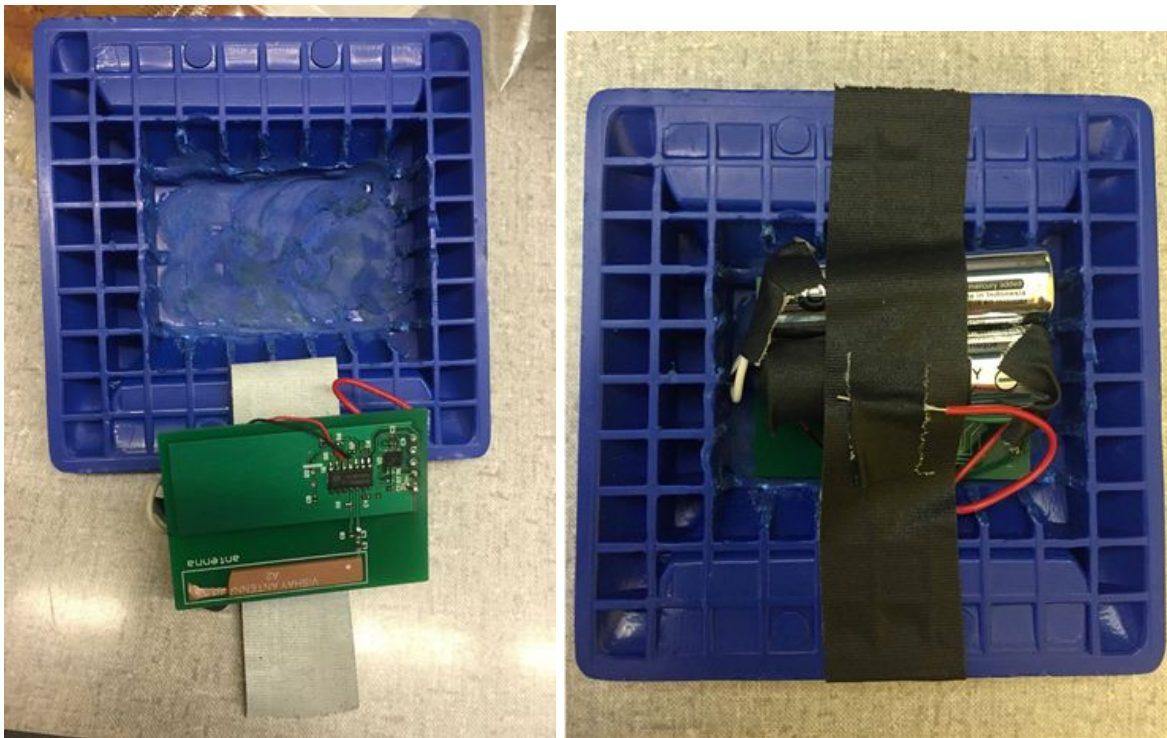


Figure 7. Final prototype of PCB under road reflector

When the design was finished and approved, we made the purchase through Sunstone Circuits and purchased corresponding component parts. One problem we ran into was that we could not find any fresh Magnetometers (HMC5883L). Meaning all the magnetometers we could found were already soldered on a pre-existing board. So when we came down to soldering we had to desolder the magnetometer from a pre-existing board and re-solder it on our PCB. This yields a higher risk of defecting the magnetometer as the actual chip is very small in size. We soldered two PCB boards with the magnetometer and only one was able to get any reading in. Also, as we were soldering we found out that we were short on resistor and capacitor components, so we soldered on values that were closest to our original values. This probably didn't have a big

affect on our overall PCB board but it was something less than ideal. Throughout testing we found out that we have a weak signal this was most likely caused because we did not tune the antenna that we purchased. On the PCB we have all tuning capacitor, resistor, inductors are shorted. If the actual tuning values were replaced we believe it can improved the signal.

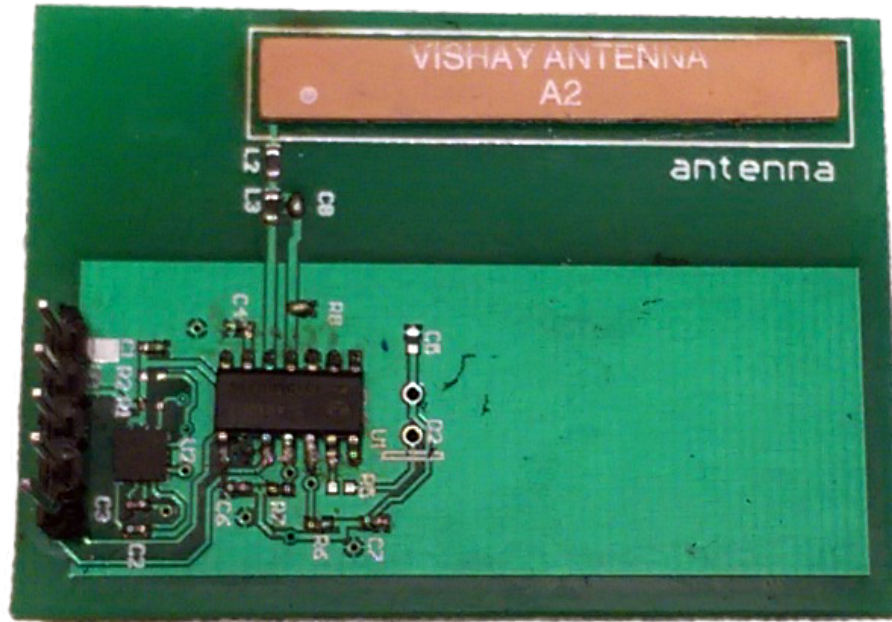


Figure 8. PCB with soldered components

Vehicle Detection Algorithm

To detect if an individual parking spot was occupied or empty, we needed to come up with an algorithm that would allow us to detect vehicles efficiently. Our goal was to detect the vehicles as they were getting parked on a specific spot or as they the vehicles were leaving. This would allow us to keep track of the available parking spots in a garage. We wanted this update in real time so that the available parking spots are getting updated pretty accurately and frequently. We wanted an algorithm that was not only limited to four wheelers, but could also detect two wheelers such as a motorcycle. This way all different types of vehicles can share the same garage and our system becomes simple and more reliable.

Initial approach

To find an algorithm for vehicle detection, we begin by setting up a magnetometer in the University of Washington CSE garage. To see how the sensor readings were affected as the vehicle was passing over the sensor, we drove a car over the sensor multiple times. We noticed that whenever the car passed over the magnetometer, there was a drastic change in the readings of the sensor. To get a better understanding of how the sensor readings were changing, we plotted these magnetometer readings. We observed that the value of the sensor at a given time rises and drops back down. That is, a high peak was observed every time a vehicle passed over the sensor. Therefore, we came up with an approach that would allow the

system to detect this peak. This detection would confirm that a vehicle has been parked on a specific spot. Hence, our system will update the total number of parking spots available. Below is the plot for vehicle detection.

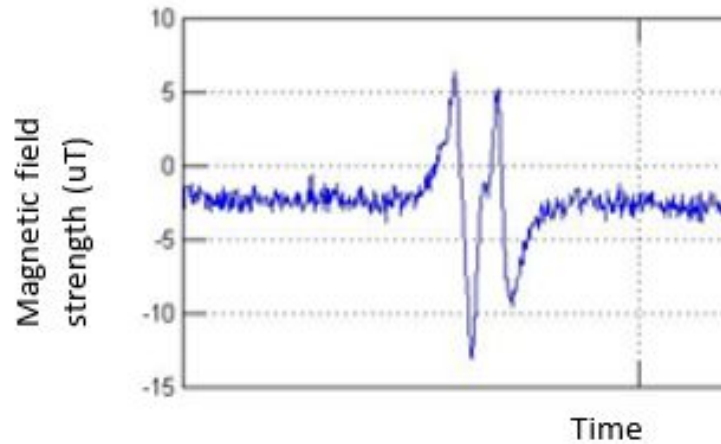


Figure 9: Peak observed in the magnetometer reading [1]

Code: Refer to appendix.

Testing and Limitations:

Once our system was able to detect a vehicle using the above mentioned approach, we set up our system in the CSE garage. To test our algorithm, we drove a car over the sensor many times, we got the expected results. This approach was detecting the vehicles being parked on a parking spot as expected. However, this approach had its own limitations. This algorithm would consume too much power as it required the system to be powered 24/7. This would raise the power cost unnecessarily. This algorithm would also adversely affect the performance of the overall system. Even though, this first algorithm would provide the right number of parking spots available, but algorithm itself was more complex which makes it prone to more errors. However, we wanted a simple system with low power consumption.

Final Approach:

Since our initial approach for vehicle detection was power hungry, we needed to figure out a better algorithm that would use less power. Therefore, instead of only looking for a peak in magnetometer readings, we focused on the readings when a vehicle was parked over the sensor for long duration and also when the vehicle leaves the parking spot. We came up with a threshold value for vehicle detection. The initial reading of the sensor is above the threshold as there is no vehicle. Therefore, if the magnetometer reading was below that threshold value, our system would count it as a vehicle being present on a specific spot. However, when the vehicle leaves, the sensor reading would again rise above the threshold value and gets equal to the initial value of the sensor. Our system would know that the vehicle is leaving, that is, the spot becomes available again.

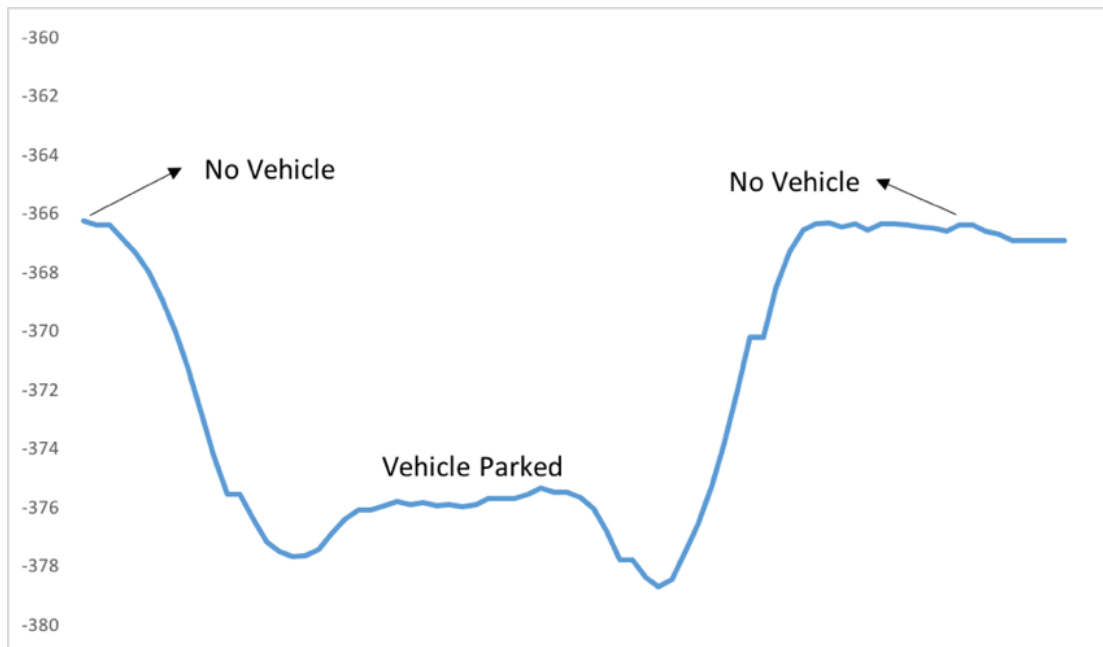


Figure 10: Graph of Magnetometer Reading as a car comes and leaves the parking spot

High Level Idea of the final approach:

- Get Calibration Data
- Calculate signal average (in x, y, z direction) during initial setup
- Calculate Threshold
- Multiple trials to calculate threshold
- Calculate Signal Difference
- If $(\text{current signal} - \text{signal average}) > \text{threshold}$
vehicle is present

Testing and Advantages:

This algorithm has many advantages over the first approach. First, it does not require the system to be powered 24/7 which would help reduce power cost. Second, this algorithm is very simple and easy to implement. To test if the new approach was working as expected, we went to Kane hall. We requested car drivers to do testing with their cars. This way we were able to test our system on multiple and different cars. We set up our sensor on different parking spots and asked the drivers to drive over the sensor. Our system was able to detect vehicles pretty accurately. We also got a chance to test our new algorithm on a hybrid car and couple of motorcycles. We were very happy with the results we got. This verified that our system is 100% reliable. Below are some of the graphs from our testing:

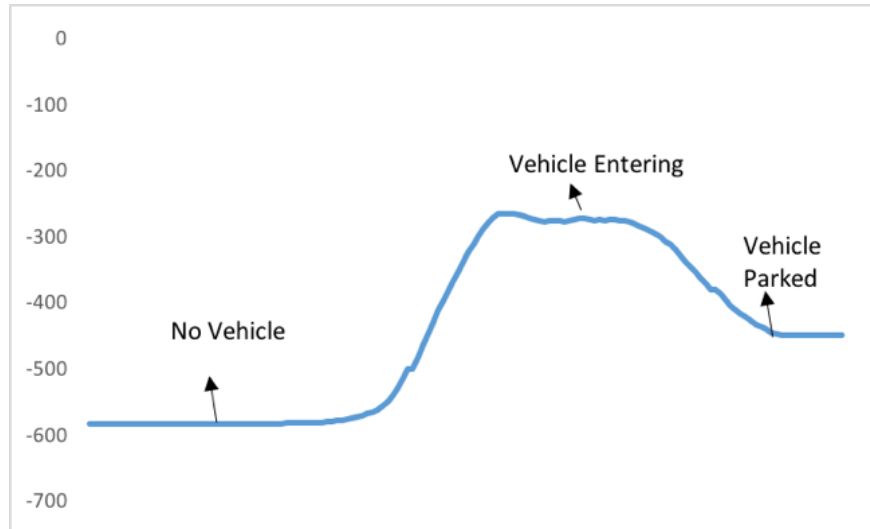


Figure 11. Hyundai driving over a magnetometer.

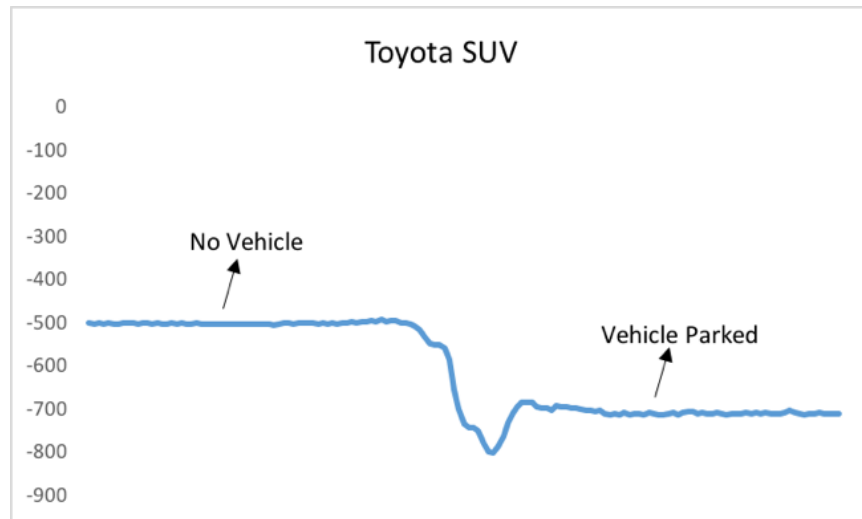


Figure 12. Results of a Toyota SUV driving over a magnetometer.

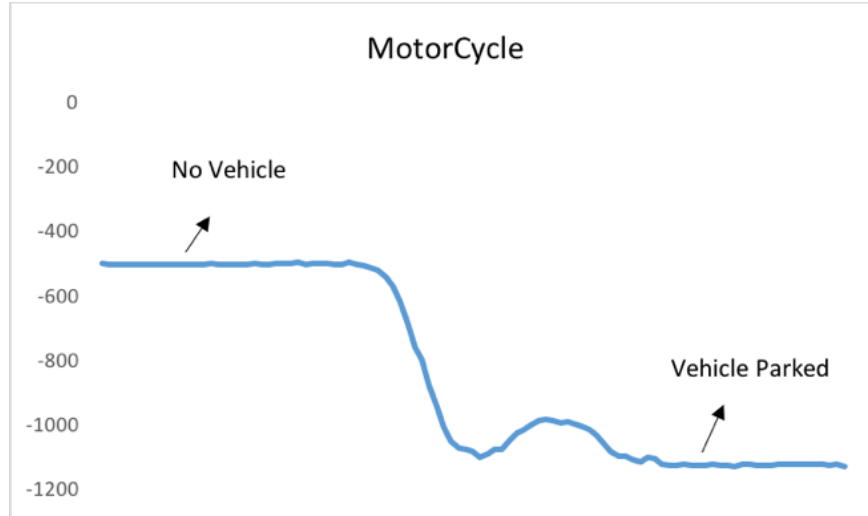


Figure 13. Results of a motorcycle driving over a magnetometer.

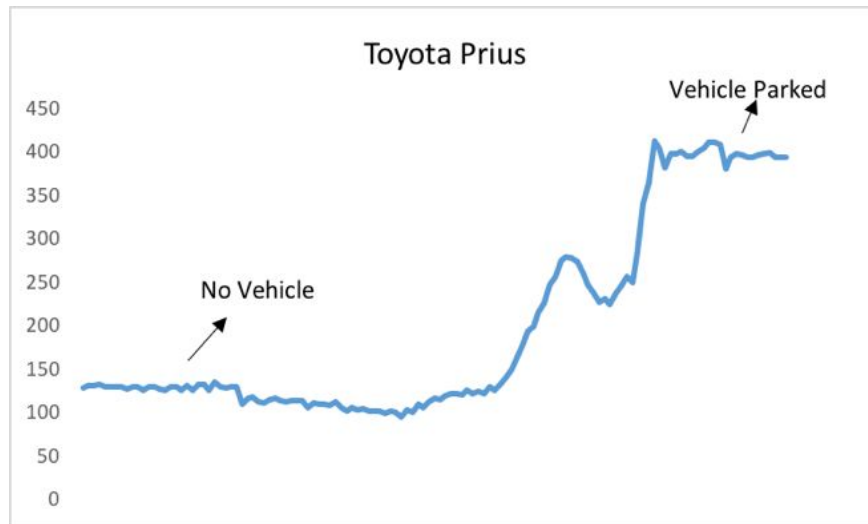


Figure 14. Results of a Toyota Prius driving over a magnetometer.

Code: Refer to appendix.

Backend and Database

Raspberry Pi has several tasks in our system. It receives parking information from each parking spot, hosts a server and manages updates in the database. The server is hosted by Python using the Flask framework and SQLite is used as our database. To add and update parking information to SQLite we use SQLAlchemy which is the Python SQL toolkit. SQLAlchemy has its own query syntax so the original query used in SQLite is also mentioned in the source code. More details on the code is in the Appendix.

User Interface - Web Application

We designed a website user interface that can update the occupancy of the parking lot in real time. The user interface is designed for whoever is monitoring the parking garage. On the website we have the exact layout of the parking garage with the corresponding spots. This could vary from lot to lot and floor to floor so it will be significant for each parking lot. As you can see in Figure 15, a green circle means the parking spot is available while a orange car means the spot is occupied. For each parking spot there is a unique sensor ID that matches with the sensor we are using. At each parking spot you are also able to see the battery life so the admin is able to know when they should replace the battery for the sensors. Also, there is a built-in temperature sensor in Si4010 and can provide where the shady spots are in the parking lot.

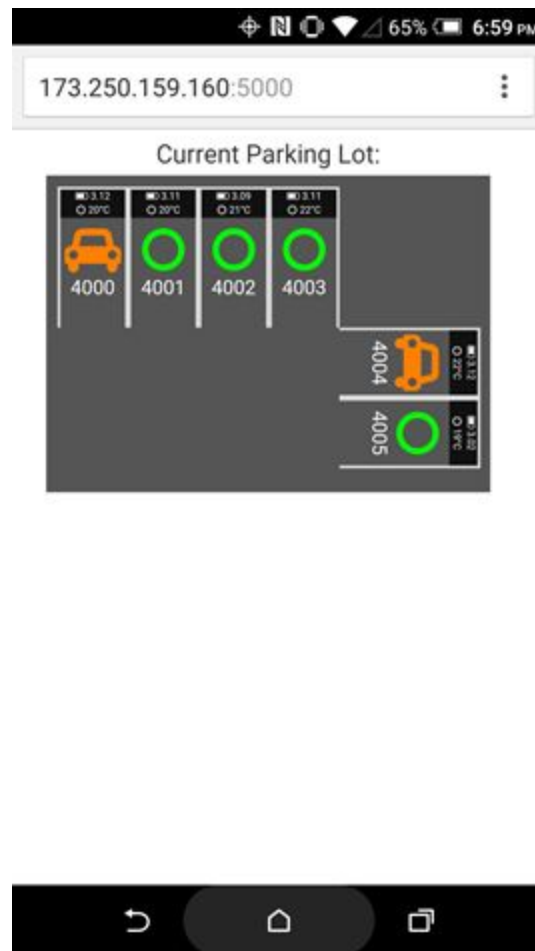


Figure 15. Using web app via a smartphone

This is a web app which can be accessed from both smartphone or computer. The mobile app has not been developed at this point. In the backend the Raspberry Pi is running a Python Flask server that is responsible for both receiving data from each individual parking spot as well as updating the SQLite database using the SQLAlchemy library.

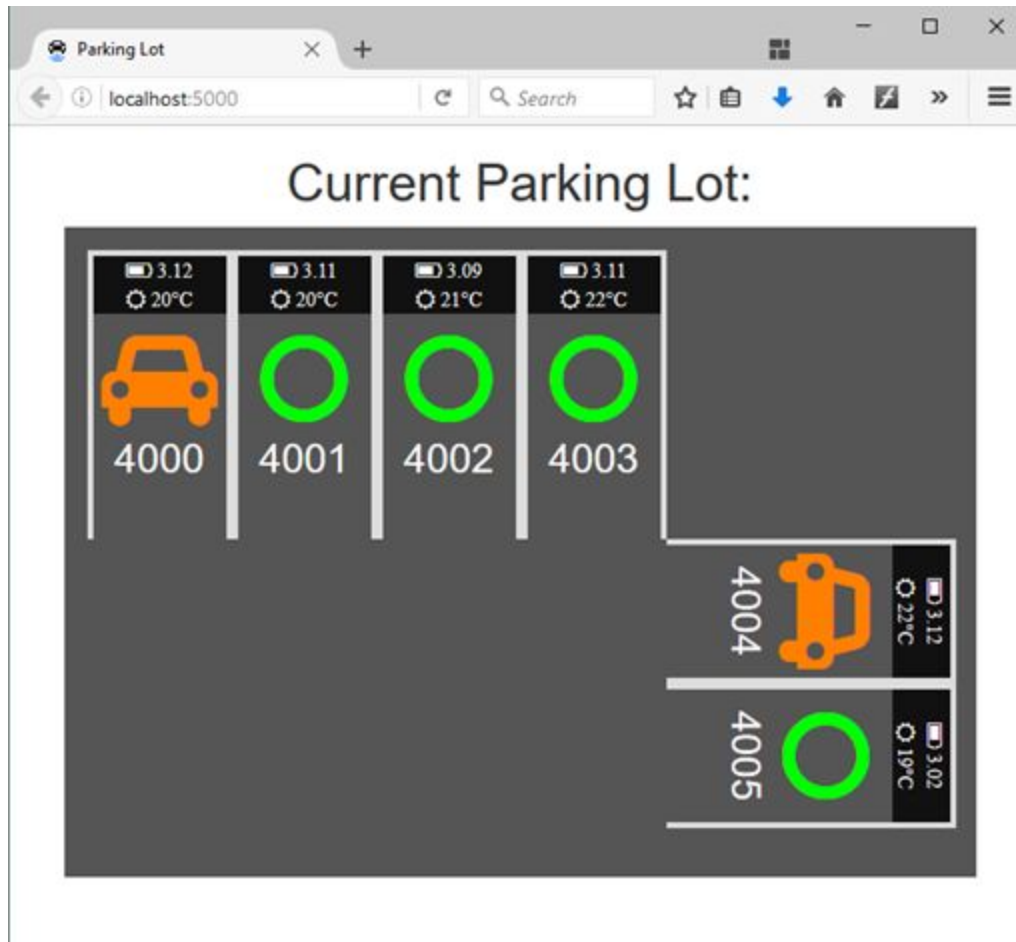


Figure 16. UI hosted by Python Flask as a backend server.

Additionally, our system is very modular and it would be easy to move parts of the system off of the Raspberry Pi and on to a more powerful computer if the parking lot status page became more popular and could not handle the web traffic.

Unknowns/Limitations/Risks

Ideally this system is supposed to support more than just two parking spots. Due to availability of PCB's we designed we only tested up to two sensors. When the system scales in an actual parking garage we might encounter unexpected behaviors. The packet receiving might not work as expected because there is too much signal interference between the multiple parking sensors. This is our biggest unknown since we have not tested the system in a larger scale. Also, the system might fail due to extreme weather or vandalism.

There is a limitations in the temperature sensor in Si4010 as it is not designed to be accurate. However, it should be a good indicator for displaying ambient temperature if the temperature difference is significantly large between shady space and sunny space.

Finally, we haven't run our system for more than a few hours at a time. While it worked well for those few hours, there may be some subtle bugs in the code that cause issues after the base station has been running for very long periods of time. The sensors should not have this issue, since they reset their internal memory after they go to sleep, which happens every N minutes.

Changes

One of the initial ideas for detecting cars in a parking lot that our group came up with was to place a magnetometer in every parking spot in the parking lot. This was reasonably close to the idea that we ended up implementing. However, there were some extra features that we added that we did not initially consider and some implementation details that we did not consider initially.

When our group first designed the sensor system, we did not consider how the sensor would get power. We had an idea of how much power the system would draw, but we were unsure if we could use the sleep mode on the Si4010. We tossed the idea of using a supercapacitor around, but we did not have any idea how to get power into the system, and the supercapacitors were expensive for the amount of power that they could store. Later, after some testing, we determined that we could use the sleep mode, and that our power requirements were very low. This led us to use a battery to power the sensor since the life before battery replacement would be measured in years and replacing the batteries should be fairly inexpensive.

Our group also thought that we would detect the disturbance that occurs when a vehicle is moving over the sensor, since that is what a blog on NXP's website said the magnetometer was capable of. It turned out that the magnetometer is able to go further and is able to detect if the vehicle is there or not in the absence of motion. This was the breakthrough that allowed us to use the low power state of the Si4010.

For new features that got implemented, we found that the Si4010 provides easy to call functions that are built in to the rom to obtain values such as the current input voltage and temperature of the chip. This allowed us to add the information into the radio packet that the Si4010 periodically sends without the need of any extra hardware. The battery voltage allows an administrator of the parking system to determine when a parking sensor might need to have its batteries replaced, while the temperature allows the front end to do things such as direct people to the coldest parking spot if the sensor is used on a hot day on an above ground parking lot.

Implementation Post-Mortem

Our group met the milestones that we laid out near the beginning of the project. Those milestones were to try out a magnetometer with an actual vehicle to see what data we could get,

achieve I²C communication between the magnetometer and process the signal into a boolean value, and lastly radio a signal back to the base station from the Si4010. We also planned that we would polish the web interface in the last two weeks. This didn't happen to the extent originally planned, but we got a reasonable display of the current status of the parking lot. The time in the last two weeks was mostly spent working on the PCB, which we failed to account for in our time calculations. Originally, our group thought that the PCB would be a nice thing to have, but we ended up actually creating the schematic and getting it manufactured. We also did not have much time to test the sensor on the PCB with as wide of a variety of cars and situations as we did with the components when they were on the breadboard.

The first major problem that we ran into was when we were unable to communicate via I²C from the Si4010 to the magnetometer. We found that pull up resistors are not optional for I²C to work well as we initially thought.

Another problem that we had was when we found that the radio signal was getting out of sync when decoded by GNURadio and spurious ones and zeros were getting added in the middle of packets during bit patterns. We rectified this problem by switching to a Manchester encoding scheme that allows the "Clock Recovery MM" to have more single ones and zeros to lock the internal synchronized clock signal on to. After implementing Manchester encoding, we were able to detect single adjacent bit flips in a signal and were able to decode data reliably regardless of the bit pattern that it produced.

There is room for optimization in car detection algorithm in terms of memory usage. Currently, to calculate an average over 20 samples we allocate a buffer size of 20 and store all 20 samples in the buffer and then calculate the average. In fact, this buffer is unnecessary. The averages can be calculated by simply adding samples to a single integer variable and then after 20 samples it can be divided by 20 to find the average.

In addition, we found that the weakest part of our system was the antenna of the PCBs. While prototyping, our group found that having a piece of wire attached to one of the pins of the Si4010 and another piece of wire sticking out of the software defined radio was enough to pass a signal from the back room of a lab through a wall and into the far corner of another room. This was with a gain setting of 20 on the SDR. After testing with the PCB, we found that the gain needed to be turned up to around a value of 50 to get reception that only worked in one room and was unable to penetrate a wall. We later took a measurement of how far packets could be reliably read from the PCB and found it was around 17 meters without obstruction. Knowing that the random wire had performance that was so much better leads us to believe that there should be a lot of room to improve the antenna on the PCB. One would expect a properly tuned antenna to outperform a random piece of wire that was not even cut to the proper wavelength of the signal, not the other way around.

Evaluation

The most important subsystem in our design would be wireless communication from Si4010 to Raspberry Pi. The Si4010 sends the data out over FSK at a frequency of around 440 MHz. To verify that the communication actually works, we ran GUI from GNUMRadio to see signals in frequency domain as shown in Figure 17. We were periodically receiving signal from 439.9 to 440 MHz band. Then we sent a number that increments for each packet so we know that proper data had been sent.

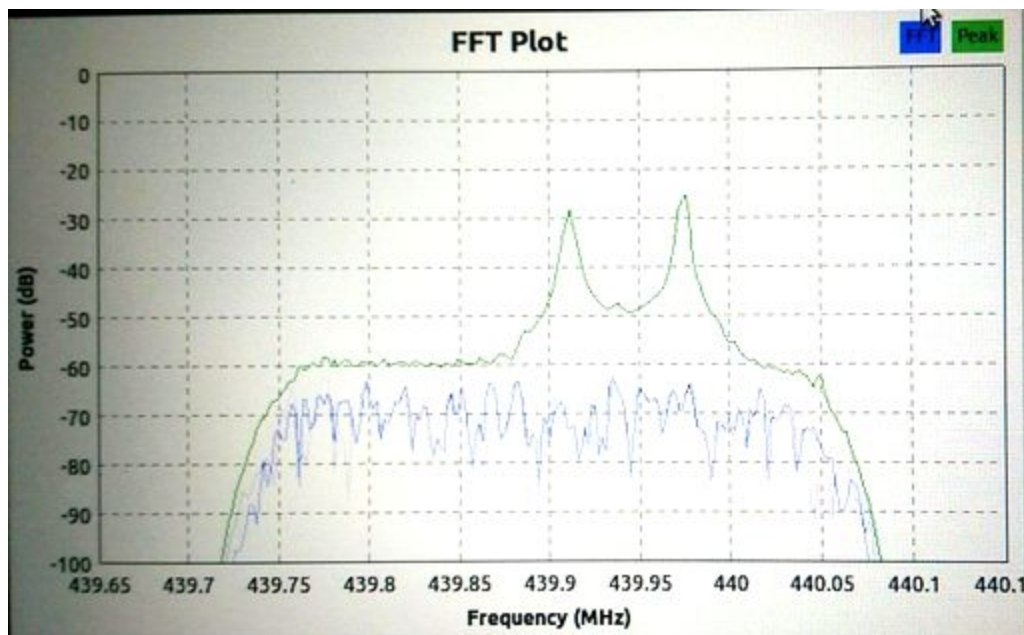


Figure 17: FFT of signal while Receiving data via FSK

During the development process, our group tested our sensor in the loading dock of the CSE building. Additionally, a version of the sensor with near final firmware was tested in a garage that is affiliated with Capitol Hill Housing. During all stages of testing, it appears that the magnetometer was able to detect if a vehicle was close in proximity reliably. The only times when it appeared that the sensor was sending false data were traced back to problems with the communication system or firmware, not the magnetometer.

The communication system used transmits data at a rate of around 2400 baud. It attempts to avoid collisions with other modems that are transmitting at the same time by sending four packets that are spaced apart by a random amount of time. The random seed is determined by the unique ID that is burned into the Si4010 at the factory. When tested with two PCBs similar to the one seen above, this idea appeared to work with two sensors. The base station was able to decode packets from each sensor when the sensors transmitted at the same time, although

there was definite evidence of interference that showed up on the debug console in the form of missing duplicate packets and clock desync errors.

As referenced above, our sensor is able to communicate with the base station with a range of 17 meters using the untuned antenna on the PCB. Like mentioned in Prototype section before there are number of things that might have affected the performance of the antenna. The magnetometer chip was out of stock later in the quarter when we started PCB design. So we had to desolder and resolder the magnetometer from pre-existing board. Also, we were short on resistor and capacitor components, so we used the values that were closest to our original values. Most likely the weak signal was caused because we did not tune the antenna that we purchased. On the PCB we have all tuning capacitor, resistor, inductors are shorted.

Related work

Previous to our implementation we came across an article that sparked our current product. Provided is a link to NXP research where they used the exact same magnetometer to detect cars driving above the sensor [1]. Because of this, going into this project we already know it was proved that the magnetometer is able to detect cars driving over. As our implementation started we saw similar results as reported by the article. Going further we came up with our own car detection algorithm and integrated the systems from the sensor to UI level.

Additionally, it appears that magnetometers are being used in a few commercial systems that detect the frequency that vehicles pass over it as a replacement for a large inductive loop. In comparison, our group uses the magnetometers to detect cars in place.

Teamwork report

Rajdeep Singh - data collection, car detection algorithm

Alexander Huttunen - Si4010 dev, finalized GNURadio receiver and packet decoder, system integration, backend, web interface, database

Jisoo Jung - Si4010 dev, test and debugging, solving packet conflict on Python decoder side

Yaning Yang - initial design of GNURadio receiver, designed PCB

Appendix

Si4010 Firmware

```
//SenseUnit.h

#ifndef _SENSE_UNIT_H
#define _SENSE_UNIT_H
#include "si4010_types.h"

#define bFStepDemo_MaxFrameSize_c (125)
#define bFStepDemo_FreqNum_c (1)
#define INT_MIN -32767
#define INT_MAX 32767
#define PACKET_SIZE 10 // including preamble, heading, payload
#define THRESHOLD 50
#define BUFFER_SIZE 20

//Types
typedef BYTE tFStepDemo_Data[bFStep_DataLength_c];
tFStepDemo_Data xdata arFStepDemo_Data[bFStepDemo_FreqNum_c];

tOds_Setup xdata rOdsSetup;
tPa_Setup xdata rPaSetup;

unsigned char ack;

// Data order: magx magz magy
short xdata magx;
short xdata magz;
short xdata magy;

LWORD xdata prodID;
BYTE xdata packetNumber;

// ----- Variables used in Car Detection -----
// variables in order to implement moving average
int xdata average_X = 0;
int xdata average_Z = 0;

int xdata counter = 1;
int xdata index = 0;

int xdata sampleSize_X[BUFFER_SIZE]; // store 20 samples of mag data
int xdata sampleSize_Z[BUFFER_SIZE];

BYTE xdata found = 0;
```

```
// Calibration data used when Si4010 wakes up from low power mode
// Only bottom 3 bits of the address will be used.
BYTE xdata calibAddrX = 0x00; // addr 0x00, 0x01 stores int value for X-axis
BYTE xdata calibAddrZ = 0x02; // addr 0x02, 0x03 stores int value for Z-axis
BYTE xdata packetNumberAddr = 0x04; // addr 0x04 stores BYTE value for packetNumber
```

```
/**
PACKET DEFINITION
- preamble: 2 bytes for 0xFF3D
- header: 4 bytes for ID
- payload: {
    1 byte { car status (parked or not) 1bit, packet number 7bits}
    1 byte temperature (Celcius),
    2 bytes battery voltage unit (mV)
}
- checksum: no checksum for now
```

```
TOTAL PACKET LENGTH = 10
```

Type	Bit-Width	Type-Definition	Prefix
unsigned char	8	BYTE	b
unsigned int	16	WORD	w
unsigned long int	32	LWORD	l
signed char	8	CHAR	c
int	16		i
long int	32		j
float	32		f

```
*/

#endif /* _SENSE_UNIT_H */
```

```
//SenseUnit.c
```

```
#include <stdlib.h>
#include <math.h>

#include "si4010.h"
#include "si4010_api_rom.h"
#include "si4010_api_add.h"
#include "si4010_types.h"
#include "i2c.h"
#include "SenseUnit.h"
```

```
//Function Prototypes
```

```
void Delay(void);
```



```

void randomDelay(void);
void sendData(BYTE xdata * dataToSend, BYTE len);
void invertBits(BYTE xdata * input, BYTE len);
void makePacket(BYTE xdata * dataBuffer);
int averageArray(int* array);
int performDetection();
void system_sleep_si4010();
void init_si4010();
void read_HMC5883L();
int isFirstPower();
// -----

void main(void) {
    BYTE xdata packet[PACKET_SIZE];
    // Initialize si4010 and I2C
    init_si4010();
    I2CInit();

    // Write to slave device
    I2CStart();
    ack = I2CSend(0x3C); //0x3C is magnetometer address
    ack = I2CSend(0x02); //select mode register
    ack = I2CSend(0x00); //continuous measurement
    I2CStop();

    Delay();
    while ( 1 ) {
        found = performDetection();

        //Turn off magnetometer (seems to save an additional ~10uA)
        I2CInit();
        I2CStart();
        ack = I2CSend(0x3C); //0x3C is magnetometer address
        ack = I2CSend(0x03); ///select mode register
        ack = I2CSend(0x02); ///continuous measurement
        I2CStop();

        //Create and send packets
        makePacket(packet);
        invertBits(packet, PACKET_SIZE);
        sendData((BYTE*) packet, PACKET_SIZE);
        randomDelay();
        sendData((BYTE*) packet, PACKET_SIZE);
        randomDelay();
        sendData((BYTE*) packet, PACKET_SIZE);
        randomDelay();
        sendData((BYTE*) packet, PACKET_SIZE);
        randomDelay();

        system_sleep_si4010();
    }
}

```

```

    }
}

//Set the Si4010 into a low power mode after saving state variables to HVRAM
void system_sleep_si4010() {
    // write to HVRAM. Data written in big endian
    // Memory layout:
    // addr: [0x00, 0x01, 0x02, 0x03, 0x04]
    // data: [MSB_X, LSB_X, MSB_Z, LSB_Z, packetNumber]
    vHvram_Write(packetNumberAddr, packetNumber);

    vSleepTim_SetCount(0x01006666); //LWORD(32bit) Value of 24 bit counter (for about 10
seconds) plus a power on bit in position 24 (25th bit)
    vSys_Shutdown();
}

int isFirstPower() {
    return ((SYSGEN & M_POWER_1ST_TIME) >> B_POWER_1ST_TIME) || !GPIO9; //Ground GPIO9 to force
initial power
}

void read_HMC5883L() {
    // Tell master to read from address 0x3C
    I2CStart();
    I2CSend(0x3C);
    I2CSend(0x03);
    I2CStop();

    I2CStart();
    I2CSend(0x3C | 1);

    magx = I2CRead();
    I2CAck();
    magx = magx << 8;
    magx |= I2CRead();
    I2CAck();

    magz = I2CRead();
    I2CAck();
    magz = magz << 8;
    magz |= I2CRead();
    I2CAck();

    magy = I2CRead();
    I2CAck();
    magy = magy << 8;
    magy |= I2CRead();
    I2CNak();

    I2CStop();
}

```

```

}

void init_si4010() {
    // Disable the Matrix and Roff modes on GPIO[3:1]
    PORT_CTRL &= ~(M_PORT_MATRIX | M_PORT_ROFF | M_PORT_STROBE);
    PORT_CTRL |= M_PORT_STROBE;
    PORT_CTRL &= (~M_PORT_STROBE);

    // Setup
    PDMD=1;
    vSys_Setup( 1 );
    vSys_BandGapLdo( 1 );
    rPaSetup.fAlpha      = 0;
    rPaSetup.fBeta       = 0;
    rPaSetup.bLevel      = 80; //77 is loud, 20 is quiet, 127 is max
    rPaSetup.wNominalCap = 192;
    rPaSetup.bMaxDrv     = 1;
    vPa_Setup( &rPaSetup );

    rOdsSetup.bModulationType = 1; // Use FSK
    rOdsSetup.bClkDiv         = 5;
    rOdsSetup.bEdgeRate       = 0;
    rOdsSetup.bGroupWidth     = 7;
    rOdsSetup.wBitRate        = 0x61B; //80 higher is slower
    rOdsSetup.bLcWarmInt      = 0; //0
    rOdsSetup.bDivWarmInt     = 5; //5
    rOdsSetup.bPaWarmInt      = 4; //4
    vOds_Setup( &rOdsSetup );

    vStl_EncodeSetup( bEnc_Manchester_c, NULL );
    vFCast_Setup();

    vFCast_Tune( 440001227 );
    bFStep_Collect( (BYTE xdata *) &(arFStepDemo_Data[0]) );

    prodID = lSys_GetProdId();
    packetNumber = 0;

    srand(prodID); // seed random number generator with si4010 device ID

    // Data written in big endian
    // Memory layout:
    // addr: [0x00, 0x01, 0x02, 0x03, 0x04]
    // data: [MSB_X, LSB_X, MSB_Z, LSB_Z, packetNumber]
    if (!isFirstPower()) {
        packetNumber = bHvram_Read(packetNumberAddr);
    } else {
        vHvram_Write(packetNumberAddr, 0x00);
    }
}

```

```

// -----

/**
 * sendData(data, length)
 * Params:
 *   - dataToSend: data with 'len' many bytes
 *   - len: number of bytes to send
 */
void sendData(BYTE xdata * dataToSend, BYTE len) {
    vDmdTs_RunForTemp( 3 ); // Skip 3 samples.

    vFStep_Apply( (BYTE xdata *) &(arFStepDemo_Data[0]) ); // Tune to the saved frequency
    vFCast_FskAdj( 12 ); //how far apart the FSK signals are

    // Wait until there is a demodulated temperature sample
    while ( 0 == bDmdTs_GetSamplesTaken() ) {}
    vPa_Tune( iDmdTs_GetLatestTemp() );
    vStl_PreLoop();
    vStl_SingleTxLoop(dataToSend, len);
    vStl_PostLoop();
}

void invertBits(BYTE xdata * input, BYTE len) {
    int i;
    for (i = 0; i < len; i++) {
        input[i] = ~input[i];
    }
}

// Packet generation function
/*
PACKET DEFINITION
- preamble: 2 bytes for 0xFF3D
- header: 4 bytes for ID
- payload: {
1 byte { car status (parked or not) 1bit, packet number 7bits}
1 byte temperature in celcius,
2 bytes battery voltage unit (mV)
}
- checksum: no checksum for now
*/
void makePacket(BYTE xdata * packet) {
    unsigned int battery_mV;
    battery_mV = (unsigned int) iMVdd_Measure(0);

    // Preamble
    packet[0] = 0xFF; //Preamble 01010101010101
    packet[1] = 0x3D; //Preamble 0110010101011011

```

```

// Header: 4 byte ID which is unsigned long int
packet[2] = (BYTE) ((prodID >> 24) & 0xFF);
packet[3] = (BYTE) ((prodID >> 16) & 0xFF);
packet[4] = (BYTE) ((prodID >> 8) & 0xFF);
packet[5] = (BYTE) (prodID & 0xFF);

// Payload: car status, packet number, temp, battery voltage
packet[6] = ((found << 7) | packetNumber);
vDmdTs_RunForTemp( 3 ); // Skip 3 samples.
while ( 0 == bDmdTs_GetSamplesTaken() ) {}
packet[7] = iDmdTs_GetLatestTemp() / 220 + 25;
packet[8] = (BYTE) ((battery_mV & 0xFF00) >> 8);
packet[9] = (BYTE) (battery_mV & 0xFF);

// packeNumber range: [0, 127]
packetNumber = (packetNumber + 1) % 0x7F;
}

void Delay(void) {
    int j;
    int i;
    for(i=0;i<10;i++) {
        for(j=0;j<10000;j++);
    }
}

void randomDelay(void) {
    // wiIntervalCount is of type WORD.
    // MAX_INTERVAL = 2^16 = 65536
    // This delay cycle ranges [63536, 65536)
    vSys_16BitDecLoop(63536 + rand() % 2000);
}

// ----- CAR DETECTION ALGORITHM -----
//returns the result of if a car is in the parking spot or not
int performDetection() {
    short i;
    int difference_X;
    int difference_Z;
    int signalAverage_X;
    int signalAverage_Z;

    //get 20 samples
    for (i = 0; i < BUFFER_SIZE; i++) {
        read_HMC5883L(); //get fresh magnetometer data
        sampleSize_X[i] = magx; // add new value to array
        sampleSize_Z[i] = magz;
        Delay();
    }
}

```

```

//average samples
average_X = averageArray(sampleSize_X);
average_Z = averageArray(sampleSize_Z);

//save average at power up to compare against later
if (isFirstPower()) {
    signalAverage_X = average_X;
    signalAverage_Z = average_Z;
    vHvram_Write(calibAddrX, ((signalAverage_X >> 8) & 0xFF));
    vHvram_Write(calibAddrX + 1, (signalAverage_X & 0xFF));
    vHvram_Write(calibAddrZ, ((signalAverage_Z >> 8) & 0xFF));
    vHvram_Write(calibAddrZ + 1, (signalAverage_Z & 0xFF));
    return 0; //assuming no car at power up
}
signalAverage_X = (int) bHvram_Read(calibAddrX) << 8;
signalAverage_X |= bHvram_Read(calibAddrX + 1);
signalAverage_Z = (int) bHvram_Read(calibAddrZ) << 8;
signalAverage_Z |= bHvram_Read(calibAddrZ + 1);

//compare magnetometer values to find car
difference_X = abs(average_X - signalAverage_X); // x axis or y
difference_Z = abs(average_Z - signalAverage_Z); // z axis

return (difference_X >= THRESHOLD || difference_Z >= THRESHOLD);
}

// Returns an average of array
int averageArray(int* ar) {
    int i;
    int average = 0;
    for(i = 0; i < BUFFER_SIZE; i++) {
        average += ar[i];
    }
    return (average / BUFFER_SIZE);
}


```

```

//I2C.h

void I2CInit();

void I2CStart();

//void I2CRestart();

void I2CStop();

void I2CAck();

void I2CNak();

```

```

unsigned char I2CSend(unsigned char Data);

unsigned char I2CRead();

void i2c_delay(void);

```

```

//I2C.c
//Based on: http://www.8051projects.net/wiki/I2C\_Implementation\_on\_8051

#include "I2C.h"
#include "si4010.h"

#define SDA GPIO3
#define SCL GPIO2

void I2CInit()
{
    SDA = 1;
    i2c_delay();
    SCL = 1;
    i2c_delay();
}

void I2CStart()
{
    SDA = 0;
    i2c_delay();
    SCL = 0;
    i2c_delay();
}

/*void I2CRestart()
{
    SDA = 1;
    i2c_delay();
    SCL = 1;
    i2c_delay();
    SDA = 0;
    i2c_delay();
    SCL = 0;
    i2c_delay();
}*/

void I2CStop()
{
    SCL = 0;
    i2c_delay();
    SDA = 0;
    i2c_delay();
    SCL = 1; //1

```

```

        i2c_delay();
        SDA = 1; //1
        i2c_delay();
    }

    void I2CAck()
    {
        SDA = 0;
        i2c_delay();
        SCL = 1;
        i2c_delay();
        SCL = 0;
        i2c_delay();
        SDA = 1;
        i2c_delay();
    }

    void I2CNak()
    {
        SDA = 1;
        i2c_delay();
        SCL = 1;
        i2c_delay();
        SCL = 0;
        i2c_delay();
        SDA = 1;
        i2c_delay();
    }

    unsigned char I2CSend(unsigned char Data)
    {
        unsigned char i, ack_bit;
        for (i = 0; i < 8; i++) {
            if ((Data & 0x80) == 0)
                SDA = 0;
            else
                SDA = 1;
            i2c_delay();
            SCL = 1;
            i2c_delay();
            SCL = 0;
            Data<<=1;
            i2c_delay();
        }
        SDA = 1;
        i2c_delay();
        SCL = 1;
        i2c_delay();
        ack_bit = SDA;
        SCL = 0;
    }

```



```

        i2c_delay();
        return ack_bit;
    }

    unsigned char I2CRead()
    {
        unsigned char i, Data=0;
        for (i = 0; i < 8; i++) {
            SCL = 1;
            i2c_delay();
            if(SDA)
                Data |=1;
            if(i<7)
                Data<<=1;
            SCL = 0;
            i2c_delay();
        }
        return Data;
    }

    void i2c_delay(void) {
        int i;
        for(i=0;i<6;i++)
        {
        }
    }

```

Note: Libraries that this code needs to compile using Keil for the Si4010 can be found at <http://www.silabs.com/products/wireless/EZRadio/Pages/Si4010.aspx>

Raspberry Pi Radio and Decoding

```

#!/usr/bin/env python2
#####
# GNU Radio Python Flow Graph
# Title: Top Block
# Generated: Mon May  9 18:11:22 2016
#####

from gnuradio import analog
from gnuradio import blocks
from gnuradio import digital
from gnuradio import eng_notation
from gnuradio import filter
from gnuradio import gr
from gnuradio.eng_option import eng_option
from gnuradio.filter import firdes

```

```

from grc_gnuradio import blks2 as grc_blks2
from optparse import OptionParser
import math
import osmosdr
import time

class top_block(gr.top_block):

    def __init__(self):
        gr.top_block.__init__(self, "Top Block")

        #####
        # Variables
        #####
        self.variable_slider_0 = variable_slider_0 = .846
        self.test = test = .005
        self.shift = shift = .906
        self.samp_rate_0 = samp_rate_0 = 1.2e6
        self.samp_rate = samp_rate = 1.2e6/4
        self.pows = pows = 1.3
        self.lpf = lpf = .724
        self.go = go = 0.564
        self.gm = gm = 1.61
        self.centre_freq = centre_freq = 439.95e6

        #####
        # Blocks
        #####
        self.rtlsdr_source_0 = osmosdr.source( args="numchan=" + str(1) + " " + " " )
        self.rtlsdr_source_0.set_sample_rate(samp_rate_0)
        self.rtlsdr_source_0.set_center_freq(439.9e6, 0)
        self.rtlsdr_source_0.set_freq_corr(0, 0)
        self.rtlsdr_source_0.set_dc_offset_mode(0, 0)
        self.rtlsdr_source_0.set_iq_balance_mode(2, 0)
        self.rtlsdr_source_0.set_gain_mode(False, 0)
        self.rtlsdr_source_0.set_gain(20, 0)
        self.rtlsdr_source_0.set_if_gain(10, 0)
        self.rtlsdr_source_0.set_bb_gain(10, 0)
        self.rtlsdr_source_0.set_antenna("", 0)
        self.rtlsdr_source_0.set_bandwidth(0, 0)

        self.low_pass_filter_1 = filter.fir_filter_fff(10, firdes.low_pass(
            1, samp_rate, 2.56e3*lpf, (2.56e3/2)*lpf, firdes.WIN_HAMMING, 6.76))
        self.freq_xlating_fir_filter_xxx_0 = filter.freq_xlating_fir_filter_ccc(4,
            (firdes.low_pass_2(1, samp_rate_0, 100e3, 50e3, 40)), 0, samp_rate_0)
        self.digital_clock_recovery_mm_xx_1 = digital.clock_recovery_mm_ff(11.6439*(1+test),
            0.25*0.175*0.175*go, 0.5, 0.175*gm, 0.005*variable_slider_0)
        self.digital_binary_slicer_fb_0 = digital.binary_slicer_fb()
        self.blocks_add_const_vxx_0 = blocks.add_const_vff((-12*shift, ))

```

```

self.blks2_tcp_sink_0 = grc_blks2.tcp_sink(
    itemsize=gr.sizeof_char*1,
    addr="127.0.0.1",
    port=9000,
    server=False,
)
self.analog_quadrature_demod_cf_0 = analog.quadrature_demod_cf(10)
self.analog_pwr_squelch_xx_0 = analog.pwr_squelch_cc(-40*pows, .001, 0, False)

#####
# Connections
#####
self.connect((self.analog_pwr_squelch_xx_0, 0), (self.analog_quadrature_demod_cf_0,
0))
self.connect((self.analog_quadrature_demod_cf_0, 0), (self.low_pass_filter_1, 0))
self.connect((self.blocks_add_const_vxx_0, 0), (self.digital_clock_recovery_mm_xx_1,
0))
self.connect((self.digital_binary_slicer_fb_0, 0), (self.blks2_tcp_sink_0, 0))
self.connect((self.digital_clock_recovery_mm_xx_1, 0),
(self.digital_binary_slicer_fb_0, 0))
self.connect((self.freq_xlating_fir_filter_xxx_0, 0), (self.analog_pwr_squelch_xx_0,
0))
self.connect((self.low_pass_filter_1, 0), (self.blocks_add_const_vxx_0, 0))
self.connect((self.rtlsdr_source_0, 0), (self.freq_xlating_fir_filter_xxx_0, 0))

def get_variable_slider_0(self):
    return self.variable_slider_0

def set_variable_slider_0(self, variable_slider_0):
    self.variable_slider_0 = variable_slider_0

def get_test(self):
    return self.test

def set_test(self, test):
    self.test = test
    self.digital_clock_recovery_mm_xx_1.set_omega(11.6439*(1+self.test))

def get_shift(self):
    return self.shift

def set_shift(self, shift):
    self.shift = shift
    self.blocks_add_const_vxx_0.set_k((-12*self.shift, ))

def get_samp_rate_0(self):
    return self.samp_rate_0

def set_samp_rate_0(self, samp_rate_0):

```

```

        self.samp_rate_0 = samp_rate_0

self.freq_xlating_fir_filter_xxx_0.set_taps((firdes.low_pass_2(1,self.samp_rate_0,100e3,50e3,4
0)))
        self.rtlsdr_source_0.set_sample_rate(self.samp_rate_0)

    def get_samp_rate(self):
        return self.samp_rate

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate
        self.low_pass_filter_1.set_taps(firdes.low_pass(1, self.samp_rate, 2.56e3*self.lpf,
(2.56e3/2)*self.lpf, firdes.WIN_HAMMING, 6.76))

    def get_pows(self):
        return self.pows

    def set_pows(self, pows):
        self.pows = pows
        self.analog_pwr_squelch_xx_0.set_threshold(-40*self.pows)

    def get_lpf(self):
        return self.lpf

    def set_lpf(self, lpf):
        self.lpf = lpf
        self.low_pass_filter_1.set_taps(firdes.low_pass(1, self.samp_rate, 2.56e3*self.lpf,
(2.56e3/2)*self.lpf, firdes.WIN_HAMMING, 6.76))

    def get_go(self):
        return self.go

    def set_go(self, go):
        self.go = go
        self.digital_clock_recovery_mm_xx_1.set_gain_omega(0.25*0.175*0.175*self.go)

    def get_gm(self):
        return self.gm

    def set_gm(self, gm):
        self.gm = gm
        self.digital_clock_recovery_mm_xx_1.set_gain_mu(0.175*self.gm)

    def get_centre_freq(self):
        return self.centre_freq

    def set_centre_freq(self, centre_freq):
        self.centre_freq = centre_freq

```

```

if __name__ == '__main__':
    parser = OptionParser(option_class=eng_option, usage="%prog: [options]")
    (options, args) = parser.parse_args()
    tb = top_block()
    tb.start()
    try:
        raw_input('Press Enter to quit: ')
    except EOFError:
        pass
    tb.stop()
    tb.wait()

#####
#!/usr/local/bin/python3.5

####
#Process Radio - Sink for FSK GNU Radio Demodulator
####

import socket
import redis

# precondition: n should be 16bit integer
def unsigned2signed(n):
    if n >= 2**15:
        n -= (2**16)
    return n

TCP_IP = '127.0.0.1'
TCP_PORT = 9000

print("INFO: Server for GNURadio Launched")

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((TCP_IP, TCP_PORT))
s.listen(1)

r = redis.StrictRedis(host='localhost', port=6379, db=0)

#preamble starts with a 0 - not included in array size:32 - 1 = 31 elements -> index 30
preamble = ( 1, 0, 1, 0, 1, 0, 1,
              0, 1, 0, 1, 0, 1, 0, 1,
              0, 1, 1, 0, 0, 1, 0, 1,
              0, 1, 0, 1, 1, 0, 1, 0)
bytesInFrame = 8

while True:
    connection, ClientIP = s.accept() # Wait for a connection
    print(ClientIP)

    preamblepoint = 0

```

```

bitbuf = 0
bitstep = 0
frame = bytearray()

ID_map = {}
prevPacketNumber = None
prevID = None
count = 0
while True:
    data = connection.recv(1) #GNU Radio gives us a one or zero here
    if (preamblepoint == 31): #start processing the packet
        #print("processing...")

        if (data == b'\x01'):
            prevbit = 1
        else:
            prevbit = 0
        data = connection.recv(1) #get another number
        if (data == b'\x01'):
            if (prevbit == 0): #01 -> 1
                dbit = 1
            else:
                #11 -> clock error
                print("clock desync error")
                preamblepoint = 0
                frame = bytearray() #clear frame
        else:
            if (prevbit == 1): #10 -> 0
                dbit = 0
            else:
                #00 -> clock error
                print("clock desync error")
                preamblepoint = 0
                frame = bytearray() #clear frame

        if (dbit == 1):
            bitbuf |= 0b10000000
        bitstep += 1
        if (bitstep % 8 == 0):
            frame.append(bitbuf)
            bitbuf = 0
            if (bitstep == 8 * bytesInFrame):
                bitstep = 0
                #print("***NEW PACKET***")
                preamblepoint = 0
                last = 0

            #-----PACKET PROCESSING-----
            if (len(frame) > 4):
                sendDeviceID = frame[3] + (frame[2] << 8) + (frame[1] << 16) +
(frame[0] << 24)

                batteryVoltage = frame[7] + (frame[6] << 8)

```

```

temperature = frame[5]
packetNumber = frame[4] & 0x7F
carStatus = (frame[4] & 0x80) != 0

'''-----PACKET DEBUG-----'''
# Error handling showing N / 4 confidence percentage
if (prevPacketNumber != None and prevPacketNumber != packetNumber):
    print("***NEW PACKET***")
    print("-->PrevPacket number %d" % prevPacketNumber)
    print("-->ID %d: %d / 4 packets received" % (prevID, count))
    count = 1
else:
    count = count + 1
    prevPacketNumber = packetNumber
    prevID = sendDeviceID

# Handle duplicate packet sent from si4010
if (str(sendDeviceID) in ID_map):
    if (ID_map[str(sendDeviceID)] == packetNumber):
        frame = bytearray()
        continue
    else:
        ID_map[str(sendDeviceID)] = packetNumber
else:
    ID_map[str(sendDeviceID)] = packetNumber
'''-----'''

print("Sent From : " + str(sendDeviceID))
print("Voltage : " + str(batteryVoltage))
print("Temperature: " + str(temperature))
print("Packet Num : " + str(packetNumber))
print("Car : " + str(carStatus))
print("FRAMEDEBUG : " + str(bin(frame[4])))
r.publish("radio_pkt", "%d %d %d %d %s" % (sendDeviceID,
batteryVoltage, temperature, packetNumber, carStatus))
#-----

else:
    print("Malformed Packet");
    frame = bytearray() #clear frame
else:
    bitbuf = bitbuf >> 1
else:
    if (data == b'\x01'):
        if(preamble[preamblepoint] == 1):
            preamblepoint += 1
        else:
            preamblepoint = 0
    elif (data == b'\x00'):
        if(preamble[preamblepoint] == 0):
            preamblepoint += 1

```

```
    else:
        preamblepoint = 0
```

```
#!/usr/bin/python3
```

```
import subprocess
subprocess.Popen(["./processradio.py"]) #Server for GNURadio to connect to
subprocess.Popen(["./top_block.py"]) #Compiled GNURadio Script
```

Flask Web Server and Database

/initDB.py

```
#!/usr/local/bin/python3.5
#from migrate.versioning import api
import os.path
import os
os.remove("./park.db")
from config import SQLALCHEMY_DATABASE_URI
from config import SQLALCHEMY_MIGRATE_REPO
from app import db

db.create_all()
```

/config.py

```
import os
basedir = os.path.abspath(os.path.dirname(__file__))

SQLALCHEMY_DATABASE_URI = 'sqlite:/// ' + os.path.join(basedir, 'park.db')
SQLALCHEMY_MIGRATE_REPO = os.path.join(basedir, 'db_repository')
SQLALCHEMY_TRACK_MODIFICATIONS = False

SECRET_KEY = '*****'
```

/run.py

```
#!/usr/local/bin/python3.5
from app import app
app.debug = True #auto reload on code change + debug messages
app.run(host='0.0.0.0', use_reloader=False)
```

/app/views.py

```
from app import app, db

from flask import render_template, flash, url_for, redirect, g
```



```

from datetime import datetime
from sqlalchemy import desc, func
from sqlalchemy.orm import aliased

import redis #for IPC with radio parser

from .forms import DataForm, ParkingSpaceForm
from .models import ParkPacket, Sensor

def redis_thread():
    app.logger.info("Redis Init")
    r = redis.StrictRedis(host="localhost", port=6379, db=0)
    r = r.psubsub()
    r.subscribe("radio_pkt")
    while True:
        for m in r.listen():
            if m['type'] == "message":
                sendDeviceID, batteryVoltage, \
                temperature, packetNumber, \
                carStatus = m['data'].decode('utf-8').split(" ", 4)

                newData(sendDeviceID, batteryVoltage, temperature, \
                packetNumber, carStatus)

#Called when there is updated information about the state of the parking lot
def newData(sendDeviceID, batteryVoltage, temperature, packetNumber, carStatus):
    print("New DATA from" + sendDeviceID)
    dataIn = ParkPacket(
        sensorId=sendDeviceID,
        occupied=(carStatus == 'True'),
        temperature=temperature,
        battVolt=batteryVoltage,
        timestamp=datetime.utcnow()
    )
    db.session.add(dataIn)
    db.session.commit()

@app.route('/')
def home_page():

    ''' What the following SQLALCHEMY query should do
    SELECT pp.*
    FROM
        park_packet as pp,
        (SELECT sensorID, MAX(timestamp) as maxts
        FROM park_packet
        GROUP BY sensorID) as maxpp
    WHERE pp.sensorID = maxpp.sensorID
    AND pp.timestamp = maxpp.maxts;
    ...

```

```

subq = db.session.query(ParkPacket.sensorId.label('sid'), \
    func.max(ParkPacket.timestamp).label('mts')). \
    group_by(ParkPacket.sensorId).subquery()
sensors = db.session.query(ParkPacket). \
    filter(ParkPacket.sensorId == subq.c.sid, \
    ParkPacket.timestamp == subq.c.mts)

lot1 = db.session.query(Sensor)

print(lot1)

return render_template('home.html', parkingSpots=sensors, lotLayout=lot1)

@app.route('/sensorin', methods=["GET", "POST"])
def sensorin():
    form = DataForm()

    if form.validate_on_submit():
        dataIn = ParkPacket(
            sensorId=form.sensorId.data,
            occupied=(form.occupied.data == 'True'),
            temperature=form.temperature.data,
            battVolt=form.battVolt.data,
            timestamp=datetime.utcnow()
        )
        db.session.add(dataIn)
        db.session.commit()

    return render_template('sensorin.html', form=form)

@app.route('/setup', methods=["GET", "POST"])
def lotsetup():
    form = ParkingSpaceForm()

    if form.validate_on_submit():
        dataIn = Sensor(
            uid = form.uid.data,
            parkingspace = form.parkingspace.data,
            rotation = form.rotation.data,
            leftCoord = form.leftCoord.data,
            topCoord = form.topCoord.data
        )
        db.session.add(dataIn)
        db.session.commit()

    return render_template('lotsetup.html', form=form)

@app.errorhandler(404)
def page_not_found(error):

```

```
app.logger.error('404d')
return redirect(url_for("home_page"))
```

/app/models.py

```
from sqlalchemy.ext.hybrid import hybrid_property

from app import db

class Sensor(db.Model):
    uid = db.Column(db.Integer, primary_key = True)
    parkingspace = db.Column(db.String(26)) #name of parking space
    rotation = db.Column(db.String(1)) #b l t r
    leftCoord = db.Column(db.Integer) #each "unit" moves space by 1/2
    topCoord = db.Column(db.Integer)

class ParkPacket(db.Model):
    packetId = db.Column(db.Integer, primary_key = True) #incrementing unique ID
    sensorId = db.Column(db.Integer, db.ForeignKey('sensor.uid'))
    timestamp = db.Column(db.DateTime)
    occupied = db.Column(db.Boolean)
    temperature = db.Column(db.Integer)
    battVolt = db.Column(db.Integer)
```

/app/forms.py

```
from flask.ext.wtf import Form
from wtforms import StringField, BooleanField
from wtforms.validators import DataRequired

class DataForm(Form):
    sensorId = StringField('sensorId', validators=[DataRequired()])
    occupied = StringField('occupied', validators=[DataRequired()])
    temperature = StringField('temperature', validators=[DataRequired()])
    battVolt = StringField('battVolt', validators=[DataRequired()])

class ParkingSpaceForm(Form):
    uid = StringField('uid', validators=[DataRequired()])
    parkingspace = StringField('parkingspace', validators=[DataRequired()])
    topCoord = StringField('topCoord', validators=[DataRequired()])
    leftCoord = StringField('leftCoord', validators=[DataRequired()])
    rotation = StringField('rotation', validators=[DataRequired()])
```

/app/__init__.py

```
from flask import Flask
from flask.ext.sqlalchemy import SQLAlchemy
import threading
```

```

app = Flask(__name__)
app.config.from_object('config')
db = SQLAlchemy(app)

from app import views, models

t = threading.Thread(target=views.redis_thread, args=())
t.setDaemon(True)
t.start()

```

```

/app/static/css/style.css

```

```

.status_display {
    text-align:center;
}

.parking_lot {
    background-color: #555;
    border: 20px #555 solid;
    display: inline-block;
    min-width: 630px;
    min-height: 450px;
    position: relative;
}

.parking_space_info {
    color: #FFF;
    background-color: #111;
    position: absolute;
    top: 0;
    left: 0;
}

.parking_spot_number {
    color: #FFF;
    font-size: 2em;
}

.parking_space {
    border-top: 4px solid #DDD;
    border-right: 4px solid #DDD;
    border-left: 4px solid #DDD;
    border-bottom: 4px solid #DDD;
    height: 200px;
    width: 100px;
    display: inline-block;
    vertical-align: top;
    padding-top: 50px;
    margin: -4px;
    position: absolute;
}

```

```

        top:0px;
        left:-200px;
    }

    i.icon {
        font-size: 5em;
    }

    i.icon_car {
        color: #FF7F00;
    }

    i.icon_open {
        color: #0F0;
    }

```

/app/templates/base.html

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>{% block title %}{% endblock %}</title>
    <link rel="shortcut icon" href="/static/img/favicon.ico">
    <link rel="stylesheet" href="/static/css/style.css">
    <link rel="stylesheet" href="/static/css/font-awesome.min.css">
    <link rel="stylesheet" href="/static/css/bootstrap.min.css" rel="stylesheet">
    <script src="/static/js/jquery-1.12.4.min.js"></script>
    <script src="/static/js/bootstrap.min.js"></script>
    <!--[if IE]>
        <script src="http://html5shiv.googlecode.com/svn/trunk/html5.js"></script>
    <![endif]-->
    {% block specificHead %}
    {% endblock %}
</head>

<body id="home">
{% block content %}
{% endblock %}
</body>
</html>

```

/app/templates/home.html

```

{% extends "base.html" %}

{% block title %}Parking Lot{% endblock %}

{% block specificHead %}
    <meta http-equiv="refresh" content="3">

```

```

<style>
  {% for spot in lotLayout %}
  #pksp_{{spot.uid}} {
    left: {{(100/2) * spot.leftCoord}}px;
    top: {{(100/2) * spot.topCoord}}px;

    {% if spot.rotation == 'l' %}
      border-bottom: none;
      transform: rotate(90deg);
    {% endif %}
    {% if spot.rotation == 'r' %}
      border-bottom: none;
      transform: rotate(-90deg);
    {% endif %}
    {% if spot.rotation == 'u' %}
      border-top: none;
    {% endif %}
    {% if spot.rotation == 'd' %}
      border-bottom: none;
    {% endif %}
  }
  {% endfor %}
</style>
{% endblock %}

{% block content %}
  <div class="container status_display">
    <h1>Current Parking Lot:</h1>
    <div class="parking_lot">

      {% for spot in parkingSpots %}
      <div id="pksp_{{ spot.sensorId }}" class="parking_space">
        {% if spot.occupied == True %}
          <i class="icon icon_car fa fa-car"></i>
        {% else %}
          <i class="icon icon_open fa fa-circle-o"></i>
        {% endif %}
        <div class="parking_spot_number">
          {{ spot.sensorId }}
        </div>
        <div class="parking_space_info">
          <i class="fa fa-battery-three-quarters"> {{ spot.battVolt
}}</i>

          <!--<i class="fa fa-clock-o"> {{ spot.timestamp }} </i>-->
          <i class="fa fa-sun-o"> {{ spot.temperature }}&deg;C</i>
        </div>
      </div>
      {% endfor %}
    </div>
  </div>

```

```

    </div>
{% endblock %}

/app/templates/lotsetup.html

{% extends "base.html" %}
{% block content %}
    <h1>Parking Lot Setup</h1>
    <form action="{{ url_for('lotsetup') }}" method="post">
        {{ form.hidden_tag() }}
        <label for="uid">Sensor ID:</label> {{ form.uid(class_="form-control") }}<br>
        <label for="name">Space Name:</label> {{
form.parkingspace(class_="form-control") }}<br>
        <label for="topCoord">Top Coord:</label> {{ form.topCoord(class_="form-control")
}}<br>
        <label for="leftCoord">Left Coord:</label> {{
form.leftCoord(class_="form-control") }}<br>
        <label for="rotation">Rotation:</label> {{ form.rotation(class_="form-control")
}}<br>
        <input type="submit" class="btn btn-default btn-success" value="Post">
    </form>
{% endblock %}

```

```

/app/templates/sensorin.html

{% extends "base.html" %}
{% block content %}
    <h1>Test data ingest</h1>
    <form action="{{ url_for('sensorin') }}" method="post">
        {{ form.hidden_tag() }}
        <label for="sensorId">Sensor ID:</label> {{ form.sensorId(class_="form-control")
}}<br>
        <label for="temperature">Temperature:</label> {{
form.temperature(class_="form-control") }}<br>
        <label for="occupied">Occupied:</label> {{ form.occupied(class_="form-control")
}}<br>
        <label for="battVolt">Battery Voltage:</label> {{
form.battVolt(class_="form-control") }}<br>
        <input type="submit" class="btn btn-default btn-success" value="Post">
    </form>
{% endblock %}

```

Si4010 Support Scripts

(for use with si4010prog on the Raspberry Pi)

setup.sh

```
#!/bin/bash
echo Setting up C2 Programmer...
sudo insmod c2_gpio.ko
sudo mknod /dev/c2_bus c 243 0
sudo chmod 777 /dev/c2_bus
./siprog -d c2drv:///dev/c2_bus reset identify
```

progit.sh

```
#!/bin/bash
./siprog -d c2drv:///dev/c2_bus reset prg:finalprog.hex run
```

Makefile (makefile that runs in Windows OS)

keilpath := D:\Keil_v5\C51\BIN

main:

```
    ${keilpath}\C51.exe SenseUnit.c DB OE BR BROWSE NOAREGS NOINTPROMOTE DEBUG OBJECTTEXTEND
CODE SYMBOLS INCDIR(common\src)
    ${keilpath}\C51.exe isr_dmd.c DB OE BR BROWSE NOAREGS NOINTPROMOTE DEBUG OBJECTTEXTEND
CODE SYMBOLS INCDIR(common\src)
    ${keilpath}\C51.exe i2c.c DB OE BR BROWSE NOAREGS NOINTPROMOTE DEBUG OBJECTTEXTEND CODE
SYMBOLS INCDIR(common\src)
```

```
    ${keilpath}\a51.exe startup.a51 NOMOD51 DEBUG XREF EP INCDIR(\common\src)
    ${keilpath}\a51.exe common\src\si4010_rom_keil.a51 NOMOD51 DEBUG XREF EP
INCDIR(\common\src)
    ${keilpath}\a51.exe common\src\si4010_rom_all.a51 NOMOD51 DEBUG XREF EP
INCDIR(\common\src)
```

```
    ${keilpath}\BL51.EXE startup.obj, isr_dmd.obj, SenseUnit.obj, i2c.obj,
common\src\si4010_rom_keil.obj, common\src\si4010_link.obj,
common\lib\si4010_api_add_keil.lib, common\src\si4010_data.obj TO finalprog.omf PL(68) PW(78)
IXREF RS (256) CODE (0x0-0X0EFF) XDATA (0X0F00-0X107F) STACK (?STACK (0x90))
```

```
    ${keilpath}\OH51.EXE finalprog.omf
```

clean:

```
    del *.M51
    del *.OMF
    del *.LST
    del *.OBJ
```

upload:

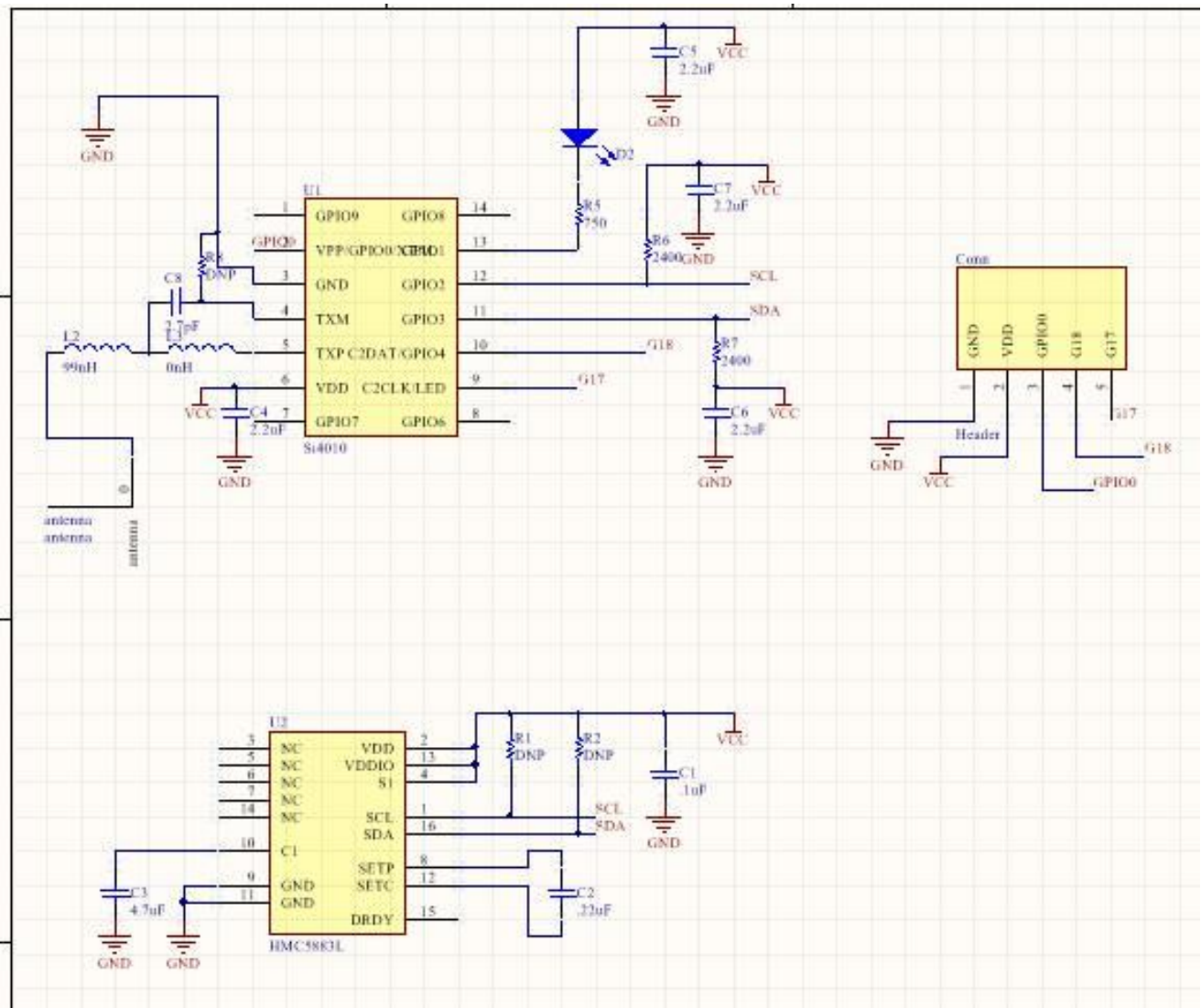
```
    -winscp /script=winscpupload.txt
```

uploadrun: **main upload clean**

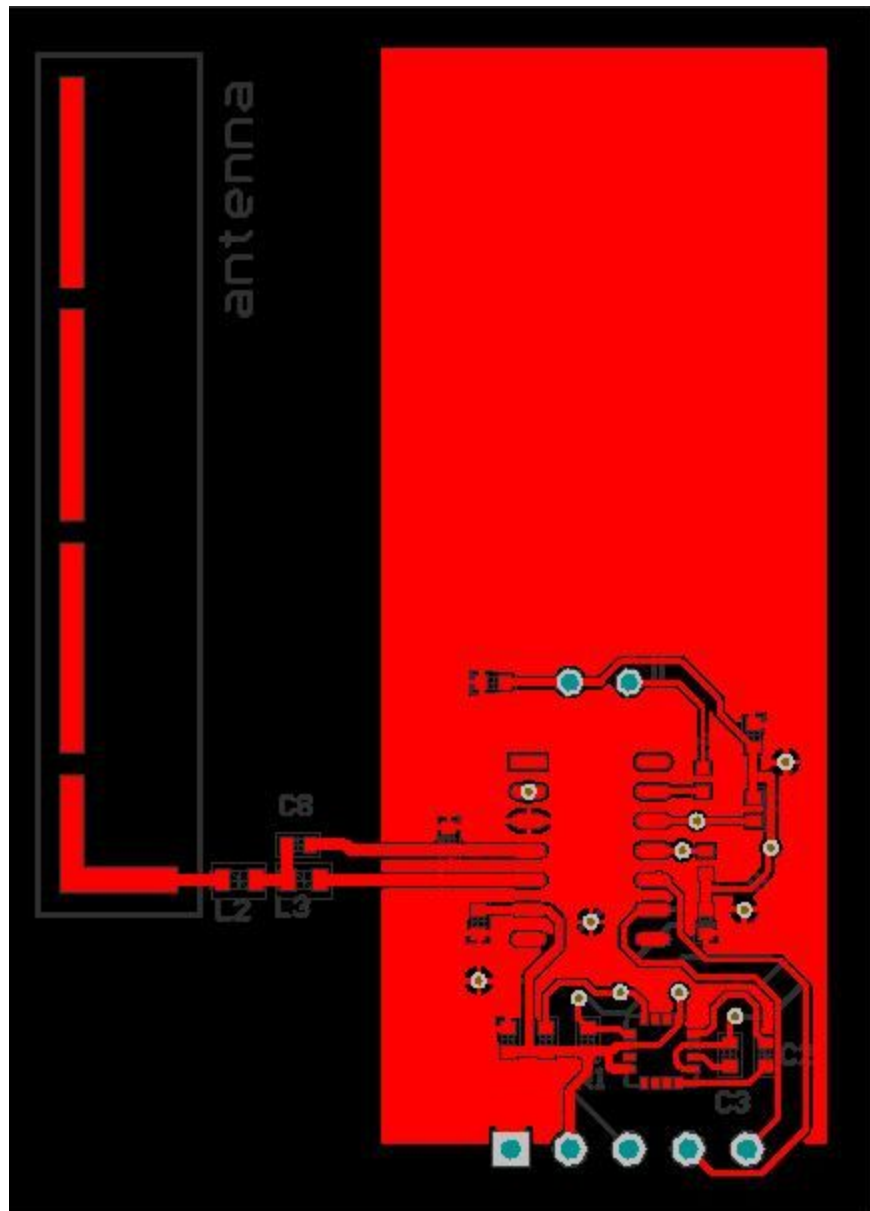
```
winscpupload.txt
```

```
open sftp://pi:park!andgo@192.168.2.3
put finalprog.hex /home/pi/si4010/
call cd si4010
call ./progit.sh
exit
```

Schematic of PCB Layout



PCB Layout



References

- [1]A. Sivakumar and A. Sivakumar, "#HowTo Detect vehicle presence or movements with magnetometers", *Blog.nxp.com*, 2015. [Online]. Available: <http://blog.nxp.com/connected-car/howto-detect-vehicle-presence-or-movements-with-magnetometers/>. [Accessed: 13- Jun- 2016].
- [2]"I2C implementation on 8051 microcontroller", *8051projects.net*, 2016. [Online]. Available: http://www.8051projects.net/wiki/I2C_Implementation_on_8051. [Accessed: 13- Jun- 2016].
- [3]*Si4010 -- Crystal-less SOC RF Transmitter*, 1st ed. Silicon Labs, 2016.
- [4]*Si4010 Software Programming Guide -- AN370*, 1st ed. Silicon Labs, 2016.
- [5]D. Imhoff, "dimhoff/si4010prog", *GitHub*, 2016. [Online]. Available: <https://github.com/dimhoff/si4010prog>. [Accessed: 13- Jun- 2016].
- [6]"SparkFun Triple Axis Magnetometer Breakout - HMC5883L - SEN-10530 - SparkFun Electronics", *Sparkfun.com*, 2016. [Online]. Available: <https://www.sparkfun.com/products/10530>. [Accessed: 13- Jun- 2016].