Magnetometer Parking Sensor

EGGN 383 Final Project

December 12, 2013

Roy Stillwell

Andrew Wilson

Introduction:

The objective of our project was to develop a wireless sensor device to be used for detecting motor vehicles, along with their direction of travel. The sensor is to be used in a larger system to monitor parking lot traffic by counting ins and outs of vehicles, thus showing which lots are full or not via a website and mobile application. The sensor is a magnetometer and works by detecting the Earth's magnetic field. As a vehicle drives over the device, a disturbance in the Earth's field can be detected by the sensor in the x,y, and z planes. This data is analyzed by an Arduino Fio microcontroller, and if a vehicle is detected, a local car count is sent to a Raspberry Pi base-station. In further development of the project by the ACMx (Association for Computing Machinery) club on campus, the Raspberry Pi would analyze the data from multiple sensors, and update a web page that can be viewed on a mobile device or a web page as shown in Figure 1.

Figure 1. Mobile Application mockup showing the end design goal.

The design criteria was met using a low power microcontroller (Arduino Fio), a sensor (HMC5883L), a prototyping Otterbox waterproof case, an Xbee Pro wireless module, and a large capacity battery (6000mAh LiPo). The software was implemented in multiple steps. First, an I2C interface was initialized on the microcontroller and the magnetometer sensor was placed in the proper mode. Sensor data was then acquired at regular polling intervals. Once a threshold was reached, this data was analyzed and then sent to a base station, currently a Raspberry Pi with a receiving Xbee Pro wireless module. The following is detailed description of how the components work together to sense both the direction and count of vehicles traveling over the sensor.

Hardware Overview:

The hardware implementation for the parking sensor system is shown in the block diagram below in Figure 2. Considering the limited time constraints and feasibility of developing the system end to end, we decided to limit the scope of our project to the Sensor device itself, as well as a simple script on a Raspberry Pi (which we call the base station) to show the data being sent by the sensor.

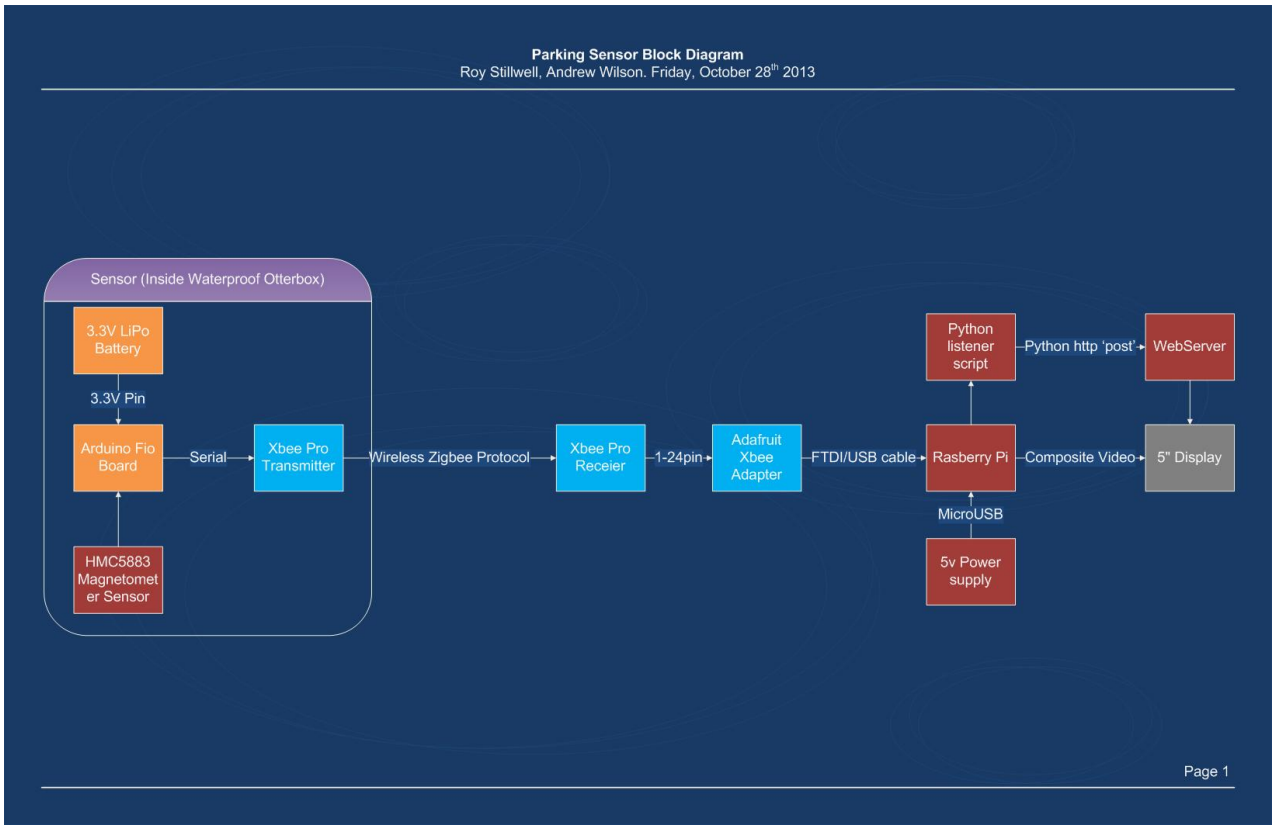

Figure 2. Sensor Block Diagram.

The diagram in Figure 2 shows the hardware for the sensor, as well as a 'base station,' as we named it, that is ran by a Raspberry Pi. The Sensor and Raspberry Pi is linked wirelessly by two identical Xbee Pro devices, with a reported range of 1-mile. An LCD 5" display is connected to the Raspberry Pi via a simple composite cable to display data received from the sensor. The Raspberry Pi runs a flavor of linux named 'Raspbian.' Currently, the Raspberry Pi also runs an Apache Web server, though the workings of this is outside the scope of our project.

Sensor:

The parking sensor consists of :

- An Arduino Fio
- 6000 mAh Lithium Ion Polymer (LiPo) battery

- An HMC5883L Magnetometer sensor
- An Xbee Pro (Series 1)
- Waterproof and Crushproof (as we found out) Otterbox (Small).

The sensor device hardware is pictured below in Figure 3 (without the Otterbox, shown later).

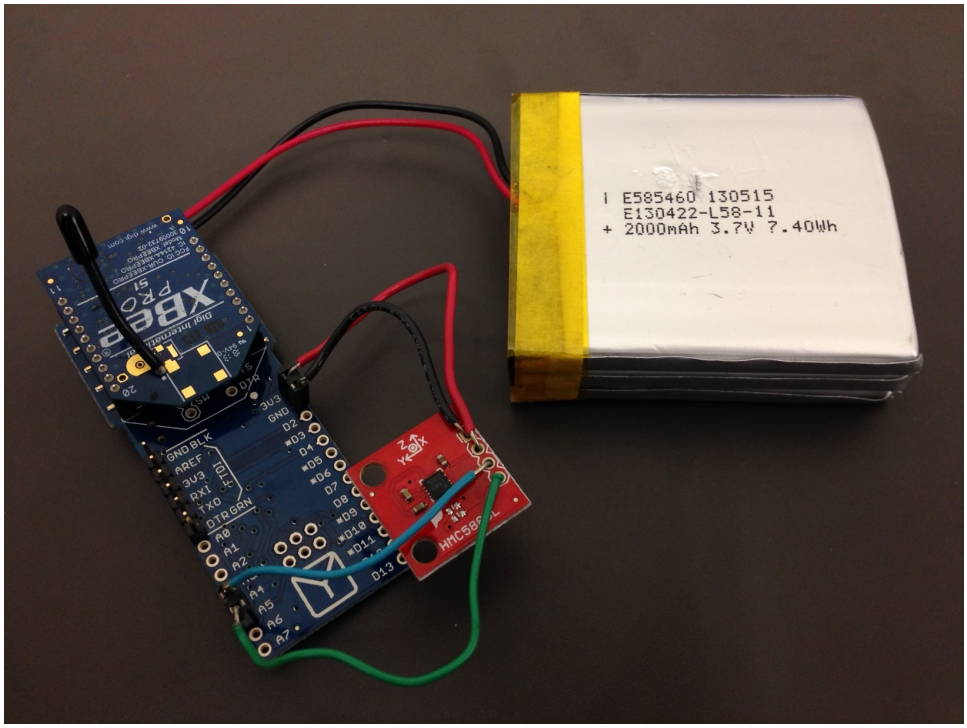

Figure 3. Pictured: Xbee Pro, Arduino Fio, HMC5883L (red), 3.3V LiPo battery (From left to right).

The parking sensor is based around the Honeywell HMC5883L device shown in red above. The HMC5883L is a “surface-mount, multi-chip module designed for low-field magnetic sensing with a digital interface” [1]. It uses magneto-resistive sensors to detect the magnetic field and a built in 12-bit ADC to digitize the signal. The device also has a built in I^2C serial bus for easy interfacing. We used the I^2C pins on the Arduino Fio to communicate with the device (detailed later). The device is usually used to find magnetic North to operate as a compass or to be used for magnetometry. However, we decided to try and use it in reverse. Rather than detecting a strong magnetic north pole, we used the device to detect a changing magnitude of the Earth’s field caused by the metal in a large vehicle. Data collection was done to see if a) we could even detect a vehicle and b) if we could detect it’s direction. The results are explained in the ‘Verification’ section.

We chose the Arduino Fio as the microcontroller for the project due to a) its small form factor, b) its lower power consumption c) its native use of 3.3V logic (vs 5V) and, d) because the device has built in headers for an Xbee Pro wireless device for wireless communication. The entire

package fits into a small waterproof, crushproof Otterbox case as shown in Figure 4.

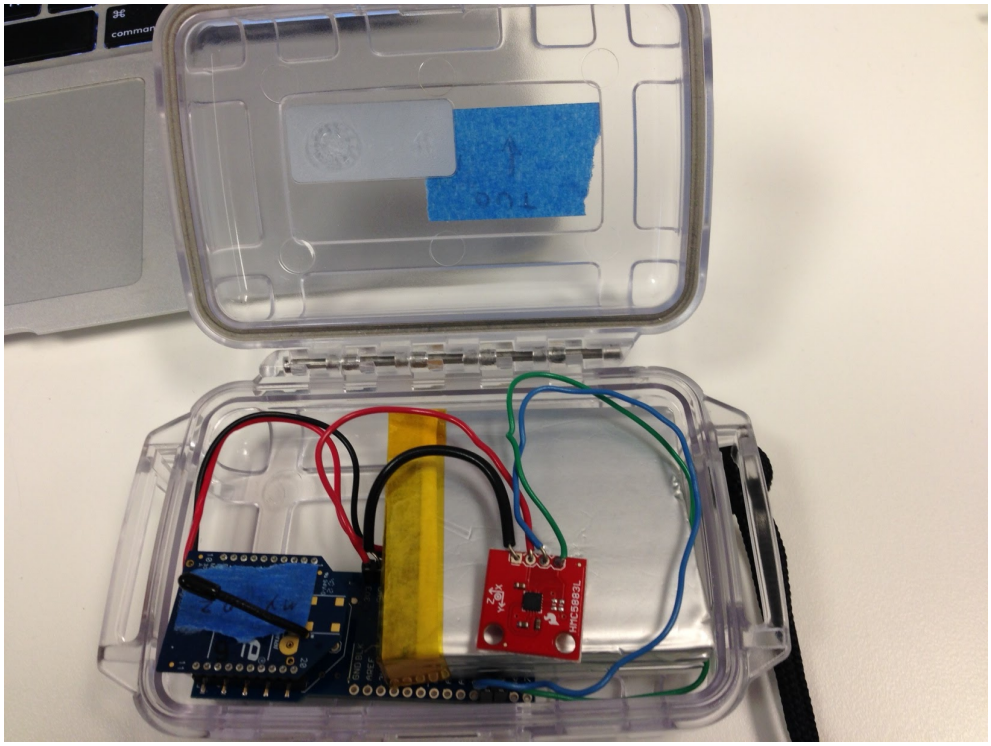

Figure 4. Otterbox enclosure for sensor hardware.

We found a distinct advantage in using this hardware. Because the Arduino Fio was connected to the Xbee, essentially emulating a long range serial connection, we were able to remotely program the device whenever we wanted to test new code. It was a welcome addition when testing the device in -4 degree (F) weather!

Xbee Pro:

The Xbee Pro is a wireless communication device built on top of the Zigbee/802.15.4 communication protocol developed by Digi Corporation [2]. The details of the 802.15.4 protocol sits well outside the scope of this report, but a curious reader can read the specification in [2]. At the most basic level, the Xbee Pro's are used to transmit and receive data sent out the tx/rx pins of the Arduino Fio to the receiving base station. Remarkably, the Fio still uses the simple RS232 serial protocol. The Xbee fulfills the role of a wire, essentially, by transmitting this serial data to a paired Xbee. An 'ID' is set for each Xbee device using a proprietary programming software developed by Digi. Each Xbee can be set to 'listen' to all IDs, or masked to specific IDs. The Xbee receiver on the base station is set to listen to all Xbee communication, whereas the Xbee transceiver on the sensor is set only to listen to the 'ID' of the base station XBee. Further information can be found on the excellent tutorial found on AdaFruit's learning site [3]. The sensor enclosure is shown in Figure 5 below.

Figure 5. Sensor enclosure.

The base station:

The base station consists of a Raspberry Pi device running a variant of Linux developed specifically for the Raspberry Pi named ‘Raspbian.’ Programs are written natively in the Python scripting language, and the Linux environment has various libraries installed to help with communication. One such library is the Python.serial library, which for our demo, was used to read data from the serial port as transmitted by the Arduino Fio.

An Xbee Pro is connected via an FTDI cable. The FTDI FT232RL is a usb/serial chip embedded in a cable that has a 6-pin socket at the end. The cable emulates an RS232 serial port via USB. The USB cable is connected to an Xbee Adapter Kit upon which the Xbee Pro module is connected [4]. The pinout connections are described in the ‘Circuit Description’ section. The physical setup is shown below in Figure 6.

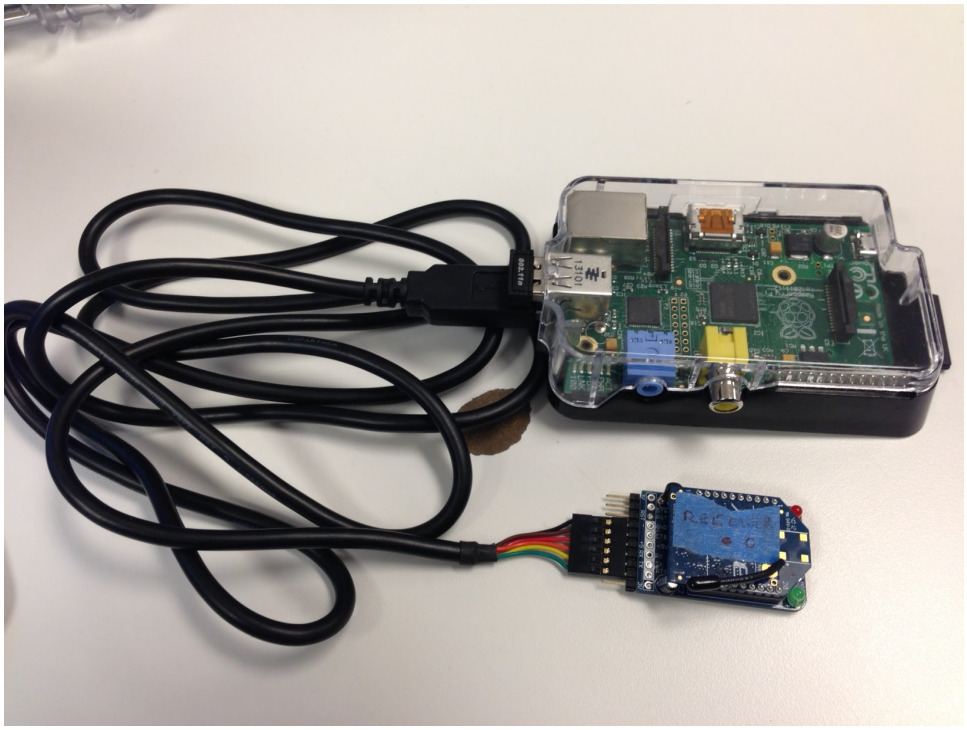

Figure 6. Raspberry Pi connected to an Xbee Pro via an FTDI cable and Xbee Adapter Kit.

Circuit Description:

The Circuitry of this project consists of two separate parts which are connected wirelessly. The two parts are the sensor circuit and the base station circuit. The sensor circuit receives and evaluates the magnetometer data. The base station circuit receives the evaluated data from the sensor circuit, compiles the data, and will eventually send it to a web page.

The sensor circuit is controlled by an Arduino Fio microcontroller which is powered by a 6000mAh LiPo battery connected at the input and ground terminals. Connected to the Arduino is a magnetometer sensor. This sensor is connected to the Arduino by four different pins, the power and ground pins, as well as a clock pin and the data pin. The clock pin(SCL) is connected to pin A5 on the microcontroller, and the data pin(SDA) is connected to A4 on the microcontroller. The sensor uses the SCL and SDA pins to send data to the Arduino Fio using I^2C protocol.

Also connected to the Arduino Fio is an XBee Pro. The XBee Pro is used to wirelessly send the data to the base station or to write to the arduino. It is connected to the Arduino by four pins, the power and ground pins, as well as the receiving and transmitting pins. The receiving pin(Rx) of the XBee is connected to the transmitting pin(TX0) of the microcontroller, and the transmitting pin(Tx) of the XBee is connected to the receiving pin (RX1) of the microcontroller. These pins are used for serial communication between these devices. The schematic for the Arduino Fio is below in Figure 7, as well as the schematic for how it interfaces to the other devices in Figure 8.

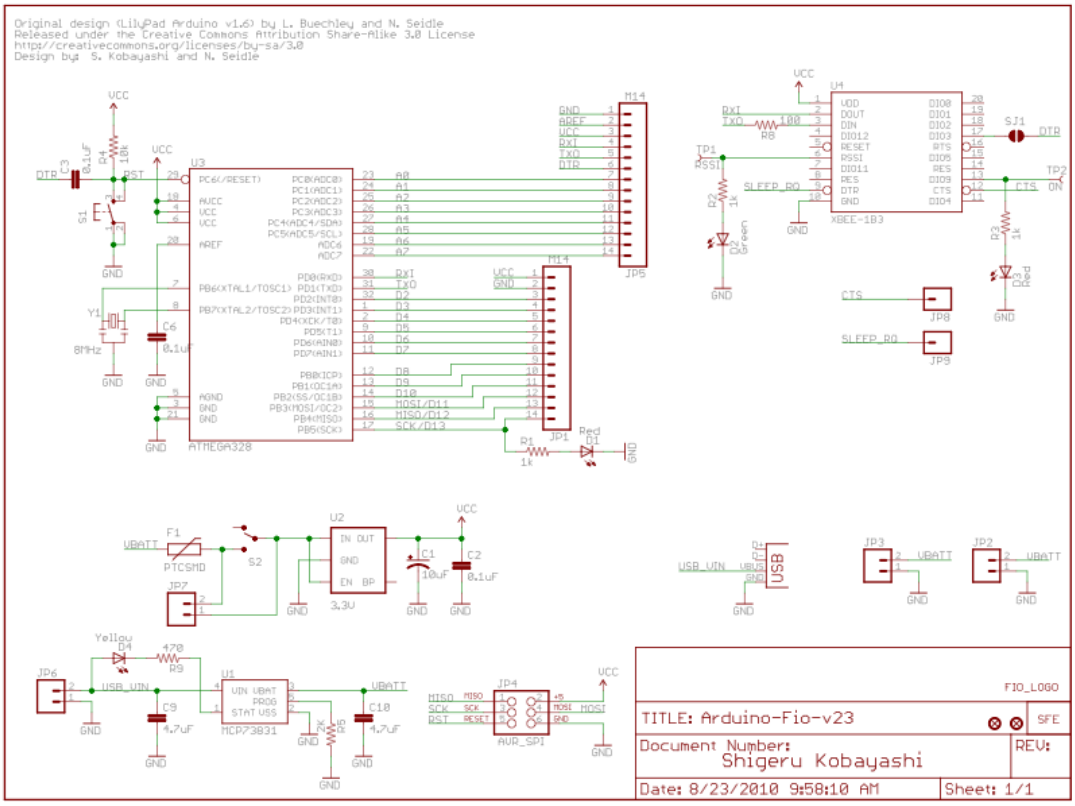

Figure 7: Schematic of the Arduino Fio.

Magnetometer Sensor Diagram

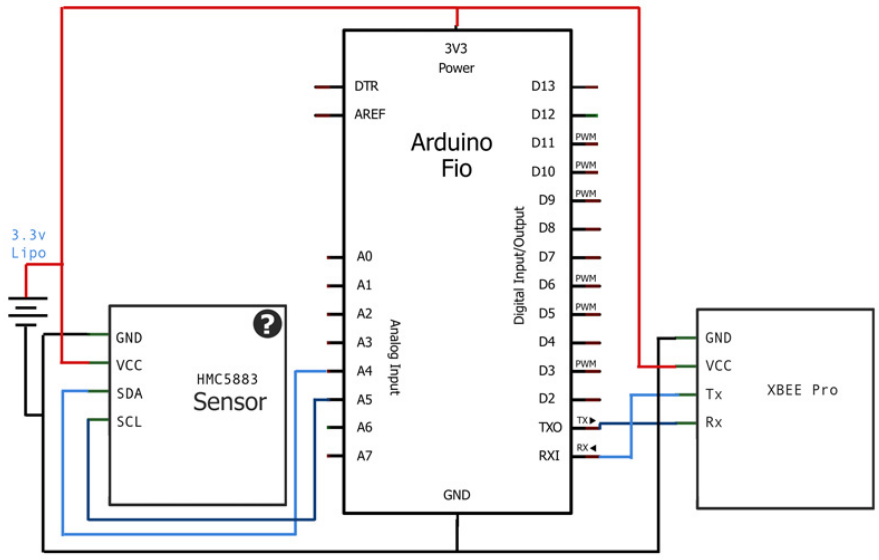

Figure 8: Sensor circuit diagram

The base station circuit has 2 main devices connected by a FTDI to USB cable. The XBee receiver is connected to an XBee adapter specifically made so that is can use an FTDI to USB cable to connect to a computer or in this case the Raspberry Pi. The FTDI cable connects to six pins on the XBee adapter. We only use four of them: ground, 5 V power, RX, and TX. The RX and TX use RS232 serial communication to communicate to the Pi. Below in Figure 9 is the schematic for the base station circuit. Pins CTS and RTS are unused but still connected.

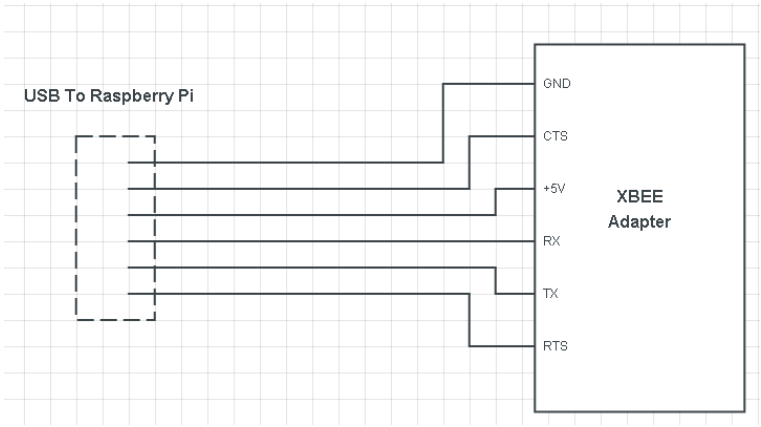

Figure 9: Base station circuit

Software Design:

There are two different sets of code being used in this project. The code for the Arduino Fio is used to evaluate the data from the magnetometer and detect vehicles. The code for the Raspberry Pi is for us to view the number of vehicles and output it to a web page. Below is a description of the code for each system as well as the corresponding pseudocode.

The task of the code used by the Arduino Fio board is to detect vehicles passing over the sensor and then output the detection to the Raspberry Pi via an XBee transmitter. The code must first initialize all the systems and variables such as the serial and I^2C communications. Then it immediately calculates a baseline value for the magnetic field on the y axis at that time by averaging values over a few seconds.

After this the code enters a loop which will poll the sensor data on the y axis and compare it to the stored baseline value. Every time the y value differs by more than a predetermined threshold it gets stored in an array named window and a count gets incremented. Every time the y value doesn't differ by more than the threshold, the count gets decremented towards a minimum value of zero. Once that count reaches a large enough value it decides that a car has passed by. It will then take the average of the first three values of the window and compare it to the baseline. If this value is less than the baseline, it means a car is entering the lot. Conversely, if the value is greater than the baseline, it means a car has exited the lot. It will then change the local car count and send it away to the base station via Xbee. Occasionally, if the system hasn't detected a car recently, the arduino will automatically send the local car count to the base station. The exact values for the variables are not final and the code can be seen in

the appendix. The pseudocode for the Arduino is below in Figure 10.

```
Initialize:
        I2C Data input:
                y axis
        Serial communications
        Variables:
                baselineSize
                windowSize
                threshold
                window[windowSize]
                detectorCount = 0
                carCount

for(100 values){
        Receive data from sensor
        Store y axis data in array
}
average = Average of all values in array

while(1){
        if(the y axis value differs from average by more than threshold){
                if (the detectorCount < windowSize) {
                        store y value in window
                }
                detector count ++ //we may have detected a car
                ef(detectorCount > windowsize){  //a car has been detected
                        if((windowaverage – average)<0){
                                carCount++
                        }
                        else{
                                carCount--
                        }
                        data= carCount, device id, batterylife
                        Transmit data to serial port
                }
        }else{
                If(detector count > 0){detector count --}
        }
}
```

Figure 10. Arduino Fio pseudocode.

The code running on the Raspberry Pi at this stage in the overall project is fairly simple. All it has to do is initialize the serial port and then display data received via the serial port. Eventually the Pi will send the total car count to a web page will will be viewable by students and

staff. The pseudo code for displaying the data to an LCD screen is shown in Figure 11.

```
Initialize serial port
Initialize LCD
while(1){
        if(serial port data is ten bits long){
                Echo data to screen
        }
}
```

Figure 11. Raspberry Pi Pseudocode.

Verification:

We had two major tasks to complete in order to use the magnetometer as a sensor. The first was to collect data to discern whether or not the sensor would even work for our application. The second was to analyze the data to discern whether we a) could detect a vehicle and b) detect the direction of a vehicle. To our delight, it turned out that both were possible with a single sensor! Data was collected from the x, y, and z axis as we drove vehicles over the sensor in different directions. Figure 12 shows the results of a typical data collection.

As a vehicle drives over the sensor, (or next to it, it's rather sensitive) the magnitude of the Earth's magnetic field (typically measured in Gauss) as detected by the sensor, changes dramatically. The blue inscription in Figure 12 shows that if a vehicle moves in the +x direction, there is a large positive spike, followed by a large negative spike, after which the sensor returns to what we call its 'baseline' values. Should a vehicle cross over the sensor from the -x direction, the opposite is observed (as shown in Figure 12 by the orange inscription); the pulse becomes largely negative initially, then pulses positively, finally to return to the baseline values.

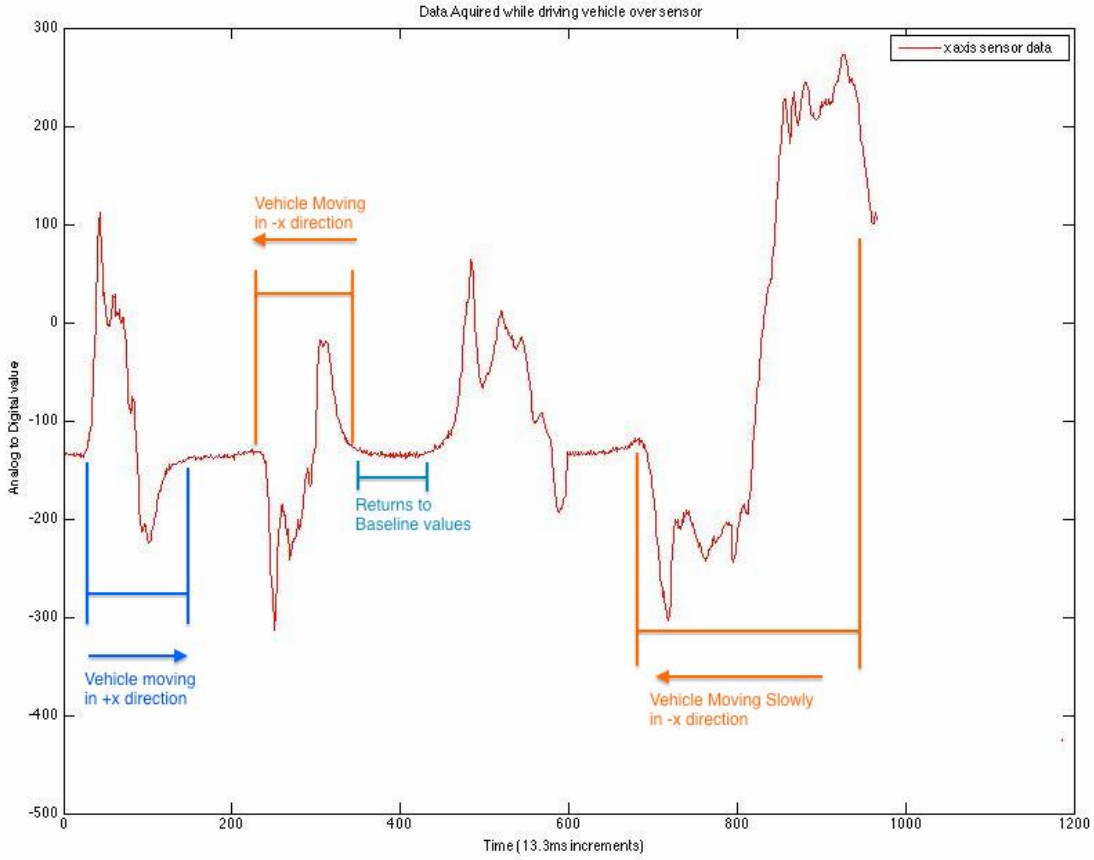

Figure 12. HMC5883L Magnetometer raw data showing vehicle direction.

Regardless of how long or how quickly a vehicle moves over the sensor, this basic pattern is observed. Quing on this, and as mentioned earlier, code was developed to detect a vehicle and its direction. To verify our design worked, the code was uploaded many times to the microcontroller. After which, we would drive a vehicle in multiple directions over the sensor. The results were successful. The Arduino Fio would echo via RS232 serial, “Car heading in” or “Car heading out” along with its local car count successfully. We also found we could replicate the demonstration by moving a device containing metal (we used a cell phone), back and forth inches over the sensor to which it would correctly display a detection and the direction.

Conclusion:

Given our scope of the project, we achieved everything that we intended to. We wanted to setup the hardware and create the detection algorithm for this project. The hardware setup we implemented works very well for the entire project and will continue to be used as the project is completed. The detection algorithm works successfully as well. We would also like to do long duration data measurements (weeks of data) for more fine tuning of the sensor algorithm.

We ran into a few problems while designing the detection algorithm. Detecting a vehicle

was simple to detect. Getting the system to calculate the correct direction, however was far more challenging. Our first problem was deciding which axis of the magnetic field to use. We weren't clear on how the magnetic field was quantified for each axis. Eventually we decided to use the y axis, which was the axis parallel to the direction the vehicles were going.

Using the 'window' method we developed to capture initial values, there were still problems with the window not detecting a vehicle passing over it. To fix this we had to tweak the values for the threshold and the window size so that each car would trigger it once even if it were going faster or slower. We had to continually change these values while we were changing other variables to keep it as accurate as possible.

The last problem we ran into was how to determine which direction the car was moving. We started out taking the average of the entire window because of a pattern we noticed in the graphs of the magnetic field. It appeared that the average magnetic field would spike more one way or another depending on which direction it was going. This method wasn't reliable enough and would detect the wrong direction about 25 percent of the time. The next plan was to take the first value of the window and compare it to the baseline. This was also not accurate enough because sometimes the first value would be skewed the wrong direction due to the noise. The last method we used was to take the average of the first three values. This would ensure that the values all come from the first spike in the graph while mitigating the problems due to noise.

We are very happy with the outcome of the project. The proof of concept using the magnetometer as a sensor, with wireless Xbee transmission definitely works. We both plan to continue to work on the project after the microcontroller course and see its completion. The next phase would be to continue improvement on the detection algorithm, and then begin work on the web page connections.

References:

[1] HMC5883L Spec sheet.
http://www51.honeywell.com/aero/common/documents/myaerospacecatalog-documents/Defense_Brochures-documents/HMC5883L_3-Axis_Digital_Compass_IC.pdf

[2] Xbee 802.15.4/Zigbee devices. Digi Corp.
http://www.digi.com/pdf/ds_xbeemultipointmodules.pdf

[3] Xbee programming. Adafruit Learning system.
http://www.ladyada.net/make/xbee/arduino.html

Appendix:

Arduino Fio Sensor Code:

```
/*
Arduino, HMC5883, magnetometer, XBee code
 by: Roy Stillwell, Andrew Wilson
 Colorado School of Mines
 created on: 10.30.13
 license: This work is licensed under a Creative Commons Attribution license.

 Arduino code example for interfacing with the HMC5883
 by: Jordan McConnell
 SparkFun Electronics
 created on: 6/30/11
 license: OSHW 1.0, http://freedomdefined.org/OSHW

 Analog input 4 I2C SDA
 Analog input 5 I2C SCL

 EEPROM code adapted from Tomorrow Lab
 Developer: Ted Ullricis_measuringh <ted@tomorrow-lab.com>
 http://tomorrow-lab.com


 ******************
 HOW IT WORKS:
 ******************

 An array of 'baseline' or nominal values -essentially when the sensor does NOT have a vehicle over it- is gathered initially.
 This baseline can be 'sized' to allow for fine tuning using the 'baselineSize' variable in the 'Variables you can change' section.
 After the initial buffer is filled, its average is calculated.
 A sensor value is then read and compared to the average and a threshold.  If it is outside this threshold, a counter is started.
 Once The counter is larger than the window size ( a very good indication of a car), we have detected a vehicle!
 We then grab the last three values in the window and check to see if it is greater or smaller than the baseline average.
 This gives us a direction of the vehicle.
```

```
 The code <will be> designed to re-calibrate the 'baseline' every 10 minutes (basically in the
event a sensor is hit wiht a solar storm or something).

 Currently this only uses the Y axis data (as that may be all we need)
 Using datasets taken at the CTLM exits and entrances, it correctly calculates entrances and
exits.

 */

#include <Wire.h> //I2C Arduino Library, required for interface communication with HMC5883
Device
#include "stats.h"  //a custom library for doing Average and Standard deviation calculations.
#include <cmath>  //Standard cmath library
using namespace std;

//----------Variables you can change--------------------
double recalibrateTime = 600000; //time in seconds. 600000 = 10 min -- used to
auto-recalibrate sensor, if time is greater than 10 minutes, recalibrate
#define baselineSize 100  //Size of baseline to use for baseline values
#define windowSize 30  //Size of 'window' to use for previous values of the sensor to be
considered in calculating a positive detection.
#define detectorThreshold 30
#define windowConsidered 3
double carThreshold = 15.0; //Used to sort out car 'hits'. Anything above or below this
'threshold' is counted as a hit
double pingTime = 60; //(in seconds) Used to make sure sensor is still alive after specified
time
//------------------------------------------------------

double pingCounter;   //Used to count cycles so system can 'ping' basestation every 30
seconds
int localCarCount=0;
long startTime ;                  // start time for the algorithm watch
long elapsedTime ;                // elapsed time for the algorithm
bool didWeCalculateAveDevAlready, saidit = false;
double lastCalibrateTime, stdDev;
double baselineAvg = 0;
float calibrationPercentDone;
int x, y, z;
int windowTotal = 0, count = 0, detectorCount = 0;
int baseline[baselineSize];
int window[windowSize];
int windowx[windowSize];
int windowz[windowSize];

#define address 0x1E //0011110b, I2C 7bit address of HMC5883
```

```
//Data packet to send
//Car count
//Arduino Id
//Battery life
//Separate by space
//end in semi colon



void setup(){
  //Initialize Serial speed
  Serial.begin(57600);


  configureSensor();

}

void configureSensor() {
  Wire.begin(); //Initalize I2C interface in Arduino

  //Put the HMC5883 IC into the correct operating mode
  Wire.beginTransmission(address); //open communication with HMC5883
  Wire.write(0x02); //select mode register
  Wire.write(0x00); //continuous measurement mode
  Wire.endTransmission();

  //Crank the speed up to 75Hz read speeds (default is 15Hz)
  Wire.beginTransmission(address);
  Wire.write((byte) 0x00);
  Wire.write((byte) 0x18); //this jumps it to 75Hz
  Wire.endTransmission();
  delay(5);

  //Tell the HMC5883 where to begin reading data
  Wire.beginTransmission(address);
  Wire.write(0x03); //select register 3, X MSB register
  Wire.endTransmission();

  //Read data from each axis, 2 registers per axis (helps initialize readings)
  Wire.requestFrom(address, 6);
  if(6<=Wire.available()){
          x = Wire.read()<<8; //X msb
          x |= Wire.read(); //X lsb
          z = Wire.read()<<8; //Z msb
          z |= Wire.read(); //Z lsb
          y = Wire.read()<<8; //Y msb
```

```
        y |= Wire.read(); //Y lsb
  }
  //Pause
  delay(1000);

}

//The main loop to repeat indefinitely
void loop(){

  //Tell the HMC5883 where to begin reading data
  Wire.beginTransmission(address);
  Wire.write(0x03); //select register 3, X MSB register
  Wire.endTransmission();


  //Read data from each axis, 2 registers per axis
  Wire.requestFrom(address, 6);
  if(6<=Wire.available()){
        x = Wire.read()<<8; //X msb
        x |= Wire.read(); //X lsb
        z = Wire.read()<<8; //Z msb
        z |= Wire.read(); //Z lsb
        y = Wire.read()<<8; //Y msb
        y |= Wire.read(); //Y lsb
  }

  //build the baseline array to be used for average calculation
  if (count < baselineSize) {
        baseline[count] = y;

        //Echo out to serial how far we are in calibrating.
        double calibrationPercentDone =  baselineSize;
        calibrationPercentDone = count / calibrationPercentDone * 100;
        Serial.print("Hang on, Calibrating ");
        Serial.print(calibrationPercentDone);
        Serial.print("% done. ");
        Serial.print(x);
        Serial.print(" ");
        Serial.print(y);
        Serial.print(" ");
        Serial.println(z);

        delay(40);  //Pause for a bunch of cycles so the values are collected over a few
seconds.
        count++; //increment counter to fill buffer of size 'baselineSize'
  }
```

```
  //Figure out baseline values
  if (count == baselineSize && didWeCalculateAveDevAlready == false) {  //The baseline array
is full, so we can begin collecting data!
          baselineAvg = Average(baseline); //get average
          didWeCalculateAveDevAlready = true;
          startTime = millis();
  }

  //  elapsedTime = millis() - startTime;
  //  //TODO: write recalibration code
  //  if (elapsedTime > recalibrateTime) {
  //        //look to recalibrate by building a new temp array
  //
  //        //check standard deviation of new array against current standard deviation
  //        //if within limits, use the new calibration data
  //        Serial.println("Uh hang on a minute, recalibrating.");
  //        startTime = millis();
  //        count = 0;
  //  }


  bool something = false;  //used to flag whether or not something is over the sensor

  //This algorithm calculates whether 'something' was found, and then to be sure, starts a
detectorCount to verify
  //there are enough hits to constitute an actual car passing, and not some random flux in the
Earth's field.
  if ( didWeCalculateAveDevAlready == true && (y >= (baselineAvg + carThreshold) || y <=
(baselineAvg - carThreshold))) {

          something = true;  //something is probably out there!

          delay(14); //The HMC is set to run at 75Hz, this delays just that mount!

          //When 'something' may be out there, there are three scenarios.
          //1) The detector count goes up if there are less 'something' hits than the window size,
          //2) The detector count hits the windows size -what we qualify as a true car over the
sensor!- and as such reports it,
          //3) There wasn't a new event, and the detector count is decremented.  Once it hits
zero, the car is thought to have passed the sensor
          // Effectively, while a car is over the sensor, detectorCount stops counting, and sort of
cruises at the max window size.  As the car passes
          //and sensor values drop to the nominal baseline, it starts clearing the detectorCount
buffer and readies for a new car.

          // 1) Something may be in the works, we need to count the 'something' events to be
```

```
sure and store it in the window
        // We will use the detectorCount as the indexer
        if (detectorCount <= windowSize) {
        window[detectorCount] = y;  //We add the current value to the window, for analysis
later
        windowx[detectorCount] = x;
        windowz[detectorCount] = z;
 }
        detectorCount++;

        if (detectorCount >= windowSize && saidit==false) { //here we check to see if we have
detected enough events
        //We did detect a car! Now to see direction
        double windowAve;

        for (int i=0; i < windowConsidered; i++) { //using the first few values of the window (the
orignal direction of the Magfield)
                                        //we find the direction.
        windowAve += window[i]; //add up all the values
        }
        windowAve = windowAve / windowConsidered;  //generate the average value to be
used

        //If the first value is less than the average, we have a car heading in!
        if (windowAve < baselineAvg  ) {
        localCarCount++;
        Serial.print("Car heading in! localCarCount:");
        Serial.print(localCarCount);
        Serial.print ("Data: ");
        delay(500);
        pingCounter = 0; // Device has communicated with the base station, so reset counter
for communication
        }
        //Otherwise the car was heading out!
        else {
        localCarCount--;
        Serial.print(" car heading out: localCarCount:");
        Serial.print(localCarCount);
        Serial.print ("Data: ");
        delay(500);
        pingCounter = 0; // Device has communicated with the base station, so reset counter
for communication
        }
        //For debugging, we echo the window average and baseline average to makes sure the
algorithm is working
        Serial.print("windowAve: ");
        Serial.print(windowAve);
```

```
        Serial.print(" baselineAvg: ");
        Serial.println (baselineAvg);
        saidit = true; //mark that we have said there is a car already, so we don't repeat it
        }

  }

  else {
        //3) An event was not detected, so the detectorCount begins decrementing its values
        if (detectorCount > 0) detectorCount--;
        //Once it zero, the algorithm is essentially ready for a new car.
        if (detectorCount == 0) saidit = false;

        //This code does is used for diagnostics.  It essentially sends a 'heartbeat' to the base
station
        if (pingCounter >= (pingTime*860)) { //860 is about 1 second with this program code
        //To be sure everything is working (mostly for diagnostics, ping every 30 seconds).
        Serial.print("All Still Quiet. localCarCount:");
        Serial.println(localCarCount);
        pingCounter = 0;
        //delay(40);
        }
        pingCounter++;
  }

}
```

Averaging Function:

```
//Returns average value from array
double Average(int window[]) {
   double average = 0;
   for(int i = 0; i < sizeof(window); i++) {
         average += window[i];
   }
   average = average / sizeof(window);
   return average;

}
```

Python script on Raspberry Pi to display data sent via Serial from Arduino Fio sensors:

```
#!/usr/bin/env python
```

```
#dummy dictonaries, replace these with actual data on deploy

import serial, time, datetime, sys, dataChecker, update
#from xbee import XBee

# use App Engine? or log file? comment out next line if appengine
LOGFILENAME = "Sensordatalog.csv"   # where we will store our flatfile data

# open up the FTDI serial port to get data transmitted to xbee
print "opening serial port"
serial_port = serial.Serial("/dev/ttyUSB0", 57600)
print "opened serial port"
#xbee = XBee(serial_port)
def readFromSerial(serial_port):
    while True:
        try:
            readData = serial_port.next()
            print readData

        except KeyboardInterrupt:
            break
    serial_port.close()


print "about to read"
readFromSerial(serial_port)
```