OpenFusion

sponsored by Lawrence Livermore National Laboratory

Michael Baptist <mbaptist@ucsc.edu>
Michael Bennett <mijbenne@ucsc.edu>
Bardia Keyoumarsi <bkeyouma@ucsc.edu>
Vincent Lantaca <vlantaca@ucsc.edu>
David Tucker <dmtucker@ucsc.edu>

Contents

1. Abstract

1.1 Motivation

The Lawrence Livermore National Laboratory's Data Science Initiative was established to advance state-of-the-art technology in the "big data" domain including data collection methodology, storage, visualization, analytics, modeling, simulation, and high performance computing capabilities. To support this initiative, the students and faculty of the School of Engineering (SOE) at University of California, Santa Cruz (UCSC) produced a real-time sensor collection system whose acquired data can be fused with publicly available "open" data (e.g. social media, climate data, traffic data, etc.).

1.2 Objectives

The goal of this project was to produce a web application that will merge gathered data with publicly available information using standard formats and open platforms. Another dedicated application (for iOS) would allow any iPhone to become an information collection source, and relationships among data can be determined using attached geotags. Related data may be visualized using a number of methods including graphing and geographic plotting. Implementation specifics will need to be clearly documented and cleanly presented.

In addition to the real-time sensor collection system, the GSF project team provided tools and frameworks for data query, analysis, and visualization of the fused data for utility and ease of adaptation to later potential applications.

2. Approach

2.1 Pluggable Sensor Hardware

The pluggable sensor refers to a hardware design that interfaces a microcontroller, communicating with an external sensor, to an iOS device's 3.5 mm audio jack. After discussing the options for how to incorporate external sensors to our mobile sensor suite, the headphone jack was chosen as the interface for communication to keep the design as one unit. The goal of the pluggable sensor was to provide cheap, small, and low power sensory input. Specifically, we wanted to power a microcontroller and a digital sensor from the headphone jack to show that a wide range of data could be possible inputs to our mobile sensor suite.

2.1.1 Power Design

Before any power design could be done, we needed to know how we would power a device that would be plugged into the headphone jack of the iPhone. The headphone jack had 4 contacts: Left, Right, Microphone, and Ground. One thing that we thought would be nice is if we could use the Left or Right channel to output a DC wave which we could use to power the circuit. So, we tried using Audacity to create a mp3 file which was essentially a DC tone. When we played this mp3 file on an iPhone, however, we weren't able to measure any voltage or current from the Left or Right contacts. We also tried to see if we could measure any voltage or current from an AC tone that was played by the iPhone. This was done by downloading and experimenting with various tone generator applications that were available on the App Store. We found that by playing a sine wave on the Left Channel, we could measure about 2.7Vpp by probing the Left Channel with respect to ground, with a short circuit current of about 100mA.

In order to power a microcontroller and a sensor, however, we needed DC at the correct voltage levels. To achieve this, we first stepped up the amplitude of the sine wave by using a transformer, then used a full-wave rectifier to rectify the wave. Finally, we filtered this wave to limit oscillations and passed it to a 3.3V regulator. The output of the regulator would then power a microcontroller and a digital sensor.

A MOSFET full-wave rectifier was used instead of the more common diode full-wave rectifier to reduce power loss. This was a key design consideration due to the fact that we had limited power to work with.

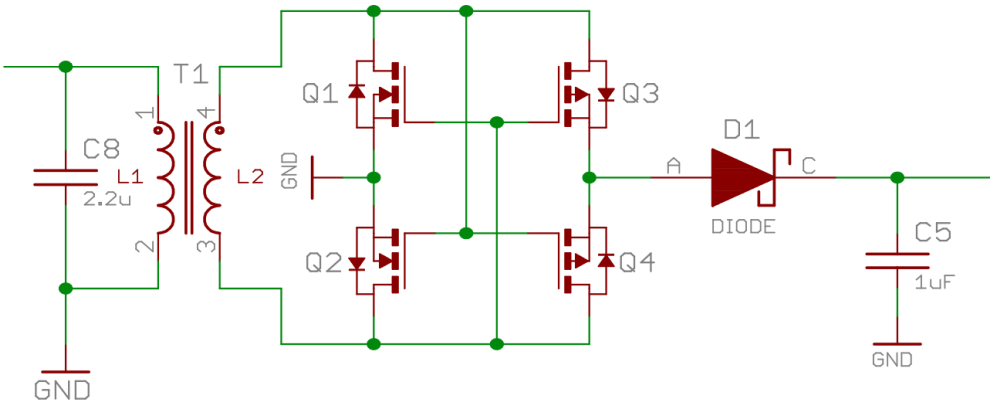


Figure 2.1.1i MOSFET Rectification Bridge

The configuration for the MOSFET rectification is shown above in **Figure 2.1.1i**. As seen in the figure, it uses two P-type MOSFETS (Q3 and Q4) and two N-type MOSFETS (Q1 and Q2).

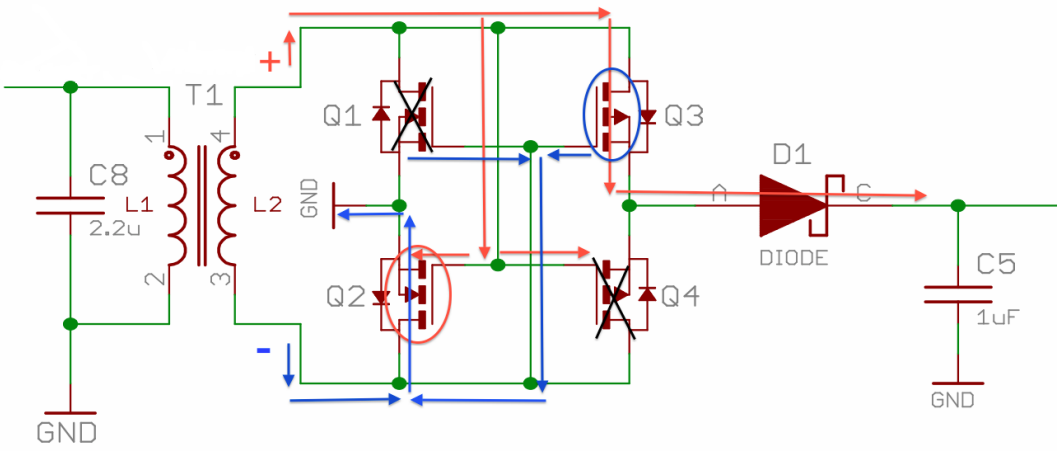


Figure 2.1.1ii Positive voltage at node 4 of the transformer

Figure 2.1.1ii shows that when the positive side of an AC wave comes out of node 4 of the transformer, a positive voltage is applied at the base of Q2 and Q4. Because Q4 is P-type and Q2 is N-type, only Q2 is activated. Simultaneously, a negative voltage is applied to Q3 and Q1, which only activates Q3. Thus, a positive voltage at node 4 of the transformer causes current to pass through diode D1.

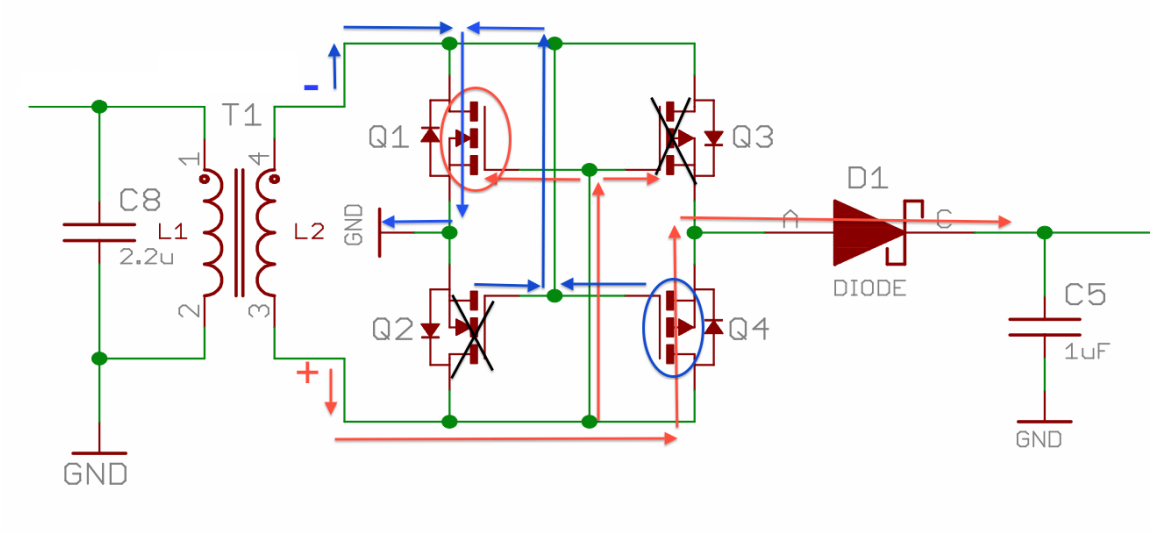


Figure 2.1.1iii Negative voltage at node 4 of the transformer

Similarly, **Figure 2.1.1iii** shows that when the output of node 4 with respect to node 3 of the transformer is negative, this also allows current to flow through D1. Thus, at any given time when a sine wave power tone is being played, the MOSFET rectification bridge allows for full-wave rectification with loss from the voltage drop across two MOSFETs and an additional diode for protecting the MOSFETs from backcharge. This is much better than having a voltage drop from two diodes in the typical diode full-wave rectification scheme. The final design was able to provided 3V at about 9mA to the pluggable sensor. It was difficult enough to provide 3V at 9mA to the pluggable sensor, so the choice of MOSFET rectification made it easier to provide the correct voltage levels to both the microcontroller and the sensor.

2.1.2 Recognition by the iPhone

Because we were dealing with audio, we wanted to understand user interaction that involved using the iPhone headset. By using the small remote on the iPhone headset, a user is able to pause, play, increase volume, and decrease volume. By double-tapping the pause/play button, the user is also able to bring up Siri. When music is played by the iPhone and the headset is plugged in, music automatically stops when the headset is detached. In order to be able to emulate some of these characteristics, we tried to figure out how the iPhone headset was signaling with the iPhone.

The easiest one to figure out was the ability to pause/play music. We tried probing with a multimeter to measure resistance of the Left, Right, and Microphone lines of the iPhone headset with respect to ground. When the Pause/Play button was pressed, the resistance measured on the Microphone line with respect to ground was 2Ω. We suspected that this was just the resistance of the wiring from the Pause/Play button to the male headphone jack on the iPhone. By opening the housing of the iPhone headset remote and examining the PCB inside, we found that only two wires were connected to the remote: Microphone and Ground. This told us that somehow, the iPhone headset was using the Microphone and Ground line to achieve Pause/Play, Volume Up, Volume Down, and to bring up Siri. After more research online, we have found previous teardowns and reverse-engineering of older iPhone headsets. It seemed that for at least the older iPhone headsets, the three buttons on the remote simply

shorted the Microphone line to Ground through varying resistors[5]. To see if this was the case for the newer headsets, we built the following circuit in **Figure 2.1.2i**. Note that at the far right of the schematic, an iPhone headset remote is attached. Because we didn't know how the iPhone headset was being recognized, we left it in the circuit so that our circuit was recognized as a headset. In fact, the circuit didn't work without the headset remote there. By doing this, we were able to isolate the experiments relating to the functionality of Pause/Play, Volume Up, and Volume Down.

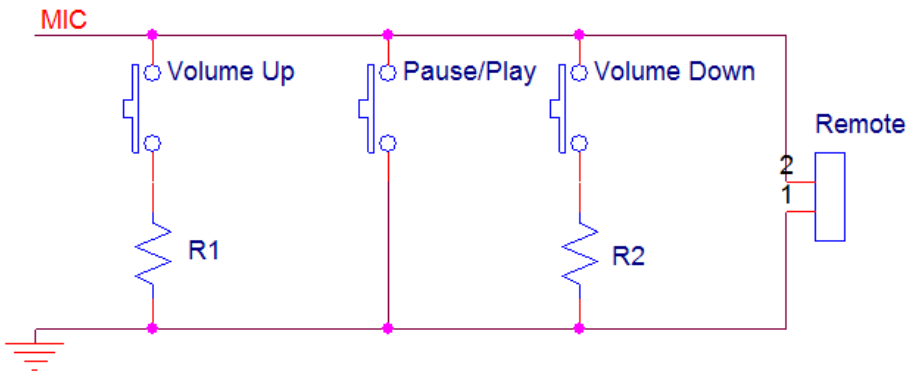


Figure 2.1.2i Headset Circuit

We measured the resistance of the Microphone line with respect to Ground when the Volume Up button was pressed, and did the same for Volume Down. We were able to Pause/Play the iPhone, but were unsuccessful in controlling Volume Up and Volume Down. Also, when we double-tapped the Pause/Play button, we were able to bring up Siri. Because we were able to control Volume Up and Volume Down in software, we did not pursue how to emulate these two controls in headware.

The last problem that needed to be solved how the iPhone was recognizing the chip. In previous reverse-engineering of the iPhone headset, it seemed that the headset was generating an ultrasonic chirp, which let the iPhone know that a headset was inserted. After trying to reproduce this chirp with a microcontroller, we were able to get our own circuit recognized by the iPhone without an iPhone headset attached. We eventually figured out that we could still control the circuit even if no power was supplied to the microcontroller. This lead us to figure out that the iPhone merely senses the input resistance looking into the headset remote. Our next step was to use a potentiometer to find the range of resistances that can trick the iPhone into thinking that a headset was attached, which we found to be between 150Ω and 10kΩ. After input resistances of above 10kΩ, the iPhone seems to inconsistently detect that there's a headset attached. This explanation for the iPhone recognizing the headset from input impedance, in our opinion, is much more reasonable than the idea of the headset generating an ultrasonic chirp, since the iPhone's codec can only reliably sample sonic frequencies. With this information, we are able to provide the user with alerts that tell them whether or not our own pluggable device is attached.

2.1.3 Temperature and Humidity Sensor

The temperature and humidity sensor chosen is a digital sensor from Amphenol Advanced

Sensors called the ChipCap2. There were many criteria that were taken into consideration when deciding which sensor to use. Because iOS devices operate from 0°C to 35°C, we wanted a sensor that could operate in this range. Another thing that had to be considered was that the sensor had to be low-power due to the limited power from the sine wave tone. Finally, we wanted to choose a sensor that was relatively cheap and small.

The ChipCap2 was chosen because it had a good combination of specifications that fit our application. The ChipCap2 costs $11.39, and measures humidity between 0 and 100% RH. It operates from -40°C to 125°C, operates between 2.7 and 5.5V, and has a maximum current draw of 1.1mA. The ChipCap2 provides temperature and humidity readings over I^2C or TWI.

2.1.4 Microcontroller

The choice of the microcontroller shared some of the restraints that applied to choosing the sensor: it needed to be low power, have the appropriate operating voltage level, and needed to be able to communicate with the I^2C sensor. The microcontroller that was chosen was the ATxmega128a4u because it met the mentioned specifications well, and had a low-cost development board. An added bonus was that the ATxmega128a4u uses an AVR CPU, which meant that it had a wide user base and support. The development board chosen is called the MT-DB-X4 and can be found from the following website: http://www.mattairtech.com/index.php/mt-db-x4.html. Because this development board isn't one of the devices supported by Atmel Studio, it could not be directly programmed from Atmel Studio. Instead, we created a hex file from a project from Atmel Studio and then loaded this hex file onto the microcontroller using Atmel FLIP. The same procedure used to program the development board is also used to program the microcontrollers on the Pluggable Sensor PCB once the bootloader is installed on the PCB's microcontroller. Installation of the bootloader will be described in a later section.

2.1.5 Pluggable Sensor Prototype

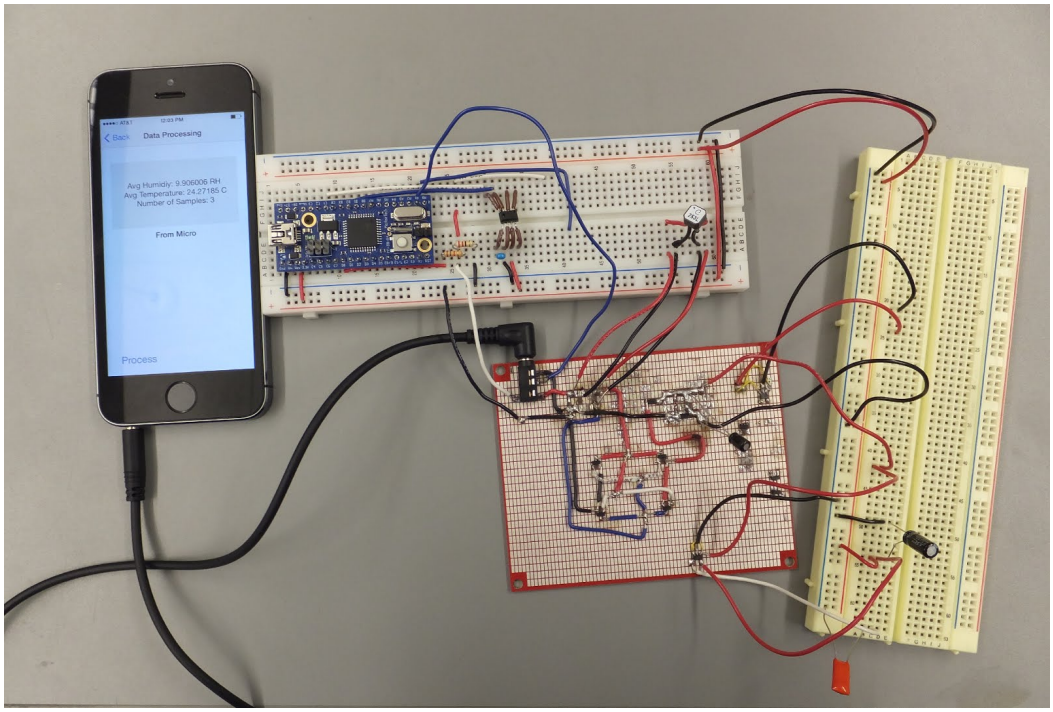


Figure 2.1.4i Pluggable Sensor Prototype

The pluggable sensor prototype is shown above in **Figure 2.14.i**. Shown on the red surf-board is the portion of our prototype that conditions the sine wave into DC. This included the step up transformer, the MOSFET rectification bridge, the filtering, and the regulator. The microcontroller development board and the sensor is on the breadboard above it. This style of prototyping was chosen instead of PCB prototyping, since it was more flexible to change.

The microcontroller uses its ADC listens on the Right channel from the iPhone for an enable. If it sees a 500mVpp sine wave, it will request data from the sensor, and then send this data to the iPhone using amplitude modulation and Manchester encoding.

2.1.6 Printed Circuit Boards

Once we had the prototype fully functional, the next step was to condense the design onto a small two-layer PCB board. The PCB schematics were designed using OrCAD Capture. Using OrCAD Capture, a netlist was created which was then imported in Allegro for layout. Next, Gerbtool was used for additional verification. Once verification was completed in Gerbtool, we sent the Gerber files to Alberta Printed Circuits to be fabricated. Alberta Printed Circuits was chosen because of the quick turn-around time, and the option for Basic prototype and a Plus prototype.

For the first PCB run, we chose a Basic prototype. The Basic prototype included only the top and bottom copper layers. The Basic prototype was chosen because they were cheaper. This way, if we had made any mistakes relating to footprint sizes, part placement, signal integrity, or pluggable sensor functionality, it would be less costly. This first run was called Revision 1 of our Pluggable Sensor PCB. It was not until Revision 1 was tested after the parts were soldered when we figured out that the microcontroller didn't have the bootloader installed. To load the bootloader, a AVRISP MKII was used. Wires were attached to the appropriate pins of the microcontroller and then the microcontroller was

programmed. After loading the bootloader, we were able to program the microcontroller with the microcontroller code that was developed and tested on the prototype. Despite some mistakes, Revision 1 of the PCB was able to accomplish its main goal which was to use it to send temperature and humidity data to the iPhone.

After testing Revision 1, we went on to Revision 2 of our Pluggable Sensor PCB. Revision 2 had a contact for attaching a PDI programmer, fixed some part placement issues, and was slightly smaller. This time, we chose a Plus Prototype from Alberta Printed Circuits which included the top and bottom copper layer, the top and bottom solder-mask layers, as well as the top and bottom silkscreen legend layers. Revision 2 of our Pluggable Sensor PCB worked as expected, and its schematics can be found attached at the end of this document. A photo of Revision 2 attached to an iPhone is shown below in **Figure 2.1.6i**.

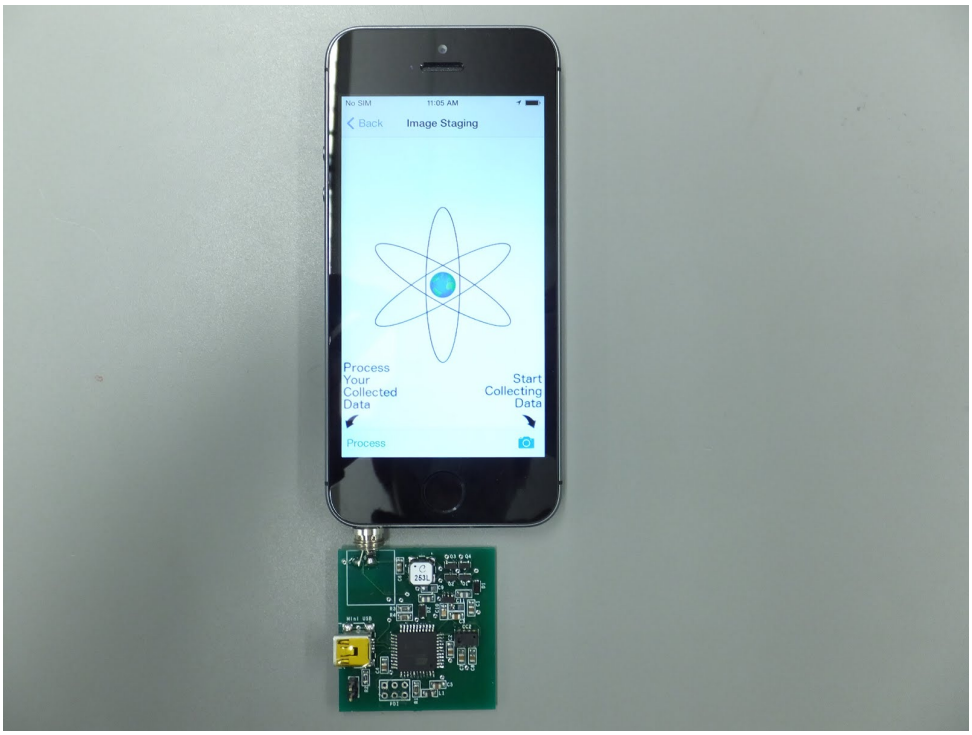


Figure 2.1.6i Revision 2 of Pluggable Sensor PCB

2.1.7 Microcontroller Bootloader

The bootloader used on the final PCB revision is the same bootloader that came on the MT-DB-X4 development board. It is "MT-DB-X4-128a4u_104.hex" which can be found on the development board's web page: http://www.mattairtech.com/index.php/mt-db-x4.html. Because the ATxmega128a4u microcontroller does not come with a USB bootloader, the USB port of the microcontroller can't be used to program the microcontroller until the bootloader is installed. To load the bootloader, we used an AVRISP MKII external programmer. The procedure to load the bootloader is as follows:

1. Connect the AVRISP MKII to the PDI pins of the Pluggable Sensor PCB. The microcontroller also needs external power to be programmed by the AVRISP MKII, so connect the USB power of the Pluggable Sensor PCB to your computer with a USB cable and then open Atmel Studio.

2. Open up the Device Programming window by selecting Tools->Device Programming. Under the "Tool" tab, select the AVRISP MKII, and select the ATxmega128a4u for "Device." For the "Interface" tab, select PDI. On the PCB, the 5V power line of the USB is fed as one of the inputs to the regulator so that the PCB gets power when programming via USB. Here, we are using it for external power. You can make sure that the microcontroller is getting power by selecting "Read" Target Voltage.

3. The next step is to do a Full Chip Erase, shown in **Figure 2.1.6i.** In the "Memories" section, select Erase Chip and then click "Erase now."

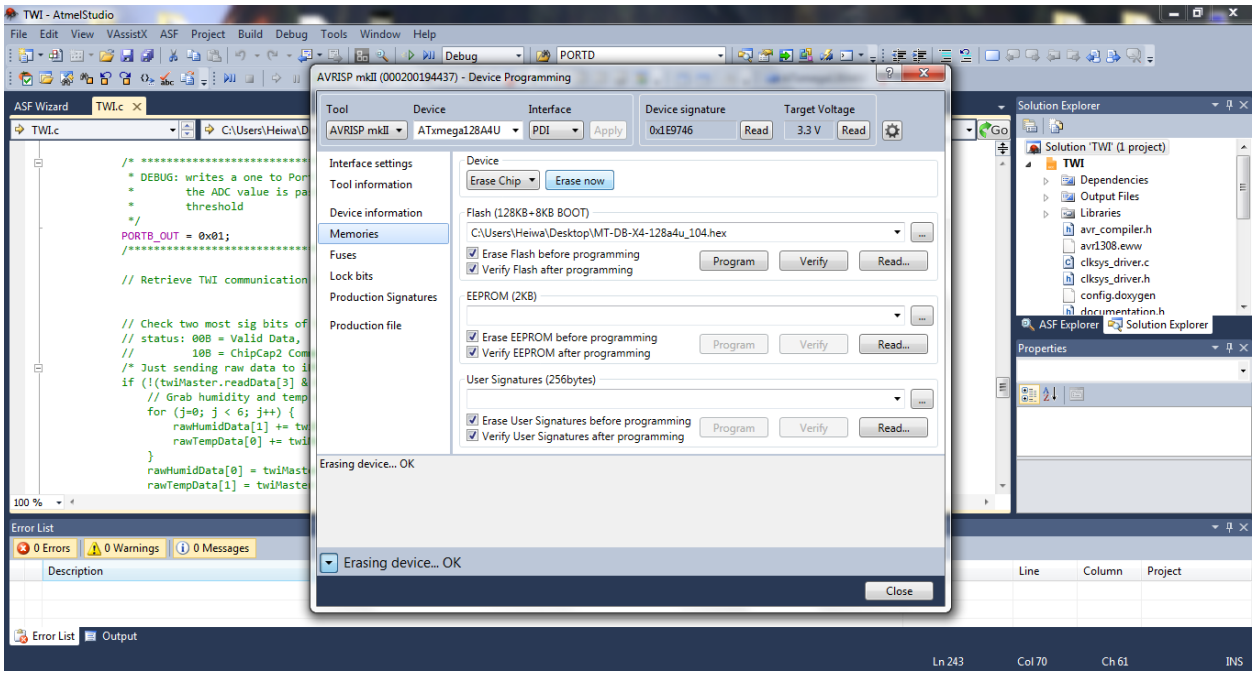


Figure 2.1.6i Memories

4. Next, make sure that the lockbits are set to NOLOCK, and then hit program.

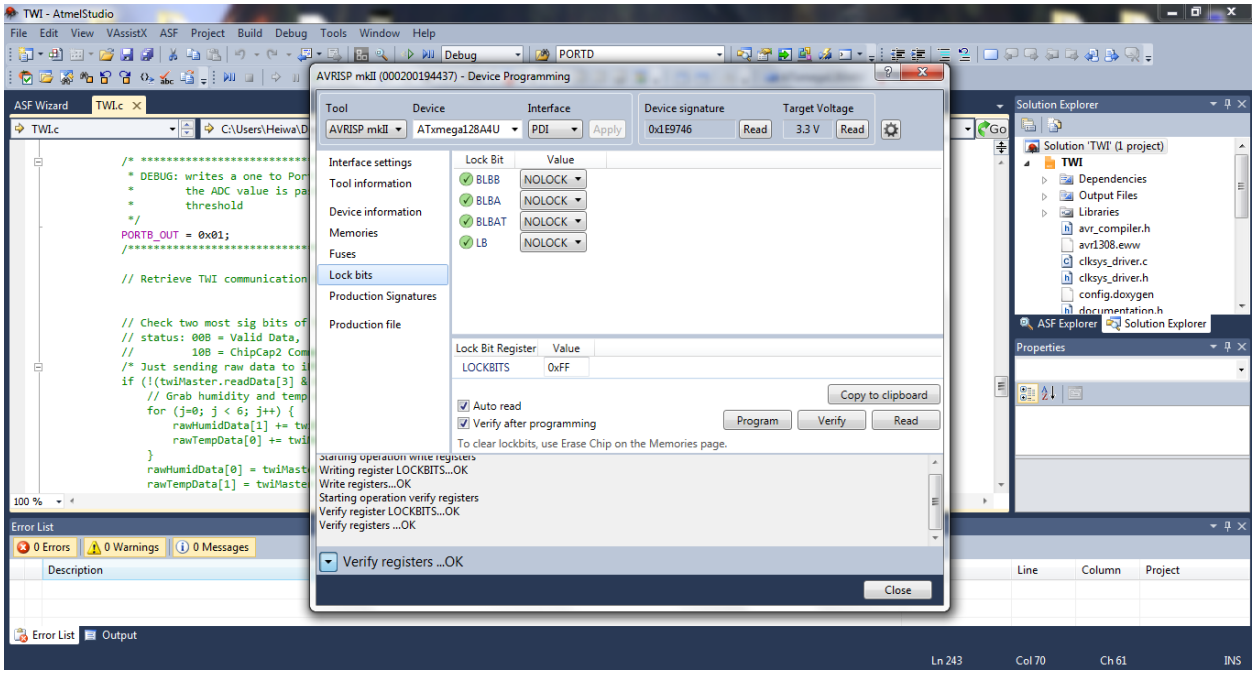


Figure 2.1.6ii Lock Bits

5. Next, in the "Fuses" section, set Fusebyte1 to 0x00, Fusebyte2 to 0x9F, Fusebyte4 to 0xFF, and Fusebyte5 to 0xFF. This is shown in **Figure 2.1.6iii**. Click "Program."

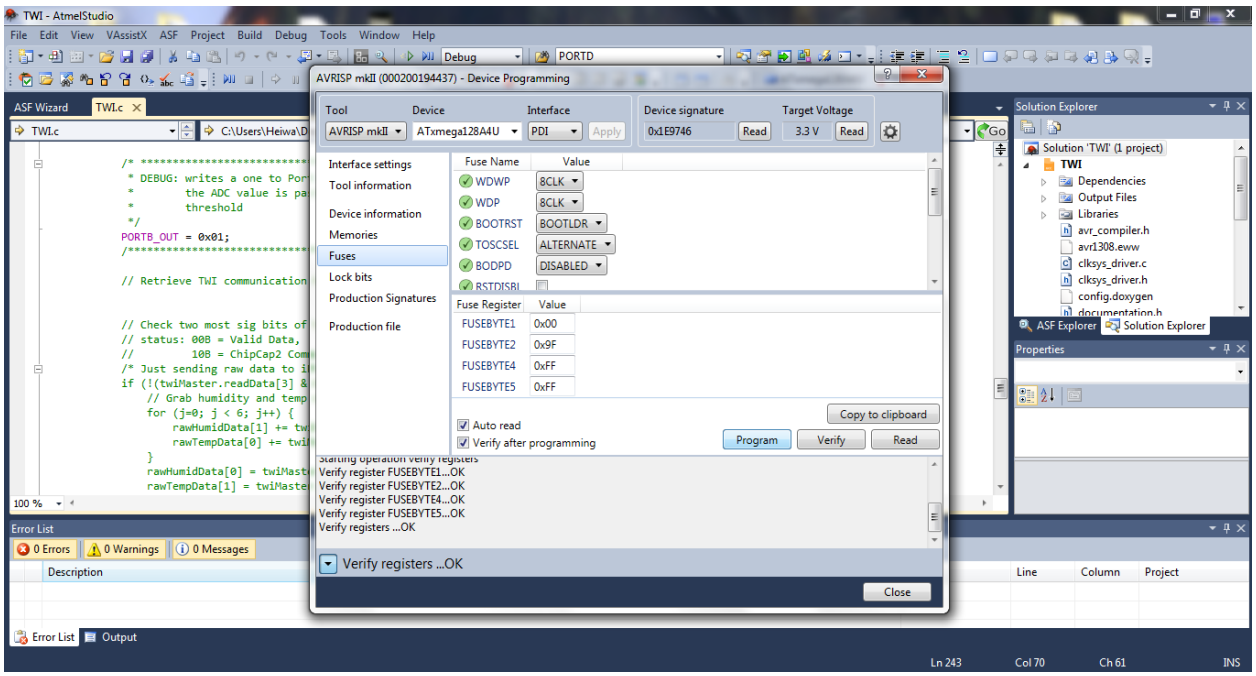


Figure 2.1.6iii Fuses

6. The last step is to flash the bootloader. After downloading the bootloader, select it in the "Memories" section and program it. This step is shown in **Figure 2.1.6iv**.

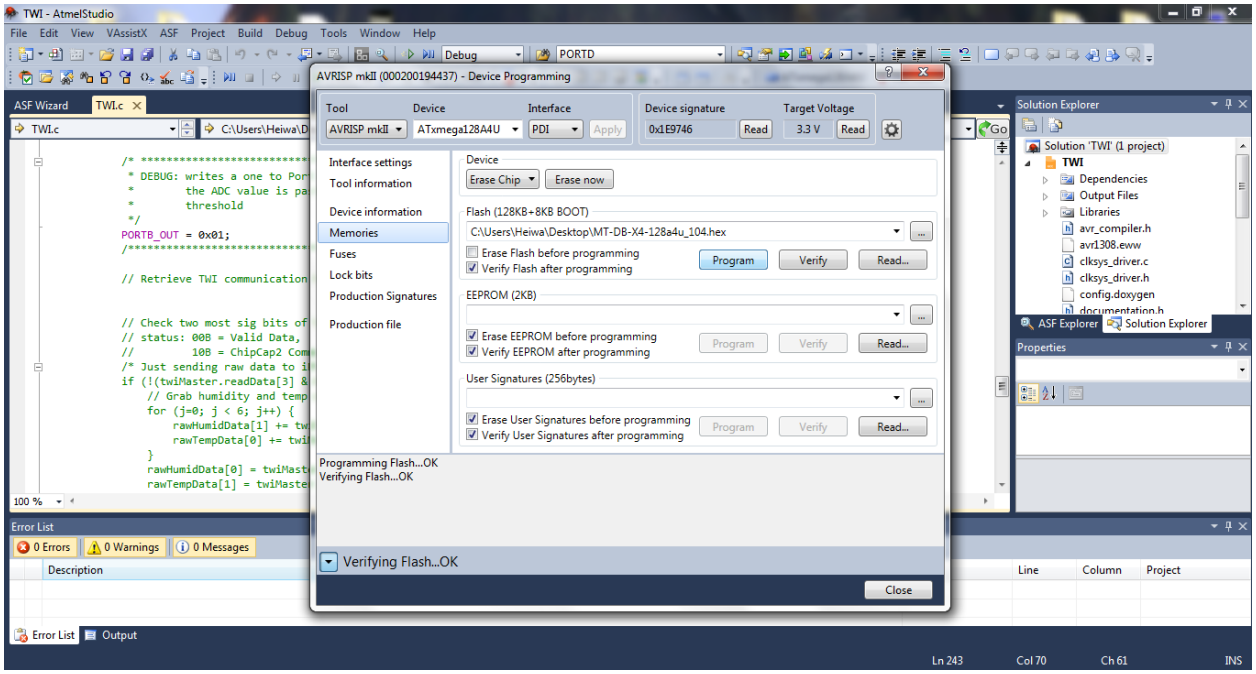


Figure 2.1.6iv Selecting the bootloader

After following the steps above, the Pluggable Sensors PCB should be able to be programmed via USB.

2.2 iOS Application

The main goal of the iOS application was to collect mass quantities of data for our visualization software. This is where our first design choice was made. Before we knew that we were going to be writing the app for iOS, Apple's mobile operating system, we had to decide between Android and iOS. Both contains a very large number of users, and both mobile operating systems would allow us to achieve our goal, but the trade-offs which were choice of programming language, our experience, and security considerations, helped us make the decision of which platform we would be developing for.

Objective-C is currently the language of choice for iOS programming. It is a strict superset of the C programming language meaning that standard C programs will compile using the Objective-C compiler, however it is an object-oriented programming language. That being said, Objective-C is a lower level programming language due to the fact that we can use both a combination of C and Objective-C in the same program. This allows better access to the hardware on the device via Apple API's. We decided since we were going to be making a hardware sensor system we needed to be able to interface directly to the hardware on the device. Android on the other hand is written in the Java programming language, which does not have not have much low-level access. Therefore firmware development is done using a mixture of C and Java; however, you need to different compilers to do this, and you cannot distribute firmware over the Google Play store. Apple's App Store would allow us to distribute our project because it still falls in the application space not firmware.

As stated earlier, our team had experience using Objective-C and C as well as Apple's API from other side projects, therefore this was another good reason to go with iOS. We decided that since no

member of the team had written Java code for Android we would be a bit behind and develop the app much slower than on iOS due to our current knowledge of iOS and the languages it uses.

The final trade-off which helped our choice for iOS was the security concerns. Both systems are not perfect, however Android is the currently the main target for Mobile malware. Currently Android users can go to any website and just download applications. This allows users to download apps easier and more quickly, however there is no formal system for application submission. Also this creates distributed access to apps. Compare this with the way Apple handles their application submissions. All apps must be entered to the Apple app store if they want to be redistributed, which makes it harder to get into the store, however once accepted, all apps can be found in one central place, and the users do not have to be concerned with downloading any Malware from the store. For this reason we figured it would allow users to trust our application rather than hosting in on the internet somewhere, where is would have less of a chance of getting seen.

That being said our team developed an iOS application that supports both iPhone 4 through iPhone 5s running iOS7.1.1 and can be used to collect data from both internal and external sensors, upload this data to our server, and plan optimized collection routes for high priority data. By using the iOS platform as our main data collection source, we are able to quickly and cost effectively distributed this part of our ecosystem.

2.2.1 Data Collection Overview

There are so many forms of data that can be collected via an iOS device. The main problem we faced was which forms of data should we collect, or what data can we collect that we can turn into useful information. For example, we have access to accelerometer, gyroscope, magnetometer, gps, camera, microphone, and ambient light sensor. We could have used all of these, but the focus here is to get the most useful information. The accelerometer, gyro, and magnetometer are incredible devices, but not the most useful for this application. These would mainly be used to determine the orientation of the device, which is very interesting, just not for our dataset. With that said, our main interests were population data which is collected using imagery data captured from the front and rear camera, ambient noise data that is collected via the iOS device's microphone, and environmental data such as temperature and humidity data collected using our custom sensor system design. Using images the user captures, our app natively runs computer vision detection algorithms to determine faces and pedestrians which give us an estimate of population in an image. We provided the average decibel level data using the built in microphone, and addition to the sensors that are built into the iOS device itself, we use our custom sensor system add-on. This sensor system, or pluggable sensor as it is referred to throughout this document, interfaces with the iPhone’s 3.5mm headphone jack. This pluggable sensor system is capable of interfacing with any sensor that is powered off of 3V DC and less than 2mA. The user simply inserts the device into the headphone jack and they now have access to the connected sensor data, which as our proof of concept is a temperature and humidity sensor.

Before diving into the details, this document needs to describe how the data is organized behind the scenes. Over the entire application we use a data model, which is an Objective-C class, which formalizes the structure of a single data entity. The required parameters of data are the GPS coordinates in which the data was collected (longitude, latitude), and the time it was taken. This is due to all of our data being geotagged, or geographically collected, and timestamped. Our data visualization software takes advantage of the required fields. In addition, the data model leaves space for altitude above sea level,

horizontal and vertical accuracy of the gps coordinates, an image, ambient noise level, temperature, humidity, and image processing fields which are faces and pedestrians detected in the image in this feature. In **Figure 2.2.1i** we can see the GeoJSON format of the data that can be collected currently on the iOS device.

```
{
    type: "FeatureCollection",
    features: [
        {
            type: "Feature",
            geometry: {
            type: "Point",
                coordinates: [Longitude, Latitude]
            },
            properties: {
                time: String,
                altitude: Double,
                h_accuracy: Double,
                v_accuracy: Double,
                text: String,
                noise_level: Double,
                temperature: Double,
                humidity: Double,
                faces_detected: Integer,
                people_detected: Integer
            }
        },
        ...
    ]
}
```

Figure 2.2.1i REST API GeoJSON Interface

The properties dictionary is the main bulk of data in addition to the geometry dictionary's coordinates. Any field with a star is required as well. Over the entire project we are using GeoJSON formatted data, therefore our data model conforms to the GeoJSON standard, and can easily be converted between an Objective-C dictionary object and a GeoJSON string. GeoJSON is a subset of JSON (JavaScript Object Notation) and is explained in detail in section 2.4.2. When the user collects a data entry it is referred to as a feature, and a group or set of data entries are called FeatureCollections. Using this data model we are easily able to package FeatureCollections and send them to our server for storage. In addition, this allows our web framework to query the database for any data the iOS app sends up without any additional overhead, meaning the data is already in the correct format for usage. In **Figure 2.2.1.ii** is a quick image of how a FeatureCollection is built what types of feature can be added to a FeatureCollection.

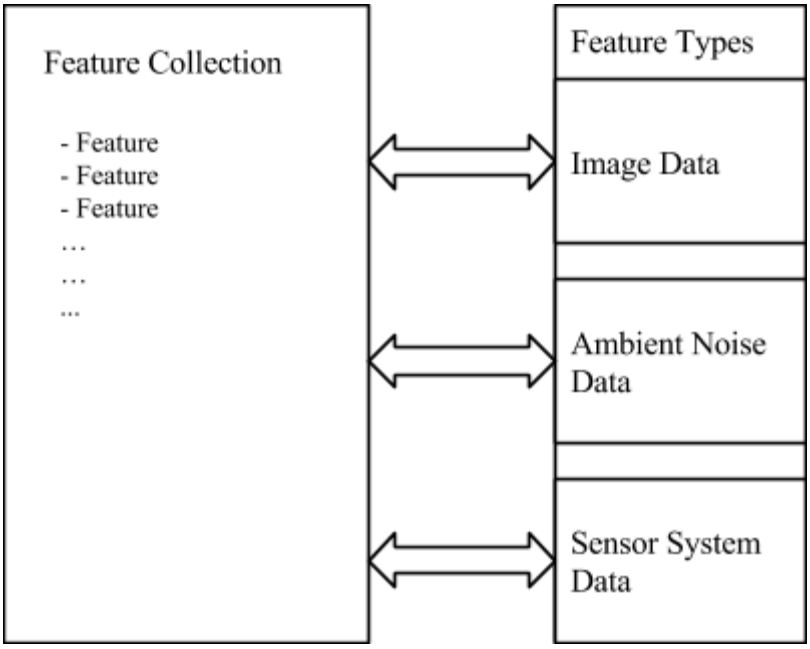


Figure 2.2.1ii Live Data Collection Diagram

2.2.2 Images

As stated above, the camera is how we collect imagery data in our application, and the user has access to both front facing and rear facing camera on the device. The user can easily select imagery data in the data selection menu and they will be brought to a screen which presents the camera. Now once the user takes a picture is where the interesting things happen. Each image that is taken, gets geotagged immediately after being taken. This eliminates the possibility of it being falsely tagged with incorrect gps coordinates, and of course this is all behind the scenes. The user can decide if they like the picture, and can take several pictures to add to their current FeatureCollection. Once the user is happy with their images, the next step is image processing. Our application runs OpenCV, or open computer vision algorithms to detect the number of faces, or pedestrians in an image. These algorithms are written by contributors of the OpenCV framework. The framework allows the programmer to enhance the haar classifier, which is an xml file that contains the trained data. This way they programmer can detect anything they want in an image, however for our proof of concept we used the provided haar classifiers for facial and pedestrian detection. However, the expandability of OpenCV shows how this technology can easily evolve. By processing the images this allows us to estimate the number of people at a location, and because these images are geotagged it allows us to make this correlation. One important thing to note is that images are reduced down to a smaller size, instead of leaving them at the native resolution at which they are taken. There are several reasons for this. First OpenCV has a direct correlation between processing time and image size. For example, when the user takes a picture with the main or rear facing camera, the data size can be upto 30MB. The first problem with this is that running a file of this size through our natively running OpenCV algorithm could potentially cause the application to crash. Secondly the user would have to wait several minutes for the algorithms to complete before they could see the results and that is far too long. Finally saving 30MB images to disk heavily deteriorates the usability and user experience of the application. Chunks of data of that size will make the user interface lag as it is

currently implemented. When the user is ready to upload 10 images that are 30MB each, that will mean they upload 300MB of data using their data plan or wifi, and depending on the upload speed of the connection could take a very long time. By reducing the image resolution we increase speed of the user interface, speed up the image processing time, and speed up our upload rates to the server. Another quick side note, normal image processing on images that are 30MB up to 1GB or larger are generally processed using cluster graphics processing units. We are attempting to make this work on a multi core A6 or A7 iPhone processor, so there are quite a few design trade-offs.

There is a potential update to the way images are taken in our application that would allow more user flexibility and in a future implementation should be added in. Our application uses a `UIImagePickerController`, which is an Objective-C class that loads a camera interface very quickly and easily. There are some major trade-offs with the `UIImagePickerController`. First, it is very easy to get started with, however it has limitations, such as not allowing the programmer to select a camera resolution. Instead of the `UIImagePickerController`, our application could be updated to use a different camera framework, the `AVFoundation` framework which has an API that would allow our camera resolution to be programmatically set. This way we could set the exact resolution of the images before we even take the picture, and we would not need to resize any images. One problem we faced with the `UIImagePickerController` is that the image resizing would rotate the image as well as resize it and we would have to rotate the images back to the proper orientation before running the OpenCV algorithms on the image. This would be eliminated if our application would have used the `AVFoundation` framework. Also we could build in a user interface that would allow the user to select the resolution as well using a one-dimensional slider that kept the resolution of the images at the correct aspect ratio.

There are currently limitations of how many images can be taken in one FeatureCollection or any type of data for that matter, and it depends on the model of iPhone the user is collecting data with. The newer models have more RAM, therefore can store more data in a FeatureCollection at the moment. The live data FeatureCollection is stored in RAM, and when RAM fills up, the operating system will send the application a message that it is taking up to much memory. If the user collects too much data in their live feature collection the application will be terminated by the operating system. Normally in this situation a developer would release as much memory back to the system, however since we are collecting data, which is the reason for the systems low memory errors, there is not much we can release back. Now there are several workarounds which we are aware of and would be a great update to the current implementation. An easy fix would be to save the current FeatureCollection to disk and start a new one, however this may confuse the user to thinking that they lost all their recently collected data. However it is not a bad idea and can be easily implemented in using our data model if needed. For this reason, we currently suggest the user's FeatureCollection contain less than ten features with images. If the user wants to collect more they can send the current FeatureCollection to the server and continue, or just save it to disk and continue. The reason we chose to leave the FeatureCollection in RAM was to reduce writing everything to a file until it was necessary, and trying to reduce user confusion. In addition, while the FeatureCollection is loaded in RAM, this allows our user interface to have a smooth feel such as scrolling through all the data. If each feature was loaded from the file system rather than from RAM, the scrolling would feel very slow or have laggy feel. There are enhancements that could be done to fix the memory issues and allow the live FeatureCollections to grow to very large sizes. The best approach would be to save the data to the file system when memory was getting low, and in addition keep only the data that is

viewable in the table in RAM. When the user scrolls load some of the data into RAM that will be viewable soon, while releasing some data from RAM that is no longer viewable. This workaround will allow FeatureCollection to grow massively in size, while keeping the user interface clean and responsive. This functionality will be added in version 1.1.

2.2.3 Ambient Noise

To collect ambient noise data we took advantage of the onboard microphones present on iOS devices. In order to maintain user privacy we do not store the audio data that is obtained to provide our ambient noise results. This is accomplished using Apple's `AVAudioSession` framework its `AVAudioRecorder` property. The `AVAudioRecorder` provides a callback routine, `updateMeters`, that collects a small number of audio samples and performs a conversion from amplitude to a decibels (dB), the relative power of the audio signal. The `updateMeters` call saves the average and peak dB levels, for it's duration window. The window of time in which the audio samples are collected by `updateMeters` is too small for our application of estimating the audio levels in a given environment. To correct for this we call `updateMeters` the minimum number of times to provide the most accurate results, determine through our own testing. Our ambient noise functional is accurate within 5% of a known audio sample. To maintain this level of accuracy and to avoid from collecting bad data that may occur from alternate microphone we limit the audio collection to on board microphone collection only. This is ensured by our sensor alert system, described in section 2.2.5 later in this document. If a developer would like to use our ambient noise detection class in their own project that just need to follow this steps: (1) Import our `GSFNoiseLevelController` class into their project, seen in **Figure 2.2.3iii**. This means that both the `GSFNoiseLevelController.h` and `GSFNoiseLevelController.m` files need to be copied into the project.

```
#include "GSFNoiseLevelController.h"
```

Figure 2.2.3iii Import the `GSFNoiseLevelController` class

(2) Declare and instantiate an instance of the class, seen in **Figure 2.2.3iv**. This requires the developer to pass the view controller that will be associated the sensor collection to the `initWithView` function call. This view will be used for any alerts that need to be displayed on during the collection process (the alert system is described in section 2.2.5 later in this document).

```
@property (nonatomic) GSFNoiseLevelController *ambientNoise;

// Initialize the noise level controller
self.ambientNoise = [[GSFNoiseLevelController alloc] initWithView:self.view];
```

Figure 2.2.3iv Declare, allocate and initialize a `GSFNoiseLevelController` object

(3) Begin the sensor monitoring process by calling `monitorNoise`, as seen in **Figure 2.2.3v**, with the sensor attached. The `monitorNoise` function takes a boolean value where YES/ture starts the collection process and NO/false stops the collection process regardless of packet transfer status.

```
// Start ambient noise monitoring services
[self.ambientNoise mointorNoise:YES];
```

Figure 2.2.3v Start ambient noise monitor services

(4) Final call the function `collectNoise`, seen in **Figure 2.2.3vi**, and store the returned value into an `NSMutableArray`.

```
// Collect ambient noise readings
[self.ambientNoise collectNoise];
```

Figure 2.2.3vi Collect sensor data

2.2.4 Pluggable Sensor Power and Two-way Communication

The backend of the iOS application handles both power and two-way communication for our pluggable sensor. In order to provide both these functionalities via the 3.5mm audio jack on the iOS device we needed full access and control to both the input and output lines of the iOS device. The Apple audio frameworks, for iOS devices available to us, works on a multi level abstraction system, seen in **Figure 2.2.4i**.

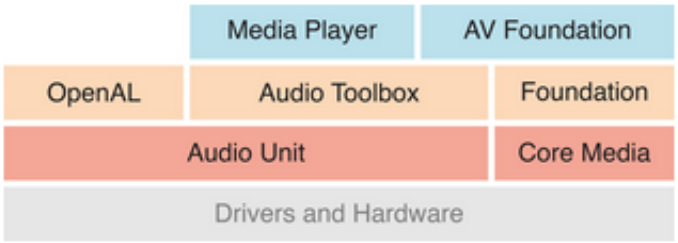


Figure 2.2.4i Audio frameworks in iOS[1]

We attempted to build our power and communication network from the higher level `AVFoundation` but the response time of this approach was too slow to incorporate real time communication between the iOS device and microcontroller. So to fix this we accessed the `Audio Units` directly which provided us with the low latency response for our system. Setting up the `Audio Units` ourselves allowed us to specify the how our application interacted with the iOS devices hardware and although this was ideal from performance standpoint it made making our application expandable amongst various iOS device challenging. This problem exists because the ADC, used for audio processing between, differs greatly between models of iOS devices mainly in terms of buffer sizes. Apple developed a helper class for just this issue, `CAStreamBasicDescription` but they have not taken the time to update this class for versions of iOS greater than 6.0, and it contains many function calls and defined variables that are now deprecated. Because of the time constraints and Apple's lack of upkeep on their API calls we put our focus into making our external system work with the latest hardware available, found on the iPhone 5s at the time of writing this document. Future updates to the iOS application should include the update version of the `CAStreamBasicDescription` helper class when it becomes available.

[1] The Audio Unit Host Fundamental PDF can be found at https://developer.apple.com/library/ios/documentation/MusicAudio/Conceptual/AudioUnitHostingGuide_iOS/AudioUnitHostingFundamentals/AudioUnitHostingFundamentals.html

To provide power to our pluggable sensor system we generate a 20 kHz sine wave in software at 95 percent of the available amplitude for our `Audio Unit`, and this percentage is based on the variable types used for the input and output buffers, a signed integer. We send this tone out of the left channel of the 3.5 mm audio jack by filling the audio buffer for that channel in an interrupt callback function we setup at the time of the `Audio Unit`'s initialization. This callback function is invoked anytime the input buffer becomes full, and when invoked it refills the output buffers with the next section of the sine wave. Setting the amplitude of our sine wave in software is not enough to provide a enough voltage for the pluggable sensor, so in order to reach the required amplitude for powering our pluggable system we adjusted the master volume to the maximum amount. Apple does not allow for direct programmatic setting of the master volume, but rather requires the user to manually set this with either the physical volume buttons, or with a `UISlider` object. We didn't want to inconvenience the user with having to manually increase the volume each time they wanted to collect data using our pluggable sensor, and we wanted to avoid the risk that they forgot to restore the volume to a safe listening level before using other audio applications. To bypass this problem we notice that when we instantiated the master volume slider, from the `MediaPlayer` framework that the view associated with it used a standard `UISlider` object that had just been altered to disable setting it's level in software. We assumed that this special `UISlider` inherited from the standard `UISlider` we had access to, and in fact it did. This allowed us to instantiate our own standard `UISlider` and bind it the master volume slider. By binding the these two `UISliders` together we are able to programmatically set the master volume to the level our application needs and back done to a safe listening level without hassling the user. Because this method is may be blocked by Apple at a later date we have placed a backup system that will ask for manual user volume setting and only be invoked if ever the autoset fails.

In the callback function we also handle the processing of incoming data from the audio jack's microphone line, as well as filling the output communication on the right audio channel when necessary. The data coming in on the microphone line is Manchester IEEE encoded data in the form of an AC AM signal. This means that we use amplitude modulation to distinguish between a high and low state, where a high state is represented by the presence of an AC wave greater than a set amplitude and a low state is anything that falls below this amplitude threshold. Manchester IEEE encoding is a binary phase encoding scheme that provides frequent state changes because the binary data is represented as: a change from the high to low state to represent a binary zero and a change from the low to high state to represent a binary one. An example of the resulting data looks like that of **Figure 2.2.4ii** which represents one encoded packet from our pluggable.

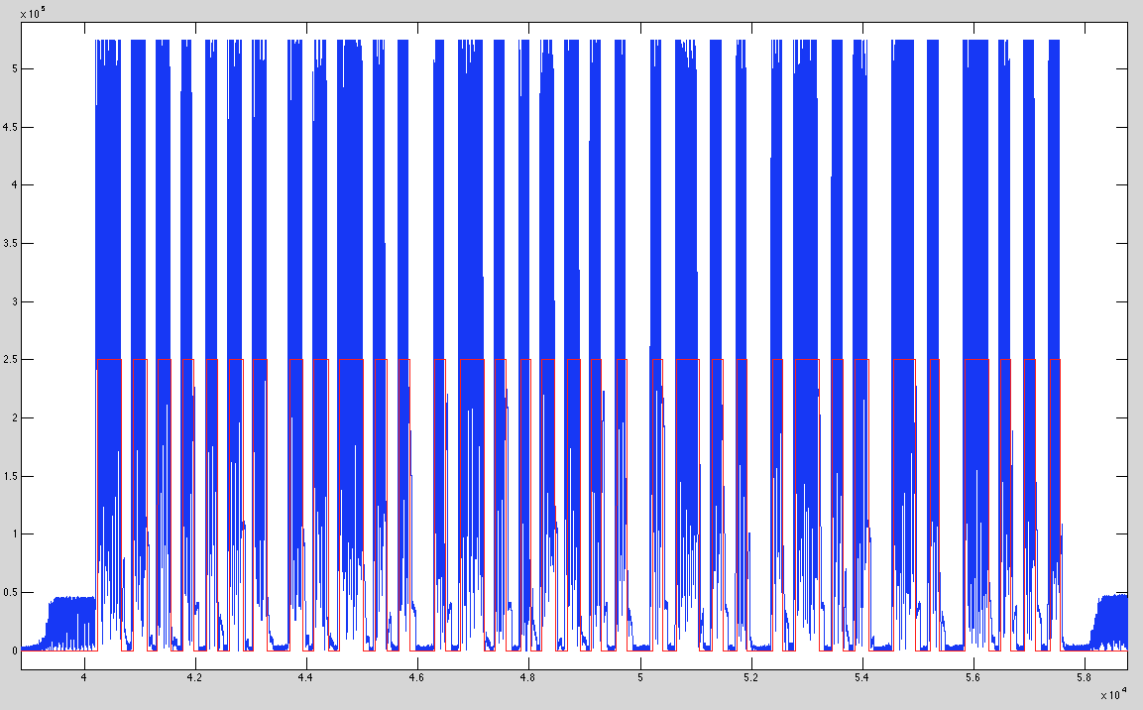


Figure 2.2.4ii Example input data as seen by the iOS device

In **Figure 2.2.4ii** the blue represents the absolute value of the raw data input that is returned from the `AudioUnitRender` function called within the callback function. `AudioUnitRender` is used to grab the data processed by the hardware and conform to the format specified at the `Audio Unit` initialization. The red line is a representation of the corresponding manchester states given a threshold 250,000. The x-axis the sample number and the y-axis the absolute value of that sample.

Each packet of data sent over the line consists of a start pattern, four bytes of data and a byte with the checksum value of the four data bytes. The checksum is a summation of all the bits in the four data bytes and is created on the microcontroller. The start pattern is composed of two low states followed by a high state. The data is sent over the line as little-endian (least significant bit first). Our decode algorithm hits the appropriate number of bits, an error is detected, or the line goes flat we take a summation of the received bits and compare it against the received checksum. If the checksums do not match we throw out the data and continue requesting a packets, otherwise we decode the data bytes into their appropriate temperature and humidity values. The request for packets is sent to the microcontroller through the right audio channel in the form of an enable signal. This enable signal is represented by a sine wave at half the amplitude of the power tone wave and is fed directly into the ADC of the microcontroller.

All of the communication happens behind the scenes for a developer if the iOS device is communicating with our pluggable sensor system. The collected sensor data is returned in the form of a `NSMutableArray` where the object at the zero index is the average humidity reading, the object at index one is the average temperature reading, and the total number of packets received that the two reading were averaged over at index two. To collect this data a developer simply needs follow these steps: (1) Import our `sensorIOController` class into their project, seen in **Figure 2.2.4iii**. This means that both the `sensorIOController.h` and `sensorIOController.m` files need to be copied into the project.

```
#import "GSFSensorViewController.h"
```

Figure 2.2.4iii Import the `sensorIOController` class

(2) Instantiate an instance of the class, seen in **Figure 2.2.4iv**. This requires the developer to pass the view controller that will be associated the sensor collection to the `initWithView` function call. This view will be used for any alerts that need to be displayed on during the collection process (the alert system is described in section 2.2.5 later in this document).

```
@property (nonatomic) GSFSensorIOController *sensorIO;

// Intialize pluggable sensor collection object
self.sensorIO = [[GSFSensorIOController alloc] initWithView:self.view];
```

Figure 2.2.4iv Declare, allocate and initialize a `sensorIOController` object

(3) Begin the sensor monitoring process by calling `monitorSensors`, as seen in **Figure 2.2.4v**, with the sensor attached. The `monitorSensors` function takes a boolean value where YES/ture starts the collection process and NO/false stops the collection process regardless of packet transfer status.

```
// Start collection
[self.sensorIO monitorSensors:YES];
```

Figure 2.2.4v Monitor sensor data

(4) Final call the function `collectSensorData`, seen in **Figure 2.2.4vi**, and store the returned value into an `NSMutableArray`.

```
// Grab collected sensor data
self.sensorData = self.sensorIO.collectSensorData;
```

Figure 2.2.4vi Collect sensor data

2.2.5 Sensor Alert System

By using the 3.5 mm audio jack as our power and communication channel there were certain user interaction circumstances we needed to handle for both the ambient noise and pluggable sensor data collection. To handle these circumstance we developed an alert system that built into both the ambient noise and sensor collection classes. As stated in section 2.2.3 earlier we only wanted to collect ambient noise data from the onboard microphone and as mention in section 2.2.4 a sensor must be connected via the 3.5 mm audio jack in order to collect sensor data. The iPhone automatically detects whether or not a headset is inserted in the headphone jack. When a headset is detected, the iPhone reads microphone data from the microphone on the headset. When a headset isn't detected, the microphone will instead read from the onboard microphone. Thus, we wanted to ensure the user doesn't have the sensor plugged in when he is trying to collect ambient noise data, and that the user does have the sensor plugged in when he is trying to collect pluggable sensor data. To do this, we took advantage of audio route change and interruption callback functions available for set up with the `AVAudioSession` framework. We use the

audio route change callbacks to check for sensor/headset insertion and removal, and we use the interruption callbacks to deal with interruptions in the audio streams due to a switch in active audio application. When any of these audio callbacks is invoked it provides the necessary tasks associated with the event. If the sensor is removed during the collection of data or the user starts to get a phone call our application immediately stops the collection and sets the master volume to to 50 percent, a safe listening level (seen in **Figure 2.2.5i**). Once the user returns to our application they are prompted with an alert that notifies them how to quickly get back to finishing their collection process right where they left off.

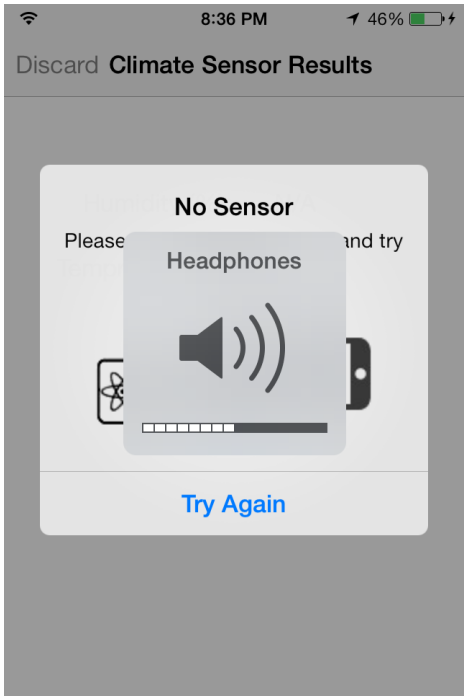


Figure 2.2.5i Audio callback function automatically adjusting volume

To implement alerts that would not only tell the user the problem but give them a instructions and visuals that would about how to fix the problem. Apple's provided alert UI system does not allow for graphics or other UI elements to be added, which makes it difficult to give the user quick and easy interaction with an application. To get this functionality we used the help of a third party system called SDCAlertView[2], which is available on GitHub and Cocoa Pods. This alert class has the same look and functionality as the latest implementation of the Apple aler UI but with the ability to add any UI element to it. This allowed us to add image showing the user whether or not the sensor needed to inserted or removed depending on the data collection type (seen in **Figure 2.2.5ii**), and adjust the volume up or down directly in the alert if the auto volume set were to fail.

[2] SDCAlertView can be found at https://github.com/Scott90/SDCAlertView

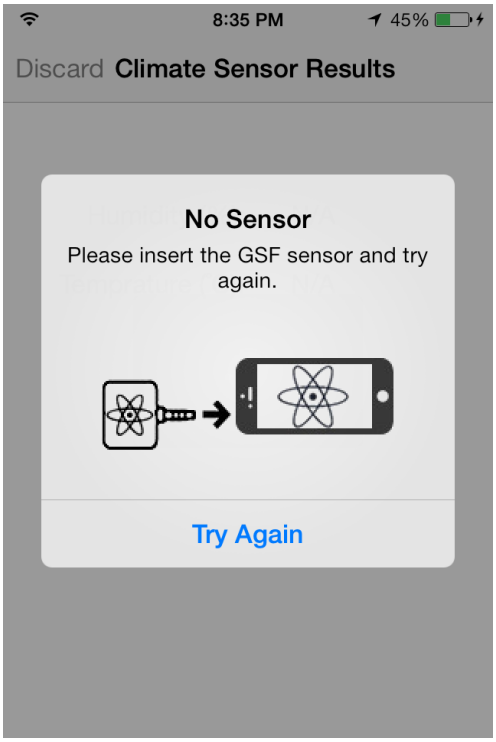


Figure 2.2.5ii Easy to follow alert messages

2.2.6 Offline Usage and Persistence

An interesting problem that we accounted for was offline usage of the application, or data collection while the iOS device was not connected to wifi or had no cellular reception. In addition, our application accounts for data that failed to upload to the server for any reason. In our application, we use the iOS file system to save collected data that the user is either unable to send, fails to upload to our servers, or simply wants to save and send later. This interface is similar to the data collection interface, however, instead of just a single FeatureCollection, the saved data interface will show all saved FeatureCollections that are stored in the file system. In iOS the file system works as shown in **Figure 2.2.6i**.

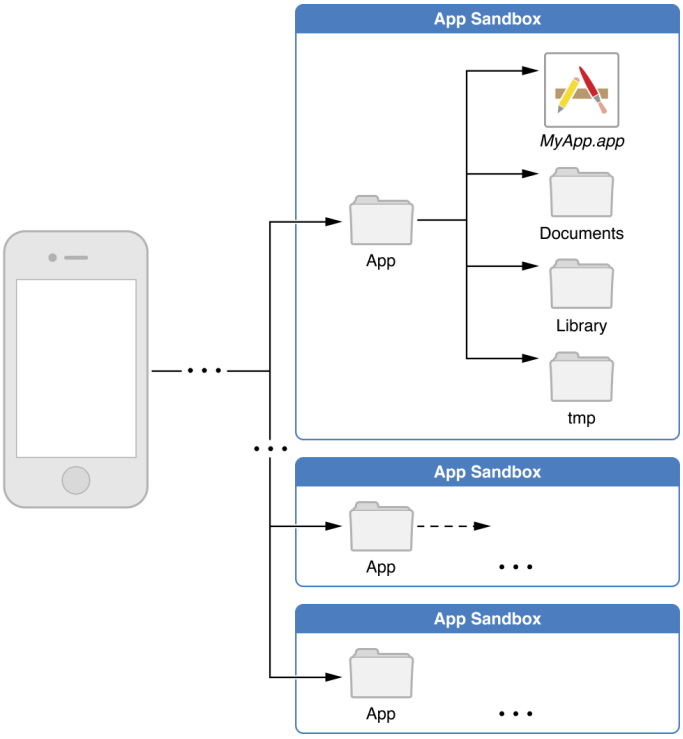


Figure 2.2.6i The iOS file system sandbox methodology.[3]

Figure 2.2.6i demonstrates the iOS sandbox methodology. Each application is isolated from all other applications on the device. This enhances security of each application. No other applications can go into our applications file system and tamper with the data the user stores in anyway. This is one of the reasons why we chose iOS for this project. We use the `Documents` directory of our applications sandbox, and create a subdirectory called `GSFSaveData` inside the `Documents` directory which is where all the user's FeatureCollections are stored. In our saved data interface our application allows the user to see all Features in a FeatureCollection grouped together with clear separations between other FeatureCollections. This allows the user to upload a single FeatureCollection easily, or they can upload all of them with a single tap. In addition, the user can tap on a single feature and see all the data that is collected in that feature.

All the data has to be stored in the filesystem as `NSData` which is an Objective-C class which is essentially a byte buffer. As stated in section 2.2.1, our data model or custom data class, can easily be converted between an `NSDictionary`, which is Apple's implementation of a dictionary data structure, and a `GSFData` object which is our custom data class. The reason we convert our data model into an `NSDictionary` is so we can convert the data into JSON data. This may seem strange that there are three steps to converting to JSON, however having the collected data in an `NSDictionary` is a mandatory step because we cannot convert our `GSFData` class directly into JSON format due to the requirements of the Apple JSON API. The JSON API only converts certain data types into JSON. Therefore, we created our data model to conform to Apple's JSON model, which will turn our `NSDictionary` into an `NSData` which is then written to the file system. If the data model is

[3] iOS Sandbox methodology.
https://developer.apple.com/library/ios/documentation/iphone/conceptual/iphoneosprogrammingguide/TheiOSEnvironment/TheiOSEnvironment.html

expanded upon it must continue to conform to the Apple JSON model[4]. In addition, we send `NSData` to our server with the Restful API, described in section 2.3.4, due to the fact that we are using TCP/IP layer and that layer sends bytes. Storing `NSData` also allows us to load the data straight from disk and send it with no additional steps.

In our saved data interface the saved `NSData` stored in the file system can be read out and converted back into an `NSDictionary` or `GSFData`, which is used to fill the table with all the information the user wants to see. Now when the user decides to upload some of the data, as mentioned above, the application uses the Restful API. This is very easy on the iOS side because our application creates an HTTP message with a header containing the server url, an eight digit API key used to determine which iOS device sent the data, and a body containing the data. Also the API key is stored in the users iOS keychain so they only need to enter it in once. The Restful API only can handle one FeatureCollection currently, therefore if the user sends multiple FeatureCollections using the upload all selection, all the FeatureCollections are packaged into one large FeatureCollection and sent off to the server. When our application creates the body of the http message it just passes an array of `NSData` JSON formatted objects to the data transfer class for upload, therefore it is easily expandable to allow for any number of FeatureCollections.

The saved data interface would also be updated in version 1.1 exactly the same way as the live data collection interface as mentioned, in section 2.2.2. There are currently limitations of how many FeatureCollections can be loaded into the saved data interface and it depends on the model of iPhone once again. The newer models have more RAM, therefore can load more data from disk into RAM and into the saved data interface. Images are the main problem due to their large size, and if the user takes a large number of images and saves them all to disk the application will be terminated by the operating system if they are all loaded into RAM when loading the saved data interface. This is due to our current implementation. The application currently creates an image cache so that scrolling through the interface is seamless, however if too many images are saved to disk the application will get locked into a state that will always crash when the user goes to their saved data table because the application will try to load all of the images into RAM. For this reason we currently recommend to either send all saved data periodically before it gets extremely full. In the event that the user does not do this and they get the application in an unusable state, they can connect their device to a computer and using iTunes or another program that allows the user to see into their iOS file system they can recover their collected data. The data is not lost if saved to the disk, however we encourage the user to send their data to the server periodically to avoid this situation.

The best approach to fixing this problem would be to keep only the data that is viewable in the table, or will be viewable soon in RAM. When the user scrolls, some of the data is read from the file system and loaded into RAM, while releasing some data that is not in view from RAM. This workaround will allow the saved data to grow as large as the disk space of the device will allow, while keeping the user interface clean and responsive. This functionality will be added in version 1.1, but overall the saved data

[4] Apple JSON documentation.
https://developer.apple.com/library/ios/documentation/Foundation/Reference/NSJSONSerialization_Class/Reference/Reference.html

interface is a really nice way to allow the user to save data and send data to the server that had any errors while transferring over the network.

2.2.7 Route Planning

Our application is all about collecting data, so we decided that the faster our users can collect data the better. We created a route planning feature built into the application that uses the Google Maps API[5]. Google provides an iOS SDK for their map service, and in conjunction with their direction service[6], we can plot a near optimal route between many points so the user can travel areas of interest as efficiently as possible.

Our application allows the user to tap areas of interest and the software will plot an efficient route to the newly added area from the users current location. Each tap will add a new location to the current list of target areas and recompute the route from the users current location. Also the route is currently set up to be round trip. meaning it will plot the route from the user location to all other points and back to the users original location. We designed it like this assuming the user would be starting from a headquarters of some kind and need to return after their data collection route. An A-Z implementation, which goes from the users current location to their end destination was created as well but it currently not built in, however would be added in update 1.1.

Mention above we are using the Google Maps SDK for iOS as well as their direction service. The Google Maps SDK is used for rendering views on the users screen. For example, drawing the map on their screen, drawing markers, and drawing polylines. Polylines[7] are extremely important and provided in the SDK, however polylines that show a route are retrieved using the Google direction service. A polyline is a line between two points that can be drawn on the map, however the polylines that are returned by the direction service are called complex polylines because they follow the roads that a user would take to follow a route. Polylines encodings are quite simple. The direction service figures out all the GPS coordinates that the road follows. Each bend and turn in a road needs to have a separate polyline, and because polylines are simply lines on the map, Google provides an encoding scheme to append a group of polylines together. Each time a part of a road is no longer straight there is a new polyline appended to the current group of polylines. The direction service will give back all the gps coordinates along that road to create an overview polyline, which is all the encoded polylines all appended together. For details on the encoding see footnote 7. An simple example of how polylines are drawn can be seen in **Figure 2.2.7i**.

[5] Google Maps SDK for iOS. https://developers.google.com/maps/documentation/ios/
[6] Google Maps Direction Service. https://developers.google.com/maps/documentation/directions/
[7] Google Maps Polyline encoding. https://developers.google.com/maps/documentation/utilities/polylinealgorithm

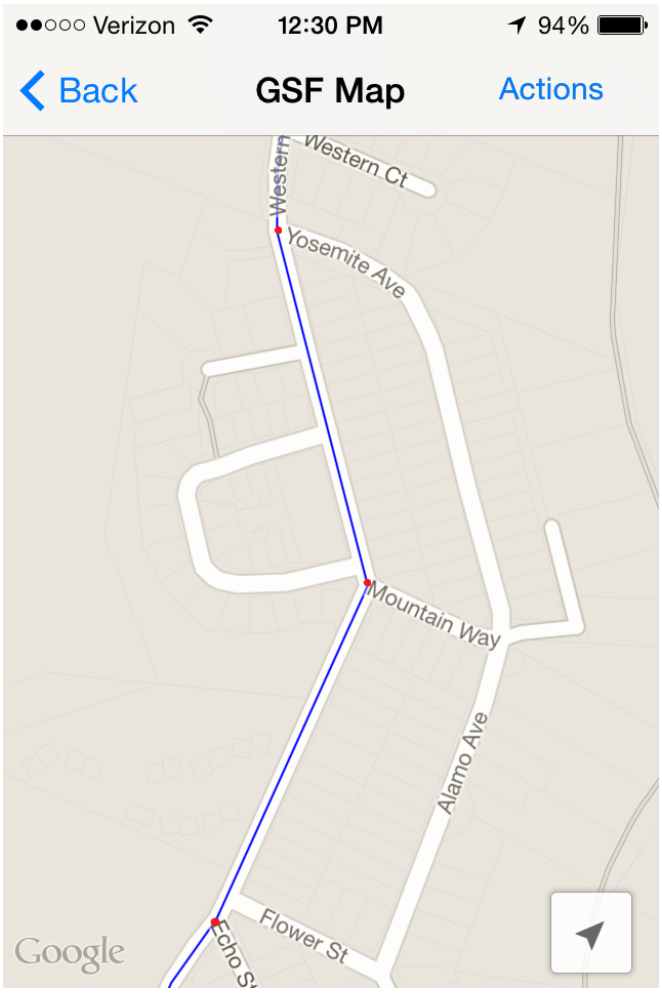


Figure 2.2.7i Basic Complex Polyline

As shown above there are two polylines that are clearly visible and have marked using red dots. They are straight lines between two gps coordinates and when encoded together it forms a large continuous line. Our application uses polylines to plot the route for the user round trip.

Once the user has plotted a route they can travel to those locations and collect, but in addition to that our application gives the user several actions they can take. First, fit the camera, which is the current view, to fit the entire route on the screen. This is useful for seeing the overview of the entire route before the user heads out. Second, the user can plot the first leg of the route in the native Google Maps iOS application if it is installed on their phone. To do this we use the Google Maps url schema, which is a way to send a message from one app to another app. Remember the apps are sandboxed in iOS so this can cause a security breach, however these breaches can be avoided. Google most likely has their app securely programmed. When this action is selected, the Google Maps app is launched and the leg of the route is plotted with all the features of their application such as voice directions and more. Once feature we wanted to add was Geofencing which is a way of alerting the user that they are very close to the target destination for data collection, and this would be another update in version 1.1. The Google Maps application provides this geofencing which is very useful incase the user is driving or not paying attention to their mobile device. The third action is getting a list of directions from the Google Maps website. Unlike

the Google Maps iOS app, their website in the iOS Safari app will provide a complete list of directions for the entire route start to finish. This is great to have if the user wants text directions for their trip. This feature also uses the url schema mentioned above. Finally if the user is done collecting on this route they have the option to reset the map, which removes all markers and polylines on the map for the current route and allows the user to start over.

In addition to the user input, our application allows for an even better method of plotting routes. We created our own custom and secure URL Schema[8] and carefully followed the secure programming guide[9] to ensure our users safety. Our application allows a server administrator to send a text message to a user, or field agent in this case, which they can tap to launch our application from any text message application they prefer to use. This text will pass the GPS coordinates to our application and automatically launch the route planner and plot the route with a single tap. The user can then use all the actions described above as well, however in version 1.0 of the app, if the user opens the route planner through a text they cannot add any points by tapping the map. This is extremely powerful because it allows an admin to see that there is not enough data collected at certain geo-locations and they can send a text to a field agent to request more data. The field agent can easily plot the route and go collect all the data they need. In **Figure 2.2.7ii** there is an example of what an agent's route looks like in our user interface. The route goes to each marker shown, so in this case there are two in Santa Cruz, one in Livermore, and one in San Francisco. The polylines are optimized between all markers so the user can reach these locations as quickly as possible. Under the hood we are actually sending a download request to the server, which as a file stored containing the gps coordinates of where data needs to be collected. It fiolows the GeoJSON format as well to remain standard with the the rest of our tools. The format is shown in **Figure 2.2.7.iii** using an example we used for testing. The route planning software is a great additional feature we added in, and could even branched off into an additional application if a developer needed to do so.

[8] Apple Documentation on creating custom url schemas.
https://drive.google.com/a/ucsc.edu/#folders/0B6pHxlneVJpCbm55RVpCYVNKUFU
[9] Apple Secure Programming Guide
https://developer.apple.com/library/mac/documentation/security/conceptual/SecureCodingGuide/Introduction.html

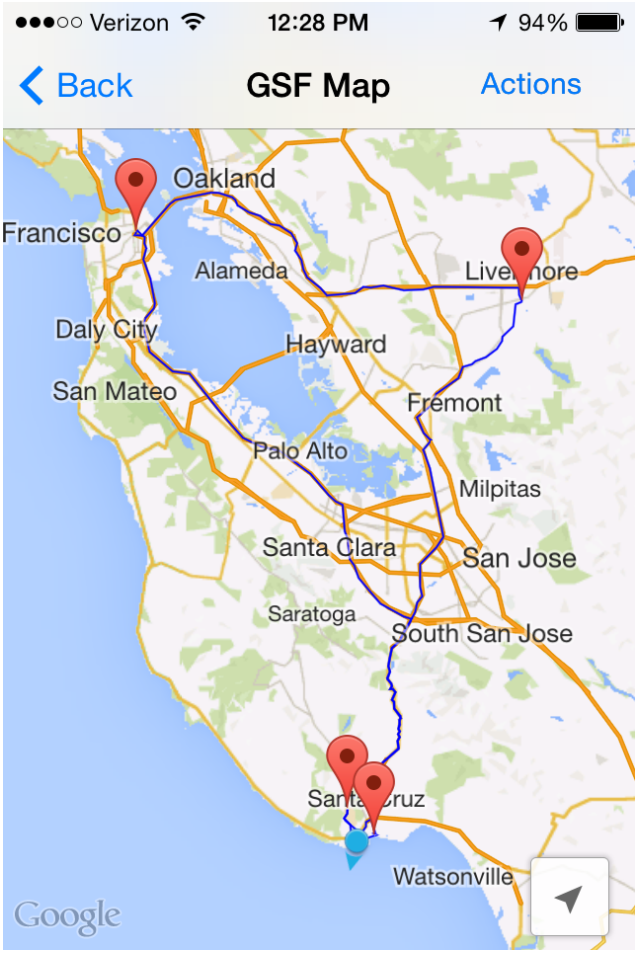


Figure 2.2.7ii GSF Route Planner in action.

```
{
      "type": "GeometryCollection",
      "geometries": [
            {
                  "type": "Point",
                  "coordinates": [-122.0617279, 37.0032456]
            },
                        {
                  "type": "Point",
                  "coordinates": [-122.0213875, 37.0072211]
            },
            {
                  "type": "Point",
                  "coordinates": [-121.9918617, 36.9878901]
            },
            {
```

```
                "type": "Point",
                "coordinates": [-122.0047363, 36.9791141]
            },
            {
                "type": "Point",
                "coordinates": [-122.0375236, 36.9596388]
            }
        ]
}
```

Figure 2.2.7.iii Coordinates File Format for Routing Software

2.3 Server and Web Application

To be able to store the collected data and process and visualize them we had to build a centralized server which also served as a frontend for users to query the collected data. To do so, we used Django as our web framework and Apache as the server program.

Our web application is hosted on UCSC's School of Engineering servers where we were given a Virtual Machine, running Ubuntu 12.04 server OS with 8GB of RAM and 4 CPU cores for intensive data and image processing. Users can interact with our web application at http://gsf.soe.ucsc.edu

2.3.1 Server & Web Framework

The website is hosted using the Apache version 2.2 which serves as a proxy layer that directs HTTP messages to the mod_wsgi daemon. The mod_wsgi program, a module of the Apache server, is the handler that forwards HTTP requests to the underlying Python programs. In this case, users do not have direct access to the scripts and therefore protecting the server from any harmful scripts or injections (**Figure 2.3.1i**).

The Apache configuration uses the default for any standard Django application but we added a daemon mode (**Figure 2.3.1ii**) so that any updates to the python scripts would automatically get transferred to the new version rather that us having to restart the server every time we make a change in the source code.

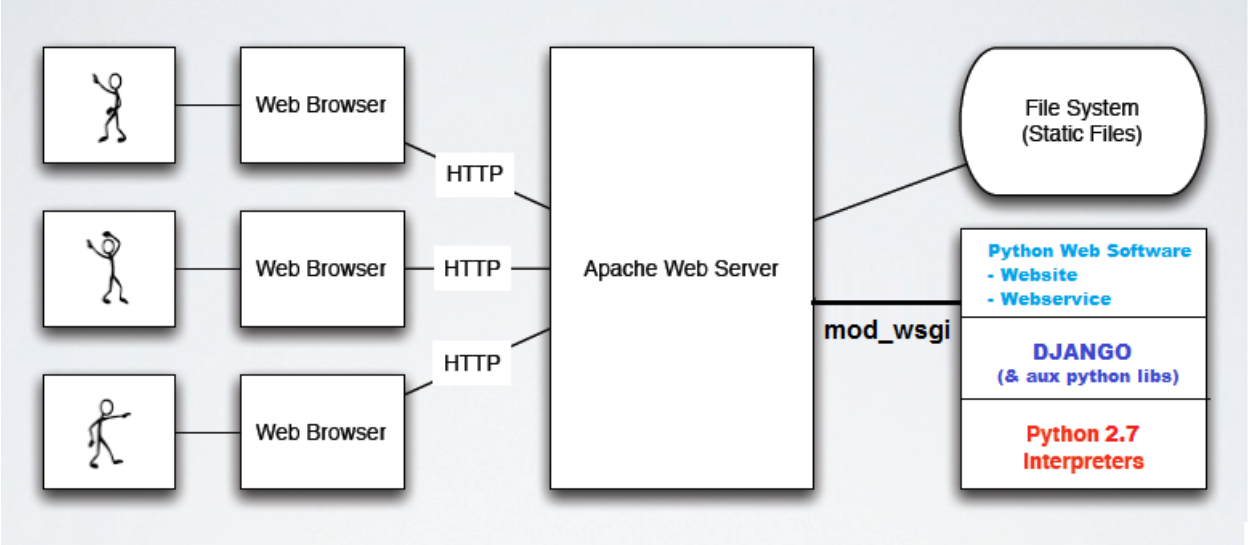


Figure 2.3.1i Apache server combined with mod_wsgi

To allow authentication on the web application and also authenticating incoming upload messages we added SSL encryption to communications coming into the server. As seen in **Figure 2.3.1ii**, any requests sent to the `/admin/*` and the `/api/*` sections of the web app are redirected over to HTTPS to protect any sensitive information such as passwords and API keys.

```
Redirect permanent /api https://gsf.soe.ucsc.edu/api
Redirect permanent /admin https://gsf.soe.ucsc.edu/admin

WSGIDaemonProcess gsf.soe.ucsc.edu processes=1 threads=15 display-name=%{GROUP}
WSGIProcessGroup gsf.soe.ucsc.edu
WSGIPassAuthorization On
```

Figure 2.3.1ii Part of the Apache configuration file showing the redirects over HTTPS

We decided to use a web framework that can provide an underlying structure for our web application rather than constructing everything from scratch. After considering various web frameworks, we decided to chose Django due to its vast community support and extensive amount of third-party modules.

2.3.2 Database

To meet one of the requirements set by LLNL, we decided to use the fairly new concept of No-SQL databases to stay up-to-date with the trend in database technologies. The main difference between a No-SQL and a traditional SQL database is that No-SQL databases are schema free. In a traditional SQL database (db), first thing you have to do is to create a table with columns that would specify strict data types to be stored in them. Imagine an application that uses a SQL db at the backend. If you have to change the schema of your SQL db (add a column or delete one) then you would have to create a new instance of the SQL db with the new schema and migrate the old data to the new db. Then

you would have to halt your application to switch between databases. With a No-SQL database, all you have to do is add a new key, known as a column title in SQL, to the db and start collecting data for it.

After research on various No-SQL Database Management Systems (DBMS) we decided to use MongoDB as the the underlying DBMS. MongoDB was chosen mainly for the vast community support and the guaranteed long term support due to it being owned by the 10gen company. The core of MongoDB is written in C++ and has easy and fast installations on Windows, Linus/Unix, and OS X so it will be easy for other people to recreate our environment on their personal systems. In a paper published by Google[10], the performance of different programming languages are compared and based on their findings, C++ provides the best performance (**Table 2.3.2i**). Since MongoDB is written in C++, it will perform better compared to other No-SQL databases written in other languages like Java.

Benchmark	Time [sec]	Factor
C++ Opt	23	1.0x
C++ Dbg	197	8.6x
Java 64-bit	134	5.8x
Java 32-bit	290	12.6x
Java 32-bit GC*	106	4.6x
Java 32-bit SPEC GC	89	3.7x

Table 2.3.2i C++ performance compared to Java

Following are a few MongoDB scripts to demonstrate the advantages of No-SQL on scalability. If you are familiar with JavaScript then you would notice that MongoDB documents (rows in SQL) are very similar to Javascript Objects widely known as JSON strings.

This is a JavaScript object: `var a = {age: 25};` and as demonstrated in **Figure 2.3.2i,** a document can be easily saved into a MongoDB collection without pre-specifying any schema or rules. The document saved simply consists of a set of key-value pairs in JSON format that resembles the traditional SQL table row.

```
db.atmosphere.save({
      temp: 45,
      latitude: 37.000371,
      longitude: -122.063279
})
```

Figure 2.3.2i Example of saving data into MongoDB

Now that there is a new document in the `atmosphere` collection we can use the following script to a new key-value pair to the document that will allow us to collect the humidity related to a temperature reading. This operation is done in real-time without disrupting the process of the application:
`db.atmosphere.update({temp: 45}, {$set:{humidity: 30}});`

[10] Programming language performance comparision.
https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf

Since MongoDB and No-SQL is a fairly new concept, our Python based web framework, Django, does not have current support for No-SQL databases therefore we were forced to use Mongoengine to connect our Django web app to our MongoDB database backend. Mongoengine is a document-object mapper that would allow Python applications like Django to work with MongoDB. It uses a seamless API that resembles the traditional Django ORM system therefore making it easy to integrate.

We designed four sets of collections to store various types of data. The four collection systems are: Features, OgreQueries, Coordinates, and APIKey.

The FeatureCollection holds all the data sent to the server by the GSF iOS app and also the open data retrieved by our open data retriever (**Table 2.3.2ii)**. Each Feature document includes an Embedded Document Field that is a reference to the Properties embedded document class (**Table 2.3.2iii**). The layout was designed to resemble a standard GeoJSON FeatureCollection (explained in section 2.4.2).

Key	Value-type	Description
type	String	Defaults to "Feature" to comply with GeoJSON formating
geometry*	GeoSpatial Point	GeoJSON formated point field dictionary that stores coordinates in the [longitude, latitude] format
properties*	Embedded Document	Reference field to the Properties class (**Table 2.3.2iii**)

* Required field for each document

Table 2.3.2ii The layout for the FeatureCollection

Key	Value-type	Description
source*	String	Specifies the source of the measured data, either from the iOS application or the open data source retrieved by OGRe
date added	DateTime	The copy of the "time" field converted from ISO string to standard Python datetime field for ease of sorting and lookup
time*	String	Standard ISO 8601 timestamp string
altitude	Double	The altitude of the measurement picked up by the iOS app
horizontal accuracy	Double	The horizontal accuracy of the GPS coordinates measured by the iOS app
vertical accuracy	Double	The vertical accuracy of the GPS coordinates measured by the iOS app
text	String	The text associated with a geotag, it can be a tweet from Twitter or an status update from Facebook

image	String	Base64 encoded string of the image captured by the iOS app or retrieved using the OGRe
noise_level	Double	The noise level of an area measured by the iOS app.
temperature	Double	The temperature of the location (iOS app or web crawler).
humidity	Double	The humidity of the location (iOS app or web crawler).
faces detected	Integer	The number of faces detected in an image stored in the database
bodies detected	Integer	The number of bodies detected in an image stored in the database
opencv flag	Boolean	The flag used to indicate whether the collection with an image has been processed through OpenCV for facial and body detection.

* Required field for each document

Table 2.3.2iii The Properties class layout

The OgreQueries collection is used to store any queries passed to the open data retriever so that our database cache builder script can retrieve more data in the background (**Table 2.3.2iv**).

Key	**Value-type**	**Description**
date added	DateTime	The time the query was entered into our web app
metadata	String	The HTTP POST request metadata including the user's IP address is stored for later assessment and limiting the numbe of queries used in the cacher system
sources	List of Strings	The sources field that gets passed to OGRe
media	List of Strings	The media field that gets passed to OGRe
keyword	String	The keyword field that gets passed to OGRe
location	List of Floats	The coordinates that get passed to OGRe

Table 2.3.2iv The layout for the OgreQueries collection

The Coordinates collection is used to store the coordinates that the site administrator want to deploy an agent to (**Table 2.3.2v**). Each document's unique ID in this collection is then sent to the agent where the iOS app uses it to retrieve the designated coordinates from our server. To have a uniform way of sending data across our applications, the Coordinates collection was designed to resemble a GeometryCollection in GeoJSON format.

Key	Value-type	Description
type	String	Defaults to “GeometryCollection”
geometries	List of GeoSpatial Points	The list of coordinates that the site admin has assigned to a agent
date added	DateTime	The time the coordinates were added to the database for later clean up

Table 2.3.2v The layout for the Coordinates collection

For the APIKey storage we had to use a traditional SQL database so that we could use the built-in admin section of the Django framework for managing and creating new API Keys. To do so we used the simple and ready to deploy SQLite3 database system that comes preinstalled with the Django framework. We created an schema for the APIKey table in the `api/models.py` section of the web app (**Table 2.3.2vi**).

Key	Value-type	Description
date added	DateTime	The time the API key was created
key	String	System generated API key
application	String	The third party application associated with the API key
organization	String	The organization that the third party application is associated with
full name	String	The developer or agent’s name who requested the API key
phone number	String	The cellphone number of agent associated with the API key
cell carrier	String	The service provider for the cell service to be used for the email-to-sms gateways
email	Email	The email of the developer or the agent associated with the API key
upload	Boolean	The upload flag that indicates the upload privilege for the API key (defaults to false)
download	Boolean	The flag that indicates the download privilege for the API key (defaults to true)

Table 2.3.2vi The layout for the APIKey table

2.3.3 REST API

To be able to send collected data from the iOS app to the server we designed a RESTful API. Our API allows our iOS app to send the data embedded in HTTP POST requests where each request is paired with an API key for authentication. To keep our project open-source and our data public, we decided to allow third party applications to be able to use our REST API to download data from our database.

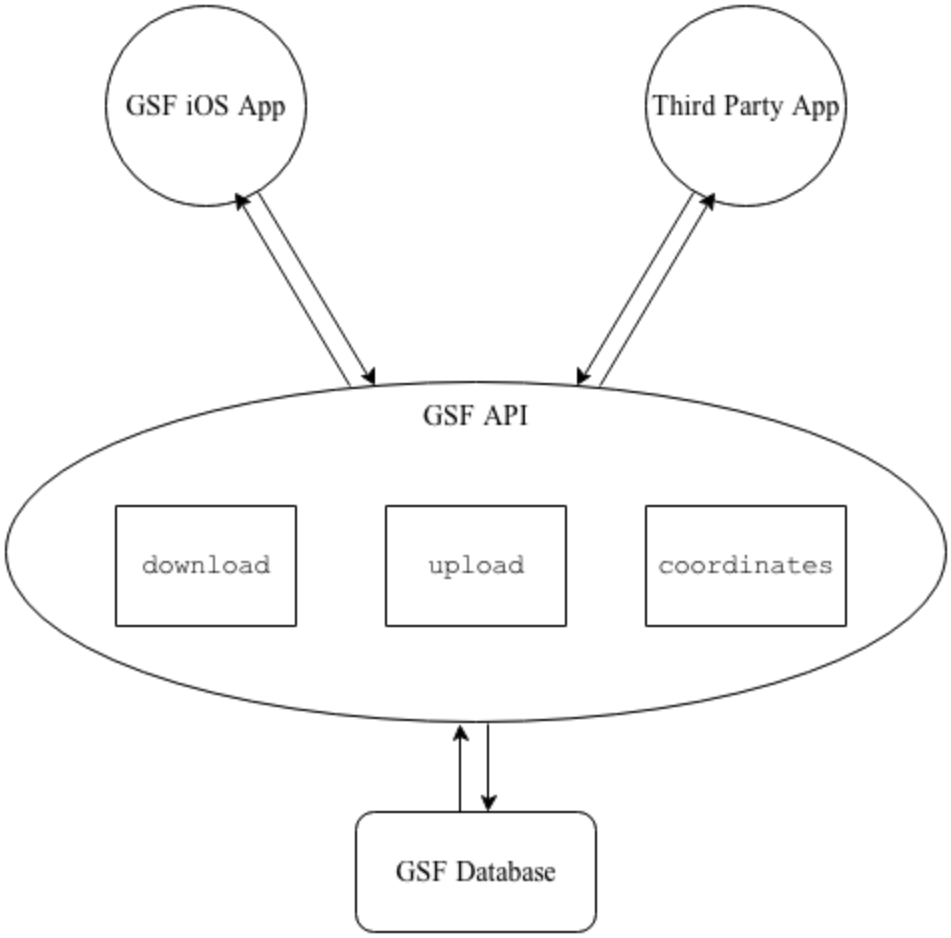


Figure 2.3.3i The GSF API structure

As you can see in **Figure 2.3.3i,** our API consists of three functionalities. Our `download` API allows third party applications to run raw MongoDB queries on our server and download the returned results through an HTTP response with the data embedded in the message body in GeoJSON format.

The `upload` API is currently only accessible by our iOS application. When an HTTP Post request is received from our iOS app, the body of the request is then searched for a GeoJSON formated FeatureCollection. The FeatureCollection is then parsed, validated and then saved into our database. If any validation fails or any errors occur during the process, an appropriate error message is then returned to the client that generated the request.

Our `coordinates` API is also another specific functionality for our iOS application. As discussed in section **2.2.7,** when an admin sends a set of coordinates to an agent's phone, the coordinates are actually stored in our Coordinates collection in the database. We only send the document id to the iOS app. The iOS app then sends the doc id as a GET request to our `coordinates` API where the doc id is then extracted and used to retrieve the set of coordinates stored in the database. The set of coordinates is then returned to the app in GeoJSON format and through a HTTP response message.

2.3.4 Authentication

To protect our server from any attacks and database corruptions we use API keys to authenticate any request coming into our REST API. We use two different variations of API keys. One set is specifically generated for our iOS app and the other is generated for third party developers.

If a developer is interested in querying our database on the backend, they would have to sign up to receive a unique API key for their application. A web form on the website (**Figure 2.3.4ii**) will collect information about the application and the developer's email. After the form is submitted, a unique `download` API key is generated (**Figure 2.3.4i**). The sequence to generate the key is as follow:

1. Generate a random 256 bit number using the Mersenne Twister Pseudo Random Number Generator (PRNG)
2. Hash the 256 bit number using SHA-256
3. Encode the hashed string using Base64 encoding
4. Flip random characters within the encoded string with a choice chosen randomly from a pool of characters

The result is a fairly complex and reasonable API key to handout to third party developers.

```
def generate_key():
    key = base64.b64encode(hashlib.sha256( \
              str(random.getrandbits(256)) ).digest(), \
              random.choice(['rA','aZ','gQ','hH','hG','aR','DD'])).rstrip('==')
    return key
```

Figure 2.3.4i The developer API key generation

The generated key is then emailed to the email address provided in the form. Upon receiving the API key the third-party application can then send query requests bundled with their unique key to get data from our database. Each application will have limited number of read requests per day.

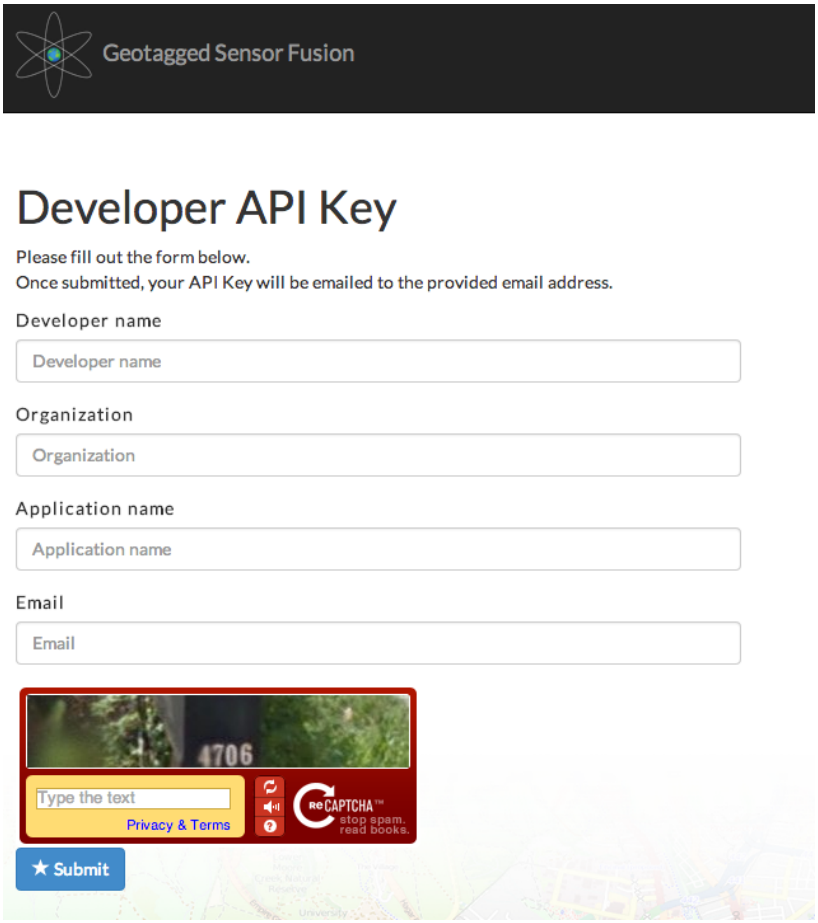


Figure 2.3.4ii The developer API key registration form

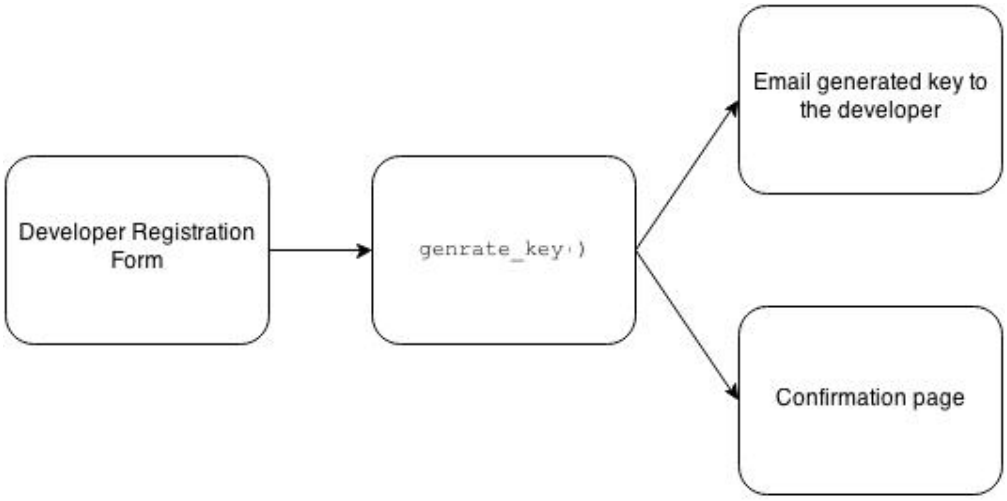


Figure 2.3.4iii The developer API key generation process

We had to develop a custom authentication system for our iOS application due to its upload privileges. In order to give the iOS app POST privileges, the web app admin would have to visit the admin panel and enter user's info with their phone number (**Figure 2.3.4iv**). After processing the given

information, the web app will generate a random 8 digit pin which the admin provides the user with. The user would enter the key into the iOS app only once. Once the app has the 8 digit access code, it can upload data to our database by bundling the data packages with their 8 digit key. We chose to only generate an 8 digit or less key for ease of input on the user side.

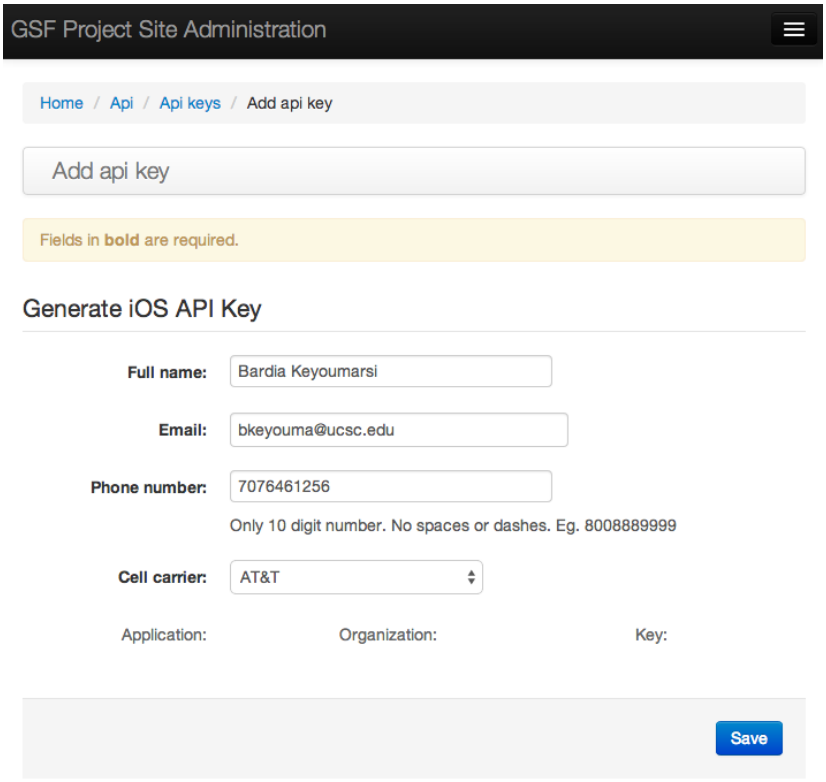


Figure 2.3.4iv The iOS API key generation form

	Full name	Organization	Email	Phone number	Cell carrier	Application	Key	Upload	Download
☐	Michael Baptist	LLNL	mbaptist@ucsc.edu	3107558917	Verizon	iPhone	30178196	✓	✓
☐	Agent 2	LLNL	sdp@gmail.com	1111111111	AT&T	iPhone	58200951	✓	✓
☐	Agent 1	LLNL	bkeyouma@ucsc.edu	8008009999	AT&T	iPhone	7261473	✓	✓
☐	Michael Bennett	LLNL	mjbennett31@gmail.com	9165410788	AT&T	iPhone	15057978	✓	✓
☐	Bardia Keyoumarsi	LLNL	bkeyouma@ucsc.edu	7076461256	AT&T	iPhone	93048877	✓	✓

Figure 2.3.4iv List of generated API keys viewed by the administrator

2.3.5 Fusion Search Interface

We created a unique query interface on our web application to allow users to run queries, fuse data together and visualize them on our map visualizer (section 2.5). Our Fusion interface consists of two

sections. Since we do map visualizations, any query that users run must return a set of geospatial coordinates. For this we created the Epicenters section of the UI (**Figure 2.3.5i**). The user has the ability to create random epicenters around the globe using the Twitter and Sensory Data sections under the Epicenters or do searches around specific locations in the world using the Locations drop down. After creating epicenters, the user has the ability to enter queries under the Aftershocks column. The Aftershocks are results found within a certain radius of each Epicenter.

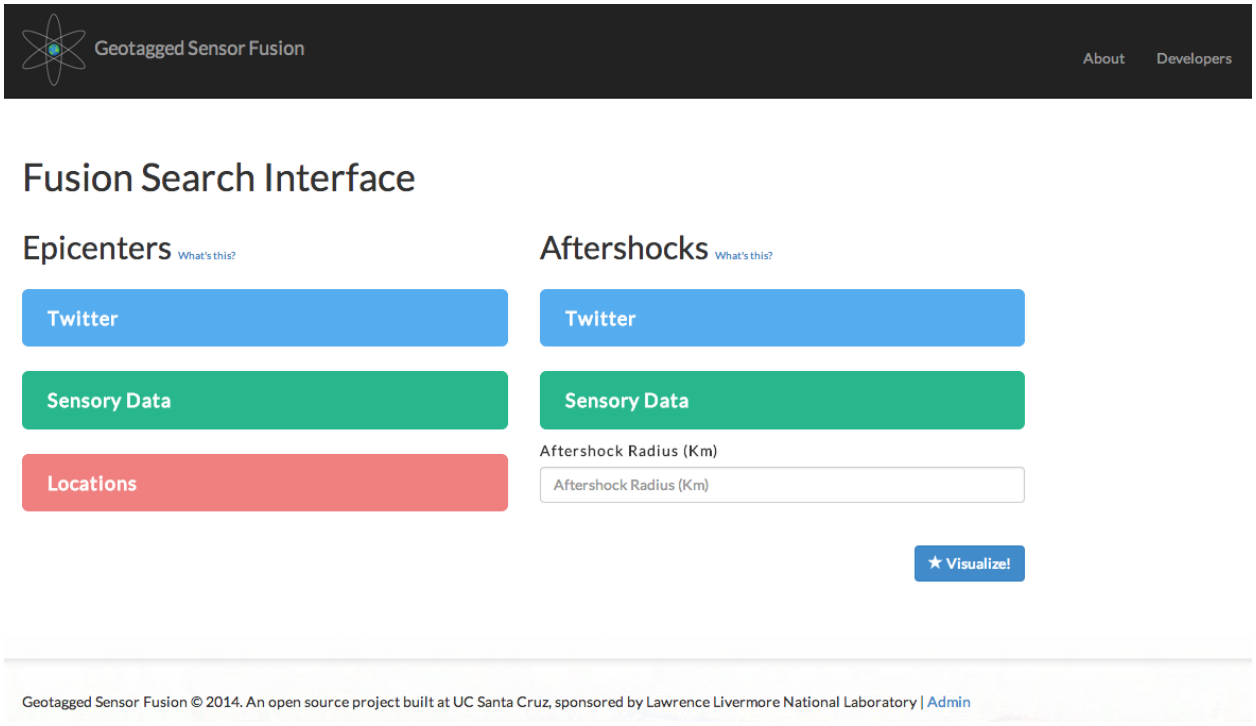


Figure 2.3.5i Fusion Search Interface

cached and sensory data • *temperature, population, tweets, etc.*
live third party data • *text, image, etc.*
sensory data fusion • Show population measurements within 1km of any place where the temperature was measure to be above 80°.
live third party data fusion • Show images posted to Twitter within 1km of any place where “Obama” is mentioned in Tweet.
sensory data & live third party data fusion

• Show images posted to Twitter within 1km of any place where the temperature was measure to be above 80°.

Table 2.3.5i Fusion query scenarios

We use custom JQuery functions to validate form inputs before they actually get sent to the backend for processing. One form of validation is to check whether the users have entered values for Epicenters if they are asking for Aftershock results (**Figure 2.3.5ii)**. This way we do not waste server resources for queries that will fail.

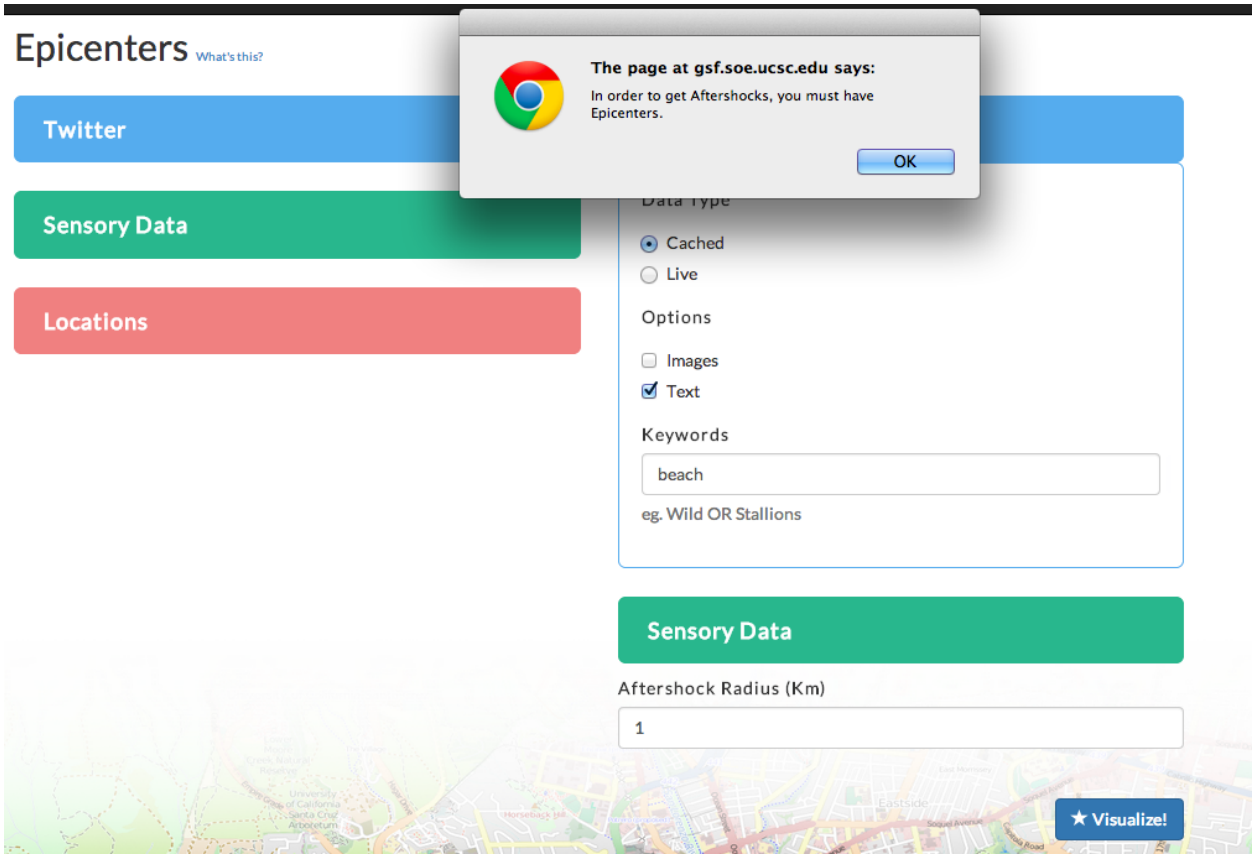


Figure 2.3.5ii Custom form validations

2.3.6 Cache Building System

After noticing the wait time to retrieve live data from third party sources, we built a caching system that uses stored queries entered by the users to retrieve and cache data in the background. The script was added to the Django project as a management command so that we can run it periodically as a cron job. The command to invoke the caching system is as follows: `python manage.py runcacher [number_of_queries]`.

The script accepts one argument which is the number of queries to pass to the OGRe. This allows us to max out our rate limit at times when server has no traffic and when the server is busy the number of queries can be reduced. It also creates a new instance of the retriever rather than importing an already existing one from the app.

2.3.7 Server-side OpenCV

To be able to get better population estimates using images retrieved from third party sources we added an OpenCV script to our web application. This script is also a management command so server administrators can invoke it periodically through a cron job. It takes one argument that is the number of images to retrieve from the database and run OpenCV on. The command to invoke the script is `python manage.py runopencv [number_of_images]`.

We could not simply pass the images retrieved from the database to the OpenCV script due to the fact that they are stored as Base64 encoded strings. Therefore, we had to take an extra step and decode the strings, write the image to a temporary file in the file system and then use the OpenCV library to load the image from the file. The next image would be overwriting the same temporary file in the filesystem so after the process is done only one temporary file is present on the file system.

2.4 Open Data Retriever

The retriever is responsible for gathering data from publicly-accessible sources such as social media, news outlets, or government databases. For our prototype, we decided to implement Twitter as a proof-of-concept source. Twitter offers 500 million new posts daily from 230+ million monthly-active users with about 77% originating outside the United States. It supports JSON natively and does not require a user context to retrieve data.

Many services mandate that developers retrieve data in the name of an existing user (hence, “user context”). Such a context only exposes information from other members that the primary user is associated with (e.g. friends, followers, etc.). The ones that don’t require a user context tend to use application-specific keys to identify the origin of a query and apply regulations such as rate limits. Twitter limits applications to 450 queries every 15 minutes which affected the architecture of both our web application and the retriever itself. Before discussing the applications of OGRe (the OpenFusion GIS Retriever), it should be noted that user contexts are not currently supported; however, implementing said functionality ought to be simple considering the fact that a user context only requires a different set of keys.

Our team took a number of steps to avoid denial of service due to rate limiting. Primarily, certain features were added to ensure the developer has fine-grained control over how fast the rate limit is approached. Secondarily, the way our web application employs this retriever is designed to utilize the maximum bandwidth of data retrieval from Twitter while simultaneously avoiding the imposed rate limit. We also investigated a number of methods that could be used to circumvent the rate limit, but, for the most part, Twitter would not approve of these solutions. The architecture, deployment, and potential of this retriever will now be explored in detail.

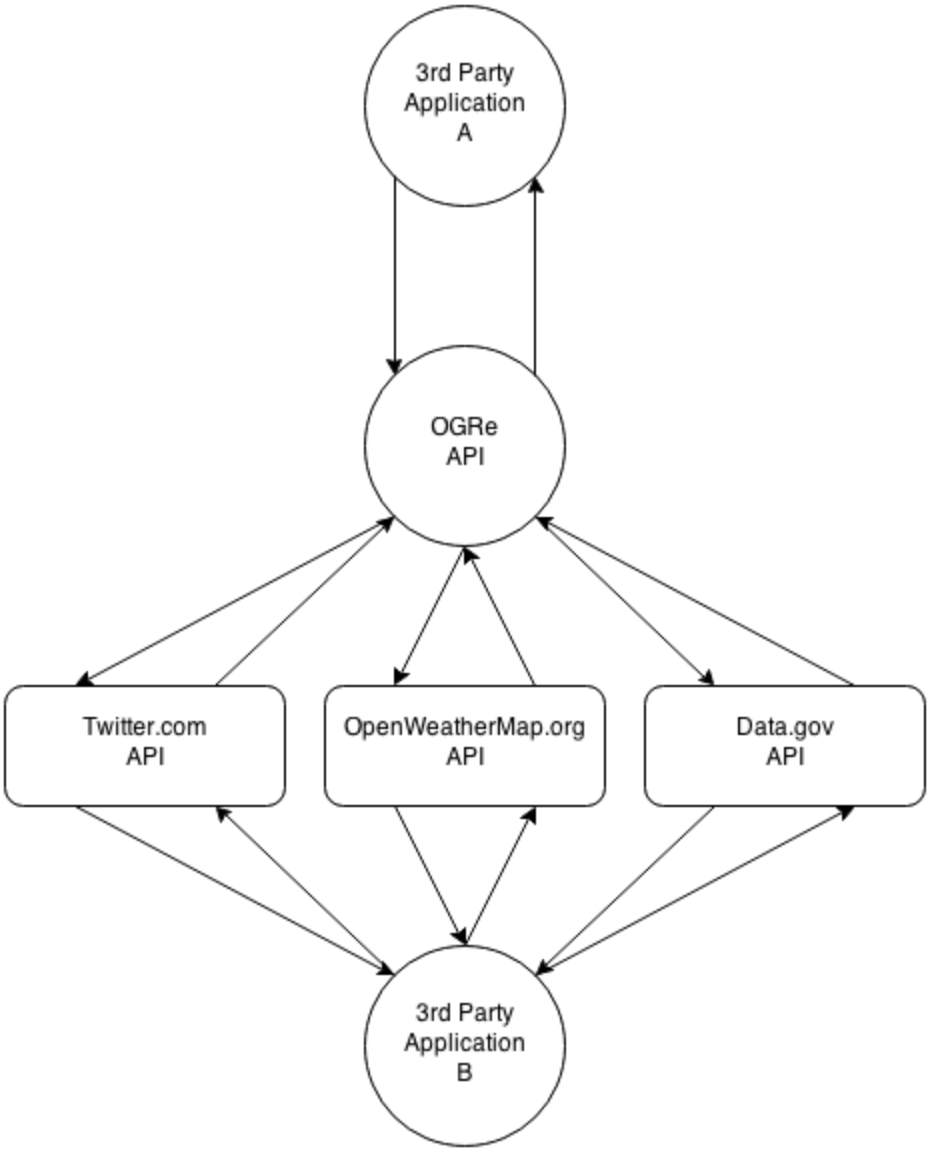


Figure 2.4i OpenFusion GIS Retriever Architecture

As shown in **Figure 2.4i,** the retriever is comprised of two distinct components: modules for each supported source (currently only Twitter.com) and a single unifying interface that acts as an intermediary between the sources and 3rd-party applications. When using the retriever, such applications have the option of using the main interface ("OGRe API" in the figure) or interacting with the sources directly. The recommended method of access is through the main interface for a few reasons. Most importantly, the it ensures that all data is returned in GeoJSON format (see 2.4.2). This contrasts with the alternate method of receiving data from a given source separately which provides GeoJSON Feature objects in a Python list structure. Also, a number of conditions that cause resulting data sets to be empty can be detected before they reach a source module if the main interface is used. This means faster delivery for applications that format and relay user-provided queries to the retriever. Another benefit of the primary OGRe API is that it can gather results from multiple sources in a single call which further reduces overhead. To do this, a caller must provide API keys for each source that will be queried. In order for the retriever to store these

keys, the main interface is implemented as a class that must be instantiated before any data is retrieved. After that, the instance may be used to get geotagged data indefinitely.

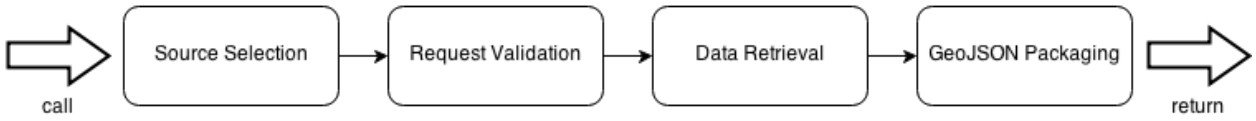


Figure 2.4ii Stages of a call to the OGRe API

The handling of a request sent to the retriever can be broken down into four stages, as shown in **Figure 2.4ii**. The first and the last stages are provided by the main OGRe API (a method called `fetch`) while the middle stages are delegated to each source module. After a caller initiates a request, a GeoJSON FeatureCollection object (see 2.4.2) is generated and each specified source module is invoked to retrieve data that will be added to it. However, before a query can be made on a given source, the parameters of the request must be validated. The validation, sanitation, and retrieval functions (including `fetch`) all share a common, generic interface consisting of the following elements: `media`, `keyword`, `quantity`, `location`, and `interval`. Note that, while these parameters define a common interface, each function is not limited to these parameters.

The `media` parameter can be used to identify specific types of digital mediums that should be included in the result set. Acceptable values include `image`, `sound`, `text`, and `video`; although, Twitter currently only supports text and image data. These values are specified as a list or tuple and default to all media if unspecified. If an empty list/tuple is passed for `media`, `fetch` will detect that no media should be included in the result set and immediately return a FeatureCollection that contains no Feature objects. No guarantee is made that only specified media will be returned. This is because of the way results are sent from Twitter which is predominantly text-based. There is no way to request a specific media type from Twitter, so the retriever adds a keyword that is very likely to return Tweets (that is, text) with images attached. This means that even if only images are requested, the text results associated with those images are included with the results, so OGRe includes it in the result set. If only the requested media types are desired, the boolean `strict_media` flag may be used to filter out all undesired types.

The `keyword` parameter allows a caller to search available tags, text, titles, captions, etc. for relevant words, phrases, or symbols. The use of this argument will vary by source, but many support embedded operators for omitting keywords, identifying contexts, or narrowing categories.[11] Some sources, including Twitter, require a `keyword` in nearly all searches.

The `quantity` parameter controls the size of the result set and, therefore, must be a positive integer. The retriever attempts to satisfy this on a best-effort basis. Whether the resulting GeoJSON object contains the specified number of Features depends on a number of other factors including data availability, rate limit status, and `query_limit`. The reason for these dependencies revolves around the fact that not all data included in a response from a given source is assumed to be geotagged. For example, the Twitter Search API does not allow applications to request geotagged data only, and it does not respond

[11] Twitter's scheme may be found at https://dev.twitter.com/docs/using-search.

with all available data all at once.[12] Instead, it uses a paging mechanism to throttle the rate at which Tweets may be extracted. Suppose 150 geotagged Tweets mentioning "sunshine" are requested. Twitter will send no more than 100 Tweets at a time (even though it is likely that more than 100 Tweets satisfying this request exist). The retriever inspects each Tweet for a geotag and discards any that do not have one. This means that, at most, 100 geotagged Tweets could be retrieved in a single page. OGRe then requests the next page of the same request's result set and repeats the previously described process. It continues doing this until one of the three following conditions are met: no more Tweets (pages) exist that meet the requested criteria, the rate limit is reached (in which case Twitter will refuse to send more data for, at most, fifteen minutes), or `query_limit` is reached. The `query_limit` parameter was specifically implemented to help application developers avoid hitting the rate limit associated with their API key. It works just like the `quantity` parameter, except it controls how many queries (that count against the rate limit) the retriever may make on the API it is fetching data from.

The `location` parameter allows callers to specify a geographic area to search for results. It is composed of four pieces of information: longitude, latitude, radius, and radius unit. Longitude and latitude make a coordinate that identify a location while radius (a positive integer) and unit prescribe an area that results ought to be contained within. Currently, kilometers (`km`) and miles (`mi`) are acceptable units as those are supported by Twitter; however, kilometers are recommended over miles due to the likelihood of more widespread support from sources that may be added to OGRe in the future.

Finally, the `interval` parameter details a period of time that constrains results temporally just as `location` constraints results spatially. It is made up of two boundaries, earliest and latest (POSIX timestamps), that establish a bounding window. When requesting data from Twitter, each timestamp is converted to a special format Twitter understands. The format is known as Snowflake, and Twitter uses it to uniquely identify every Tweet ever posted. Snowflake IDs may be generated by separate machines without the need for those machines to communicate with each other, and the time of creation is encoded within each ID.[13] The validation functions for this parameter allow the boundaries to be specified in any order because the lower value will always be assumed to mean the earliest boundary.

After the validation phase, data is retrieved from a given source. As previously mentioned, the common interface is not exclusive. More "runtime modifiers" may be accepted by a source and could potentially be specific to that source only. The Twitter module currently includes 4 runtime modifiers (in addition to `strict_media` and `query_limit`). Those are `fail_hard`, `secure`, `network`, and `api`, and they will now be discussed individually.

The `fail_hard` parameter causes an exception to be raised in scenarios that are not typically fatal. Such scenarios include rate limit detection (before a query is sent) and complex query detection (after a query is sent). These conditions raise a special, custom class of exceptions unique to OGRe that identify each scenario in a recoverable way. This is useful at the moment since Twitter is the only source module and the listed conditions imply an empty result set; however, the utility of this parameter will likely decrease over time.

The origin of the `secure` parameter stems from the requirement of SSL/TLS that Twitter imposes on users of its Search API. When images are requested, OGRe adds a specific keyword

[12] Even if Twitter wanted to (which would not be beneficial for its bandwidth), it is not always capable due to the vast number of new Tweets that are generated regularly. See https://dev.twitter.com/docs/working-with-timelines for more information.

[13] For more information, see https://blog.twitter.com/2010/announcing-snowflake.

("pic.twitter.com") in the query to Twitter. This works because all images attached to Tweets are added as shortened URLs to images hosted by Twitter. The retriever finds these links and makes a separate HTTP request to get them. By default, these separate requests are made using SSL/TLS too; however, this adds potentially unnecessary overhead that can be reclaimed by forcing the separate requests to be made without encryption (i.e. setting `secure` to `False`).

The `network` and `api` parameters are used to do dependency injections for testing. OGRe is a complete, production-quality package which means it ships with test modules designed to verify the functionality it intends to offer. In order to avoid a dependency on network availability or Twitter availability, Python MagicMock objects are typically passed to the source modules when testing. The MagicMocks replace the libraries that access the network and the source and accommodate detailed analysis after the test on how they were used. The `network` and `api` parameters are intended for use by OGRe developers not intended for average users or application developers.

The retriever is designed to be generically useful for any application that requires geotagged data. It can be easily installed from PyPI (https://pypi.python.org/pypi/OGRe) using either pip or setuptools, and, after installation, it can be accessed programatically in Python programs via import or directly on the command line by invoking a script that is automatically created and installed with the OGRe package.[14]

2.4.1 OGRe CLI

The retriever can be accessed via a command line interface from any shell capable of running Python simply by invoking the command `ogre`. Since it requires authentication to access data from any sources that may be specified (using the `-s`/`--sources` options), API keys may be specified using the `--keys` option. Alternatively, specific variables in the environment will be checked for credentials if the `--keys` option is omitted. The common interface shared by all functions in OGRe is accessible by name (e.g. `--media`, `--keyword`, `--quantity`, etc.) or more simply by the first letter of the parameter (e.g. `-m`, `-k`, `-q`, etc.). All currently supported runtime modifiers are also available from the command line; however, there is no short option for them. Finally, since the CLI is a front end for the OGRe back end, it controls how the retriever logs as it operates. A log level (e.g. DEBUG, INFO, WARN, etc.) can be specified using the `--keyword` option. The OGRe CLI is useful for gathering geotagged data and storing the results in files or piping them into other utilities that can handle GeoJSON data.[15]

2.4.2 GeoJSON

GeoJSON is an open standard format introduced in 2008 based on the widely-used JSON format. [16] It has proven to be indispensable to this project due to its specialization in handling geographic data. Data is classified as one of the following types of objects: Geometry, Feature, and FeatureCollection.[17]

A Geometry object is used to precisely identify a geographic point or region. For the most part, this project is only concerned with the following Geometries: Points and GeometryCollections. A Point

[14] More information on how to use OGRe can be found in the documentation at https://ogre.readthedocs.org/en/latest/.

[15] Over the course of the project, a series of tools were developed to simplify the handling and verification of the results produced by the retriever. These tools have been bundled and distributed as the GeoJSON ToolKit and are available at https://github.com/dmtucker/gjtk.

[16] The full specification may be found at http://geojson.org/.

[17] This is not an exhaustive list, but rather, a generalized enumeration of the most useful and relevant objects employed by most applications (including this one).

object consists of a type property and a coordinates property (see **Figure 2.4.2i**). A GeometryCollection object is a Geometry that can contain multiple Geometry objects which implies the possibility of an infinitely deep, recursive Geometry object (although it is not used like this for our purposes).

```
{
    type: 'Point',
    coordinates: [
        longitude,
        latitude,
        altitude[18]
    ]
}
```

Figure 2.4.2i GeoJSON Point Geometry object

A Feature object is a container for Geometries that provides a mechanism for associating arbitrary data with a location. It is composed of type, geometry, and properties attributes in the format depicted in **Figure 2.4.2ii**. Just as a GeometryCollection can be used to contain many Geometry object, a FeatureCollection can contain multiple Features. Most of our data is best represented as Features (potentially in a FeatureCollection) with Point Geometries and particular properties that will be discussed in section 2.5.

```
{
    type: 'Feature',
    geometry: <GeoJSON Geometry object>,
    properties: <arbitrary JSON object>
}
```

Figure 2.4.2ii GeoJSON Feature object

2.5 Data Visualizer

All gathered data must be presented in a sensible way to be useful. The visualizer, called Vizit, is capable of rendering any GeoJSON file; however, our web application is designed to produce files that follow a convention for demonstrating how data was fused. This is illustrated with an earthquake analogy. Fusion happens when a given set of points is mapped to another set or piece of data. This is similar to the way aftershocks, small quakes that occur near a larger earthquake, are associated with a particular epicenter. As shown in **Figure 2.5i**, we implement this idea with GeoJSON by use of a Feature property that contains a FeatureCollection of associated data. Features in the outermost FeatureCollection are considered epicenters (green), and Features found in the `related` property of an epicenter are considered aftershocks (blue).

```
{
```

[18] Note that the altitude element is optional.

```
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [
          <longitude>,
          <latitude>
        ]
      },
      "properties": {
        "time": <ISO 8601>,
        "text": <string>,
        "image": <Base64 JPEG>,
        "radius": <meters>,
        "related": {
          "type": "FeatureCollection",
          "features": [
            <GeoJSON Feature object>,
            ...
          ]
        }
      }
    },
    ...
  ]
}
```

Figure 2.5i GeoJSON Interface of Vizit

The architecture described above has the additional benefit of supporting an arbitrary depth of relation. This means that aftershock Features could themselves be considered epicenters if they too have a `related` property that specifies relevant aftershocks. Vizit supports this by recursively processing all Features the same way. The only limits on the depth of related data are imposed in a browser-dependent manner.

By default, epicenters and aftershocks are rendered differently by Vizit to visually distinguish fused data sets (see **Figure 2.5ii**). The white circles shown represent a radius that ideally contains all spatially-related data (but this is not strictly enforced). The default configuration also styles individual aftershocks differently according to the data associated with them. Points with affiliated images are displayed as red markers while all other types (currently only text) are rendered blue. This styling can be easily overwritten from within the GeoJSON file itself because Vizit supports dynamic styling for spatial

visualizations![19] The visualizer can show any data at a given location by selecting the marker at that location (as shown in **Figure 2.5iii**). Since images are embedded directly in the GeoJSON file being rendered,[20] the only significant load time a user should expect is the time it takes to retrieve the GeoJSON data.

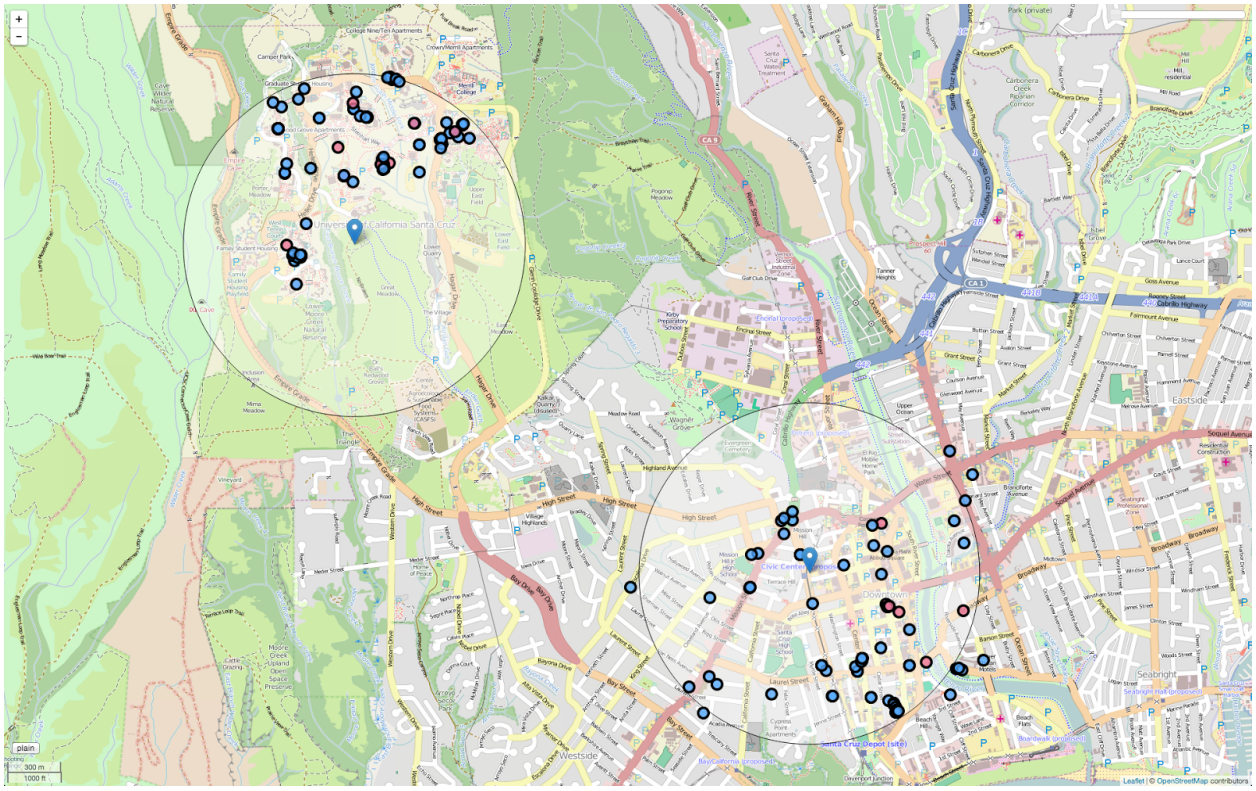


Figure 2.5ii Epicenters and Aftershocks

Vizit is written as a completely client-side JavaScript application, and it can be run in any modern browser with support for HTML and CSS. A GeoJSON file can be specified in the query string component of the URL as the `data` parameter (e.g. http://localhost/?data=example.geojson). The visualizer will attempt to retrieve http://localhost/data/example.geojson using AJAX before validating and displaying it.[21]

[19] Descriptions of the exact properties that the visualizer looks for to style a spatial visualization can be found at http://dmtucker.github.io/vizit/.

[20] This is achieved through the use of Base64 encoding.

[21] Vizit recognizes GeoJSON FeatureCollections that contain an additional `OpenFusion` property as having been produced by the web application. The property identifies the version of the interface being used and applies default styles accordingly.

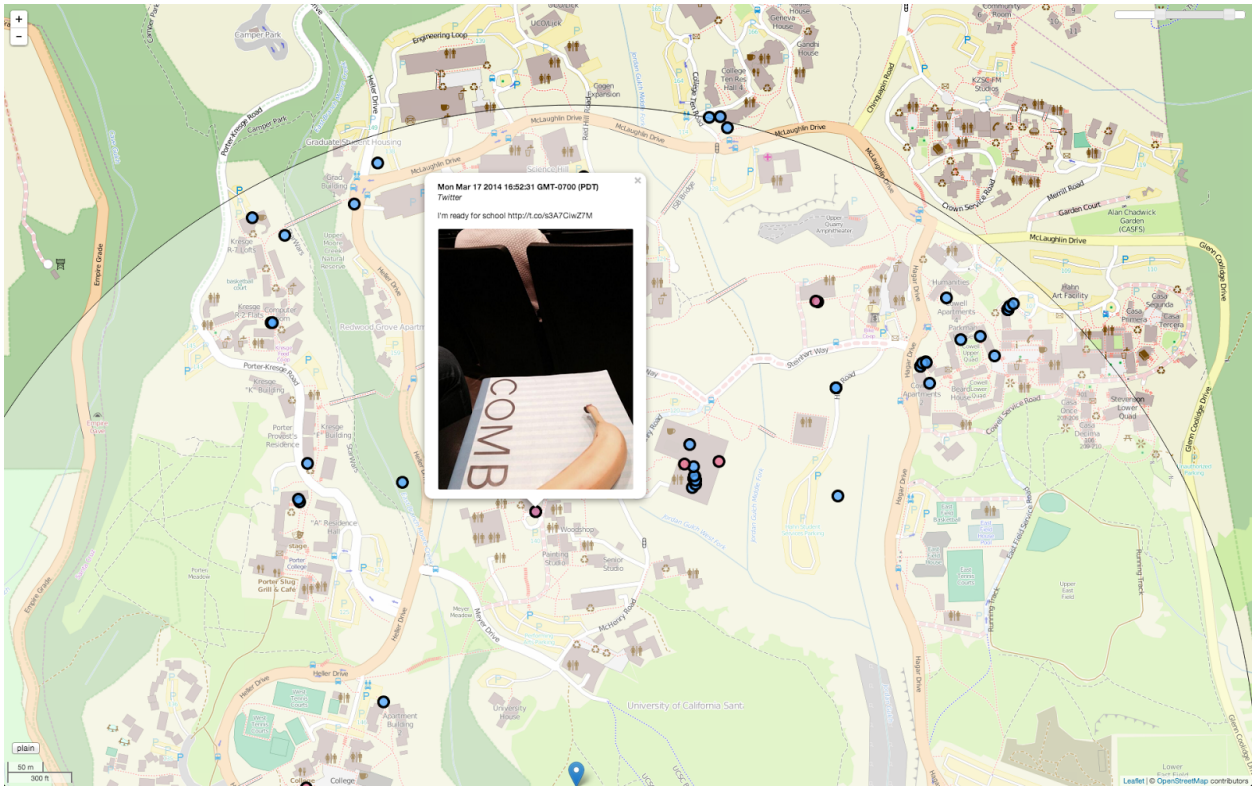


Figure 2.5iii Spatial/Temporal Visualization

Vizit can visualize data in a number of ways. Since this project focuses on geotagged data, perhaps the most relevant visualization is the map which shows how data is spatially related. The map is created using OpenStreetMap tiles to complement the theme of visualizing crowd-sourced data that is woven throughout the project.

The map can also constrain data temporally to show how relationships in the data. this is possible with the slider present in the upper right corner of the screen. Two toggles are present to allow users to specify a window of time that should be displayed. This feature relies on the data being timestamped with a `time` property in each Feature object. The spatial/temporal visualization is very useful for showing how data is related; however, it is not convenient for viewing the entire data set at once. So, a second type of visualization is available for this, and it is depicted in **Figure 2.5iv**. Switching between the two types is as simple as activating the toggle in the lower left portion of the screen. Alternatively, either visualization can be indicated in the query string of the URL by specifying "spatial" or "plain" for the `type` parameter (e.g. http://localhost/?type=spatial).[22]

[22] Note that the `data` parameter is required even though it is included in this example. It is omitted for brevity and clarity.

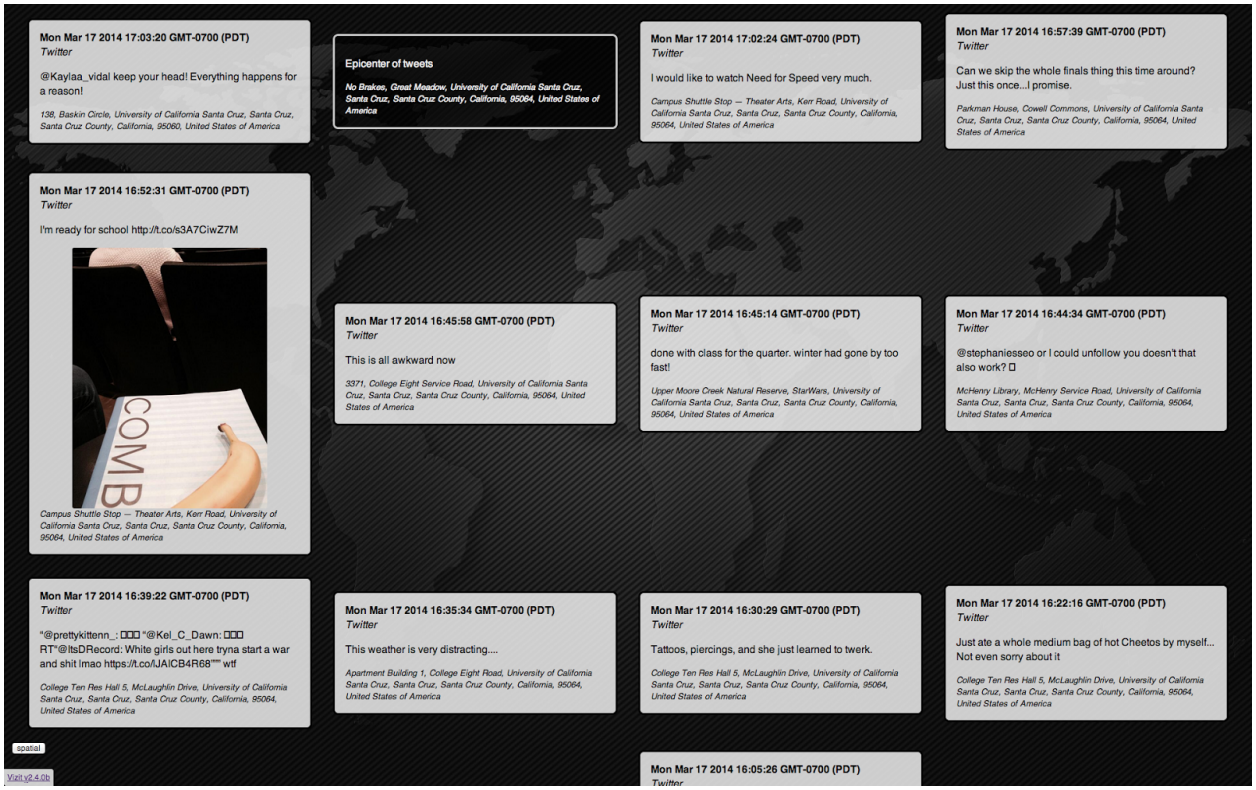


Figure 2.5iv Plain Visualization

Since this second type of visualization does plot points, it must represent the spatial data in some other simple manner. It does so by reverse geocoding the coordinates in the Point object of each Feature. It does so asynchronously when needed and caches the results to avoid undue burden on the server that provides this service.[23] A side effect of this process is that each Feature is not rendered until its Geometry has been resolved to a human-readable address (if possible).

3. Challenges

3.1 Using audio to power pluggable devices

Powering the pluggable sensor proved to be a greater challenge than expected. Because power is conserved in a transformer[3], stepping up the sine wave so that we get a high enough voltage at the output of the regulator also meant lowering the current that the regulator could provide. Originally, we were using a 1:20 transformer. However, the use of an ADC and TWI caused the prototype to draw more current than what we thought the prototype would draw based on the information we retrieved from datasheets. By temporarily powering the development board and the sensor with a power supply, we were able to determine how much current our prototype drew. To isolate the power problem, we removed

[23] Currently, the nominatim implementation on OpenStreetMap servers are used, but this may change in the future due to the maintainers' usage policy.

temporarily removed the regulator from our power circuit. The resulting circuit is shown below in **Figure 3.1i**:

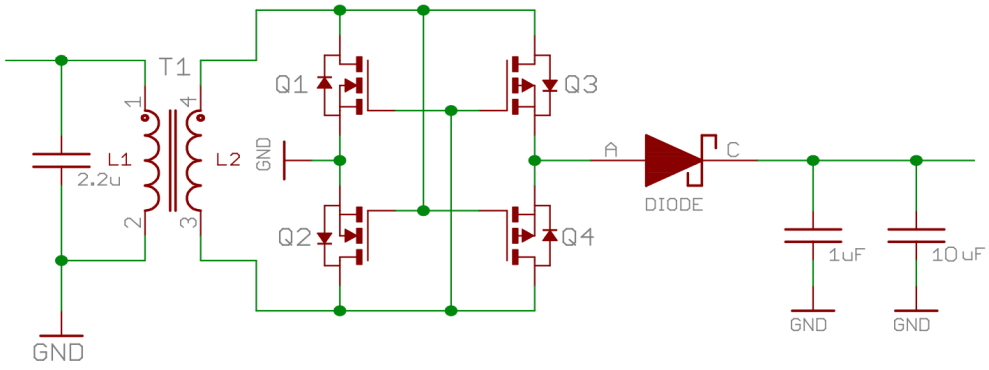


Figure 3.1i AC Step Up and Rectification

We played a 1.25Vpp, 20kHz tone on the iPhone as input to the above circuit. We then measured the output of the diode and found that the voltage out was 25VDC, and the short circuit current was 5.6mA. Assuming a power level of about 25VDC × 5.6mA implies that before step-up, the voltage level was 25VDC/20 = 1.25Vrms and the current level was 5.6mA × 20 = 112mA. There weren't many transformers that were small, cheap, and fit our audio step up application. We tried 1:3 transformers from Coilcraft, which ended up being extremely fragile. We finally were able to provide 3V at 9mA to the sensor and microcontroller by switching to a 1:10 transformer. This was enough for the iPhone to reliably power the pluggable sensor and read temperature and humidity data.

Thus, for a device that is powered only by a sine-wave tone, the microcontrollers sensors need to have both a very small current draw and the ability to operate at less than or equal to 3V. Sensors that are more power hungry would need external power.

3.2 Pluggable Sensor Communication Speed

The communication channel between the iOS device and pluggable sensor is rather slower than we'd like it to be, operating at about 50 bps. The max theoretical communication speed of a manchester encoded signal is 10 kbps, given a sample rate of about 44.1 kHz on the iOS devices ADC. The reason we couldn't reach anywhere near these speeds was due to the Audio Unit setup process. Despite setting up the `Audio Unit` based on the documentation provided by Apple there appears to be data being lost.

Our best guess as to where this is happening is during the `AudioUnitRender` function call. This function call takes as its argument a list of audio buffers to be filled by the hardware buffer. When we attempt to set up the `Audio Unit` to handle the size of more than one audio buffer only one buffer is still filled when sent to the the `AudioUnitRender` function. As mention in section 2.2.4, earlier in this document, this should be fixed by incorporating the `CAStreamBasicDescription` helper class once it has been updated for the latest iOS and no longer depends on deprecated function calls.

3.3 User Interface Design

Another challenge we quickly came to realize was user interface design. Every member of the team had a different opinion on how the application should look and how it should function. This made it quite difficult to make everyone pleased with the final design. There is still some debate on the final design we chose for version 1.0, however the team all agrees that it gives the user an intuitive experience with clean animations between views. The team all participated in design choices or by creating images that are used throughout the application to create a professional look and feel for the application. Overall we spent about a quarter of our efforts creating the user interface for the application and were proud to show it.

4. Results

We can now detect, power, and communicate with a custom pluggable sensor over a 3.5 mm audio jack. We can now provide the user with intuitive alerts and handle any audio interruption gracefully. Our iOS application can collect both onboard and external sensor data, and upload the collected data from the device to a web application via a custom, RESTful API. In addition the application is very expandable. It is written in such a way that simply adding more data to the data model to include any data the user feels is necessary to collect, future updates can be rolled out very simply by a developer to add more possibilities of data collection. Our web application can retrieve data from social media service and create meaningful location-based visualizations of fused data sets.

5. Cost Analysis

One of the goals of the project was to give anyone with a smartphone the ability to collect data. As such, we went with a mobile phone application to take advantage of the distributability of software. There is a great disconnect between the majority of the population and scientific sensor data. By using a mobile application, we attempt to break this social barrier by making it less of a commitment to collect sensor data. Moreover, we believe that most people would be more willing to collect sensor data if they didn't have to carry around a big sensor box. In short, most people have a smartphone, but not everyone has a $2000 sensor suite.

Aside from purchasing iPhones, we developed Pluggable PCBs. A large portion of the expenses actually came from 2-day or overnight shipping costs. This was done in the interest of time as the development of the hardware was on the critical path of our implementation phase. When parts ran out, we ordered more. Therefore, in the following estimation of the cost for 1 Pluggable Sensor PCB, we exclude shipping. The *overall* budget will be discussed later.

The cost for four boards (Revision 2) was $63.20 without shipping. Adding up the prices for the components on 1 PCB, we get about $30.54 including tax. Thus the price for one board is approximately $63.20/4 + $30.54 = $46.34 or less than $50. This doesn't take into consideration the fact that parts could be purchased in bulk to reduce costs.

Overall, we allocated about $2500 for the project which turned out to be a rather accurate estimation. The majority of our budget went toward purchasing two Apple iPhone 5S smart phones, and the remaining amount was mostly spent on circuit components and hardware fabrication. **Table 5i** illustrates a more clear breakdown of the usage of available funds. In the end, we were able to spend

about $200 less than we expected to carry out the project. A complete record of the budget for this account can be found in section 9.3 of the Appendix.

Item	Allocation	Allocated Weight	Amount	Actual Weight	Difference	Weight Difference
iPhones	$1419.82	56.59%	$1443.12	57.52%	($23.30)	(0.93%)
PCBs	$651.40	25.96%	$170.36	6.79%	$481.04	19.17%
Sensors	$249.35	9.94%	$91.34	3.64%	$158.01	6.30%
Circuit Components	$188.50	7.51%	$578.91	23.07%	($390.41)	(15.56%)
Remainder	$0	0.00%	$225.34	8.98%	($225.34)	(8.98%)

Table 5i Budget Analysis

6. Project Management

6.1 Team Members

At the start of the project we formed a team based on the skills and the work effects required to complete the project. The resulting team was composed of all members with great software skills, as it was the backbone to the project, but it also contained members with skills in hardware system design and PCB fabrication using CAD tools. As a team we voted that Bardia Keyoumarsi be the team lead, and that Michael Bennett be the vice lead. We distributed the project responsibilities as follows:

- Bardia Keyoumarsi (CE, CS Major) - Team Lead
- Michael Bennett (CE Major/EE, CS Minor) - Vice Lead
- Vincent Lantaca (CE, CS Major/EE Minor) - Part and Equipment Manager
- David Tucker (CE, CS Major) - Treasurer
- Michael Baptist (CE Major) - System Administrator

6.2 Project Roles

The work distribution for the project tasks are roughly outlined as follows:

- Bardia Keyoumarsi
    - Web application design and testing
    - Database development
    - Assisting with Fusion API and data visualization
- Michael Bennett
    - Writing iOS firmware for Pluggable Sensor power and two-way communication
    - Assisting with hardware and software for Pluggable Sensor development
    - Writing onboard iOS application sensor code

    - Assisting with front and back ends of iOS application
- Vincent Lantaca
    - Designing hardware for AC to DC power rectification
    - Writing Pluggable Sensor code for sensor collection
    - Creating PCB for Pluggable Sensor
    - Assisting with iOS firmware for Pluggable Sensor
- David Tucker
    - Developing and writing data Fusion APIs
    - Implementing data visualization framework
    - Assisting with the web application development
- Michael Baptist
    - Planning and implementing iOS User Interface
    - Writing onboard iOS application sensor code
    - Creating network link between iOS device and server
    - Writing TSP solver for route planning
    - Assisting with database development

6.3 Organization And Scheduling

Any formal documentation was added to (git) version control and uploaded to our shared Google Drive. Group documentation was updated after any major changes/developments were made. All source code contains header documentation and function documentation. All personal engineering notebooks have been maintained and kept up-to-date for other team members to peruse and reference. A lab binder has always be present in the lab includes data sheets, application notes, components, and misc. information relevant to the project. The datasheet binder has been maintained by the Parts and Equipment Manager.

Each team member was expected to work a minimum of 30 hours per week, but should not have exceeded 55 hours per week. We broke both bounds of these limits several times during duration of our project. Each team member was required to completed his task on schedule in accordance to the Gantt Chart so that the project remained on schedule. When a team member could not meet these expectations, they alerted the team and provided adequate justification so that a new deadline could be decided upon.

Weekly meetings were held in order to make group decisions, to discuss plans, successes, setbacks, and considerations. Additional team meetings were organized and scheduled as needed. The Team Lead was responsible for scheduling, and creating an agenda for each meeting. Every member should was well aware of the schedule on the agenda before each meeting. All members were required to attend weekly scheduled meetings unless prior arrangements were made. Members that could not be at the meetings at the specified time notified the group ahead of time, as soon as possible. Members who were tardy, or missed a meeting were responsible for acquiring information on all topics discussed during their absence. For any meetings that included external attendees such as professors and/or sponsors, all team members were expected to arrive at least fifteen minutes before the scheduled time.

6.4 Project Repositories

- Web application: https://github.com/bkeyoumarsi/open-fusion-webapp

- Open Data Retriever (OGRe): https://github.com/dmtucker/ogre
- Map Visualizer (Vizit): https://github.com/dmtucker/vizit
- iOS Application: https://github.com/mbaptist23/open-fusion-iOS
- Atmel Microcontroller: https://github.com/vlantaca/ATxmega128a4u-TWI

7. Errata

- Our iOS application is currently under review by the Apple App Store to be added in for consumer usage. In the case that it is accepted, our application will be available to the general public. This means that anyone will be able to collect data, however our servers need to continue running for these users to send collected data to us. Also we would need to provide a way for them to send coordinates to themselves for route planning rather than a system admin.

- The server-side OpenCV script only works on .jpg formated images. A few lines of code can be added to the script to account for other image types.

8. References

[1] Apple iOS Developer Library
https://developer.apple.com/library/ios/navigation/ accessed on June 8th 2014.

[2] “Audio Unit Hosting Fundamentals”,
https://developer.apple.com/library/ios/documentation/MusicAudio/Conceptual/AudioUnitHostingGuide_iOS/AudioUnitHostingFundamentals/AudioUnitHostingFundamentals.html, accessed on June 8th 2014

[3] Apple JSON API Documentation
https://developer.apple.com/library/ios/documentation/Foundation/Reference/NSJSONSerialization_Class/Reference/Reference.html accessed on June 8th 2014.

[4] Giancoli, Douglas C. *Physics for Scientists & Engineers with Modern Physics Volume II*: Pearson Education, Inc. 2008. Print. Page 771.

[5] Reverse Engineering the iPod Shuffle 3G headphone remote protocol
http://david.carne.ca/shuffle_hax/shuffle_remote.html, accessed January 20th, 2014.

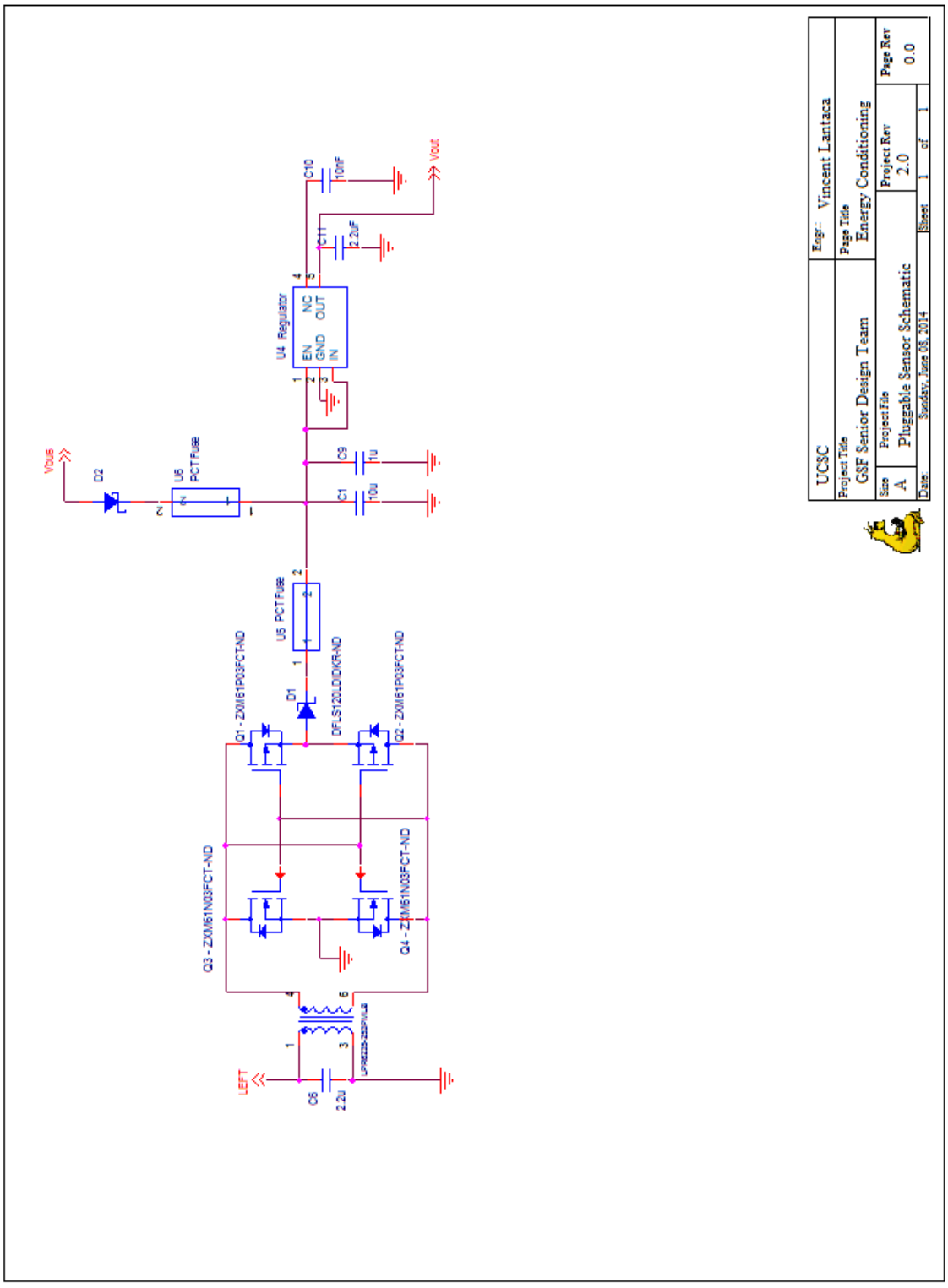
9. Appendix

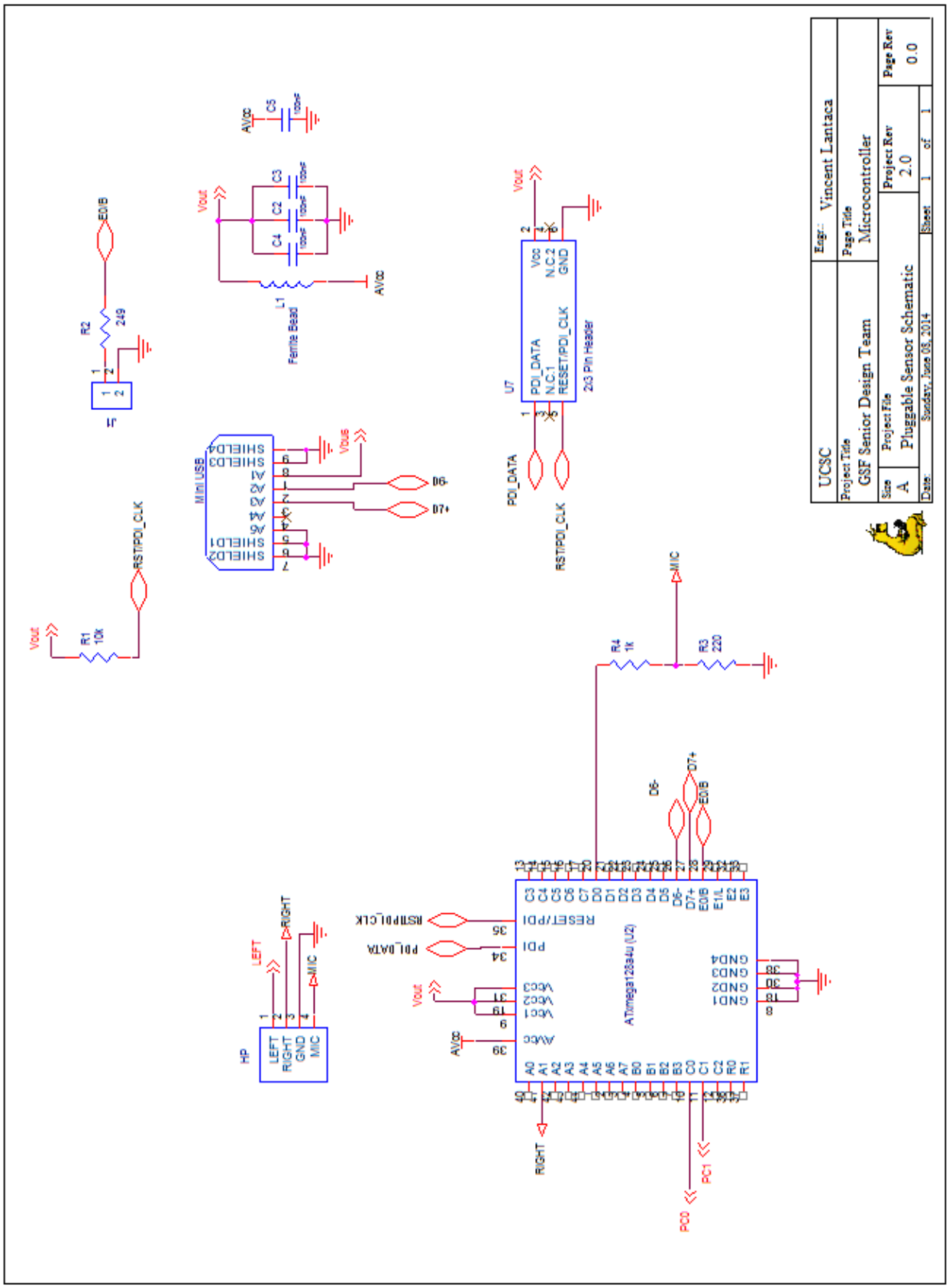
9.1 Pluggable Sensors Parts List

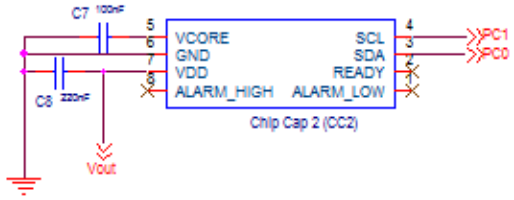
Item	Type	Description	Part #	Quantity Needed Per PCB	Price Per Unit ($)
1	Transformer	1:10 Transformer	LPR6235-253LMLB	1	1.06
2	Voltage Regulator	IC REG LDO 3.3V 0.15A SOT23-5	296-18476-1-ND	1	0.62
3	3.5mm Jack	3.5mm Jack 100 Megaohm min. @ 500VDC	171-7435-EX	1	2.6
4	Diode	DIODE SCHOTTKY 20V 1A POWERDI123	DFLS120LDIDKR-ND	2	0.49
5	PMOS	MOSFET P-CH 30V 1.1A SOT23-3	ZXM61P03FDKR-ND	2	0.49
6	NMOS	MOSFET N-CH 30V 1.4A SOT23-3	ZXM61N03FCT-ND	2	0.49
7	ATxmega Micro	IC MCU 8BIT 128KB FLASH 44TQFP	ATXMEGA128A4U-AU-ND	1	4.5
8	ChipCap Humidity/Temp	CHIPCAP2 DGTL 2% 3.3V	235-1337-ND	1	11.11
9	Ferrite Bead	FERRITE CHIP 220 OHM 3A 0805	445-1568-1-ND	1	0.12
10	Mini USB	CONN USB MINI B R/A SMD	ED2992CT-ND	1	0.87
11	Resistor 10k	RES 10K OHM .4W 1% 0805 SMD	RHM10.0KAECT-ND	1	0.16
12	Resistor 1k	RES 1K OHM 1/8W 1% 0805 SMD	RHM1.00KAHCT-ND	1	0.17
13	Resistor 220	RES 220 OHM 1/8W 1% 0805 SMD	P220CCT-ND	1	0.1
14	Resistor 249	RES 249 OHM 1/8W 1% 0805 SMD	P249CCT-ND	1	0.1
15	Capacitor 100nF	CAP CER 0.1UF 50V 10% X7R 0805	587-1279-1-ND	5	0.12
16	Capacitor 1uF	CAP CER 1UF 10V 10% X7R 0805	399-1172-2-ND	1	0.19
17	Capacitor 10uF	CAP CER 10UF 10V 10% X5R 0805	LMK212BJ106KG-T	1	0.2
18	Capacitor 220nF	CAP CER 0.22UF 50V 10% X5R 0805	587-3514-1-ND	1	0.3

19	Capacitor 2.2uF	CAP CER 2.2UF 10V 10% X7R 0805	587-1286-6-ND	2	0.16
20	Fuse 20V 40mA	THERMISTOR PTC 15 OHM SMD	490-3990-1-ND	2	1.03
21	Capacitor 10nF	CAP CER 10000PF 50V 10% X7R 0805	BC1293CT-ND	1	0.19

9.2 PCB Schematics




UCSC
Engr: Vincent Lantaca
Project Title
GSF Senior Design Team
Page Title
Microcontroller
Size
A
Project File
Pluggable Sensor Schematic
Project Rev
2.0
Page Rev
0.0
Date:
Sunday, June 08, 2014
Sheet
1
of
1
ATmega128a4u (U2)
Mini USB
2x3 Pin Header
HP
Ferrite Bead
RESET/PDI
PDI
PDI_DATA
RST/PDI_CLK
MIC
LEFT
RIGHT
GND
Vout
AVcc
D6-
D7+
E0/B
PC1
PC0



UCSC		Engr.: Vincent Lantaca	
Project Title: GSF Senior Design Team		Page Title: Sensor	
Size: A	Project File: Pluggable Sensor Schematic	Project Rev: 2.0	Page Rev: 0.0
Date: Sunday, June 08, 2014	Sheet 1 of 1		

9.3 Budget

Date	Item	Cost	Quantity	Subtotal	Tax	Shipping	Amount	Weight
1/1/2014	*iPhone 5*	*649*	*2*	*1298*	*116.82*	*5*	*1419.82*	56.59%
2/24/2014	16GB iPhone 5S Space Gray (GSM - unlocked)	-649	2	-1298	-113.58	0	-1411.58	56.26%
2/24/2014	*Apple*	*-649*	*2*	*-1298*	*-113.58*	*0*	*-1411.58*	56.26%
6/10/2014	EarPods	-29	1	-29	-2.54	0	-31.54	1.26%
6/10/2014	*Apple*	*-29*	*1*	*-29*	*-2.54*	*0*	*-31.54*	1.26%
1/1/2014	*3.5mm Jacks*	*5*	*10*	*50*	*4.5*	*5*	*59.5*	2.37%
4/28/2014	3.5mm Jacks	-2.6	6	-15.6	0	0	-15.6	0.62%
4/28/2014	*Mouser*	*-2.6*	*6*	*-15.6*	*0*	*0*	*-15.6*	0.62%
1/1/2014	*2-pack double layer PCB*	*92*	*5*	*460*	*41.4*	*150*	*651.4*	25.96%
5/14/2014	PCB	-22.58	2	-45.16	0	-31	-76.16	3.04%
5/14/2014	*Alberta Printed Circuits*	*-22.58*	*2*	*-45.16*	*0*	*-31*	*-76.16*	3.04%
5/27/2014	PCB v2	-63.2	1	-63.2	0	-31	-94.2	3.75%
5/27/2014	*Alberta Printed Circuits*	*-63.2*	*1*	*-63.2*	*0*	*-31*	*-94.2*	3.75%
1/1/2014	*Temperature Sensor*	*10*	*5*	*50*	*4.5*	*5*	*59.5*	2.37%
1/1/2014	*Humidity Sensor*	*30*	*5*	*150*	*13.5*	*5*	*168.5*	6.72%
1/1/2014	*Light Sensor*	*3*	*5*	*15*	*1.35*	*5*	*21.35*	0.85%
4/7/2014	CHIPCAP2 SIP DGTL SLP 2% 3.3V	-15.21	3	-45.63	-3.992625	-2.73	-52.352625	2.09%
4/7/2014	CHIPCAP2 DGTL 2% 3.3V	-11.11	3	-33.33	-2.916375	-2.74	-38.986375	1.55%
4/7/2014	*Digi-Key*	*-26.32*	*6*	*-78.96*	*-6.909*	*-5.47*	*-91.339*	3.64%
1/1/2014	*Circuit Board Components (Per Board)*	*20*	*5*	*100*	*9*	*20*	*129*	5.14%
2/19/2014	2.2uF Ceramic Capacitor	-0.09	10	-0.9	-0.07875	-2.731	-3.70975	0.15%
2/19/2014	3.3V/10mA Regulator	-2.151	10	-21.51	-1.882125	-2.731	-26.123125	1.04%
2/19/2014	20V/1A Schottky Diode	-1.112	10	-11.12	-0.973	-2.731	-14.824	0.59%
2/19/2014	1uF Ceramic Capacitor	-0.046	10	-0.46	-0.04025	-2.731	-3.23125	0.13%
2/19/2014	10uF Ceramic Capacitor	-0.14	20	-2.8	-0.245	-2.731	-5.776	0.23%
2/19/2014	100uF Ceramic Capacitor	-1.38	5	-6.9	-0.60375	-2.731	-10.23475	0.41%
2/19/2014	54mW Green LED	-0.54	5	-2.7	-0.23625	-2.731	-5.66725	0.23%
2/19/2014	1.47kO Resistor	-0.43	5	-2.15	-0.188125	-2.731	-5.069125	0.20%
2/19/2014	30V/1.1A P-type MOSFET	-1.194	10	-11.94	-1.04475	-2.731	-15.71575	0.63%
2/19/2014	30V/1.4A N-type MOSFET	-0.494	10	-4.94	-0.43225	-2.731	-8.10325	0.32%
2/19/2014	*Digi-Key*	*-7.577*	*95*	*-65.42*	*-5.72425*	*-27.31*	*-98.45425*	3.92%
2/23/2014	ProtoBoard - Bandicoot (SMD)	-9.95	1	-9.95	0	-15.09	-25.04	1.00%
2/23/2014	*SparkFun*	*-9.95*	*1*	*-9.95*	*0*	*-15.09*	*-25.04*	1.00%
3/6/2014	MT-DB-X4 Atmel AVR XMEGA development board	-21.99	1	-21.99	0	-6.99	-28.98	1.16%
3/6/2014	*MattairTech*	*-21.99*	*1*	*-21.99*	*0*	*-6.99*	*-28.98*	1.16%
3/6/2014	IC REG LDO 3.3V/0.15A	-0.62	5	-3.1	-0.27125	-4	-7.37125	0.29%
3/6/2014	LED 630nm Red	-0.46	5	-2.3	-0.20125	-4	-6.50125	0.26%
3/6/2014	Mini-USB SMD	-0.87	5	-4.35	-0.380625	-4	-8.730625	0.35%
3/6/2014	CHIPCAP 2% 3.3V	-10.886	5	-54.43	-4.762625	-4	-63.192625	2.52%
3/6/2014	*Digi-Key*	*-12.836*	*20*	*-64.18*	*-5.61575*	*-16*	*-85.79575*	3.42%
4/7/2014	MT-DB-X4 Atmel AVR XMEGA development board	-21.99	2	-43.98	0	-6.99	-50.97	2.03%
4/7/2014	*MattairTech*	*-21.99*	*2*	*-43.98*	*0*	*-6.99*	*-50.97*	2.03%
4/29/2014	0.22uF Ceramic Capacitor	-0.202	10	-2.02	-0.17675	-0.425	-2.62175	0.10%
4/29/2014	0.1uF Ceramic Capacitor	-0.084	30	-2.52	-0.2205	-0.425	-3.1655	0.13%
4/29/2014	249O Resistor	-0.1	10	-1	-0.0875	-0.425	-1.5125	0.06%
4/29/2014	220O Resistor	-0.1	10	-1	-0.0875	-0.425	-1.5125	0.06%
4/29/2014	1kO Resistor	-0.145	10	-1.45	-0.126875	-0.425	-2.001875	0.08%
4/29/2014	10kO Resistor	-0.133	10	-1.33	-0.116375	-0.425	-1.871375	0.07%
4/29/2014	Switch	-0.339	10	-3.39	-0.296625	-0.425	-4.111625	0.16%
4/29/2014	220O Ferrite Chip	-0.101	10	-1.01	-0.088375	-0.425	-1.523375	0.06%
4/29/2014	*Digi-Key*	*-1.204*	*100*	*-13.72*	*-1.2005*	*-3.4*	*-18.3205*	0.73%
5/12/2014	Power Inductor	-1.03	5	-5.15	0	-24	-29.15	1.16%
5/12/2014	*Coilcraft*	*-1.03*	*5*	*-5.15*	*0*	*-24*	*-29.15*	1.16%
5/14/2014	2.2uF Ceramic Capacitor	-0.11	10	-1.1	-0.09625	-0.515	-1.71125	0.07%
5/14/2014	N-type MOSFET	-0.67	5	-3.35	-0.293125	-0.515	-4.158125	0.17%
5/14/2014	20V 1A Diode	-2.24	4	-8.96	-0.784	-0.515	-10.259	0.41%
5/14/2014	Thermistor	-0.853	10	-8.53	-0.746375	-0.515	-9.791375	0.39%
5/14/2014	3.3V Regulator	-0.62	5	-3.1	-0.27125	-0.515	-3.88625	0.15%
5/14/2014	10000PF 50V Ceramic Capacitor	-0.19	8	-1.52	-0.133	-0.515	-2.168	0.09%
5/14/2014	0.1uF Ceramic Capacitor	-0.084	30	-2.52	-0.2205	-0.515	-3.2555	0.13%
5/14/2014	8-bit MCU	-5.21	3	-15.63	-1.367625	-0.515	-17.512625	0.70%
5/14/2014	*Digi-Key*	*-9.977*	*75*	*-44.71*	*-3.912125*	*-4.12*	*-52.742125*	2.10%
5/20/2014	In-System Programmer	-37.5	1	-37.5	-3.28125	-98.62	-139.40125	5.56%
5/20/2014	*Digi-Key*	*-37.5*	*1*	*-37.5*	*-3.28125*	*-98.62*	*-139.40125*	5.56%
5/25/2014	0.22O DMOS	-0.0909	6	-0.5454	0	-16.5	-17.0454	0.68%
5/25/2014	0.35O DMOS	-0.0909	10	-0.909	0	-16.5	-17.409	0.69%
5/25/2014	*Future Electronics*	*-0.1818*	*16*	*-1.4544*	*0*	*-33*	*-34.4544*	1.37%
	Spent		334	-1837.9744	-142.762875	-302.99	-2283.727275	91.02%
	Allocated		37	2123	191.07	195	2509.07	100.00%
	Remaining		-	285.0256	48.307125	-107.99	225.342725	8.98%