**Final Lab Report**

**Navigation and Emergency Device for Outdoor Recreation**

**EE 459: Embedded Systems Design Laboratory**

**Team FitBit**

**Jonathan Rodriguez, Christian Wagner, Ryan Louie**

**Table of contents:**

* **Introduction: 2**
* **Project Requirements: 2**
* **Development Method: 2 - 3**
* **Implementation: 3 - 14**
  + **Communication Protocols: 3 - 4**
  + **Pin layout: 4 - 5**
  + **Component implementation: 5 - 11** 
    - **GPS Sensor: 5 - 6**
    - **RFM69 Transceiver: 6 - 7**
    - **Monochrome 128x64 OLED Screen: 7 - 8**
    - **Heart Rate Monitor: 8 - 9**
    - **LSM9DS1 - Accelerometer: 10**
    - **LSM9DS1 - Temperature: 10**
    - **LSM9DS1 - Magnetometer: 10 - 11**
* **Putting it all together: 11 - 13**
* **Board Functionality: 13**
* **Project Challenges: 14**
* **Cost Analysis: 14 - 15**
* **Conclusion: 15 - 16**

Introduction

The goal of this project is to develop a proof of concept for a waterproof device that can track a friend relative to the user’s position. As this device is intended to be used underwater, there wouldn’t necessarily be any cellular signal that can connect to it. As a result, our group developed the Aqua-Link band. The Aqua-Link band is a device that can communicate with other bands via wireless RF communication protocols. In addition, a separate hub is included to keep track of all Aqua-Link devices. Keep in mind that there is no master-slave form of communication between the hub and the bands. This is especially useful in case the hub gets lost. If the hub user gets lost band users can still communicate with one another. However, each Aqual-Link band can only track one other device at a time. This contrasts with the hub, which keeps track of the position of every connected device.

Project Requirements

Before proceeding with project development, there are certain requirements defined earlier that need to be achieved. These requirements are basic requisites needed for an outdoor device. First, the device shouldn’t be dependent on wireless networks. Because these devices are intended to be used outdoors, the absence of cellular networks is expected due to reasons such as remoteness of location, landscape interference and more. As discussed earlier, these devices should also be able to communicate with other devices through RF communication. In addition, our device should be practical and easy to use for someone with little technical background. Finally, our product should be secure in that it does not suffer from interference from other similar devices. These are the basic requirements for our device, though we include additional features in order to differentiate our device from the competition.

## Development Method

Our developmental method is broken up into the following parts: planning, implementation, and testing.

### Planning

### When planning out our device we first identified the key functionalities we wished to incorporate. These functionalities are as follows:

|  |  |  |
| --- | --- | --- |
| * GPS Locating | * Travel Heading | * Display |
| * Step Counting | * RF Communication | * Heart Rate Tracking |

In order to incorporate these functionalities into our device, we decided we needed the following sensors and modules.

|  |  |  |
| --- | --- | --- |
| * GPS Module | * RF Transceiver | * Heart Rate Sensor |
| * Accelerometer | * LED Display | * Magnetometer |

To enable tracking capabilities we needed the following three devices: GPS, RF Transceiver, Magnetometer. The GPS module is a critical aspect of the whole project, as it is needed to provide location and tracking capabilities. Second to the GPS in importance, is a means of wireless communication between the devices so they can exchange GPS data for tracking purposes. We decided to use Radio Frequency “RF” communication, as it has a longer range than bluetooth. The final piece of data needed to provide full tracking capabilities between the devices is a compass heading for each device. To achieve this we decided to use a magnetometer as an electronic compass, enabling our device to provide the relative direction from each other to improve tracking capabilities.

The other side to our device is a fitness tracking which is accomplish through the accelerometer and heart rate sensor. For the heart rate sensor, we decided to use an optical heart rate sensor that affixes to the users finger as an analog for the optical heart rate sensors used in wearable bands like the Fitbit HR. As for tracking the user's steps, we decided to implement a pedometer using an accelerometer.

Lastly, we decided to include a LED display to allow the user the ability of seeing the data gathered by our device. Data includes the following: heading, GPS coordinates, steps taken, and heart rate. Therefore, we needed a display with fine enough resolution to allow characters to be displayed. See Table 2 for a complete parts list.

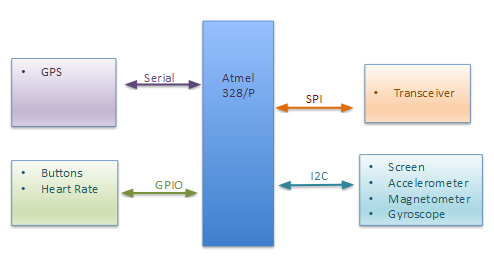
The final step of our planning was determining how best to test all the functionality of our board. For testing fitness tracking capabilities, it was feasible to build only one device. However, a single device would not have been appropriate for testing GPS tracking and RF communication, both of which are essential to our devices overall goal. As such, we decided to build two nearly identical devices to allow for testing of GPS tracking and parallel development of the fitness attributes. Parallel in the sense that multiple engineers are able to work on the device at the same time.

## Implementation

Our approach for developing our device revolved around the implementation of each sensor and module independently with our microcontroller, before incorporating it into the device as a whole. As such we determined to make independent libraries of code with test files for everything we included into our overall device. This method allowed for faster development time, as we could debug each component of the device independently, thereby allowing us to isolate any issues that arose throughout the project to a specific component.

### Communication Protocol

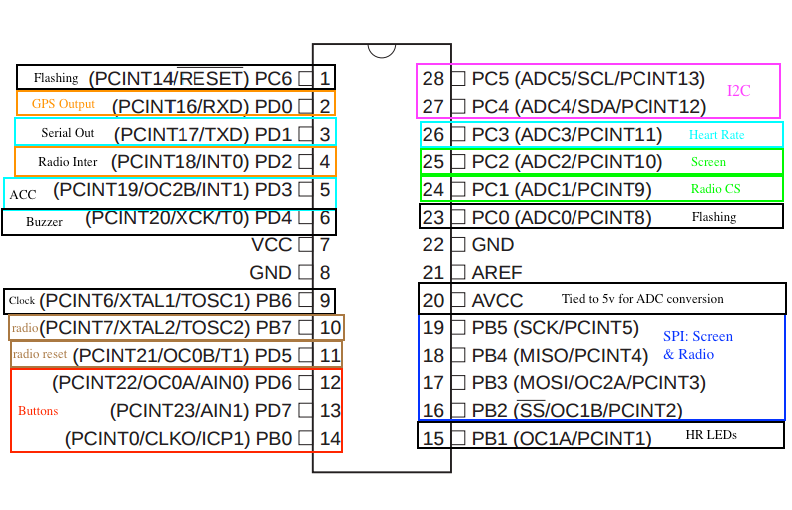
When choosing our devices we took into account the communication protocols that they used. With the devices we chose we ended up with the following communication protocols: I2C, Serial, and SPI (see Figure 1). While it might seem advantageous to use a single communication protocol, this was not the case. The Atmega328p microcontroller only has a single TX and RX pin for serial communication, as such we wanted to avoid having multiple devices using serial communication to prevent having to implement a purely software serial link. This led to us choosing a RF transceiver that uses SPI for communication. As for I2C, we chose to use it for the rest of our components as it uses fewer pins than SPI with its dedicated slave select lines. See Figure 1 for a block diagram demonstrating which protocol each component uses.

Since we are using three different communication protocols, we decided to develop independent communication code libraries to allow for easier implementation and testing. Within each of these libraries there is code for initializing the protocol, as well as code for sending and receiving data, in addition to other features appropriate for each protocol.

**Figure 1: Basic block diagram of our communication system**

### Pin Layout: Connecting all Interfaces

Figure 1 presents a basic block diagram, showing how the microcontroller will communicate with each device, from a communication protocol perspective.

As shown above, the GPS sensor outputs data through the serial communication interface thereby utilizing the RX ports for data input. We used the TX pin for the serial monitor on our computer terminal for debugging purposes. The radio transceiver communicates through SPI, thereby using the Atmega328p SPI pins in addition to a chip select pin and reset pin. The accelerometer and the screen use I2C pins on the microcontroller in addition to the screen also having a reset pin. For the heart rate monitor we used an ADC pin, as we needed to convert the raw analog output from the device. Note the heart rate monitor was only implemented on the larger device and not the smaller second device. The remainder of the pins were used for various buttons and LEDs. 

**Figure 2: Pin layout of the Atmega328p**

### Component implementation

For implementing individual components we created a library for each individual component used in our device. Each library contains a variety of support functions, such as initialization and reading, as well as a header file. The header file allows for easy integration with other code as well as our test file. The test file is designed to demonstrate the basic top level usage of the library and output relevant information pertaining to its functionality. For example, the GPS test file outputs latitude and longitude coordinates.

#### GPS Sensor

In terms of keeping track of geographic location, perhaps one of the most important components is the GPS sensor. For our project we decided to use the Adafruit Ultimate Breakout GPS sensor, primarily due to its accuracy when compared against rival GPS sensors. While this was one of the most accurate GPS sensors available, it also came with its challenges, primarily with connection and efficient code development.

Initially, we planned on using the components internal antenna for the device, however this proved inadequate. With the built-in antenna, it would often take the GPS upwards of 10 minutes to acquire a satellite fix while outdoors. If we moved the device indoors,, the GPS component simply couldn’t get a connection. As such, to obtain accurate readings, we were forced to order a separate GPS antenna. With the external antenna we would generally be able to get a satellite fix under a minute.

After addressing this connection issue, we were able to easily obtain GPS information without further configuration of the component. Therefore, the last step in the GPS implementation was devising efficient code for parsing the GPS data into something useful. The GPS device spits out a stream of NMEA data in its different formats. As our device doesn’t need all the information, we choose to only utilize the “GPGGA” format which had one of the fastest update rates and all the information we needed: time, latitude, longitude, altitude, and number of satellites.

Once the data was parsed and stored into variables, we are able to display the current latitude and longitude of the device. From this information we were also able to calculate the distance the individual device travelled while on as well as the distance between the devices when tracking each other. For calculating the distance we used the Haversine distance formula. We perform this calculation once to find peer to peer distance, and we perform it again with previous coordinates to calculate distance travelled.

#### RFM69 Transceiver

In order to allow for peer to peer tracking we needed to develop a way of wirelessly transmitting GPS coordinates between devices. As such, we decided use RF communication. We chose to use the RFM69HCW LoRa transceiver as it had a max range of 500 meters and there was plenty of online documentation for the device. The RFM69HCW also uses SPI for communication with the microcontroller which fit nicely into our overall design.

Implementing the component to send and receive radio communications was a tricky endeavour for the following reasons: SPI issues, large number of configuration registers, and antenna issues. To begin working with the component, we first had to establish SPI communication with it. To do so we had to configure the SPI protocol on the microcontroller for the RFM69 as specified in the datasheet, a step that we originally missed as the RFM69 does not use the default SPI communication configuration.

The RFM69HCW is by far the most complex component we implemented on our device for no other reason than the large number of configuration registers necessary for communication. The component is extremely versatile and can be customized for a wide variety of applications. As such, there are many possible configurations for its operation. Some of the configuration possibilities include: transmission frequency, operation mode, sending format, power configuration for each mode, preamble options, sync words, interrupt configuration, and the list goes on. The following provides the basic configuration we decided upon and our reasons why we choose it.

* Frequency: 434 MHz
  + For long range communication
* Modes of operation: TX, RX
  + Our component is always in either TX or RX simply because if it is not sending GPS coordinates it should be listening for another device.
* Sending format: Packet mode
  + This provided the best transmission accuracy as fewer bits were dropped when sending information in this format as opposed to continuous mode.
* Power Configuration:
  + We used the max transmission power of +20dB gain in order to maximize range.
* Preamble:
  + We chose to go with the default preamble format for the packet mode, as there was no need to customize.
* Sync Words:
  + We chose to go with 2 sync words. This was to prevent accidental receival of other radio transmissions and to add a level of encryption to our device.
* Interrupt configuration:
  + We enabled interrupts for both transmission confirmation and receiving confirmation. These interrupts were critical to the overall operation of the device.
  + We used a hardware interrupt pin on the microcontroller to handle these interrupts. This gave a more accurate response as opposed to a pin change interrupt.

There are still more configuration elements, however this provides an overview of some of the most important. Please see code if more configuration details are necessary.

After implementing the component and writing the code library for it, we still experienced problems working with the device. Randomly, during operation, we would experience unexpected dropping of the communication for minutes at a time. This proved to be a poor connection with the wire antenna we were using for the adafruit RFM69 breakout board. Because of this, we added more robust antennas which instantly fixed the issue in addition to providing us range up to 150 meters without line of sight.

Once we developed the code for basic transmission and receiving of information between the devices, we were able to transmit the GPS coordinates between devices enabling peer to peer tracking.

#### Monochrome 128x64 OLED Screen

In order to efficiently display all the data gathered from the various sensors and provide the user with an interface to initiate tracking, we chose to implement a display on the device. In choosing a screen we originally decided to implement a 2.7” color LCD display to provide a fun colorful interface. However, this proved to be both too complex for adequate implementation for our time frame and too memory intensive for our microcontroller. As such we decided to go with a 1.3” monochrome OLED display. This small display is only 128x64 pixels in size allowing its implementation to be on the memory scale of our microcontroller while still proving adequate at displaying all the relevant information. The display from adafruit uses I2C communication which also fits nicely with our overall device.

The implementation of the display was relatively easy, requiring almost no configuration of the component. The challenging aspects were refreshing the screen and enabling character writing to the display. The basic operation of the screen is that every pixel corresponds to a single byte. When the byte is one, the pixel is on. When the byte is zero, the pixel is off. This is a waste of memory as it only needs a single bit to represent this binary state, but addressing this issue proved outside the scope of our development time. When something is needed to be displayed on the screen, a buffer of 1024 bytes is written to the microcontroller then transferred to the screen. However, there is one trick. The buffer does not reset after a write. This caused some small issues where shapes and characters remained after they were no longer wanted. There is a simple solution, simply memset the microcontrollers buffer to zero before drawing new shapes and characters.

Besides this small issue of refreshing the screen, there was also a problem with initializing the components internal buffer. It turns out on boot up, the component buffer is initialized with garbage data. And before any writing to the screen can occur the reset must be triggered on the component. For this reason we have a dedicated reset pin for the screen on our microcontroller.

The last challenge for the screens implementation was enabling character writing to the display. To accomplish this we utilize a font header file found online. This header file contained the bit layout for standard ASCII characters that fit nicely on or board. Therefore, in order to display characters we simply had to retrieve the character’s bit layout then write the bits in the proper location within the microcontroller screen buffer.

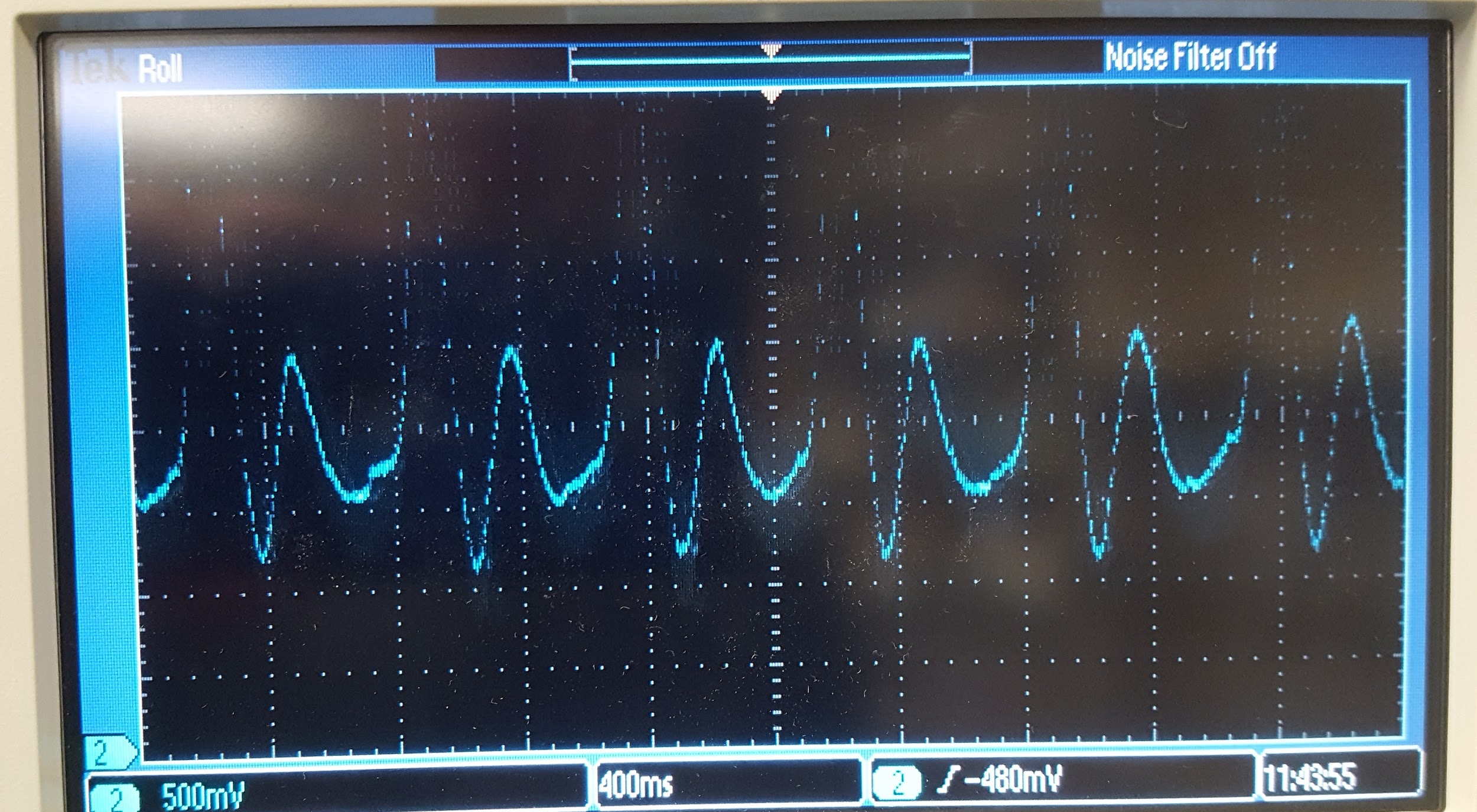
This screen was primarily used to display our various outputs, ranging from local GPS data to distance and direction from one device to the other. While data could be displayed in other methods, such as terminal screens or LEDs, we felt a physical screen was the easiest method to read data as it allowed more portability of the device when testing it in the field. Also our intended finished device employs a screen so we believed it an important aspect to the overall functionality of the device.

#### Heart Rate Monitor

In keeping with the Fitbit brand, we decided to incorporate an optical heart rate monitor on our device since it is a common wearable activity tracker feature. In choosing the sensor we opted for simplicity and ease of use as opposed to pure performance, simply because the sensor is not critical to the functionality of the device. Therefore, we went with an Optical Pulse Sensor from Sparkfun that mounts on a finger.

Implementing the sensor required ADC conversion and a means for translating raw data readings into a Beats Per Minute “BPM” measurement. Since the sensor only outputs raw voltage values corresponding to the light intensity that photodiode is experiencing at the moment, we had to use analog to digital conversion to get useful measurements on the microcontroller. This is why the heart rate sensor’s data pin is connected to the ADC pin on the microcontroller.

Once we were getting the usable data we implemented a basic state machine to measure the beats. The state machine was needed in order to determine when the individual heart beats happened. As it is shown in Figure 3,the analog reading varies with the pulses, as expected. This meant we couldn’t simply say count when the beats happen. The state machine goes between a high state and a low state, transitioning whenever the current reading went above or below a set threshold. The beat count is only incremented on the transition from low to high state as to avoid counting the same heart beat multiple times. In order to turn the beat count into BPM, we implemented a 5s timer. This allowed us to measure how many beats happen every 5s which we used to get BPM by multiplying the count by 12.



**Figure 3: Heart rate monitor output**

While this initial method provided decent results, we wanted more accuracy. To achieve this we implemented a dynamic threshold setting for the state transition. Initially we used a static threshold value for determining when to transfer between high and low state. However, the output range for the sensor would sometimes be shifted, meaning this default threshold would be too low or too high. As such, we implemented a dynamic threshold that would shift with the shifting sensor output. To do this for every 5s batch of data, we would measure the lowest reading and the highest reading. Then we would change the threshold to be approximately 75% between the lowest reading and the highest. This provide more stable results.

Overall the sensor is a little finicky and not the most accurate, often times reporting a heart rate off by about 10 BMP. However it does provide reliable results for distinguishing between when a person is at rest and when they have an elevated heart rate. For instance, it would often read ~80BPM for rest, however after elevating someones heart rate it would noticeable increase to within the 100’s. We were able to notice this by watching the BPM measurement increase on the display as the test subject elevated their heart rate.

#### LSM9DS1 - Accelerometer

Also in keeping with the Fitbit brand we wanted to include a step counter and distance tracking. The distance tracking we were able to implement with the GPS component, however the step counter required us to implement an accelerometer. We chose to go with the Adafruit LSM9DS1 breakout as it combines multiple sensors that we needed: temperature, magnetometer, and accelerometer. In addition to packaging all three of these sensors into a nice component, it also allowed the accelerometer to set interrupt thresholds, an important feature for implementation.

The accelerometer implementation was fairly simple. We use I2C communication to configure the component as well as get raw data readings if needed. With communication established, we configure the device to set an interrupt pin high whenever the x-axis or y-axis acceleration exceeded the threshold. This interrupt pin was connected to a hardware interrupt pin on the microcontroller. Internally, in the code, whenever the interrupt was activated, we simply increase the step count thereby allowing us to keep track of steps. We also displayed the step count on the screen.

#### LSM9DS1 - Temperature

We decided to include this feature purely because the sensor came with it. All it required was reading the measurement from the device and converting the raw readings into the actual temperature in fahrenheit. However, the readings are not very accurate. So much so, if we want to make it a more explicit feature of the device we should implement an independent sensor.

#### LSM9DS1 - Magnetometer

Paramount to the overall functionality of the device is the ability to peer to peer track. As a part of this, we wanted to not simply provide the functionality of saying the tracked person was 50 meters west of the user’s current position, but rather, provide which direction west is relative to the way the user is facing. As such we needed a way to measure the user's heading, thus the inclusion of a magnetometer. The magnetometer is capable of acting as an digital compass by measuring the earth’s magnetic field.

Implementing the magnetometer proved to be tricky when it came to data conversion. On our first attempt to implement the component, the readings we were seeing seemed to be completely garbage with only the smallest trend. We mistook this to mean that the device was not high enough quality and only provided noisy data, requiring a kalman filter in order to obtain useable results. As such we were going to abandon its implementation all together. However, on revisiting the problem, we discovered a slight error in the code dealing with bit fiddling with signed numbers as the source of the problem. After rectifying the issue, we were able to obtain useable data from the magnetometer.

We use this information to display a NSEW compass on the screen that rotates with the user. The quality of the sensor only enables us to dictate whether the device was point in one of the four main directions. However, this was enough resolution for determining the direction the user should travel when tracking another device. We also displayed the raw compass heading on the screen.

### Putting it All Together

As previously mentioned, the code for individual components was developed independently of one another. As such, in order to make a device that function properly we needed to have a single main file that held everything together. For the larger main device, we called this file *Menu.c* and for the smaller device, *Small\_Menu.c*. These files are one large infinite while loop with a state machine implemented within it. That is for every iteration of the while loop, only the code corresponding to the current state would be activated. The state machine goes through all the different functionality of our device. In this section we will detail how this state machine functions to make our device work.

The first element of implementing our main state machine, was enabling an easy means of switching between the individual states. We decided to accomplish this through simple hardware buttons utilizing pin change interrupts. These buttons are debounced using a simple while loop with a tiny delay before and after it. For instance on the main device, the green button takes you backwards through the menu, and the blue button takes you forwards through the menu. We also included a third red button, to go directly to the radio connection state, which we will describe later.



**Figure 4: Simple debouncing code**

### Main State Machine Description

We have a total of seven states, which we will now go through in detail. For each state display we show at least two pieces of information: whether there is a GPS connection, and the current state number. A lack of GPS connection is indicated by a small star in the top right hand corner of the screen.

#### State 1:

In this state we display the current local time, the temperature, and the current compass heading. The current compass heading is the number in the top left of Figure 5. The time is obtained directly from the GPS data, however if the GPS signal is lost the clock continues to operate correctly. This is done by utilizing the 5s timer that was implemented for the heart rate monitor. By using this timer, if the GPS signal is dropped the clock will continue to advance in 5 seconds intervals until the satellite fix is re-obtained. 

#### State 2:

In this state we display the steps taken and the distance traveled. Both of these measurements are continually updated while the device is turned on. Constantly measuring distance proved a slight challenge as we needed to obtain frequent GPS updates regardless of which state the device was in. This caused some issues, as obtaining GPS data takes a few microseconds. This meant that updating the GPS for every iteration of our main loop left little time for our device to do anything else like track steps or measure heart rate. Thus, to avoid this problem, we implemented a second timer and only update the GPS information once every 100 milliseconds. This solved the problem nicely. 

#### State 3:

In this state we measure the user's heart rate using the relevant sensor. The dynamically changing threshold is displayed above the current beats per minute reading. 

#### State 4:

In this state we display the location information obtained from GPS component.

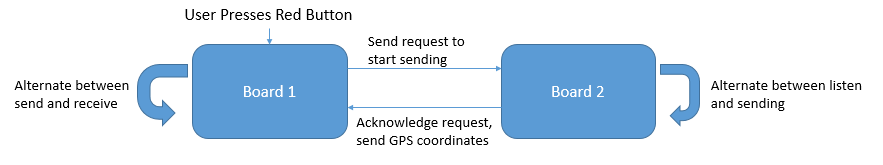
#### State 5:

Displays the compass feature implemented by the magnetometer. The NSEW letters rotate such that the direction the device is currently facing is the topmost letter, so in Figure 9 the device is currently facing E. Along with the compass, the heading in degrees is also displayed.



#### State 6:

This state is accessed when the main device pushes the red button. While in this state the main device is attempting to form a radio link with the small device. It does this by alternating between repeating sending a start command to the small board and then pausing to listen to a response. When the small device receives a start command from the main device, it will immediately jump to its sending state and begin transmitting its coordinates to the main device. The main device leaves state 6 and enters into state 7 once it starts to receive GPS information from the small device. The simple sudo-handshake process is described in the following diagram.



**Figure 11: Handshake process**

#### State 7:

This is the tracking state where the main device is able to track the small device. On this screen we display the distance between the devices, the dynamic compass, and the relative direction the tracked device is from the main device. The relative direction is denoted by the star that appears in between the NSEW. Note, one limitation of our current implementation is that the relative direction can only point in between the main directional axes.

## Board Functionality

Through multiple field tests we are able to accurately gauge the abilities our board possesses. In terms of RF functionality, we are able to connect up to a distance of 0.4 kilometers given line of sight. Once a building is in the way, the connection drops almost immediately. In terms of GPS connectivity the connection time varies dramatically by location and weather settings. Inside the lab it takes around five to ten minutes to achieve a connection even with the GPS antenna right next to the window. Outside the lab, out on the engineering quad, it takes around three to six minutes to acquire a lock. Weather also seems to play a factor, primarily due to the interference caused by clouds or forms of precipitation. It understandably takes a little longer to connect on a cloudy day versus a day with a sunny clear sky. The distance measurement between the devices is very accurate. However, the direction element relative to NSEW varies due to the unpredictable nature of the GPS. Some tests have the GPS coordinates immediately right on mark, while other tests, minutes later, have GPS coordinates that are off by up to 0.4 kilometers. Given time, the GPS does eventually receive the correct coordinates and the direction works properly.

## Project Challenges

Although the board was difficult to build, some challenges stick out in particular. First and foremost is the memory usage. While we were developing the libraries and test functions for each communication protocol and mechanical component memory was never an issue. However, as we assembled everything under the menu file our memory usage hit up to as high as 93% full. This made coding very difficult because we found that sometimes our variables and buffers were being unintentionally overwritten due to memory shortages. In order to combat this, we shortened buffers to their minimum length and constantly reused them. We also made our variables generally shorter, taking longs and changing them into uint8\_t. What ultimately solved this problem by putting the font header file into PROGMEM, which reduce our ROM usage to about 70%

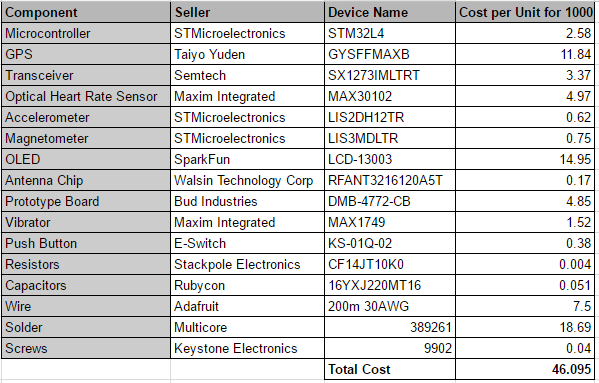
Another challenge that we caught early on is wire management. There were cases where the test board couldn’t flash as a result of snapped wires or accidental short circuits. Also, wire organization was much worse in the beginning, where wires were assigned colors at random or based on component type. As a result, it became rather difficult to determine the appropriate location for each wire. Thankfully, we experienced a frayed wire early on and decided to color coordinate the entire board. While the color coding procedure started with the small board, we decided to proceed with the larger board for organization. This made debugging the final product much easier.

## Cost Analysis

Throughout the building of our device we kept a record of how much each component costs. The total cost to build the two boards is $239. 

This is a rough estimate, as we do not include the cost of the board, wires, screws, resistors, and other small parts that we use. If we were to produce 1000 units of our device using components purchased in bulk, the cost per unit would decrease dramatically.

**Table 2**



Here we present a parts list to make a barebones device without any of the extra components we added such as LEDs, VGA connectors, and switches. For components such as resistors, where we use multiple in one board, the Cost per Unit for 1000 reflects the total cost of all the resistors needed to make one board. The total cost of each board unit is $46.095, excluding the cost of buying a spool of wire for $7.50 and a spool of solder for $18.69.

Conclusion

Although our project is completed, there are still a number of things we wish we could add or improve upon. Given more time, we would have liked to add touch controls for the screen. Using buttons to swap between menu screens was an easy means to an end. We needed a form of input and this was the easiest way to do it. However, in keeping with our Fitbit brand, implementing a touch screen would have made more sense product wise.

Another couple things we would have liked to add is additional safety features. For example, when the devices get too far away from each other the signal is dropped and the circle disappears. If the user is not looking at the device the whole time, there would be no way of knowing that they lost connection. To counteract this, we would have liked to implement haptic feedback via the vibrator. In our current build it only vibrates when a button is pressed or it is in the connecting mode. It would be nice if it also vibrated a couple times to let the user know they walked out of range of their partner.

Also in regard to safety, we would have liked to add notifications for the heart rate monitor. If the user’s heart rate is abnormal, something is obviously wrong. However they could be unconscious or stuck such that they can not signal for help. The heart rate monitor should be able to pick up the anomaly and signal to the partnered device that an emergency has occurred. This way the person with the other device knows that something is wrong immediately rather than waiting and wondering why their dot is not moving.

18339250_10210968690231159_1095636234_o.jpg18339252_10210968690671170_1629638422_o.jpg

**Figure 13: Main Device Figure 14: Small Device**