#

#

#

#

EEar

Final Report

Garrett Dougherty, Gina Gilmartin, Nick Van Lieshout, Kyle Wisbrock

#

#

#

#

# **2 Table of Contents**

[**2 Table of Contents** 2](#_Toc482136857)

[**3 Introduction** 3](#_Toc482136858)

[**4 Detailed System Requirements** 5](#_Toc482136859)

[**5 Detailed Project Description** 7](#_Toc482136860)

[**5.1 System Theory of Operation** 7](#_Toc482136861)

[**5.2 System Block Diagram** 8](#_Toc482136862)

[**5.3 Design/Operation of LCD Subsystem** 8](#_Toc482136863)

[**5.4 Design/Operation of LED Subsystem** 10](#_Toc482136864)

[**5.5 Design/Operation of Vibration Motor Circuit Subsystem** 11](#_Toc482136865)

[**5.6 Design/Operation of Clear Alert Button Subsystem** 12](#_Toc482136866)

[**5.7 Design/Operation of Battery Subsystem** 13](#_Toc482136867)

[**5.8 Design/Operation of Bluetooth Subsystem** 17](#_Toc482136868)

[**5.9 Design/Operation of Hub Subsystem** 21](#_Toc482136869)

[**5.10 Design/Operation of Microphone Subsystem** 23](#_Toc482136870)

[**5.11 Design/ Operation of Touchscreen Interface Subsystem** 24](#_Toc482136871)

[**5.12 Design/Operation of 3D Printing Enclosure** 26](#_Toc482136872)

[**5.13 Interfaces** 27](#_Toc482136873)

[**6 System Integration Testing** 28](#_Toc482136874)

[**6.1 Testing the integrated set of subsystems** 28](#_Toc482136875)

[**6.2 How the design requirements were met through testing** 30](#_Toc482136876)

[**7.1 How to install your product** 30](#_Toc482136877)

[**7.2 How to setup your product** 30](#_Toc482136878)

[**7.3 How the user can tell if the product is working** 31](#_Toc482136879)

[**7.4 How the user can troubleshoot the product** 31](#_Toc482136880)

[**8 To-Market Design Changes** 32](#_Toc482136881)

[**9 Conclusions** 33](#_Toc482136882)

[**10 Appendices** 35](#_Toc482136883)

# **3 Introduction**

Human deafness can make everyday interactions within the home harder to interpret and impossible to notice; since so many alarms or sensors alert humans by making a noise, there has to be a way to detect when something interacts with the objects outside or inside the home, whether it may be a neighbor ringing a doorbell or fire that is growing by the second. A great way that a deaf person can be aware of each audio input in the home is through a wristband that vibrates and displays a text alert on a display when such an input (that ranges in importance) occurs in the home.

Deaf people need a way to be informed when various alarms go off in their homes.

Without a system containing a sound based sensor, they would have no idea if something is wrong within their household. While technologies exist that flash when their phone rings or when someone is at the door, these technologies would be of no help to a deaf person who is sleeping. In addition, there is no product that alerts a person in a different way for the many types of interactions that occur within the modern day household. There is similarly not a device that tells a person the severity of a situation that needs a response. For example, someone ringing the doorbell would elicit a small response whereas a fire or carbon monoxide would trigger a much greater response from the person. Based on this issue present in society, the concept of the EEar product system was born to help illuminate and fix this problem. The goal of the EEar product system is to solve these issues for people who are hearing-impaired with efficiency, accuracy, ease, and comfort for the user.

**High Level Description:**

The EEar product system is based on the intention to make a wristband that communicates with all of the alarm systems in a home, such as smoke alarm, carbon monoxide detector, and intruder alarm for front and back doors. The product will accomplish this by utilizing a hub device designed to always be listening for these types of alarms. The hub will be able to analyze the sound made by the alarm and determine which alarm audio is present. Then, the wristband will receive a signal through Bluetooth capabilities built into both the hub and wristband. This signal will determine which sound is present in the home and will cause the wristband to vibrate and to alert the user of the appropriate alarm. In addition to vibrating, the wristband will display text and light up a correspondingly colored LED indicating which alarm is present at that time. In order to ensure that the device can always be active, it will have a rechargeable battery with a decent lifetime to ensure that it remains on when the user is in the home.

**How well did the design meet expectations?**

Overall, the design was very close to meeting expectations set at the beginning of the semester. Although there were several aspects that were subtracted from the entirety of the design (such as the aspect of multiple hubs, teachable alert names, and microphone separation from hub), the rest of the design was very close to working without issue as an entire integrateds system. The hub was successfully able to be programmed by the user (with predetermined alert names) via a touchscreen GUI, able to identify alert sounds with great accuracy based on the software and microphone integrated into it, and able to connect/push data to the wristband Bluetooth chip without issue.

The wristband was where a little issue was found that caused the design to not be completely flawless. The board was difficult to solder and didn’t end up working perfectly. The board was able to power the nRF board but was not outputting enough power to power the LED and vibration motor when an alert was received (although the LCD would still function). Similarly a solder issue appeared with the fuel gauge chip that made the data received garbage (although data was being transmitted). However, when the wristband was powered with a usb cable the wristband was completely functional.

This product was very innovative and the EEar team put a significant amount of work and effort into making it functional. The lateness of the pending board design seems to be the root of the slight lack of functionality in this product. With a little more time to debug the board issues (and possibly another iteration of the board if necessary), this product would most likely be flawless. However, the EEar team is pleased with the progress made on the product in one semester overall, especially since the EEar team had never attempted to engage in a full design of an engineering product implementation like this before.

# **4 Detailed System Requirements**

The product will consist of two subsystems, including a wristband and a hub sensor device. The hub device will include a microphone, software to analyze the input sound and bluetooth capability. On the hub, there will be a touchscreen interface that will allow the user to program their own alarms by using a GUI to teach the hub what each of the alarms sound like. Once the alarms are input, the hub device will sit and listen for alarms. When a sound above a certain volume threshold occurs, the hub will record the sound. That sound file will then be analyzed via code that uses a fast fourier transform and statistical analysis to identify if the sound is an appropriate alarm to send an alert to the wristband. If the sound is not an identifiable alarm, the hub will not carry out any process and continue listening for the next alarm. If the sound is an alarm, the hub will run a script to send bluetooth data to the wristband to identify the presence and type of alarm. The hub will be plugged into an outlet so that the user will not have to worry about charging it.

The wristband will communicate with the hub via Bluetooth, which is the foundational element of the product. In modern homes, if one smoke alarm goes off, the rest of them also go off which means we will only need one hub in the house. The wristband will wake up from bluetooth low energy mode and receive the alert from the hub. Based on the alert, the wristband will display the appropriate text related to the alarm analyzed by the hub. Additionally, the wristband device will vibrate and light a LED of corresponding color to help the user identify the presence of an alarm within the home. On the wristband, there will be a button that is pressed by the user to clear the warnings once the user has been made aware of these warnings.

The wristband will include a strap for the user’s arm and should be comfortable for the

user in terms of weight and size. Based on research, it was discovered that watches that are considered heavy watches can be upwards of 250 grams. Our product will ideally set 250 grams as the maximum weight and aim for making it lighter. Typical watch faces are about 1.5-1.75 inches in diameter so this product will be planned to have the display be no wider than 1.75 inches at its widest point but design changes might be needed to be made.

The wristband is worn by the user and can be charged by a micro-USB cord when the battery is low. The wristband is powered using a rechargeable Lithium-Polymer battery. The battery will be recharged by placing a micro-USB port on the device. The system would ideally be able to run for at least a day without needing to be charged. Additionally, the bracelet will alert the

user when the battery power has dropped below 20% by using a battery sensing chip that will be

on the bracelet. Since the user is not always at home, they will not always need to be wearing the device. It is at this time that it could be charged via a micro-USB port located on the wristband.

This product is not expected to involve dangerous voltage or current levels. However,

while using electronic devices there is always a risk of electric shock, such as when the device is being plugged into a power source for charging. In addition, lithium-ion batteries present a constant danger of inciting a spontaneous fire if charged improperly. The wristband product will be carefully designed to ensure a safe connection between the battery and the charger so that there is not a safety risk for the user, including charge monitoring and protection circuits within the device.

 Overall, the EEar product system will capable of identifying an alarm and alerting a hearing-impaired user within the home of the appropriate alarm to ensure the user’s everyday safety.

# **5 Detailed Project Description**

## **5.1 System Theory of Operation**

The system will consist of a hub and the wristband. The hub has a microphone which is used for two programs- one that initially records the alarm to store its frequency content (via an FFT), and a program that always listens for alarms that match the test case. Each program creates a recording of five seconds and stores the data in a text file. The continuously running program only records a five second segment when the microphone detects a sound above a certain threshold volume. Both recording programs write the FFT of the recorded audio files to text files; these text files are then opened and read by the last program run involving the audio processing; the FFT comparison program. This is done automatically after the continuously running program records the audio file.

 The FFT comparison program takes the input alarm sound from the microphone and compares it to the saved alerts within the hub software using the stored txt files . If the sound it heard does not match any of the stored alarm sounds, the hub will not alert the wearer of the sound and will continue to listen for the next possible alarm. However, if a match is found, the hub will send a value to the wristband via Bluetooth that will be written into a switch case, allowing the wristband to print a different message to the display depending upon the value received from the hub.

The wristband will also simultaneously vibrate and light up an LED to ensure that the message is received by the user. The message on the LCD screen, the vibration frequency, and the LED color will be specific for the type of alarm that is being detected. The user will receive the corresponding warning until they press the clear button on the wristband.

## **5.2 System Block Diagram**

Wristband

Hub

LCD

LED

Vibrator

Battery Check

Microphone

Touchscreen

Interface

Clear Alert Button

## **5.3 Design/Operation of LCD Subsystem**

The requirements of this subsystem was the ability to pass a message from the wristband board system to an LCD using SPI communication based on the Bluetooth signal the wristband receives. The Sparkfun micro oled breakout LCD part was used as the LCD in this subsystem. The system was designed to be attached via wires to the battery system and nRF breakout boards. The pin configurations are as shown below.

|  |  |
| --- | --- |
| LCD Pins  | nRF Board Pins |
| GND | GND |
| 3V3 | 3V3 |
| SD1 | 11 |
| SCK | 13 |
| D/ | 20 |
| CKST | 23 |
| CS | 22 |

In order to make the LCD system work, code was used via Arduino software, which contained a library available for the oled. The software would wait for a Bluetooth alert to wake it up from Bluetooth Low Energy mode (to minimize battery power consumption) and based on the message received, the code would use the SPI network and code using the library to display the appropriate message on the LCD. Choosing these parts and software methods simplified the transitions from an incoming Bluetooth message to a displayed message on the LCD simple and effective. To test this system, a each different Bluetooth message was pushed to the wristband board system and the LCD was checked that it displayed each independent message correctly.

## **5.4 Design/Operation of LED Subsystem**

The requirements of the LED subsystem was that the tricolor LED should be lit brightly by the wristband board system in the appropriate color for each Bluetooth message received. This was accomplished through a combination of an I/O pin from the wristband board and a MOSFET drive circuit to allow for enough current to light the LED brightly. An issue that needed addressing was the low current drive from the I/O pins of the wristband board which led to the addition of the MOSFET drive circuit. The pin connections were as follows:

|  |  |
| --- | --- |
| LED Pin | nRF Board Pin |
| GND  | GND |
| Blue  | 19 |
| Green  | 20 |
| Red | 21 |

Additionally the schematic for the MOSFET network is shown below:



This circuit functions to boost current through the LED by using a 2N2222A n-channel MOSFET when the attached I/O pin is driven high. The Arduino code functions to drive the LED pin high when an alert is received which in turn drives the correct color pin high, lighting the LED. This was a fairly easy way of lighting the LED in three different colors with a limited amount of output current, which is why these parts and schematics were designed and utilized. In order to test the functionality of this subsystem, all available Bluetooth messages were sent from the hub to the wristband schematic and the LED was checked that it both lit up and was the correct color.

## **5.5 Design/Operation of Vibration Motor Circuit Subsystem**

The requirements of the vibration motor subsystem was that the motor should be vibrating strongly enough to wake up the user when a Bluetooth message was received. This was accomplished through a combination of an I/O pin from the wristband board and a MOSFET drive circuit similar to the LED subsystem to allow for enough current to allow the motor to vibrate. An issue that needed addressing was the low current drive from the I/O pins of the wristband board which led to the addition of the MOSFET drive circuit. The pin connections were as follows:

|  |  |
| --- | --- |
| Vibration Motor Pin | nRF Board Pin |
| 1 | 17 |
| 2 | Collector Pin of FET |

Additionally the schematic for the MOSFET network is shown below:



This circuit functions to boost current through the motor (which is essentially an inductor) by using a 2N2222A n-channel MOSFET when the attached I/O pin is driven high. In addition, a protection diode was added to protect the motor from a back emf within the circuit. The Arduino code functions to drive the I/O pin high when an alert is received which in turn drives the FET circuit causing the motor to vibrate. This was a fairly easy way of making the motor vibrate at a strong enough frequency to wake up a user with a limited amount of output current, which is why these parts and schematics were designed and utilized. In order to test the functionality of this subsystem, all available Bluetooth messages were sent from the hub to the wristband schematic and the vibration motor was checked that it vibrated.

## **5.6 Design/Operation of Clear Alert Button Subsystem**

The requirement of the clear alert button subsystem is that it functions to clear an alert received by the user so that the wristband can in turn be able to receive the next alert pushed to it via Bluetooth. This was a very simple subsystem, the button was simply attached to the reset pin and to ground on the other side. Pushing the button would subsequently reset the wristband and prepare it for the next alert. The pins used were as follows:

|  |  |
| --- | --- |
| Switch Pin | nRF Board Pin |
| 1 | 21 (RESET) |
| 2 | GND |

The schematic was as follows:



Since the wristband board is designed to reset when the reset pin is pulled low, this system was clearly the simplest way to achieve this requirement. This subsystem was tested by sending an alert to the wristband via bluetooth, seeing the presence of an alert, and pushing the reset button to ensure that the alert was cleared.

## **5.7 Design/Operation of Battery Subsystem**

The subsystem of the battery system was required to be able to consistently power the nRF board it was combined with to create the wristband sandwich board. In addition, the battery system was also required to be able to charge the battery and give the nRF data about the state of charge of the battery. To complete this requirement, a breakout board was designed that attached to the nRF board to accomplish these requirements. The entire schematic of the battery subsystem is shown below:



This schematic was transformed into a PCB that perfectly stacked on top the of nRF board and is shown below:



There are various systems within this schematic, the first of which is the charging circuit. The circuit was designed to allow the Lithium Polymer battery to be charged by a microUSB input. In order to ensure that the battery was not overcharged and was managed efficiently, a microchip MCP73831 battery management chip was added between the micro-USB and battery to ensure safe and efficient charging. These parts were chosen for the charging circuit because it was a simple, compact and elegant way of charging the battery in a small space. No software was required for this part of the board.

Next, was the Li-Po Converter circuit. This essentially consisted of connecting the battery to a Texas Instruments TCP61200 input voltage buck boost converter chip. This was necessary in order to ensure the battery was outputting a steady 3.3V to the entire wristband system. This chip was chosen because it was relatively simple and efficient chip that was specially designed by TI for battery voltage conversion. This output was then connected to the nRF via header pins (connection network part of schematic). No software was required for this part of the board.

Finally, was the Fuel-Gauge circuit. This consisted of a connecting the battery to the Maxim MAX17043 chip and connecting the outputs to the nRF board. The purpose of this chip was to collect charge data from the battery and alert the user when the battery is low and needs to be charged. This part was chosen because it was relatively simple compared to most other fuel gauge chips on the market and had a one-shot algorithm built in with an accuracy of 1/256% state of charge. The outputs of the fuel gauge consisted of an I2C data output to the nRF chip and an ALRT threshold pin connected to an I/O on the nRF board.

|  |  |
| --- | --- |
| Fuel Gauge Pin | nRF Board Pin |
| ALRT | 22 |
| SCL | 37 |
| SDA | 29 |

Software in Arduino was then written that initialized the I2C network to make the fuel gauge chip the slave, initialize the data transfers, set the ALRT threshold to 20% SOC, read the data when a bluetooth alert is received so the chip is awoken from low power sleep mode and charge data can be converted to an integer/printed to the LCD, and the check the ALRT pin if it goes to logic 0 indicating a low state of charge (then push alert to the user).

This subsystem was tested by checking all of the solder on the board, debugging with a multimeter, and soldering to the nRF board. Then, a bluetooth alert was sent to the nRF chip and the LCD, tricolor LED, and vibration motor were checked for functionality and correct power values. Unfortunately, this was a subsystem that did not quite work due to low output voltage current and voltage based on the output of the Li-Po conversion circuit.

## **5.8 Design/Operation of Bluetooth Subsystem**



Using terminal commands, it was possible to complete all of the necessary Bluetooth operations. The first tool used, bluetoothctl, has the ability to pair and connect Bluetooth devices based on their Bluetooth address, but it does not include any sort of data communication commands. The Bluetooth address of the wristband is learned by using the command “scan on” when using bluetoothctl. “Scan on” shows the address and name of nearby Bluetooth devices. Prior to this, the wristband board is programmed to have the device name “EEar” to make it easier to identify. After finding out the Bluetooth address of the wristband, the search for Bluetooth devices is stopped using “scan off”. The wristband is then paired using “pair XX:XX:XX:XX:XX:XX” with the address learned from “scan on”. When the hub and devices are paired using bluetoothctl, they are also connected. Pairing and connecting are two different processes. “Pairing” refers to first having the two devices find each other and accepting any PIN that might need to be given. “Connecting” refers to the devices actually being connected and able to transmit data. Devices that have been paired will only be connected if they are in range of each other. It is important to then disconnect the devices while keeping them paired. This is because to use the second tool, gatttool, properly, the devices must be connected within that tool. Keeping the devices connected via bluetoothctl and simply trying to send an integer using gatttool does not work. After disconnecting the devices using bluetoothctl, gatttool is opened. It is important to include the “-t” when opening gatttool, as this is necessary for nRF boards to connect properly. The wristband is then connected and a value can be written using “char-write-req”. This command writes an integer to change the value of a Bluetooth characteristic. The two arguments that are required for this command to work are the handle of the characteristic and the value to be sent, ranging from 0-99. To get the handle of the characteristic, the command “char-desc” is used. This command lists all of the available services on the device. It will list the attribute handle, and the UUID for each service. Using the Arduino code programmed in the wristband, the UUID is set to a recognizable value so that the user can properly identify the handle of the characteristic to be changed. The syntax for the handle is important. For example, if the handle after using “char-desc” is listed as 0x0001, then the handle to be used should be “1”. After the value is sent, gatttool is exited and bluetoothctl is reopened to remove the paired device. This is necessary for automating the Bluetooth process.

 After figuring out how to get Bluetooth working using the terminal, it was necessary to find a way to write a script that will run through all of these commands sequentially on its own. To do this, a terminal command called “autoexpect -f XXXX” is run that takes any user inputs into the terminal and writes them into a special script where “XXXX” is whatever the user wants to name the program. Below is a screenshot of what this script looks like.



 The top part of the code is used to change the rate at which the script steps through each command. This is used if one command is dependent upon the action of a previous command. There are two important functions generated by the autoexpect command, send and expect. The send command takes the place of the user typing in the terminal. Expect has two variations, one in which it expects exactly the text in its argument, and another in which is just needs to see a portion of the argument. Expect exact is used when corresponding to commands that are sent using the send function. Expect exact is used for all of the commands used. The more generic version, exact, is used in cases where the next command cannot be run until some task is complete. For example, “expect ‘EEar’” can be seen in the photo as it is used after the scan on command so that it will not stop scanning until the device address for the wristband is found. It was later changed so that the code would expect the Bluetooth address of the wristband so that, in the event that there are multiple users of EEars from different households in the same house, the hub would pair with the correct device, i.e., the wristband of the person who lives in that house. Expect is also used when pairing, disconnecting, connecting, and sending values because these processes all must be completed before the next command is run. The first time the user attempts to send an integer via Bluetooth, scan on must be used to find the device address. After that, upon opening bluetoothctl, the EEar device will be listed as a device without having to use scan on. That would mean that the autoexpect script that is generated only works the first time it is used. Because the device address would be listed before running scan on, it would always be waiting to see the device address and never turn off scanning. To get around this, the devices are unpaired after sending the alert to the wristband. This causes the hub to forget the wristband and requires scan on to see it nearby again. Though it seems counter-intuitive to pair and unpair the devices every time an alert it sent, this must be done to automate the process successfully using the script generated by autoexpect.

 After generating the script that automates the Bluetooth process, the Python library subprocess is used within the program that is continuously listening for alerts. Subprocess can execute any line of code. In this case, it executes the code necessary to send the alert. One of the deficiencies of sending the alerts in this way is that a script must be generated for each value that could possibly be sent to the wristband. Each script is then placed into a nested if statement that corresponds to each alarm.

 Upon receiving the integer, the wristband first wakes up out of a low energy mode. This mode is used to draw as little current as possible since power was a big consideration for the design of the product. The value is passed into a switch statement where each case corresponds to a different alarm. The name of the alarm is printed on the LCD screen, and the pins for the vibrator and LED are set differently for each alarm in an infinite while loop. It was the original intention to be able to exit this loop by pressing the clear button and causing a I/O pin signal to go low. The wristband would then set the LED and vibrator pins low and clear the display before going back into low power mode. After struggling to get this to work, we instead attached the clear button to the reset pin on the nRF board, which instead just sets the wristband back into its initial setup state in low power mode.

 This subsystem was tested in increments. To first test the Bluetooth connection, we used an app called “nRF Connect” that is available in the iOS and Android app stores. This app allows the user to pair with nearby Bluetooth devices and write values to Bluetooth characteristics. After verifying the Bluetooth connection with the app, the Bluetooth connection was then tested using the terminal commands. Finally, the subsystem was tested by running the script generated by autoexpect to send the Bluetooth integers multiple times to verify that it could properly pair and connect each time. The only problem found was that if it was attempted to connect the devices immediately after disconnecting, the wristband may not show up when scanning for it, but this would not be an issue when actually detecting alerts because the alert verification process is slow enough that it would never be trying to send a message immediately after sending a message.

## **5.9 Design/Operation of Hub Subsystem**

 The hub was programmed using the C language. This language was chosen since it’s array processing is relatively impressive, which is important for the functionality needed in the program; it was also chosen since the use of functions within the main function was specifically what was needed when testing the results of the program. The primary objective of the hub was to compare any signals two audio signals passed in using text files, which can be written and opened in separate programs. The audio signals within the text files being compared within the compare function were seven alarm sounds and either one of those seven alarms or another sound that was used to test the function’s response to a non-alarm.

 In order to compare two audio signals (specifically two alarms we are trying to differentiate between) we need to import the FFT (Fast Fourier Transform) of each alarm from their respective text files into two separate arrays. These arrays are then scaled and normalized so that the maximum of each array is 1, which can help eliminate the problem of one alarm being louder than another. Next, the arrays are subtracted and the absolute value of this new array is taken; this is done to get see how different the two arrays are. Ideally, two very different alarms would give frequency differences that are very high- on the order of .1 or greater (since the maximum value possible for scaled frequency difference is 1).

 A number of statistics is then taken from the scaled frequency difference array. The two most important statistics from the list are the mean frequency difference and the maximum frequency difference. This is because the mean frequency difference of the tested alarm would be the lowest of the seven alarms nearly every time. However, when the mean frequency difference of the tested alarm was not lowest, the maximum frequency difference of the tested alarm would often be the lowest of the seven alarms; therefore, to ensure the system would work as many times as possible, the output of the function that compared the FFTs was the product of the mean frequency difference and the maximum frequency difference.

 Within the main function, the compare FFT function would run seven times; one time for each alarm. The product from each alarm comparison output was stored into an alarm product array- the minimum was then found within the array and whichever alarm corresponded to the minimum product was displayed on screen. Initially if an alarm was tested that was not one of the initial seven recorded, it would simply find the minimum product with the seven and output the respect alarm label. However, as we tested more audio signals such as music clips, dogs barking, etc., the usual product of the mean frequency difference and maximum frequency difference was on the order of 10-4, whereas the product for the alarms recorded was 10-7 or lower. Therefore, a threshold was set within the product that if any product was greater than 10-6, the program would not recognize it as an alarm. The last part of the program is to store the number associated with the alarm detected in a text file; this text file is then opened in one of the programs detailed in the microphone subsystem below.

## **5.10 Design/Operation of Microphone Subsystem**

 The microphone subsystem was split into two programs: a program that would record an alarm for five seconds then store it in a text file for comparing to other alarms, and a program that would continuously listen (i.e. record) for a sound greater than a set decibel level and record for five seconds and store it in a text file for comparing to one of the stored alarms. Both programs were written in python since the recording process was more streamlined and the FFT process was built in, unlike C or C++ where a function for an FFT would have to be written within the program.

 The alarm-storing program starts with opening the audio stream and preparing for the recording process; after the audio clip is recorded, a .wav file is created of the five second audio clip. In the next section of the code the .wav file is opened and put into an array. The next section of code is more manageable since the recording process used one channel (i.e. mono). The FFT is then taken of the array and is stored within a text file to be opened and manipulated in the FFT compare program described in the previous section. This program is stored within the GUI program, so whichever button is pressed sets the OPTIONS variable, which then lets the alarm storing program know which alarm is being stored.

 The continuously recording program is slightly more complicated than the alarm-storing program. This time, the first audio stream needs to be opened twice; first to be continuously listening for the audio from the surrounding area to break a decibel threshold; once this is done, a five second clip is recorded by reopening the audio stream and repeating the process from the audio-storing program. In doing so, a .wav file is made that is then put into an array. The FFT of this array is then taken and stored in a new text file that will be compared to the seven stored alarms. To do this, The command “subsystem.call” is used to start the FFT comparing program; once that is finished, the text file containing the number of the detected alarm mentioned in the previous section is opened and depending on the number, a different message is written to the screen using bluetooth and the “subsystem.call” command.

## **5.11 Design/ Operation of Touchscreen Interface Subsystem**

 The 3.5 inch touch screen interface for the Raspberry Pi was programmed using the Python scripting language. Python was chosen because of its compatibility with the Pi and the ease of learning the TkInter module. The graphical user interface (GUI) was created to allow easy setup and use of EEar for future users of the product. The welcome page, or frame, seen by the user is a home page. There are two button options that lead to two different states. If the user clicks on the "Teach Alarms" button, the GUI goes into the teaching mode of the program.

The teaching mode is a new frame of the GUI, where the user is given seven different alarms to choose from to teach the hub. It is necessary for the user to teach the hub the different sounds of the alarms in their house because there is not a universal alarm sound for the alarms in houses, so these sounds cannot be preprogrammed. The seven alarms the user can choose from are: fire alarm, carbon monoxide alarm, security alarm, doorbell, low severity alarm, moderate severity alarm, and high severity alarm. The first four alarms were chosen because they are common sounds in homes, and it would be useful to the user to know if one of them was sounding. The last three alarms are put in so that the user can teach the hub other sounds they wish to be notified about within their own home that are not as common as the other four. For example, someone may want to set up the EEar so that it will alert them when the microwave buzzes to signal that their food is finished cooking. Because not all of the sounds that EEar can recognize are high severity alarms that should cause the user to take safety precautions, they are given a range of severity options for the final three alarm choices. The user can choose one of the seven alarms to teach the hub by pressing on one of the buttons on the teaching page. Once an alarm name is chosen, the user can press the "REC" button which will cause the hub to listen and record for five seconds. Once the user has repeated this step enough times to teach the hub all of the alarm sounds, they can hit the "Back" button to return to the home page.

When the user has returned to the home page, they can put the device into running mode by pressing "Run Program." This puts the hub into its listening mode, where it continuously listens for a sound that is louder than the volume threshold. Once the hub hears an alarm that it matches to one of the alarms the user taught it previously, it loads a new frame that says "ALARM DETECTED," while the wristband simultaneously alerts the wearer which alarm has been detected. This frame also has a "Back" button that the user can press to get back to the home page and set the hub back into listening mode once the alarm has been taken care of.

 The code for the hub GUI can be found in the appendix. A flow chart diagram of the different states is below:

STATE 1:

 home page

STATE 2:

 teach buttons

Choose one of seven alarm names

"Teach Alarms"

"Run Program"

STATE 4:

 run program

Hub listens until an alarm is detected

"REC"

"Back"

STATE 5:

 alarm detected

STATE 3:

 record new alarm

"Back"

 This program was tested by running the GUI in different states of the writing process. The GUI was capable of switching between different frames by clicking buttons on screen before the other subprocesses of listening and recording were added into the code.

## **5.12 Design/Operation of 3D Printing Enclosure**

 The pink casing for the wristband was created by using the MakerBot Replicator. CAD files were created using Creo 3.0 software. The enclosure was printed out as two separate pieces, the top:



and the bottom:



The top piece needed to have holes that could fit the LCD screen, the tricolor LED, and the clear button. The bottom piece needed to have a hole cut out of one of the ends to allow the micro USB to be plugged into the breakout board to charge the battery. It also needed to have a hole on each side to allow a band to be connected to it so that it can be worn on a wrist.

Each piece took about twenty minutes to print. Once the pieces were fit to the correct size for the circuit boards, a wristband was added to the bottom through the holes that were kept open on the sides. The band was opened up and the vibration motor was placed into the end of the band. The wires were then strung back to the main board and connected to the MOSFET circuit within the case. The band was sewn back together and secured to the case.

## **5.13 Interfaces**

The program that compares the incoming sound with the stored alarms was made using C, while the rest of the programs in the hub used Python. To fully integrate the code, it had to be in a program that would be able to complete all of the necessary processes. This would either involve translating the C code into Python or find a way to run the C code immediately after a sound is detected. By utilizing a Python library known as subprocess, it was possible to execute the compare function that was written in C in the Python code. The subprocess command can run any line of code the user inputs, which includes compiling and running other programs. Because C had more convenient libraries to perform the mathematical operations necessary to compare the sounds, it was easier to find a way to run that program rather than rewrite it entirely.

When first trying to figure out how to send bluetooth messages from the hub to the wristband, commands were sent via the terminal. After this was achieved, it was necessary to find a way to execute these same commands programmatically rather than through the terminal. Ideally, this would be done using Python since all of the functionality on the hub was done using Python. Using the terminal command "autoexpect", a script was generated based upon the user inputted terminal commands. The script steps through the same commands and tools necessary for bluetooth communication that would be used in the terminal one after the other. After this script was generated, it was executed using subprocess in the Python code to send an integer value that corresponds to the alarm detected.

A 3-color LED and a vibration motor are used to generate different alerts that correspond to each type of alarm. The LED would be green for low severity alarms, blue for moderate alarms, and red for the high severity alarms such as a fire or carbon monoxide. The vibration motor, which was sewn into the wristband, pulses at different rates depending on the alarm as well. The LED and vibration motor were both turned on using I/O pins from the nRF52832 chip. Unfortunately, the current from these pins was too low to turn on either the LED or vibration motor and so a MOSFET network was designed. Biasing a MOSFET using the I/O pin would allow the current to be amplified enough to drive the LED and vibration motor. The first MOSFET network designed using Eagle did not function, so a new network was designed that properly amplified the signals.

In an effort to save space on the wristband, two small boards were used rather than one large board. The two boards were stacked on top of each other, connected by header pins coming out of the nRF board. The header pins were soldered to the nRF board but not the breakout board to make it easier to remove and program the nRF. The display was on a separate board that was connected to the nRF board by soldering wires where the header pins would have gone. All of the components of the MOSFET network (MOSFET, resistor, diode, etc) were soldered together at their leads and then soldered to the nRF board using wires.

# **6 System Integration Testing**

## **6.1 Testing the integrated set of subsystems**

 The subsystems were tested individually before they were brought together as a cohesive unit. The software was tested by adding subprocesses into larger programs once they were believed to be working independently. The Bluetooth code was embedded into the code for the wristband and sent a hardcoded message to be displayed on the LCD screen. Once the hardcoded message was successfully being displayed, the code was adjusted to send a variable from the parent code through Bluetooth to the wristband. Additionally, the code written to turn on the vibration motor and the tricolor LED was added to the wristband code so that it could react accordingly when it received an alert message.

 Once the recording program was working, it was embedded in the GUI code so that the code would run when the user pressed a button on the GUI. To test that the recording process was running correctly and that it was saving files under the correct .wav and .txt files, print statements were added to the code that stated where the code was saving the incoming information. Once it was confirmed that the recordings were happening and saving to the correct files, the print statements were removed from the code.

 The sample comparing program was integrated into the GUI code as well using a subprocess.call. To make sure the correct alarms were being detected, the program originally printed out on the terminal which alarm matched the sound that was heard. When the program was correctly listening, comparing, and matching sounds, the print statements were removed and the Bluetooth messages being sent to the wristband became the way to know if the correct alarm was being detected.

 The breakout board was tested by connecting it to the nRF board and using a multimeter to test the voltages at different points on the board. The measured levels were compared to the levels found when the nRF board was being powered by a computer. The MOSFET circuit was tested in a similar way, using the multimeter to determine the current that could be drawn when the output pin was set to high.

## **6.2 How the design requirements were met through testing**

Testing the alarms and having the results sent to the wristband uses every subsystem; since the entire process of testing (teaching the hub an alarm, listening for an alarm, then comparing it and sending the result to the wristband) takes about 90 seconds, this could be tested many times relatively easily. Therefore, when the testing consistently showed positive results, every subsystem would then be working as planned.

The design requirements set for this project set the goal for a functioning, wearable device that could be taught to recognize different alarms and alert the wearer if one was sounding. By testing the individual subsystems, it was concluded that the teaching function, the listening function, the recording function, and the matching function all worked. This meant the design met the requirements for the functioning hub. By getting the Bluetooth communication working between the hub and the wristband, the design requirement for a wearable device that could alert the wearer of alarms ringing was also met.

**7 Users Manual/Installation manual**

## **7.1 How to install your product**

 To begin the installation of EEar, plug your hub unit into a wall outlet. Once the device is powered on, you can start the teachButtons2.py program. This will bring up the EEar GUI.

## **7.2 How to setup your product**

* When you see the welcome page on the GUI, click on the "Teach Alarms" button to go to the teaching page.
* Once on this page, click on one of the seven prenamed alarm options.
* After choosing an alarm type, hit the "REC" button and trigger the alarm so it will give a test signal of the alarm. The hub will listen and record the sound that is made for five seconds. Once the five seconds are over, the sound is saved in the hub and will be recognized by the hub when it is running in the future.
* Repeat this process for the remaining six alarms.
* After recording samples for all alarms, hit the "Back" button and return to the home page.
* Press the "Run Program" button to have the device enter into the continuously listening mode.
* If it recognizes an alarm, the GUI will show a page that says "ALARM DETECTED." Once you have taken care of the problem, press the "Back" button and put the device back into listening mode.

## **7.3 How the user can tell if the product is working**

 The user can test if EEar is working at any time by triggering an alarm that they have saved and checking if the wristband alerts them that the alarm is sounding.

## **7.4 How the user can troubleshoot the product**

 If an alarm is not being detected by EEar, it is possible that the volume of the alarm is not loud enough to trigger the recording process. The hub may need to be moved closer to whichever alarm this problem is occurring with. If there is too much interference with the noise when an alarm is originally recorded, it may need to be rerecorded to try and save a more clear sound for the program to compare future samples to.

# **8 To-Market Design Changes**

 There are many more modifications that can be made to EEar given more time and money. If this product was to be taken to market, more budget would be directed towards creating smaller circuit boards to be placed in the wristband. This would reduce the size of the case and potentially increase comfort for the wearer. A customized lithium ion battery would also be used in future models of EEar. A better battery would increase the length of life between charge times and create another opportunity to make the wristband more compact.

 With more time, a more customizable alarm naming system can be created for EEar. Currently, the alert message being sent from the hub to the wristband via Bluetooth is too small to carry information including a new name for an alarm. In the future, the user could be able to create a new name for an alarm and have that information be passed to the wristband when the alarm was sounding. Additional time and money would also allow for a hub network to be designed. This network would give the user the opportunity to place hubs on different floors of multi-level houses, expanding EEar's coverage of the home. These hubs will communicate using Wi-Fi while continuing to use Bluetooth to communicate with the wristband.

 Although the microphone used in this prototype of EEar was sufficient, a larger budget would result in the hub using a higher quality microphone that can detect possible alarms better and record more clear audio samples. This would improve the reliability of the product. Another option that would come with a larger budget would be the opportunity for the wearer to choose from a variety of colors and patterns for the wristband circuit casing and band. The band would also come in different sizes. These choices would allow for a more unique experience for each consumer, while also giving them a chance to express themselves with their wristband.

 Because there are already smartwatches on the market, going forward it might be best for EEar to focus on the design of the hub and the wristband software. If these aspects of the project could be sold to preexisting smartwatch companies, more time and energy can be put into making the hub network and wristband communications even stronger. Another possible design addition for the future is a way to have the hub connect to emergency responders if one of the severe alarms is triggered, especially if the wearer does not press the clear button to acknowledge that they have been made aware of the alarm sounding.

# **9 Conclusions**

In the beginning, EEar was just a team of four individuals with an idea to change the world. Through days of toil and trouble, this team consistently beat the odds. Some would compare this team’s heroic nature to the US olympic hockey team that took the ice on February 22nd, 1980 and defeated the Soviet Union. EEar came together with a concept and put in a significant amount of work throughout the semester to develop a product designed to help the world and save the lives of people afflicted with hearing impairment.

This project benefited the knowledge of all members of the EEar team and greatly challenged them to develop a functional product. When the dust cleared, the EEar team was left with a product that was nearly functional and that they could all take a lot of pride in. The hub system was functional and the wristband only had an issue with the breakout board soldering chips and I2C interface between the fuel gauge. Considering all the issues that both did and could have presented themselves throughout the development, this progress made is impressive. The EEar product system proved itself to be an impressive idea that was not completely made functional. With more time or better planning on board design, the EEar team would most likely have succeeded in fixing the remaining bugs and creating a completely functional, life-saving product.

Some conclusions on the technical side of things that the EEar team came to, besides the poor board design timeline, was the difficulty of using wireless technologies and integrating different platforms of devices. Bluetooth communication between Raspberry Pi platforms and Sparkfun platforms proved to be difficult to figure out. Similarly, creating a nRF board and attempting to integrate it into the Arduino code via bootloader integration was difficult as well. Debugging and soldering boards proved difficult as well and the EEar team wishes they would have had more experience with these processes before entering senior year.

The EEar team worked cohesively throughout the semester to develop a project that needed many different areas of electrical engineering. Some problems proved to be trivial and others non-trivial, but the team strove to fix every single problem it faced. The teamwork skills and technical knowledge gained from this project are essentially priceless. Overall, the EEar team is very proud of the work they did this semester and would like to thanks all those that helped them along the way.

# **10 Appendices**

**Complete Hardware Schematics**

LCD Schematic:



Vibration Motor Schematic:



Clear Alert Button Schematic:



Wristband Breakout Board Schematic:



**Complete Software Listings:**

**Bluetooth Alert Code:**

\*each alert file is identical to this one, except for the integer being sent. Values range from 01-07

set force\_conservative 0 ;# set to 1 to force conservative mode even if

 ;# script wasn't run conservatively originally

if {$force\_conservative} {

 set send\_slow {1 .1}

 proc send {ignore arg} {

 sleep .1

 exp\_send -s -- $arg

 }

}

set timeout -1

spawn $env(SHELL)

match\_max 100000

expect -exact "^[\]0;pi@raspberrypi: ~^G^[\[01;32mpi@raspberrypi^[\[00m:^[\[01;34m~ \$^[\[00m "

send -- "sudo bluetoothctl\r"

expect -exact "sudo bluetoothctl\r"

send -- "scan on\r"

expect -exact "scan on\r"

expect "F5:67:94:92:EB:FB"

send -- "scan off\r"

expect -exact "scan off\r"

send -- "pair F5:67:94:92:EB:FB\r"

expect -exact "pair F5:67:94:92:EB:FB\r"

expect "Pairing successful"

send -- "disconnect F5:67:94:92:EB:FB\r"

expect -exact "disconnect F5:67:94:92:EB:FB\r"

expect "disconnected"

send -- "quit\r"

expect -exact "quit\r"

send -- "gatttool -t random -b F5:67:94:92:EB:FB -I\r"

expect -exact "gatttool -t random -b F5:67:94:92:EB:FB -I\r"

send -- "connect\r"

expect -exact "connect\r"

expect "Connection successful"

send -- "char-write-req e 01\r"

expect -exact "char-write-req e 01\r"

expect "successfully"

send -- "quit\r"

expect -exact "quit\r"

send -- "sudo bluetoothctl\r"

expect -exact "sudo bluetoothctl\r"

send -- "remove F5:67:94:92:EB:FB\r"

expect -exact "remove F5:67:94:92:EB:FB\r"

send -- "quit\r"

expect -exact "quit\r"

send -- "exit\r"

expect -exact "exit\r"

expect eof

**Wristband Code:**

// Import libraries

#include <SPI.h>

#include <Wire.h>

#include <SFE\_MicroOLED.h>

#include <BLEPeripheral.h>

#include <variant.h>

//defines for I2C

//changed SDA and SCL pins in variant.h file to 24 for SDA and 25 for SCL

//defines for SPI

/\*

 \* #define PIN\_SPI\_MISO (12) D2 on board

 \* #define PIN\_SPI\_SCK (13) D0 on board

 \* #define PIN\_SPI\_MOSI (11) D1 on board

 \* Chip select active low

 \*/

//#define PIN\_SPI\_MISO (12)

//#define PIN\_SPI\_SCK (13)

//#define PIN\_SPI\_MOSI (11)

#define PIN\_DC 20 // Connect DC to pin 8 (required for SPI)

#define PIN\_CS 22 // Connect CS to pin 10 (required for SPI)

#define PIN\_RESET 23 // Connect RST to pin 9 (req. for SPI and I2C)

MicroOLED oled(PIN\_RESET, PIN\_DC, PIN\_CS);

//#define DC\_JUMPER 1 // (I2C only) Set to either 0 or 1, matching the value of the DC Jumper based on the solder jumper on the board

//MicroOLED oled(PIN\_RESET, DC\_JUMPER);

//Buzzer Pin

const int BuzzPin = 14;

//3-Color LED Pins

const int RedPin = 16;

const int BluePin = 17;

const int GreenPin = 18;

//Alert Pin

const int ALRT\_PIN = 19;

//LED Pin

const int ledPin = 7;

// BLE Advertisments

const char \* localName = "EEar";

BLEPeripheral blePeriph;

BLEService bleServ("1207");

BLECharCharacteristic alarmChar("1207", BLERead | BLEWrite);

void setup() {

 Serial.begin(115200); // Set up serial at 115200 baud

 oled.begin(); //Initialize LCD Screen

 oled.setFontType(0); //Set Font Size

 oled.clear(PAGE);

 //Wire.begin();

 pinMode(ledPin, OUTPUT); // Set Led output

 pinMode(BuzzPin, OUTPUT); // Set Buzzer output

 pinMode(RedPin, OUTPUT); // Set Red LED output

 pinMode(BluePin, OUTPUT); // Set Blue LED output

 pinMode(GreenPin, OUTPUT); // Set Green LED output

 setupBLE(); // Set up Bluetooth

 digitalWrite(ledPin, LOW);

 sd\_app\_evt\_wait();

}

/\*void setupFG(){

 Wire.beginTransmission(54);

 Wire.write(byte(0xFF));

 Wire.write(byte(0x54));

 Wire.endTransmission();// Power ON RESET

 delay(70);

 Wire.beginTransmission(54);

 Wire.write(byte(0x0D));

 Wire.write(byte(0x8C));

 Wire.endTransmission();

 delay(70);

}\*/

void loop() {

 int SOC = 50;

 blePeriph.poll();

 oled.setCursor(0, 0);

 if (alarmChar.written())

 {

 /\*oled.print("test"); //DEBUGGING

 oled.display(); //DEBUGGING

 Wire.beginTransmission(54);

 Wire.write(byte(0x0D));

 Wire.write(byte(0x0C));

 Wire.endTransmission(); //wakes up FG chip

 delay(250); //let SOC data refresh

 oled.print("A"); //DEBUGGING

 oled.display(); //DEBUGGING

 if(ALRT\_PIN){

 Wire.beginTransmission(54);

 Wire.write(byte(0x04));

 Wire.endTransmission();//request SOC data MSB

 Wire.requestFrom(54,1);

 oled.print("B"); //DEBUGGING

 oled.display(); //DEBUGGING

 if(1 <= Wire.available()){

 SOC = Wire.read();

 oled.print("C"); //DEBUGGING

 oled.display(); //DEBUGGING

 } //set SOC int equal to SOC data in bus

 //print SOC value to appropriate place on LCD screen

 Wire.beginTransmission(54);

 Wire.write(byte(0x0D));

 Wire.write(byte(0x8C));

 Wire.endTransmission();// put into sleep mode again

 delay(70);

 oled.setCursor(0,30);

 oled.print(SOC);

 oled.display();

 }

 else if (!ALRT\_PIN){

// print LCD message to tell user to charge battery and <20% battery value displayed

 Wire.beginTransmission(54);

 Wire.write(byte(0x0D));

 Wire.write(byte(0x8C));

 Wire.endTransmission();

 delay(70);

 oled.setCursor(0,20);

 oled.print("Low Battery - Needs Charging");

 oled.display();

 oled.setCursor(0,30);

 oled.print(SOC);

 oled.display();

 delay(5000);

 }

 \*/

 int alarm = alarmChar.value();

 switch (alarm) {

 case 0:

 \_\_SEV(); // wake up from low-energy mode

 oled.clear(PAGE);

 oled.setCursor(0, 0);

 oled.print("All Good");

 oled.display();

 sd\_app\_evt\_wait(); // put device in low-energy mode

 break;

 case 1:

 \_\_SEV(); // wake up from low-energy mode

 oled.clear(PAGE);

 oled.setCursor(0, 0);

 oled.print("Fire");

 oled.display();

 while(1)

 {

 digitalWrite(BuzzPin,HIGH);

 digitalWrite(RedPin,HIGH);

 delay(375);

 digitalWrite(BuzzPin,LOW);

 digitalWrite(RedPin,LOW);

 delay(375);

 }

 digitalWrite(BuzzPin,LOW);

 digitalWrite(RedPin,LOW);

 oled.clear(PAGE);

 sd\_app\_evt\_wait(); // put device in low-energy mode

 break;

 case 2:

 \_\_SEV(); // wake up from low-energy mode

 oled.clear(PAGE);

 oled.setCursor(0, 0);

 oled.print("Carbon");

 oled.display();

 oled.setCursor(0, 10);

 oled.print("Monoxide");

 oled.display();

 while(1)

 {

 digitalWrite(BuzzPin,HIGH);

 digitalWrite(RedPin,HIGH);

 delay(375);

 digitalWrite(BuzzPin,LOW);

 delay(375);

 digitalWrite(RedPin,LOW);

 digitalWrite(BuzzPin,HIGH);

 delay(375);

 digitalWrite(BuzzPin,LOW);

 delay(375);

 }

 digitalWrite(BuzzPin,LOW);

 digitalWrite(RedPin,LOW);

 oled.clear(PAGE);

 sd\_app\_evt\_wait(); // put device in low-energy mode

 break;

 case 3:

 \_\_SEV(); // wake up from low-energy mode

 oled.clear(PAGE);

 oled.setCursor(0, 0);

 oled.print("Security");

 oled.display();

 oled.setCursor(0, 10);

 oled.print("Alarm");

 oled.display();

 while(1)

 {

 digitalWrite(BuzzPin,HIGH);

 digitalWrite(BluePin,HIGH);

 delay(375);

 digitalWrite(BuzzPin,LOW);

 digitalWrite(BluePin,LOW);

 delay(375);

 }

 digitalWrite(BuzzPin,LOW);

 digitalWrite(BluePin,LOW);

 oled.clear(PAGE);

 sd\_app\_evt\_wait(); // put device in low-energy mode

 break;

 case 4:

 \_\_SEV(); // wake up from low-energy mode

 oled.clear(PAGE);

 oled.setCursor(0, 0);

 oled.print("Doorbell");

 oled.display();

 while(1)

 {

 digitalWrite(BuzzPin,HIGH);

 digitalWrite(GreenPin,HIGH);

 delay(750);

 digitalWrite(BuzzPin,LOW);

 digitalWrite(GreenPin,LOW);

 delay(750);

 }

 digitalWrite(BuzzPin,LOW);

 digitalWrite(GreenPin,LOW);

 oled.clear(PAGE);

 sd\_app\_evt\_wait(); // put device in low-energy mode

 break;

 case 5:

 \_\_SEV(); // wake up from low-energy mode

 oled.clear(PAGE);

 oled.setCursor(0, 0);

 oled.print("Low");

 oled.display();

 oled.setCursor(0, 10);

 oled.print("Severity");

 oled.display();

 while(1)

 {

 digitalWrite(BuzzPin,HIGH);

 digitalWrite(GreenPin,HIGH);

 delay(750);

 digitalWrite(BuzzPin,LOW);

 delay(750);

 }

 digitalWrite(BuzzPin,LOW);

 digitalWrite(GreenPin,LOW);

 oled.clear(PAGE);

 sd\_app\_evt\_wait(); // put device in low-energy mode

 break;

 case 6:

 \_\_SEV(); // wake up from low-energy mode

 oled.clear(PAGE);

 oled.setCursor(0, 0);

 oled.print("Moderate");

 oled.display();

 oled.setCursor(0, 10);

 oled.print("Severity");

 oled.display();

 while(1)

 {

 digitalWrite(BuzzPin,HIGH);

 digitalWrite(BluePin,HIGH);

 delay(375);

 digitalWrite(BuzzPin,LOW);

 delay(375);

 }

 digitalWrite(BuzzPin,LOW);

 digitalWrite(BluePin,LOW);

 oled.clear(PAGE);

 sd\_app\_evt\_wait(); // put device in low-energy mode

 break;

 case 7:

 \_\_SEV(); // wake up from low-energy mode

 oled.clear(PAGE);

 oled.setCursor(0, 0);

 oled.print("High");

 oled.display();

 oled.setCursor(0, 10);

 oled.print("Severity");

 oled.display();

 while(1)

 {

 digitalWrite(BuzzPin,HIGH);

 digitalWrite(RedPin,HIGH);

 delay(375);

 digitalWrite(BuzzPin,LOW);

 delay(375);

 }

 digitalWrite(BuzzPin,LOW);

 digitalWrite(RedPin,LOW);

 oled.clear(PAGE);

 sd\_app\_evt\_wait(); // put device in low-energy mode

 break;

 }

 }

}

void setupBLE()

{

 // Advertise name and service:

 blePeriph.setDeviceName(localName);

 blePeriph.setLocalName(localName);

 blePeriph.setAdvertisedServiceUuid(bleServ.uuid());

 // Add service

 blePeriph.addAttribute(bleServ);

 // Add characteristic

 blePeriph.addAttribute(alarmChar);

 // Now that device6, service, charactertistic are set up,

 // initialize BLE:

 blePeriph.begin();

 // Set alarm characteristic to default value:

 //alarmChar.setValue(0);

}

**GUI Code:**

from functools import partial

from Tkinter import \*

import Tkinter

import subprocess

import pyaudio, wave, sys, math, csv, numpy

from scipy.fftpack import fft

from scipy.io import wavfile # get the api

Keyboard\_App = Tk()

Keyboard\_App.title("Keyboard")

Keyboard\_App ['bg']='powder blue'

Keyboard\_App.geometry('477x252')

Keyboard\_App.resizable(0,0)

def go\_to\_bp():

 bp.pack()

 p1.pack\_forget()

def go\_to\_run():

 run.pack()

 p1.pack\_forget()

# subprocess.call("python cont\_record.py", shell=True)

 execfile("cont\_record.py")

def p1\_from\_bp():

 p1.pack()

 bp.pack\_forget()

def p1\_from\_run():

 p1.pack()

 run.pack\_forget()

# subprocess.call("^C", shell=True)

# start page (p1)

p1 = Frame(Keyboard\_App)

p1 ['bg']='powder blue'

labelSpace3 = Label(p1, text = " ", font = ('arial',
 10,'bold'),bg='powder blue', fg='white').grid(row=0,
 columnspan = 40)

labelp1 = Label(p1, text = "Welcome to EEar", font = ('arial',
 30,'bold'),bg='powder blue', fg='white').grid(row = 1, columnspan = 40)

labelSpace = Label(p1, text = " ", font = ('arial',
 10,'bold'),bg='powder blue', fg='white').grid(row = 2,
 columnspan = 40)

toButtons = Button(p1, text = 'Teach Alarms', width=19,
 bg="light green", activebackground="green", padx=3, pady=3, bd=12,
 command = go\_to\_bp).grid(row=3, columnspan = 40)

labelSpace2 = Label(p1, text = " ", font = ('arial',
 10,'bold'),bg='powder blue', fg='white').grid(row = 4,
 columnspan = 40)

toRunButton = Button(p1, text = 'Run Program', width=19,
 bg="light green", activebackground="green", padx=3, pady=3, bd=12,

# button page (bp)

bp =Frame(Keyboard\_App)

bp ['bg']='powder blue'

OPTION = 1

alarmChosen = StringVar()

alarmChosen.set(' ')

def fire():

 global OPTION

 OPTION = 1

 global alarmChosen

 alarmChosen.set('Fire Alarm')

 return

def co():

 global OPTION

 OPTION = 2

 global alarmChosen

 alarmChosen.set('Carbon Monoxide Alarm')

 return

def security():

 global OPTION

 OPTION = 3

 global alarmChosen

 alarmChosen.set('Security Alarm')

 return

def doorbell():

 global OPTION

 OPTION = 4

 global alarmChosen

 alarmChosen.set('Doorbell')

 return

def low():

 global OPTION

 OPTION = 5

 global alarmChosen

 alarmChosen.set('Low Severity Alarm')

 return

def moderate():

 global OPTION

 OPTION = 6

 global alarmChosen

 alarmChosen.set('Moderate Severity Alarm')

 return

def severe():

 global OPTION

 OPTION = 7

 global alarmChosen

 alarmChosen.set('High Severity Alarm')

 return

def rec():

 global OPTION

 CHUNK = 256

 FORMAT = pyaudio.paInt16

 CHANNELS = 1

 RATE = 44100

 RECORD\_SECONDS = 5

 if OPTION == 1: #smoke

 WAVE\_OUTPUT\_FILENAME = 'smokealarm.wav'

 elif OPTION == 2: #carbon monoxide

 WAVE\_OUTPUT\_FILENAME = 'coalarm.wav'

 elif OPTION == 3: #security alarm

 WAVE\_OUTPUT\_FILENAME = 'securityalarm.wav'

 elif OPTION == 4: #doorbell

 WAVE\_OUTPUT\_FILENAME = 'doorbellalarm.wav'

 elif OPTION == 5: #low

 WAVE\_OUTPUT\_FILENAME = 'lowalarm.wav'

 elif OPTION == 6: #moderate

 WAVE\_OUTPUT\_FILENAME = 'moderatealarm.wav'

 elif OPTION == 7: #severe

 WAVE\_OUTPUT\_FILENAME = 'severealarm.wav'

 p = pyaudio.PyAudio()

 stream = p.open(format=FORMAT,

 channels = CHANNELS,

 rate = RATE,

 input = True,

 input\_device\_index = 0,

 frames\_per\_buffer = CHUNK)

 print("\* recording")

 frames = []

 for i in range(0, int(RATE / CHUNK \* RECORD\_SECONDS)):

 data = stream.read(CHUNK)

 frames.append(data)

 print("\* done recording")

 stream.stop\_stream() # "Stop Audio Recording

 stream.close() # "Close Audio Recording

 p.terminate() # "Audio System Close

 wf = wave.open(WAVE\_OUTPUT\_FILENAME, 'wb')

 wf.setnchannels(CHANNELS)

 wf.setsampwidth(p.get\_sample\_size(FORMAT))

 wf.setframerate(RATE)

 wf.writeframes(b''.join(frames))

 wf.close()

 # TAKE FFT OF RECORDING

 fs, data = wavfile.read(WAVE\_OUTPUT\_FILENAME) # load the data

 a = data.T[0:data.size] # this is a two channel soundtrack, I get the first track

 b=[(ele/2\*\*16.)\*2-1 for ele in a] # this is 16-bit track, b is now normalized on [-1,1)

 alarm\_fft = abs(fft(b)) # calculate fourier transform (complex numbers list)

 print len(alarm\_fft)

 if OPTION == 1: #smoke

 numpy.savetxt("smokealarm.txt",alarm\_fft,"%f")

 print("Writing Smoke Text File")

 elif OPTION == 2: #carbon monoxide

 numpy.savetxt("coalarm.txt",alarm\_fft,"%f")

 print("Writing Carbon Monoxide Text File")

 elif OPTION == 3: #security alarm

 numpy.savetxt("securityalarm.txt",alarm\_fft,"%f")

 print("Writing Security Text File")

 elif OPTION == 4: #doorbell

 numpy.savetxt("doorbellalarm.txt",alarm\_fft,"%f")

 print("Writing Doorbell Text File")

 elif OPTION == 5: #low

 numpy.savetxt("lowalarm.txt",alarm\_fft,"%f")

 print("Writing Low Text File")

 elif OPTION == 6: #moderate

 numpy.savetxt("moderatealarm.txt",alarm\_fft,"%f")

 print("Writing Moderate Text File")

 elif OPTION == 7: #severe

 numpy.savetxt("severealarm.txt",alarm\_fft,"%f")

 print("Writing Severe Text File")

label1 = Label(bp, text = "EEar Alarm Setup", font = ('arial',

 25,'bold'),bg='powder blue', fg='white').grid(row = 0, columnspan = 40)

fireButton = Button(bp, text = 'Fire', width=14,

 bg="coral", activebackground="orange red", padx=1, pady=1, bd=12,

 command = fire).grid(row=2, column = 7, columnspan = 6)

coButton = Button(bp, text = 'Carbon Monoxide', width=14,

 bg="pale green", activebackground="forest green", padx=1, pady=1, bd=12,

 command = co).grid(row=2, column = 15, columnspan = 6)

securityButton = Button(bp, text = 'Security Alarm', width=14,

 bg="light blue", activebackground="blue", padx=1, pady=1, bd=12,

 command = security).grid(row=3, column = 7, columnspan = 6)

doorbellButton = Button(bp, text = 'Doorbell', width=14,

 bg="medium orchid", activebackground="dark violet", padx=1, pady=1, bd=12,

 command = doorbell).grid(row=3, column = 15, columnspan = 6)

lowButton = Button(bp, text = 'Low', width=14,

 bg="light green", activebackground="green", padx=1, pady=1, bd=12,

 command = low).grid(row=4, column = 7, columnspan = 6)

moderateButton = Button(bp, text = 'Moderate', width=14,

 bg="light goldenrod", activebackground="gold", padx=1, pady=1, bd=12,

 command = moderate).grid(row=4, column = 15, columnspan = 6)

severeButton = Button(bp, text = 'Severe', width=14,

 bg="pink", activebackground="red", padx=1, pady=1, bd=12,

 command = severe).grid(row=5, column = 7, columnspan = 6)

backButton = Button(bp, text = 'BACK', width=8,

 bg="red", activebackground="pink", padx=1, pady=1, bd=6,

 command = p1\_from\_bp).grid(row=2, column = 1, columnspan =1)

recLabel = Label(bp, text = "Record ", font = ('arial',

 15,'bold'),bg='powder blue', fg='white').grid(row = 3, column=1, columnspan = 1)

recLabel2 = Label(bp, text = "Sound for: ",font = ('arial',

 15,'bold'),bg='powder blue', fg='white').grid(row =4 , column=1,

 columnspan = 1)

recLabel1 = Label(bp, textvariable=alarmChosen, font = ('arial',

 15,'bold'),bg='powder blue', fg='white').grid(row = 9, column=1,

 columnspan = 15)

recButton = Button(bp, text = "REC", width=8,

 bg="red", activebackground="pink", padx=1, pady=1, bd=6,

 command = rec).grid(row=9, column = 15, columnspan = 6)

labelSpacebp = Label(bp, text = " ", font = ('arial',

 30,'bold'),bg='powder blue', fg='white').grid(row = 13,

 columnspan = 40)

# running page (run)

run = Frame(Keyboard\_App)

run ['bg']='powder blue'

labelSpacerun = Label(run, text = " ", font = ('arial',

 20,'bold'),bg='powder blue', fg='white').grid(row = 0,

 columnspan = 40)

runLabel = Label(run, text = "ALARM", font = ('arial',

 30,'bold'),bg='powder blue', fg='white').grid(row = 1, columnspan = 40)

runLabel2 = Label(run, text = "DETECTED", font = ('arial',

 30,'bold'),bg='powder blue', fg='white').grid(row = 2, columnspan = 40)

labelSpacerun2 = Label(run, text = " ", font = ('arial',

 20,'bold'),bg='powder blue', fg='white').grid(row = 3,

 columnspan = 40)

endRunButton = Button(run, text = 'BACK', width=18,

 bg="red", activebackground="pink", padx=3, pady=3, bd=12,

 command = p1\_from\_run).grid(row=4, column = 15, columnspan = 6)

# subprocess.call(run, "python cont\_record.py", shell=True)

p1.pack()

Keyboard\_App.mainloop()

**Continuously Listening Code:**

import pyaudio, wave, sys, audioop, math, csv, numpy, subprocess

from scipy.fftpack import fft

from scipy.io import wavfile # get the api

# this is the threshold that determines whether or not sound is detected

THRESHOLD = 140

#open your audio stream

p = pyaudio.PyAudio()

CHUNK = 256

FORMAT = pyaudio.paInt16

CHANNELS = 1

RATE = 44100

RECORD\_SECONDS = 5

stream = p.open(format=FORMAT,

 channels = CHANNELS,

 rate = RATE,

 input = True,

 input\_device\_index = 0,

 frames\_per\_buffer = CHUNK)

# wait until the sound data breaks some level threshold

while True:

 data = stream.read(CHUNK)

 # check level against threshold, you'll have to write getLevel()

 if audioop.max(data,2) > THRESHOLD:

 break

# record for however long you want

print("\* recording")

frames = []

for i in range(0, int(RATE / CHUNK \* RECORD\_SECONDS)):

 data = stream.read(CHUNK)

 frames.append(data)

print("\* done recording")

stream.stop\_stream() # "Stop Audio Recording

stream.close() # "Close Audio Recording

p.terminate() # "Audio System Close

WAVE\_OUTPUT\_FILENAME = 'smoke\_alarmtest.wav'

#WAVE\_OUTPUT\_FILENAME = 'other\_alarmtest.wav'

#WAVE\_OUTPUT\_FILENAME = 'wonderwall\_oasistest.wav'

wf = wave.open(WAVE\_OUTPUT\_FILENAME, 'wb')

wf.setnchannels(CHANNELS)

wf.setsampwidth(p.get\_sample\_size(FORMAT))

wf.setframerate(RATE)

wf.writeframes(b''.join(frames))

wf.close()

# TAKE FFT OF RECORDING

fs, data = wavfile.read('smoke\_alarmtest.wav') # load the data

#fs, data = wavfile.read('other\_alarmtest.wav')

#fs, data = wavfile.read('wonderwall\_oasistest.wav')

a = data.T[0:data.size] # this is a two channel soundtrack, I get the first tra$

b=[(ele/2\*\*16.)\*2-1 for ele in a] # this is 16-bit track, b is now normalized o$

smoke\_alarm\_fft = abs(fft(b)) # calculate fourier transform (complex numbers list)

print len(smoke\_alarm\_fft)

# open a file for writing.

numpy.savetxt("alarmtest.txt",smoke\_alarm\_fft,"%f")

subprocess.call("gcc compare\_fft.c -o compare\_fft -lm",shell=True)

subprocess.call("./compare\_fft")

num = numpy.loadtxt("signalmatch.txt")

if num == 0:

 subprocess.call("expect fire.py",shell=True)

elif num == 1:

 subprocess.call("expect CO.py",shell=True)

elif num == 2:

 subprocess.call("expect security.py",shell=True)

elif num == 3:

 subprocess.call("expect doorbell.py",shell=True)

elif num == 4:

 subprocess.call("expect low.py",shell=True)

elif num == 5:

 subprocess.call("expect moderate.py",shell=True)

elif num == 6:

 subprocess.call("expect high.py",shell=True)

elif num == 7:

 subprocess.call("python cont\_record.py",shell=True)

#if min\_loc = 0 -> Smoke Alarm

#if min\_loc = 1 -> Carbon Monoxide

#if min\_loc = 2 -> Security Alarm

#if min\_loc = 3 -> Doorbell

#if min\_loc = 4 -> Low Level

#if min\_loc = 5 -> Moderate Level

#if min\_loc = 6 -> Severe Level

**Compare FFT Code:**

#include <stdlib.h>

#include <stdio.h>

#include <math.h>

float min\_array(float a[], int num\_elements)

{

 int c,location;

 float minimum;

 minimum = a[0];

 for ( c = 1 ; c < num\_elements ; c++ )

 {

 if ( a[c] < minimum )

 {

 minimum = a[c];

 location = c+1;

 }

 }

 return (minimum);

}

int min\_elem(float a[],int num\_elements)

{

 int c,location;

 float minimum;

 minimum = a[0];

 for ( c = 1 ; c < num\_elements ; c++ )

 {

 if ( a[c] < minimum )

 {

 minimum = a[c];

 location = c+1;

 }

 }

return(location);

}

float max\_array(float a[], int num\_elements)

{

 int i;

 float max=-32000;

 for (i=0; i<num\_elements; i++)

 {

 if (a[i]>max)

 {

 max=a[i];

 }

 }

 return(max);

}

int max\_elem(float a[], int num\_elements)

{

 int i;

 float max=-32000;

 for (i=0; i<num\_elements; i++)

 {

 if (a[i]>max)

 {

 max=a[i];

 }

 }

 return(i);

}

float compare\_max(char \* myFileName)

{

FILE \*myFile, \*myTestFile;

myFile = fopen(myFileName,"r");

myTestFile = fopen("alarmtest.txt","r");

// read file into array

int i=0,j=0;

int size\_fft = 220160;

float record\_fft[220160],scaled\_rf[220160],max\_record,max\_alarm;

float alarm\_fft[220160],scaled\_af[220160],max\_scale\_r,max\_scale\_a,diff\_freq[30];

float min\_scaled;

long double scaled\_diff[220160],mean\_freq\_diff,sd\_freq\_diff,var\_freq\_diff;

for (i = 0; i < size\_fft; i++)

{

fscanf(myFile, "%f", &record\_fft[i]);

}

fclose(myFile);

for (j = 0; j < size\_fft; j++)

{

fscanf(myTestFile, "%f", &alarm\_fft[j]);

}

fclose(myTestFile);

// Setting first elements to zero

alarm\_fft[0] = 0;

record\_fft[0] = 0;

// Here we are going to compare the ffts!

// SCALING

max\_record = max\_array(record\_fft,size\_fft);

max\_alarm = max\_array(alarm\_fft,size\_fft);

for (i = 0; i<size\_fft; i++)

 {

 scaled\_rf[i] = record\_fft[i]/max\_record;

 scaled\_af[i] = alarm\_fft[i]/max\_alarm;

 }

max\_scale\_r = max\_array(scaled\_rf,size\_fft);

max\_scale\_a = max\_array(scaled\_af,size\_fft);

for (i = 0; i<size\_fft; i++)

 {

 scaled\_diff[i] = scaled\_rf[i] - scaled\_af[i];

 if (scaled\_diff[i] < 0)

 scaled\_diff[i] = -1\*scaled\_diff[i];

 }

// FINDING DIFFERENCES / ALLOWABLE ERROR

double spacing = 7000;

for (i = 1; i<27; i++)

 {

 int c;

 c = i\*spacing;

 diff\_freq[i] = scaled\_diff[c];

 }

diff\_freq[27] = scaled\_diff[17210];

diff\_freq[28] = scaled\_diff[17211];

diff\_freq[29] = scaled\_diff[17212];

diff\_freq[30] = scaled\_diff[17213];

double sum=0;

for (i = 0; i<30; i++)

 {

 sum = sum + diff\_freq[i];

 }

 mean\_freq\_diff = sum/30;

printf("Mean Frequency Difference: %lf\n",mean\_freq\_diff);

double sum1=0;

for (i = 1; i<=30; i++)

 {

 sum1 = sum1 + pow((diff\_freq[i] - mean\_freq\_diff), 2);

 }

 var\_freq\_diff = sum1/30;

 sd\_freq\_diff = sqrt(var\_freq\_diff);

long double max\_allow\_error,min\_allow\_error;

if (sd\_freq\_diff/mean\_freq\_diff < 0.000228/0.000252 )

 sd\_freq\_diff = sd\_freq\_diff;

max\_allow\_error = mean\_freq\_diff + ( 2.72 \* sd\_freq\_diff / sqrt(30) );

min\_allow\_error = mean\_freq\_diff - ( 2.72 \* sd\_freq\_diff / sqrt(30) );

printf("Standard Deviation of Freq. Diff. = %lf\n",sd\_freq\_diff);

printf("Maximum Allowable Freq Diff. = %lf\n",max\_allow\_error);

printf("Minimum Allowable Freq Diff. = %lf\n",min\_allow\_error);

float ave = .5\*(max\_allow\_error + min\_allow\_error);

if (sd\_freq\_diff < max\_allow\_error && mean\_freq\_diff < ave)

 {

 max\_allow\_error = max\_allow\_error\*2.5;

 min\_allow\_error = 0;

 }

float max\_diff = max\_array(diff\_freq,30);

float min\_diff = min\_array(diff\_freq,30);

printf("Maximum Frequency Difference = %lf\n",max\_diff);

printf("Minimum Frequency Difference = %lf\n",min\_diff);

printf("New Maximum Allowable Freq Diff. = %lf\n",max\_allow\_error);

printf("New Minimum Allowable Freq Diff. = %lf\n",min\_allow\_error);

float mean\_max\_diff = max\_diff\*mean\_freq\_diff;

printf("Product of Maximum Freq. Diff. and Mean Freq. Diff. = %.10e\n",mean\_max\_diff);

return(mean\_max\_diff);

}

int main() {

int min\_loc,firstResult,secondResult,thirdResult,fourthResult,fifthResult,sixthResult,seventhResult;

float firstMax,secondMax,thirdMax,fourthMax,fifthMax,sixthMax,seventhMax;

printf("\nSmoke Alarm Test\n");

firstMax = compare\_max("smokealarm.txt");

printf("\nCarbon Monoxide Alarm Test\n");

secondMax = compare\_max("coalarm.txt");

printf("\nSecurity Alarm Test\n");

thirdMax = compare\_max("securityalarm.txt");

printf("\nDoorbell Alarm Test\n");

fourthMax = compare\_max("doorbellalarm.txt");

printf("\nLow Level Alarm Test\n");

fifthMax = compare\_max("lowalarm.txt");

printf("\nModerate Level Alarm Test\n");

sixthMax = compare\_max("moderatealarm.txt");

printf("\nSevere Level Alarm Test\n");

seventhMax = compare\_max("severealarm.txt");

float max\_val[] = {firstMax,secondMax,thirdMax,fourthMax,fifthMax,sixthMax,seventhMax};

int a;

 int c,location;

 float minimum;

 minimum = max\_val[0];

 for ( c = 1 ; c < 7 ; c++ )

 {

 if ( max\_val[c] < minimum )

 {

 minimum = max\_val[c];

 location = c+1;

 }

 }

if ( minimum == max\_val[0] )

 {

 min\_loc = 0;

 }

else

 {

 min\_loc = location-1;

 }

int i;

float sum;

for (i = 0; i<7;i++)

 {

 sum = sum + max\_val[i];

 }

//printf("\nmin\_loc = %d\n",min\_loc);

if (max\_val[min\_loc] > .000001 ) {

 printf("\nAlarm Not Detected.\n");

 min\_loc = 7;

 }

if (min\_loc == 0 || min\_loc == -1 || min\_loc == 66789) {

 printf("\nSmoke Alarm Confirmed\n");

 min\_loc = 0;

 }

if (min\_loc == 1) {

 printf("\nCarbon Monoxide Alarm Confirmed\n");

 }

if (min\_loc == 2) {

 printf("\nSecurity Alarm Confirmed\n");

 }

if (min\_loc == 3) {

 printf("\nDoorbell Alarm Confirmed\n");

 }

if (min\_loc == 4) {

 printf("\nLow Level Alarm Confirmed\n");

 }

if (min\_loc == 5) {

 printf("\nModerate Level Alarm Confirmed\n");

 }

if (min\_loc == 6) {

 printf("\nSevere Level Alarm Confirmed\n");

 }

printf("Minimum Product = %.10e\n", max\_val[min\_loc]);

// WRITE IS\_EQUAL TO TXT FILE

FILE \*f = fopen("signalmatch.txt","w");

if (f == NULL)

{

 printf("Error opening file!\n");

 exit(1);

}

fprintf(f,"%d",min\_loc);

fclose(f);

}

**Relevant Parts with Links to Component Data Sheets and Information**

**Wristband Parts:**

Battery:

<https://www.sparkfun.com/products/13853>

microUSB port:

<https://www.digikey.com/product-detail/en/amphenol-fci/10118194-0001LF/609-4618-2-ND/2785389>

Battery Charge Management Chip:

<http://www.microchip.com/wwwproducts/en/en024903>

Fuel Gauge Chip:

<https://www.maximintegrated.com/en/products/power/battery-management/MAX17043.html>

Buck-Boost Voltage Converter Chip:

<http://www.ti.com/product/TPS61200>

Tricolor LED:

<https://www.adafruit.com/product/159>

Push-Button:

<https://www.digikey.com/products/en?keywords=450-1655>

LCD:

<https://www.sparkfun.com/products/13003>

nRF Board:

<https://www.sparkfun.com/products/13990>

**Hub Parts:**

Raspberry Pi3:

<https://www.digikey.com/product-detail/en/seeed-technology-co-ltd/114990584/1597-1366-ND/5891994>

Samson Go-Mic:

<http://www.bestbuy.com/site/samson-go-mic-direct-portable-usb-microphone-with-noise-cancellation-technology/4901848.p?skuId=4901848&extStoreId=&ref=212&loc=1&ksid=82986e88-83e9-4e57-9a68-c15cd30d3c78&ksprof_id=3&ksaffcode=pg218886&ksdevice=c&lsft=ref:212,loc:2&gclid=CM_Wkf3l49MCFY-2wAod-ooLEw>

Raspberry Pi Touchscreen Shield:

<http://kumantech.com/kuman-35-inch-tft-lcd-display-480x320-rgb-pixels-touch-screen-monitor-for-raspberry-pi-3-2-model-b-b-a-a-module-spi-interface-with-touch-pen-sc06_p0014.html>