TurnUp - Real Time Volume Control

Harold Healy, EE, Ryan Walsh, CSE, Rahaun Perkins, CSE, and Nicholas Kafasis, CSE

Abstract—**The demand for autonomous systems is rapidly increasing all over the world. Both suppliers and consumers make every effort to use autonomous systems for their efficiency and ease of use. This technology will bring that efficient and easy automated technology to the music industry. Our system is an intermediary device, designed to automatically regulate the volume of an audio source depending on how much ambient noise there is in the surrounding environment.**

I. Introduction

Autonomous systems are being used all over the world to make tedious tasks disappear. From automatic thermostats regulating the heat of buildings everywhere, to the brightness on smartphones adjusting the screen brightness depending on the light in the surrounding environment. Another nuance that many people find themselves facing is volume control. Whether it be turning up the volume on a TV or turning up the music on a speaker, someone is always adjusting the volume of an audio source depending on the ambient noise in the surrounding environment. The goal of the system in this paper, that we will refer to as our TurnUp device, is to automate that volume adjustment for everyone.

The development of "smart" homes has been a popular area of research recently. There are a wide range of ways to connect devices like phones, lights, and TVs in order to automate tasks. According to an IoT Innovation article, "Utilizing integrated technological systems in your home is one of the most significant new trends in digital innovation."[5] Our dynamic volume controller would add another element of automation to the smart home.

There have been similar volume controllers created, but they are either limited in use to very particular scenarios or very expensive. One example is the TOA Electronics Digital Ambient Noise Controller.[6] This model is on the market for well over a thousand dollars and is meant for large scale environments, such as malls and airports. This kind of solution is not suitable for a normal consumer looking for a less expensive product to use in a smaller scale environment. Another product that is currently available is the International Control Systems Automatic TV Controller.[7] This system works exclusively for when the volume on the TV suddenly becomes too loud. We want to make our DVC versatile and inexpensive so that it can be a suitable product for the normal household consumer wanting to further automate his/her house.

We decided on the following requirements to make sure our design provides the best experience for users: (1)Easy to use interface (2) System will not exceed max volume setting, (3) System will not react suddenly to isolated loud noises, (4) System will function in multiple locations within desired room. We chose an iOS app as a user interface in order provide something that consumers are familiar with. The max volume requirement is in place in order to prevent an unstable feedback system, driving the system to an unreasonably high volume and possibly causing hearing damage. The system must not react to isolated loud noises because this would cause rapid changes in volume that would be undesirable to the user. Finally, the system must be able to function within multiple locations of the desired room to allow a flexible and portable system that can be easily moved if the dynamic of the room changes.

The following specifications have been created to make sure that the requirements are satisfied.

Our device will require an initial calibration by the user in the room where the audio source is present. The system would then regulate the volume of that audio source by controlling what it sends to its speaker. The setup of the system will be explained in more detail in the next section of the report.

Table 1: Specifications and requirements

Requirements	Specification	Value
UI will be user friendly	iOS app allows for easy calibration and system set up	N/A
System will not react suddenly to isolated loud noises	System will react to noise above desired ratio only after a certain time period	Sensitivity range between 0 and 8 seconds
System will be able to function in different parts of a room	System will work within a distance from audio source.	15 ft
System will not exceed max volume desired	System will not exceed max scaling value.	Max Scale value of 17

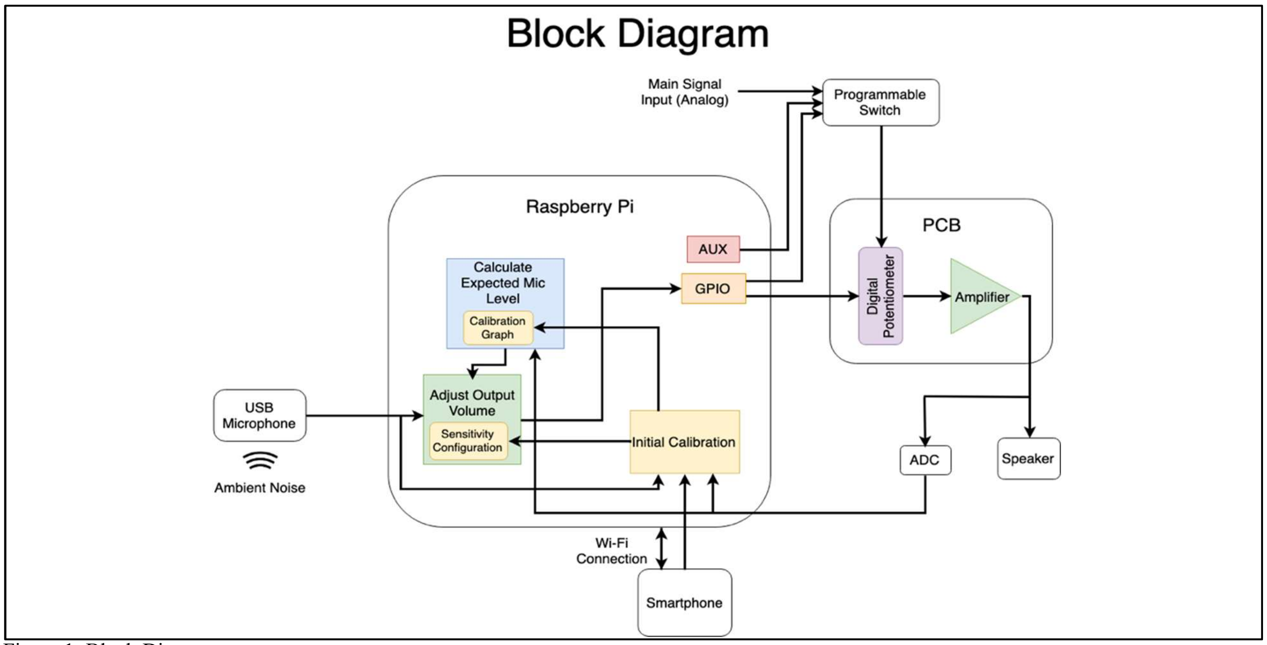


Figure 1: Block Diagram

II. Design

A. Overview

Our block diagram is shown above in Figure 1. Our basic design will be a system that lies on the path between an audio output signal source, e.g. audio from a phone or a TV, and an output speaker to play the source. The system will use an external microphone to capture the source and environment sound levels. The input sound source signal as well as the captured microphone signal will be monitored continuously over time. If the system detects a significant level of ambient noise (i.e., a large ratio of microphone signal volume to expected microphone signal volume based on the input signal), then the system will increase the output volume that is sent to the speaker until a satisfactory level of environment noise to input source signal is regained.

Our design will center around a Raspberry Pi to regulate the entire system. It will be the central hub responsible for monitoring the microphone and original audio source signals, as well as making the necessary calculations on these signals to drive the necessary volume adjustments. The Raspberry Pi will ultimately output a value that will program a digitally programmable analog amplifier, and that amplifier is what will increase the volume of the output signal sent to the output speaker.

As we considered how to build the dynamic volume adjustment system, it was clear that we needed a central processing unit that would be able to handle the monitoring and calculations of several external signals. In addition to considering a Raspberry Pi, we first considered an Arduino to handle the job. As we looked into an Arduino Uno, we saw that it primarily operated with an ATmega328P microcontroller, an 8-bit microcontroller with a clock speed of 16MHz [1]. Additionally, the Arduino had only 32 KB of flash memory and 2 KB of SRAM memory for runtime data.

Lastly, the Arduino's I/O interface includes only 14 I/O pins [2]. We determined that for the live runtime calculation necessary for our system, we would definitely need a device with higher processing power than that of the ATmega328P, and much more memory than that of the Arduino. Additionally, the volume adjustment system design requires several I/O peripherals (microphone, audio source, output signal) to correctly run, and the limitation of 14 digital I/O pins would be a hindrance to try and work a solution around. That is why we turned to a much more computationally capable device—the Raspberry Pi 3 Model B+. The Raspberry Pi operates on a 1.4GHz 64-bit quad-core Arm processor. It also contains 1 GB of SDRAM memory [3]. These processing and memory specifications are much more suitable for the level of processing that we will require for live-time audio monitoring. Additionally, the Pi includes 4 USB 2.0 ports and a 4-pole stereo output port [3], which is suitable for the necessary external peripherals that we must connect to the DVC. A bonus to the Raspberry Pi is that it contains built-in Bluetooth and Wi-Fi capability [3], which allows us to expand out to wireless connections with ease; this is a positive, as we want to use wireless microphones and connect the system wirelessly to a smartphone application.

Our design will also utilize a digitally programmable analog amplifier to control the amplification of the output sound signal. We had originally planned to simply digitally scale the output signal in software running on the Raspberry Pi before the signal was output. However, as we began working on the DVC system, we discovered that digitally processing and outputting live audio through the Pi resulted in poor quality audio. This led us to rearrange the design so that the input signal is also routed straight to an analog amplifier, which will be programmed by I/O pins on the Pi itself, which we expect to result in much higher quality audio.

As is shown in the block diagram, the Raspberry Pi is at the center of the design, responsible for taking in the audio source signal, the microphone signal, mobile device information, and running central computing modules to control the DVC system. The mobile device block refers to a smartphone application that will allow the user to connect to the system and control certain settings. The amplifier will be responsible for modifying the volume of the output signal.

B. Initial Calibration

The function of the initial calibration stage is to run a calibration process that will give the system a sense of the expected microphone pickup signal intensity (volume) given a certain input signal intensity (volume). This is necessary, as the TurnUp system is designed so that the connected microphone can be at a variable distance from the central system, and the system may be placed in a wide variety of environments, which will result in entirely different signal volume responses. The reason that we specifically want this expected microphone intensity vs. input intensity and relationship is that if we can determine the expected microphone pickup intensity over a chunk of time from an input signal, we can compare that expected intensity to the actual intensity that the microphone is observing. From that comparison, we can determine if the actual microphone intensity is significantly higher than the expected intensity (i.e., a presence of ambient noise), and if that is the case, we can increase the output signal volume to combat the ambient noise.

The user is able to run the calibration process through a smartphone application, which is described in section C. When the calibration process starts, the audio source to the speaker is first switched to the Raspberry Pi aux port by setting the programmable switch using the GPIO pins. The details of how the programmable switch and GPIO pins work is covered in section F. Next, the system plays a constant 440hz tone signal from low to high volume from the aux port to the output speaker that lasts about 10 seconds. While the tone is playing, the system will record the relationship between microphone pickup intensity and input signal intensity. The intensities of each are calculated using the `rms` function available in the `audioop` python library, which calculates the root mean square (RMS) over a chunk of values [8]. RMS values are also what we refer to as intensities. Once the signal finishes playing, the audio source to the speaker is set back to the main analog input by again setting the programmable switch. In the end of the calibration stage, the expected microphone intensity vs. input signal intensity function will be stored internally, so that it may be used later in the "Calculate Expected Mic Level" stage, which is defined in section D. After some testing, we discovered that this relationship forms a linear function ($y=mx+b$), as exhibited in an example in Figure 2.

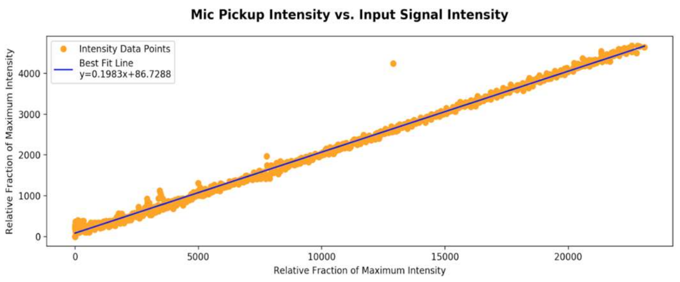


Figure 2. Microphone Pickup Intensity vs. Input signal Intensity Example

We can exploit this knowledge of a linear relationship in order to efficiently store the function and use it for later calculations. We now use a linear fit function from the `numpy` python library to determine the slope m and the y-intercept b of the data, which are only 2 variables we need to keep to efficiently model the entire relationship [9]. Then, given the intensity of an input signal, we may calculate:

$$\text{Expected Mic. Intensity} = m \bullet (\text{Input Signal Intensity}) + b \qquad (1)$$

The values of m and b are stored in variables within the main script at runtime, so that they may be used during the current run of the system. These variables are also stored in a JSON (JavaScript Object Notation) file [10]. This is so that previous calibration settings may be loaded and used, so that the user does not have to re-calibrate the system each time the system is rebooted.

Our team tested calibrating the system using a variety of different tones. We ran these tests in order to determine if the 440Hz tone would result in a graph that was good enough to handle a diverse signal. The first alternate means of calibration involved using filtered white noise and pink noise. This test resulted in the calibration graph having a steep, sudden incline in maximum intensity within the last few seconds of recording. As a result, the best fit line resulting in a graph that cause the system to overestimate the relative microphone intensity. One reason for this steep intensity graph could be that the frequency response of the microphone we were using could not handle filtered white or pink noise at the highest amplitude used for system calibration. Next, we tried calibrating by playing 220Hz, 440Hz, and 880Hz all at the same time. The resulting graph was very similar to the graph that is generated by simply playing the 440Hz tone. As a result, we continued to calibrate the system using the 440Hz tone.

C. Smartphone Application

The iOS application is the interface that the end user interacts with in order to properly set up and start using their TurnUp device. The motivation behind a mobile application for system setup is that just about everyone owns a smartphone, and it provides a way to walk a user through setup by displaying one simple instruction at a time on a screen; this is generally a more attractive method than presenting a user with an instruction manual. The application contains the following five main phases to walk a user through system setup: a welcome page, a device discovery phase, a calibration phase, a sensitivity setting phase, and summary/start listening phase. The application is written in Swift, and it communicates with the DVC system via a Wi-Fi connection.

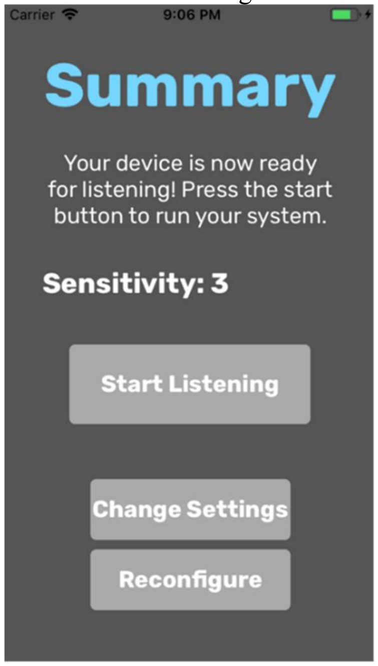
The welcome page is a simple "start" page that simply greets the user and presents them with a button that bring them to the device discovery phase in order to begin system setup.

The device discovery phase allows the user to scan the network for any TurnUp devices so that they may connect to and control it. This is necessary, as the IP address of the TurnUp device on the user network cannot be known in advance. Therefore, we must run some sort of discovery process in order to find and initiate communication with the device. When the TurnUp device is on and running, it runs a UDP (User Datagram Protocol) server on a predetermined port number, 8156. During device discovery, the user's phone sends a UDP broadcast message to the predetermined port 8156. UDP broadcast messages are sent to every device on the network, which allows us to initiate communication with the device without initially knowing its IP address. The broadcast message from the phone is a JSON file that contains the message type ("discovery") and an arbitrarily chosen port number to which the TurnUp device may send a response. The user phone then immediately opens a TCP (Transmission Control Protocol) server on that chosen port to listen for device responses. Upon receiving a UDP message, the TurnUp device will try to unpack the message as a JSON file, and then check if this is indeed a discovery message by checking the type. If it is a discovery message, it will extract the response port number from the JSON data; the device also now knows the user phone's IP address by checking the address field of the UDP message. The device now constructs a discovery response JSON message that includes the device's name and a new arbitrarily chosen main port to which the user may send all future data. The device now sends this message to the response port of the user phone, and it opens up a TCP server on the chosen main port. This new server now allows the user phone to initiate TCP connections with the TurnUp device at will, as TCP is a more reliable form of data transfer than UDP. The user phone receives the response data from the TurnUp device, and it extracts the device name and main port number from the JSON data. The device IP address can also be extracted from the TCP data. The app then loads a table of device names from devices that have responded to the discovery message; this allows a user to choose a specific device should they have multiple devices on their network. A button allows a user to rerun discovery if necessary just in case the device does not initially respond; the phone listens for responses for two seconds before reloading the device name table. After this, the user is able to select the appropriate device from the table and continue on to the calibration phase. From this point on, the phone is able to communicate with the TurnUp device by sending TCP messages to the IP address and main port number that have been obtained in this phase.

In the calibration phase, the user is presented with two buttons: a "Calibrate" button and a "Load Calibration" button. The "Calibrate" button sends a message to the device that instructs it to run the initial calibration process that has been described in section B. The "Load Calibration" button sends a message to the device that will instruct it to load a previous calibration graph if the device has been set up before, so that the user does not have to recalibrate. If the device cannot successfully find and load a previous calibration graph, the user is presented with an error message stating that recalibration is necessary. Upon successful calibration, the app continues to the sensitivity setting stage.

In the sensitivity setting stage, the user is able to select a sensitivity for their device using a slider; sensitivity is chosen on a scale of 1-5. A brief message is displayed to the user explaining that this setting will determine how quickly their system will react to significant ambient noise. The specifics of how sensitivity affects system functionality is covered in section E. After selecting sensitivity, a message is sent to the device containing the chosen sensitivity, and the app moves to the summary/start listening phase.

The summary/start listening phase is the final stage of the iOS application. This page tells the user that they have successfully set up their device, and that they may begin monitoring noise by pressing the "Start Listening" button. The chosen sensitivity setting is also displayed for the user. This is all shown in figure 3. Once the "Start Listening" button is pressed, a message is sent to the device that instructs it to run the main listening code that monitors ambient noise, which is described in the sections below. After pressed, the "Start Listening" button turns into a "Stop Listening" button, which the user may press if they want to stop the live monitoring of ambient noise; this sends a message to the device that commands it to stop the main listening code, and to stop amplification, if any. On the summary page, there is also a "Change Settings" button that allows the user to choose a new sensitivity, and also a "Reconfigure" button that allows the user to restart the entire setup of their device from the beginning if needed.


Summary

Your device is now ready for listening! Press the start button to run your system.

Sensitivity: 3

Figure 3. Summary page screenshot

The core of the iOS application was developed over the period of about two months. The app's visual appearance and user flow was refined over the period of about two additional months according to feedback a number of people that were kind enough to test and review our app. This helped to ensure that the overall application design was user friendly.

D. Calculate Expected Mic Level

The calculate expected mic level block calculates the expected microphone intensity given the input signal intensity. This calculation happens continuously over time as the system is running as a means to live-monitor the sound level of the surrounding environment. This block is implemented entirely within software on the Raspberry Pi.

As mentioned previously in section B., the expected microphone intensity vs. input signal intensity function is effectively stored and modeled in software by the two values *m* and *b* that describe the linear relationship. As chunks of an input signal come in, the intensity over that chunk is calculated using the RMS function. Next, the expected microphone intensity is calculated by using the input intensity as input to Eq. (1). This expected microphone level can then be further used in the adjust output volume block to determine if the output volume should be appropriately adjusted.

E. Adjust Output Volume

The next step in the core program of the TurnUp system is adjusting the output volume as necessary. This part of the software is the heart of the functionality of the volume control system. The output volume of the system is controlled by a digitally programmable potentiometer that the input signal travels through on its way to the output speaker. The details of how the potentiometer is controlled and how exactly it affects the input signal are discussed in section F; for now, we simply acknowledge that this is the manner in which we control output volume.

The main algorithm that controls the system volume adjustments is modeled by the finite state machine in Figure 4.

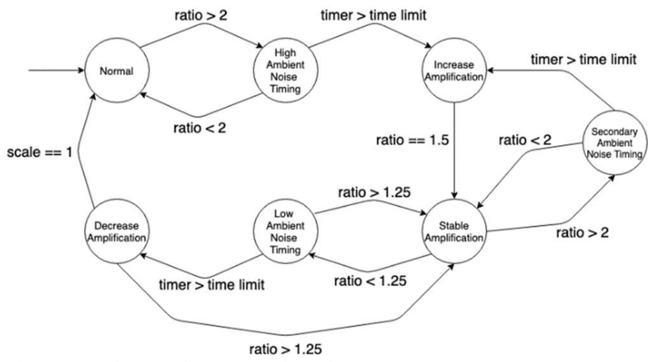


Figure 4: Volume Adjustment Algorithm FSM

Essentially, the device starts in the normal state where there is no amplification and no significant ambient noise. Over time, the system is constantly calculating the intensity of the signal picked up by the microphone, and the intensity of the sound signal that goes to the speaker, each over chunks of 1024 samples at a time as they come in. The chunk size of 1024 was chosen after testing a few different values, as it was discovered to be a good balance of a long enough window so that most of the samples for each calculation will overlap should there be any delay between the reception of the input signal and microphone signal, and it is also reasonably short enough that the system will react in a timely manner. Additionally, window sizes that are powers of 2 work most efficiently in memory. The intensities are calculated using the same RMS function of the `audioop` library that is used in calibration.

After this, the expected microphone intensity is calculated as a function of the using the method described in section D.

Next, the moving averages of the microphone signal intensity and *expected* microphone intensity are each calculated using the following moving average equation:

$$\text{Moving Avg.} = (\text{Old Moving Avg.}) \cdot 0.9 + (\text{new value}) \cdot 0.1 \qquad (2)$$

This weighted moving average is designed so that the newest intensities will contribute only a small amount to the average. This way, a brief, sudden ambient noises should not immediately trigger a reaction, instead the noise must persist for some time to build up the intensity moving average in order to trigger necessary reactions. Weights of 0.9 and 0.1 were found give satisfactory performance results after considerable testing.

Next, the ratio of the average microphone intensity to the average expected microphone intensity is calculated. This is the ratio that the FSM is checking as the system operates.

If the ratio is greater than the constant threshold of 2, then the FSM moves into a high ambient noise timing state. In this state a timer begins, and if the FSM remains in this state until either the ratio drops back below 2, or until the timer surpasses a set time limit. The time limit is determined from the user selected sensitivity setting as shown in the table 2.

Table 2:
Sensitivity

Sensitivity	Time Limit (s)
1	8
2	6
3	4
4	2
5	0

These time limits were determined after testing a handful of different time limit ranges. Sensitivity of 1 corresponding to 8 seconds was determined to be a reasonable low sensitivity response time based on our experience, and sensitivity of 5 corresponding to 0 seconds (immediate response) was also determined to be reasonable.

When in the timing state, if the ratio drops below two before the timer expires, the system will go back to the normal state. Otherwise, upon the timer expiring, the system moves to the increase amplification state. Here the system will drive up the output volume until the ratio has reached a goal value of 1.5, where it will move to the stable amplification state. The goal ratio of 1.5 was chosen because we want the amplified signal to remain at a stable volume for a reasonable amount of time, and goal values closer to the ratio of 2 resulted in more frequent back-and-forth changes between stable amplification and increase amplification.

The system will remain in this state while the ratio remains between 2 and 1.25. If the ratio rises back above 2, it will go to the secondary ambient noise timing state, similar to the original timing state, where it will again time to see if it should return to the increase amplification state, or go back to stable amplification. On the other hand, if the ratio drops below 1.25, the system will continue to the low ambient noise timing state, where timing before decreasing amplification will occur. The ratio of 1.25 was determined after testing as a reasonable goal before decreasing amplification, because a such a low ratio is a good indicator that the ambient noise may have subsided.

In the low ambient noise timing state, the system now checks if the ratio remains below 1.25 for the time limit, where it will then move to the decrease amplification state. The system will jump back to stable amplification should the ratio rise back above 1.25 before the timer expires.

Finally, as the name suggests, the output volume amplification will decrease while in the decrease amplification state. If the ratio rises back above 1.25 while there is still some amplification (scale > 1), the system will go back to stable amplification. Otherwise, if the system keeps decreasing amplification until the scale reaches 1, that means the system is no longer amplifying, so the system returns to the normal state. The scale relates to the potentiometer value, which will be explained in more detail in the following section.

F. GPIO, PCB and Analog Switch

These blocks make adjustments to the signal sent to the speaker. We use the built in GPIO pins on the Raspberry Pi to send a string of bits to an amplifier with a digital potentiometer at its input soldered to a PCB seen in Figure 5. The GPIO pins from the Pi set the input signal to the amplifier based on the scale factor calculated in the previous module.

In order to first test this block, we created the circuit on a breadboard and manually changed resistance of a mechanical potentiometer to make sure we can achieve the desired range of signals from the circuit. Once we confirmed the range of resistance needed from the potentiometer (10kOhm), we included the model that best suits this specification on our breadboard. Ultimately, we chose to use MCP41010 Digital Potentiometer and LM481n-4 Audio Amplifier for our circuit design. The MCP41010 model uses the Serial Programmable Interface pins on the Pi to communicate with the rest of system. [4]

The MCP41010 potentiometer can be set to 256 discrete resistance values in the range of values 0Ω-10kΩ. The higher the resistance value of the potentiometer, the more the signal will be diminished, so the lower the output volume will be. Therefore, we can control the output volume by assigning a high base potentiometer value, and if amplification is necessary, the potentiometer can be assigned a lower value to achieve a higher output volume.

We tested the entire circuit with the rest of our system to make sure that our speaker received the correct range of amplitudes before sending out our PCB design to the manufacturer.

After receiving and testing our PCB, we encountered a problem with input signals to the PCB. During calibration, our system would read input signals much lower than it should have at the ADC at the input to the Pi. We later determined that this was because that the two inputs to the PCB were seeing different input impedances and therefore, the signal going through the PCB and to the Pi and speaker was different depending on which audio source it came from. We solved this problem by integrating an analog audio switch at the input to the PCB and using separate GPIO pins on the raspberry pi to switch between the calibration signal and audio source. This ensured that both sources saw the same impedance and ultimately solved the problem.

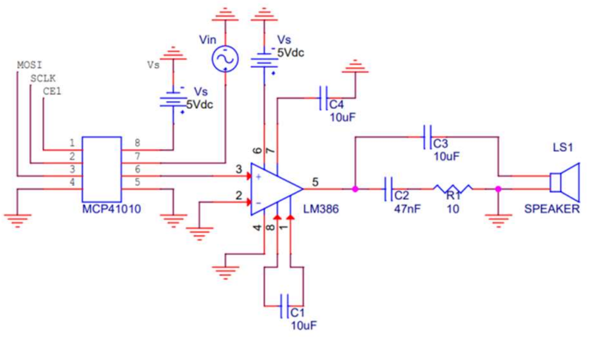


Figure 5: PCB

G. Enclosure

The final step before the completion of the project was to design an enclosure to house the various components of the TurnUp system. The main design goals for the enclosure were that it was to be compact, portable, and adequately protect the working components that it housed. To accomplish we settled a sort of double clam shell design with a middle section for mounting and a lid on either side to close up the system.

As seen in figure 6 on the next page, one side of the middle section the raspberry pi is mounted while on the reverse of that same section the PCB and breadboard containing the programmable switch are also mounted. A cut out allows GPIO and power wires to connect the components on each side of the section. Once the two lids fastened with hinges are closed the system becomes a streamlined and compact final product.

In addition to accomplishing our original design goals for the enclosure we achieved the added benefits of increased reliability and increased audio fidelity. With the components now rigidly mounted in an enclosure, and all audio wires being directly soldered, the amount of distortion and signal noise caused by loose and easily movable wire connections will drastically be reduced. The enclosure was designed in Autodesk Fusion 360 before being 3d printed at Umass digital media lab using the fused filament fabrication process. The resulting enclosure provided us with a cost effective, but a portable and protective housing for the TurnUp system fit for commercial release.

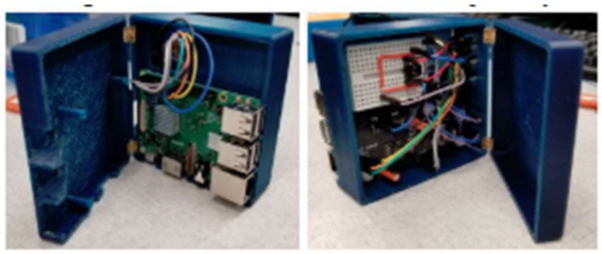


Figure 6: Enclosure

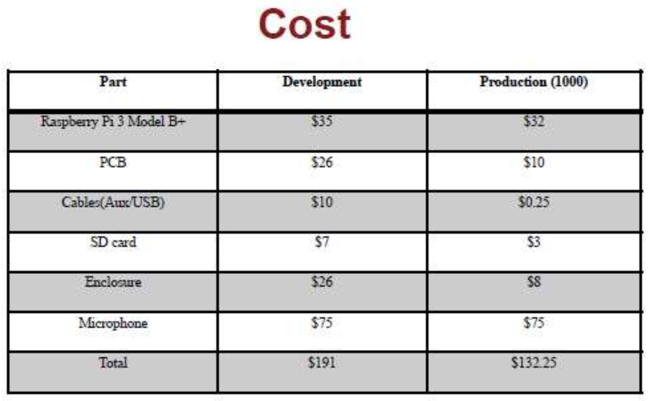
III. Project Management

Our team consists of three Computer Systems Engineering students and one Electrical Engineering student. This means that our group has a strong software background. The different roles of our project have been split up according to the abilities of each member. The chart below shows the responsibilities of each team member.

The core software design was been split up between all 4 group members. This was the largest, and most time-consuming part of the project. PCB design was the responsibility of Harry, as he is the electrical engineer on the team. Ryan had some experience in app development and therefore handled the iOS app for our project. The other software related parts of the project were split up evenly between Nicholas and Rahaun, as they are computer systems engineers.

To keep a workflow and schedule, our team had weekly meetings amongst ourselves and with our advisor. The communication between our group consists of e-mail, text, and in person meetings. Ultimately, we took a very unified approach towards completing our project. This means that we had been working on our portions of the project together rather than combining individual work every week. This ultimately worked well for the completion of our project.

Finally, we put together a table for the cost of our project both in terms of development stage and production stage. We based our production costs off of bulk supply prices online, typical manufacturer bulk pricing for PCBs and the cheapest options we could find for single items online. We estimated that our production cost would be around $132.25, which is much better than other controllers in the market mentioned earlier, such as the TOA Electronics Volume Controller.



Cost

Part	Development	Production (1000)
Raspberry Pi 3 Model B+	$35	$32
PCB	$26	$10
Cables(Aux/USB)	$10	$0.25
SD card	$7	$3
Enclosure	$26	$8
Microphone	$75	$75
Total	$191	$132.25

Figure 7: Cost

IV. Conclusion

Since MDR, we have added a few key features to our project. We integrated our final PCB design that was described early in the paper. The switch used to send the signal to either the Pi or analog output was also integrated. This was an important step in our design as it solved many of our signal issues. Furthermore, the FSM described earlier was designed and uploaded to the final draft of the code. This FSM ended up being a key feature of our design as it made the system react in the smooth manor we hoped for. Finally, the user interface was updated from the previous version and the enclosure for the system was finished. Our team achieved our goals and successfully created a volume control device.

While the project was completed there are still some aspects that could be improved. One improvement would be to move the code over to an embedded system that could process the code without having to fun an entire OS in the background. Another change that could be made is to write the code in C rather than Python. While our python code works with our design, the project seemed to have trouble running long term without needing to be rebooted. One cause of this problem could be due to memory overflows that could be worked around in C, however, the problem needs to be further investigated. Also, our algorithm for calibration and adjustment works for our system design, however, testing with different hardware and changing our adjustment algorithm could potentially lead to a system that reacts smoothly for all types of audio input. Finally, more powerful peripherals, such as a 360˚ microphone with a higher frequency response could improve the system.

V. Acknowledgment

Team Turn Up would like to thank Professor Pishro-Nik for his guidance on this project. A special thanks to Professors Kelly and Ganz for all of their helpful advice and feedback. Thank you, Fran Caron, for providing us with a workspace and tools. Finally, we would like to thank Professor Hollot, Professor Soules, and Shira Epstein for their continued support.

References

[1] Ww1.microchip.com. (2018). ATmega48A/PA/88A/PA/168A/PA/328/P Data Sheet. [online] Available at: http://ww1.microchip.com/downloads/en/DeviceDoc/ATmega48A-PA-88A-PA-168A-PA-328-P-DS-DS40002061A.pdf [Accessed 20 Dec. 2018].

[2] Store.arduino.cc. (2018). Arduino Uno Rev3. [online] Available at: https://store.arduino.cc/usa/arduino-uno-rev3 [Accessed 20 Dec. 2018].

[3] Raspberry Pi. (2018). Raspberry Pi 3 Model B+ - Raspberry Pi. [online] Available at: https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/ [Accessed 20 Dec. 2018].

[4] C. Wells, J. Becker, *Low-Cost Digital Programmable Gain Amplifier Reference Design.* Texas Instruments, 2015.

[5] IoT Innovation. (2018). *The Impact of Smart Homes Technology | IoT Innovation*. [online] Available at: https://internet-of-things-innovation.com/insights/the-blog/smart-homes-technology-impact/#.XBw691VKjIU [Accessed 21 Dec. 2018].

[6] Toaelectronics.com. (2018). *Products - TOA Electronics*. [online] Available at: http://www.toaelectronics.com/products/audio-signal-

processors/dp-l2-digital-ambient-noise-controller [Accessed 21 Dec. 2018].

[7] Amazon.com. (2018). [online] Available at: https://www.amazon.com/International-Controls-Systems-TVSR-Automatic/dp/B000Q37TBY/ref=cm_cr_arp_d_product_top?ie=UTF8 [Accessed 21 Dec. 2018].

[8] “21.1. audioop - Manipulate raw audio data,” 21.1. audioop - Manipulate raw audio data - Python 2.7.16 documentation. [Online]. Available: https://docs.python.org/2/library/audioop.html. [Accessed: 14-May-2019].

[9] “NumPy,” NumPy. [Online]. Available: https://www.numpy.org/. [Accessed: 14-May-2019].

[10] “Introducing JSON,” JSON. [Online]. Available: https://www.json.org/. [Accessed: 14-May-2019].