Dynamic Volume Adjustment Module

A Major Qualifying Project Report:

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the Degree of Bachelor of Science

By:

Sam Shurberg   Sam List   Anthony Ratte   Stephen Blouin

Date: April 27, 2017

Approved:

Professor Berk Sunar, Advisor

Keywords:

1. Dynamic Volume Adjustment
2. Digital Signal Processing
3. Signal-to-Noise Ratio

ABSTRACT

The Dynamic Volume Adjuster Module is a piece of consumer audio hardware capable of automatically adjusting a sound system's volume to remain louder than the noise of the environment. It accomplishes this by monitoring the sound in the environment and comparing the feedback from the sound system to the total sound level. The project focused on signal processing using a specialized DSP Microcontroller and a selection of peripheral components.

Table of Contents

Table of Figures

1 Introduction and Problem Statement

When watching television, or listening to music or the radio, there are many signals that can interfere with a listener's experience. Whether the interference is uncontrollable noise, the wind blowing, a neighbor's dog barking etc., or controllable noise, someone vacuuming in the same room as the user, people talking, etc., these sounds detract from the ideal experience that the user is looking for. Each one of these noise sources has a distinct sound texture, duration, amplitude, and period. Each of these characteristics can cause different types of interference against the desired audio that the user is looking for, making it harder to hear and understand the information being broadcast. The easiest way to combat this is to increase the volume of the audio source, but the user does not want to have to constantly change the volume as the noise in the room changes. Instead, an automatic system could be implemented to listen to the ambient noise, and adjust the volume of the desired audio source such that the signal to noise ratio is above the desired threshold. In other words, as the noise in the room starts to interfere more, the audio device will automatically adjust the volume of the signal the user wants to hear to counter the disrupting signal. Then, if the disrupting signal goes away, the signal will automatically adjust back down to a comfortable level.

1.2 Proposed Solution

The solution that the team has come up with is a system that exists between an audio source and its speakers. The system will take an audio signal in and modify this signal based on the ambient noise that the microphone picks up in the room. The goal of the modification is to create a sound signal that can overpower the noise in a room so that the desired audio is easily understandable over the noise. The device does not attempt to implement any noise cancelling algorithm; the device instead attempts to drown the noise out base on an input Signal-to-Noise Ratio (SNR or S/N). This system will also be modifiable to do other similar things to the user's listening experience, such as reducing its volume to below the level of the ambient noise of a room so that someone can talk over the audio without having to change the volume level, or normalizing the volume of a television set when overzealous companies amplify their commercials much more than the program the user is watching.

1.3 Market Research

The method used to establish the market for Hush was to do some cursory Google searches to determine the general market. Search engines and databases such as IBISWorld, EBSCOhost, and Google Scholar were utilized to gather some more specific information. The team also polled some of the people in this market to understand the needs and wants of our market. People were asked if they were interested in a product that would deliver the same quality of sound as common speakers, but could adjust the volume according to external noises as well. In addition, a sample of WPI students were asked the following questions:

- What are some things you generally look for in a speaker?
- How durable would you like it to be?
- Would you rather have a wired speaker, or one that requires batteries?

The background information found indicated that in the audio player market, 53% of products are speakers/speaker systems, providing a large market to sell to. From this market the age range of those who would be buying this product ranges from age 25 to age 54 as seen in Fig. 1 (IBISHost). This range is anyone who is old enough to have their own apartment or house and young enough to still be interested in buying technology. The market depends on households that have disposable incomes, and is expected to grow in the future, making it a viable market to sell to in the long run.

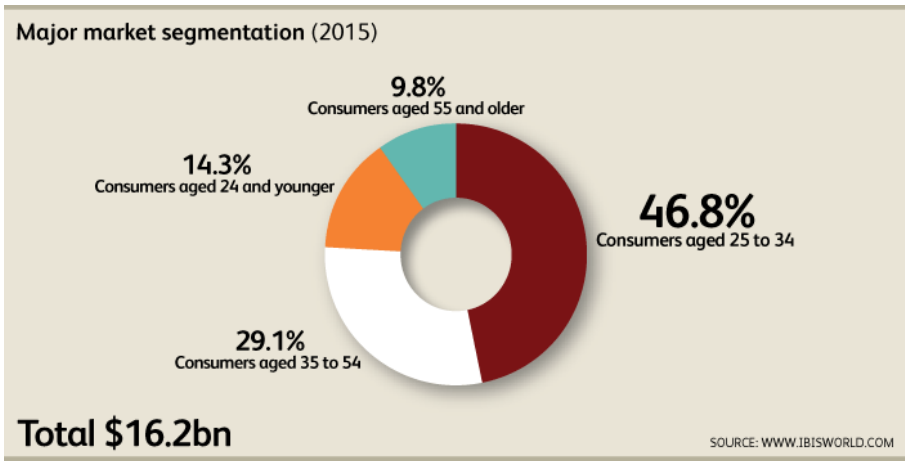

Figure 1. Market Breakdown by age bracket

Currently, there are no products with the exact same functions as the proposed device. The closest competitor to this speaker system is a speaker that automatically changes the equalization based off of the music being played. While this kind of system does change and lower the sound coming out of it, it does not provide the same utility, which is adjusting volume based on unexpected external noises. In addition, there are products on the market that are voice activated. Such products range from music players and printers to refrigerators. However, these products require specific voice commands, and can often perform an undesirable function, or fail to recognize the command at all.

Most of the popular speakers from Best Buy have a price tag of between $20 for small lower end speakers to around $300 for very high quality speakers. One main feature that the competing products do have which is most likely what drives their prices higher is that they are all wireless to some degree. While it is possible to create this ability in Hush, the team is not currently investigating this avenue, as this device is more of a black box for a speaker system, most of which are still wired. The wireless feature in speakers is on that appeals to a generally younger audience who may be constantly on the go. Due to this, the aim is to market to the older age range of the audio market because they are less likely to have newer audio technology (IBISWorld) and need devices that are compatible with their older technology.

Competing products tend to range from 15W output to 60W output depending on the size of the speaker. The speakers output can be broken into two categories, subwoofers and satellite speakers. Subwoofers are larger and make the lower sounds for the systems but draw more power, they tend to have an output of 50-60W. The satellite speakers are much smaller than subwoofers and make higher pitched sounds which require less power and use around 10-20W. An example of the different types of speakers can be seen in Fig. 2 below.

Other features that the competing products have are batteries, allowing them to be very mobile, tied with wireless capabilities. Some of the battery powered speakers are also rechargeable with either a charging system given by the company or through micro USB 3.0. We wanted to stick to a wired design as it allows for a prolonged listening experience without the hassle of changing or recharging batteries. While the product we are creating does not have these extra features, it will have the ability to reduce volume when someone comes into the room to talk to the listener. This feature is unique from all other products that we have researched, which allows us to differentiate our product from the other speakers on the market. Our target market is also on the older side, which makes them less likely to want more features that are harder for them to learn to use. The product will be an easy to use and useful device that appeals to those who do not want to mess with technology but rather just be able to use it.

It is important to note that the concept of "background noise" in terms of listening to music or other media is very subjective. Not only will different people have different ideas of how much noise would be considered intrusive, different types of noise will elicit different responses. While obvious variables include magnitude of noise and frequency, high-pitched "screeching" noises typically being worse, other variables such as periodicity, how often the noise is repeated, and whether it is constant or not. Even the importance of the noise would make a difference, people will be more apt to listen to noises such as human voice or banging that should prompt a response, even if not an actual situation, can be just as important if not more so. While the device obviously cannot be tailored to each user's exact preferences, trends can be inspected to determine the most common opinion.

The most important factor is how loud the intruding noise is. In a study determining response to environmental noise, 15% of respondents claimed their environmental noise as "highly annoying" ranging from 50 to 60dB, increasing to 25% at 70dB and 50% at 80dB [1]. While this is a useful measure of what noises are considered loud enough to be annoying, it is important to note the time and context of the study: in 1978 media was decidedly different because people did not have rooms solely for media entertainment, and noise is perceived differently while listening to media. Noise could also be easier to drown out, or more prone to distract the user from the media they want to focus on. Louder noises not only cause more distraction, they may also drown out the media being listened to. It is important to decide a threshold as to how loud a user's media will be playing in the first place: common reference points include 80-100dB for orchestral performances whereas rock concerts can push 115dB. Comfortable listening includes the 60-90dB range where 60-70dB is considered soft, 70-80dB moderate, and 80-90dB loud. Hearing damage can occur when exposed to volumes of 90-140dB over long periods of time, so raising the volume to ~100dB should be done rarely, unless the customer insists on it [2]. This is a notable obstacle as, although the first instinct is to drown out environmental sounds by becoming louder, because of this there is a risk of spiking the volume very high and irritating the users. This is obviously a problem if it gets into a “screaming match” with a constant loud noise, but also in the case of brief, intermittent noise where the media becomes frustratingly loud a few seconds after the intruding noise.

Another of the most important factors in response to noise is similarity to human voice. Humans are simply more prone to react to noises that have historically been important because of evolution, more specifically natural selection. Important frequencies include typical human voice ranges, 85-255Hz for adults, as well as common danger warnings such as screaming or a baby crying, which can range from 1000-5000Hz. Because of these important sounds, human hearing, which has a total range 20-20kHz, has the strongest response at 2k-5kHz [3]. Noises either in these frequency ranges or sharing characteristics with the noises should be given special consideration due to human propensity to pay attention to them.

Of course, not all factors regarding subjective matters can be collected entirely empirically. Different people will react to disruptions differently, so it may sometimes be best to gather informal survey responses based on what sounds potential users may find the most annoying. Samples of the noises described will be collected and analyzed for quantifiable factors that allow for better reaction from the product from that data. Extensive testing of the final product with customers listening will be desirable to ensure the list of suspect sounds is complete and that the product does not cause more harm than good by drowning out sounds which may be important to the users.

2 Product Requirements

There are a number of requirements the team had to meet in order to create a product that can compete in a very large market of consumer electronics in the music industry. Some of the requirements are customers’ requirements, others are cost restrictions, and still others are component requirements that require compatibility and correct functionality of the product.

2.1 Customer Requirements

The customer is the sole reason for the invention of an audio enhancement device. This device would be added onto a stereo system that has already been implemented in a customer’s listening space, As such, the device must retain the customer the same quality audio that they are used to, while adding to the overall experience by adjusting the volume of the audio for them. To start the design of the device, the team needed to recognize the requirements the customer needs to create an enjoyable experience while using it. There are several elements that go into a user’s listening experience.

First and foremost the customer will be looking for the quality of the sound that is output. Most users would not be willing to sacrifice the sound quality of their system, just to have the automatic volume adjustment feature. The main goal of the product is to create a uniform and undisturbed user experience, so the sound quality of the device must be equal or greater than the input system and output system. This will guarantee that if there is a dip in audio quality, it will no be due the device.

Another quality that is important is running the device in real time. Many customers not only use their stereo systems for music, where running in real time is not as important, but also for watching television and movies. When watching video, a person will notice latency if there is over a three second delay. This means that the clock on the board we choose should be fast enough to handle all of our processing without giving the customer any noticeable latency, and still be able to operate at the sampling frequency of 44.1kHz, which is the sampling frequency for human hearing, and most audio files in general.

The next customer requirement is that the device is easy to use. The best devices on the technical market are the devices that are relatively cheap to buy, but also just work appropriately out of the box with minimal setup. Ideally, the device would be a “plug and play”. This is essentially a small device that takes one or more inputs, and manipulates the inputs to create a desired output. So, ideally all the customer would have to do to use this device would be to plug the auxiliary cable from their phone to the new device, and then from the device to the stereo system. The device would have a brief set up procedure, and then work without having to touch it again. In other words, the customer doesn’t care how the device works, they just care that it works as advertised. This autonomy would bring the device from a good product to a great product.

The final main requirement that the customer may not even know they need is that the device works in any size room. Depending on the size of the room, the signal that the microphone on the device picks up, may be louder (small room) or softer (large room). This is because of the multipath loss involved in wireless signals. On average, a signal will lose 3dB of sound over every ten meters. So, if the device is in a large room, the microphone will pick up a much softer signal than if it was placed in a small office or a room of that nature. The device must calibrate itself at some point because of the nature of these audio signals. This calibration is essential in allowing the device to be used in any environment.

There are some of the other requirements that are secondary to the ones already discussed, as they are important for every device that is designed. One of these requirements is size. The device must be small and compact to allow for it to be hidden away, or conform to the stereo system, and not stand out in a room where style and fashion are important to a lot of people, while also having functionality. Another of the requirements is cost. Our device must not be cost prohibitive. Because this device would be one-of-a-kind on the market, the device may still sell for upwards of one hundred dollars if the quality is high enough. However, it is doubtful that anyone would buy the device for over three hundred dollars. So, the cost to produce the device must be kept as low as possible.

2.2 Product Specifications

Based on the customer requirements above, the team had to decide on a general design for a system that would meet all the customers' needs.

- Preserve Sound Quality
- Real-Time Operation
- Ease of Use
- Utility for Cost

The first criterion was that customers demand a high-quality sound output from the device. The quality of the sound depends on the resolutions of the Digital-to-Analog Converter (DAC) and the Analog-to-Digital Converter (ADC). The two choices for the DAC and ADC were either sixteen bit or twenty-four bit. Compact Disks are typically sampled at sixteen bits. Compact Disks are typically better quality than what most people use for their music files, although recently with the emergence of streaming services, the quality of music has started to increase. According to a study done by Anthony Cocciolo, Columbia University, "Students [people] have difficulty discerning the difference between oral histories at archival quality (24-bit/96 kHz) and CD quality (16-bit/44.1 kHz) [4]. Another issue that arises when using 24 bit DACs and ADCs is that 24 bit uses significantly more storage than 16 bit over a longer period of time 24 bit. Over a minute period

24 bit will require roughly 10Mbytes of storage, while 16 bit will require about 85Kbytes at the highest quality [5]. Because of these two factors, the team decided that 16 bits would be sufficient for this prototype.

The next quality that is important for the customer is running the device in real time. In order to accomplish that, the clock crystal on the board that is chosen must be fast enough to process the entire algorithm before a new sample is taken from the ADC. The original estimation was that a 200MHz clock would give the team plenty of time to process the algorithm, and therefore give the customer a quality listening experience, even when watching videos. A 200MHz clock would result in about 4500 clock cycles to get all of the processing done.

The easiest devices to use in the technology world are “black boxes.” A black box is a device that the customer can just plug in, and it will work exactly as expected. To create a black box, an initialization step was added into the algorithm to detect the gain in a given room. This gain analysis should allow the owner of the device to plug the device in once, and the system will be able to analyze how noise reacts in a noiseless room by analyzing the difference between the output signal and input signal. This black box functionality also allows the customer to use the device in any environment. If the device is placed in a large room, it will realize that the gain is smaller, and therefore increase the volume of the speaker accordingly, and the music will make larger changes when the noise grows louder. If the room is smaller, the gain will be higher, therefore telling the device that the room is smaller, therefore smaller changes in music, as to not completely overpower everyone inside the small room.

The final requirement from the customer is that the cost of the device be low. As of right now the price is a little bit high, but that is because many developers’ components are being used to allow for better debugging of the system. If a Printed Circuit Board was designed from the beginning, there would be bugs in the hardware system as well as in the algorithm, so the team decided a developer’s board with a working microphone and speaker would be best for the testing stages. When the device is taken to the refining stages after a proof-of-concept, the cost could be easily five-percent of what it currently costs, and a fifty to one-hundred-dollar price point is realistic.

2.3 Selected Components

A set of parts to be used for prototyping was compiled after outlining the desired criteria of the components for the device.

2.3.1 Processor/Board

The TMS320C6713DSK evaluation board was selected for processing. This board is built around the TMS320C6713 Digital Signal Processor, which itself was chosen due to its clock speed of 225MHz, which makes it optimal for real-time signal processing. This processor was also preferred because it is programmable with C/C++ through the Code Composer Studio application, which the team was experienced with. Another key factor of the evaluation board is its AIC23 codec, which is used to perform both ADC and DAC functionality. The ADC and DAC are critical for working with audio signals. Its resolution is 16 bits which is suitable for this application. The board possesses 8MB of RAM which is needed for hysteresis in the volume control algorithm, and 512KB of flash memory which is sufficient for the program. Furthermore, the board possesses four 3.5mm jacks connected the codec, which makes them accessible for manipulation by the processor. In a production environment, only the critical components would be used. The AIC23 codec is a viable candidate for this, although in the future 24 bit resolutions may also be considered. Overall, the TMS320C6713DSK features allow for the board to fulfill the needs of the product, at least in a testing environment. Thus, this board was selected, as extra boards were readily available.

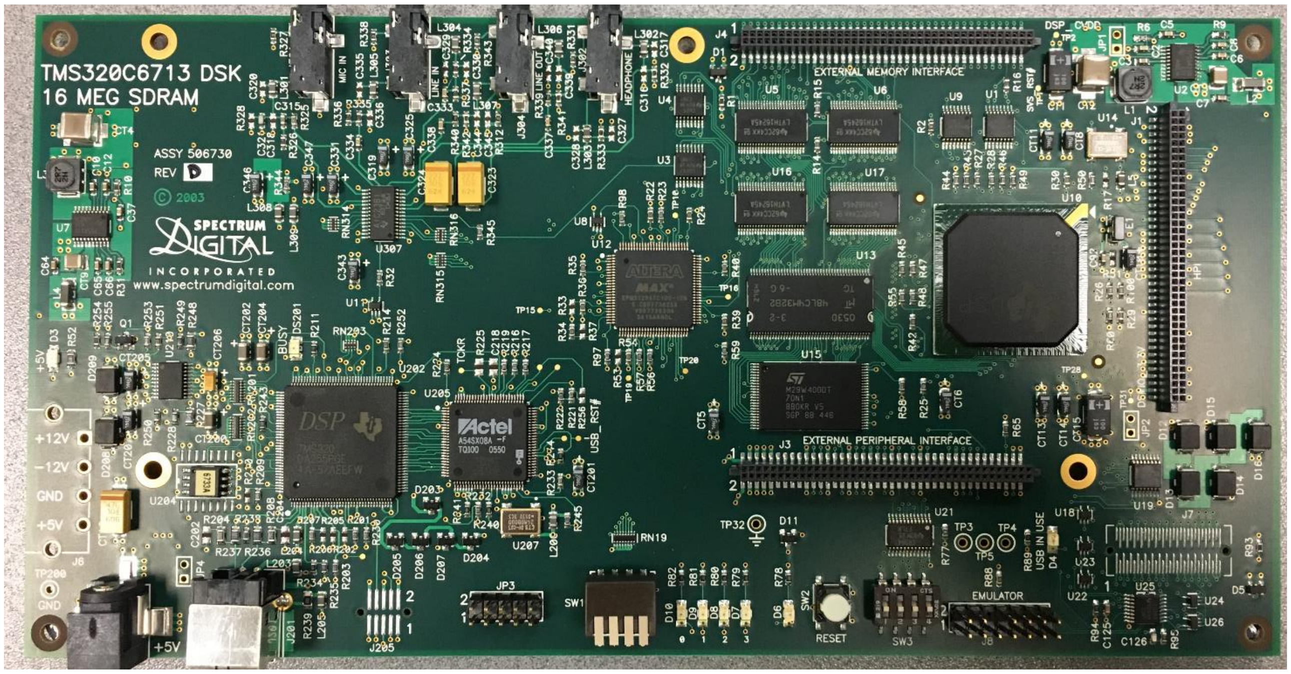

Figure 2. TMS320C6713DSK DSP Board.

2.3.2 Microphone

The original microphone selection was a stereoscopic condenser microphone from Soonhua which allows it to interface with the board. A stereoscopic microphone is preferable to a directional microphone as the goal is to gauge ambient noise in a whole room, but in the future a 180-degree or even 120-degree microphone may be considered, as most customers will have the device against a wall. A condenser microphone is preferable to a dynamic microphone, as it has stronger response to higher-frequency noise as well as louder noises which are both important for this device. As the device moves into production a directional microphone will be a more viable option because it will most likely be smaller, but still pick up the important sounds in the room.

Figure 3. Soonhua Condenser Microphone.

The problem with the microphone that was originally selected was that it was an electrolytic microphone with no amplifier on board. The development board being used could not pick up the signal from this microphone because it was much too weak. To combat this, the group had to choose a new microphone that fit all of the credentials listed above, along with an amplifier.

Unfortunately, the Soonhua microphone alone was not sufficient for the application. The microphone does not come with an amplifier on board, and the DSP Board does not amplify its inputs, so the input magnitude to the DSP was too small. To remedy this, a microphone amplifier is required. For continued testing, a higher-quality microphone from Behringer was obtained. Despite being a dynamic microphone, it provided a more reliable signal. The microphone does not, however, have an internal amplifier, so one was still required for testing. An Alto Professional XMX52 mixer board was used. This device can be used to adjust the scaled output of the mic as needed for testing purposes. For production purposes, a simple amplifier circuit would be used to reduce cost and size.

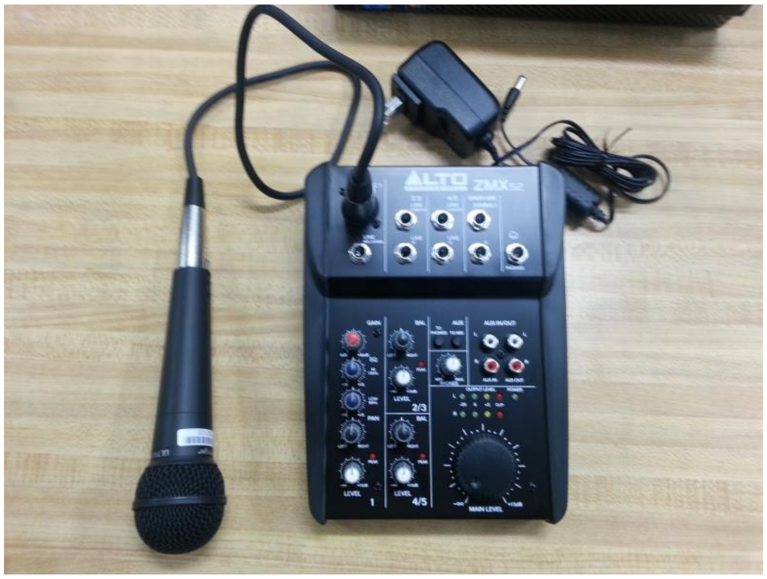

Figure 4. Behringer Dynamic Microphone and Alto Mixer Board.

2.3.3 Speaker

An Anker portable speaker was selected for the testing speaker. The specifics of this component are not important as the production model will be used with the customer's speaker, but a speaker was still needed for testing. This speaker was selected for its portability, as it is much easier to test with versus a large home system quality speaker, quality of sound, as reasonable fidelity is important for the feedback-reliant volume control program, and 3.5mm compatibility with the board.

Figure 5. Anker Portable Speaker for Testing

2.4 Economic Considerations

The prototype does not use the most cost-effective components, but rather the ones easiest to test functionality with to eliminate any unnecessary unreliability in the system. This includes features such as real-time adjustment and ease of use. Thus, these components are not indicative of the components to be used in a final design.

The prototype uses a full Texas Instruments TMS320C6713DSK development board, whereas the final design will use a custom integrated circuit with only the TMS320C6713 chip itself. The development board costs $395, so the cost difference is significant, but a lone TMS320C6713B chip costs only $19.20 when purchased in bulk of 1000 or more. This is roughly a 20x smaller cost, neglecting any necessary integrated circuits and fabrication costs.

The prototype also uses a full-size Alto Professional ZMX52 mixer board as a microphone amplifier. This mixer board was chosen because the gain of the microphone could be easily adjusted in real-time instead of being constant, thus making testing and debugging the system much easier. This mixer costs $49, whereas a simple audio amplifier chip such as the Texas Instruments LM386 only costs $0.43 when purchased in bulk of 1000 or more. An analog circuit would need to be constructed around an LM386 or similar chip, but the price difference is significant and would indeed be worth implementing in a final design.

The prototype uses a few different microphones for testing, these being the Soonhua Condenser Microphone and the Behringer Dynamic Microphone, which cost $14 and $20 respectively. These were used due to the simplicity of testing and connection, as they had normal XLR/3.5mm outputs. The CMA-4544PF-W Electret Condenser Microphone is much cheaper, as it is $0.38 when bought in bulk of 1000 or more, and would work quite well in combination with an LM386 or similar analog amplifier circuit.

The speaker for the system is not relevant in the context of economic consideration, as the speaker is not part of the product. Users of the product will be able to connect any speaker that they desire to the system.

Overall, the system component cost can be reduced from around $460 in the testing phase to around $25 in the production phase. This gives a consumer-end product price of around $100, which is quite reasonable.

3 Design Approach

After identifying each of the requirements that needed to be met to create a viable product, the team had to actually identify a software approach to solving the problem. The design began with the block diagram shown below, then some MATLAB simulations to theorize an algorithm for the board implementation, and finally implementing the MATLAB solution on the board, and debug the board implementation.

3.1 System Architecture

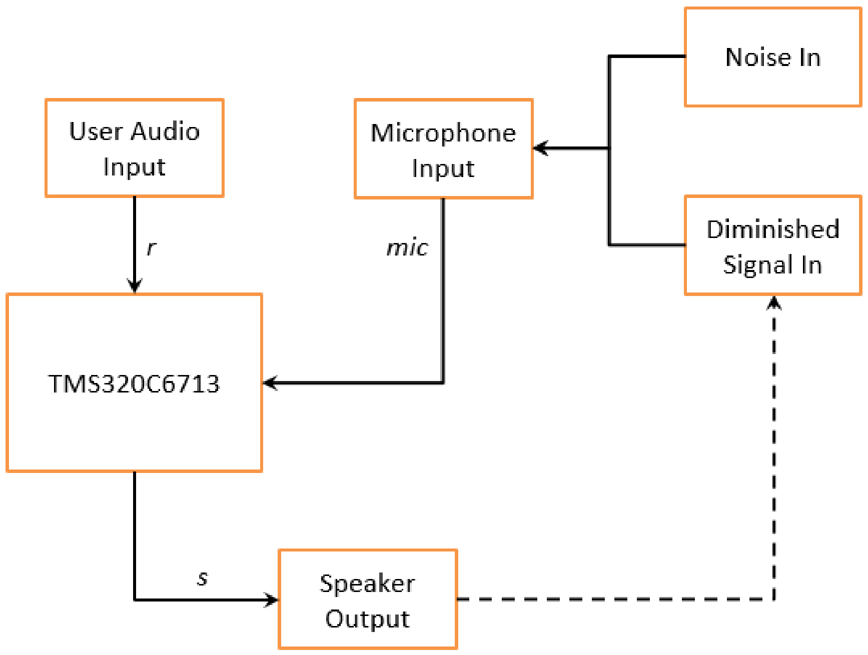

Figure 6. System Block Diagram

The system block diagram (to be further edited) is shown above. The system plays music, with amplification set by the algorithm, which is described below.

Essentially, the user plays an audio file in through the TMS320C6713DSK board, and the board runs a recursive loop to analyze how loud the `Speaker Output` is versus the `Noise In` in the room, the output is then adjusted accordingly. The `Microphone Input` is comprised of a `Diminished Signal In`, which is the `gain*Speaker` output, and the `Noise In`, which is the noise that is being created in the environment. The exact algorithm is described below.

3.2 Algorithm Cost Function

The initial algorithm that was selected for the project was the gradient descent algorithm. This method was suggested by the team advisor who had used the method effectively in projects in the past. This algorithm proved to be a poor decision for a number of reasons, the most important being that running in real time was not easily possible, so the team implemented a simple gain update function instead:

$$a[k+1] = a[k] - \mu(T - d)$$

Where a is the current/next amplitude, μ is the adjustment step size, T is the noise threshold, and d is the weighted ratio of calculated noise amplitude to total microphone-in amplitude.

3.3 MATLAB Algorithm Design

The algorithm was first modeled in MATLAB before the algorithm was implemented on Texas Instrument's TMS320C6713 Digital Signal Processing Board. The first thing the model does is initialize several parameters. A sample audio signal was first read into the r array. In the test model, it looks like the following:

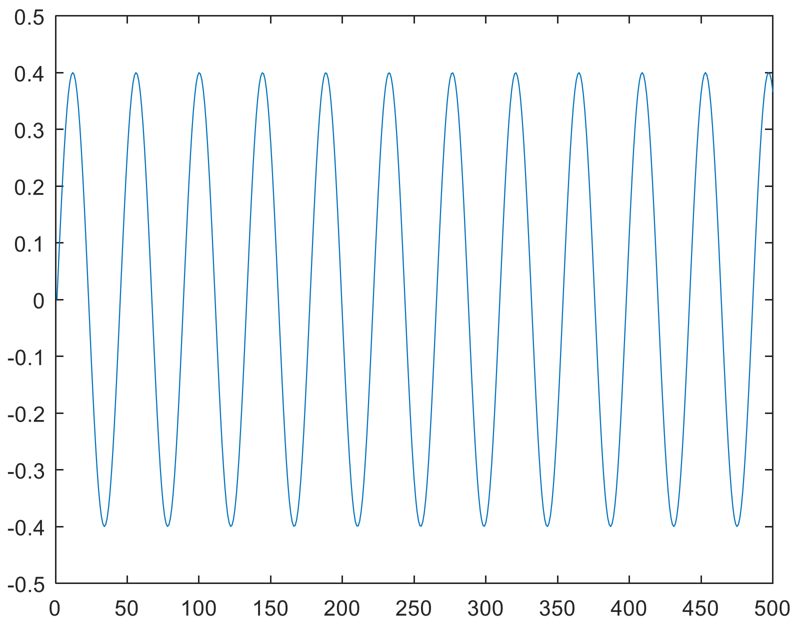

Figure 7. Simulated Audio Signal at 440 H

This frequency was sampled for a duration of about 1.3x10^6 seconds as shown in Figure 9. Note that the sine wave oscillates too quickly to see the swings with this resolution.

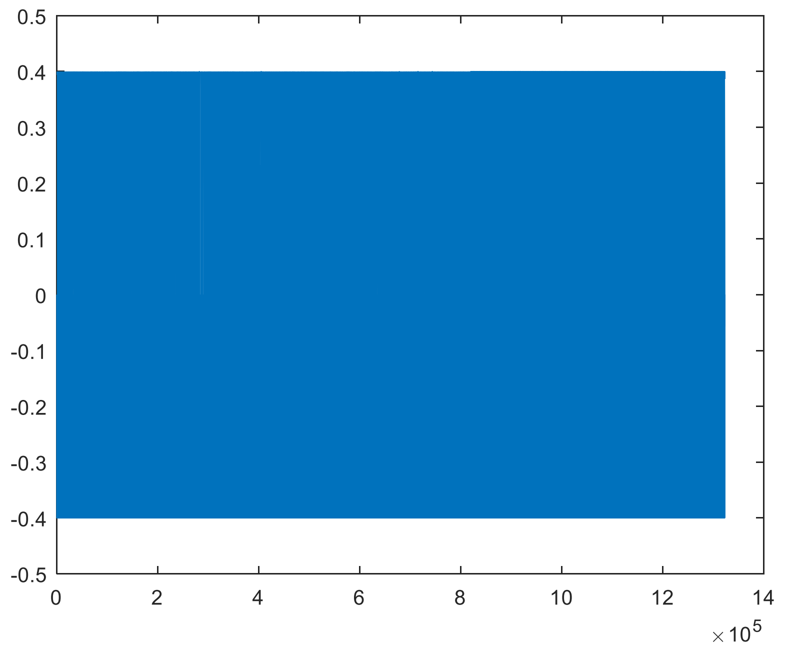

Figure 8. Extended 440 Hz Sine Wave Tone.

Then, a sample pure noise signal is read into the noise array. The noise signal in the script is modeled in Figure 10 below

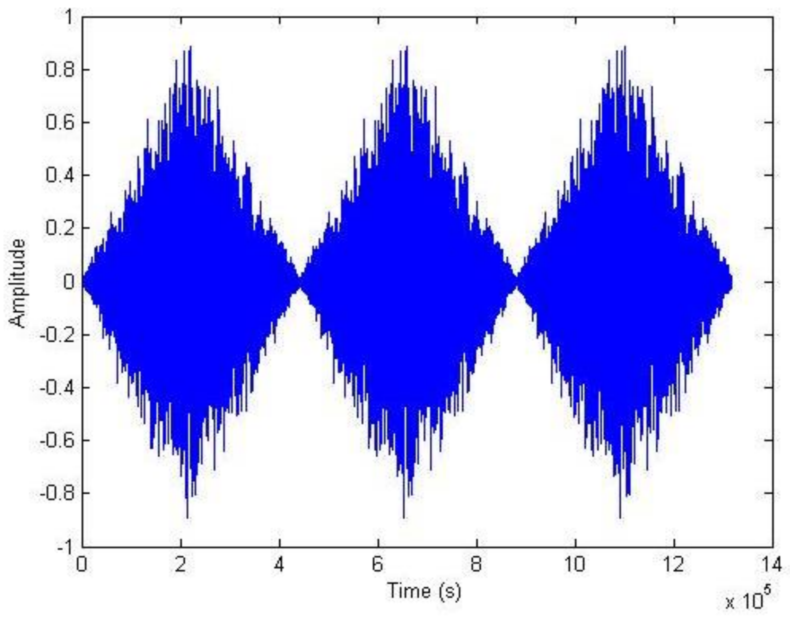

Figure 9. Simulated Noise

The minimum length of these two arrays is then calculated, so that the algorithm does not encounter any out-of-index errors. Next, the noise threshold, the number of samples to perform the calculations over, and the adjustment step per sample are defined. The gain is calculated as a part of a theoretical initialization setup phase. During this phase, the room which the device is in is assumed to be noiseless. A sine wave is then played and subsequently recorded to calculate the gain coefficient of the microphone and room.

Next, the functional loop of the algorithm begins. First, the modeled noise and user inputted signal is calculated using the gain that was just found in the previous step. This resulted in the following shape:

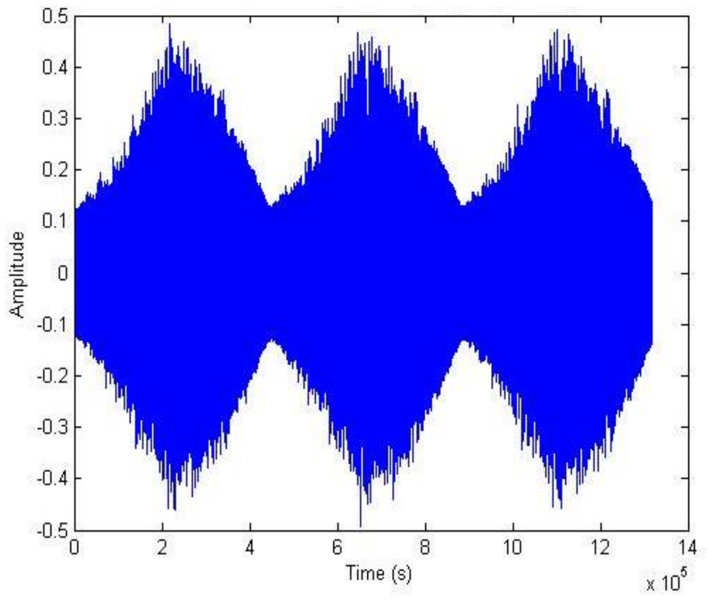

Figure 10. Simulated Microphone Signal

Finally, the actual leveling routine with the cost function implemented is performed to calculate an amplitude coefficient for the played sound, which is done over the entire time domain of the signal. The resulting scaled sound looks like this in the model:

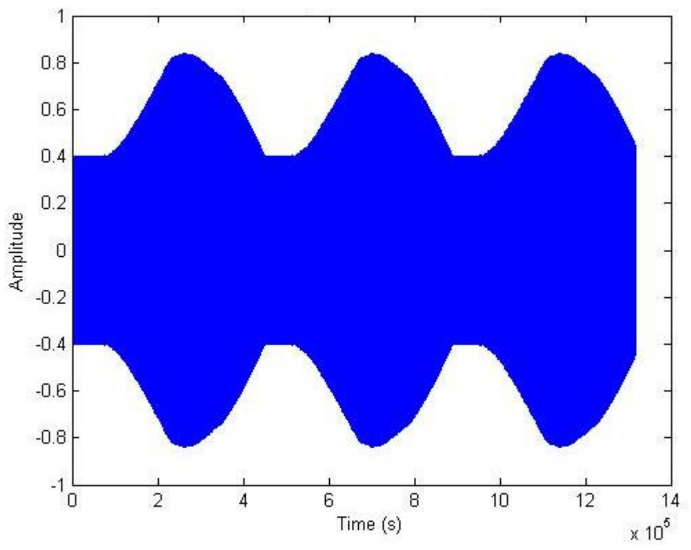

Figure 11. Simulated Speaker Output Based on Noise

3.4 MATLAB Iterations

The MATLAB has undergone several iterations to get where it is currently. The first iteration was to simply get the desired behavior to work, as outlined in the Algorithm Design section. Then, because the MATLAB cannot run our algorithm in real-time, a real-time environment was simulated. First, the `r` array, which was the input signal and `N`, which was the pure noise signal, were manually created in Audacity and saved as .wav files. These .wav files were then read into MATLAB arrays using the `audioread()` function. Next, the microphone input signal, microphone, was created by combining the pure noise signal `N` with the last output `s`. Following this, the actual algorithm was carried out:

<u>Step 1. Averaging Input Signals</u>: Using the average power of a certain number of past samples of the `r` and microphone signals, as shown below, where `k` was the current sample and `nSamples` were the number of past samples to analyze.

```
noiseAvg = mean(noise(k - nSamples : k) .^ 2);
micAvg = mean(micSim(k - nSamples : k) .^ 2);
```

Figure 13. Run the Cost Function.

<u>Step 2. Signal to Noise Ration Calculation</u>: After these calculations were completed, the main algorithm function was ready to be executed. It is shown below, where `u` is the step size, this the acceptable noise threshold, and `d` is the calculated noise-to-signal ratio. Division by zero errors are avoided here by a simple if statement.

```
 if micAvg ~= 0
     d = noiseAvg / micAvg;
end
```

Figure 12. Run the Cost Function.

These steps were repeated for every sample of the input signal, then the MATLAB script terminated and showed the graphs seen in Algorithm Design Section.

<u>Step 3. Gain Calculation</u>: The next iteration of the MATLAB code introduced gain calculation and implementation. The gain is calculated using a sine wave played out of the system as a part of the initialization process. The calculation of average power was then changed to the following:

```
noiseCalc(k) = micSim(k) / g - s(k - 1);
sigCalc(k) = s(k - 1) * g;
noiseAvg = mean(noiseCalc(k - nSamples : k) .^ 2);
micAvg = mean(micSim(k - nSamples : k) .^ 2);
```

Figure 13. Average the Noise and Microphone Signals.

<u>Step 4. Cost Function Implementation</u>: The first two lines of code needed to be included because the MATLAB code made assumptions that were impossible to put on the board, as some of the code was non-causal and needed to be fixed. The cost function calculation of the algorithm also needed to be changed to account for the calculated gain, as shown below.

```
a = a - u * (th - d);
```

Figure 14. Cost Function implementation in MATLAB.

Gain is an important environmental effect in the system. As the speaker plays out sound, which is passed into the microphone with the ambient noise, the sound that leaves the speaker does not completely return to the microphone. Some of the soundwaves dissipate through various environmental effects while traveling through the air and around the room where the system is installed; this is caused by multipath fading. The speaker and microphone themselves have intrinsic gains, in addition to the environmental factors of gain, as the speaker could be powerful and the microphone could be picking up signal at a diminished volume. Conversely the speaker could be weak and the microphone could be very sensitive, or any such combination.

It is important for the system to have knowledge of the net gain because the algorithm needs to subtract the signal from the microphone from the currently playing signal to accurately adjust the signal out of the speaker. Doing so correctly requires a gain scalar on the microphone samples. The system calculates gain by playing a short 1000Hz sine wave out of the speaker and reading in the signal from the microphone upon powering on. This gain encompasses the microphone's intrinsic gain, the speaker's gain, as well as any environmental gain.

3.5 First Functional Algorithm Design in C Language

The device's main component is the Digital Signal Processor or DSP, a specialized type of microcontroller. As such, its operation is defined in the C Language [6].

Step 1: Setup Procedure: It is worth mentioning beforehand how certain tasks are accomplished by the program. A union variable named `temp` is often used, consisting of one Uint32 divided into two shorts (each ADC reading is 16 bits). A union variable is useful in this situation as the two channels of the ADC can be included in the same variable. The variable is set when reading values from the ADCs, a function provided by the board libraries, placing the media input into the lower 16 bits while placing the microphone input into the upper 16 like below.

```
union {Uint32 combo; short channel[2];} temp;
temp.combo = MCBSP_read(DSK6713_AIC23_DATAHANDLE);
```

Figure 15. Read Values from ADC.

The values placed into the temp union are first moved into other variables for further manipulation, and temp is then recycled as an output variable. The output signals calculated are placed into each union channel and the combo is passed into the DAC, another board library function, for output to the user's stereo. As there are still two channels in the temp union, stereo is possible using this implementation.

Step 2: Gain Calculation: The program must initialize the board's support library. This allows the board to utilize the functions provided by Texas Instruments. Next, the codec is opened and its handle is stored. Most variable arrays are set to 0 to initialize, but the `gainSin[]` array is set as a one second long, 1 kHz sine wave sampled at 44.1 kHz.

```
for(i = 0; i < GAIN_LENGTH; i++) {
      gainSin[i] = (short)(30000 * sin(1000.0 * 2.0 * PI * i /
      SAMPLE_FREQ));
}
```

Figure 16. Play 1000Hz Board-Generated Sine Wave.

This is to allow the board to expect a certain signal, so that the gain can easily be pulled from the signal output from the speakers. Serial ports are then configured for 32 bit operation, allowing for two signals (left and right channels) to be transferred in a single read or write. The codec that was opened is set to the pre-defined 44.1 kHz sampling frequency.

```
// set codec sampling frequency
DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_44KHZ);
```

Figure 17. Set 44kHz Sampling Frequency.

<u>Step 3: Interrupt Configuration</u>: the main program maps an event (codec activity) to an interrupt function, which comprises the remainder of the program's functionality. When the initialization step finishes, the board enters the volume adjustment algorithm permanently.

The interrupt has three branches. The first checks for a reset signal- if found, the codec is reset to prepare for operation. The second is taken every interrupt if the gain-calculation length has not yet elapsed: a sine wave is played using the `gainSin` array explained earlier and by comparing the power received by the microphone and the power of the sine wave during this period the gain of the environment can be calculated and used in normal operation. The third branch is taken after gain initialization is completed to run the algorithm that actually raises and lowers the volume of the system.

<u>Step 4: Read in Input Signals and Normalization</u>: the temp union is populated by the media and mic inputs just like in the gain calculation. The two channels, currently typed as shorts, are then

stored as normalized floats by dividing the sample by 32768 and squaring it (and in the case of the mic channel, dividing by the square of the gain).

```
// r
      short s_sample = temp.channel[LINE_IN];
      float s_norm = ((float)s_sample / (float)(32768)* ((float)s_sample /
      (float)32768);
      // mic
      short mic_sample = temp.channel[MIC];
      float mic_norm = ((float)mic_sample / (float) 32768) * ((float)
      mic_sample / (float)32768) / (gain * gain);
```

Figure 18. Read in Input Signal and mic channels to corresponding variables then normalize

Step 5: Average Input Signals: The rotating averages of the media and mic are kept by adding the new normalized sample minus the oldest normalized sample, each divided by the number of samples in the average.

```
      mic_avg = mic_avg + (mic_norm - mic[index]) / N;
      s_avg = s_avg + (s_norm - s[index]) / N;
```

Figure 19. Average Mic Signal and Input Signal.

Step 6: Noise Calculation: The noise is calculated by subtracting the square root of the rotating media average from the square root of the rotating mic average. The noise is then normalized by squaring it. This noise reading is then used to update a third rotating average of noise levels.

```
      // calculate noise
      float noise_norm = (sqrt(mic_avg) - sqrt(s_avg));
      noise_norm *= noise_norm;
      n_avg = n_avg + (noise_norm - noise[index]) / N;
```

Figure 20. Calculate the Noise based on mic average.

Step 7: SNR and Cost Function Calculations: At this point, because it is required to refer to previous values to subtract them from the rotating average, the normalized noise, mic, and media values are stored in an index. Unless the rotating average for the mic is zero (in which case behavior is undefined) the variable `delta` is set to the rotating noise average divided by the rotating mic average. This is used in the cost function where the gain, stored in variable `a`, is reduced by the step size `mu` multiplied by the difference between `NOISE_THRESHOLD` (both explained in the Section 3.2 Algorithm Cost Function) and `delta`.

```
if(mic_avg != 0)
      delta = n_avg / mic_avg;
a = a - mu * (NOISE_THRESHOLD - delta);
```

Figure 21. Cost Function Applied to Determine How Much the Signal Needs to Change.

Step 8: Scaling Output Signal: Note that `delta` may be larger than `NOISE_THRESHOLD`, causing `a` to increase. In essence, the greater the amount of noise compared to the total signal received from the microphone, the higher the gain will become. Finally, the gain is restrained from 1 to 6 (muting the user's media, peaking their speakers, or otherwise being obnoxiously loud is not desired), the temp union's channels are populated with the media samples multiplied by the calculated gains, and sent to the DAC for output to the speaker.

```
temp.channel[0] = a*s_sample;
temp.channel[1] = a*s_sample;
MCBSP_write(DSK6713_AIC23_DATAHANDLE, temp.combo);
```

Figure 22. Adjust Signal by Cost Function Factor "a."

3.6 Refining Preliminary Code

The team built and tested the design as described above. From testing, it was decided that the current implementation was not suitable for production-level. Feedback was rapid and most media, especially songs with louder notes or quickly changing volume levels, would result in a positive feedback loop.

One solution considered was to strengthen the hysteresis of the system. If the system “remembered” sound levels from longer ago, it would not only cause the system to handle louder notes more effectively, but also to react to input noises slower. Unfortunately, storing these data samples is costly in terms of system memory, so this was not a feasible approach. Another suggestion was to store every second or third cycle in a sort of duty cycle instead. While this could potentially help with the system’s inability to recognize its own louder notes, it would not actually increase the number of samples used and as such would not slow down the response rate. Additionally, refactoring requirements stemming from the duty cycle, which include complications with the interrupt, not memorizing half the samples, possibly dropping samples where volume spikes, etc., would simply make the task more trouble than it was worth.

<u>Refined Step 5</u>: Ultimately, using the power of the signals by squaring the inputs, which was done to account for negative inputs and more usefully compare averages, proved to be ineffective. Instead the group decided to record the sample of the last 200 samples with the greatest value. Most of the code’s framework such as how data was obtained was the same, but there were important changes to how the data was processed:

```
if(s_max < abs(s_sample))
            s_max = abs(s_sample);
      if(mic_max < abs(mic_sample))
            mic_max = abs(mic_sample);
```

Figure 23. Setting Max Values.

<u>Entire Refined Algorithm</u>: If the absolute value of the input from the media or the microphone was not greater than the current maximum value, which is to say not the loudest point, its values are not saved in memory. If it is, however, the values are saved and used in a rotating average of loudest samples per batch:

```
if(index == N - 1)
      {
            s_avg = s_avg + ((float)s_max - (float)s[max_index]) / N;
            mic_avg = mic_avg + ((float)mic_max - (float)mic[max_index]) / (N
* gain);

            // avoid division by zero
```

```
        if(mic_avg > 0)
              delta = (mic_avg - s_avg)/ mic_avg;
        else
              mic_avg = 1;

        // apply the cost function
        a = a - mu * (NOISE_THRESHOLD - delta);

        // never get quieter than default
        if(a < 1)
              a = 1;
        if(a > 7)
              a = 7;

        s[max_index] = s_max;
        mic[max_index] = mic_max;
        max_index = (max_index + 1) % N;

        s_max = 0;
        mic_max = 0;
  }

  index = (index + 1) % N;
```

Figure 24. Rotating Average of Batches.

While prior implementations have made heavy use of rotating averages, they only covered a total of 200 samples (at 44.1kHz sampling rate, mind) leading to the extremely sensitive and overactive behavior. The updated rotating average covered 200^2 samples, as only the sample with the greatest magnitude is forwarded to the rotating average. Additionally, the gain of the system is only updated during the end of each batch as well, also contributing to a more gradual volume change. Both effectivity and user experience were enhanced due to the new changes in the algorithm.

While this slower implementation has proven much better in testing, it should still be noted that it suffers from the same problem of omitting samples. This is mitigated by ensuring the greatest magnitude sample is stored and that the 200-sample window is very small: a song would need over 20 beats/second to register as constant in the rotating average. Even if the response is found to be too slow, the constant `mu` can easily be changed in order to adjust the rate of change as desired.

3.7 Normalization

During testing the team encountered problems with the microphone input. The average signal magnitude of the microphone was much smaller than the audio input which caused issues in the calculation of the noise. One proposed way to handle this was to normalize the audio input so that the algorithm can calculate the noise more accurately. The simplest way to normalize the audio input was to track the max value and normalize the data input, which was then used as a scaling factor. This allowed handling of any input from the user without worrying about how loud the input signal was. This was important because the microphone was already normalized based off of the amplifier. By normalizing both inputs, we can create a more accurate noise value.

The results of normalization were very promising after MATLAB simulation. The issue with normalization was that if the user makes the audio source quieter, then the normalization will not compensate for the fact that the volume was lowered. So, in order to create a more robust normalization function, the algorithm was designed to forget what the max value was after a certain period of time. This allowed for the normalization to be more reactive. However, it also introduced a new issue. If the audio signal was quiet for a long period of time, then the new normalization factor would scale the quiet signal up to a relatively large value. This could have caused issues with the adjustment algorithm but can be compensated for.

3.7.1 MATLAB Simulation Results

All of the signals that were used in the MATLAB simulation are 500hz sine waves that have been put together to create a longer signal.

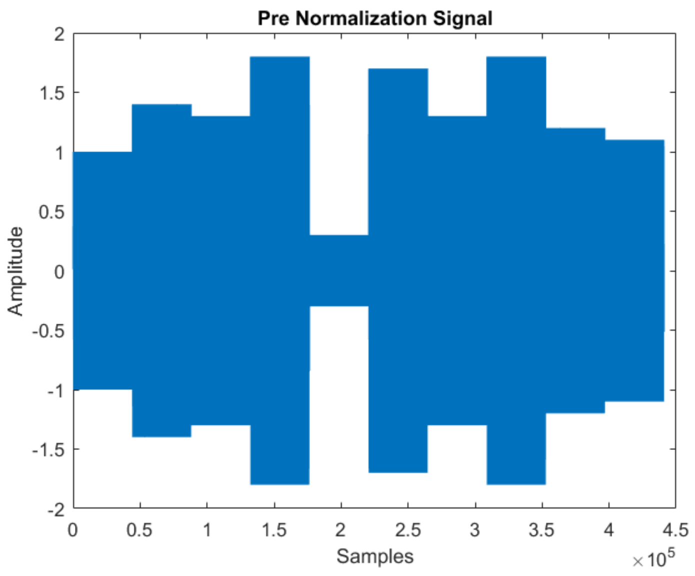

Figure 25. Pre-Normalization Signal.

This is a graph of the amplitude of the signal before normalization. This signal was chosen because the beginning is quieter than the max so that the normalization algorithm shows its ability to adjust to a new maximum.

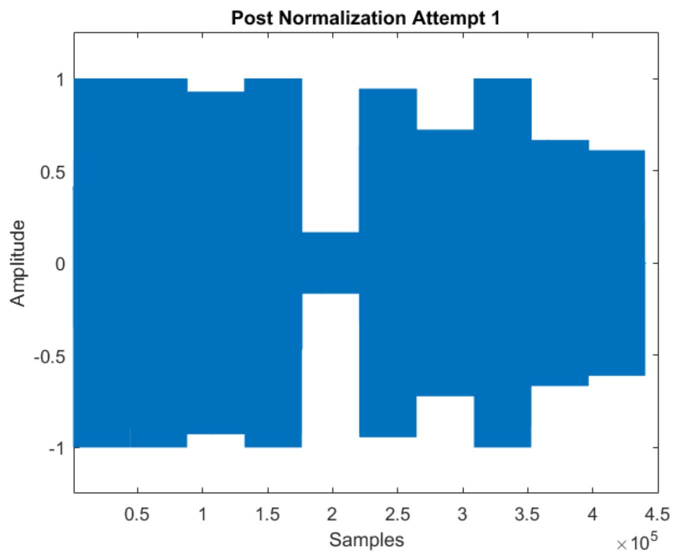

Figure 26. Post Normalization Signal Using Absolute Max as Gain Factor.

This graph shows the results of the normalization attempt that uses the overall maximum value that has been found up until that sample as the normalization factor. This signal holds a similar shape to the pre-normalization signal after the max value of the entire data set has been reached. But as long as the data set's maximum value has not been reached, the data will be normalized properly to 1.

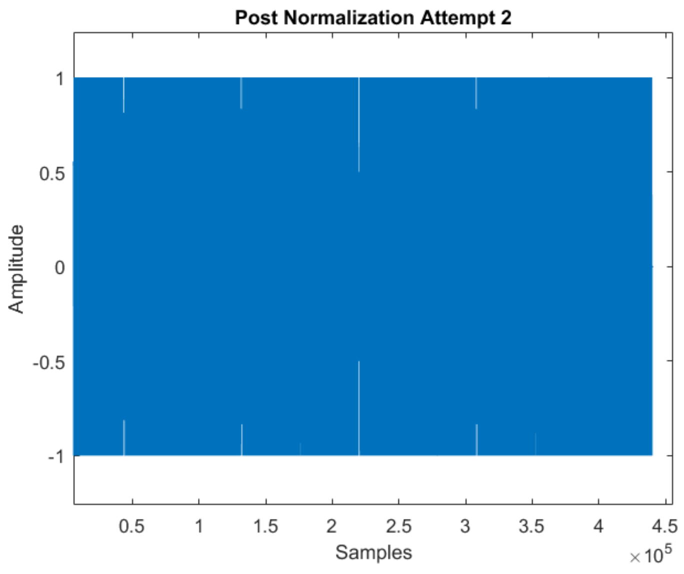

Figure 27. Post Normalization Signal Using Local Max as Gain Factor.

While this graph is much less interesting to look at, it is a better normalization attempt than the previous pass. By using a window for detecting a maximum value, we are able to better adjust all of the signals to the same maximum value. The only problem with this method is that as the window increases, the accuracy of normalization decreases. The small white lines in figure 3 are normalization error and as the window increases those lines become wider, as seen in figure 4.

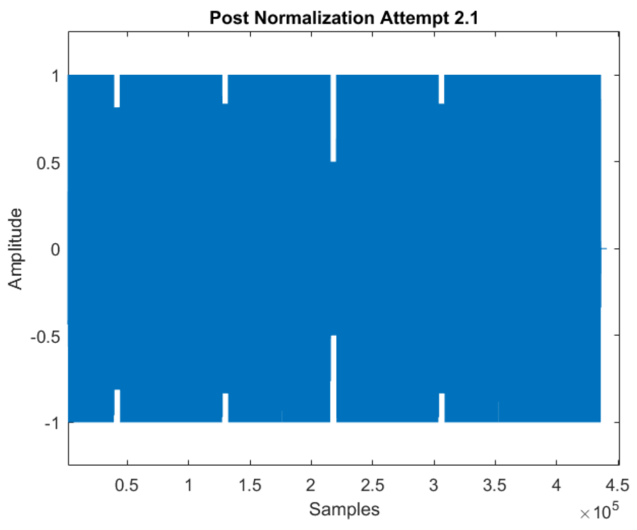

Figure 28. Post Normalization Signal Using Local Max and a Wider Window.

The obvious solution to errors occurring during normalization with a wide window is to reduce the window size, but this will cause issue elsewhere in our algorithm. The signal begins to lose its meaning if it is normalized too well. It will no longer be close to the same signal that was passed into this system and that will make it impossible for the rest of the system to work properly.

3.8 Dynamic Mu

Once the algorithm was correctly implemented on the board, the next step was to create a more responsive and precise system. The first step in doing so was to make the system more reactive in scenarios where the difference between the noise and speaker output was very large. This was accomplished by creating a dynamic step size, `mu`. The first iteration of the dynamic step size was created with different step sizes for when the system is increasing volume and when the system is decreasing volume. This was accomplished by having a larger step size when the system was reducing volume and when it was increasing.

```
if(delta > NOISE_THRESHOLD)

        mu = mu_rising;

else

        mu = mu_falling;
```

This attempt did create a better listening experience, as it was more important to not have a large volume of sound when there are small amounts noise than it was for the system to increase quickly to a rapid increase in noise. This was not an elegant solution to the problem, because the system should have just reacted quickly only if there was a large difference between the noise and the output of the speaker. This was accomplished by creating an equation with an output of zero when the input was close to zero and quickly approached one as the input moved away from zero. The rate at which the equation approaches one was what would determine how aggressive the system would react to a change in noise.

$$\mu = \mu_{center} + \mu_{delta}\left[\frac{2\tan^{-1}(20 * delta)}{\pi}\right]^2$$

The equation above satisfies the behavior required for the system. μ_{center} and μ_{delta} are two constants that the equation will be between, μ_{center} being the smallest value possible and μ_{center} + μ_{delta} being the largest value possible.

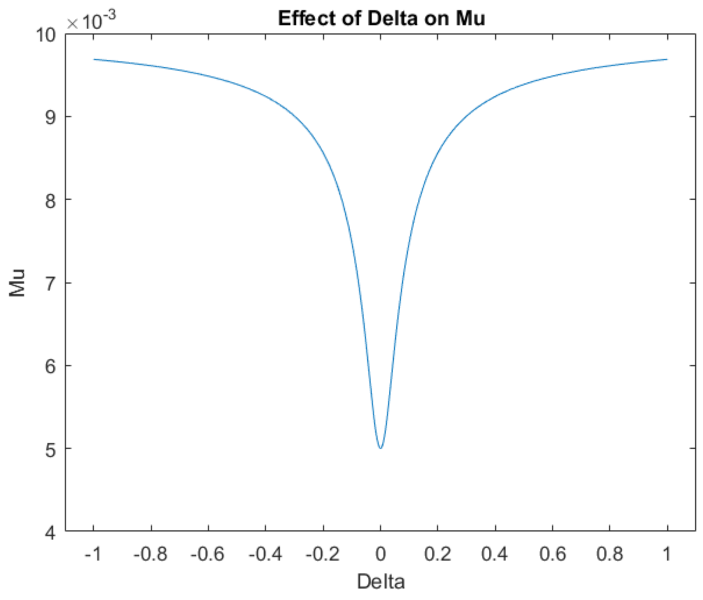

Figure 29. Effect of Delta on Mu.

The graph above shows the way that `mu` will vary based off of the noise percent in the room. This behavior shows that the system will react more aggressively when the delta is either higher or lower than zero. The `mu` is at an unstable equilibrium when the delta is close to zero, but it overall makes the system more stable and smoother due to the fact that the system reacts more quickly when the noise changes quickly.

4 Product Results

After implementing a somewhat functional algorithm on the board, the team moved into the testing and analysis portion of the project. Testing was very important for this device because it is a human experience product, and there are no numerical or physical benchmarks to hit besides the ensuring the device operates in real time. The testing methodology is described below along with the results of the testing process that allowed the device to be tuned to the human ear.

4.1 System Testing Methodology

Two signals, `Audio In` and `Microphone In`, must be provided to test the device, using either MATLAB algorithms or hardware with the debugging tools of Code Composer Studio. Audio In is the media to be amplified. Testing shall include many types of media that users may input into the device including different genres of music, TV shows, movies, etc. Signals tested should include some with and some without human voice. Hearing and understanding what is said to the level of the original media is a priority when human voice is included, as any modulation or distortion will be significant to the user. Microphone In is any sound that the microphone picks up that could be present in the environment that the device is immersed in, and will determine the level of amplification of the Audio In. It was important to test as many types of noise as possible because the noise can vary drastically in the same environment. Factors used to distinguish noise signals include frequency, duration, amplitude, and periodicity. Numerous types of everyday noise were collected and used in testing to ensure an appropriate and enjoyable response for the user. Test signals were obtained through free online sources when available (YouTube, stock sound databases, etc.).

For the purposes of testing, users of the device are categorized into one of three groups: Primary Users, who are listening to sound coming out of the speakers and may be making noise themselves; Secondary Users, who exist in the area for a variable amount of time but do not care for the audio coming out of the device but still inherently create interference that should not be completely drowned out as these users try to interact with the Primary Users; and Noise makers, who are any of the people creating noise that the device should drown out completely.

Testing the device with these three groups in mind was important to ensure that the device created a good experience for both the primary and secondary users in as many cases as possible.

Primary and Secondary Users desire that both the amplified audio and the ambient noise can be understood clearly. This interaction can be modeled with human dialogue as the noise signal, and the test can be considered successful if both the noise and the media can be understood. The noise is more varied and carries no desirable message for the Noisemakers. It is unimportant for someone performing the loud task to understand the environmental noise for the most part, so the noise signal can be safely overpowered. The goal of testing is to find any signals that the device does not react well with, either by not increasing the volume enough or by increasing it too much, and determine the correct adjustments to react appropriately to those signals. Suitable noise signals to test with include sundry household noises both with and without voice overlaid.

4.2 Debugging the Algorithm

During the first week of testing on the actual Developer's Board, the team determined a few bugs with the system and with the IDE. The first issue the team encountered was that there was a high-pitched pinging sound that came out of the speaker every time the codec switched inputs from line in to microphone in. In order to fix the problem, the team first tried to only take one sample per interrupt. To do this, they doubled the sampling frequency and alternated what input they sampled from and only did calculations once every other sample. Unfortunately, this did not fix the pinging issue. So, the next attempt to fix it was to use a splitter to have both the Microphone and Audio In signals on the same line but one is left and one is right. A splitter does not actually do this, so that solution did not work. But the idea of passing the audio and microphone in through the same line would solve the issue of the pinging but would make the system not able to handle stereo audio.

Another one of the issues that the team encountered during the testing process was the microphone audio sounded like it was bleeding through into the output audio. The team was not able to remove this problem due to some issues with the IDE, but they believed that the problem was that the volume adjustment was too reactive, as well as the algorithm not running in real time. If the system is too reactive, then the output audio will begin to sound like the microphone noise. To fix this the team just needed to reduce the value of the step size of the adjustment equation.

Finally, the team ran into issues with the Code Composer Studios. On the first day of testing, the team was able to successfully run the code on the board, even though there were issues with performance of the system.

While attempting to port the program to the board after confirming its functionality in MATLAB, the team faced several issues. First attempts to run the program on the board with testing equipment lead to erroneous results: rhythmic background static was present, as well as a reduction in music quality and feedback from the microphone coming through the speaker. The first speculated cause was a time-step variable set too low. If the reaction speed was too fast, then the output of the microphone could effectively be modulated into the input media, causing the volume to be increased finely enough to vaguely preserve the input from the microphone rather than generally increasing the volume of the output. The second speculated cause was an insufficiency in the codec due to the team's implementation of input from two sources. Ideally, the input and output would be multiplexed rapidly between microphone and media, putting the two sources through a single ADC using a duty cycle- this was suspected due to the reduced quality of both the media and microphone and the fact that they both came through the same output. The board also required significantly more clock cycles to run the multiplexer procedure, making it impossible to run in real time.

Another issue discovered during preliminary testing was that some variables (including output gain) were diverging to NaN values. By porting over proper variable initialization from another build of the code, these variables stopped diverging and the background static ceased in further tests; however, this did not stop the microphone feedback. The first fix attempt for the feedback was increasing the response time: if there was no issue with the codec and multiplexer, then decreasing the resolution of the microphone's input would stop the modulation. However, this did not solve the problem, indicating a conflict with the board's codec. Initial tests involved reducing the sampling frequency of the program, even though this would reduce quality, to 16kHz from 44.1kHz, but this stopped all output to the speaker and thus was reverted. CCS's diagnostic tools were used, and it was found that switching the codec between inputs was not viable for real-time. The code too took over 10000 clock cycles, while the loop otherwise takes about 500 clock cycles. In production, it would be plausible to fix this issue by including another codec such that the microphone and media can be digitally converted independently. In order to continue to utilize the available board, the Audio In signal will be converted to mono to exploit the codec's multi-channel ADC (requiring an extra channel-splitting cable). Although this will reduce audio quality, it will be required to implement the algorithm with the limited ADC resources available on the board. An extra codec will allow for the media to supply both left and right channels to the two-channel ADC, and allow the microphone to supply its input directly to a second ADC.

There were difficulties with getting the system hardware to interact with the board properly. Some hardware difficulties that the team ran into were microphone and line in input switching and audio hardware interaction.

Input switching was one problematic area. The TMS320C6713DSK has two inputs that can be switched between via a multiplexer. One is a 3.5mm microphone jack that features amplification, and the other is a standard 3.5mm line-in jack without any amplification. The issue with using both of these inputs simultaneously was that the protocol for switching between the inputs takes far too long to sample both the microphone input and the line in input in a real-time environment. This meant that a stereo input could not be used for the microphone and line in, as they both needed to connect to a single stereo input to avoid the problems with using input switching. Thus, the line-in and microphone inputs both need to be mono inputs and connected to the line-in jack on the board via a mono-to-stereo cable.

Another area of difficulty was audio hardware interaction. If the line-in and microphone were to be connected to the line-in port on the board, then the line-in would have proper volume but the microphone would not be amplified correctly. If the line-in and microphone were to be connected to the microphone input on the board, then the microphone would be amplified correctly but the line-in would experience extreme clipping, due to it being amplified as well. Thus, the solution to this problem was to pre-amplify the microphone signal before being fed into the shared line-in port. This was first done by connecting the microphone to a laptop. This laptop's audio was placed into loopback mode, where the microphone input was fed back out through the audio output. The audio output was then connected to the board and combined with the microphone input to form a stereo input. This worked quite nicely, as the laptop microphone input is amplified correctly. The trouble with using a laptop as a microphone amplifier is that there is a fair amount of latency involved. Thus, a proper solution would be to get a dedicated microphone pre-amplifier to avoid latency problems. Another solution would be to get a different microphone with a built-in amplifier, but that would increase the cost of the system dramatically

4.3 Human Reactions

Human interaction with the device is the most important testing period because the device is meant to provide convenience, but at the same time be completely effective. In each test the team used the same three pieces of audio. However, toward the later stages of testing, new and more complex audio signals will be used to push the limit of the device's functionality.

4.3.1 Preliminary Testing

Simple Tone Testing

The most important aspect of the project is that it creates a pleasant listening experience for the user. In that vein, our team first tested the device on themselves. The first sound used for testing was a 512 Hz tone, which is a C-note. This is a relatively pleasant note, but the most important part was that this sound has a constant amplitude. The device performed well on the first test. When the group was conversing about the functionality, an audible change could be heard in the volume of the signal, and when the group stopped talking to just listen to the music, the volume would fall to the original signal level. This was a confirmation that the algorithm implemented on the board would be functional for controlling the volume of the system. However, no user is just listening to a constant pitch for an extended period of time, so it is not a practical test. For the second test, a slightly more complex song was chosen.

Simple Songs Testing

The second song that was chosen was a sample of 8-bit music, so the 16-bit sampling rate of the AIC23 codec preserved the quality of the music well. The music is a step towards what a consumer would want to listen to on an everyday basis, but the music is still simple enough that the algorithm should be able to handle it as is. The songs in this video are essentially a collection of simple tones like the tone that was originally tested with, there is no human voice in this video, so that will be addressed in the next test. The algorithm worked with this music, but the sound fluctuated a little bit more than we would have liked. The root cause of this problem did not lie within the algorithm's core, but rather with how many samples the board stored before changing the volume of the signal. Taking the average over a larger body of samples will change the volume more gradually instead of instantly reacting to the volume changes in the song. Overall the device passed this test, however some small tweaks need to be made to make the device operate better. The next step was to test how the device reacts to complex music with human voice.

Complex Sound Testing

The final song that was used to test the device was a complicated song with a lot of variations in both pitch and volume throughout the entire song. The team expected the device to have some trouble handling this song as it had some slight troubles with handling the Mario theme songs. The device did not operate very well with this song. Initially it did work well, however, as the song went on the changes in volume that are naturally prevalent in the music before it was run through the algorithm proved to be a bit too much for the device to handle at the time. Because the device was reacting almost instantaneously to the music being played, the varying volume of the song caused inconsistencies. However, the team believed that using a larger set of samples will allow the device to rely less on the constantly changing volumes of the song in the moment, but rather react to what the average volume of the song is over a few seconds.

Preliminary Testing Results

Overall, the first round of testing went well. The team could prove that the algorithm developed is a feasible solution to the problem statement proposed. However, there were still some bugs in the system. The first bug the team noticed, which was where many of the debugging problems arose from was that the gain of the room was not picked up as accurately as planned. This would sometimes drive the signal up to the limit that was set in the program to avoid clipping, to combat this, the team had to implement a hard-coded gain. This is not practical, as the gain is different in every single room, so this problem needed to be debugged before the device could stand as a “black box.”

The next problem, as stated above, was that the volume of the music was changing too rapidly. This created some harsh volume changes, instead of the gradual change that creates a more pleasing listening experience. The proposed solution was to allow the board to hold more samples before making a change in the volume. The board that is being used allows for 2Mb of storage, so there is plenty of room to hold more samples.

The final problem that was realized during testing was that the microphone being used did not pick up sound well enough. The microphone the team originally bought was an electrolytic microphone. This microphone was high quality, but there was no amplifier on board. The output of the microphone was much too weak for the board to pick up, so the value the board read was essentially zero. To combat this for testing, the team could run the microphone through a laptop, and amplify the signal in real time through audacity. After doing this, the signal was much stronger, and the board could pick up the microphone much better. To fix this problem, a microphone with a good amplifier on board will be obtained. This should allow the device to have better resolution and accuracy, which will allow the device to operate better. The good news was all the problems encountered in preliminary testing were relatively easy fixes.

Before the larger sample size could be implemented, a new microphone needed to be obtained to allow for proper gain testing. The team believed after fixing the microphone situation, the gain bugs, would fix themselves, and the calibration step would allow the algorithm to function properly. These fixes were evaluated in the next test.

Also in the next test, the firmware on the board would include a larger data set. At that time, the volume changed every one-hundred samples, and one sample took about 500 clock cycles to take. The speed of the clock on board the TMS320C6713 is 225MHz, so at the time the device was updating the volume every (500*100)/225* 10^6 seconds which is 200μs. Ideally, the device would actually change the volume about every .1seconds. This figure would require further testing in the next step, but a solid number would be settled on at that time. To achieve .1 seconds, the number of samples would have to be (.1s*225*10^6s)/500, which is 45,000 samples. At 45,000 samples, the board would have to be able to store 16 bits*45,000 samples because the AIC23 Codec is a 16bit DAC, which is 720Kb. There are 2Mb of storage available on the board, so there should be room for all this data, plus the rest of the data that the board needs to operate the algorithm. This change will be evaluated at the next testing period.

4.3.2 Secondary Testing

A much better algorithm had been implemented on the board for secondary testing. This algorithm changed the volume based on max values rather than average values, allowing for more accurate noise detection. Since the device was functioning almost exactly as the group wanted it to, a testing procedure was created to allow for consistent testing of subsequent adjustments.

The four variables that the group would look at in this next step were: Type of noise, type of signal in, size of room, and volume of noise. There were tests set up for each variable to determine the bugs that still existed in the system. It is important to note that these tests were done in an otherwise noiseless room, and if one variable was changing, the others remained constant. This testing phase was important for tweaking the noise threshold and μ values to get the best possible listening experience for the user.

The first variable that was tested is how the type of noise affect the system. In theory, different pitches of noise should be absorbed better into the air, according to this graph from a study on outdoor sound propagation

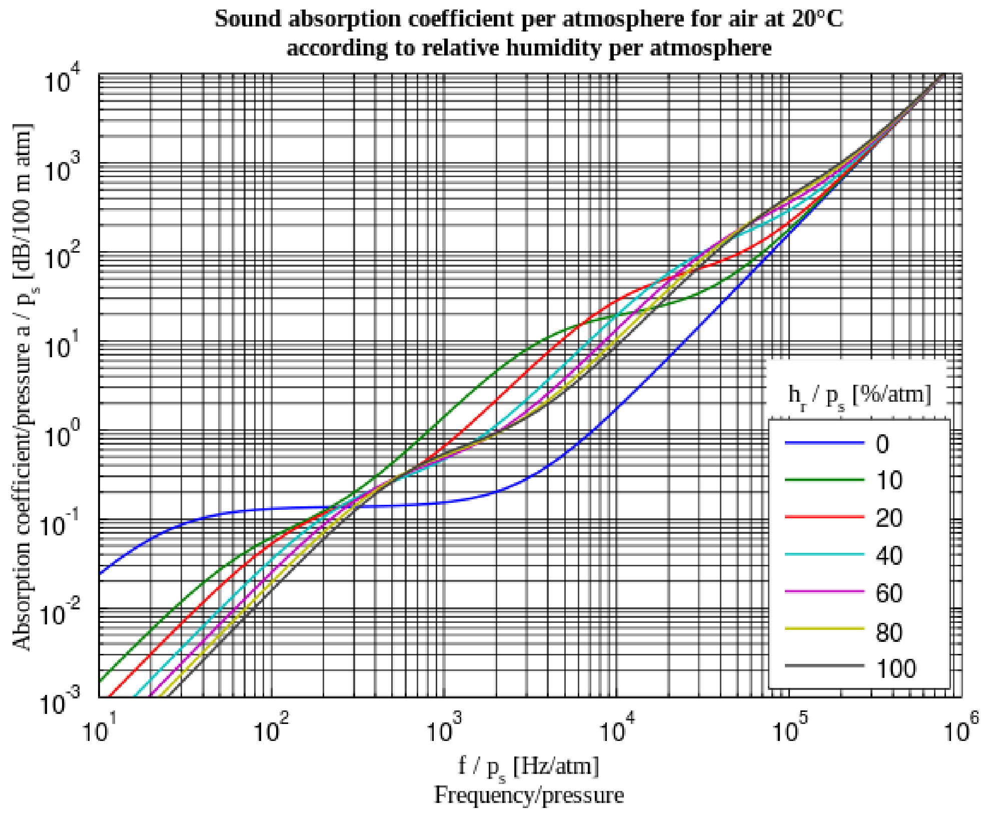

Figure 30. Sound Absorption per atm. for air according to relative humidity per atm. [7]

This graph essentially states that low frequencies travel through air better than high frequencies [7]. However, in practice, different types of noise should not absorb quick enough into the air within a room. In fact, overall it seemed that the signal dissipation should have a negligible effect on the system. Instead our signal was not highly affected by dissipation, but rather multipath fading. High frequencies generally bounce off walls better than low frequencies, which will travel through walls better than high frequencies. This means, that high frequencies should have a larger impact on our system than the lower frequencies, as they will be present in the room for longer. Experimentation will prove whether this is a correct hypothesis or not. To determine the more dangerous signal in terms of creating bugs in our system, the group will pump 4 different types of noise into the test room of different frequencies, and determine which one had the greatest influence on the system. The group will ensure that everything else in the room remains constant during this testing. It was even important for each member to stand in about the same spot during testing as this changed the reflection of the sound.

The next variable that was tested was type of Signal In. This experiment used five different types of signals, some talking, some music, and a consistent sine wave for control. A consistent type of noise will be played at different volumes to see how the system reacts. The goal of this experiment was to see which types of signals our system struggles at adjusting the volume for, and see the system could be debugged accordingly. The team predicted that complex music with lots of changes in volume would be the hardest for the device to adjust to because the noise calculations relied on the volume to be somewhat consistent over a short period, and if the volume dropped off quickly, there could have been unexpected reactions.

The third variable tested was the size of the room. The size of the room should have the most impact on the created algorithm. The gain calculation was somewhat inconsistent, and the gain is what should allow noise to be calculated correctly. In larger rooms, the sound in the room dissipates much faster than in smaller rooms because it does not reflect to the user as fast. A small room, a medium room, and a large room will be used to test this. The most problematic room will be the large room because it will be more difficult to pick up noise, and the amount of noise varies wildly throughout the room. Luckily, the noise most immediately in front of the device is most likely what would be the most annoying to the user.

The final variable that will be tested is the volume of the noise. This was an interesting experiment, as clipping is a very relevant problem in the experiment. This experiment allowed us to fine tune the range for the cost function variable to stop increasing. There is only so much the volume can increase before the signal starts to clip. There is a "sweet spot" for the cost function variable a, which needs to be limited on the upper and lower bounds. If the device allows a calculation to grow to infinity, positive feedback becomes an issue. If a is limited to around seven or eight, then the signal will not clip. To accomplish this, an if statement is used ensuring that a is kept between one and eight. This will allow the volume of the signal to rise and fall correctly without any clipping.

4.4 Product Functionality

This product adjusts the volume of an audio signal according to the ambient noise present in its environment. First, it plays a sine tone out of the speaker to calibrate the environmental gain, whether it be a small room or large hall. It then uses this gain to compare the microphone input with the audio output currently playing. The noise from the microphone is isolated via signal subtraction, then the noise level is compared to an acceptable noise percentage threshold. This is done utilizing running averages for the base signal and the microphone signal, as it gives the system a delayed response. This is because the system should not react strongly to brief instances of noise. From the signal and noise levels, a delta is calculated. This delta is then used in a cost function that computes a scalar, which is then multiplied with the audio input signal to form a scaled audio output signal. This cost function utilizes a dynamic step size mapped to an arctangent function, which makes the system react to noise more appropriately. It also enables the system to react more slowly for gradual noise changes and more quickly for large changes in noise. The audio output is then played through the speaker and returns a diminished form through the microphone and the process repeats.

5 Conclusions

There were several breakthroughs that lead to the proper functionality of the device. MATLAB was an extremely effective tool for theorizing an algorithm to be implemented onto the DSP board, and many of the breakthroughs came from simulation in a MATLAB. The other tool that was useful for testing functionality was the debug tool in Code Composer Studio (CCS). There were many tuning factors that had to be controlled to get an enjoyable response from the system. Code Composer allows the programmer to manually change these factors while in debug mode. This allowed us to change variable values to see what caused the system to either converge or diverge. Some of the tuning factors of the system are described below.

5.1 Device Tuning Factors

One of the issues faced during this project was how to properly recognize the feedback from the system. The device needed to calculate what percentage of the noise was heard through the microphone after being played through the speaker that the device was attached to. Early on, the team realized that it would be necessary to determine the gain of the environment to eliminate one unknown variable. This was determined by fixing the environmental noise by requesting silence in the area for about a second, and fixing the media input as a sine wave to determine the input and output Signal-to-Noise Ratio (SNR). However, once all the other outputs began to change, depending on implementation, the functionality began to degrade as the device's output became greater. Early implementations used a hard-coded limit after crossing a certain threshold. Later, attempts were made that included the current gain to mitigate the next gain change, thus preventing any serious positive feedback issues. These attempts came with the drawback of requiring louder noises to reach the higher gain bounds, and thus the device would not become louder than the noise as desired.

These problems were solved by changing where the environmental gain was implemented into the algorithm. The final version only uses the gain as a coefficient of the media input while determining what portion of the microphone input is noise:

```
delta = (mic_avg - a * s_avg )/ (mic_avg);
```

This algorithm allows the gain of the next output sample to rise and fall as needed without being hampered by the current gain during calculations, and avoiding any positive feedback loops by properly recognizing the feedback signal.

5.2 Future Considerations

The device's main purpose is to amplify media to become louder as the environment does, but the device's functionality could be expanded to include more features. Some features would be added before any proper commercial release of the device, such firmware updates, and some applications may require additional hardware and design than has currently been done.

One interesting pivot would be to create a wide variety of customization options. While the most obvious option, master volume, should be relegated to the user's audio setup, there are other settings available in the algorithm that could be easily changed, (For more information see: Section 3.2 Algorithm Cost Function) to adjust the output of the device. Some variables that could easily be adjusted include causing the output's volume to adjust aggressiveness or tolerate different levels of environmental noise before attempting to adjust the volume. Other settings to consider include minimum and maximum amplification, allowing a user to set custom limits on how much the scaled output should vary from the original. As the device attempts to calculate the gain of its environment, an option should be provided for the user to manually recalculate the gain. The gain of an area can change for many reasons, such as people or furniture moving, the volume of the sound system changing, or moving the device to a new environment altogether. Savvy users may also benefit from the option to tweak the calculated gain, but for the most part the automatically calculated gain should be optimal.

The next feature that would be useful is the reciprocal of the device's function: making the media quieter than the environment, to ensure the media is not overpowering. This would be used in scenarios where the media serves as background entertainment, rather than the other way around, allowing for easier conversation. With this feature in mind, the other permutation is worth considering, to increase media volume as environmental volume decreases, to fill a lull in conversation with music, and decrease media volume as environmental volume increases, to give way to something else that people will want to hear. Such modes would operate using similar versions of the current program, only requiring parameter and comparison changes, and while retooling the firmware would take time, it would require no additional hardware.

Another feature, would be a sound-normalizing feature for the media being used. This feature would quiet exceptionally loud moments to maintain a more uniform volume. This feature would be much more difficult to implement. However, as there is no immediately clear way to identify when an advertisement or similar is being played off audio alone. Thus, one risks quieting a loud section of their media, even though it could be desirable. There is another problem; there is no good way to define a “loud moment”. If the device is too aggressive in lowering volume when the media’s volume is above average then any fluctuation could lead to noticeable distortion, and if it is not aggressive enough then the user would be irritated that the device only quiets the noise a few seconds after it’s already happened. Such a system would require saving the overall volume of the media over time, requiring more memory if used alongside the main functions, and introducing more complexities, which could interfere with the system running in real-time. If the system “remembers” too far back, then different points could conflict, such as dead air in movies or between songs lowering the running average volume. If the system does not remember long enough, then the device may not react to a gradual increase in volume at all. Ultimately, even if this feature was added to the device, it would need to be placed in an optional setting.

References
[1] Schultz, T. J. (1978). Synthesis of social surveys on noise annoyance. The journal of the acoustical society of America, 64(2), 377-405.

[2] Staum, Myra J., and Melissa Brotons. "The effect of music amplitude on the relaxation response." Journal of Music Therapy 37.1 (2000): 22-39.

[3] Gelfand, Stanley (2011). Essentials of Audiology. Thieme. p. 87. ISBN 1604061553

[4]Anthony Cocciolo , (2015)," Digitizing oral history: can you hear the difference? ", OCLC Systems & Services: International digital library perspectives, Vol. 31 Iss 3 pp. 125 - 133

[5] Fox, E. A. "Advances in Interactive Digital Multimedia Systems." IEEE, Oct. 1991. Web. 26 Apr. 2017

[6] Jarvis, Susan. "ECE4703: Real-Time Digital Signal Processing." ECE4703 Useful Links. Worcester Polytechnic Institute, Oct. 2016. Web. 26 Apr. 2017.

[7] Bass, Sutherland, Piercy, Evans, Absorption of sound by the atmosphere, Physical acoustics: Principles and methods. Volume 17 (A85-28596 12-71). Orlando, FL, Academic Press, Inc., p. 145-232, 1984

Appendix A

```
/**************************************************************************
 *  MQP 2017: DrownOut™
 *  Steve Blouin
 *  Sam List
 *  Anthony Ratte
 *  Sam Shurberg
 *************************************************************************/

#define CHIP_6713 1
#define N 200 //number of samples held for an average
#define NOISE_THRESHOLD .05
#define SAMPLE_FREQ 44100
#define GAIN_LENGTH SAMPLE_FREQ
#define PI 3.1415927
#define LINE_IN 0
#define MIC 1
#define MU_CENTER .005
#define MU_DELTA .005

#include <stdio.h>
#include <stdlib.h>
#include <c6x.h>
#include <csl.h>
#include <csl_mcbsp.h>
#include <csl_irq.h>
#include <math.h> // what sin was it using before?

#include "dsk6713.h"
#include "dsk6713_aic23.h"

int index = 0;    // Rotating index
int max_index = 0;
int indexMic = 0;
int s_max = 0;
int mic_max = 0;

short s[N];      // Output signal normalized
float mic[N];
float a_data[N];

short gainSin[GAIN_LENGTH];
int gainIndex = 0;
```

```
float micPower = 0;
float rPower = 0;
float gain;

float a = 2;        // Current and next gain array
float mu = MU_CENTER;    // Step value. 0.01 is a same value for this but it might be
lower for testing
float delta = 0;
float n_avg = 0;
float mic_avg = 0;
float s_avg = 0;
float a_avg = 2;
int input = 1;      //1 for line in, 0 for mic

char init = 1; // initialization mode
char rset = 1;

DSK6713_AIC23_CodecHandle hCodec;      // Codec handle
DSK6713_AIC23_Config config = DSK6713_AIC23_DEFAULTCONFIG;  // Codec configuration
with default settings

#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011

// define the functions
interrupt void serialPortRcvISR();
void initStuff();
void mainStuff();

void main()
{

      DSK6713_init();           // Initialize the board support library, must be
called first
    hCodec = DSK6713_AIC23_openCodec(0, &config);    // open codec and get handle

    // initialize the variables
    int i = 0;
    for(i = 0; i < N; i++) {
      mic[i] = 0;
      s[i] = 0;
      a_data[i] = 0;
    }

    // generate a sine wave for gain calculation
    for(i = 0; i < GAIN_LENGTH; i++)
    {
      gainSin[i] = (short)(10000 * sin(1000.0 * 2.0 * PI * i / SAMPLE_FREQ));
    }

      // Configure buffered serial ports for 32 bit operation
      // This allows transfer of both right and left channels in one read/write
      MCBSP_FSETS(SPCR1, RINTM, FRM);
      MCBSP_FSETS(SPCR1, XINTM, FRM);
      MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
```

```
	MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);

	// set codec sampling frequency
	DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_44KHZ);

	// interrupt setup
	IRQ_globalDisable();			// Globally disables interrupts
	IRQ_nmiEnable();			// Enables the NMI interrupt
	IRQ_map	(IRQ_EVT_RINT1,15); // Maps an event to a physical interrupt
	IRQ_enable(IRQ_EVT_RINT1);	// Enables the event
	IRQ_globalEnable();			// Globally enables interrupts

	while(1)					// main loop - do nothing but
wait for interrupts
	{
	}
}

// the interrupt
interrupt void serialPortRcvISR()
{
	if(rset) {
		DSK6713_AIC23_rset(hCodec, 0x0004, DSK6713_AIC23_INPUT_LINE);
		rset = 0;
	}
	if(init) {
		initStuff();
	} else {
		mainStuff();
	}
}

// any initialization steps for the ISR to use go here
void initStuff()
{
	union {Uint32 combo; short channel[2];} temp;

	//DSK6713_AIC23_rset(hCodec, 0x0004, DSK6713_AIC23_INPUT_LINE);
	temp.combo = MCBSP_read(DSK6713_AIC23_DATAHANDLE);

	// mic
	short mic_sample = temp.channel[MIC];

	if(abs(mic_sample) > mic_max)
		mic_max = abs(mic_sample);

	if(abs(gainSin[gainIndex]) > s_max)
		s_max = abs(gainSin[gainIndex]);

	if(index == N - 1)
	{
		s_avg = s_avg + ((float)s_max - (float)s[max_index]) / N;
		mic_avg = mic_avg + ((float)mic_max - (float)mic[max_index]) / N;

		s[max_index] = s_max;
		mic[max_index] = mic_max;
		max_index = (max_index + 1) % N;

		s_max = 0;
		mic_max = 0;
	}

	index = (index + 1) % N;

	// output stereo
	temp.channel[0] = gainSin[gainIndex];
	temp.channel[1] = gainSin[gainIndex++];

	MCBSP_write(DSK6713_AIC23_DATAHANDLE, temp.combo);

	// process the results
	if(gainIndex > GAIN_LENGTH) {
		gain = mic_avg / s_avg;
		mic_max = 0;
		s_max = 0;
		init = 0; // end
		index = 0;
		max_index = 0;
		s_avg = 0;
		mic_avg = 0;

	    int i = 0;
	    for(i = 0; i < N; i++) {
		mic[i] = 0;
		s[i] = 0;
	    }
	}
}

// the main program
void mainStuff()
{
	float mic_max_gain;

	union {Uint32 combo; short channel[2];} temp;

	//Read the Line in
	//DSK6713_AIC23_rset(hCodec, 0x0004, DSK6713_AIC23_INPUT_LINE);
	temp.combo = MCBSP_read(DSK6713_AIC23_DATAHANDLE);

	// r
	short s_sample = temp.channel[LINE_IN];

	// mic
	short mic_sample = temp.channel[MIC];

	if(s_max < abs(s_sample))
		s_max = abs(s_sample);

	if(mic_max < abs(mic_sample))
```

```
		mic_max = abs(mic_sample);


	if(index == N - 1)
	{
		mic_max_gain = mic_max / gain;

		s_avg = s_avg + ((float)s_max - (float)s[max_index]) / N;
		mic_avg = mic_avg + ((float)mic_max_gain - (float)mic[max_index]) / N;

		// avoid division by zero
		if(mic_avg > 0)
			delta = (mic_avg - a * s_avg )/ (mic_avg);
		else
			mic_avg = 1;

		if(delta < -1)
			delta = -1;

		if(delta > 1)
			delta = 1;


		//float cost = delta/(1 + 0.28125*delta*delta) / (PI/2);

		float cost = atan(20*delta) / (PI/2);

		mu = MU_CENTER + MU_DELTA * cost * cost;

		//mu = .01;
		// apply the cost function
		a = a - mu * (NOISE_THRESHOLD - delta);

		a_avg = a_avg + (a - a_data[max_index]) / N;

		a_data[max_index] = a;

		// never get quieter than default
		if(a < 1)
			a = 1;

		if(a > 7)
			a = 7;

		s[max_index] = s_max;
		mic[max_index] = mic_max_gain;
		max_index = (max_index + 1) % N;

		s_max = 0;
		mic_max = 0;
	}

	index = (index + 1) % N;

	// output stereo
```

```
      temp.channel[0] = a*s_sample;
      temp.channel[1] = a*s_sample;

      MCBSP_write(DSK6713_AIC23_DATAHANDLE, temp.combo);
}
```