Adaptive Speakers: Listening Made Better

EECS 452 Final Report

Sophia Mehdizadeh, Jack Nonnenmacher, Socrates Papageorgiou, Abdulrahman Shehadeh
EECS 452, College of Engineering
University of Michigan
Ann Arbor, Michigan
skmehdi@umich.edu, jnonno@umich.edu, sdkpapag@umich.edu, aboodsh@umich.edu

***Abstract*—This paper examines the development of an adaptive speaker system. Our objective was to create a prototype that was completely modular, allowing the users to connect their own speakers and microphone, and control the system via a mobile application that could be downloaded to their own device. The product was implemented using a Raspberry Pi 3 and two STM32 Nucleo F7 boards. The success of the system was measured qualitatively through user feedback, and quantitatively by looking at frequency-domain metrics.**

I. Introduction

For many years, architects and engineers have struggled to design and build acoustically "perfect" concert halls and musical venues. The goal of these spaces is to allow sound to travel from the source (musician) to the listener as intact and untampered as possible. However, the design and construction of these spaces is extremely long, arduous, and expensive, and must be customized for each building shape and location. Realistically, because a single sound source might result in hundreds of sound wave reflections within a space (Fig. 1), it is extremely difficult for a sound to exist in a physical space without being impacted by that space [6][7]. This impact has most commonly been noticed in terms of reverberation. Large, empty rooms produce echoes, while heavily carpeted and furnished rooms absorb sound. However, not all frequencies are reflected or absorbed equally. The structure and arrangement of a space significantly impacts which frequencies are boosted or attenuated, and by how much. These environmental filters make it extremely difficult for the average listener to experience clear, undistorted audio. Our solution to this problem is not to eliminate these filters by creating acoustically "perfect" listening spaces, but rather to develop a product that can identify these filters and negate them by adjusting its audio output accordingly.

Fig. 1: Simulation of sound reflection in a space

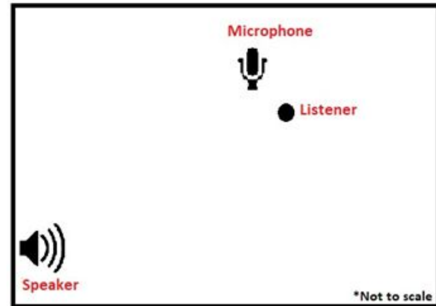

Fig. 2: Overhead view of example room layout (symbolic)

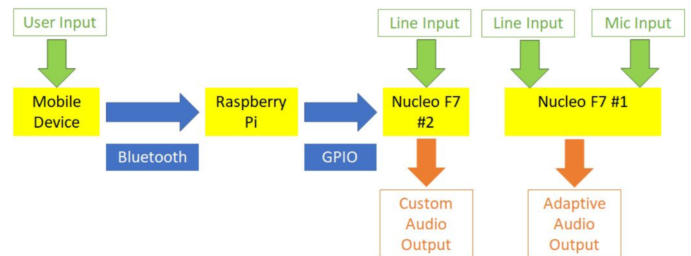

Fig. 3: Low-level system diagram

Our adaptive speaker system creates a more clear and authentic listening experience by canceling out the filtering effects of the listening environment. This negation is done by comparing what the listener hears to the original audio file. A microphone placed at the location of the listener (Fig. 2) picks up the room-filtered audio and sends it to a Nucleo F7 microprocessor (Fig. 3). The Nucleo then uses a digital signal processing (DSP) algorithm to determine the differences between room-filtered audio and the original audio file being sent to the speakers. This DSP algorithm will be discussed more in section IV. In the case that the user would like to directly customize their audio, a mobile application allows the user to add certain effects and filters, and choose between preset levels. The user interface and data transmission to the rest of the system will be discussed in section II.

II. User Interface

A. Mobile Application

The purpose of having a phone application is to let the user connect to the system via bluetooth and make custom changes to the audio by applying the available effects such as filters, delay or distortion. By choosing one of the eight available presets of pre-programmed effects, the slider of each

corresponding effect will automatically move to its chosen value; the user could turn on or off each individual effect by toggling on or off each effect's switch (Fig. 4). Swift is the official programming language of iOS development and was used to develop the application's functionality, graphical user interface (GUI), and the bluetooth connection. Xcode was used as the integrated development environment (IDE) for its similar affiliation to iOS development. The phone application establishes a connection with the Raspberry Pi by using Bluetooth Low Energy technology.

B. Bluetooth Low Energy

We chose to use Bluetooth Low Energy (BLE) technology for our system since it is ideal for situations where small data are being transferred for a long period of time. BLE is well-suited for providing fast transfer of small data while saving a lot of the device's battery. A BLE Profile consists of several services, with various characteristics for each service (Fig. 5). Characteristics contain a value, which is what the central either reads from, or writes to. Therefore a single characteristic may only be a read or a write characteristic, and should be defined by the peripheral before any connection occurs. Services act as a holder for characteristics that share similar functionality, or provide a single service. Both services and characteristics are defined by a 128-bit universally unique identifier (UUID) that describes a service or characteristic provided by an electronic device. A device that provides services and characteristics is called a peripheral, which in our case is the phone or other mobile device. A device that reads services and reads or writes characteristics is called a central. The Raspberry Pi was assigned the central role because it will communicate with all of the system's devices: the phone and the two Nucleo boards. The central will detect the peripheral's service(s) only when the peripheral advertises the desired service(s) [1].

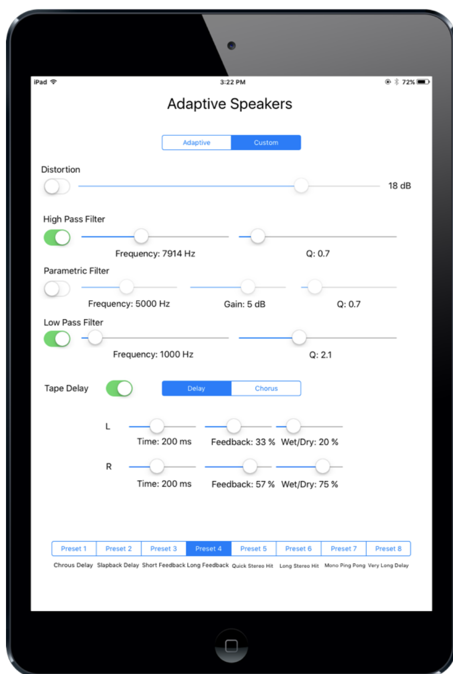

Fig. 4: Mobile application to select desired effects and presets

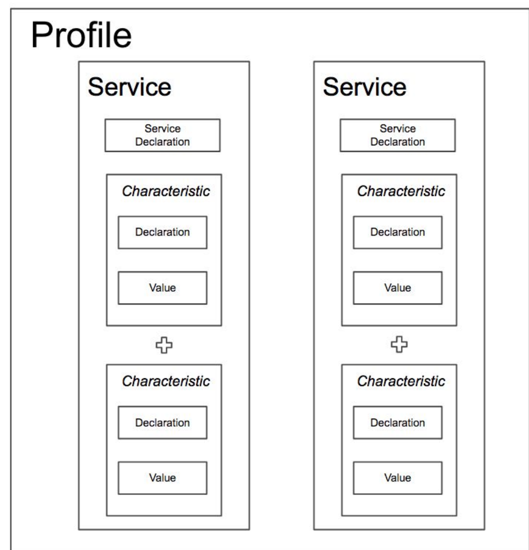

Fig. 5: Bluetooth Low Energy Profile Hierarchy

C. Peripheral Bluetooth Handling

For the peripheral (mobile device), the CoreBluetooth library was used in Xcode which provides the classes needed for the app to communicate with BLE wireless devices. We desired only one service that will hold two characteristics, a read and a write characteristic. To do that, three variables must be initialized to hold the 128-bit UUID (randomly generated and assigned) of each service and of the characteristic. These variables must be defined as constants because it is very crucial that they never change. This process is done by writing the following lines of code at the beginning of the executable file (Swift defines a constant variable by using the reserved keyword "let:"):

$$let\ service_uuid\ =\ =\ "e20a39f4-73f5-4bc4-a12f-17d1ad07a961"$$

$$let\ read_char_uuid\ =\ "08590f7e-db05-467e-8757-72f6faeb13d4"$$

$$let\ write_char_uuid\ =\ "886570b6-2e41-11e8-b467-0ed5f89f718b"$$

Next, three other variables need to be declared that will be the actual service and the two characteristics. These variables are objects of "CBMutableService" and "CBMutableCharacteristic" classes [2], respectively. While calling the constructor of the characteristic variable, the characteristic's UUID needs to be passed with a property that specifies whether this characteristic is a read or write. For the service variable constructor however, only the service's UUID has to be passed. As a final step, it is possible to assign characteristics to a specific service by calling the service object's member variable "characteristic" and passing the desired characteristics in an array, using the following code:

$$transferService.characteristics\ =\ [readCharacteristic!,\ \ writeCharacteristic!];$$

Finally, the service needs to start advertising at the application startup. Reading from and writing to a characteristic is straightforward, and is done by using the following two functions from the library [2]:

```
func peripheralManager(_ peripheral: CBPeripheralManager,
                       didReceiveWrite requests: [CBATTRequest])

func peripheralManager(_ peripheral: CBPeripheralManager,
                       didReceiveRead requests: [CBATTRequest])
```

It is important to note that all of the data transferred were formatted in hexadecimal, because the BLE protocol enforces the use of hexadecimal system.

D. Central Bluetooth Handling

The Bluepy library was used to handle the bluetooth low energy connection from the central, or the Raspberry Pi. It is a library written in Python that expands BlueZ's (official Linux Bluetooth stack) functionality. This library provides classes to discover and connect to the device that is advertising services, and allow the retrieval of the device's characteristics. The connection occurs by passing the device's bluetooth address, which is unique to every single phone. Then an object of the Peripheral class is created that will allow the retrieval of services that the phone is advertising by using the *getServiceByUUID()* function and their corresponding characteristics by using *getCharacteristics()* function. Once the characteristics are accessed and stored in variables that are objects of Characteristics class, it is possible to read or write to any characteristic – depending on its property – by using the *read()* or *write(data)* functions which are provided in the Characteristic class [3]. The data that are read and written are formatted using hexadecimal system.

When the Raspberry Pi receives data via BLE from the mobile device, it then needs to relay that information to the Nucleo board doing the DSP. We initially considered several different communication protocols to connect the Raspberry Pi to the Nucleo (Ethernet, UART, SPI). Due to hardware and software library limitations, as well as time constraints, we settled on using general purpose input/output (GPIO) pins to connect the two boards. The Raspberry Pi 3 has 30 GPIO pins (Fig. 6) while the Nucleo F7 has upwards of 100. To maintain simplicity, we used only 13 GPIO pins on each board for this communication -- five to represent each of the five different effects on the user interface, and eight more to represent each of the presets. The 13 pins on the Raspberry Pi were set up as output pins. When the Raspberry Pi received which effects and presets were on/off from the mobile device, it would set the corresponding GPIO pins HIGH or LOW respectively. The 13 pins on the Nucleo were set up as input pins to allow them to read the HIGH or LOW values being sent by the Raspberry Pi. By reading the GPIO input pin levels, the Nucleo would determine which effects to apply to the audio signal. The different effect options and algorithms will be discussed in section III.

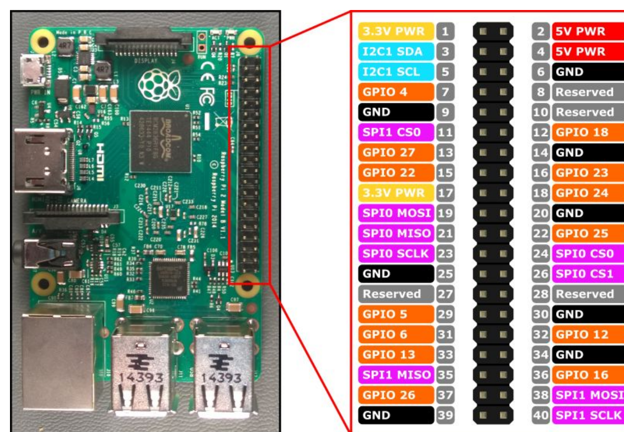

Fig. 6: Raspberry Pi GPIO pin diagram

III. Effects

Three types of audio processing effects are available for the user to apply using the phone application: filters, delay and distortion. The filter effects consist of three band filters: low-pass, parametric, and high-pass. The parameters available to choose for these filters are frequency cutoff, which is beyond which frequency one desires to cut or let pass, and the quality control or Q which defines the width of the cut or boost. Additionally, the parametric filter has a third unique parameter which is gain, and it is possible to boost or attenuate the chosen frequency along with the Q parameter. The delay effect consists of a stereo delay where the left and right channels have their separate parameters, which are the delay time, the feedback percentage which specifies how much of the output signal is fed back to the input, and wet/dry percentage which specify how much of the processed signal will be mixed with the unprocessed signal. Finally, the distortion consists of a nonlinear input-output mapping that behaves linearly at low input levels but saturates at high input levels. The output sample value is acquired by raising the input sample value to the power of the inverse of small, odd integers.

The process of applying any effect is identical: the processor reads a buffer of 1024 samples for each channel by indexing each individual sample in a for loop, the sample variable that is currently being read by the processor is assigned to the value that the effect's function returns. Each sample is represented by a floating-point data with a dynamical range of [-1, 1], which is the standard data format for most of digital signal processing systems.

Some information from the DSP book [4] was used only in the following two sections, filters and delay.

A. Filters

Filters were implemented by using analog designs in frequency-domain and converting them to Linear Constant Coefficient Difference Equations (LCCDE) in time-domain. The reason behind designing filters in time-domain, opposing to frequency-domain, is because filters in time-domain are simpler to code and are much more efficient in terms of computations needed. This result is due to the fact that filters in time-domain consist of a delay algorithm that delays the

input and output sample twice – because a second order filter was used – at each iteration and multiply different coefficients for each delayed sample depending on the filter type. It should be emphasized that the three type of filters employ the exact same delay algorithm and only the coefficients' value is the factor that specifies the filter type.

The process of converting an analog filter into LCCDE begins by choosing an analog design of a filter. For instance, a generic resistor capacitor (RC) second-order low-pass filter was chosen in our case (Fig. 7), which has the following transfer function in frequency-domain:

$$H(s) = \frac{V_o(s)}{V_i(s)} = \frac{1}{s^2 R_1 R_2 C_1 C_2 + s(R_1 C_1 + R_1 C_2 + R_2 C_2) + 1} \quad (1)$$

The next step consists of converting this analog design in frequency-domain to a digital representation while still in frequency-domain, and this conversion could be done by using the Bilinear Z-Transform (BZT), which is a first-order approximation of the natural logarithm function that is an exact mapping of the *s*-plane to the *z*-plane by using the following *s* to *z* relationship:

$$s = \frac{2}{T} \frac{z-1}{z+1} \quad (2)$$

where T denotes the sampling period. To overcome the poles that lie outside Nyquist's range in the *s*-plane, the BZT wraps the exceeding poles around the unit circle by using a tan function (Fig. 8), which is a great approximation of the analog's design. By replacing each s with the BZT's equation, we get the following digital transfer function in frequency-domain:

$$H(z) = \frac{a_2 Z^{-2} + a_1 Z + a_0}{b_2 Z^{-2} + b_1 Z^{-1} + 1} \quad (3)$$

Where the coefficients *a* and *b* depend on the filter used. Once the filter's representation is obtained in digital frequency-domain, converting it to a digital time-domain representation (Fig. 9), or LCCDE, is done by using the inverse Z-Transform to obtain the following LCCDE:

$$y(n) = a_0 x(n) + a_1 x(n-1) + a_2 x(n-2) - b_1 y(n-1) - b_2 y(n-2) \quad (4)$$

Translating the LCCDE equation, along with the corresponding coefficients could be easily done by making a function that takes the current sample input x(n) as its argument, and returns the current output sample y(n) which is computed from the LCCDE Equation 4. Four variables are needed to store the four delayed samples from the input and output. The function starts by reading the stored delayed samples, compute the output y(n), delay the four variables by one sample for the next iteration and returns the output value. This process should be duplicated for the right channel of audio, and it is necessary to create different variables for each channel. The other two filters use the exact same approach, however only the coefficients that gets multiplied by each delayed sample will be different.

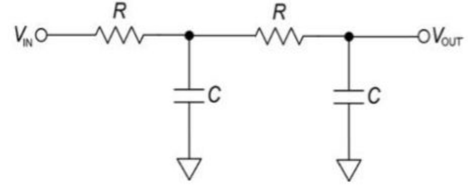

Fig. 7: A generic analog RC second-order low-pass filter

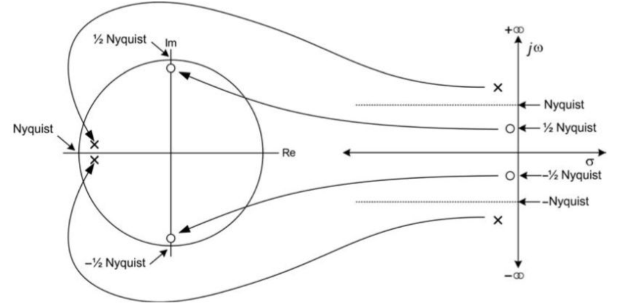

Fig. 8 : The BZT's act of mapping the *s*-plane to *z*-plane

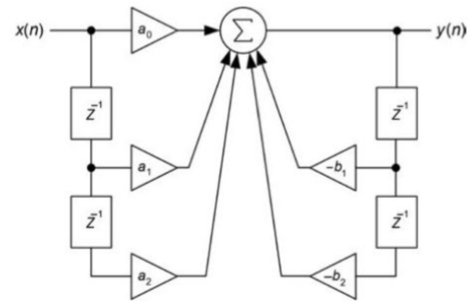

Fig. 9: Generic biquad structure of a second-order filter

B. Delay

The delay algorithm (Fig. 10) consists of using a delay line register to store the samples to be delayed. The samples are stored in a buffer (Fig. 11), or array that is fixed in size throughout the entire runtime, otherwise, a lot of errors could occur such as indexing memory location that are out of bound which will crash the system. The size should be specified after the code compiles, but before it runs, because it is necessary for the processor to know how much memory should be allocated for that buffer. The buffer's size corresponds to the maximum amount of delay time desired, and should be expressed in terms of the sample rate, because different systems have different sample rate. It is computed by choosing the maximum time in seconds, which corresponds to x in the following equation:

$$maximum\ delay\ (samples) = x\ (seconds) * sample\ rate\ (\frac{samples}{seconds}) \quad (5)$$

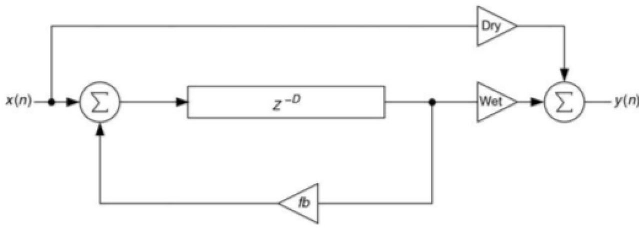

Fig. 10: Delay system diagram

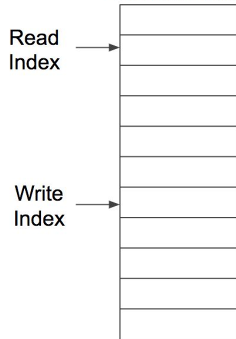

Fig. 11: A delay buffer with read a write indices

Two index variables should be declared as integer variables and they will serve as the array's read and write index (Fig. 11). At this point, it is necessary to treat the buffer as a circular buffer to save memory space, especially since the two indexes will be constantly incrementing their value by one for each iteration. A circular buffer is a buffer where its two indexes wrap to the beginning of the buffer once any of them reach the end of the buffer, which is achieved by the following pseudocode:

$write\ index\ =\ write\ index\ +\ 1;\ (incrementing\ the\ write\ index)$

$if(write\ index\ >=\ maximum\ delay)\ write\ index\ =\ 0;$

The read index follows the same approach. The write index will copy all the samples to the new allocated buffer whereas the read index will read the stored value depending on the delay time chosen. This fact means that the write index will always be incrementing by one, and wrapping if necessary, however, the read index could jump to different values, especially if the user changes the delay time during playback. Whenever this change happens, the read index's new value should equal to the write index's value minus the new specified delay time (in samples), as shown in the following pseudocode:

$read\ index\ =\ write\ index\ -\ delayInSamples;$

The *delayInSamples* value corresponds to a conversion of the specified delay time in seconds into samples. Since it is possible that the new read index's value to become negative, it is necessary to check if read index's value is negative after applying that code, and in case it is, it is wrapped back to its equivalent positive value in the circular buffer by adding the maximum delay time value, as shown in the following pseudocode:

$if(read\ index\ <\ 0)\ read\ index\ =\ read\ index\ +\ maximum\ delay;$

In addition to the delay time parameter, the user could specify the feedback and wet/dry percentages. The feedback percentage corresponds to how much percent of the output sample's value should be fed back to the input, which also means fed back to the buffer again (a feedback of 100% corresponds to playing the delayed sample infinite times). The wet/dry percentage corresponds to the mix between delayed and non delayed signal. For example, a wet/dry value of 30% will play 30% of the delayed sample's value y(n) and will be mixed with 70% of the non delayed input sample's value x(n). The LCCDE that relates the feedback parameter and the input and output relationship that the delay function will return is the the following equation:

$$y(n) = x(n-D)\ +\ feedback * y(n-D) \quad (6)$$

To apply the wet/dry parameter's value, the value that the delay function will return should include the wet/dry amount, as shown in the following pseudo code:

$$return\quad (1-\frac{wet/dry}{100}) *\ x(n)\ +\ \frac{wet/dry}{100} * y(n);$$

Note that in this pseudo code, x(n) denotes the input variable that is passed to the delay function, or in other words, it is the sample that the processor is currently reading, and y(n) is the value of the delayed signal that the delay function computes using Equation 6.

Because the delay system is stereo, it is necessary to repeat this entire process twice for each channel, and because the user could choose different parameters for each channel, it is necessary to allocate two buffers for each channel, as well as, duplicate all the variables used earlier. Depending on the delay time chosen, it is possible to achieve a flanger effect by choosing a delay time between 1ms and 10ms, or a chorus effect by choosing a delay time between 10ms and 20ms.

C. Distortion

In the analog world, distortion is often caused by an overdriven transistor or tube. While the inner workings of such devices deserve a paper (or many) of their own, they can be very simply be described as non-linear devices whose nonlinearity increases with increased incoming voltage. At low voltages, the behavior is essentially linear. It is only as the voltage increases that saturation occurs. If great precision is not a concern, these devices can be modeled in the digital world.

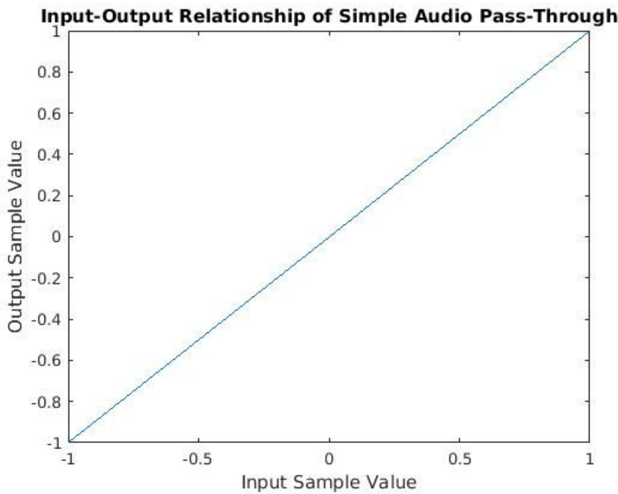

Fig. 12: Input-output relationship of simple audio pass-through

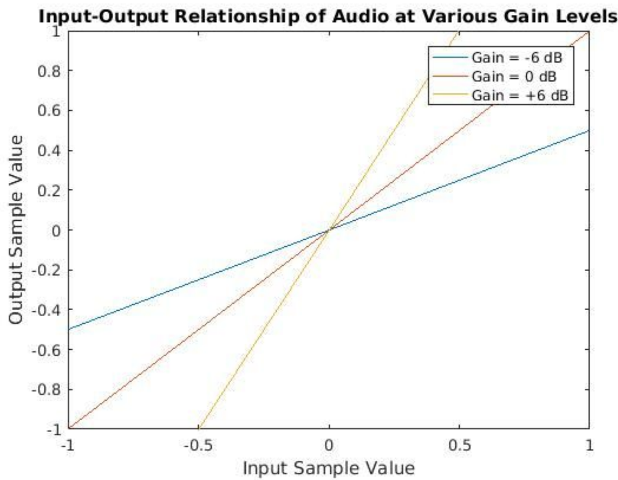

Fig. 13: Input-output relationship of audio at various gain levels

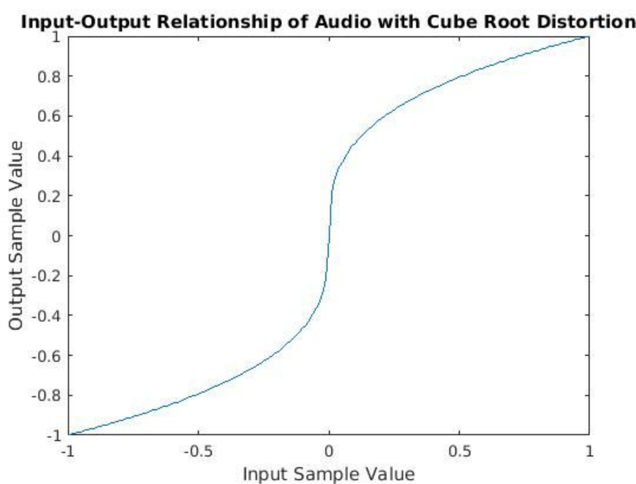

Fig. 14: Input-output relationship of audio with cube root distortion

In order to visualize non-linear effects, one useful tool is simply a 2-dimensional graph of the input-output relationship. For example, a simple audio pass-through (Fig. 12) would appear as the line $y = x$, where x is the incoming audio and y is the outgoing audio. We will assume the sample value range is $[-1, 1]$.

In this representation, the gain is simply the slope of the graph. Increasing or decreasing the gain (Fig. 13) simply increases or decreases the slope of the graph.

This can be easily verified. For example, at the input sample value of $x = 1$, when the gain is $Gain = -6\ dB$, an output value of $y = 0.5$. The same value does not map to an output at $Gain = +6\ dB$, for such a value would be outside the digital ceiling.

As previously mentioned, the distortion algorithm creates a nonlinear input-output relationship (Fig. 14). Specifically, the relationship $y = x^{1/3}$ creates a distortion not unlike that created by a transistor.

Clearly, this has created a nonlinear input-output relationship. Close to zero, the relationship seems to be linear with slope $m > 1$, that is, the gain is being increased. With input $|x| > 0.5$, the relationship seems to be linear as well, albeit with slope $m < 1$, that is, the gain is being decreased. Between these two extremes, the graph is nonlinear. This change in applied gain as a function of input level is what creates a distortion sound.

It is perhaps worth noting that not every type of distortion can be modeled with a simple input-output equation. Although this type of algorithm is efficient and can accurately model many analog distortions, the output of more complicated algorithms may rely on the input as well as past inputs and outputs. In other words, the input-output equation may change depending on if the sample is, for example, part of a transient as compared to a sustained sound.

One may wonder at this point about the significance of the seemingly arbitrary cube root. Why not a square root design? The answer is more pragmatic than profound; when applied to negative numbers, the square root algorithm produces complex numbers. If so desired, one can modify the algorithm to account for this, such as in the following pseudo-code:

$$input = abs(input)/input \ * \ pow(abs(input),\ 1/2);$$

A parameter that may be introduced is perhaps analogous to the “Drive” knob on a guitar amplifier. As the algorithm uses reciprocals of greater and greater odd integers, the signal distorts more and more. As the integer in question continues to increase, the distortion of the output signal intensifies (Fig. 15). In the limit as this integer approaches infinity, the input-output relationship begins to resemble a square wave. At this point, the audio would no longer be recognizable. The algorithm, even at its mildest setting, produces a noticeable effect on a pure sine tone (Fig. 16).

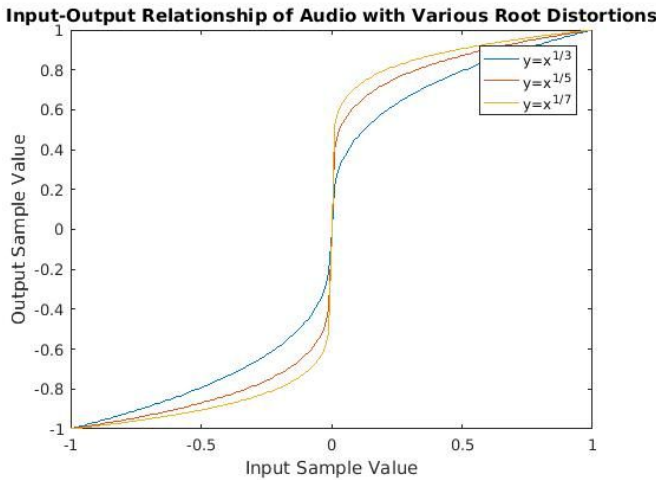

Fig. 15: Input-output relationship of audio with various root distortions

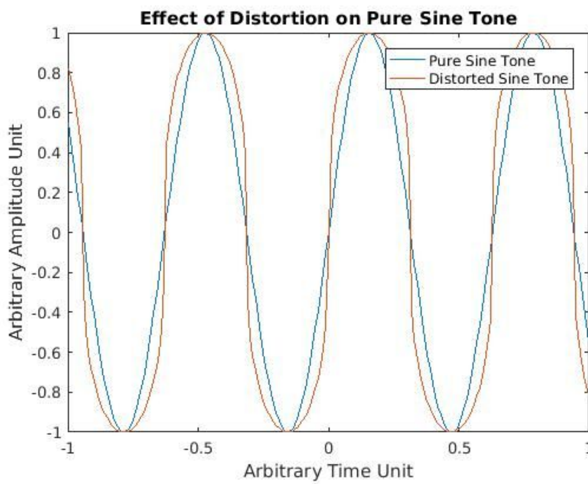

Fig. 16: Effect of distortion on pure sine tone

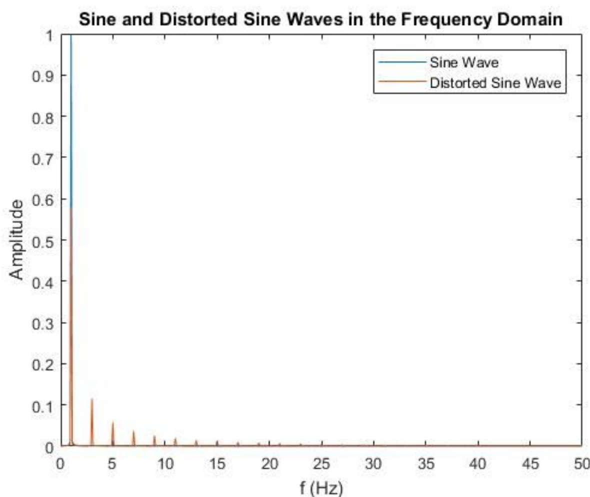

Fig. 17: Sine and distorted sine waves in the frequency domain

In the time domain, the effect can perhaps be described as "beefing up" the sine wave. At every point on the wave, the magnitude of the distorted sine tone is equal to or greater than that of the original sine tone.

In the frequency domain (Fig 17), this adds frequencies that are integer multiples of the frequency of the inputted sine wave. For example, if the input was a sine wave at $f_0 = 100Hz$, the distorted sine wave would contain frequencies of $100Hz$, $200Hz$, $300Hz$, etc.

Upon implementing this algorithm, however, it was found that there is no efficient way to vary the power or root of a float number efficiently. Instead, this parameter was fixed at 3. The "drive" parameter then applied a pre-gain, and, if this resulted in sample values higher than 1 or lower than -1, they were simply clipped (Fig. 18). Such an operation looks like this in pseudo-code:

$$output = pow(input * gain,\ 3);$$

$$if\ output > 1,\ then\ output = 1;$$

$$if\ output < -1,\ then\ output = -1;$$

This provided an acceptable approximation and was much more computationally efficient. While there are many ways to produce audio distortion in the digital domain, this algorithm and its approximation best met the required specifications.

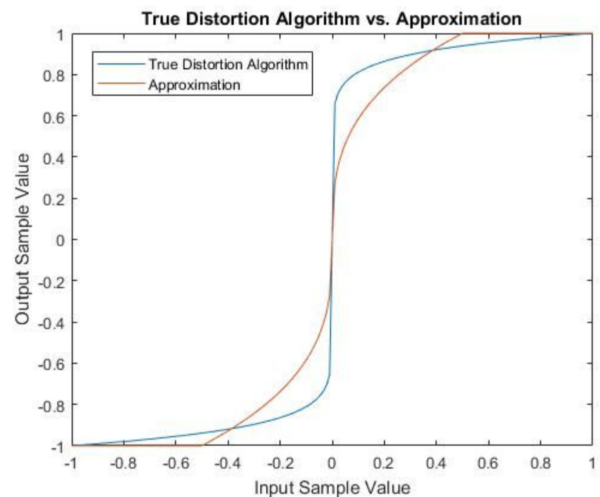

Fig. 18: True distortion algorithm vs. approximation

IV. Adaptive Filtering

A. Basic Concepts

Adaptive filtering is traditionally described as a system with a filter in it that changes based on a set of variables that is passed into the system which then results in an optimization of the filter. Over time the filter will improve and become more accurate to the needs of the system which then will produce the desired result. This is usually done by calculating the mean square of the error signal which can then be used to represent a cost function which is just a way to measure how well the filter is doing to modify the signal and reach an ideal. This

error is derived from comparing the output signal to the desired signal. Eventually this error value will approach zero which means that the filter will only make minimal adjustments as long as the input signal does not have any significant variations or change in pattern.

For this project's implementation of adaptive filtering, the ideal signal, or desired signal, is the input to the system while the output signal is the result of the system's filtering. By changing the transfer function of our system there will eventually be a near equality between the input and output of the system.

B. Theory

Although the result of this project is a perceived adaptive filter, the theory behind it is a transfer function trick that is done without any feedback other than measurement of the output signal. Because signals can be multiplied, divided, and inverted in the frequency domain, there are clever mathematical manipulations that can be made to force equality between the input and output of the system. Initially the system can be described by the input, output, and transfer function H:

$$Input * H = Output \qquad (7)$$

In this case the transfer function H is the effect of the room on the input signal when it is played through the speakers. The input is initial signal that is put on the wire going into the speakers and the output is what is measured by the microphone after the room has affected and filtered the sound. The transfer function as well as the inverse transfer function of this system can be determined by simple division, again, in the frequency domain. The result follows below:

$$H = \frac{Output}{Input} \qquad (8)$$

$$H^{-1} = \frac{Input}{Output} \qquad (9)$$

With the well defined calculation of the transfer function, the inverse of the transfer function can be found and multiplied by the input signal so that the transfer functions cancel and leave only the input and the output left in the equation:

$$Input * H^{-1} * H = Output \qquad (10)$$

$$Input = Output \qquad (11)$$

Because of the way that this multiplication works, the output exactly equals the input and the signal should be theoretically perfect when the output and input are compared. As this formula was implemented on the STM32 Nucleo F7 hardware it became increasingly clear that this theory does not work quite the same in practice. The Nucleo F7 board is pictured here (Fig. 19) for the reader's visualization.

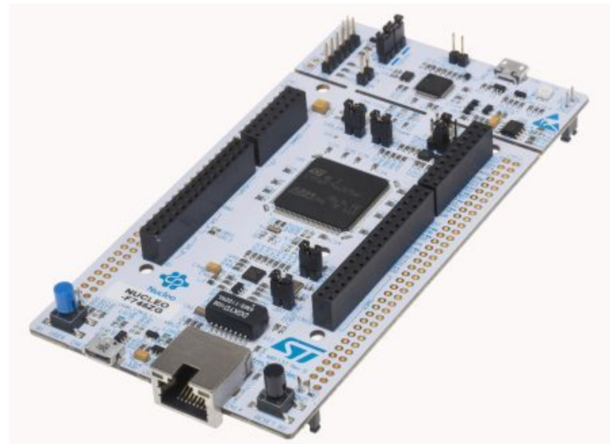

Fig. 19: STM32 Nucleo F7 development board

C. Implementation

The initial implementation of this system was first tested in MATLAB on finite data sets that consisted of an input file and a filtered version of that input file to act as the microphone signal. This was primarily to test the theoretical mathematics and to confirm that this solution architecture really could work on audio signals. MATLAB allows for quick and painless operations on complex values and easy transition between the time and frequency domain which allowed the team to have a solid understanding of whether or not the algorithm would work or not. After testing and confirming on multiple sets of input audio the team moved the solution onto the Nucleo F7 board for implementation in the C programming language.

When moving from the MATLAB testing to the embedded systems implementation there are a number of items that the team had to take into consideration in order to have a functional system. The first of these considerations is that the system must be real-time and be able to receive audio, apply the adaptive algorithm, and send audio back to the analog world without any breaks or pops. This means that the algorithm must process the incoming samples faster than they arrive so that they can be output immediately. Another consideration that is important to the success of the code conversion is the fact that the data structures in C are not easily set up to handle complex values and therefore all operations, even simple elementwise operations, must be carefully set up so that the data is not altered in ways other than intended while going through the algorithm. This is particularly important in inverting the transfer function and multiplying the inverse transfer function and the input signal together. In addition to the real-time and complex factors of the system, another parameter that must be optimized is the size of the input and output audio buffers as well as the corresponding size of the Fourier transforms that must be applied to these buffers. For this particular implementation, the algorithm would run when the size of the buffer was set to be 512 samples long with a sampling rate of 48 kHz. Using larger buffer sizes made it difficult to push samples through the system in real time and therefore a smaller buffer size had to be used to eliminate any drops in the audio.

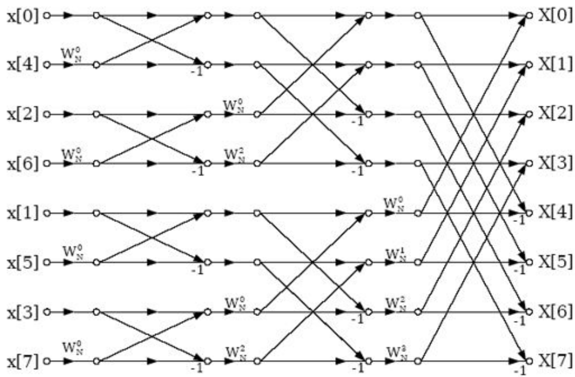

Fig. 20: Radix-8 butterfly FFT algorithm diagram

The Fast Fourier Transform that was used to convert the time based samples into the frequency domain and back was from the CMSIS DSP library. This uses a floating point precision function to calculate the fourier transform or inverse fourier transform of a set of complex values where the real values are the even numbered indices and the complex values are the odd numbered indices. The function that operates on the audio buffer takes in a pointer to the instance of the transform, a pointer to the data that is being operated on, and two flags that indicate whether or not the fft will be a regular or reverse fft and if the bits are reversed or not. This function is called every time there is a transition from time domain to frequency domain or back. Part of this algorithm is pictured above (Fig. 20) and illustrates the number of operations that must happen for just 8 bits of data to be transferred into the frequency domain.[5]

For more specifics on where the FFT is used, it is used immediately to translate both input audio and mic audio into the frequency domain. While in the frequency domain, the multiplication is done and then once the output signal has been determined, the FFT is used again, this time in reverse, to convert the output signal to time domain. These computations are very intensive for the hardware and require a lot of processing power to complete which is a major factor as to how the code is written and implemented. Some of the calculations must be done every single time the audio buffer is taken in and others are only done when necessary.

Going through each step of the algorithm, there are some clear points where computational complexity could be reduced by only allowing an FFT to be taken at regularly spaced intervals. The sampling of audio and output of audio has to happen every single time the program loops so there's not much room for reduction there. However, the calculation of the transfer function can be delayed because of the fact that the transfer function will always be stored in memory and won't be overwritten with the next set of input audio. The other reason that the transfer function doesn't need to be computed every time is because in the actual application of this system, the transfer function will not be changing quickly or drastically so there would be no reason to calculate it with every set of samples. Given this, the implementation that was chosen is to create a sub-counter that waits until it is reset at a large value before recalculating the transfer function.

D. Challenges

Though this implementation is theoretically and mathematically robust, there are a number of factors that come into play and affect how the system reacts to the microphone input. This has to do with the way that the microphone in the room is sampled, how large of a transfer function is used to calculate the new output, and a few other computational factors that make it difficult to qualitatively hear quick changes in the filtering system.

When the system receives microphone input to calculate the transfer function by comparison to the ideal signal, the system and math assumes the same amplitude between both signals and does not account for how much input is coming into the microphone. A possible solution to this could be implementing a normalization function for the input so that all audio levels would be the exact same. However, this could harm the quality of the audio and result in a more low fidelity system which would have the opposite effect of the goal of the system. Instead, the solution that was used to fix the levels was to use a pre-amplifier to bring the microphone to a correct level before inputting the signal into the hardware. This calibration has to be done depending on what microphone is used in the system but is generally not too difficult to do just by listening closely to if the audio is being affected by the mic placement relative to the speakers.

The primary issue that affects the filter's performance is a combination of the size of the FFT that is used i the algorithm and the computational complexity of the algorithm as a whole. The reason that the FFT size is only 512 samples long is because the system can't process the audio and output it in a continuous stream when longer buffer sizes are used. This could be solved by using a 16 bit FFT instead of a floating point FFT which would help with the number of computations but would hurt the fidelity of the audio. The problem with having an FFT of this size is that it is particularly short and because it's only able to be updated so infrequently because of the inability of the system to update every cycle, the calculated transfer function is often taken from input that does not give a good representation of the transfer function of the room which means it may take another update to hear any difference or may make the difference very subtle and difficult to hear.

Possible solutions to this problem could include getting hardware that is fast enough to make all the necessary calculations on larger FFTs while updating at a faster rate in order to keep the transfer function relevant to the signal. It could also be beneficial to average the transfer functions that are calculated over time so that even if there a few transfer functions that are not quite representative of the room, the overall transfer function will still be effective when applied to the input signal.

We found that after hitting the computational limit of the Nucleo F7 many times it was difficult to add more tasks to the list of things that might have to be done within a single cycle such as output data or communicate with another board entirely. For this reason, it was difficult to obtain output from the board that is from a real-time test and not from a MATLAB simulation. The bottom line is that this particular algorithm either needs to be optimized and changed to run on the Nucleo F7 or needs to be run as it is on a faster processor.

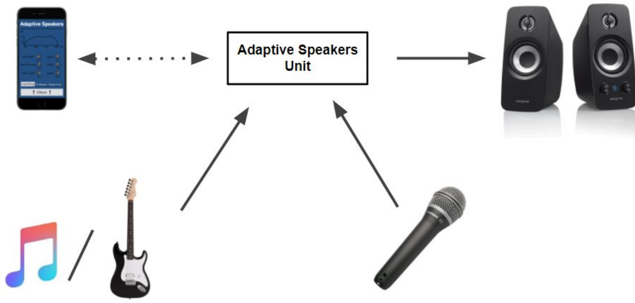

Fig. 21: High-level block diagram

V. Results

A. Overall Project Outcome

The Adaptive Speakers project was a success in that it evolved and changed scope to become a reasonable product given the time and scope constraints. The primary functions of the two different aspects of the project worked in a satisfactory way and could be combined into a workable and demonstration ready product. The high level block diagram of how the final system works can be seen above (Fig. 21). The effects section of the project produced 960 different combinations of effects that could be applied to any input in real time. The adaptive filtering aspect of the project was able to filter input in real time using a microphone in the room and applied subtle yet audible filtering to the audio depending on where the microphone was moved. There are still many optimizations to be made to the project but given the time constraints and scope of the course, the final product was satisfactory.

B. Plan Modifications

Throughout the project process there were many times that the solution architecture was changed due to finding better solutions, having difficulty in communication implementation, following instructor recommendations, and reducing the scope of the project and class. In an unfortunate series of events, the project team tried many forms of communication between the Nucleo F7 boards as well as the Raspberry Pi 3. These forms included but are not limited to SPI, UART, ethernet, and bluetooth. There were a number of problems with finding correct information and documentation about how these communication protocols worked with the Nucleo boards and this proved quite difficult to troubleshoot. This resulted in multiple drastic changes to the solution architecture of the entire system and eventually split up the two Nucleo boards so that they do not communicate with each other at all. This made the overall product much more robust and consistent but eliminated some flexibility and features from the product.

The initial design for the adaptive speakers unit involved the two Nucleo connected through UART while the Raspberry Pi was connected to the first Nucleo via ethernet and connected to the iOS mobile device via bluetooth. A more detailed account of this preliminary design is described in the flowchart below (Fig. 22). The team found that they could effectively implement the bluetooth connection to the Raspberry Pi but there was no clear way to implement an ethernet connection using the Nucleo. The

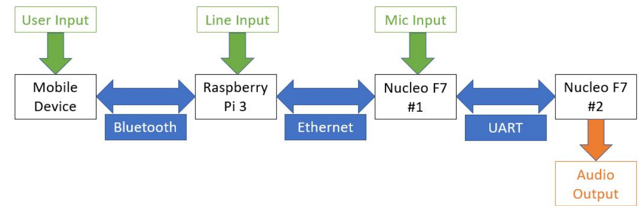

Fig. 22: Preliminary signal flow diagram

UART connection between the Nucleo boards was documented and usable but because the initial design was passing audio packets between the two devices, the connection had to be perfect. The UART connection between the Nucleo was abandoned because of an issue of dropped packets which put holes in the audio stream and therefore affected the way that the algorithms processed the audio data and determined the adaptive filter. The block diagram below shows the original plan for the device communication.

With the option of ethernet connection between the Nucleo and Pi looking bleak, the team tried an implementation of SPI between the two devices because the data only needed to include controls data and nothing more. During the second week of controls testing using SPI the team discovered an issue with the stability of the data being passed over the wire and it was recommended that we use a different protocol due to the fact that the functions for SPI on the Nucleo were not well documented, easy to understand, or supported by other developers who also complete projects on the Nucleo.

It should also be noted that at this point there would be no more audio data passed between devices as a simple splitter was being used to put input into both of the Nucleos via clever wiring. This eliminated the need for any communication between the Nucleos and insured that the audio stream would be consistent between the two devices as well as within the two devices.

At this point the solution architecture has split the project into two parts and has left the effects Nucleo unable to receive controls from the Raspberry Pi. To remedy this, the team developed a 13 pin GPIO configuration that could do an effective one-way control communication protocol from the Pi to the Nucleo. This control system was significantly more limited than the planned serial communication but because of time constraints this was the best option to insure the functionality of the device. There would not have been enough time to effectively implement our own communication protocol and debug and test the system so we reduced the scope of the project and made sure that what we did implement worked consistently and with a non-trivial specification.

In this way, the project evolved from a multi-device communication network into a simple GPIO controlled effects rack that is functional but not necessarily as customizable or flashy as a packeted communication controls system.

C. *Future Steps and Development*

With the project in its current state, there are many directions and opportunities that the team could take with the technology developed in this project. Because adaptive filtering is only available within specific products on the market, having a device or piece of software that can be implemented on any system in a modular fashion could change the way that consumer set up their audio systems.

As a hardware product, the adaptive system would need to be refined to handle the heavy workload that the algorithm demands so that consumers could have a simple and easy experience with the unit. The biggest roadblock after the refinement of the algorithm would be improving the quality of the ADC and DAC converters on the unit so that they could provide a much higher fidelity experience rather than bit crush incoming and outgoing audio.This is yet another reason why it might be effective to transfer this technology to a software platform to develop it further, allow for increased modularity, and give it an easy way to spread to users with minimal production cost.

As a pure software product, the adaptive speakers module would run as a plugin or standalone program on a user's computer. This program would be placed on the master output audio stream of the machine and would take input from a microphone that users would have to route through the application. For users with audio interfaces this could include microphones that are connected via DAC and ADC converters and for users that only have access to their laptop it could just use the built in microphone to detect the sound of the room. The already modular technology becomes even more modular when it can be fit onto any piece of hardware through flexible programming. This type of software is immensely popular and there is a huge market for downloadable DSP as shown by just a few examples in the following image (Fig. 23).

Another advantage to using this technology as a pure software product is that the algorithm no longer needs to handle the analog to digital conversion, the low quality that comes with using a bad converter, or any kind of multithreading that would take place to make the program run more smoothly. All of these factors make the software much more computationally feasible and practical for any user in any demographic.

Fig. 23: Distributed studio technology examples

With this in mind, there are many applications of this technology that could work as a software or hardware DSP implementation. Because this technology adapts to any space it is used in, it can be used for live concerts, recording studio settings, and even for simple home acoustics treatment. A more robust iteration of this technology could provide an alternative to paying for expensive acoustic treatments by simply identifying exactly what the space does to the sound and then eliminating it.

As more development offers are communicated to the team, there are further options that we could take as developers to use this technology in the real world. The team is excited to be able to apply the project to practical uses and looks forward to seeing a more refined version of Adaptive Speakers on the market.

VI. Conclusion

The Adaptive Speakers project was successfully demonstrated at the University of Michigan's 2018 Engineering Design Exposition. The final product consisted of a microcontroller system that takes input audio, filters it in either a custom or adaptive way, and returns the output audio to be played through speakers. The adaptive filtering mode relies on a microphone in the room while the custom filtering and effects happen through an iOS user interface that sends signals to control the parameters of the system.

The effects implemented in this system include three types of equalization filters, distortion, and delay. The effects can be turned on independent of each other and will change parameters based on which preset is selected. The combination of these effects allow for 960 different custom effects to be created. The system is robust and can take any signal as input and can be used for live performance.

The adaptive filtering mode works in a subtle way but still completes the task and proof of concept of changing the output of the audio based on the detection of the room by the microphone. On a large scale, when the microphone receives mostly low frequencies, the low frequencies will be filtered down to their original levels. When the microphone receives more high frequency content, the system will reduce the amplitude of the high frequencies in order to compensate for the discrepancy between the input signal and the signal received by the microphone.

Through many design and solution architecture changes, the team found a way to successfully connect the necessary elements to create a functional product. The changes from ethernet to UART to SPI to GPIO communication got progressively simpler but this allowed the team to create a more robust solution without the complications of packeting and creating new communication protocols to use on the Nucleo boards. Along with the simpler communication protocols came limitations of the software and controls system but this also offered a well defined system that was quick to debug and worked consistently.

Overall this project was a matter of finding the limits of the available hardware, adjusting the scope and goals of the product, and pushing those limits to create a product that is both a proof of concept as well as marketable in the up and coming adaptive filtering field. The team enjoyed pushing the envelope in this technology and had a thorough educational experience in creating and completing this project.

Acknowledgments

We would like to acknowledge Professor Hun-Seok Kim, Dr. Kurt Metzger, Siddharth Venkatesan, and Carl Steinhauser for their support and advice throughout the entire project. Their guidance and instruction provided a solid foundation for us to build on to complete our project and success would not have been possible without them. Thank you for offering your knowledge and expertise to the class!

References

[1] K. Astebol. " Intro to Bluetooth low energy and BLE development with Nordic Semiconductor" *YouTube*, Oct 6 2016. [Video file]. Available: https://www.youtube.com/watch?v=pLgnHuGI69s&t=1257s. [Accessed: 21-Feb-2018].

[2] Apple Inc, "Core Bluetooth," *developer.apple.com*. [Online]. Available: https://developer.apple.com/documentation/corebluetooth. [Accessed: 22-Feb-2018].

[3] I. Harvey, "bluePy," *GitHub*. [Online]. Available: https://github.com/IanHarvey/bluepy [Accessed: 10-Feb-2018].

[4] W. Pirkle, *Designing Software Synthesizer Plug-Ins in C : For RackAFX, VST3, and Audio Units*, 1st ed. Focal Press, 2014.

[5] ARM-software, "ARM-software/CMSIS," *GitHub*. [Online]. Available: https://github.com/ARM-software/CMSIS/blob/master/CMSIS/DSP_Lib/Source/TransformFunctions/arm_cfft_f32.c. [Accessed: 20-Apr-2018].

[6] Acoustics Insider. (2018). *My Room's Frequency Response Seems Very Uneven — Acoustics Insider*. [online] Available at: http://www.acousticsinsider.com/frequency-response-uneven/ [Accessed 20 Apr. 2018].

[7] Gikacoustics.com. (2018). *What Are Room Modes - GIK Acoustics*. [online] Available at: http://www.gikacoustics.com/what-are-room-modes/ [Accessed 20 Apr. 2018].

Team Contributions

A. Sophia Mehdizadeh

Sophia Mehdizadeh worked on prototyping and testing the adaptive filtering algorithm in MATLAB. She then moved to the Raspberry Pi where she worked on developing and debugging a communication protocol between the Pi and the Nucleo board. Sophia also wrote the Python code on the Raspberry Pi to relay the data coming from the mobile device to the Nucleo via GPIO.

B. Jack Nonnenmacher

Jack Nonnenmacher contributed to writing, debugging and testing the MATLAB prototypes of the adaptive filter algorithm, the distortion algorithm, and other experimental algorithms. He then wrote, debugged and tested prototypes for the adaptive filter algorithm and the distortion algorithm in C to be used for the hardware implementation. Jack also provided audio files to be used to test various algorithms used in the project.

C. Socrates Papageorgiou

Socrates Papageorgiou is responsible for the solution architecture and code implementation of both the adaptive and effects section of the Adaptive Speakers module. His primary focus was on writing and testing C code on the STM32 Nucleo F7's as well as working to develop communication protocols between the different devices. Socrates also handled and orchestrated the live demonstration setup as well as provided the necessary materials and audio engineering knowledge to successfully display the final product.

D. Abdulrahman Shehadeh

Abdulrahman Shehadeh worked on establishing a bluetooth low energy connection between the phone app and the Raspberry Pi. He was also responsible for developing the phone app's functionality and GUI using Swift programming language and Xcode IDE. Additionally, Abdulrahman designed and implemented the three filters and the stereo delay algorithms in C code.