EE 41440 Senior Design II - *Guitar Effects*

Jack Doherty, Sydney Heller, Luis Hernandez, Henry Van Ess

University of Notre Dame

Department of Electrical Engineering

7 May 2023

Table of Contents

1. Introduction……………………………………………………………………………….. 3

2. System Requirements……………………………………………………………………... 5

3. Project Description.....……………………………………………………………….…...... 6

3.1 System Theory of Operation……………………………………………………..... 6

3.2 System Block Diagram………………………………….................................. 7

3.3 Detailed Design/Operation of Subsystem 1……………………………….….... 8

3.4 Detailed Design/Operation of Subsystem 2………………………………...... 13

3.5 Effects Theory………………………………... 16

4. System Integration Testing……………………………………………………….. 20

4.1 Subsystem 1 - Audio Effects………………………………………………... 22

4.2 System 2 - Web Interface……………………………….......................... 23

4.3 Integrating Subsystems……………………………................................ 23

5. To-Market Design Changes……………………………………………………….. 24

5.1 Latency……………………………………….................................... 24

5.2 WiFi Network Selection…………………………….......................... 25

5.3 Housing Device………………………….. 25

5.4 Web Interface on Mobile Devices………………………………..................... 26

6. Users Manual/Installation Manual……………………………………………………….. 26

6.1 Setup/Installation…………………………………………………..................... 26

6.2 Troubleshooting……………………………….. 28

7. Conclusions…………………………………………………………………….………… 28

8. Appendices……………………………………………………………………………….. 29

8.1 Schematic Design..…................................. 29

8.2 Board Design..............……………………………………........................... 29

8.3 CAD Design....…………………………….. 30

8.4 Major Hardware Components.....………………………………..................... 30

8.5 Software...................................………………………………..................... 31

1 Introduction

Common guitar pedals (built with analog components) are generally very expensive - usually upwards of $60 - and only offer one audio effect per pedal. The cost of having a diverse variety of options to alter guitar tone quickly adds up with the more effects desired. Furthermore, these pedals are all activated with their own buttons by foot. In a situation where many effects are required, a large amount of pedals on a board can become clumsy and difficult to navigate around.

Our solution to these inconveniences is the creation of a “pedal” that effectively emulates common guitar effects digitally. With our design, many effects can be uploaded to just one “pedal" with a single interface. Not only does this save money, but also space. Easy storage and transportation make the use of our product a very user-friendly experience along with its control interface being Wi-Fi connection to a cell phone or any internet accessible device. Additionally, a suction cup clamp which holds the user’s phone to the front of the guitar allows the player to have control of the effects lineup at their fingertips as well as giving mobility to the effects board.

Achieving the desired functionality of our digital guitar effects system required the use of the ESP32 WROOM 32-E microcontroller along with the WM8960 audio codec to provide the necessary ADC/DAC conversion of the audio signal and the digital signal processing that was needed to apply the effects that are typically provided by physical guitar pedals. The I2S

communication protocol was used to transmit the audio data captured from the input between the ADC and the DAC. Utilizing the I2S communication protocol provided the ability to manipulate the audio data using various methods to achieve effects such as distortion, tremolo, fuzz, and an 8-bit effect. The board was developed for a 5V power supply using a USB-C connector and a 3.3V voltage regulator to supply voltage to the microcontroller and the audio codec. A website interface was developed with sliders that could change the respective intensity factors for each of the effects through Wi-Fi communication with the ESP-32 microcontroller.

The design met the expectations that were established for the project. The digital effects were successfully developed and controlled using the digital sliders on the website interface. If connected to the corresponding Wi-Fi network, users are able to connect to the website on their phone and control the digital effects with the sliders on their phone as they play the guitar. There was little to no latency when changing between different effects, and effects like tremolo and distortion could be utilized simultaneously as the user played.

Some aspects of the project that could be improved is the latency with the audio signal at the output. It was noticed that at times when the user played, there was delay in the time that the audio signal was processed and was heard at the output. This presented itself as one of the central issues of the project because the audio that was being played by the user and the audio at the output was asynchronous. This issue could be resolved by optimizing the size of the audio buffer to reduce the amount of processing time needed by the

microcontroller. The project would have also benefited from the use of a volume slider which could provide the user with more control over the amplitude of the audio signal at the output instead of having to adjust it directly on the amplifier.

2 System Requirements

The ESP32-WROOM-32E microcontroller was chosen due to it possessing the necessary processing capabilities for digital signal processing of the audio samples. It was also chosen because of the Wi-Fi communication capabilities it provides for the website interface.

The circuit board that was designed needed to receive 5V via USB-C and then through a voltage regulator provide 3.3V to the microcontroller and the audio codec. The circuit board also needed to include the necessary pull-up resistors for the I2C communication between the microcontroller and the audio codec. Online libraries developed for the ESP32 and the audio codec were utilized to enable the ADC/DAC for the left and right audio channels, and establish the I2S communication between the ADC and DAC where an audio buffer was utilized to transmit the signal.

The circuit board that was designed for this project required a 3-D printed housing which contained space for the board itself and for the two ¼" stereo audio jacks that were used for the audio input coming from the guitar and the audio output going to the amplifier. The circuit board housing must also have a

designated space for the USB-C connector that supplies 5V from the wall outlet to the board.

A website interface was required to control the digital effects as the user played the guitar. The website interface was developed using ESP32 libraries for the web server and HTML was utilized to create the web page of the interface. HTML was also used to create the sliders which were used to control the digital effects.

A mount was placed on the body of the guitar to provide functionality for remote controlling of the digital effects using a cell phone. The cell phone connects to the corresponding Wi-Fi network that is associated with the web server which can then be utilized to control the effects remotely.

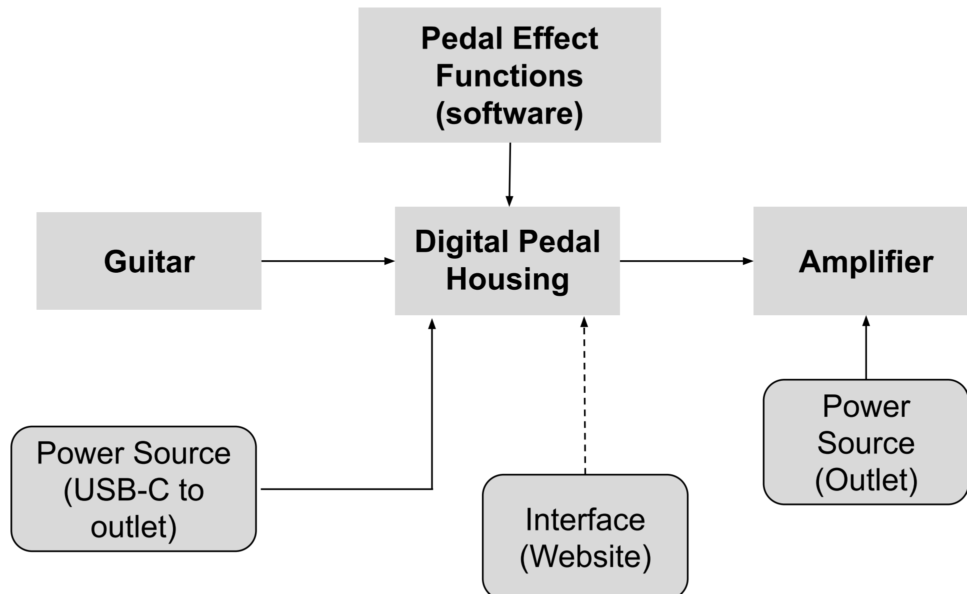
3 Detailed Project Description

3.1 System theory of operation

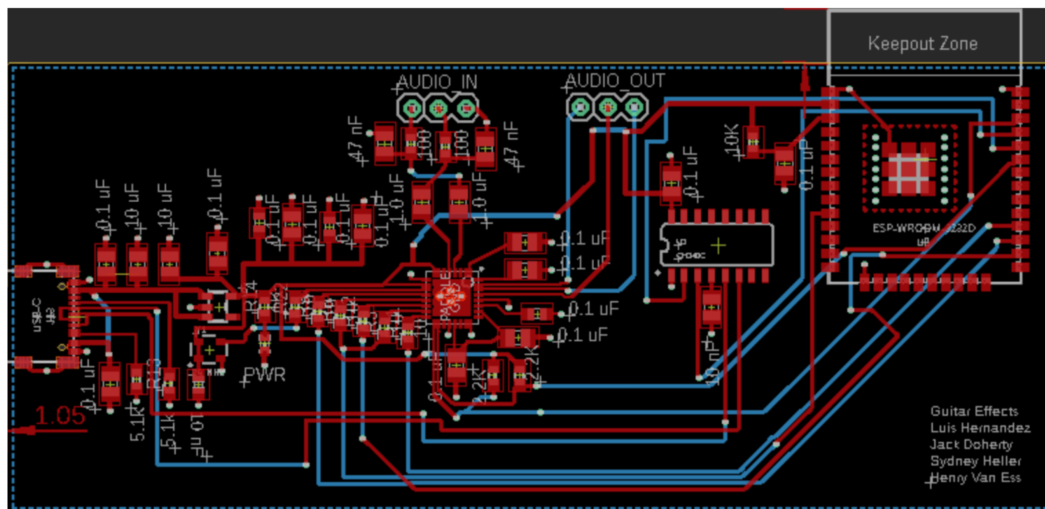
Our solution utilizes the digital signal processing capabilities of the ESP32 to emulate various guitar pedal effects, specifically distortion, fuzz, tremolo, and our experimental effect to recreate the sounds heard in 8-bit video games. We route the guitar signal first through a WM8960 codec chip, which includes an analog-to-digital converter as well as a digital-to-analog converter. On the input side, it makes use of the former. After quantization and digitization, the digital signal is passed through the ESP32 chip using I2S communication and the digital

audio signal is modified to achieve the desired effect. After the altered signal is processed, it is written back to the codec and converted back to an analog signal using the digital-to-analog converter. The resulting audio signal will be passed through a quarter inch TS guitar cable to a standard guitar amplifier that will output the new signal in real time (but with a small, limited amount of latency due to processing time). Our way of eliminating analog pedal circuits and using a single small processing board in a compact housing and developing an interface that can be used to adjust certain parameters such as gain and volume makes the device more convenient for the user.

3.2 System Block diagram



3.3 Detailed Design/Operation of Pedal Effect System



Our board choice is the SparkFun IoT Redboard that runs an Espressif ESP32-WROOM-32E. One reason for choosing this model is because of its wifi capabilities. Developing a website interface with sliders was a requirement for the project, and choosing a microcontroller capable of Wi-Fi communication was necessary. Another reason was because this board was designed for audio signal processing. The IoT Redboard is able to do a variety of tasks including CPU and on chip memory, bluetooth capabilities, I2C communication, Qwiic connection to our codec breakout board and MP3 decoding. This system provided more features than we knew we would need for our project. If we needed to adapt to a new approach, then this development board would have allowed us to do that.

The ESP32-WROOM-32E is a powerful chip that is perfect for our project. This offers dual-core processing that is able to run from 80 to 240 MHz, allowing

for real-time audio processing. The chip is also fairly inexpensive and widely available which was an important factor in developing our product because we want to make a cheap and accessible device. The microcontroller is energy-efficient as well, consuming low power to make it suitable for either a battery or wall adapter power system. This allowed us to have choices in what we wanted to do for our powering scheme. We also needed the device to be easily programmed with the Arduino IDE in Python or C which gave us more freedom in our choices for development.

We decided to go with powering our device through a wall adapter, requiring 6 volts maximum at the input. The board uses a USB-C connector for power. This can also be easily replaced with a battery pack that outputs to a USB-C if the user wants to take the device somewhere there are no places to plug the board into the wall and still supply the required voltage rating. The board uses a low dropout voltage regulator (LDO) to balance the input voltage to the required 3.3 volts for the ESP32-WROOM-32E chip. The specific regulator is the AP2112K-3.3TRG1. This device uses a voltage reference and error amplifier that compares the output voltage to the reference voltage, then it adjusts the resistance of a pass resistor to maintain a stable output voltage. It is a linear voltage regulator that dissipates the excess energy as heat, which is less efficient than switching regulators. It is still good because it is simple and less noisy which is important for an audio processing device.

The circuit includes a 40 MHz crystal oscillator that is important for the function of the chip. The crystal serves as an external clock source for the ESP32

microcontroller. It generates a stable frequency as a precise clock signal. This in turn is used to synchronize the timing of operations done inside the chip to operate reliably and provide stability. The ESP32 has an internal RC oscillator that can also be used as a clock source. However, this RC oscillator is not as accurate as the external crystal oscillator. Both the crystal and voltage regulator are connected to decoupling capacitors. Specifically, the regulator has these capacitors connected to its 3.3 V pin which is also supplied to the crystal oscillator. There is little room for error when handling audio signals due to the immediate requirements of the system; any mistiming can cause the audio signal to not sound pleasant.

The integrated circuit on the PCB is essential to the functioning of the serial monitor. The IC allows us to see this serial monitor and the Arduino IDE by converting the signal coming from the USB. This signal is converted into a serial signal. Because of this, we are able to see values being updated from the website interface and also see the IP of the website so that it can be accessed. Without the IC we would not be able to use the serial monitor to retrieve that information.

The system all starts with the guitar input signal at the top-left of the schematic. Our signal is passed into a quarter inch TRS audio jack that is soldered to the through holes under the “AUDIO_IN” text on the board. Attached to the through holes for the TRS jack are two sets of resistors and capacitors at each of the left and right audio input channels of the WM8960. We used 47 nanofarad capacitors and 100 ohm resistors. These values were specifically

chosen to function as a low pass filter, allowing only the frequencies below the chosen cut-off frequency to be passed into the circuit. The value for the cut-off frequency can be calculated using the formula:

$$f = 1 / (2\pi RC)$$

Our desired cut-off frequency was chosen to be 33.8 KHz. This filter allows us to attenuate any high frequency noise from the output of the TRS jack to the input of the analog to digital converter.

This analog signal then gets passed to the Sparkfun Audio Codec Breakout WM8960. The WM8960 is a low power stereo audio codec. This versatile device includes preamplifiers for our line inputs, speaker driver, equalization for frequency manipulation, and dynamic range control. We use this codec to convert our analog signal to a digital signal as well as using its programmable gain amplifier (PGA) to create the distortion effect. Series resistors are used to stabilize the digital signal and prevent any reflections from happening. These reflections could cause the digital signal to be misrepresented and create issues at the output.

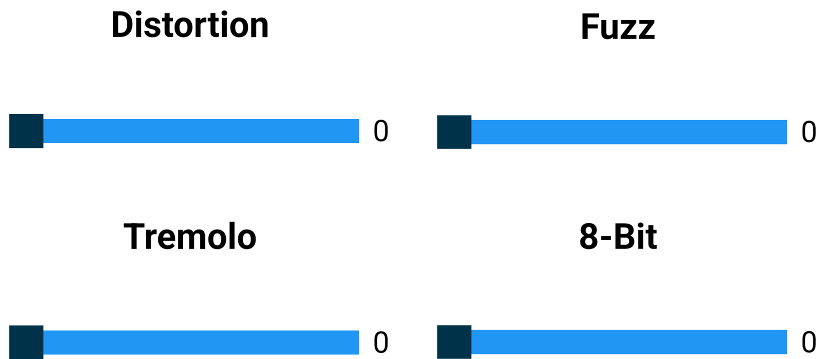
This board is ideal for handling a variety of audio signals. It supports I2S communication, which is what we chose for this specific project. I2S is beneficial over other formats like SPI and UART because it is designed specifically for transferring audio data. SPI and UART may not provide the same level of reliability as I2S. I2S is also supported by a lot of microcontrollers which is important in the rare case that vital hardware in our system becomes obsolete.

I2S uses separate clock and data lines that are advantageous for reducing timing errors and ensuring an accurate transfer of data. We needed our data to be transferred as clean as possible to make sure we get high quality sound with low latency.

This digital post-codec signal is passed into the ESP32. Inside the chip we use digital signal processing to achieve our effects. We have four different possible effects the user can experiment with. Depending on what is done in the user interface an effect will be produced by the chip. This new signal is relayed back to the WM8960. At this stage, the codec translates the altered audio signal from digital back to an analog signal. This analog signal is relayed to the output quarter inch TRS audio jack where it is picked up by a guitar cable and transmitted to an external amplifier.

3.4 Detailed Design/Operation of Website Interface

Interfaces



Our second subsystem involves a web interface that can be accessed by the user through a phone, tablet, computer or any other device capable of connecting to the internet. Upon uploading the program to the ESP32, the serial monitor will print an IP address that can be pasted into a web browser. The above interface will appear with four sliders. Each slider corresponds to the intensity of one effect and can be adjusted to the desire of the user. Values range from 0-10 for distortion, fuzz, and tremolo. The 8-bit guitar effect only has values 0 and 1 signifying on or off. 8-Bit will not work with the other effects, so they will be disabled when the user turns the 8-bit effect on. This is reflected in real time with the slider values displayed on the web interface. Fuzz and distortion work in the same way that when one is turned on, the other will automatically turn off. Tremolo is able to function with the fuzz or distortion effects simultaneously.

This web interface is controlled by HTML for content and structure, CSS for styling, and JavaScript for dynamic features like the interaction and animation of the sliders. Upon uploading the code, the sliders are initialized and set to zero so that all the effects are disabled by default. There are update functions that use XMLHttpRequest to send GET requests to the server to retrieve data from the webpage, reading the current value of the sliders as controlled by the outward-facing web interface. This value is read by the code as a string and converted to an integer so that it can be used in our functions as a determinant of the intensity of the effect being outputted. We want to give the user as much control as possible, since there are many different styles of playing guitar. This is the main reason we chose not to use buttons on the web interface. The buttons would give little to no control of the levels at the output. The website sliders provide greater control of the gain for distortion, oscillations for tremolo, or wet mix for fuzz.

Our interface was chosen to be a web page for a variety of reasons. One major deciding factor was being untraditional. In an increasingly growing world of technology it is important to stay up to date and try new methods of achieving the same goal. It would have been easy to be like the rest and have a physical pedal which is beneficial for obvious reasons, but having the control at your fingertips also has its advantages. For one, if the user is impaired or handicapped then they may not be able to actually use a step pedal to turn their effects on. They could possibly press it down with their hands but this can be difficult when in the midst of playing a song. Another advantage is if the guitar player wants their

sound engineer to control the output of effects. This way they would not have to worry at all about which pedal they have to step on next. The interface would be great in this manner, making sure that the guitar player can focus on playing as opposed to messing around with a heap of pedals. The web page also allows us to condense our effects on one screen. The burden of having ten pedals to carry around, each with its own sound effects can be a hassle. Even if the effects were condensed into one housing device with knobs and buttons, the user would have to strategically place the housing device within close reach while dealing with cables that could become twisted. With our design, all that is required is the housing containing our board, which does not need to stay right next to the user, and the user’s phone, which is likely already on them at all times.

3.5 Effects Theory

3.5.1: Distortion

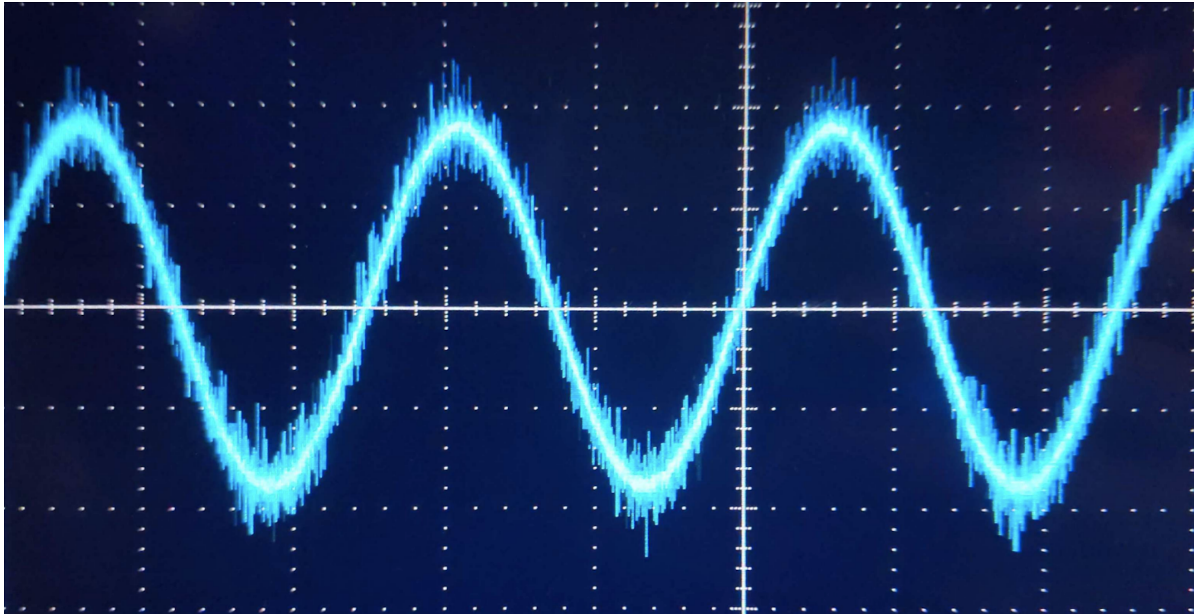


Figure 3.5.1(a): Clean Input (Sine Wave)

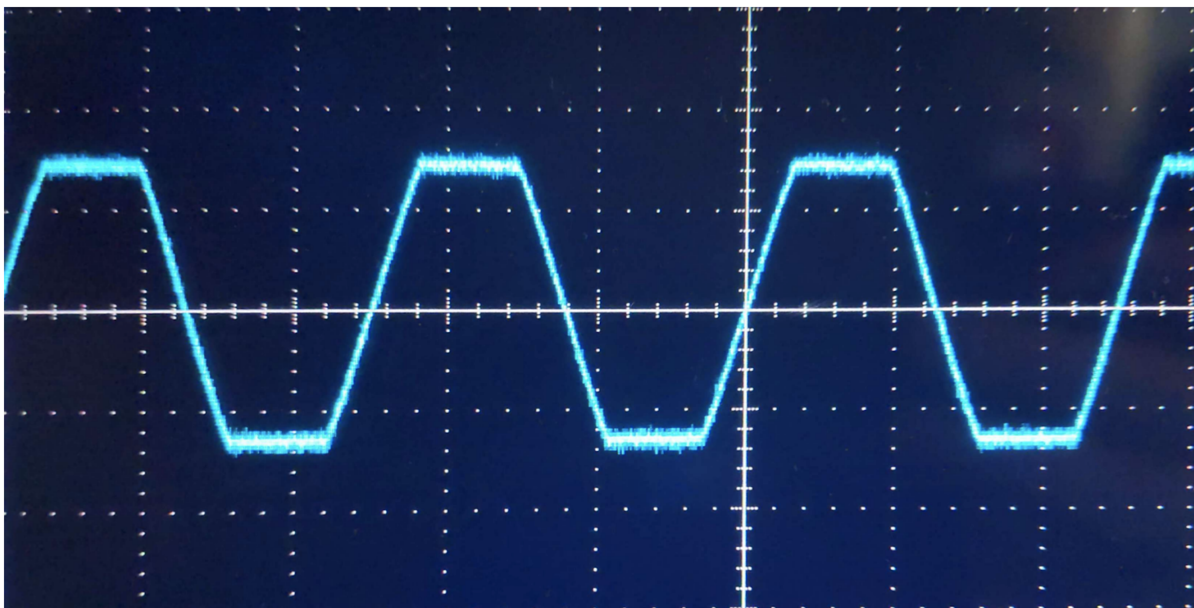


Figure 3.5.1(b): Distortion Output (Level 7)

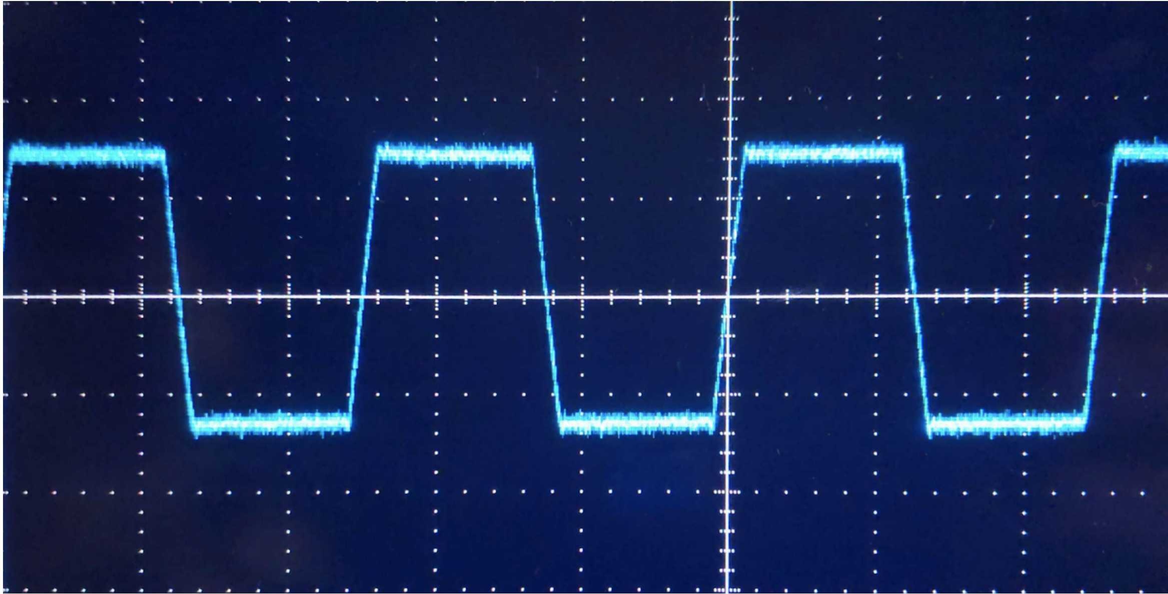


Figure 3.5.1(c): Distortion Output (Level 10)

Pictured above are oscilloscope traces of a demonstration test of our system. Figure 3.5.1(a) shows the trace of a 400 Hz sine wave applied to the input in all tests (distortion as well as following effect tests) and Figure 3.5.1(a) and Figure 3.5.1(b) show the output of the pedal with distortion turned to levels 7 and 10, respectively. Our distortion functions essentially as traditional overdrive distortion, which essentially creates a grittier sound, but still preserves a tone similar to the original. This is accomplished by amplifying the strength of the signal by the user-controlled amount of gain and clipping the output amplitude at a value slightly higher than the max amplitude of the input signal. Clipping slightly higher than the input amplitude allows a greater variety of distortion as for the lower input values, the signal begins to clip but doesn’t hard clip as shown in the scope traces. Having the clipping level like this does however come at the cost of an increase in volume when distortion is in use. While this does not affect tone, it

does require user attention to volume knobs if they wish to hold at a steady sound level.

3.5.2: Fuzz

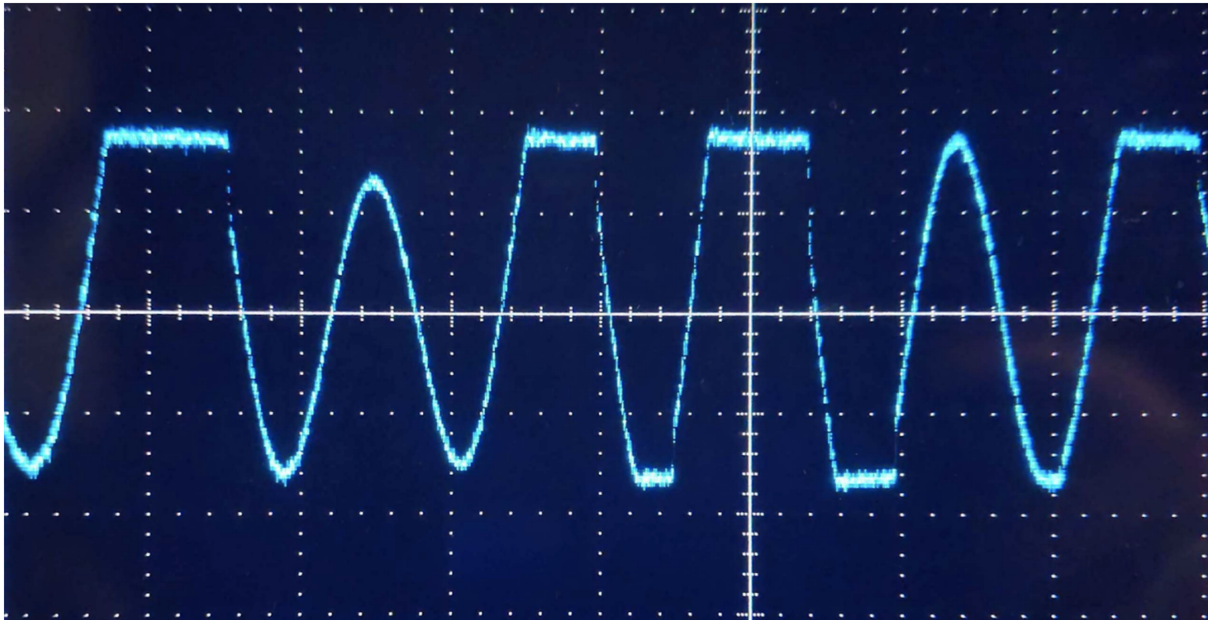


Figure 3.5.2: Fuzz Output

For our fuzz effect, we went about level adjustment slightly differently than the conventional method of varying the amount of distortion. The output audio signal is a mix between the ‘wet’ and ‘dry’ signals. Increasing the slider value to ten would essentially be listening to a 100% distorted signal, matching the unevenly-clipped trace shown above. This mix made obtaining a snapshot of the modified waveform difficult to capture as the fuzz slider had to be maxed out to find a single wave. In this case, the clean signal is still getting passed through to the output, but the ‘wet’ signal basically overwhelms the clean. The lower the value for the fuzz slider, the more of the clean signal will pass through to the

audio output. The benefit of this effect compared to distortion is that the distorted signal is not achieved through applying gain using PGA from the codec breakout WM8960. The audio is more of a harmonic shift from a clean tone and is useful for applying a unique buzzing sound, without losing original intonation, compared to a bold kind of distortion.

This effect was achieved using a series of floating point numbers passed as an array to the fuzz function. The function takes three arguments: a pointer to the array containing the audio samples, an integer that indicates the number of bytes in the array containing the samples, and another integer that represents the gain factor. First, the gain factor is established by setting it equal to the value passed in as an argument. Then the number of samples in the original array is calculated by dividing the number of samples by the size of each sample. The function then loops through each sample using a for loop. Within the loop, the original sample is multiplied by the gain factor to introduce distortion. The distorted signal is then added to the original signal, and then the original signal again. This is to ensure that the clean sound can still be heard. Distorted signals can easily override the original, so we wanted to really emphasize the original as the priority. This also contributes to boosting harmonic content through constructive interference. The loop will continue until all of the samples in the original array have been processed. When completed, the data can be used for output or further processing with tremolo.

3.5.3: Tremolo

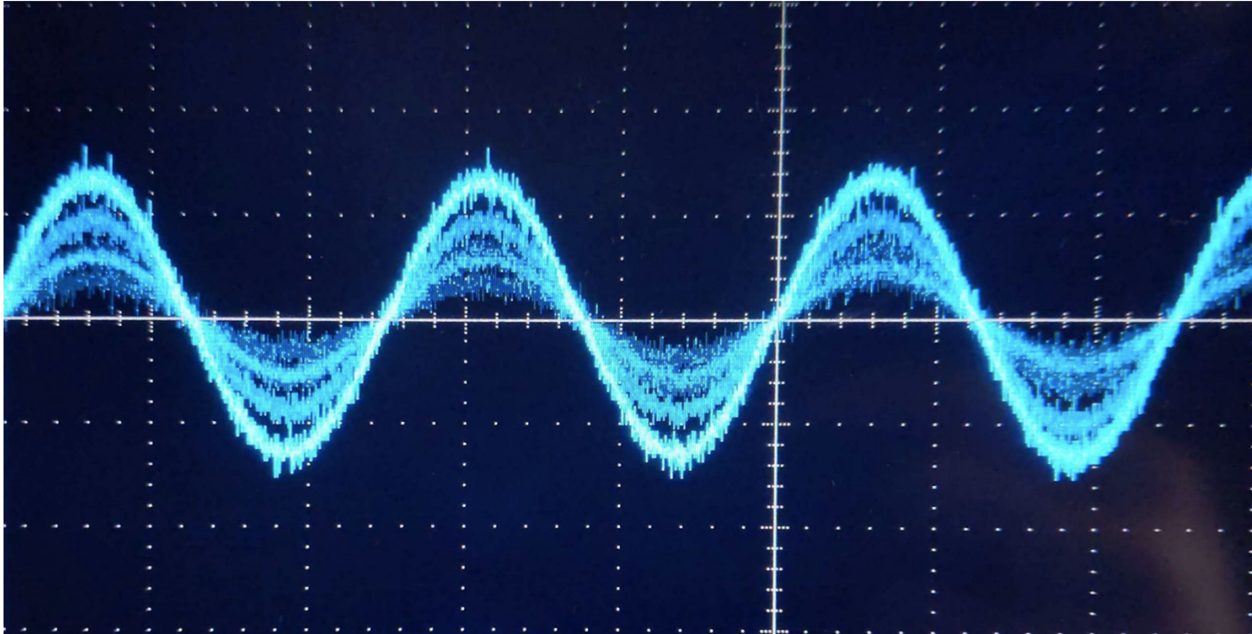


Figure 3.5.3: Tremolo Output

The oscilloscope image above shows, in a still shot, oscillation in the amplitude of the output signal. Unlike the two types of distortion our system creates, the tremolo effect does not change the tone of the signal; rather, it adds a variable low-frequency oscillation of the presence of sound output from the system. So for this effect, other than the strength of the signal, this effect does not impact the shape of the wave.

This was accomplished in code by utilizing a built-in codec function that controls the chip volume and incrementing up and down with iterations of the main loop. This iteration bounced back and forth continuously and gradually between the original signal strength and roughly -18 dB quieter, giving a pulsing feeling to the audio. The slider on the interface that controls this effect varies from 0 to 10 in 0.5 unit increments which represent speeds at which the sound

volume is oscillated. The lowest “on” setting of 0.5 creates an oscillation period of about 1.67 seconds and an input value of 10 creates one of about 0.084 seconds.

3.5.4: 8-Bit

The 8-Bit audio effect was an experimental sound that was unique and offered an alternative type of guitar playing for the user. We were inspired to create such an effect from old school video games. The grainy visuals of games like Tetris, Super Mario Brothers, and Space Invaders were only made more memorable with their upbeat and memorable music. The effect was achieved by adding some clipping through gain, similar to the fuzz effect. We then wrote the audio signal using a smaller buffer length. This shortening of the buffer length allowed us to effectively reduce the bit resolution of the audio signal and output sounds to emulate the theme songs from these iconic games.

These four effects serve as just a starting point to what can be accomplished. With improvements to communication between devices and optimization of our memory usage, we can add many more effects. The main goal of this project was to get at least three effects and we were able to achieve four. The limitations of our system would require us to use more memory than we were able to use. Passing the signal between an SD card and the microcontroller caused too much delay, to where the signal became hard to play along with and keep a constant train of thought. We were able to improve our performance by limiting the delay at the output through keeping the buffer lengths as small as

possible to limit the number of calculations being performed for each iteration of a loop. This project was all about finding this balance to give the user the most enjoyable experience possible.

4 System Integration Testing

To ensure that all subsystems were integrated successfully, we tested the elements of each subsystem separately throughout the course of the project before eventually combining elements into their functioning subsystems and then one complete system.

4.1 Subsystem 1 - Audio Effects

To begin, we attempted audio passthrough using I2S communication between the IoT Redboard and the audio codec. We needed to confirm that the audio signal could pass cleanly through the AD/DA conversions in our setup, and that a clear signal could be heard through the amplifier from a music source.

After audio passthrough was a success, we tackled each effect individually. We tested each effect in the main loop first to check if it was producing the desired sound. Use of an oscilloscope as described in the previous section was also essential in this step to troubleshoot and tweak our programming to match the effects we were looking for. We then tested the ability to turn the effects on and off programmatically by calling a function that handled

the effect outside of the main loop. Next, we tested the ability to vary effect intensity by hardcoding a change in parameters.

After the successful execution of each audio effect, we collected the code for all the working effects and checked that we could call multiple effects within one script. This tested whether or not we could observe the same effect at the output that we heard when testing each effect individually.

4.2 Subsystem 2 - Web Interface

For the web interface, we first tested the visual display to ensure that it looked and functioned properly. We wanted to make sure that we could successfully communicate with the ESP32 to display the name of the effect, an interactable slider, and the slider's value for each of our guitar effects.

4.3 Integrating Subsystems

Once both subsystems were individually tested and functioning properly, we integrated them to ensure that they worked together as expected. After the web interface looked and behaved as expected, we tested the ability to programmatically read the current slider values from the web interface so that we could access the effect intensities that the user chose. We then incorporated the read slider values into the functions controlling each audio effect, adjusting the appropriate parameters to reflect the effect intensity selected. This tested our ability to control the audio effects through the web interface.

Finally, we verified that we could also control the slider values from the script, and observe that the web interface display updated properly to reflect these changes in value. This tested our ability to control the web interface from the script, which was essential for cases where our effects could not be applied simultaneously, and one effect had to be disabled if another effect was enabled by the user.

5 To-Market Design Changes

5.1 Latency

Reducing the latency between the input and output signals is critical before taking our product to market. This latency, when significant, can be noticeable to the user. Consequently, it can be distracting to the guitarist and inhibit their enjoyment of the experience. Before bringing our digital guitar effects pedal to market, we would need to reduce this latency until it is imperceptible to the user. To achieve this, we can search for ways to optimize the code by exploring the following: a decrease in the sampling rate, a reduction of the number of calculations performed when applying an effect, and a shorter buffer length that does not compromise the quality of the output signal. Another option is to use a faster processor that can handle more calculations in less time. This may include leveraging the capabilities of a dual-core processor, or even using a dedicated DSP chip.

5.2 WiFi Network Selection

Another challenge of our present design is that the WiFi network is hard-coded into the backend of the product. In order to sell our product commercially, the user must be able to choose the WiFi network from the frontend so that they are able to access the webpage with the sliders controlling the intensity of the effects. One solution could be to add a small OLED or LCD display that can display a list of available wifi networks. The user could then use a button or series of buttons to navigate the list and select the desired network. The electrical components required for this approach are inexpensive, so we could continue to keep the cost of our project low. Another approach is to allow the device to create its own WiFi hotspot, which the user can connect to from their smartphone or computer. Once connected, the user could access a configuration page through a web browser to enter the WiFi network credentials.

5.3 Housing Device

In addition to these challenges, the housing device would need to be adapted so that it possesses off-the-shelf production quality. The enclosure should be sturdy and simple to use, with clear labeling of inputs and outputs, as well as intuitive controls. The housing design should be compact and durable so that the user can easily transport our product without hassle or damage.

5.4 Web Interface on Mobile Devices

Finally, before selling our product commercially, we would need to update the web interface so that the layout is designed for a mobile device. That is, the size of the sliders and their corresponding labels should be reduced so that all four sliders of our current design could fit onto one page of a mobile screen, thus eliminating the need for the user to scroll to change a specific effect at the bottom. For an even more streamlined experience, this web interface could be adapted into a mobile app, allowing us to streamline the process of launching the effects slider interface, and to continuously update our product with additional effects for the user over time.

6 Users Manual/Installation manual

6.1 Setup/Installation

In order to use our product, the user needs their own guitar, two standard guitar cables (1/4"), an amplifier, a guitar phone mount, and a USB-C connector (all sold separately). To begin, they can use the USB-C connector to connect to power. An LED will indicate if the device is successfully powered. Then, the user can connect one of the standard guitar cables from their guitar to the “audio in” port of our device. The second standard guitar cable then connects from the “audio out” port of our device to the input port of the user’s amplifier. These

connections are necessary to apply the desired effects to the guitar's audio output and play the edited signal through the amp.

To continue setting up their guitar effects "pedal," the user must connect the device to a WiFi network. If the final product design ultimately provides its own hotspot, the user can use their phone to search for the corresponding "Guitar Effects Pedal" network and connect to it. Alternatively, if the market product includes a display on the physical device itself, the user can navigate to their desired network using the buttons or touchscreen provided. Another LED can indicate if the device is successfully connected to WiFi. The most important step is to ensure that our guitar effects "pedal" and the user's phone are connected to the same WiFi network in order to communicate with each other.[1]

Once connected, the user can open their browser of choice and navigate to the IP address of the device. This can be found on the device itself in the case of providing its own hotspot, or by logging into the WiFi's router if using another network. If connected successfully, the sliders for each guitar effect will display on the phone screen of the user. The user can then use the web browser to control the application and the intensity of each effect. The only thing left to do is to place their phone into a guitar phone mount, and to jam out to their heart's desire.

[1] If using our device in its present state on Notre Dame's campus, the user must connect their phone to the SDNet network and enter the password "CapstoneProject".

6.2 Troubleshooting

In the event that the user is not hearing the anticipated effect applied at the output, the first step is to check all cable connections. If the LED indicating power is not illuminated, the USB-C connector may not be connected properly, or the USC-C cord or power source might be faulty. If the LED indicating power is illuminated, check that the guitar cables are connected securely to the guitar and to the amp, and check that the volume of the amplifier is up. Specifically for the distortion effect, if minimal distortion is noticed even when the slider is set to 10, make sure that the output volume knob on the guitar itself is turned to max - this may be necessary to ensure that signal is strong enough to experience clipping when processed.

If the user must continue troubleshooting, ensure that our product and the user's phone are connected to the same WiFi network, and check the strength of the WiFi signal. If the WiFi connection is weak, the devices may not communicate successfully.

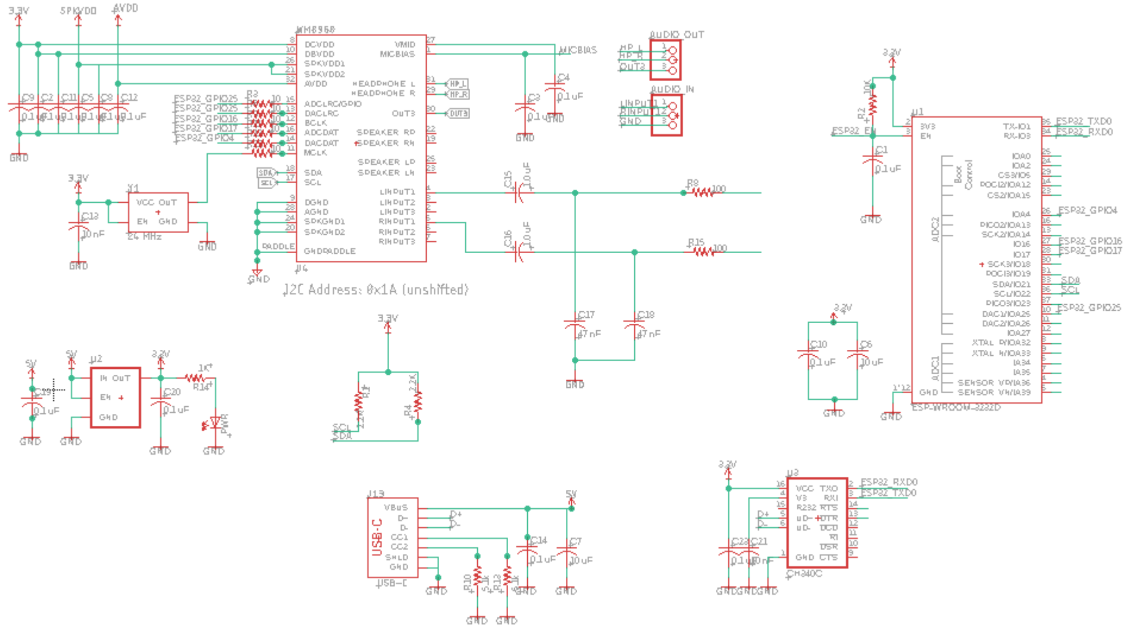
If none of these steps resolve the problem, try turning the device off and back on, and refresh the webpage.

7 Conclusions

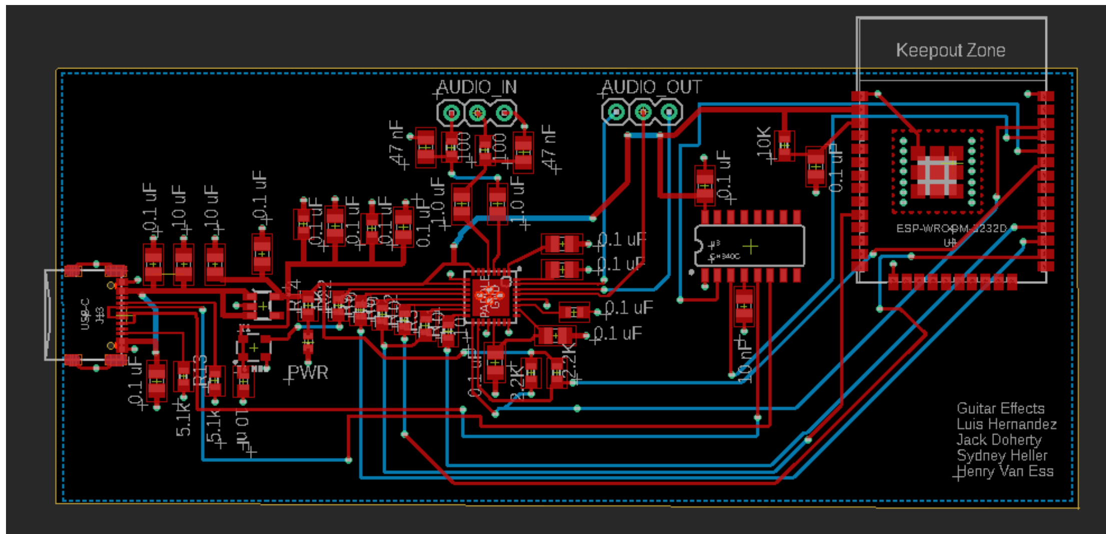
The digital guitar effects system conceived in this project is a unique, cost-effective solution that integrates multiple effects into a single system. Single-effect pedals can quickly create a mess of a floorspace when used in performance or practice and require a hardware purchase when a new effect is wanted. Our effort to solve these problems with a single digital pedal can offer musicians of all skill levels and budgets an alternative option to ease their physical and/or financial hassles. While our project is only a sample of the capabilities of this technological application, it offers an insight into the possibilities of a cheaper, more versatile alternative to traditional effects pedals. Currently our system offers 4 effects, all performing signal alteration in the time domain. Future updates to this device would likely include a more expansive lineup of effects available for download, including those performing signal processing in the frequency domain to greatly expand capabilities. Additionally, a more powerful chip would likely be required to execute more complex computations at a fast enough rate, as even with the effects we created this was an issue. The chip and software, in addition to upgrades to the housing as well as interface connection would produce a product ready to market.

8 Appendices

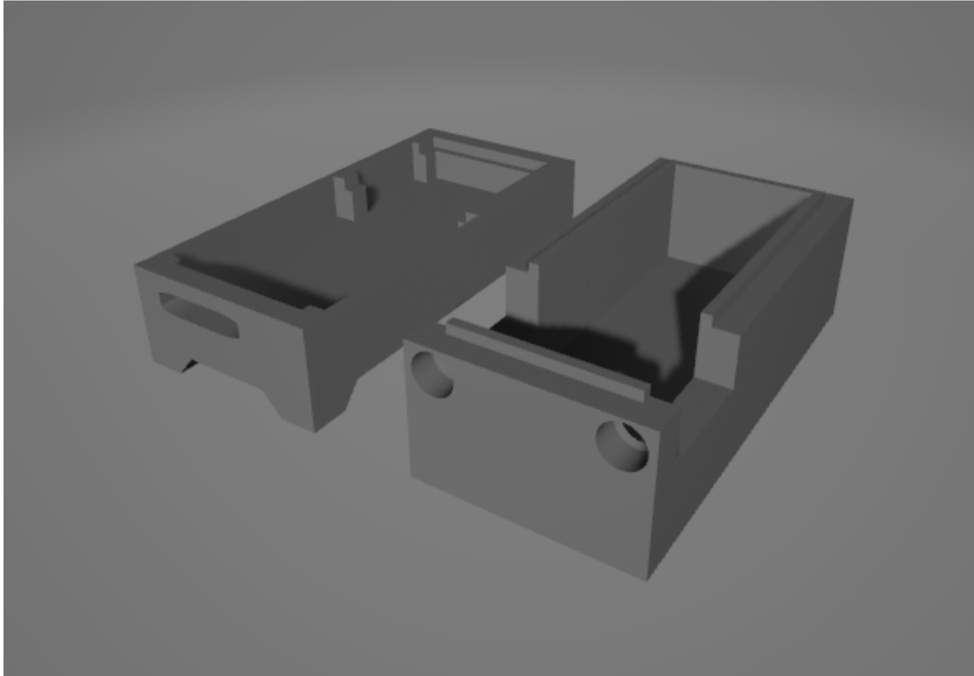
8.1 Schematic Design



8.2 Board Design



8.3 CAD Design



Board Housing

8.4 Major Hardware Components

PART NAME	**DATASHEET LINK**
SparkFun IoT RedBoard - ESP32 Development Board	https://cdn.sparkfun.com/assets/4/3/1/7/6/esp32-wroom-32e_esp32-wroom-32ue_datasheet_en_v1-6.pdf (ESP32-D0WD-V3 chip)
Audio Codec Breakout - WM8960	https://cdn.sparkfun.com/assets/a/3/a/7/4/WM8960_datasheet_v4.2.pdf

8.5 Software

```
#include <Wire.h>

#include <SparkFun_WM8960_Arduino_Library.h>

#include <algorithm>

#include <WiFi.h>

#include <AsyncTCP.h>

#include <ESPAsyncWebServer.h>

#include <iostream>

#include <string>


WM8960 codec;



// Include I2S driver

#include <driver/i2s.h>


// Connections to I2S

#define I2S_WS 25

#define I2S_SD 17

#define I2S_SDO 4

#define I2S_SCK 16


// Use I2S Processor 0

#define I2S_PORT I2S_NUM_0


// Define input buffer length
```

```
#define bufferLen 512
int16_t sBuffer[bufferLen];


// Define Sample Rate
#define SAMPLERATE 44100


// WiFi login credentials
const char* ssid = "SDNet";
const char* password = "CapstoneProject";


// Variables to read the slider values from the web interface
String distortionValue = "0";
String tremoloValue = "0";
String fuzzValue = "0";
String eightBitValue = "0";


const char* PARAM_INPUT = "value";


// Create AsyncWebServer object on port 80
AsyncWebServer server(80);


// Create the visual display of the webpage
const char index_html[] PROGMEM = R"rawliteral(


<!DOCTYPE html>
```

```
<head>

  <meta name="viewport" content="width=device-width, initial-scale=1.0">

  <title>Guitar Effects</title>

  <style>

    html {font-family: Arial; display: inline-block; text-align: center;}

    h2 {font-size: 2.3rem;}

    p {font-size: 1.9rem;}

    body {max-width: 400px; margin:0px auto; padding-bottom: 25px;}

    .slider { -webkit-appearance: none; margin: 14px; width: 360px;
height: 25px; background: #2196F3;

      outline: none; -webkit-transition: .2s; transition: opacity .2s;}

    .slider::-webkit-slider-thumb {-webkit-appearance: none; appearance:
none; width: 35px; height: 35px; background: #003249; cursor: pointer;}

    .slider::-moz-range-thumb { width: 35px; height: 35px; background:
#003249; cursor: pointer; }

    .slider-container { display: flex; align-items: center; };

    .slider-value { display: inline; margin-left: 10px; };

  </style>

</head>


<body>


  <h2>Distortion</h2>

  <div class="slider-container">

  <label for="distortion-slider"></label>
```

```
  <p><input type="range" onchange="updateDistortionSlider(this)"
id="distortion-slider" min="0" max="10" value="%DISTORTIONVALUE%" step="1"
class="slider"></p>
  <p><span id="textDistortionValue">%DISTORTIONVALUE%</span></p>
  </div>

  <h2>Tremolo</h2>
  <div class="slider-container">
  <label for="tremolo-slider"></label>
  <p><input type="range" onchange="updateTremoloSlider(this)"
id="tremolo-slider" min="0" max="10" value="%TREMOLOVALUE%" step="0.5"
class="slider"></p>
  <p><span id="textTremoloValue">%TREMOLOVALUE%</span></p>
  </div>

  <h2>Fuzz</h2>
  <div class="slider-container">
  <label for="fuzz-slider"></label>
  <p><input type="range" onchange="updateFuzzSlider(this)"
id="fuzz-slider" min="0" max="10" value="%FUZZVALUE%" step="1"
class="slider"></p>
  <p><span id="textFuzzValue">%FUZZVALUE%</span></p>
  </div>

  <h2>8-Bit</h2>
  <div class="slider-container">
```

```
  <label for="eightBit-slider"></label>

  <p><input type="range" onchange="updateEightBitSlider(this)"
id="eightBit-slider" min="0" max="1" value="%EIGHTBITVALUE%" step="1"
class="slider"></p>

  <p><span id="textEightBitValue">%EIGHTBITVALUE%</span></p>

  </div>



  <script>


function updateDistortionSlider(element) {

      var distortionValue =
document.getElementById("distortion-slider").value;

      var fuzzValue = document.getElementById("fuzz-slider").value;

      var eightBitValue =
document.getElementById("eightBit-slider").value;

      if (fuzzValue != 0) {

        document.getElementById("fuzz-slider").value = 0;

        document.getElementById("textFuzzValue").innerHTML = 0;

        fuzzValue = 0;

        console.log(fuzzValue);

        var xhr = new XMLHttpRequest();

        xhr.open("GET", "/fuzz-slider?value=" + fuzzValue, true);

        xhr.send();

      }

      if (eightBitValue != 0) {
```

```
        document.getElementById("eightBit-slider").value = 0;
        document.getElementById("textEightBitValue").innerHTML = 0;
        eightBitValue = 0;
        console.log(eightBitValue);
        var xhr = new XMLHttpRequest();
        xhr.open("GET", "/eightBit-slider?value=" + eightBitValue, true);
        xhr.send();
      }
      document.getElementById("textDistortionValue").innerHTML =
distortionValue;
      console.log(distortionValue);
      var xhr = new XMLHttpRequest();
      xhr.open("GET", "/distortion-slider?value=" + distortionValue,
true);
      xhr.send();
    }

    function updateTremoloSlider(element) {
      var tremoloValue = document.getElementById("tremolo-slider").value;
      var eightBitValue =
document.getElementById("eightBit-slider").value;
      if (eightBitValue != 0) {
        document.getElementById("eightBit-slider").value = 0;
        document.getElementById("textEightBitValue").innerHTML = 0;
        eightBitValue = 0;
        console.log(eightBitValue);
```

```
        var xhr = new XMLHttpRequest();
        xhr.open("GET", "/eightBit-slider?value=" + eightBitValue, true);
        xhr.send();
      }
      document.getElementById("textTremoloValue").innerHTML =
tremoloValue;
      console.log(tremoloValue);
      var xhr = new XMLHttpRequest();
      xhr.open("GET", "/tremolo-slider?value=" + tremoloValue, true);
      xhr.send();
    }

    function updateFuzzSlider(element) {
      var fuzzValue = document.getElementById("fuzz-slider").value;
      var distortionValue =
document.getElementById("distortion-slider").value;
      var eightBitValue =
document.getElementById("eightBit-slider").value;
      if (distortionValue != 0) {
        document.getElementById("distortion-slider").value = 0;
        document.getElementById("textDistortionValue").innerHTML = 0;
        distortionValue = 0;
        console.log(distortionValue);
        var xhr = new XMLHttpRequest();
        xhr.open("GET", "/distortion-slider?value=" + distortionValue,
true);
```

```
        xhr.send();
      }
      if (eightBitValue != 0) {
        document.getElementById("eightBit-slider").value = 0;
        document.getElementById("textEightBitValue").innerHTML = 0;
        eightBitValue = 0;
        console.log(eightBitValue);
        var xhr = new XMLHttpRequest();
        xhr.open("GET", "/eightBit-slider?value=" + eightBitValue, true);
        xhr.send();
      }
      document.getElementById("textFuzzValue").innerHTML = fuzzValue;
      console.log(fuzzValue);
      var xhr = new XMLHttpRequest();
      xhr.open("GET", "/fuzz-slider?value=" + fuzzValue, true);
      xhr.send();
    }

    function updateEightBitSlider(element) {
      var eightBitValue =
document.getElementById("eightBit-slider").value;
      var distortionValue =
document.getElementById("distortion-slider").value;
      var tremoloValue = document.getElementById("tremolo-slider").value;
      var fuzzValue = document.getElementById("fuzz-slider").value;
      if (distortionValue != 0) {
```

```
        document.getElementById("distortion-slider").value = 0;

        document.getElementById("textDistortionValue").innerHTML = 0;

        distortionValue = 0;

        console.log(distortionValue);

        var xhr = new XMLHttpRequest();

        xhr.open("GET", "/distortion-slider?value=" + distortionValue,
true);

        xhr.send();

      }

      if (tremoloValue != 0) {

        document.getElementById("tremolo-slider").value = 0;

        document.getElementById("textTremoloValue").innerHTML = 0;

        tremoloValue = 0;

        console.log(tremoloValue);

        var xhr = new XMLHttpRequest();

        xhr.open("GET", "/tremolo-slider?value=" + tremoloValue, true);

        xhr.send();

      }

      if (fuzzValue != 0) {

        document.getElementById("fuzz-slider").value = 0;

        document.getElementById("textFuzzValue").innerHTML = 0;

        fuzzValue = 0;


        console.log(fuzzValue);

        var xhr = new XMLHttpRequest();

        xhr.open("GET", "/fuzz-slider?value=" + fuzzValue, true);
```

```
        xhr.send();
      }
      document.getElementById("textEightBitValue").innerHTML =
eightBitValue;
      console.log(eightBitValue);
      var xhr = new XMLHttpRequest();
      xhr.open("GET", "/eightBit-slider?value=" + eightBitValue, true);
      xhr.send();
    }

  </script>

</body>

</html>

)rawliteral";


// Replaces placeholder with button section in your web page
String processor(const String& var){

  if (var == "DISTORTIONVALUE"){
    return distortionValue;
  }

```

```
  if (var == "TREMOLOVALUE"){

    return tremoloValue;

  }


  if (var == "FUZZVALUE"){

    return fuzzValue;

  }


  if (var == "EIGHTBITVALUE"){

    return eightBitValue;

 }


  return String();

} // ends String processor()


// All the code for the web interface setup and functionality, called in
setup()

void web_interface_setup()

{

  // Connect to Wi-Fi

  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {

    delay(1000);

    Serial.println("Connecting to WiFi..");

  }
```

```
  // Print ESP Local IP Address
  Serial.println(WiFi.localIP());

  // Route for root / web page
  server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send_P(200, "text/html", index_html, processor);
  });

  server.on("/distortion-slider", HTTP_GET, [] (AsyncWebServerRequest
*request) {
    String inputMessage;
    // GET input1 value on <ESP_IP>/slider?value=<inputMessage>
    if (request->hasParam(PARAM_INPUT)) {
      inputMessage = request->getParam(PARAM_INPUT)->value();
      distortionValue = inputMessage;
    }
    else {
      inputMessage = "No message sent";
    }
    Serial.println("Distortion Value: " + inputMessage);
    request->send(200, "text/plain", "OK");
  });

  // Send a GET request to <ESP_IP>/slider?value=<inputMessage>
  server.on("/tremolo-slider", HTTP_GET, [] (AsyncWebServerRequest
*request) {
```

```
    String inputMessage;
    // GET input1 value on <ESP_IP>/slider?value=<inputMessage>
    if (request->hasParam(PARAM_INPUT)) {
      inputMessage = request->getParam(PARAM_INPUT)->value();
      tremoloValue = inputMessage;
    }
    else {
      inputMessage = "No message sent";
    }
    Serial.println("Tremolo Value: " + inputMessage);
    request->send(200, "text/plain", "OK");
  });

  server.on("/fuzz-slider", HTTP_GET, [] (AsyncWebServerRequest *request)
{
    String inputMessage;
    // GET input1 value on <ESP_IP>/slider?value=<inputMessage>
    if (request->hasParam(PARAM_INPUT)) {
      inputMessage = request->getParam(PARAM_INPUT)->value();
      fuzzValue = inputMessage;
    }
    else {
      inputMessage = "No message sent";
    }
    Serial.println("Fuzz Value: " + inputMessage);
    request->send(200, "text/plain", "OK");
```

```
  });

  server.on("/eightBit-slider", HTTP_GET, [] (AsyncWebServerRequest
*request) {
    String inputMessage;
    // GET input1 value on <ESP_IP>/slider?value=<inputMessage>
    if (request->hasParam(PARAM_INPUT)) {
      inputMessage = request->getParam(PARAM_INPUT)->value();
      eightBitValue = inputMessage;
    }
    else {
      inputMessage = "No message sent";
    }
    Serial.println("8-Bit Value: " + inputMessage);
    request->send(200, "text/plain", "OK");
  });

  // Start server
  server.begin();

} // ends web_interface_setup()

int disValInt = distortionValue.toInt();

// Code for setting up the codec, called in setup()
void codec_setup()
```

```
{
  // General setup needed
  //int* reg_address = WM8960_REG_AUDIO_INTERFACE_2
  //*reg_address = 0b0010000;
  codec.enableVREF();
  codec.enableVMID();

  // Setup signal flow to the ADC

  codec.enableLMIC();
  codec.enableRMIC();

  // Connect from INPUT1 to "n" (aka inverting) inputs of PGAs.
  codec.connectLMN1();
  codec.connectRMN1();

  // Disable mutes on PGA inputs (aka INTPUT1)
  codec.disableLINMUTE();
  codec.disableRINMUTE();

  // Set pga volumes
  codec.setLINVOLDB(3*disValInt);
  codec.setRINVOLDB(3*disValInt);

  // Set input boosts to get inputs 1 to the boost mixers
  codec.setLMICBOOST(WM8960_MIC_BOOST_GAIN_0DB);
```

```
codec.setRMICBOOST(WM8960_MIC_BOOST_GAIN_0DB);

// Connect from MIC inputs (aka pga output) to boost mixers
codec.connectLMIC2B();
codec.connectRMIC2B();

// Enable boost mixers
codec.enableAINL();
codec.enableAINR();

// Disconnect LB2LO (booster to output mixer (analog bypass)
// For this example, we are going to pass audio throught the ADC and DAC
codec.disableLB2LO();
codec.disableRB2RO();

// Connect from DAC outputs to output mixer
codec.enableLD2LO();
codec.enableRD2RO();

// Set gainstage between booster mixer and output mixer
// For this loopback example, we are going to keep these as low as they
go
codec.setLB2LOVOL(WM8960_OUTPUT_MIXER_GAIN_NEG_21DB);
codec.setRB2ROVOL(WM8960_OUTPUT_MIXER_GAIN_NEG_21DB);

// Enable output mixers
```

```
  codec.enableLOMIX();

  codec.enableROMIX();


  // CLOCK STUFF, These settings will get you 44.1KHz sample rate, and
class-d
  // freq at 705.6kHz
  codec.enablePLL(); // Needed for class-d amp clock
  codec.setPLLPRESCALE(WM8960_PLLPRESCALE_DIV_2);
  codec.setSMD(WM8960_PLL_MODE_FRACTIONAL);
  codec.setCLKSEL(WM8960_CLKSEL_PLL);
  codec.setSYSCLKDIV(WM8960_SYSCLK_DIV_BY_2);
  codec.setBCLKDIV(4);
  codec.setDCLKDIV(WM8960_DCLKDIV_16);
  codec.setPLLN(7);
  codec.setPLLK(0x86, 0xC2, 0x26); // PLLK=86C226h
  //codec.setADCDIV(0); // Default is 000 (what we need for 44.1KHz)
  //codec.setDACDIV(0); // Default is 000 (what we need for 44.1KHz)
  codec.setWL(WM8960_WL_16BIT);
  //codec.writeRegister(0x09, 0b00100000);


  codec.enablePeripheralMode();
  //codec.enableMasterMode();
  //codec.setALRCGPIO(); // Note, should not be changed while ADC is
enabled.


  // Enable ADCs and DACs
```

```
  codec.enableAdcLeft();
  codec.enableAdcRight();
  codec.enableDacLeft();
  codec.enableDacRight();
  codec.disableDacMute();

  //codec.enableLoopBack(); // Loopback sends ADC data directly into DAC
  codec.disableLoopBack();

  // Default is "soft mute" on, so we must disable mute to make channels
active
  codec.disableDacMute();

  codec.enableHeadphones();
  codec.enableHeadphoneZeroCross();
  codec.enableOUT3MIX(); // Provides VMID as buffer for headphone ground

  Serial.println("Volume set to +0dB");
  codec.setHeadphoneVolumeDB(0.00);

  Serial.println("Codec Setup complete. Listen to left/right INPUT1 on
Headphone outputs.");
}

// Code for setting up the i2s communication, called in setup()
void i2s_install() {
```

```
  // Set up I2S Processor configuration
  const i2s_driver_config_t i2s_config = {
    .mode = i2s_mode_t(I2S_MODE_MASTER | I2S_MODE_RX | I2S_MODE_TX),
    .sample_rate = 44100,
    .bits_per_sample = i2s_bits_per_sample_t(16),
    .channel_format = I2S_CHANNEL_FMT_RIGHT_LEFT,
    .communication_format = i2s_comm_format_t(I2S_COMM_FORMAT_STAND_MSB),
    .intr_alloc_flags = 0,
    .dma_buf_count = 8,
    .dma_buf_len = bufferLen,
    .use_apll = false,
    .tx_desc_auto_clear = false,
    .fixed_mclk = 0,
    .mclk_multiple = i2s_mclk_multiple_t(I2S_MCLK_MULTIPLE_DEFAULT),
    .bits_per_chan = i2s_bits_per_chan_t(I2S_BITS_PER_CHAN_DEFAULT)
  };

  i2s_driver_install(I2S_PORT, &i2s_config, 0, NULL);
}

void i2s_setpin() {
  // Set I2S pin configuration
  const i2s_pin_config_t pin_config = {
    .mck_io_num = -1,
    .bck_io_num = I2S_SCK,
    .ws_io_num = I2S_WS,
```

```
    .data_out_num = I2S_SDO,
    .data_in_num = I2S_SD
  };

  i2s_set_pin(I2S_PORT, &pin_config);
}

void setup()
{
  Serial.begin(115200);

  Wire.begin();

  if (codec.begin() == false) //Begin communication over I2C
  {
    Serial.println("The device did not respond. Please check wiring.");
    while (1); // Freeze
  }
  Serial.println("Device is connected properly.");

  codec_setup();

  // Set up I2S
  i2s_install();
  i2s_setpin();
  i2s_start(I2S_PORT);
```

```
  web_interface_setup();
}



float tWave = 125;
float direction = -0.5;
int iClip = 0;
void triangolo(){
    // Update the value based on the current direction
    if (iClip == 0){
      tWave += direction*tremoloValue.toInt();
    }

      // Check if the value has reached the lower or upper limit
      if (tWave <= 107) {
        iClip++;
        if(iClip == 25){
            direction = 0.5;
            iClip = 0;
            tWave = 107;
        }
      }

      else if (tWave >= 125) {
        iClip++;
```

```
        if(iClip == 25){
            direction = -0.5;
            iClip = 0;
            tWave = 125;
        }
      }
    codec.setHeadphoneVolume(tWave);
}
void fuzz(float* audioData, int numBytes, int gainFactor)
{
  const float mixGain = 2*gainFactor; // tone shift factor
  // Apply the fuzz effect to each sample in the input audio data
  int numSamples = numBytes / sizeof(float);

  for (int i = 0; i < numSamples; i++) {
    // Apply tone shift to the sample
    float ampedSample = audioData[i] * mixGain;
    // Mix the original and shifted samples
    audioData[i] =  (audioData[i] + ampedSample) + audioData[i];
  }
}

void eightbit(float* audioData, int numBytes){
  const float eightbitfactor = 10;

  int numSamples = numBytes / sizeof(float);
```

```
  for (int i = 0; i < numSamples; i++) {

    float ampedsample = audioData[i]*eightbitfactor;
    audioData[i] = audioData[2*i];
  }
}

void loop()
{
  if(eightBitValue.toInt() == 0){
  size_t bytesIn = 0;
  size_t bytesOut = 0;
  esp_err_t result = i2s_read(I2S_PORT, &sBuffer, bufferLen, &bytesIn,
portMAX_DELAY);

  // codec.writeRegister(0x09, 0b00100000); //uncomment to apply 8-bit
effect

  // If no distortion, no fuzz, and no tremolo (clean)
  if (distortionValue.toInt()== 0 && tremoloValue.toInt() == 0 &&
fuzzValue.toInt() == 0)
  {
    // Set PGA values to 0 dB for no distortion
    codec.setLINVOLDB(0);
    codec.setRINVOLDB(0);
```

```
    // Immediately play back what was read in
    esp_err_t result_w = i2s_write(I2S_PORT, &sBuffer, bytesIn, &bytesOut,
portMAX_DELAY);

    // If there is an I2S write error, let us know on the serial terminal
    if(result_w != ESP_OK){
      Serial.print("I2S write error.");
    }
  } // ends the if-statement for no distortion, no fuzz, and no tremolo

  // If distortion and no delay and no fuzz
  if (distortionValue.toInt() != 0 && tremoloValue.toInt() == 0 &&
fuzzValue.toInt() == 0)
  {
      // Sets PGA values for distortion using function
      codec.setLINVOLDB(3*distortionValue.toInt());
      codec.setRINVOLDB(3*distortionValue.toInt());

     // Play back what was read in with distortion
     esp_err_t result_w = i2s_write(I2S_PORT, &sBuffer, bufferLen,
&bytesOut, portMAX_DELAY);

  } // ends distortion and no delay case

  // If tremolo and no distortion and no fuzz
```

```
  if (distortionValue.toInt() == 0 && tremoloValue.toInt() != 0 &&
fuzzValue.toInt() == 0){

    // Set PGA values to 0 dB for no distortion
    codec.setLINVOLDB(0);
    codec.setRINVOLDB(0);

    triangolo();
    esp_err_t result_w = i2s_write(I2S_PORT, &sBuffer, bytesIn, &bytesOut,
portMAX_DELAY);

   }

   // If fuzz and no distortion and no tremolo
   if (distortionValue.toInt() == 0 && tremoloValue.toInt() == 0 &&
fuzzValue.toInt() != 0)
   {
    // Set PGA values to 0 dB for no distortion
    codec.setLINVOLDB(0);
    codec.setRINVOLDB(0);

    float* audioData = (float*) &sBuffer; // Cast buffer as float*
    int numBytes = bytesIn; // Get the number of bytes of audio data

    fuzz(audioData, numBytes, fuzzValue.toInt());
```

```
    esp_err_t result_w = i2s_write(I2S_PORT, &sBuffer, bytesIn, &bytesOut,
portMAX_DELAY);

   }

  // If distortion and tremolo and no fuzz
  if (distortionValue.toInt() != 0 && tremoloValue.toInt() != 0 &&
fuzzValue.toInt() == 0)
  {
    //Set PGA values for distortion
    codec.setLINVOLDB(3*distortionValue.toInt());
    codec.setRINVOLDB(3*distortionValue.toInt());

    triangolo();
    esp_err_t result_w = i2s_write(I2S_PORT, &sBuffer, bufferLen,
&bytesOut, portMAX_DELAY);
  }

  // If tremolo and fuzz and no distortion
  if (distortionValue.toInt() == 0 && tremoloValue.toInt() != 0 &&
fuzzValue.toInt() != 0)
  {
    float* audioData = (float*) &sBuffer; // Cast buffer as float*
    int numBytes = bytesIn; // Get the number of bytes of audio data
    fuzz(audioData, numBytes, fuzzValue.toInt());

```

```
    triangolo();

    esp_err_t result_w = i2s_write(I2S_PORT, &sBuffer, bufferLen,
&bytesOut, portMAX_DELAY);

  }

  }

  // 8-bit switch

  if (eightBitValue.toInt() != 0){

    size_t bytesIn = 0;

    size_t bytesOut = 0;


    esp_err_t result = i2s_read(I2S_PORT, &sBuffer, bufferLen/2, &bytesIn,
portMAX_DELAY);

     // Set PGA values to 0 dB for no distortion

    codec.setLINVOLDB(0);

    codec.setRINVOLDB(0);


    float* audioData = (float*)&sBuffer; // Cast buffer as float*

    int numBytes = bytesIn; // Get the number of bytes of audio data


    eightbit(audioData, numBytes);


    esp_err_t result_w = i2s_write(I2S_PORT, &sBuffer, bufferLen/2,
&bytesOut, portMAX_DELAY);


  }

}
```