Hand Gesture Audio Effects System

By

Zachary Baum
(zbaum2@illinois.edu)
Sergio Bernal
(sergiob2@illinois.edu)
Sarthak Singh
(singh94@illinois.edu)

Final Report for ECE 445, Senior Design, Spring 24

TA: Zicheng Ma

1 May 2024

Team No. 39

Abstract

This project proposes a gesture-controlled audio effects processor to address accessibility and practicality concerns in audio production and live music settings. By leveraging camera-based gesture detection, users can manipulate audio effects in real-time without the need for traditional physical interfaces like buttons or knobs. This innovative approach offers a dynamic and expressive means of controlling audio effects, particularly beneficial for individuals with disabilities or those navigating complex production environments. With the ability to adjust various effect parameters through intuitive hand gestures, the device provides a streamlined alternative to conventional DJ controllers or hardware processors, enhancing both accessibility and versatility in audio manipulation.

Contents

1. Introduction

1.1 Purpose

Both in video production and live music, individuals often want the ability to apply effect to audio in real time. There are solutions for this such as DJ controllers or hardware effects processors. For all of these systems, you usually need to click buttons, turn knobs, etc. Unfortunately, not all people have the ability to perform these functions due to disabilities. Furthermore, those within live settings, operating physical equipment may be challenging to manage with all the other aspects of production.

This project aims to develop a gesture-controlled audio effects processor. This device will allow users to manipulate audio effects through hand gestures, providing a more dynamic and expressive means of audio control. The device will use a camera to detect gestures, which will then adjust various audio effect parameters in real-time. This will allow for the same features of something like a DJ mixer with less equipment & the ability to with your hands free.

1.2 Functionality

Our first high-level requirement was that the gesture-controlled audio effects processor is able to detect hand gestures using a camera with 95% accuracy +-5% within 5 seconds of making the gesture. This high-level requirement is necessary so that the user is not deterred by a slow detection system. In addition, it is much more feasible to achieve based on the hardware being used to drive the subsystem pertaining to detecting hand gestures.

Our second high-level requirement was that the gesture-controlled audio effects processor is able to apply five audio effects to an input audio signal and send the new audio signal to the external speaker within a second of receiving a control signal from the gesture detection subsystem with a tolerance of +-½ of a second. The purpose of the timing constraint is to make sure that the user does not notice a delay between a hand gesture detection and the duration needed to execute an audio effect on the input audio signal.

Our last high-level requirement was that the external display should accurately show the current playing sound effect with 99% accuracy with a tolerance of +-1%. This requirement is integral to make sure that the user gets accurate feedback on what audio effect is being applied onto the input audio signal based on a message shown by the external display.

1.3 Subsystem Overview

1.3.1 Power Subsystem

This subsystem provides essential power distribution to all components within the three other subsystems. Specifically, it gives 5 V to the Raspberry Pi and Amplifier. Also, it supplies 3.3 V (generated by a linear regulator) to the rest of the components such as the external display and microcontroller.

1.3.2 Gesture Control Subsystem

This subsystem consists of a Raspberry Pi that communicates with an imaging sensor (camera) with the use of a camera serial interface (Csi) communication to receive captured frames. These frames are

analyzed by software to detect hand gestures, and control signals with respect to a type of hand gesture are sent to the GPIO pins of the microcontroller. The programmed microcontroller will execute the software pertaining to the Audio Subsystem and User Interface Subsystem.

1.3.3 Audio Subsystem

This subsystem contains a microSD, amplifier, and speaker. The microSD contains the input audio signal (.wav file) and sends its data to the microcontroller using serial peripheral interface (SPI) communication [17]. After the microcontroller receives control signals coming from the Gesture Control Subsystem, an algorithm is performed to apply one of five audio effects (reverb, distortion, gain up, gain down, or low-pass filter) to the input audio signal. The modified audio signal gets converted from digital to analog within the microcontroller, and the analog audio signal goes through an amplifier that boosts it to a speaker.

1.3.4 User Interface (UI) Subsystem

This subsystem is comprised of a liquid crystal display (LCD) which communicates with the microcontroller through inter-integrated circuit (I^2C) communication [12]. The control signals being sent from the Gesture Control Subsystem determines what seven characters will be displayed to the LCD. The seven character message will represent either an audio effect applied to the input audio signal or no hand gesture detection.. This subsystem is essential for the user to know which audio effect is currently being applied to the input audio signal based on a hand gesture. Moreover, this project does not assume that the user can distinguish between audio effects through sound alone since they may not have prior exposure to the different audio effects.

1.3.5 Top-Level Diagram

The following top-level diagram (Figure 1) shows how the subsystems interconnect with each other.

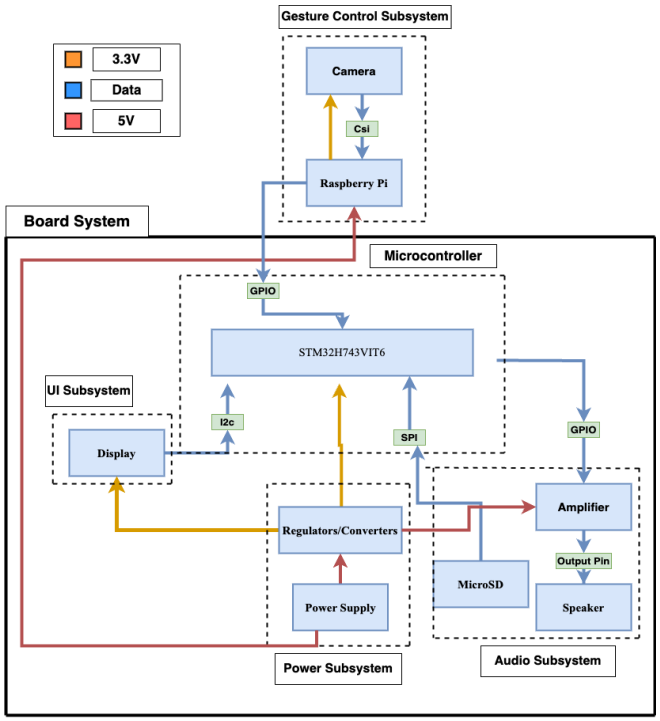

Figure 1. Top-Level Diagram of the Hands Gesture Audio Effects System

2. Design

2.1 Power Subsystem Design

The power subsystem used a DC barrel jack (PJ-102AH) to connect to a 5V 4.0 A wall power adapter which is the primary source for the entire project. In one path, power is going straight into the USB-A connector (87583-3010RPALF). Furthermore, a micro USB B to USB-A cable is used to connect the Raspberry Pi to the power source. In the other path, power is going into the linear regulator (LM1085ISX-3.3/NOPB) which steps down the input 5 V to 3.3 V. This lower voltage is used to power most of the components on the board with the exception of the amplifier. The amplifier is connected to the 5 V line since that is the minimal voltage to power it based on its datasheet. This original design is shown in Figure 2.

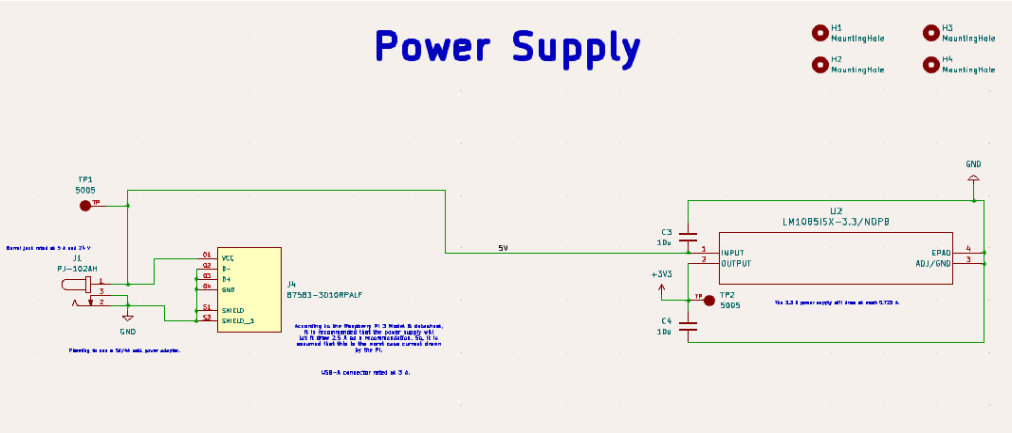

Figure 2. Original Power Supply Design

For the linear regulator, this part was chosen based on its thermal dissipation rating. Moreover, the worst case current drawn (754 mA) at the linear regulator output was found based on datasheets for the components listed in Table 1.

Table 1. Worst-Case Scenario Current Drawn

Component	Current drawn (worst case) @ 3.3 V
STM32H743VIT6	700 mA
LCD	1.5 mA
microSD connector	50 mA
Amplifier	< 2.5 mA

Next, the thermal power dissipation equation (Equation 1.1) given by the ECE-445 website was utilized such that it has an assumption that the input current to the linear regulator is the same as the output current.

$$P_D = i_{out}(v_{in} - v_{out}) \tag{1.1}$$

The thermal power dissipation for the linear regulator based on the worst-case drawn current is calculated below as roughly 1.28 W.

$$P_D \approx (0.754\ A)(5\ V - 3.3\ V) \approx 1.28\ W \tag{1.2}$$

Hence, the linear regulator was chosen to have a power dissipation rating much higher than 1.28 W. On DigiKey, a relatively cheap linear regulator (LM1085ISX-3.3/NOPB) that met that requirement was rated for 3.0 A at 1.5 V [13]. Using Equation 1.1, the power rating for that linear regulator is calculated to be 4.50 W.

$$P_{D,rating} \approx (3.0\ A)(1.5\ V) \approx 4.50\ W \tag{1.3}$$

Finally, the junction temperature for this linear regulator was observed to see if it can handle hot ambient temperature. The thermal junction temperature equation (Equation 1.4) is given by the ECE-445 website. Note that $R_{\theta JA}$ is the junction-to-ambient thermal resistance given by the linear regulator's datasheet as 22.8 degree Celsius per Watt [13].

$$T_j = i_{out}(v_{in} - v_{out})R_{\theta JA} + T_a \tag{1.4}$$

Assuming that there is an ambient temperature outside of the product at roughly 38 degree Celsius, then the junction temperature is calculated to be around 67.184 degree Celsius.

$$T_j \approx (1.28\ W)\left(22.8\ \frac{℃}{W}\right) + 38\ ℃ \approx 67.184\ ℃ \tag{1.5}$$

This temperature is much smaller than the recommended temperature (125 degree Celsius) stated by the linear regulator's datasheet [13]. Hence, a heat sink will not be required for the linear regulator. As a final note, the linear regulator has a capacitor at its input and output in order to stabilize the voltages. The capacitor values were given by the linear regulator datasheet.

A design alternative made (after the Design Review) for the Power Subsystem was the addition of an eFuse (TPS259814ARPWR) to protect the main printed circuit board components from incorrect voltages being connected to the DC barrel jack or any shorts that have been formed due to poor soldering.

The voltage lockout circuit design for the eFuse was given by its datasheet as illustrated in Figure 3 [14]. Moreover, the eFuse required resistors to set up its undervoltage lockout (UVLO) and overvoltage lockout (OVLO) [14]. The respective node voltage gets compared with a threshold [14]. If the UVLO voltage pin or OVLO voltage pin is above or below the threshold, power will be shut off to the entire board [14]. The Raspberry Pi has an exception since its needed track width for worst-case drawn current is too large to connect to the eFuse output pin.

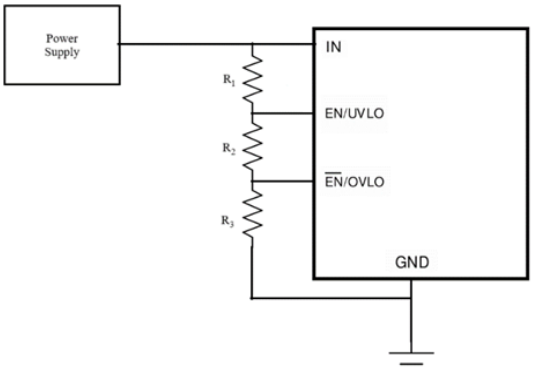

Figure 3. Voltage Divider Circuit for eFuse Voltage Lockout

Based on the circuit shown in Figure 3, I obtained Equation 2.1 and Equation 2.2.

$$\frac{V_{UVLO}}{V_{IN}} = \frac{R_2+R_3}{R_1+R_2+R_3} \tag{2.1}$$

$$\frac{V_{OVLO}}{V_{IN}} = \frac{R_3}{R_1+R_2+R_3} \tag{2.2}$$

After using maximum and minimum voltage thresholds for the lockout pins from the eFuse datasheet, the resistances were calculated to be R_1 = 7.5 kΩ, R_2 = 510 Ω, and R_3 = 2 kΩ. These resistances only allow an input voltage range of 4.8 V to 5.4 V to be used as a source from the wall power adapter. Otherwise, the power will shut off at the eFuse output when input voltage is outside of that range.

Aside from undervoltage and overvoltage protection, the eFuse has a current limiter in case of a short occuring somehow on the custom PCB side [14]. Using Equation 3.1 given by the eFuse's datasheet, the required resistor to connect to its current limiter pin was calculated to be 5.6 kΩ [14]. The current limiter being set to 1.176 A is more than enough since the worst case current drawn at the eFuse output is smaller than that value. Any current above the current limiter will trigger the eFuse to turn off power at its output since a short circuit has occurred somewhere in the PCB.

$$R_{ILM} = \frac{6585}{I_{LIM}} \tag{3.1}$$

$$I_{LIM} = \frac{6585}{5600} \approx 1.176\ A \tag{3.2}$$

As for the final schematic design of this power subsystem, all the component connections are shown below in Figure 4.

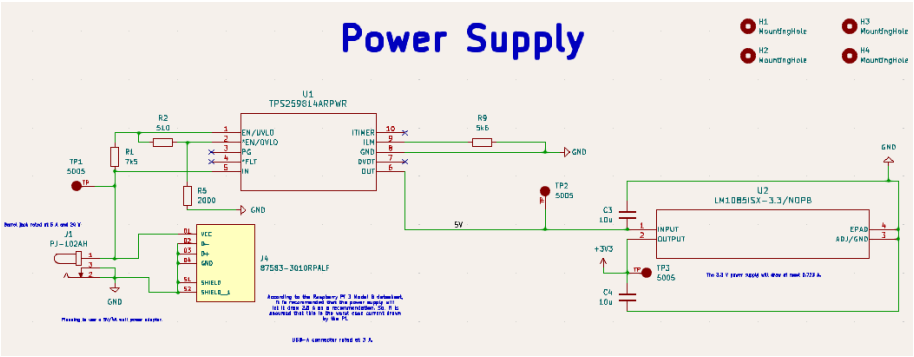

Figure 4. Final Power Supply Circuit Schematic

2.2 Gesture Control Subsystem Design

For the design of the Gesture Control Subsystem, a camera module (made by Raspberry Pi company) was used to send captured frames via Csi to the Raspberry Pi Model 3b. A single captured frame gets analyzed by software to detect a hand gesture which is translated into a three-bit control signal passed to the global parameter input (GPI) pins of the microcontroller (STM32H7VIT6) that is a part of the main printed circuit board (PCB). In short, three global parameter output (GPO) pins will be utilized from the Raspberry Pi to form a total of eight unique signals that this subsystem can send to the microcontroller. The layout of the Raspberry Pi 3b layout pins are shown in Figure 5, and the pins to be used will be GPIO 17, 22, and 27.

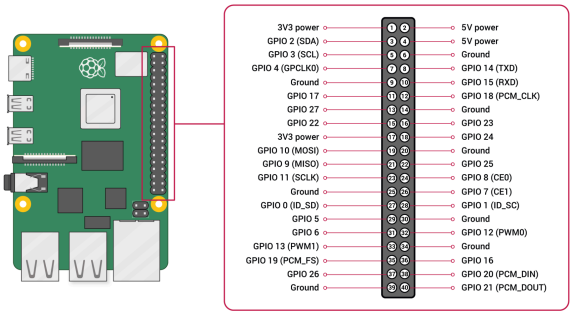

Figure 5. GPIO layout of Raspberry Pi 3B

As a final note, a Raspberry Pi was chosen to handle the image processing aspect of the design since it requires a more complex algorithm that needs a lot of memory for the libraries associated with the software. According to its datasheet, it has about 1 Gb of memory which should be plenty for the Gesture Control Subsystem [18].

2.3 Audio Subsystem Design

The design of the Audio Subsystem revolved around applying one of the five audio effects shown below using software:

Reverb: Simulates the echo and ambiance of a physical space, adding depth to the audio. To implement, we will use the Schroeder Reverb Algorithm, which uses multiple feedback delay lines and all-pass filters to simulate room ambiance.

Distortion: Clips and modifies the audio waveform, adding complexity and texture by boosting aggressive tones

Gain Adjustment (Up and Down): Varies the amplitude of the audio signal, controlling volume. Multiply each incoming audio sample by a gain factor. A gain factor greater than 1 increases volume, while a gain factor less than 1 decreases it.

Low-pass Filter: Attenuates frequencies above a cutoff, reducing high-frequency noise or brightness. Implement the filter using a simple discrete time equation: $y[n] = \alpha * x[n] + (1 - \alpha) * y[n-1]$, where $x[n]$ is the input signal, $y[n]$ is the output signal, n is the sample index, and α (alpha) is the filter coefficient related to the cutoff frequency.

Note: The CMSIS-DSP library is used in software for optimized DSP functions. The software implementation for audio is shown in the Appendix C section.

In terms of hardware design, the microSD connector required a 3.3 V power supply coming from the Power Subsystem [15]. Also, the data lines for the connector were connected to the SPI ports of the microcontroller as shown in Figure 6 [17]. The SPI master in-slave out (MISO) port was mainly used to retrieve the .wav file for the input audio signal. The speed of the data retrieval was determined by the SPI serial clock (SCL) frequency.

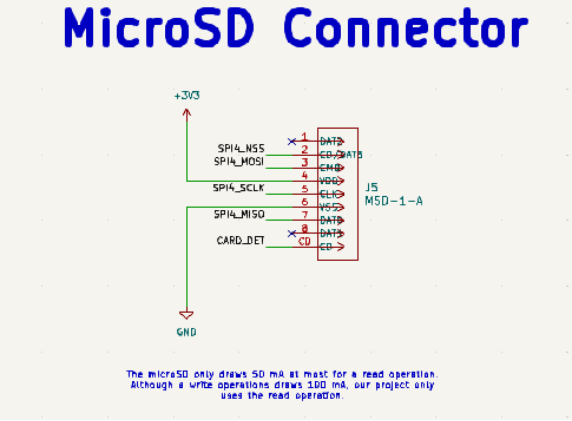

Figure 6. MicroSD Connector Schematic

As for the amplifier circuit set-up shown in Figure 7, the required circuitry was given by the datasheet of the amplifier (LM386N-1/NOPB) [16]. The capacitors act as filters for the analog audio signal coming from the microcontroller's internal digital-to-analog (DAC) converter output. It basically makes the audio sound more clearly at the speaker positive port (POS_SPK). Lastly, the amplifier gets its power from the 5 V line of the Power Subsystem.

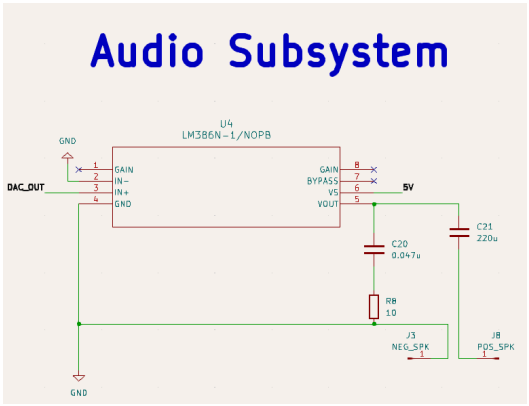

Figure 7. Amplifier circuit for Audio Subsystem

2.4 User Interface Subsystem Design

The design of the UI Subsystem consists of a LCD that communicates with the microcontroller through I^2C communication. The LCD has the potential to display up to 20 characters on two lines. However, the message displayed by the LCD is seven characters for each audio effect (including no hand gesture detection) as shown in Table 2.

Table 2. 7-Character Words displayed by the LCD

DISPLAYED 7-CHARACTER MESSAGE	AUDIO EFFECT APPLIED
“NO-HAND”	No hand gesture detection (normal input audio)
“REVERB!”	Reverb
“DISTORT”	Distortion
“GAIN-UP”	Gain Up
“GAIN-DN”	Gain Down
“LO-PASS”	Low Pass

A seven character message was chosen since the messages had to be the same number of characters to achieve timing constraints and a seven character message led to better interpretation of the displayed message. The “magic” behind the LCD displaying a specific message based on hand gesture is pulled off through the use of I^2C communication between the microcontroller and the LCD [12]. The I^2C communication begins with a start condition (S) and a slave address that gets sent to the LCD on the serial data (SDA) bus [12]. If the slave address that identifies the controlled device is correct, an acknowledgement bit gets sent to the microcontroller on the SDA bus [12]. Then, a byte can be sent to the LCD on the SDA bus to perform an operation on the LCD, and an acknowledgement bit gets sent back to the microcontroller to confirm that the byte has been received properly by the LCD [12]. For the chosen LCD, it has a control byte that determines whether the following data byte represents an instruction or data to be read from/written to the internal RAM [12]. A controller called the ST7036i (embedded within

the LCD) handles any action performed on the LCD by looking at the data byte that got written into either the instruction register or data register [12]. Finally, the communication ends with a stop condition (P) [12]. The overall I²C communication between the microcontroller and LCD is shown by Figure 8. Note that the R/W represents the bit that determines whether data is being read from the LCD (bit = 1) or being sent to be written into the LCD (bit = 0) in either the instruction register or data register [12]. However, a read operation was not necessary for the intended implementation of the LCD since only characters being written into the LCD was sought for the design.

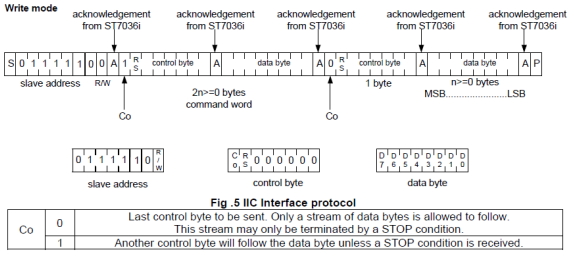

Figure 8. I²C communication Write Operation

The LCD datasheet listed different instructions to be used to modify the display settings such as character size; however, given instructions to initialize the LCD to its normal operation were used instead of having a custom initialization [12]. These instructions are shown by Figure 9. The most important piece of sending an instruction is just the delay time in order to give enough time for the controller to execute the instruction [12]. Overall, the entire LCD software revolved around Figure 8 and Figure 9, and its C code implementation is shown in the Appendix C section.

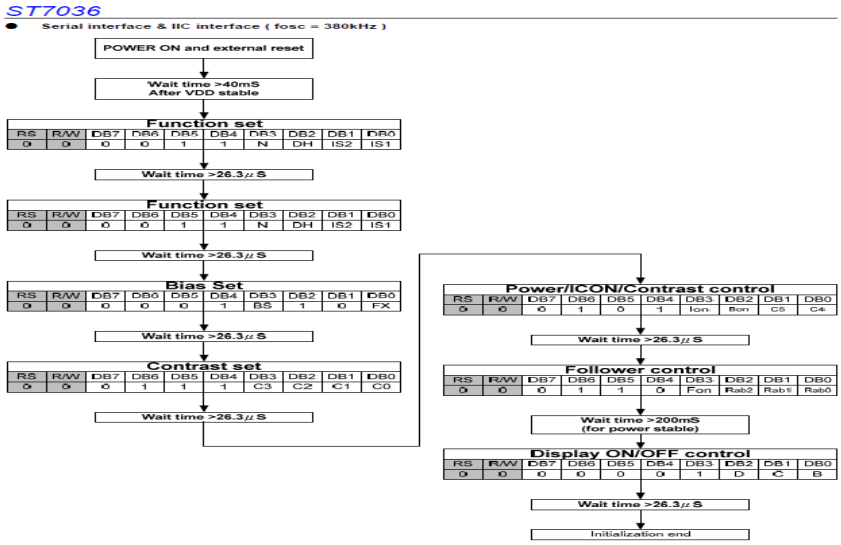

Figure 9. Flow Chart of the LCD Initialization

In terms of hardware design on the main PCB, the LCD required pull-up resistors for the SDA bus line and serial clock (SCL) lines to prevent any short from occurring during I²C communication as shown in Figure 10. Also, a separate breakout PCB was created in order to position the LCD better in the physical enclosure of the project design. A description of this enclosure is shown in the Appendix A section. The LCD breakout PCB connects to the schematic of Figure 10 through the use of jumper wires. Note that the debugger/programmer connections for the STM32H743VIT6 microcontroller was combined with the LCD connection header to save space on the main PCB. Those connections require pull-up resistors for the same reason as the LCD I²C connections.

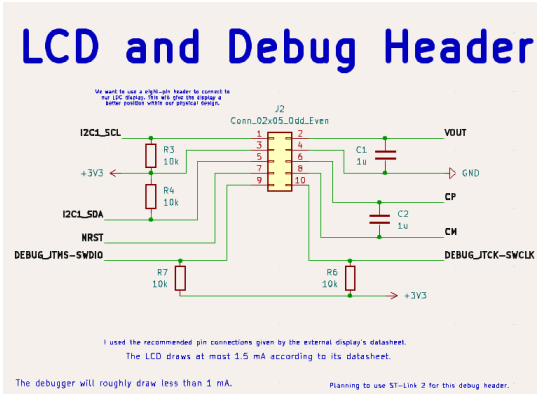

Figure 10. LCD circuit schematic

3. Design Verification

The overall final design of this project consisted of a 3D printed physical enclosure, PCB that contains the microcontroller, LCD breakout PCB, Raspberry Pi, and Camera. The Power Subsystem, Gesture Control Subsystem, and UI Subsystem were all implemented in the design. The Audio Subsystem was partially complete, but it did not fit the full RV table set at the start of the course. Overall, subsystem verification tests were performed to verify the correctness of the project device. The high-level requirements being met was determined by the subsystem verification results.

3.1 Power Subsystem Verification Results (Table 3)

Requirement	Verification	Successful	Rationale
The linear regulator should provide 3.3 V ± 0.5% at its output to the necessary supply pins of the custom PCB.	1. Connect to the 3.3 V test pin and GND test pin on custom PCB using the connectors coming from an oscilloscope. 2. Add a measurement for the average voltage. 3. Measure the peak-to-peak voltage (ripple) using horizontal cursors of the oscilloscope. 4. Check if the peak-to-peak voltage is at most 0.033 V which is 0.5% above and below 3.3 V.	Yes	The 3.3 V output coming from the linear regulator was stable at 3.29575 V which is within the minus 0.5% range. Thus, the requirement was met and the oscilloscope reading is shown below in Figure 11.

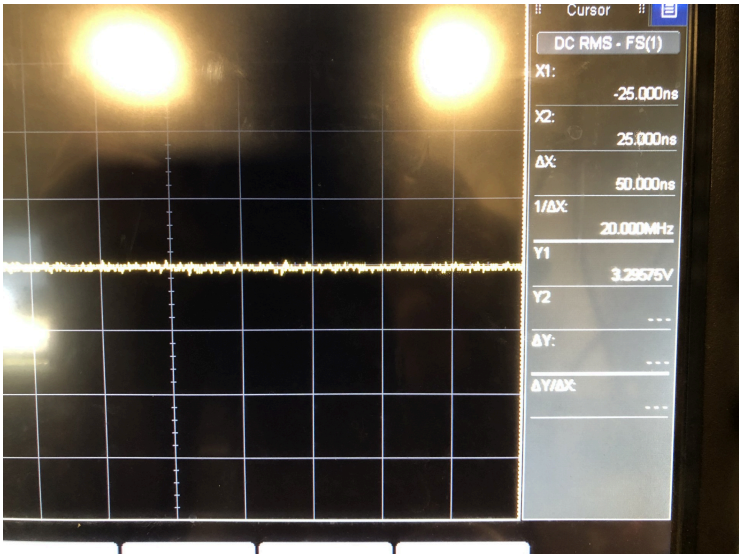

Figure 11. Power Subsystem Verification Test

Overall, the required voltage for most of the components of the PCB was achieved which means each component such as the microcontroller has enough power to work properly in the circuit.

3.2 Gesture Subsystem Verification Results (Table 4)

Requirements	Verifications	Successful	Rationale
The image processing algorithm that analyzes the captured images from the camera must have a timing of at least 20 frames per second to get an accurate detection of a hand gesture.	1. Use the <time.h> and time() library to measure the start and end time for a single while loop iteration (used to capture/read an image from the camera using OpenCV library). 2. Create the variable that stores the start time. 3. Create the variable that stores the end time. 4. Subtract the variable storing the start time from the variable that stores the end time. 5. Take the reciprocal of that value to get the number of frames that got analyzed per second. Note that this procedure is for software done in C++.	Yes	A timestamp was created within the gesture detection script to measure the time it took for a single while loop iteration that contains the read function for a single captured frame and the measured fps of the while loop was around 24 frames per second. This gives a sense of accuracy where the algorithm is not taking too long to analyze an image before reading the next

			captured image.
The gesture detection algorithm must detect a hand gesture in less than 5 seconds.	1. Connect the female end of a jumper wire to each STATE signal (3 in total) of the Raspberry Pi's pin headers. 2. Connect the male end of each jumper wire to a respective LED's cathode pin on a breadboard. 3. Connect each LED's anode pin to a respective male end of a jumper wire on the breadboard. 4. Connect the female end of each jumper wire (connected to LED anode pin) to its respective STATE pin connector on the custom PCB. 5. Start a timer for when the three LEDs are off (default state) and another person makes a hand gesture to the camera. 6. Stop timer until the LEDs change to an applied audio effect state.	Yes	A total of 50 hand gestures were shown to the camera. The timer started from when the gesture was made to when the LEDs changed color. On average, it took about three seconds which meets the requirement.

A testbench was created to evaluate the five second timing constraint of the Gesture Control Subsystem in a modular manner. In Figure 12, there is a Raspberry Pi Model 3b and its three GPIO pins are connected on a breadboard to their respective LEDs. Each LED represents one of the three control bits that the Raspberry Pi would pass to the microcontroller of the main PCB. Using this circuitry setup, an average timing of three seconds was measured which fits the subsystem requirement.

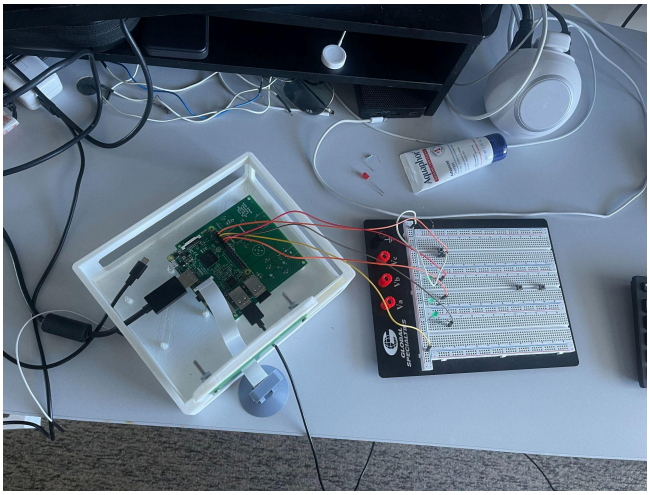

Figure 12: Gesture detection Subsystem Modular Test Bench

3.3 Audio Subsystem Verification Results (Table 5)

Requirement	Verification	Successful	Rationale
The sound being outputted by the speaker must be less than 80 dB for safety purposes	1. Connect the audio source to the input of the Audio Subsystem. 2. Set the audio source to produce a continuous sine wave signal at 1 kHz. 3. To measure the sound db, a sound level meter app on a phone will be used. Position the Sound Level Meter at a distance of 1 meter from the speaker, aligned with the center of the speaker. 4. Increase the volume of the audio source until the sound level meter stabilizes. 5. Record the maximum sound pressure level (SPL) reading displayed by the Sound Level Meter. 6. Repeat the procedure three times and calculate the average SPL to ensure reliability.	No	The microSD connector and amplifier did not arrive on time so a speaker test was not possible.
The effect applied to the input audio signal must be heard clearly such that there should not be any static sound unless it is a distortion audio effect	1. Connect the audio source to the Audio Subsystem's input and the output to both the Audio Analyzer and headphones/speakers. 2. Set the audio source to produce a clean sine wave signal at 1 kHz. 3. Apply the desired audio effect using the microcontroller within the Audio Subsystem. 4. Conduct a subjective listening test by playing the processed audio through headphones/speakers and noting any static noise or undesired artifacts. 5. Adjust the effect parameters and repeat the test if static noise is detected in the absence of a distortion effect. Add data to report	Yes	The audio effects were able to be tested on a separate compiler. All audio effects were tested successfully and the quality of sound did not contain any static sound besides the distortion effect.

The Audio Subsystem hardware was not tested due to delay of the arrival of components; however the audio effects code was able to be tested separately to see the applied audio effect on the input audio signal. Figure 13 compares the original wav file on top to the distorted version of the audio below it. From this figure, it is observed that the distortion effect is being applied due to the amplitude flattening out.

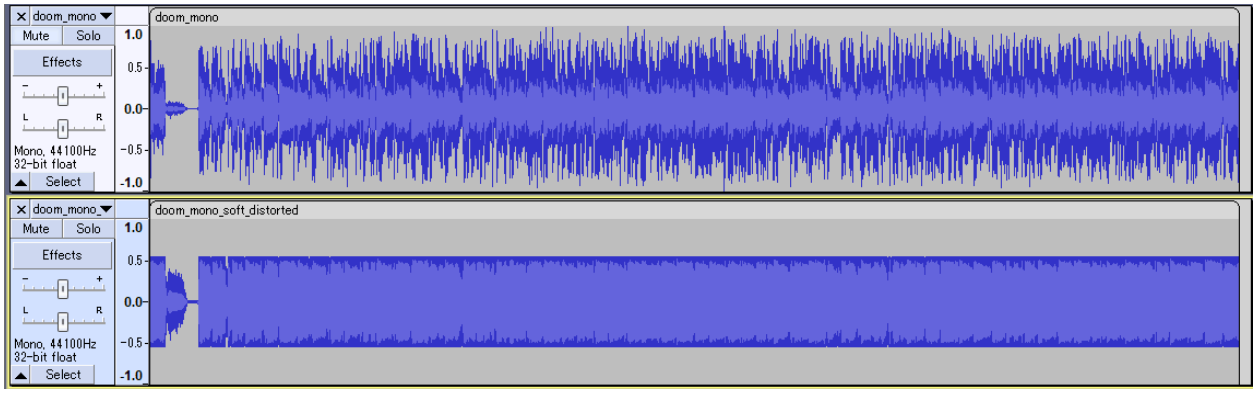

Figure 13. Audio Effect Showcase of Distortion

3.4 UI Subsystem Verification Results (Table 6)

Requirement	Verification	Successful	Rationale
The display must show the user what audio effect has been applied to the input audio signal, 0.1 seconds (at most) after the microcontroller communicates with it via I^2C that a unique hand gesture has been detected by the gesture detection algorithm.	1. Connect the female end of a jumper wire to each STATE signal (3 in total) of the Raspberry Pi's pin headers. 2. Connect the male end of each jumper wire to a respective LED's cathode pin on a breadboard. 3. Connect each LED's anode pin to a respective male end of a jumper wire on the breadboard. 4. Connect the female end of each jumper wire (connected to LED anode pin) to its respective STATE pin connector on the custom PCB. 5. Start a timer for when the three LEDs change to a new STATE value. 6. Stop timer until the display changes to the STATE value's name. If	Yes	Based on the UI subsystem software, it would take less than 50 ms for an entire 7-character word to be displayed after the 3-bit STATE variable changes. Thus, the requirement was met since that is much smaller than 0.1 seconds (100 ms). In order to get a timestamp to see the fast timing, the UART ports would have been needed to see a timestop measurement on a computer's console. However, UART was not implemented into the PCB design since it was not necessary in terms of functionality of the project.

	we cannot time how quickly it is, safe enough to say it's quicker than .2 seconds		

3.5 High Level Requirement Verification Results (Table 7)

Requirement	Successful	Rationale
The gesture-controlled audio effects processor is able to detect hand gestures using a camera with 95% accuracy +-5% within 5 seconds of making the gesture.	Yes	A testbench was made where the GPIO pins are connected to LEDs on a breadboard. Hand gestures were made in front of the camera and a timer was recorded once the LEDs changed to a different state and correctness was verified between hand gesture and state. 50 gestures were done in front of the camera and the average time was about three seconds. Due to satisfying both subsystem requirements, the accuracy of hand gesture detection was about 98%.
The gesture-controlled audio effects processor is able to apply 5 digital effects to an audio signal and apply that effect to the external speaker within a 1 second of receiving a signal from the gesture detection subsystem with a tolerance of +-½ of a second.	Yes	Code was written for the microcontroller to take in a GPI and apply five different effects, but due to parts not arriving on time and components breaking during soldering, did not have time to test the speaker. However, code was tested for each effect on a regular c compiler and verified that the code for each effect functioned properly. Also, tested how long it takes for the audio algorithm to go through the main while loop that controls which effect is played and we got around 50ms for each effect.
The external display should accurately show the current playing sound effect with 99% accuracy with a tolerance of +-1%	Yes	The software for the LCD was able to go through each loop in less than 50 ms which satisfies a subsystem requirement. As a result of this speed, the LCD was able to accurately show the audio effect applied based on the selected hand gesture every time.

4. Costs

All the parts needed for the project design are listed in Appendix B. After summing the cost for each part, the total part cost is **$148.63.** For each partner, the typical hourly rate is $50 per hour. Thus, $150 per hour is the hourly rate for all three people in the group. As for the hours needed to complete the project, about an average 20 hours per group member per week for the entire next eight weeks which is a total of 480

hours. As a result, the total labor cost after multiplying by a 2.5x overhead multiplier would be **$180000**. As a result, the total cost is calculated to be **$180148.37**.

5. Schedule

A weekly schedule about how tasks would be divided up between each team member is shown in Table 8.

Table 8. Weekly Schedule

Week	Task	Person
February 26	Order all components	Everyone
	Assign high level subsystems to each member	Everyone
	Schematic Completed	Sergio
	Install prerequisite software on Raspberry Pi	Sarthak
	Finalize five audio effects for our project	Zachary
March 4	PCB Done & audited (delayed)	Sergio
	Be able to view camera from Pi	Sarthak
	Write/Test DSP code on computer for five effects	Zachary
March 11	Spring Break	
March 18	Test Power System (delayed) First PCB Version Done & Ordered PCB	Sergio
	Test Audio System software	Zach
	Be able to detect a hand	Sarthak
March 25	Begin writing I^2C code for LCD	Sergio
	Ordered a revised PCB with a breakout PCB for the LCD	Everyone
	Write at least one effect on the microcontroller	Zach
	Successfully detect a single hand gesture and be able to output information to a breadboard	Sarthak

April 1	Hand gestures work	Sarthak
	Finish LCD code	Sergio
	Integrate SD card & speaker	Zach
April 8	Integrate software with microcontroller (delayed)	Sarthak
	Begin 3D Design Enclosure	Sarthak
	Began soldering parts but ran into issue with soldering the microcontroller. Had to order a new microcontroller.	Sergio & Zach
April 15	Had to order new audio parts due to not being salvageable from desoldering into a new PCB. Microcontroller arrived and got soldered onto the new PCB.	Sergio
	Managed to program microcontroller and connect Raspberry Pi to LCD breakout board. Still waiting for new audio parts.	Everyone
	Print 3D enclosure	Sarthak
April 22	Tested Power Subsystem, Gesture Control Subsystem, and UI Subsystem the day prior to final demo. Audio parts did not arrive until the day of the final demo.	Everyone
April 29	Final Report Work	Everyone

6. Conclusion

6.1 Accomplishments

In the end, the STM32H7 microcontroller was able to be programmed to interact with the LCD and Raspberry Pi. This accomplishment helped show that the Gesture Control subsystem works based on what is being displayed by the LCD. In addition, power was able to be delivered properly throughout the main PCB, Raspberry Pi, and LCD breakout board. Moreover, there were no shorts or improper soldering that impacted the functionality of the final project design such as a burnt out component.

6.2 Uncertainties

One primary challenge in this project was the delayed arrival of components and frequent shipping setbacks. Often, the components we received were either incorrect or damaged. Additionally, we encountered issues with components being damaged during soldering; this was exacerbated by the unexpected small size of many components, such as the STM32 pins, which measured approximately 0.49mm each across a 100-pin chip. We also overlooked the PCB size, requiring significant adjustments to accommodate all components within the required dimensions. Toward the project's conclusion, multiple ST-Links failed, further reducing our time to debug the UI subsystem.

6.3 Ethical considerations

IEEE ethics code II.7 emphasizes treating all individuals equitably and avoiding discrimination based on race, religion, gender, and other characteristics. This project could aid individuals with disabilities, potentially making the technology essential for audio-related tasks. To avoid exploiting this necessity with high prices, similar to certain prescription drugs, we commit to setting fair prices if we commercialize this product.

Secondly, IEEE ethics code I.1 prioritizes public safety, health, and welfare, emphasizing ethical design and privacy protection. Our use of a camera raises privacy concerns. To mitigate risks, we will not connect the device to the internet, eliminating the possibility of cyberattacks. All camera processing will be localized, ensuring privacy.

Lastly, IEEE ethics II mandates fair treatment and non-discrimination, while emphasizing harm prevention. Our product includes a speaker that could potentially damage hearing if audio is too loud or malicious. We will restrict audio playback to pre-approved files and use a speaker incapable of producing harmful sound levels to prevent any hearing damage.

6.4 Future work

The first set of future work for this project is fixing the subsystems which were not operational at the time of demo. This was mainly the audio subsystem and getting it working on the PCB would be the last step until we completed all of the requirements initially set for this project.

Once that is completed there are a few ideas to build on the current implementation

6.4.1 Custom Neural Net

Currently, we are using a pre-trained model from Google's mediapipe to get joint data from a jpeg. It works for our cases, but if we trained our own model which was only meant to detect gestures, then its accuracy and speed would increase with enough training data. We could create a simple convolutional neural network and feed it images of gestures and the correct label.

6.4.2 Bluetooth Audio

Currently, the quality of the audio effects are limited by the quality of the speaker soldered to our PCB. If the speaker isn't the best quality many may not be able to hear the nuances of different effects. Creating a bluetooth audio adapter would allow for us to send raw wav files and leverage superior hardware to hear more detail in the audio effects.

6.4.3 Custom Gesture Mapping

Currently, we hard code the audio effect that corresponds to each hand gesture. However, if this was a product we would sell to potential musical artists, they should have the ability to customize different gestures and what audio effect it corresponds to.

Appendix A - Physical Enclosure

Visual Aid

On a high level, the final product would contain a camera, a speaker, and the electronics in between. The camera would be monitoring the hands of someone who is in view of the camera. In this example let's say an effect “x” is triggered by a thumbs up. In a standby state, we will be playing some audio files out of the speaker with no effects.

Once the camera detects a thumbs up it will send the proper signal to our PCB which will then pass all the audio signals through a system which will apply the “x”. That effect will remain on until another gesture is shown in the camera & there will be a gesture for “normal”.

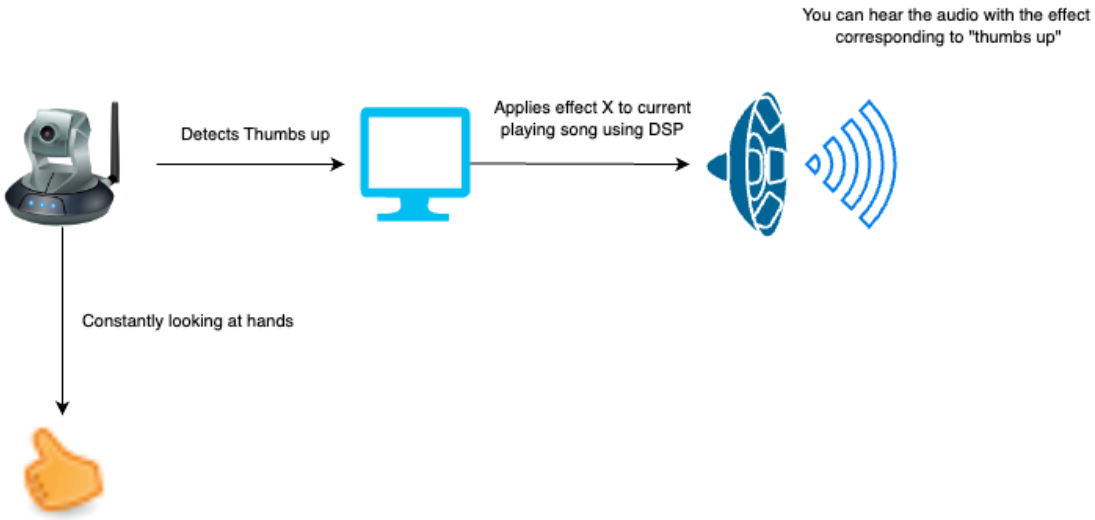

Figure 14a High-Level Overview of the Hands Gesture Audio Effects System

In terms of how the physical design will look we suspect there to be a camera viewing the subject and a speaker/display facing in the same direction all inside of one box. In section 2.2 we will expand on this idea.

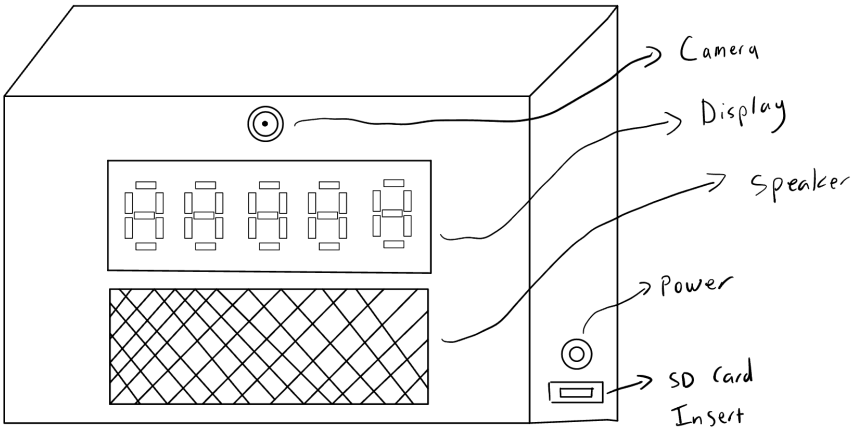

Figure 14b Physical design initial vision

Physical Design

The illustration of Figure 3 shows our physical enclosure appearance. Inside of this box, there will be a Raspberry Pi, PCB, and other components necessary for the function of the system. At the top, there will be a camera followed by a display unit, and then a speaker. On the right side of the enclosure, there will be a microSD card insert that will hold the audio file being modified by an audio effect. In addition, there is a DC power jack which will be connected to an outlet using a 5V 4.5A wall power adapter.

For the moment, there are no omitted dimensions on this diagram since there is no information about the dimensions of the custom PCB. Once there is a rough draft of the PCB board dimensions, then a decision can be made on how it will be mounted with the other components to the physical enclosure.

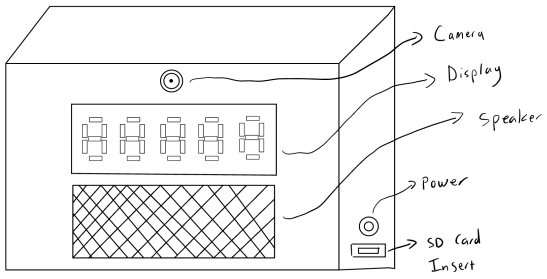

Figure 15. Initial vision for physical design

There was a design alteration for the actual physical enclosure appearance after looking at the dimensions of the LCD breakout board and main PCB with microcontroller. Figure 16 and Figure 17 shows the front and side view of the physical enclosure with the setup of the subsystems being connected with one another.

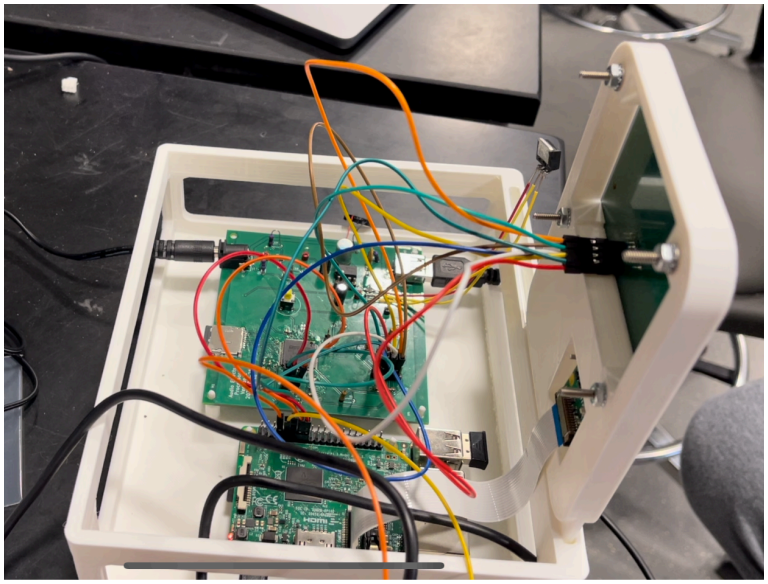

Figure 16: Side view of physical enclosure

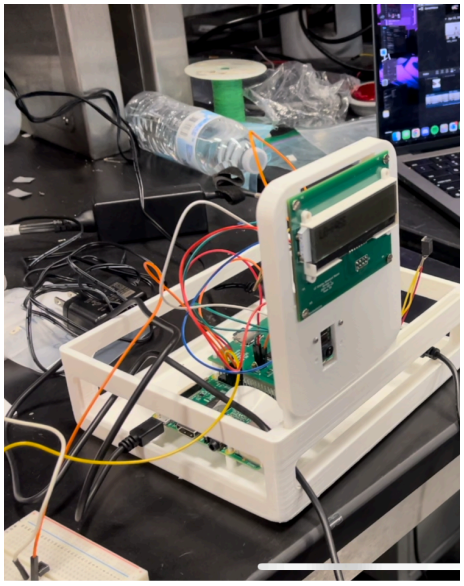

Figure 17: Front view of physical enclosure

Appendix B - Component List

UI Subsystem Component List:
1 x NHD-C0220BIZ-FSW-FBW-3V3M (Liquid Crystal Character Display Module)
Gesture Control Subsystem Component List:

1 x Raspberry Pi 3 Model B

1 x RPI-CAM-V2 (Camera module that detects hand gestures)

Power Subsystem Component List:

1 x LM1085ISX-3.3/NOPB (Linear regulator for 3.3 V power supply)

1 x WSU050-4000 (5 V 4.0 A wall power supply)

1 x PJ-102AH (DC Barrel Jack Connector for wall power supply)

1 x 87583-3010RPALF (USB-A connector soldered on custom PCB)

1 x USBAM11851M2MICBBK5A (USB-A to microUSB B Cable for Raspberry Pi Power)

1 x TPS259814ARPWR (eFuse for safety protection if user plugs in the incorrect power supply)

Audio System Component List:

1 x STM32H743VIT6 (Microcontroller)

1 x ECS-400-8-36B-7KY-TR (Crystal Oscillator)

1 x MSD-4-A (MicroSD Connector)

1 x S312TLKJM-C1000-3 (MicroSD 128 Mb)

1 x D6R90 F2 LFS (Push Button for MCU Reset)

1 x LM386N-1/NOPB (Amplifier for Audio DAC output)

1 x SPKM.15.8.A (Speaker to hear audio)

Detailed Cost of Parts Breakdown

Table 9. Project Parts Cost and Description

Manufacturer (Purchase link hyperlinked)	Part #	Quantity	Cost	Description
STMicroelectronics	STM32H743VIT6	1	$15.83	ARM® Cortex®-M7 STM32H7 Microcontroller IC 32-Bit Single-Core 480MHz 2MB (2M x 8) FLASH 100-LQFP (14x14)
CUI Devices	MSD-4-A	1	$0.35	9 (8 + 1) Position Card Connector microSD™ Surface Mount, Right Angle Gold
Delkin Devices, Inc.	S312TLKJM-C1000-3	1	$7.80	Memory Card microSD™ 128MB

				Class 10, UHS Class 1 SLC
Amphenol ICC (FCI)	87583-3010RPALF	1	$1.08	USB-A (USB TYPE-A) USB 2.0 Receptacle Connector 4 Position Surface Mount, Right Angle
Texas Instruments	LM1085ISX-3.3/NOPB	1	$1.99	Linear Voltage Regulator IC Positive Fixed 1 Output 3A TO-263 (DDPAK-3)
CUI Devices	PJ-102AH	1	$0.70	Power Barrel Connector Jack 2.00mm ID (0.079"), 5.50mm OD (0.217") Through Hole, Right Angle
Newhaven Display Intl	NHD-C0220BIZ-FSW-FBW-3V3M	1	$11.41	Character Display Module Transflective 5 x 8 Dots FSTN - Film Super-Twisted Nematic LED - White I2C 75.70mm x 27.10mm x 6.80mm
Triad Magnetics	WSU050-4000	1	$13.32	5V 20 W AC/DC External Wall Mount (Class II) Adapter Fixed Blade Input
C&K	D6R90 F2 LFS	1	$1.53	Pushbutton Switch SPST-NO Keyswitch Through Hole (0.1 A 3V)
SparkFun Electronics	PRT-12796	Bulk	$2.10	Jumper Wire Female to Female 6.00" (152.40mm) 28 AWG
Würth Elektronik	61300311121	Bulk	$0.13	Connector Header Through Hole 3 position 0.100" (2.54mm)
CNC Tech	3220-10-0300-00	1	$0.78	Connector Header Surface Mount 10 position 0.050" (1.27mm)
Seeed Technology Co., Ltd	114991786	1	$8.70	STM8, STM32 - Debugger (In-Circuit/In-System) SIPEED USB-JTAG/TTL RISC-V DEBUG
Raspberry Pi	SC0022	1	$35.00	Raspberry Pi 3 Model B Single Board Computer 1.2GHz 4 Core 1GB RAM ARM® Cortex®-A53, VideoCore
Raspberry Pi	RPI-CAM-V2	1	$35.00	Sony IMX219 image sensor custom designed add-on board for Raspberry Pi
GlobTek, Inc.	USBAM11851M2MICBBK5A	1	$6.08	Cable A Male to Micro B Male 3.94' (1.20m) Shielded
Texas Instruments	LM386N-1/NOPB	1	$1.28	Amplifier IC 1-Channel (Mono) Class AB 8-PDIP

Taoglas Limited	SPKM.15.8.A	1	$1.74	8 Ohms General Purpose Speaker 500 mW 10 Hz ~ 11 kHz Top Round
Texas Instruments	TPS259814ARPWR	1	$1.37	Electronic Fuse Regulator High-Side 10A 10-VQFN-HR (2x2)
Stackpole Electronics Inc	CF14JT10K0	10	$0.53	10 kOhms ±5% 0.25W, 1/4W Through Hole Resistor Axial Flame Retardant Coating, Safety Carbon Film
Cal-Chip Electronics, Inc.	GMC02X5R105M10NT	3	$0.30	1 µF ±20% 10V Ceramic Capacitor X5R 0201 (0603 Metric)
Samsung Electro-Mechanics	CL10A475KP8NNNC	2	$0.20	4.7 µF ±10% 10V Ceramic Capacitor X5R 0603 (1608 Metric)
Samsung Electro-Mechanics	CL10B225KP8NNNC	4	$0.40	2.2 µF ±10% 10V Ceramic Capacitor X7R 0603 (1608 Metric)
Samsung Electro-Mechanics	CL05B104KP5NNNC	10	$0.15	0.1 µF ±10% 10V Ceramic Capacitor X7R 0402 (1005 Metric)

Appendix C - Software Implementation of UI and Audio Subsystem.

UI Subsystem

The UI Subsystem code took advantage of the use of the HAL Master Transmit Function shown in Figure 18. Also, the I^2C communication that occurs when this function gets called is shown by Figure 19.

```
HAL_StatusTypeDef HAL_I2C_Master_Transmit(I2C_HandleTypeDef *hi2c,
                                          uint16_t DevAddress,
                                          uint8_t *pData,
                                          uint16_t Size,
                                          uint32_t Timeout);
```

Figure 18. HAL function called for I^2C Write Operation

Figure 19. I²C Write Operation Communication Based on called HAL function

A void function was created just to initialize the LCD before the iterative while loop starts. The init_LCD function shown by Figure 20 was translated from the flow chart shown in Figure 9. As stated before, the most important piece is adding delays between each instruction that gets sent to the LCD. The first 0x00 byte is just the control byte for each buffer and the next byte is the data byte representing an instruction. Also, there is an error handler that triggers whenever something goes wrong with the I²C communication such as no ack bit being sent to the microcontroller to confirm that the sent byte has been received properly by the LCD.

```
void init_LCD(void)
{
	HAL_Delay(100);

	uint8_t instruct1[2] = {0x00, 0x38};
	HAL_StatusTypeDef ret1;
	ret1 = HAL_I2C_Master_Transmit(&hi2c1, ST7036_I2C_ADDR, instruct1, 2, 100);
	if(ret1 != HAL_OK)
	{
		Error_Handler();
	}

	HAL_Delay(1);

	uint8_t instruct2[2] = {0x00, 0x39};
	HAL_StatusTypeDef ret2;
	ret2 = HAL_I2C_Master_Transmit(&hi2c1, ST7036_I2C_ADDR, instruct2, 2, 100);
	if(ret2 != HAL_OK)
	{
		Error_Handler();
	}

	HAL_Delay(1);

	uint8_t instruct3[2] = {0x00, 0x14};
	HAL_StatusTypeDef ret3;
	ret3 = HAL_I2C_Master_Transmit(&hi2c1, ST7036_I2C_ADDR, instruct3, 2, 100);
	if(ret3 != HAL_OK)
	{
		Error_Handler();
	}

	HAL_Delay(1);
```

Figure 20. init_LCD code snippet

The following code snippet described by Figure 21 is just an example of how one of the seven-character messages gets sent to the LCD, specifically the low-pass filter audio effect message. Basically, the first byte is the control byte that confirms data is being written into the data register of the LCD. The following byte is just the character code of the letter or dash character that is to be written to the LCD. Note that

stating a register address for the placement of the character is not necessary since there's a register address counter that increases by one for each HAL write operation call.

```
void Case1_LCD(void)
{
        uint8_t dataBuffer11[2] = {0x40, 0x4c}; //LO_PASS
        uint8_t dataBuffer12[2] = {0x40, 0x4f}; //LO_PASS
        uint8_t dataBuffer13[2] = {0x40, 0xb0}; //LO_PASS
        uint8_t dataBuffer14[2] = {0x40, 0x50}; //LO_PASS
        uint8_t dataBuffer15[2] = {0x40, 0x41}; //LO_PASS
        uint8_t dataBuffer16[2] = {0x40, 0x53}; //LO_PASS
        uint8_t dataBuffer17[2] = {0x40, 0x53}; //LO_PASS

    HAL_StatusTypeDef ret1;
        ret1 = HAL_I2C_Master_Transmit(&hi2c1, ST7036_I2C_ADDR, dataBuffer11, 2, 100);
        if(ret1 != HAL_OK)
        {
                Error_Handler();
        }

        HAL_Delay(1);

        HAL_StatusTypeDef ret2;
        ret2 = HAL_I2C_Master_Transmit(&hi2c1, ST7036_I2C_ADDR, dataBuffer12, 2, 100);
        if(ret2 != HAL_OK)
        {
                Error_Handler();
        }

        HAL_Delay(1);
```

Figure 21. Code snippet of Low Pass Filter Audio Effect Message Case

The next two figures (Figure 22 and Figure 23) shows the entire code within the while loop iteration. Figure 22 shows the three control state bits (from Raspberry Pi) being read and they get combined into a decimal number. This decimal number gets read by the switch-case block shown in Figure 23 and calls the respective void function to write the audio effect seven-character message defined by that decimal number (control state).

```c
while (1)
{
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
      STATE2 = HAL_GPIO_ReadPin(GPIOD, STATE2_Pin);
      STATE1 = HAL_GPIO_ReadPin(GPIOD, STATE1_Pin);
      STATE0 = HAL_GPIO_ReadPin(GPIOD, STATE0_Pin);
      sum = (STATE2 << 2) | (STATE1 << 1)  |  STATE0;

  ret = HAL_I2C_Master_Transmit(&hi2c1, ST7036_I2C_ADDR, dataInit, 2, 100);
      if(ret != HAL_OK)
      {
              Error_Handler();
      }

      HAL_Delay(1);
```

Figure 22. Three-Bit Control state being read

```c
switch (sum) {
    case 1:
         Case1_LCD();
         break;
    case 2:
         Case2_LCD();
         break;
    case 3:
         Case3_LCD();
         break;
    case 4:
         Case4_LCD();
         break;
    case 5:
         Case5_LCD();
         break;
    case 6:
         Case6_LCD();
         break;
    default:
         Case7_LCD();
         break;
}
```

Figure 23. Switch-case block that determines audio effect message to display to LCD

Audio Subsystem

The stm32cube code for the audio effects has the following functions: ApplyGain,ApplyReverb, applySoftClipping(distortion), PlayRegularAudio, ReadGesture, ReadAudioFile, and ProcessAudioEffects

```
void PlayRegularAudio(uint8_t *data, UINT size) {
    int numSamples = size / 2; // Assuming 16-bit audio samples
    uint16_t *samples = (uint16_t *)data;

    // Output directly to DAC or whatever output method is being used
    if (HAL_DAC_Start_DMA(&hdac1, DAC_CHANNEL_1, (uint32_t *)samples, numSamples, DAC_ALIGN_12B_R) != HAL_OK) {
        // Handle possible error
        Error_Handler();
    }
}
```

```
float ApplyGain(float sample, float gainFactor) {
    return sample * gainFactor;
}
```

```
float ApplyReverb(float inputSample) {
    // Calculate the read index
    int readIndex = reverbWriteIndex - REVERB_BUFFER_SIZE / 2;
    if (readIndex < 0) readIndex += REVERB_BUFFER_SIZE;

    // Read the delayed sample and apply feedback
    float delayedSample = reverbBuffer[readIndex] * REVERB_FEEDBACK;

    // Write the new sample to the buffer
    reverbBuffer[reverbWriteIndex++] = inputSample + delayedSample;

    // Wrap the write index
    if (reverbWriteIndex >= REVERB_BUFFER_SIZE) reverbWriteIndex = 0;

    // Mix the original and delayed samples
    return inputSample * (1.0f - REVERB_MIX) + delayedSample * REVERB_MIX;
}
```

```
float applySoftClipping(float sample) {
    float threshold = 0.8f;  // Clipping threshold
    if (sample > threshold) {
        sample = threshold + (sample - threshold) / 3;
    } else if (sample < -threshold) {
        sample = -threshold + (sample + threshold) / 3;
    }
    return sample;
}
```

```c
void ReadGesture(void) {
    uint8_t gesture = (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) << 2) | // Bit 2
                      (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1) << 1) | // Bit 1
                      HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_2);         // Bit 0
    // Reset all effects first
    useNothing = 0;  // Flag to bypass audio effects
    useLPF = 0;
    useHPF = 0;
    useGainUp = 0;
    useGainDown = 0;
    useReverb = 0;
    useDistortion = 0;

    switch(gesture) {
        case 0: // Closed Hand
            useNothing = 1;

            break;
        case 1: // Rock & Roll
            useLPF = 1;
            break;
        case 2: // Peace
            useHPF = 1;
            break;
        case 3: // Index Up
            useGainUp = 1;
            break;
        case 4: // All Fingers Up
            useGainDown = 1;
            break;
        case 5: // Pinky
            useReverb = 1;
            break;
        case 6: // Custom gesture for Distortion
            useDistortion = 1;
            break;
        case 7: // Another custom gesture (Reserved for future use)
            // Activate any other effect or combination here
            break;
        default:
            // Handle unexpected values if necessary
            break;
    }
    return gesture; // For debugging or further processing
}
```

```
void ReadAudioFile() {
    UINT br;  // Variable to store the number of bytes read
    BYTE buffer[2048];  // Buffer to store read data

    // Open the file with read access
    if(f_open(&MyFile, "audiofile.wav", FA_READ) != FR_OK) {
        // Error handling
        Error_Handler();
    }

    while(!f_eof(&MyFile)) {
        // Read data in blocks of 2048 bytes
        if(f_read(&MyFile, buffer, sizeof(buffer), &br) != FR_OK) {
            // Error handling
            break;
        }
        // Process and/or transmit data
        ProcessAudio(buffer, br);
    }

    // Close the file
    f_close(&MyFile);
}
```

```c
void ProcessAudioAndEffects(uint8_t *data, UINT size) {
    int numSamples = size / 4; // Assuming 16-bit stereo data
    int16_t *samples = (int16_t*)data;
    int16_t *pcmData = (int16_t *)data;
    static float l_buf_in[BLOCK_SIZE_FLOAT];
    static float r_buf_in[BLOCK_SIZE_FLOAT];
    static float l_buf_out[BLOCK_SIZE_FLOAT];
    static float r_buf_out[BLOCK_SIZE_FLOAT];
    // Convert 16-bit PCM to float
    for (int i = 0; i < numSamples; i += 2) {
        l_buf_in[i / 2] = (float)pcmData[i] / 32768.0f;
        r_buf_in[i / 2] = (float)pcmData[i + 1] / 32768.0f;
    }
    // Reset output buffers
    memset(l_buf_out, 0, sizeof(l_buf_out));
    memset(r_buf_out, 0, sizeof(r_buf_out));

    // Apply effects based on flags
    if (useNothing) {
        PlayRegularAudio(data, size);  // Just output the audio without effects
    }
    if (useLPF) {
        Process_LPF(l_buf_in, l_buf_out, numSamples / 2);
        Process_LPF(r_buf_in, r_buf_out, numSamples / 2);
    }
    if (useReverb) {
        for (int i = 0; i < numSamples / 2; i++) {
            l_buf_out[i] = ApplyReverb(l_buf_in[i]);
            r_buf_out[i] = ApplyReverb(r_buf_in[i]);
        }
    }
    if (useGainUp) {
        for (int i = 0; i < numSamples / 2; i++) {
            l_buf_out[i] = ApplyGain(l_buf_out[i], gainFactorUp);
            r_buf_out[i] = ApplyGain(r_buf_out[i], gainFactorUp);
        }
    }
    if (useGainDown) {
        for (int i = 0; i < numSamples / 2; i++) {
            l_buf_out[i] = ApplyGain(l_buf_out[i], gainFactorDown);
            r_buf_out[i] = ApplyGain(r_buf_out[i], gainFactorDown);
        }
    }
    if (useDistortion) {
        for (int i = 0; i < numSamples / 2; i++) {
            l_buf_out[i] = applySoftClipping(l_buf_in[i]);
            r_buf_out[i] = applySoftClipping(r_buf_in[i]);
        }
    }
    // Convert float back to 16-bit PCM
    for (int i = 0; i < numSamples; i += 2) {
        pcmData[i] = (int16_t)(l_buf_out[i / 2] * 32767.0f);
        pcmData[i + 1] = (int16_t)(r_buf_out[i / 2] * 32767.0f);
```

References

[1] *Motorola Semiconductor Data Manual,* Motorola Semiconductor Products, Inc., Phoenix, AZ, 2007.

[2] *Double Data Rate (DDR) SDRAM,* datasheet, Micron Technology, Inc., 2000. Available at: http://download.micron.com/pdf/datasheets/dram/ddr/512MBDDRx4x8x16.pdf

[3] Linx Technologies LT Series, web page. Available at: http://www.linxtechnologies.com/products/rf-modules/lt-series-transceiver-modules/. Accessed January 2012.

[4] J. A. Prufrock, *Lasers and Their Applications in Surface Science and Technology,* 2nd ed. New York, NY: McGraw-Hill, 2009.

[5] W. P. Mondragon, "Principles of coherent light sources: Coherent lasers and pulsed lasers," in *Lasers and Their Applications in Surface Science and Technology,* 2nd ed., J. A. Prufrock, Ed. New York, NY: McGraw-Hill, 2009, pp. 117-132.

[6] G. Liu, "TDM and TWDM de Bruijn nets and shuffle nets for optical communications," *IEEE Transactions on Computers*, vol. 59, no. 1, pp. 695-701, June 2011.

[7] S. Al Kuran, "The prospects for GaAs MESFET technology in dc–ac voltage conversion," in *Proceedings of the Fourteenth Annual Portable Design Conference*, 2010, pp. 137-142.

[8] K. E. Elliott and C. M. Greene, "A local adaptive protocol," Argonne National Laboratory, Argonne, IL, Tech. Rep. 916-1010-BB, 2006.

[9] J. Groeppelhaus, "Java 5.7 tutorial: Design of a full adder," class notes for ECE 290, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 2011.

[10] STMElectronics, "STM32H742xI/G STM32H743xI/G Datasheet",

https://www.st.com/resource/en/datasheet/stm32h743bi.pdf

[11] Amphenol ICC (FCI), "87583-3010RPALF (USB-A connector) Datasheet", https://www.amphenol-cs.com/media/wysiwyg/files/drawing/87583.pdf

[12] Newhaven Display Intl, "NHD-C0220BIZ-FSW-FBW-3V3M Datasheet", https://newhavendisplay.com/content/specs/NHD-C0220BiZ-FSW-FBW-3V3M.pdf

[13] Texas Instruments, "LM1085ISX-3.3/NOPB Datasheet", https://www.ti.com/lit/ds/symlink/lm1085.pdf?HQS=dis-dk-null-digikeymode-dsf-pf-null-wwe&ts=1711522636510&ref_url=https%253A%252F%252Fwww.ti.com%252Fgeneral%252Fdocs%252Fsuppproductinfo.tsp%253FdistId%253D10%2526gotoUrl%253Dhttps%253A%252F%252Fwww.ti.com%252Flit%252Fgpn%252Flm1085

[14] Texas Instruments, “TPS259814ARPWR Datasheet” , https://www.ti.com/lit/ds/symlink/tps25981.pdf?ts=1694724042849&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FTPS25981

[15] CUI Devices, “Micro SD Card Connector Datasheet”, https://www.cuidevices.com/product/resource/digikeypdf/msd-4-a.pdf

[16] Texas Instruments, “Amplifier Datasheet”, https://www.ti.com/general/docs/suppproductinfo.tsp?distId=10&gotoUrl=http%253A%252F%252Fwww.ti.com%252Flit%252Fgpn%252Flm386

[17] “SD Card.” *Wikipedia*, Wikimedia Foundation, 29 Apr. 2024, en.wikipedia.org/wiki/SD_card.

[18] Raspberry Pi, “Raspberry Pi 3 Model B Datasheet. ” [Online}. Available:

https://us.rs-online.com/m/d/4252b1ecd92888dbb9d8a39b536e7bf2.pdf