
OmniDrum

UCF Senior Design II - Final Report

Group 41 Members

Wylde Simpson - Electrical Engineering
Soufiane Louiba - Electrical Engineering
Jameson Palmer - Computer Engineering

Review Committee

Dr. Borowczak - ECE
Dr. Zhang - ECE
Robert Simpson - Leonardo DRS

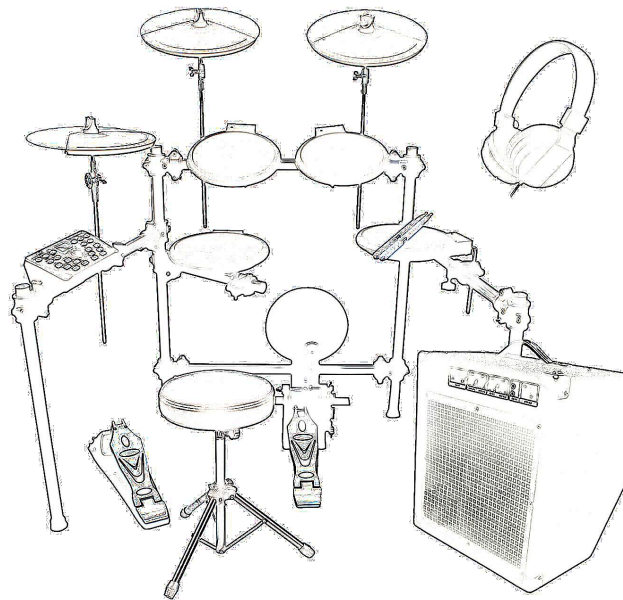


Figure 1.1 Project Mockup
by Jameson Palmer

Table of Contents

1. Project Narrative.....	1
1.1 Project Background & Context.....	1
1.2 Project Motivation.....	2
1.3 Project Functionality.....	2
2. Objectives & Requirements.....	3
2.1 Goals and Objectives.....	3
2.2 Requirements & Constraints.....	4
2.3 Block Diagrams.....	7
3. Research & Part Selection.....	9
3.1 Research.....	9
3.1.1 MCU.....	9
3.1.1.1 Purpose.....	9
3.1.1.2 External Storage Device.....	10
3.1.1.3 GPIO Pins.....	10
3.1.1.4 MIDI.....	10
3.1.1.5 ADC.....	10
3.1.1.6 DAC.....	11
3.1.2 Audio Output Device.....	11
3.1.2.1 Types of Output Devices.....	11
3.1.2.2 Frequency Response.....	13
3.1.2.3 Speaker Wattage and Impedance.....	14
3.1.3 Amplifier.....	15
3.1.3.1 Why Amplify?.....	15
3.1.3.2 How to Amplify.....	15
3.1.3.3 Low-Level & High-Level Signals.....	16
3.1.4 Auxiliary Output Jack.....	17
3.1.4.1 Audio Jack Form Factors.....	18
3.1.4.2 Tip, Ring, and Sleeve.....	20
3.1.4.3 Headphone Amplifier.....	22
3.1.4.4 Muting the Internal Speaker.....	22
3.1.5 MIDI.....	22
3.1.5.1 What is MIDI?.....	22
3.1.5.2 How does MIDI communication work?.....	23
3.1.5.3 What note values to use?.....	23
3.1.6 Types of Sensors.....	24
3.1.6.1 Applicable Sensors.....	24
3.1.6.2 Force Sensor.....	25
3.1.6.3 Velocity Sensor.....	25
3.1.6.4 Vibration Sensor.....	26

3.1.7 Sensor Design.....	26
3.1.7.1 Piezoelectric Sensors.....	26
3.1.7.2 Sensor Setup.....	28
3.1.7.3 Single-Zone Drum Pad.....	28
3.1.7.4 Dual-Zone Drum Pad.....	28
3.1.7.5 Triple-Zone Drum Pad.....	28
3.1.7.6 Sensor connection to the Control Board.....	29
3.1.8 Power Supply.....	30
3.1.8.1 Power Specifications (Voltage and Current).....	30
3.1.8.2 MSP430.....	30
3.1.8.3 Linear Voltage Regulator (TPS6212x 15-V, 75-mA Highly Efficient Buck Converter).....	31
3.1.8.4 Linear and Low Dropout (LDO) Regulator with reverse current protection.....	32
3.1.8.5 Arduino.....	34
3.1.8.6 AC Power Adapter.....	34
3.1.8.7 Internal Power Regulation.....	34
3.1.8.8 Protection and Safety.....	34
3.1.8.9 Energy Efficiency.....	34
3.1.8.10 Existing Product Examples.....	35
3.1.9 Software Languages.....	35
3.1.9.1 C.....	35
3.1.9.2 Java.....	36
3.1.10 Sound Library.....	37
3.1.10.1 Directory Structure.....	37
3.1.10.2 Filesystem.....	40
3.1.10.3 File Format.....	40
3.1.10.4 SPI (Serial Peripheral Interface).....	41
3.1.11 Controller Board.....	42
3.1.11.1 Instrument Assignment.....	42
3.1.11.2 Profile Selection.....	44
3.1.11.3 Strike Level Selection.....	45
3.1.11.4 Operating System.....	46
3.1.11.5 Control Panel.....	47
3.1.11.6 MIDI Data Output.....	51
3.2 Part Selection.....	52
3.2.1 MCU.....	53
3.2.2 Audio Output Device.....	55
3.2.3 Amplifier.....	56
3.2.4 Auxiliary Output Jack.....	57
3.2.5 Sensor.....	58

3.2.6 Power Supply.....	59
3.2.7 Development Environment: Languages and Repositories.....	61
3.2.8 Filesystem.....	62
3.2.9 Audio Format.....	63
4. Design Constraints & Standards.....	64
4.1 Industrial Standards.....	64
4.1.1 PCB Design Standards.....	64
4.1.2 SPI Standard.....	66
4.1.3 The MIDI Standard.....	67
4.2 Practical Constraints.....	69
4.2.1 Time.....	69
4.2.2 Economic.....	70
4.2.3 Manufacturability.....	71
4.2.4 Health & Safety.....	72
4.2.5 Ethical.....	72
4.2.6 Political.....	73
4.2.7 Environmental.....	73
4.2.8 Sustainability.....	74
5. Comparison of ChatGPT or Similar.....	74
Case Study 1:.....	74
Question “How can I optimize the power consumption of an electric drum?”	74
1 Efficient Circuit Design.....	75
2 Power Management Techniques.....	75
3 LED and Display Efficiency.....	75
4 Sensor Optimization.....	75
5 Firmware Efficiency.....	75
6 Testing and Measurement.....	76
7 Housing Design Impact.....	76
Case Study 2:.....	78
“What are the best microcontrollers for an electric drum project?”	78
1. ESP32.....	78
2. STM32 (e.g., STM32F4/F7 series).....	78
Why it's great:.....	78
3. Teensy 4.1.....	78
4. Raspberry Pi Pico.....	79
5. Nordic nRF52 Series.....	79
Case Study 3:.....	80
“What sensors are ideal for detecting drum hits accurately?”	80
1. Piezoelectric Sensors.....	80
2. Force-Sensitive Resistors (FSRs).....	80
3. Capacitive Touch Sensors.....	80

4. Laser or Optical Sensors.....	81
5. MEMS Accelerometers.....	81
Recommended Approach for OmniDrum.....	81
6. Hardware Design.....	82
6.1 Main Power Supply Circuit.....	82
6.2 STM32G4 Power Supply.....	83
6.3 Sensor Array.....	84
6.4 microSD IO.....	85
6.5 Profile Select IO.....	85
6.6 Pedal IO.....	87
6.7 MIDI Output.....	88
6.8 Speaker Amplifier.....	88
6.9 Headphone Amplifier.....	91
6.10 Overall Schematic.....	94
6.11 Structure & External IO.....	95
7. Software Design.....	97
7.1 External Storage Mounting and Parsing.....	102
7.2 Monitoring of the Sensor Array.....	103
7.3 Standby Mode, Idle.....	104
7.4 Runtime.....	104
7.4.1 System Startup.....	104
7.4.2 Profile Switching.....	105
7.4.3 Strike Level Detection.....	105
8. System Fabrication/Prototype Construction.....	108
8.1 PCB Design.....	108
8.2 Enclosure Design.....	110
8.3 Prototyping.....	110
9. System Testing & Evaluation.....	111
9.1 Hardware & Software Testing.....	111
9.2 Performance Evaluation.....	114
9.3 Overall Integration.....	115
9.4 Future Plans.....	116
10. Administrative Content.....	117
10.1 Budget Estimates.....	117
10.2 Parts List/BOM.....	118
10.3 Milestones.....	120
10.4 Work Distributions.....	120
11. Conclusion.....	121
Appendix.....	123

Figure 1.1 Project Mockup.....	1
Figure 2.1 House of Quality.....	6
Figure 2.2 Software Flowchart.....	13
Figure 2.3 Hardware Flowchart.....	14
Figure 3.1 Example of an Electrodynamic Speaker.....	18
Figure 3.2 Example of a Piezoelectric Buzzer.....	19
Figure 3.3 Raw FR L Graph for Audio-Technica ATH-M50x Headphones.....	20
Figure 3.4 Low-Level VS High-Level.....	22
Figure 3.5 Male & Female 3.5mm Audio Connectors.....	23
Figure 3.6 Male & Female ¼" Audio Connectors (male 3.5mm for reference).....	24
Figure 3.7 Male & Female XLR Audio Connectors.....	25
Figure 3.8 Male & Female RCA audio connectors.....	26
Figure 3.9 ¼" Audio Cable, Tip Ring Sleeve Example.....	26
Figure 3.10 Common MIDI Drum Values.....	30
Figure 3.11 System Flow Design.....	34
Figure 3.12 Circuit Design of Piezo Sensor to the Arduino MCU.....	37
Figure 3.13 TPS62120 and MSP430 Simplified Block Diagrams.....	39
Figure 3.14 Low Dropout Voltage Regulator (LDO) TPS709-Q1.....	40
Figure 3.15 Control Interface and Display.....	55
Figure 3.16 Top Menu Display.....	56
Figure 3.17 Input Port Status Monitor.....	56
Figure 3.18 Input Port Assignment Submenu.....	57
Figure 6.1 Main Power Supply Circuit Schematic.....	90
Figure 6.2 STM32G4 Schematic.....	90
Figure 6.3 Sensor Array Schematic.....	91
Figure 6.4 microSD IO Schematic.....	92
Figure 6.5 Profile Select Schematic.....	93
Figure 6.6 Pedal IO Schematic.....	94
Figure 6.7 MIDI Output Schematic.....	95
Figure 6.8 Speaker Amplifier Schematic.....	96
Figure 6.9 Headphone Amplifier Schematic.....	98
Figure 6.10 Switched Jack Functionality.....	100
Figure 6.11 Overall Schematic.....	102
Figure 6.12 Top View of the Device.....	102
Figure 6.13 Front View of the Device.....	103
Figure 6.14 Rear View of the Device.....	104
Figure 7.1 Software Use Case Diagram.....	106
Figure 7.1a Software Tasks.....	108

Figure 7.1b Software Tasks.....	109
Figure 7.2 Procedure to Collect Profiles from External Storage.....	110
Figure 7.3 Procedure to Collect and Use Sensor Readings.....	111
Figure 7.4.1 ADC sensor spike (centered).....	114
Figure 7.4.2 ADC sensor spike (leading, trailing).....	115
Figure 8.1 Tentative PCB Design.....	116
Figure 8.2 Final PCB Design.....	117
Figure 9.1 Prototype Enclosure.....	119
Figure 9.2 Prototype Enclosure Rear IO.....	120
Figure 9.3 Prototype Enclosure Top View.....	121
Figure 9.4 Force Sensor Peripherals.....	121
Table 2.1 Requirements & Constraints.....	4
2.2.1 Software.....	4
Table 3.1 Sensor set up for different types of drum pads.....	28
Table 3.2 MSP460x2xx family power consumption.....	32
Table 3.3 Power Consumption of Different Arduino Microcontrollers.....	33
Table 3.1 Comparison of MCUs.....	53
Table 3.2 Comparison of Audio Output Devices.....	55
Table 3.3 Comparison of Audio Speakers.....	56
Table 3.4 Comparison of Output Jacks.....	58
Table 3.5 Comparison of Sensors.....	59
Table 3.6 Comparison of Power Supplies.....	60
Table 3.7 Comparison of Filesystems.....	63
Table 3.8 Comparison of Audio Formats.....	63
Table 10.1 Estimated Budget.....	117
Table 10.2 Categorized BOM.....	118
Table 10.3 Condensed BOM.....	118
Table 10.4 Milestone Schedule.....	120
Table 10.5 Work Distributions.....	120

1. Project Narrative

In this section we discuss the context and intention of our project. This includes what our project is, what problem we are trying to solve with it, as well as how it works and what functionalities it provides. This acts as a short intro to our project.

1.1 Project Background & Context

Electric drum kits have been commercially available since the 1980s and are favored by many musicians for several reasons. These kits consist of various pads that mimic the shape and size of traditional drums. The pads are sensitive to being struck with a drumstick and send signals to a central unit that plays a drum sound based on which component is hit. Additionally, these pads can detect the velocity of the strike, allowing them to adjust the dynamics of the sound to match the player's style.

Musicians choose electric drum kits for various reasons. One significant advantage is their quieter operation compared to conventional drums, which is beneficial for practice in settings like apartment buildings or other locations where loud sounds may not be acceptable. Moreover, electric drum kits offer versatility; they can produce a wide range of sounds, often including a variety of built-in sounds that allow users to experience different types of drums without needing to own each one individually.

Traditional drum kits can consist of instruments that wear out with use or can be seriously damaged if mishandled or transported improperly. The financial cost associated with these instruments can be substantial, especially if any components get damaged. Digitizing the sounds of each instrument not only allows for playback of recordings but also ensures that these sounds can be preserved over the years without worrying about sound degradation, unlike their acoustic counterparts. The cost of replacing damaged components in an electric drum kit is generally lower due to the materials used and the modularity of the design.

Furthermore, electric drum kits are advantageous for recording and composition purposes. Recording a conventional drum kit can be complicated and expensive, often requiring multiple high-quality microphones, mixing equipment, and potentially even acoustic treatment for the recording space. These costs can quickly add up, often exceeding the budget of many players. In contrast, electric drum kits can be plugged directly into recording equipment, simplifying the setup and reducing the entry cost for recording drums.

Overall, there are numerous reasons why musicians may prefer to purchase an electric drum kit over a conventional one.

1.2 Project Motivation

Quality electric drum kits can be quite expensive, leading some companies to develop budget-friendly alternatives. While these budget kits come at a lower price compared to high-quality models, they can still represent a significant investment for young musicians, often lacking in quality and functionality. Our goal is to create an electric drum kit that is affordable enough for those on a tight budget, yet still meets the performance standards required for amateur musicians.

Many less experienced musicians are younger individuals who may not have the financial means to purchase expensive equipment. Traditional instruments can also be bulky and heavy, making them difficult to transport and store. Our project will produce a kit with a smaller footprint, enhancing accessibility for players who may struggle to find space for practice.

As music production software becomes increasingly available, more young and inexperienced musicians are gaining access to the tools needed to create their own music. Programs like Digital Audio Workstations (DAWs) enable beginners to produce professional-sounding tracks with built-in tools and recording functionalities. However, despite the decreasing costs of music production tools, live drum recording options remain less accessible for beginners due to budget limitations. Many aspiring musicians rely on sequencers within their DAWs, which typically lack the capability to create complex and engaging rhythms.

Our project aimed to provide an option that not only allowed beginners to develop their drumming skills but also offered a tool that integrated seamlessly with music production software. This empowered players to incorporate their own drumming into the music they wished to create.

1.3 Project Functionality

Users of OmniDrum can attach various sensors to different surfaces of their choice, corresponding to specific components of the drum set. These sensors detect vibrations and send signals to a central hub, which includes a power supply board, a microcontroller for processing information, a speaker for standalone practice, an amplifier board for the speaker, an optional headphone port, and a MIDI Out port.

The central hub connects to each of the sensors and plays sounds through the speaker based on which sensors receive input. The sensors also detect the magnitude of the vibrations, allowing the hub to adjust the volume and sound according to the intensity of the strike. Users can choose from a variety of built-in libraries featuring different styles of drum kits to customize the sounds to their preferences. By default, drum sounds play through an onboard speaker, which includes a volume adjustment knob. This design permits the device to function as a standalone drum kit, enabling users to practice without needing a separate computer. Additionally, a jack is available for connecting headphones,

allowing users to practice quietly or listen through their own device. When the headphone jack is in use, the onboard speaker mutes. The headphone jack can also connect the OmniDrum to an audio interface for sound recording.

Another key feature of the device is its ability to output MIDI information based on the player's inputs. MIDI (Musical Instrument Digital Interface) is a widely used communication standard that allows musical instruments to interact with computers and other devices. MIDI transmits information about which notes are pressed, the velocity of those notes, and other parameter adjustments. Most common DAWs (Digital Audio Workstations) rely heavily on the MIDI standard to create a seamless bridge between hardware and software. Specific MIDI note values represent different drum hits and sounds. The OmniDrum communicates which drums are played and their corresponding velocity to a DAW, enabling MIDI information to be recorded. It is important to note that while MIDI transmits note values and velocity, it does not transmit audio. This feature allows users to select any drum sound available on their computer and assign it to each MIDI note transmitted from the OmniDrum, enhancing its versatility. The MIDI capability enables the OmniDrum to integrate into the workflow of amateur musicians, helping them produce complex and engaging drum rhythms that can be directly incorporated into their music. This functionality is particularly beneficial for traditional drummers, as it allows them to express their ideas more naturally and effectively than by using a sequencer with limited capabilities.

2. Objectives & Requirements

The design of this project focused on an array of sensors that represented instruments. Responsiveness was a major goal, as each sensor was monitored to retrieve its values. The array was designed so that each sensor could be acted upon immediately and independently. The resulting behavior appeared to have a parallelized connection to each of the sensors. Multiple instruments were expected to operate simultaneously, and their outputs wove together in a concurrent playback. The time between an instrument being struck and the resulting output needed to have minimal delay. This project established a design that strived to minimize delay while providing simultaneous input and output operations..

2.1 Goals and Objectives

This section discusses in further detail what specifically we have attempted to achieve with our project. Three categories will cover the different goals we set, grouped by their complexity and feasibility.

Basic Goals:

1. The device will be able to detect input from sensors and play a corresponding sound
2. The device should be able to play at least 3 sounds at once

3. The device should be able to output MIDI signals corresponding to which sensors are triggered
4. Sensors should be able to be placed on most hard surfaces and detect a hit

Advanced Goals:

1. The device should be able to detect the velocity of each hit and play a different sound depending on the velocity
2. Sensitivity of the sensors should be adjustable through a potentiometer
3. The device should have a headphones/line out jack, which mutes the onboard speaker when in use
4. Integrate an amplifier for headphone usage listed above. The amplifier may be disabled as to not interfere when connected to another sound system to not add unnecessary gain.
5. The device should have multiple sets of built-in sounds

Stretch Goals:

1. Stylization such as waveform or frequency spectrum display
2. Modularity such that an instrument may be added/removed with ease.
3. **Bump/crosstalk detection** to prevent sensors from picking up vibrations on the frame from accidental knocks or another struck sensor.
4. **Demo mode** where sample drum routines could play on their own while lighting up the instrument that is played. A user may follow along the routine by striking an instrument when it flashes.

2.2 Requirements & Constraints

This section outlines the essential requirements and constraints for the project, ensuring that the system meets the desired specifications.

Requirement & Constraints	Description
Polyphony	Must be able to play at least 3 sounds at once
MIDI Output	Must output correct MIDI information
Vel. Sensitivity	Must be able to detect 3 levels of velocity
Detection Rate	Must be able to detect hits that are at least 100ms apart
Low Latency	Latency must be no more than 100ms
Low Power Draw	Target power for circuit board should be less than 15 Watts

Audible Output	External speaker should achieve at least 65 dBA
----------------	---

Table 2.1 Requirements & Constraints

2.2.1 Software

Objectives for software design are to implement data storage management. An external storage device is used to contain data for the project. Implementation for accessing external storage is a basic need for the purpose of this device. Software will take files from this storage and use the data for following processes. We must devise the steps necessary for communicating from the microcontroller to the external device. The software support of filesystems and file directory structure is an inherent component to the design.

Content from storage includes the audio assets that will play through the device. Knowledge and tradeoffs of audio codecs needs to be found to determine which codec fits the platform we are designing for. The embedded system will manage multiple files simultaneously along the process of decoding. They will each take CPU and memory resources away from the rest of the system. The real-time behavior of the system must be developed in a way that does not conflict with decoding our audio resources. Concurrent playback is a goal for our output signals which adds complexity to task management.

Senior Design II:

The methods available to provide multiple sound effects to play at once were considered with two options available to us. The first option is to allow our three ADC outputs to take samples from the sound effects with each sound effect going to one of the three. This will output the analogue signal on the same output circuit where the analogue signals can constructively and destructively combine their waveforms to the output devices. This has the restriction of having only as many sound effects play as we have ADC outputs so the upper limit would be to have three sound effects play at once. The other alternative is to use software to mix our sound effects into one channel so only one ADC would be needed to convert the output data. This method can allow for more than three sound effects to be combined together and play but requires more memory allocation and CPU processing. These requirements also increase as more sound effects are playing at once and are capped at a reasonable maximum of eight sound effects to mix together at a time.

House of Quality Table:

The accompanying table summarizes these requirements, indicating the desired performance levels for each criterion. The symbols in the table denote the following:

- + indicates an acceptable or positive outcome.
- - indicates an unacceptable or negative outcome.
- ↑↑ signifies a high or improved performance level.
- ↓↓ signifies a low or reduced performance level.

Each row reflects different aspects of the performance criteria, helping to identify areas of strength and those needing improvement. This structured approach facilitates a clearer understanding of the project's engineering goals.

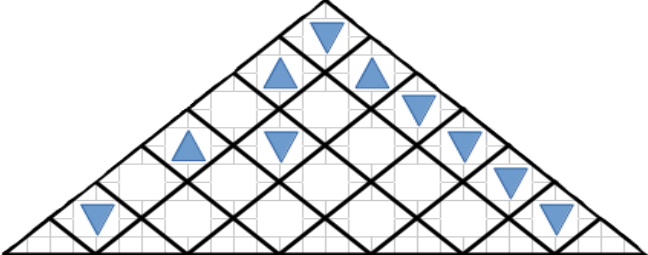
								
		Latency	Power	Sensitivity	Polyphony	Audible Output	Detection Rate	Cost
		-	-	+	+	+	+	-
Latency	-	↑↑	↓		↑↑		↑↑	↓
Power	-	↓	↑↑	↓	↓	↓↓	↓	↑
Audio Quality	+	↑	↓	↑	↑↑	↑		↓↓
Durability	+							↓
Cost	-	↓	↑	↑	↓	↓↓	↓	↑↑
Targets for Engineering Requirements		≤ 100 ms	≤ 15 Watts	> 4 levels	> 3 level mux	> 65 dBA	> 10 Hz	≤ \$115 prototype

Figure 2.1 House of Quality
by Jameson Palmer

2.3 Block Diagrams

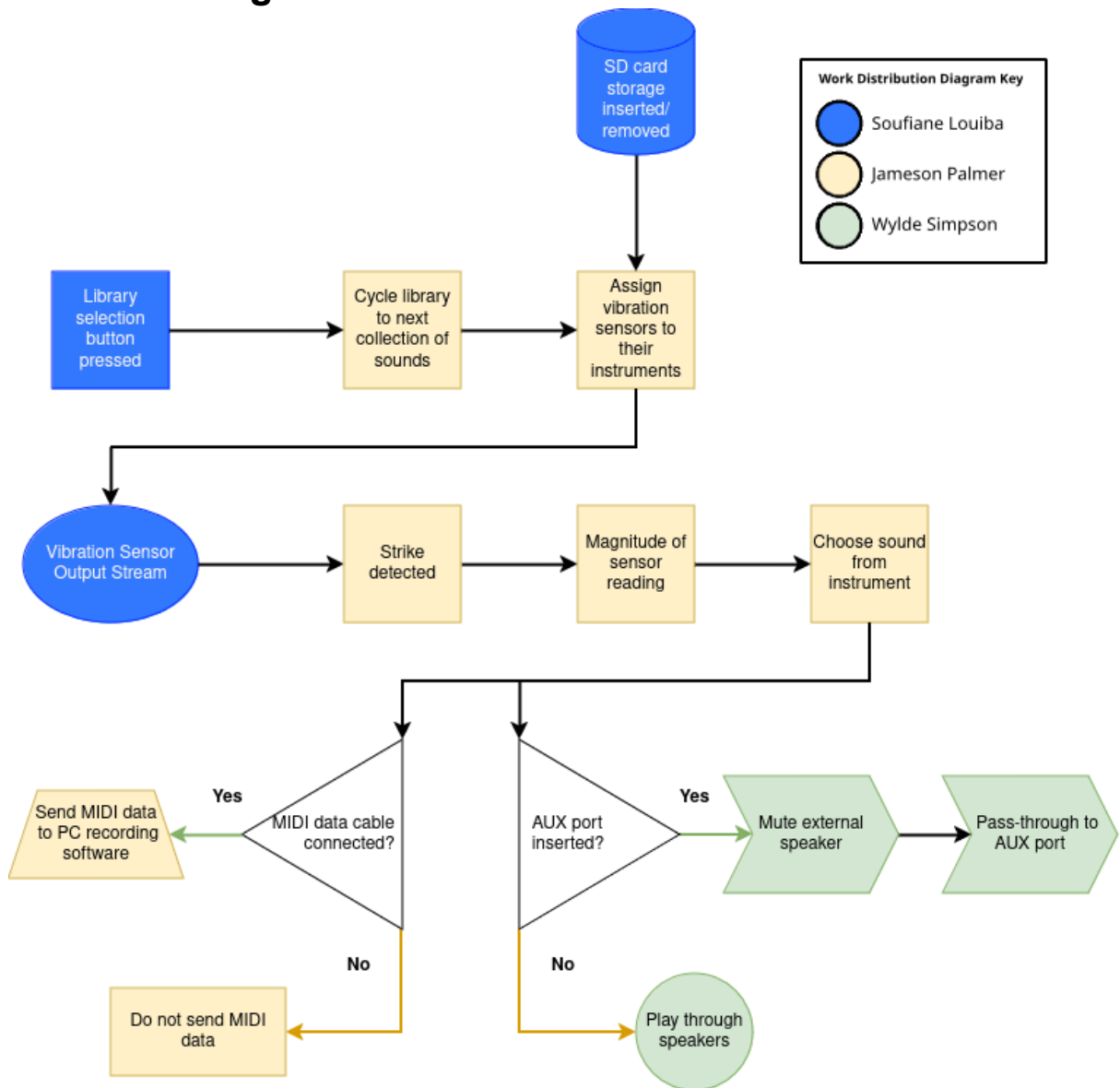


Figure 2.2 Software Flowchart
by Jameson Palmer

This diagram shows a simplified explanation of the software stages the device will follow. Software functionality is divided between these stages for better management while the code is developed. Some stages have intersection with other software functionality which helps to visualize how tasks should be arranged during operation. Determining task priority and stages of execution are detailed in their respective sections.

Hardware Diagram

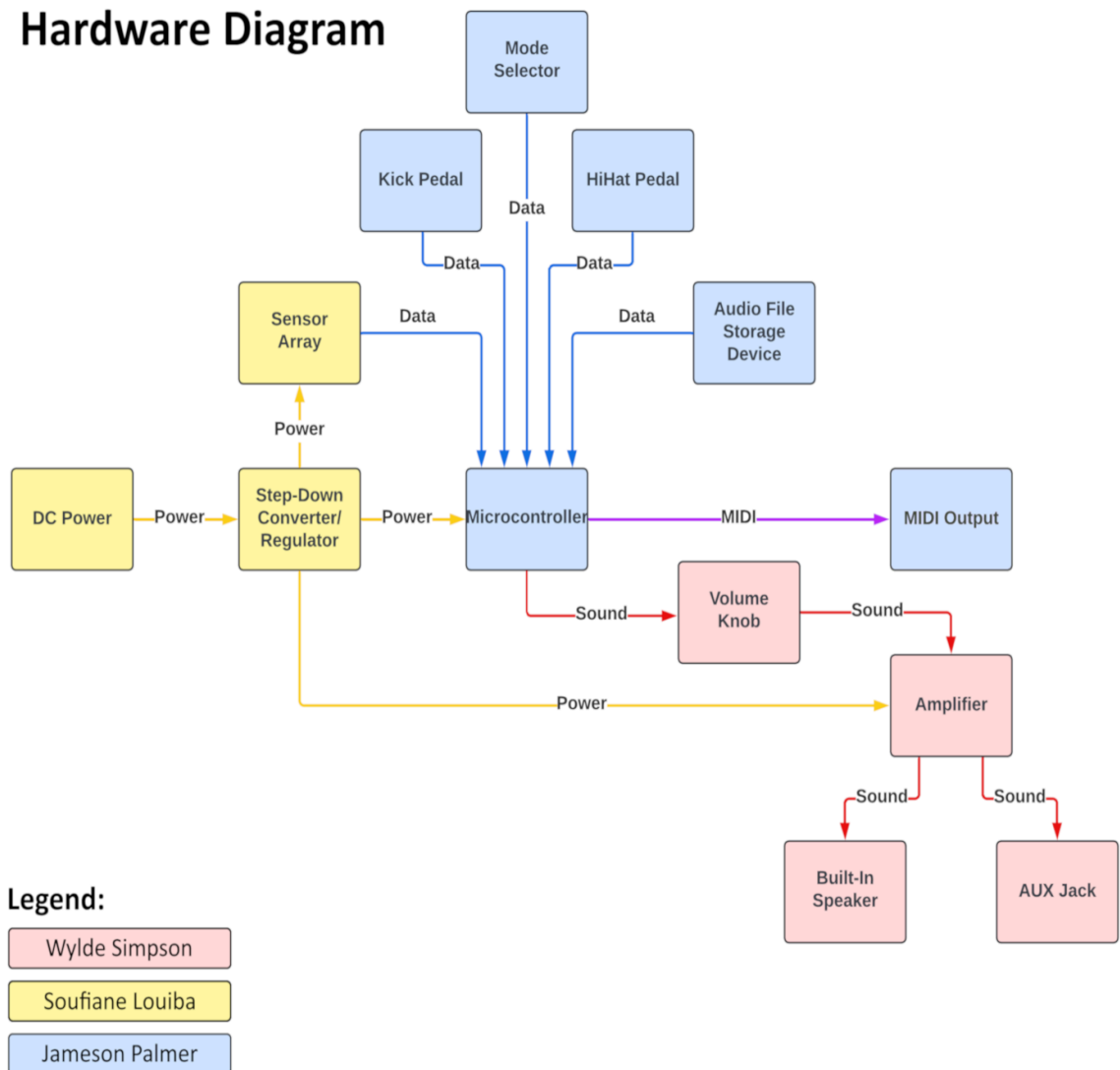


Figure 2.3 Hardware Flowchart
by Wylde Simpson

Some tasks necessitate collaboration among various team members to achieve successful completion. The tasks outlined here generally indicate individual responsibilities; however, it is important to note that several tasks will involve input from multiple team members. This collaborative approach ensures that the design functions cohesively and effectively.

3. Research & Part Selection

In this section, we detailed the research and selection process for the essential components of our drum kit project. We aimed to identify the most suitable microcontroller, audio output devices, amplifiers, sensors, and power supply options that would meet our project's functional requirements and constraints. By analyzing specifications, performance capabilities, and cost-effectiveness, we created a comprehensive system that delivered the desired user experience while ensuring reliability and efficiency. Each component's role was critical in achieving a well-rounded design that accommodated the interactive nature of the drum kit, allowing for dynamic audio playback and user input detection.

3.1 Research

The research phase of our project was focused on identifying the key components necessary for the successful implementation of our drum kit system. This involved exploring various microcontrollers, audio output devices, amplifiers, sensors, and power supply options. We aimed to gather detailed information on each component's specifications, capabilities, and compatibility with the overall design. By conducting this research, we laid the groundwork for informed decision-making in selecting parts that would achieve our project goals while ensuring optimal performance and user experience.

3.1.1 MCU

3.1.1.1 Purpose

For this project, we used a microcontroller to detect input from the user and play the desired audio files at the correct times on an audio output device. The microcontroller also handled the output of MIDI information, allowing the user to record the rhythms they played onto a third-party piece of software capable of receiving and storing MIDI data. To accomplish this, we needed to find a microcontroller that could read audio files from a mass storage device and then transmit that data as an analog signal from an internal digital-to-analog converter or transmit the file as a digital signal to an external digital-to-analog converter for amplification and playback.

Our microcontroller needed to output multiple audio files simultaneously. While playing, the user hit multiple components of the drum kit at the same time and required audio files to be able to play concurrently. The microcontroller reads these files from an external storage device, which we discussed in a later section. It also needed to cycle through different 'profiles' that served as separate collections of drum sounds for the user to choose from to play different types of drum kits.

MIDI and its role in the musical field will be discussed in a later section. Most microcontrollers should have been able to transmit MIDI data over a GPIO pin. MIDI is an asynchronous digital communication standard consisting of basic 8-bit strings used to transmit information about the notes being played on an instrument and their velocities, among other things. The MIDI standard was used ubiquitously within the world of digital music production..

3.1.1.2 External Storage Device

For our project to have worked correctly, we needed our MCU to be able to pull audio files from an external storage device. A common way of accomplishing this was by connecting our MCU to a microSD card. Using a microSD card allowed us to easily reorganize and alter the files on the card due to the interoperability between microcontrollers and personal computers that they offered. We were able to access the files on our own computers to troubleshoot and upload the sound files before placing the microSD card into the reader. Many microcontrollers had the ability to interface with microSD cards. There were different ways of doing this; the most common was by using SPI to allow for the microcontroller to communicate with the microSD card. Other forms of communication included SDIO and SDMMC. The microcontroller we chose needed to support these in some way..

3.1.1.3 GPIO Pins

In order for our project to work, our MCU needed to be able to receive several inputs and transmit many outputs. This meant that the chosen MCU required a sufficient number of GPIO pins. We needed around 6-8 input pins for our sensors, as well as additional pins for various buttons, LEDs, and audio output channels. This was achievable by most microcontrollers available on the market; however, in the next section, we discussed the specific capabilities required of these pins.

3.1.1.4 MIDI

After conducting some research, it appeared that supporting MIDI communication would actually be quite easy. MIDI communication was asynchronous at 31250 baud. Given that our microcontroller supported UART, we should have been able to easily set up MIDI communication. Most microcontrollers had UART support, so finding this should not have been a problem.

3.1.1.5 ADC

The sensors that we used for the project output analog voltage that needed to be converted into a digital signal to be understood by the microcontroller. To achieve this, we required ADCs on our microcontroller. We

needed at least as many ADCs as the inputs we used. Our project utilized at least 6 analog inputs and possibly more, depending on what needed to be done. In this case, we considered a chip with at least 8 ADCs and an absolute minimum of 6.

3.1.1.6 DAC

We needed to get our microcontroller to play sounds through a speaker. To drive the speaker, a separate amplifier circuit was constructed. However, this did not solve the issue of obtaining the audio signal from the microcontroller. For that, we needed to use a DAC. These could be either a separate IC used in conjunction with the microcontroller or built into the microcontroller itself. In fact, using multiple DACs was a way to implement simultaneous playback, as the DACs could be tied together. These DACs needed to be both external and buffered for proper use. The easiest implementation was to choose a microcontroller that had onboard DACs, preferably three.

3.1.2 Audio Output Device

This section discussed the various options we examined for our internal audio device. More specifically, it addressed the component to which we fed our amplified audio signal so that the user could hear sound coming from the device.

3.1.2.1 Types of Output Devices

The goal of this component of the system is to convert the electrical signal output from the MCU into auditory feedback for the user of the device. Converting an electrical signal into a mechanical sound can be accomplished using different types of devices including electrodynamic speakers, piezoelectric buzzers, or even outside audio devices like headphones or in-ear monitors.

Electrodynamic speakers are a well known method for turning electrical energy into acoustic energy through the use of electrodynamic transducers. These generally consist of three parts: the motor, diaphragm, and suspension. An electrical signal that has been appropriately amplified is applied to the motor which causes it to 'push' and 'pull' on the diaphragm. The purpose of the diaphragm is to help radiate and project the vibrations into the air in such a way that they are both loud enough and provide an accurate frequency response. The shape of the cone used has a significance on the frequency response of the output and thus the quality of the sound. The concept of frequency response curves is discussed in the next section. The final part of an electrodynamic transducer is the suspension, the purpose of which is to support the diaphragm and attach it to the rest of the speaker assembly while still allowing it to flex and vibrate in the desired fashion.



Figure 3.1 Example of an Electrodynamic Speaker

Image Source: Author

Piezoelectric buzzers function somewhat differently than electrodynamic speakers. To produce sound piezoelectric buzzers have a membrane that flexes based on the voltage applied to it. An advantage of piezoelectric buzzers is their low cost. They are relatively small and may require an added diaphragm to project their output. The most common use for piezoelectric buzzers is for alert tones and other applications that require only a single tone. They usually receive a digital signal in the form of a square wave at the desired frequency. These are not too useful for outputting analog audio with more harmonic content but they can be used for that purpose. Even though these buzzers generally received a 1-bit digital signal, pulse width modulation (PWM) could be used to modulate the input signal based on the frequency of the desired signal. This way, a 1-bit buzzer could be used to output varying frequencies. While 1-bit buzzers had this capability, the output is usually plagued with much harmonic distortion and has a very limited frequency range.



Figure 3.2 Example of a Piezoelectric Buzzer

Image Source:

<https://www.shutterstock.com/image-photo/electronic-piezo-buzzer-1101886475>

Other devices such as headphones and in-ear monitors are very similar in that they employ transducers except much closer to the ears of the users and at much quieter volumes. While we intended to allow users to utilize these devices in our final design, they were not provided to the users and were not meant to be the default listening method. We planned to use some form of loudspeaker so that the sound from the device would be audible to users within the room they were in; simultaneously, we intended to offer a connector jack for users to plug in their own peripheral audio devices if they chose to.

For our project, the best option was to use an electrodynamic speaker due to its preferable full-spectrum frequency response compared to the other options. It also provided us with the ability to fill the room with sound, allowing the user and others to hear what they were playing without needing to provide their own listening devices.

3.1.2.2 Frequency Response

The term "Frequency Response Curve" refers to the curve derived from relating the average amplitude of a signal and the frequency of that signal. A desirable frequency response is one that is as flat as possible such that it outputs all frequencies with the same average amplitude and in turn provides the most accurate recreation of the sound. That being said there are many cases where a flat frequency response curve may not be the most desirable to the user. While flat frequency responses are the most useful for accurate audio replication, recreational listeners often prefer a more 'V-shaped' curve that has an emphasis on lower bass frequencies as well as higher frequencies such as cymbals and

sibilance, along with less emphasis on mid-range frequencies. This is in part due to the fact that human ears do not have a flat response curve for which they pick up different frequencies and usually emphasize mid-range frequencies the most. In general, all people's ears have slightly different frequency response curves based on the physical structure of the outer ear, inner ear, and even head shape.

The image below shows an example of a frequency response curve measured from a certain pair of studio headphones. The target curve in this case is not 'flat', but represents what the source believes would be the most desirable response curve to compensate for the natural variations in sensitivity of human ears.

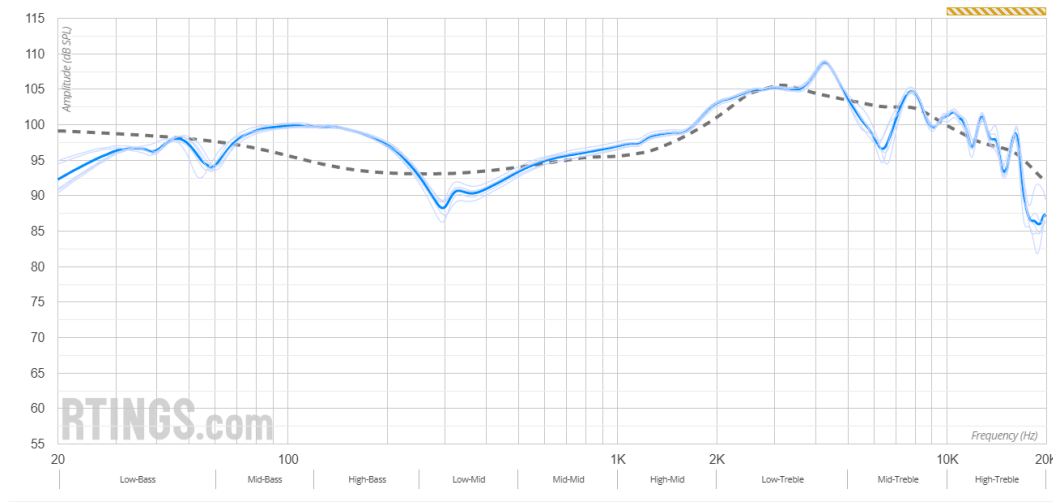


Figure 3.3 Raw FR L Graph for Audio-Technica ATH-M50x Headphones

Image Source:

<https://www.rtings.com/headphones/1-8/graph/25500/raw-fr-l/audio-technica-ath-m50x/295>

It should be noted that this description is a simplification of a very complicated topic. Which specific response curves sound good and in which application varies greatly between different situations, and acoustic measurements can be taken in a variety of ways. For our purposes, the frequency response curve allowed us to get a general idea of the sound quality of the type of speaker we chose. The goal was to use a speaker that provided decent replication of drum sounds so that the device could be used without the user needing to provide their own playback device. While an excellent frequency response and high sound quality were preferable, that was not the primary focus of the project.

3.1.2.3 Speaker Wattage and Impedance

When choosing a speaker for our project, there were specific specifications that needed to be considered. We needed to take into account the wattage of the speaker we wanted to use, as well as the impedance. The wattage of the speaker, while not solely responsible for the overall loudness,

served as a good estimate for how loud the speaker could be. The wattage listed on speakers was typically the peak power output, rather than the continuous power output. This was important for understanding how much power a speaker could be expected to use on average. Estimating the exact average power of a speaker was more challenging since it varied with the volume and harmonic content of the output, but generally, the RMS wattage was a fraction of the peak wattage. For our project, a speaker around 1-5 watts was deemed appropriate. Speakers within this range were capable of filling an entire room while requiring a modest amount of power. Although this was not an adequate amount of power for a live performance, users would be able to plug in an external playback device to the output port and amplify the sound further if needed. A power range of 1-5 watts was suitable for a practice setting.

The impedance rating of a speaker also has an effect on the sound output and the specifications of the amplifier that needed to be paired with it. Common impedance values for speakers were 4 Ω , 6 Ω , or 8 Ω , though there were additional possible values. Lower impedance was generally regarded as better, as it allowed the speaker to get louder with less distortion compared to higher impedance speakers; however, it could be more expensive since the amplifier needed to provide higher amounts of power. For our purposes, the impedance of the speaker did not matter too much as long as it could be matched with a proper amplifier. If the speaker fell within the power constraints of the project, it should have served the purpose adequately.

3.1.3 Amplifier

This section discussed the topic of amplification. It covered why an amplifier was needed, how amplification was achieved, and what possible devices could be used to amplify a signal for audio output devices.

3.1.3.1 *Why Amplify?*

In our design, the MCU pulled files from a microSD card and transmitted them to a Digital-to-Analog converter. After the digital signal was converted to analog, it needed to be amplified to an appropriate level for the speaker. The required amount of amplification that the circuit needed depended on the specifications of the speaker. The amplifier used could not be too powerful, or it might have damaged the speaker and possibly other parts of the circuit. Amplifiers needed to be matched to speakers based on both wattage and impedance. The exact specifications of the amplifier depended on the choice of speaker and would be covered in a later section.

3.1.3.2 *How to Amplify*

To amplify the signal coming from the DAC, we used an amplifier IC. There were many different types of amplifier ICs available, each with various functions, numbers of channels, amounts of gain, and output power. The choice

of chip was related to the output device selected. Typically, the ICs were not used standalone but required a small amount of circuitry around them to help stabilize the input signal and power supply. For our specific amplifier, we needed one mono channel along with a volume control potentiometer. The IC itself also needed to have built-in protections, such as short circuit protection. Additionally, we aimed for the amplifier to minimize total harmonic distortion to preserve sound quality.

3.1.3.3 Low-Level & High-Level Signals

Low-Level and High-Level were terms used to describe the strength and type of signals that passed through devices or cables in the world of audio. A low-level signal was the type of signal that came out of the Digital-to-Analog converter in circuits. Low-level signals were those that had not been amplified, such as signals from electric guitars, microphones, and unpowered mixers. On the other hand, high-level signals were those that had been amplified and were directed into loudspeakers or PA systems.

The distinction between these types of signals was important because plugging a low-level signal into a device expecting a high-level signal did not work due to the low-level signal being too weak to drive the device. Conversely, high-level signals going into a device that expected a low-level signal had the potential to damage the equipment because they were much more powerful than the device could handle.

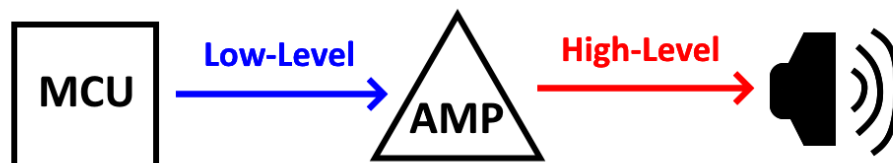


Figure 3.4 Low-Level VS High-Level

Image Source: Author

The term ‘high-level’ described a signal that had been amplified, but it was not specific about how powerful the signal was. A signal that came out of a 30-watt amplifier was much different than the signal that came out of a 1000-watt amplifier. These terms were context-dependent and had different meanings based on the situation. High-level in our project referred to a signal amplified to

drive a 3-watt speaker. In the context of car A/V, however, this meant the signal was amplified to drive 200-watt speakers.

3.1.4 Auxiliary Output Jack

For our project, we aimed to have a jack that the user could use to plug in headphones or another audio device to play sounds through a preferred device, either for privacy or performance. This section discussed different types of jacks, headphone amplification, and possible functions of the jack.

3.1.4.1 Audio Jack Form Factors

There were many different form factors for audio output jacks. We needed to decide on a specific one to mount to the main housing of the device. The purpose of the AUX jack was to provide the user with the ability to connect their own personal playback devices. These ranged from headphones to external speakers, and even included connections to a USB audio interface or other recording devices. This was especially useful if the user wanted to play the instrument quietly and without using the internal speaker, or in situations where there was too much ambient noise for the internal speaker. It was also beneficial if the user wanted to directly record what they were playing into a software program or another device.

Some examples of audio jack types included the well-known 3.5mm jack, the 1/4" jack, the XLR connector, and RCA cables. The 3.5mm jack, commonly referred to as the 'Headphone Jack' on many modern devices, was one of the most ubiquitous audio jacks found on consumer products. Some examples of the male 3.5mm connector were shown below, along with examples of the female 3.5mm jack. The 3.5mm jack could be found on computers, mobile devices, toys, instruments, and certain peripherals. Its widespread use made it a very accessible type of connector to use in our design, as most people had some type of listening device that incorporated a 3.5mm audio connection.

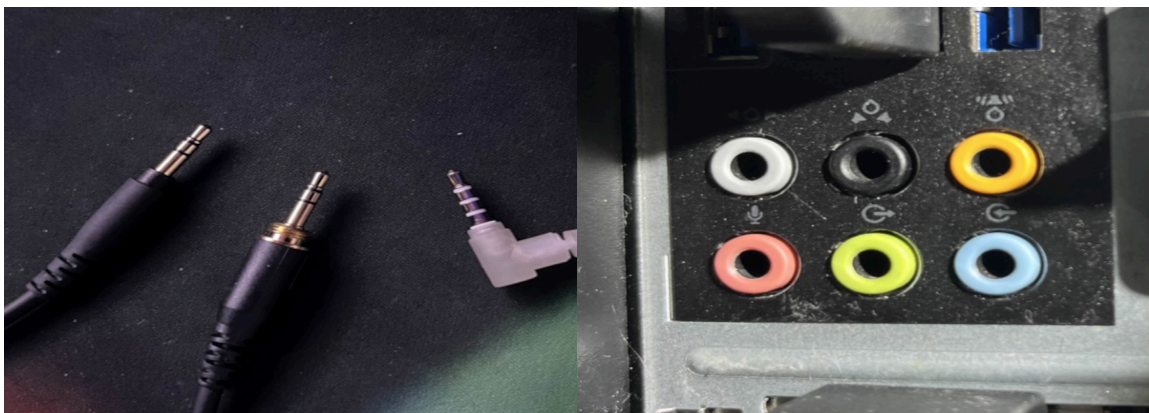


Figure 3.5 Male & Female 3.5mm Audio Connectors.

Image Source: Author

Another form factor was the 1/4" jack; cables for this kind of jack were colloquially referred to as "Guitar Cables," "Instrument Cables," or sometimes "Line Cables," as they were commonly used to carry high-level or 'Line' level signals. However, 1/4" cables were also very common for other uses within sound technology. The 1/4" jack was primarily prevalent in the sound/music space as it was used to connect guitars to amps, instruments to mixers, headphones to interfaces and monitoring equipment, as well as to patch signals in modular synthesizers and Effects Racks. The 1/4" connector and jack appeared as an oversized 3.5mm connector and jack, functioning in practically the same way. The larger jack was preferred in these situations due to its ruggedness and ability to handle higher currents than 3.5mm cables. Most studio headphones used for more professional audio work had this type of connector as they were expected to be plugged into a mixer or USB audio interface where 1/4" jacks were the norm. It was possible that a user of the drum kit would have been more likely to have listening devices that used this connector, assuming they had other audio equipment, but the 3.5mm connector could not be ruled out since the product was aimed at a low price point. A user buying equipment on a budget was more likely to be listening with simple headphones or earbuds. This consideration was important when choosing which connector type to implement into the design.



Figure 3.6 Male & Female 1/4" Audio Connectors (male 3.5mm for reference).
Image Source: Author

The XLR connector is a form factor commonly used in stage settings. This connector type most commonly contains 3 pins, though a 5 pin version does exist but is much less common in audio-related situations and is found more often in lighting control scenarios for transmitting DMX information. XLR audio cables are generally used for connecting microphones to other devices such as mixers or PA speakers. The use of pins for XLR audio transmission is very similar to that of balanced 3.5mm and 1/4" connectors such that there is a ground pin, a hot pin for carrying the audio signal, and a cold pin used for noise reduction. While XLR

audio connections are useful in the world of sound, they generally did not fit specific purposes and were unlikely to be useful.



Figure 3.7 Male & Female XLR Audio Connectors

Image Source: Author

The final form factor researched was the RCA connectors and jacks. RCA connectors had been around for a very long time and got their name from the Radio Corporation of America, which invented the design in the 1930s. The design consisted of a central protruding conductor surrounded by and separated from an outer barrel-like conductor. The central conductor carried the analog signal while the outer conductor acted as ground. These connectors had been used for many audio-related purposes in the past, such as connecting to radios, record players, and tape decks, as well as connecting TV sets and early home computers to external audio devices. They were also used to transfer video over both Composite video connections and later Component video connections before being succeeded by HDMI. The use of RCA cables had dissolved over the past few decades, aside from specialized uses today, such as Tape In/Tape Out recording on mixers and connecting car stereo systems to amplifiers and other DSP-related components.

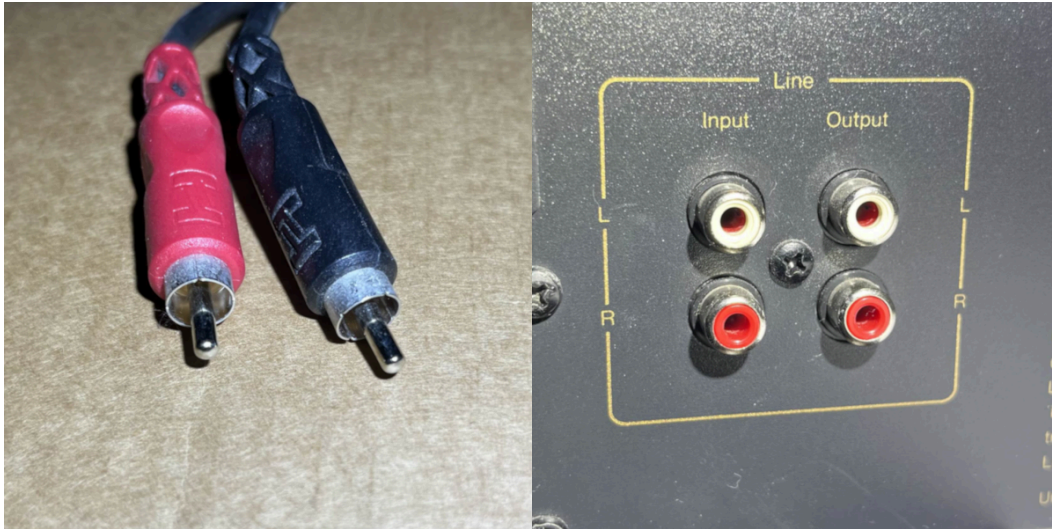


Figure 3.8 Male & Female RCA audio connectors

Image Source: Author

3.1.4.2 Tip, Ring, and Sleeve

It can be seen in **Figure 3.X** that the number of separate contacts on the male connectors of the 3.5mm and 1/4" jacks varies. These contact areas are referred to as the Tip, Ring, and Sleeve. 3.5mm and 1/4" connectors can be set up in different configurations such as TS, TRS, TRRS, and even the very uncommon TRRRS. The number of contact areas is representative of the number of individual wires running through the cable. The sleeve portion of the cable is dedicated to carrying ground. A cable set up in the TS configuration is used for carrying a mono signal, or a single channel of audio, where the tip carries the audio signal and the sleeve carries ground. This is the most basic configuration and lacks the advantages of more advanced configurations.

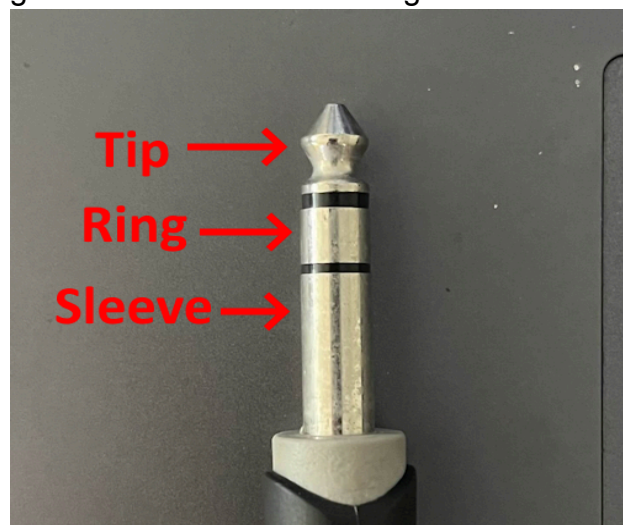


Figure 3.9 1/4" Audio Cable, Tip Ring Sleeve Example

Image Source: Author

Cables with the TRS configuration have three conductors and could be used for different purposes. One option is to use it for carrying a stereo signal, where the tip carries the left channel while the ring carries the right channel. Another use for TRS cables is carrying a single 'balanced' mono channel. In this case the tip carries the 'hot' audio signal while the ring carries a 'cold' neutral channel. As interference is picked up by cable, it has an effect on both the hot and cold conductors. At the other end of the cable, the polarity of the signal on the cold channel containing the isolated noise is inverted and the inverted noise is then added to the hot signal to phase cancel the noise in the hot signal. This technique is exceptionally effective at reducing interference noise in applications where the length of the cable is significant enough to pick up interference from the environment.

Further configurations such as the TRRS configuration are common on earbuds or headsets which include a microphone. In the case the tip carries the left audio channel, the ring adjacent to the tip carries the right audio channel, the ring adjacent to the sleeve carries the outgoing signal transmitted from the microphone, and the sleeve acts as ground. Further configurations such as TRRRS are very niche and usually only applicable in specialized applications. A cable with this configuration carries four conductors as well as ground and could be used to carry four unbalanced mono channels, two balanced mono channels, two pairs of stereo channels, or a number of other combinations of signals.

A useful feature of this form factor is that it allows for some intercompatibility between different cable configurations. For example, many computers contain a TRRS headphone jack as well as a separate microphone jack to account for the many different types of listening devices that could be connected. A pair of stereo headphones with a TRS connector could still be used normally even when plugged into the TRRS jack because the contact for the ring adjacent to the sleeve on the female connector would connect to the sleeve contact of the male connector on the headphones. This would appear to the computer as being shorted to ground when the device was plugged in, telling the computer that the listening device connected did not have a TRRS microphone connection, without transmitting any digital data between the devices. A mono listening device with a TS connector would function similarly in this scenario, playing only the left channel from the computer, which wasn't preferable but still allowed the listening device to function. However, it should be noted that certain combinations, such as this, could result in damage to devices if they were old or badly designed. Most devices today were designed to protect against possible damage from shorting certain connections, but not all devices were, so it was best to use the correct connectors when possible and when it was known that the combination would be accepted by the device.

3.1.4.3 Headphone Amplifier

For the auxiliary output to function correctly, it needed to be amplified to the required level to drive headphones. Headphones generally have much higher impedance than loudspeakers and require a different type of amplifier to drive them. This amplifier was a separate unit from the one used to drive the loudspeaker. It required a different integrated circuit (IC) to amplify the signal properly and had its own volume control.

3.1.4.4 Muting the Internal Speaker

A common feature of many musical devices was the ability to mute the external loudspeaker when a device was plugged into the auxiliary jack, without the need for the user to flip a switch. We aimed to incorporate this feature into our design to streamline the user experience. This could be achieved by using a 'switched' jack for the headphone output jack, which would ground or power a pin on the loudspeaker amplifier IC, shutting it off so that the audio would only go to the headphones.

3.1.5 MIDI

This section covers the MIDI communication standard and its function in the realm of music production. It also discusses how we implemented MIDI into our project to allow the user to record the rhythms that they play.

3.1.5.1 What is MIDI?

MIDI, which stands for Musical Instrument Digital Interface, is a communications standard that was invented in 1983 for use with synthesizers and other digital instruments. The MIDI standard allows for musicians to connect instruments to computers or other instruments in order to communicate what notes are being played, with what velocity, and other information such as Control Change codes and Program Change codes on different individual channels. Over the years MIDI has become extremely successful and is almost everywhere in the world of digital music.

In our specific case, we used MIDI to allow the user to communicate which drums they played with a computer program they owned. Digital audio workstations (or DAWs) supported communications with MIDI hosts. If the user chose to, they were able to connect to a DAW and record which drums they hit in real time so that they could play them back later or edit them.

It should be noted that MIDI does not transmit audio data; it transmits purely digital information. Different notes in the MIDI standard are communicated with values from 0 to 127. The reason MIDI was useful in our scenario was that it allowed the user to record when they played certain drums at specific times, enabling them to capture various rhythms they may have created. Each component of the drum set was assigned a 'note number,' allowing the computer

to display when each individual drum was played. The user could then assign a drum sound file on their computer to each 'note,' such that when they hit that specific drum, a chosen sound played on the computer. This allowed the user to endlessly customize the sounds of their entire drum kit, record it into the computer, and adjust the mixing and effects on the drum kit to eventually use it in a musical project alongside other instruments they may have recorded. The user had to provide their own computer and DAW to utilize this feature, as adding MIDI capability to our device only gave it the ability to connect to this kind of system in the first place.

3.1.5.2 How does MIDI communication work?

MIDI communication is done asynchronously at a baud rate of 31250 Hz. Information is sent in 8-bit packets containing note values, velocity values, and further information allowing a musician to change parameters on different devices remotely. MIDI cables generally consist of two 5-pin male ends which plug into female jacks at either end. The constraints posed by the requirement for MIDI capability are minimal and we should be able to easily find a microcontroller that supports this.

3.1.5.3 What note values to use?

MIDI note values range from 0 - 127 which gives us a lot of options for choosing which notes should correspond to each component of the drum kit. Luckily, the music production community has decided common note values for different types of drum sounds. Although these are not strictly the rule, many brand name products stick to these values as their ubiquity allows for widespread compatibility with other products. According to zendrum.com the commonly used MIDI drum values are as follows

35	Acoustic Bass Drum
36	Bass Drum 1
37	Side Stick
38	Acoustic Snare
39	Hand Clap
40	Electric Snare
41	Low Floor Tom
42	Closed Hi-Hat
43	High Floor Tom
44	Pedal Hi-Hat
45	Low Tom
46	Open Hi-Hat
47	Low-Mid Tom
48	Hi-Mid Tom
49	Crash Cymbal 1
50	High Tom
51	Ride Cymbal 1

Figure 3.10 Common MIDI Drum Values

Image Source: <https://www.zendrum.com/resource-site/drumnotes.htm>

These values encompass the different components of the drum set that we planned to implement for the user. This should allow the user to be able to easily use our drum kit with drum kit presets that they may have in their DAW, as well as customize the sounds of it to their liking, enabling more versatile use cases of the drum kit. It should be noted that our project did not incorporate all of the instruments listed here, and that the list of common MIDI drum note values continued beyond the scope of this list.

3.1.6 Types of Sensors

There were many types of sensors that could be used to detect when a user struck a drum pad. In this section, we covered the different types of sensors and their possible applications within our project to determine which sensor would be the best choice.

3.1.6.1 Applicable Sensors

The user interface for our drum kit consists of several modules, including the cymbal, hi-hat, snare, and bass. The designs for each module incorporate a sensor to detect a user's strike on the instrument. Translating a strike into digital communication with the mainboard is the responsibility of the most suitable

sensor. What can be detected from a strike on the instrument? The plane of each instrument will encounter a vibration. Acoustic instruments are fundamentally designed for vibration frequencies to create music, so a vibration sensor appeared to be a good candidate for our design. Each strike on a surface could also have its force measured. The pressure or force applied to the surface area could be connected to a force sensor capable of measuring various levels of force exerted.

Another alternative considered was connecting the strike surface to an actuator that could depress upon impact. With greater force applied to the instrument, the actuator would depress at a higher velocity. Measuring the velocity could then provide sufficient information to determine the force of a strike at various levels.

3.1.6.2 Force Sensor

A force sensor, also known as a pressure sensor, measures the amount of force applied to it. Many of these devices work by converting the mechanical energy applied to them into electrical energy. Some subtypes of this kind of device include strain gauges and piezoelectric sensors. Strain gauges are often made of metal foil wrapped by wire which encases a plastic or ceramic cylinder. Under stress the electrical changes with the deformation of its shape. For piezoelectric sensors, they are often made of a crystalline structure such as quartz. Ceramics can also be made in a molecular structure that has a piezoelectric characteristic. The piezoelectric effect is an electric charge generated by a force on this material which will bend the crystalline structure. The deviation of these molecules will shift positive and negative charge along the crystal. The amount of charge generated is proportional to the force applied to the material and this charge can be measured through an ADC. Measuring the charge through an ADC can then allow us to measure the amount of force. Both of the described sensors would need power supplied to them. Although the piezoelectric effect creates an electric signal, many of the available sensors require a power source for the added circuitry. This should be considered when wiring our instrument modules to the controller board as there should be wires for power as well as data.

3.1.6.3 Velocity Sensor

These devices can observe displacement over time. Some implementations of velocity sensors include optical sensors which measure light as it is deflected off of a target. Light can be directed back to a photodiode which generates electric current when subject to photons. A dedicated beam of light is integrated with this type of sensor. When in motion, the displacement changes the refraction of this light beam on the photodiode so a rate of change can be used to calculate the velocity of our target. For our purposes, a velocity sensor utilized a linear encoder that could detect velocity along one direction. When an instrument was struck, the sensor observed a piston as it was depressed inward.

The target of our sensor was the end of the piston. Our implementation required a low latency response; therefore, detecting velocity with a light beam minimized the time for a response from our instruments. Due to the need for an integrated light source, this type of sensor needed active power delivery to operate. The implementation of this device considered wires to dedicate a power source, as well as wires to receive measurement data.

3.1.6.4 Vibration Sensor

These sensors could be used to measure the shaking of a strike surface on our drum kit. These can implement piezoelectric sensing technology or accelerometers. The sensor could be attached to the strike surface which is our target to be measured. It would detect the changes of motion of the target over time. Piezoelectric materials would have stress placed on their shape to create a charge across the material. Accelerometer-based implementations could detect the rate of motion over time to measure a frequency of a strike. These accelerometers can also use the piezoelectric materials.

3.1.7 Sensor Design

This section goes into more detail about the possible applications of sensors within our project. It goes on to describe possible extensions to the functionality of our project that we have determined are possible through our research.

3.1.7.1 Piezoelectric Sensors

Piezoelectric sensors are one of the most important components of our project; these sensors constitute an end product combining functionality with cost-effectiveness. The term "Piezo" originates from the Greek word, meaning, press. This sensor consists of two defined zones on a 27mm-brass-disc diameter: the inner circle contains a slice of quartz crystal that generates a negative output voltage, while the outer circle generates a positive output voltage.

Piezoelectric occurs when a mechanical pressure is applied on the crystal; in our case, when the drum pad is struck with the drumsticks. The hit causes the piezo sensor to activate, producing an alternating current (AC) at the output, which runs through the cable to the module (or "brain") and triggers it to produce a sound, every sound in the OmniDrum has its own sample which is different from a pad to another, the user then will have the option to play the sound back through either speaker or headphones. The strength of the electrical signal is directly proportional to the force of the hit, making the sensor highly responsive to varying levels of pressure. A light tap produces a weaker signal, resulting in a softer sound, while a harder hit generates a stronger signal, leading to a louder sound. This makes the sensor highly responsive to different playing dynamics, letting the drummer control the sound's volume and intensity naturally.

One of the stand out reasons for choosing this sensor is its effectiveness. At less than \$7 for a pack of 50, it provides a high-performance, low-cost solution, making it ideal for our project. This combination of affordability and precision makes the Piezoelectric sensor an excellent choice for applications where sensing force and pressure are critical.

In a typical circuit with piezoelectric sensors, which generate small voltages when deformed, a pull-down resistor connected between the sensor's output and ground ensures that when the sensor is not generating voltage, the output is reliably pulled down to 0V (or the low logic level). This allows for cleaner and more consistent voltage readings when the sensor is triggered, reducing noise or false readings.

In our case, we used 100 Ohm resistors as pull-down resistors in parallel with the piezoelectric elements. The purpose of this was to stabilize the voltage readings from the piezo elements. Without the pull-down resistors, the output from the piezo sensors could float when inactive, which leads to unreliable or noisy readings..

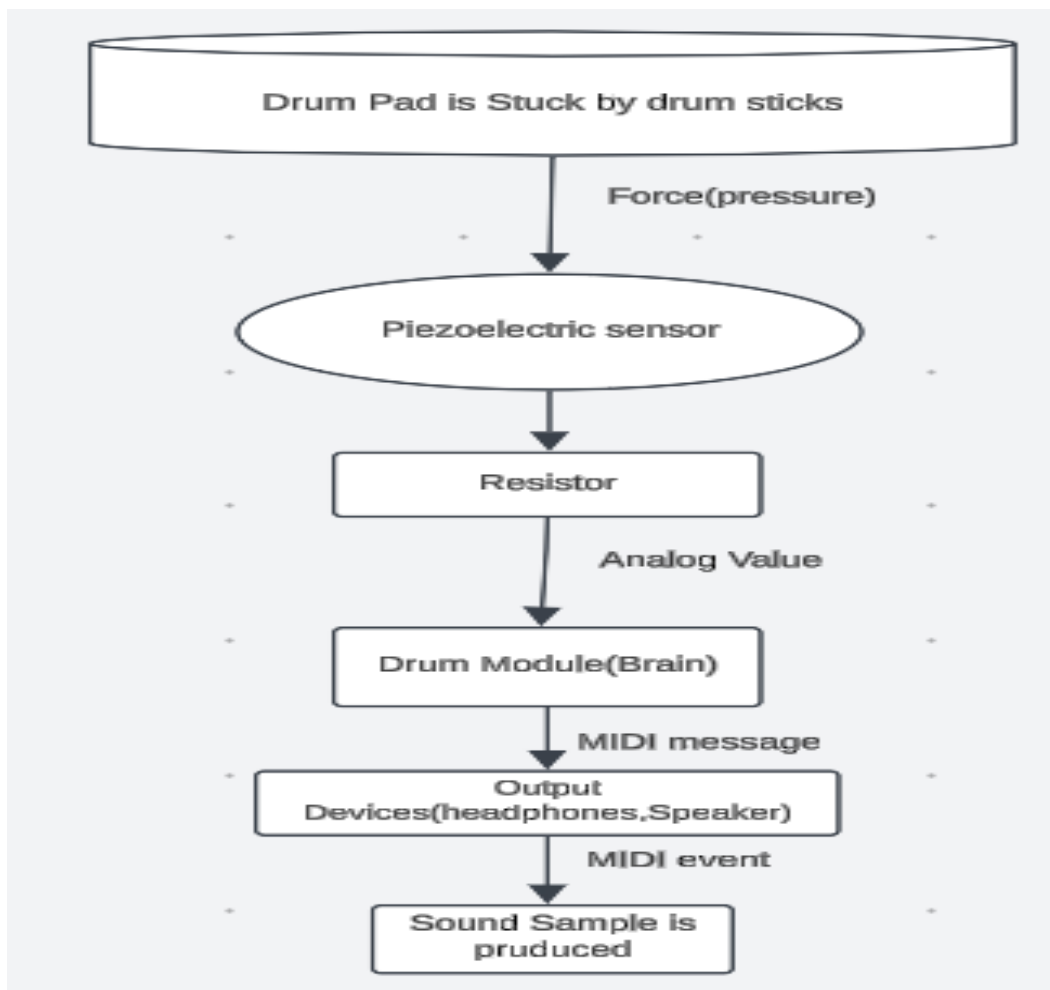


Figure 3.11 System Flow Design

Image Source: Author

3.1.7.2 Sensor Setup

There are many types of Drum Pads, with varying applicability depending on the usefulness of the different playing techniques. Therefore, the number of piezo sensors differed from the most basic pad to those that contained multiple zones. They could be categorized into the following types of electronic drum pads along with their respective piezo sensor requirements:

3.1.7.3 Single-Zone Drum Pad

These are the most basic electronic pads, designed to identify strikes in just a single zone. These pads are destined for toms, kicks, and other drums that do not demand more than one striking zone. Requires 1 sensor to be placed under the Center of the pad to detect any strike on the pad surface

Examples: Such as kick drum pads and tom pads in basic kits.

3.1.7.4 Dual-Zone Drum Pad

Such pads are made to detect hits in two different areas, usually the drumhead (center) and the rim, which could serve well for snare drums. Two sensors are required to be placed one: Head sensor placed under the center of the pad, and the Rim sensor placed at the edge to register Rim shots.

example: Dual-zone snare drum pads, dual-zone tom pads.

3.1.7.5 Triple-Zone Drum Pad

Such pads are made to detect hits from three separate zones; these pads can find application in more complex playing techniques, much like those on a ride cymbal. The sensors are usually placed under the center of the pad(the drumhead), rim, and either bell or edge.

The table below shows different types of drum pads that produce one to three different sounds, therefore, our group followed this table to place the sensors in their designated location.

Pad Type	Number of sensors	Pad placement
Single-Zone bass drum/kick Drum)	1	Under the Center of the Pad
Dual zone Drum Stool/Throne	2	Center of the and Rim of the Pad

Triple Zone Hit-Hats	3	Center, Rim, and Bell of the Pad
Single-Zone Cymbal Pad	1	Center of the pad
Dual zone Cymbal Pad	2	Bow and Edge
Triple Zone Cymbal Pad	3	Bow,Edge, and Bell
High Hat Pad (Single Zone)	1	Bow
High Hat Pad (dual Zone)	2	Bow and Edge
Kick Drum Pads	1	One Zone
Percussion Multipads	1 per pad	Single zone

Table 3.1 Sensor set up for different types of drum pads

3.1.7.6 Sensor connection to the Control Board

A single zone pad had one sensor, with the positive lead connected to the analog input pin of the microcontroller and the negative lead connected to the ground on the MCU. A 100-ohm resistor was placed in parallel to the piezo sensor as a pull-down resistor to dampen the voltage spikes and bring them down to safe levels, ensuring a good reading of the voltage produced by the sensor. The piezo sensor and the resistor were soldered together to ensure a solid connection, keeping the analog voltage reading at zero volts until the piezo sensor sent a signal (voltage), thereby preventing false triggering. After adding the resistor, the positive side was connected to the microcontroller's analog input, let's say A0, and the negative side was connected to ground (GND).

Additionally, a Schottky diode was considered for protection against high voltage spikes generated by the piezo sensor, connected in parallel with it. The diode's cathode was connected to the positive lead of the piezo sensor, while the anode was connected to the ground. If necessary, a small resistor of about 10K ohms was added in series with the piezo sensor to limit the current flowing into the analog input and protect the microcontroller.

For the double and triple zone pads, which contained more than one piezoelectric sensor, the same method of connection used for the single zone pad was followed. In a pad with two sensors, each sensor was wired to a different analog input on the control board.

A final circuit design of connecting the OmniDrum piezo sensors to the microcontroller using one diode and one resistor for the piezo sensor looks like the figure below adding more sensors.

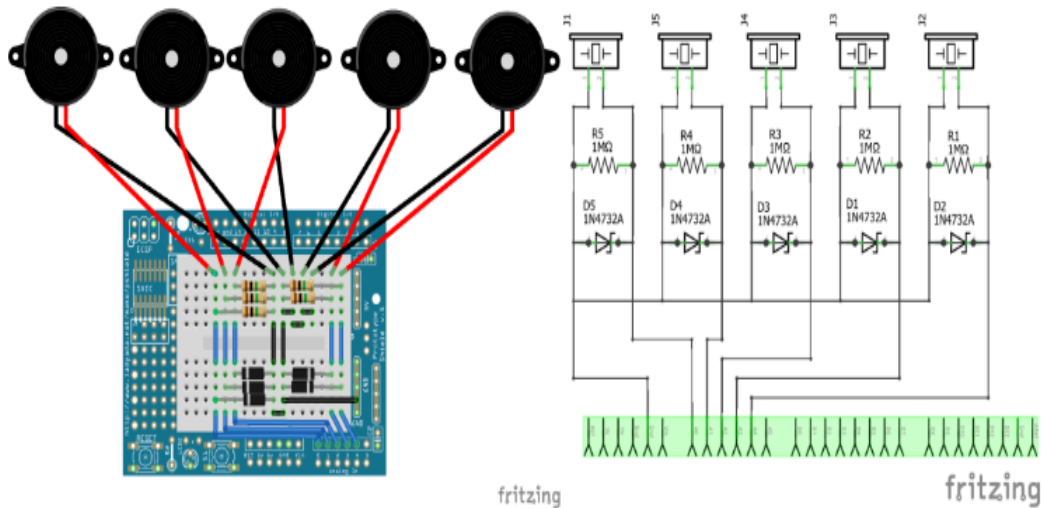


Figure 3.12 Circuit Design of Piezo Sensor to the Arduino MCU

Image Source: ([Arduino Piezo MIDI Controller – Simple DIY Electronic Music Projects](#))

3.1.8 Power Supply

This section went over the possible power supply specifications and components that would be needed to deliver power to our MCU and peripherals.

3.1.8.1 Power Specifications (Voltage and Current)

Our goal was to maintain the Omnidrum efficiency while keeping the total power consumption below 15 watts, to do that we needed to focus on both the power supply and the design of the circuit

Our group determined the voltage and current required for our OmniDrum system. In most of the electronic drum kits, the power requirements are usually between 9V and 12V DC, while the current supply will depend on the number of components (drum pads, sound module, displays, microcontroller, and amplifier). A calculation or estimation of the total current consumption for our system was performed based on these components. The power consumption for Omnidrum was broken down as follows:

3.1.8.2 MSP430

The choice of a microcontroller was based on our project objectives of efficiency along with the low power consumption also maintaining a low failure rate, for our OmniDrum we considered two types of microcontrollers.

Because we used this microcontroller in the past classes like embedded systems and Junior Design 1 this might have been a good choice for our project,

but let's look at the power consumption of the MSP430, the operating input voltage of the MSP430 ranges from 3.6V to 15V.

The power requirements for each MSP430 family are listed below in Tables 3. The power given is based on the amount of current the core consumes per megahertz (MHz). The Analog I_{MAX} column indicates the amount of current added if the additional functional blocks are used.

Table 3. MSP430x2xx Family Power Requirements⁽¹⁾

DEVICE FAMILY	PIN NAME	VOLTAGE (V)		CPU I _{MAX} ⁽²⁾ (μ A/MHz)	ANALOG I _{MAX} (μ A)	COMMENTS
		MIN	MAX			
F20xx	V _{CC}	1.8	3.6	370	Comp_A+ = 60 ADC10 = 1200, ADC10_I _{REF} = 400 SD16_A + I _{REF} = 1700 RefBuffer = 600	20x1: Comp_A+ 20x2: ADC10 20x3: SD16_A
F21x1	V _{CC}	1.8	3.6	410	Comp_A+ = 60	
F21x2	A _{VCC} , D _{VCC}	1.8	3.6	350	Comp_A+ = 60 ADC10 = 1200, I _{REF} = 400	
F22xx	A _{VCC} , D _{VCC} ⁽³⁾	1.8	3.6	550	ADC12 = 1200, I _{REF} = 400 OA = 290	22x2: ADC10 22x4: ADC10, 2 OAs OA currents for a single amplifier
F23x0	A _{VCC} , D _{VCC} ⁽³⁾	1.8	3.6	550	Comp_A+ = 60	
F23x, 24x[1], 2410	A _{VCC} , D _{VCC} ⁽³⁾	1.8	3.6	445	Comp_A+ = 60, ADC12 = 1000, I _{REF} = 700	224x1: Comp_A+ 23x, 24x, 2410: Comp_A+, ADC12
F241x, 261x	A _{VCC} , D _{VCC} ⁽³⁾	1.8	3.6	560	Comp_A+ = 60, ADC12 = 1000, I _{REF} = 700 DAC12 = 1500	241x: Comp_A+, ADC12 261x: Comp_A+, ADC12, two DAC12s DAC12 outputs not loaded; DAC12 currents for a single DAC

Table 3.2 MSP460x2xx family power consumption

Image Source:

https://www.ti.com/lit/an/slva335c/slva335c.pdf?ts=1729720208179&ref_url=https%253A%252F%252Fwww.google.com%252F

3.1.8.3 Linear Voltage Regulator (TPS6212x 15-V, 75-mA Highly Efficient Buck Converter)

Since the MSP430 requires only 3.3V input Voltage, it was necessary to use a step-down voltage regulator between the power source, which could have been an AC adapter or a battery, and the MCU. The TPS62122 was a highly efficient solution, achieving up to 96% efficiency and capable of driving 75-mA loads. With an 11- μ A quiescent current, the TPS62122 was an ideal choice for systems concerned about battery life.

The TPS62122 could be purchased for about five dollars. In addition to its low cost, this device provided extra protection to the MSP430 during the shutdown process. When VOUT reached regulation (3.3V in this example), the TPS62122 signaled the MSP430 through the PG (power good) pin. The TPS62122's SGND pin offered additional protection during shutdown. In

shutdown mode, SGND provided a discharge path from the output capacitor to ground, preventing lingering output voltages from discharging through the MSP430.

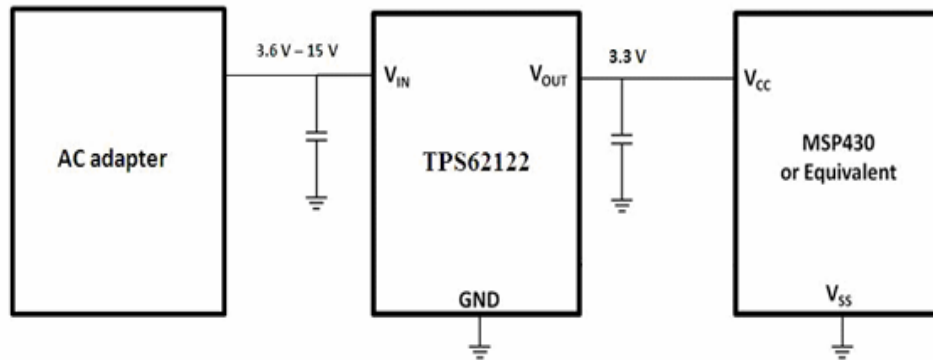


Figure 1. TPS62120 and MSP430 Simplified Block Diagram

Figure 3.13 TPS62120 and MSP430 Simplified Block Diagrams

Image Source:

https://www.ti.com/lit/an/slva335c/slva335c.pdf?ts=1729720208179&ref_url=https%253A%252F%252Fwww.google.com%252F

3.1.8.4 Linear and Low Dropout (LDO) Regulator with reverse current protection

The TPS709-Q1 low dropout voltage regulator has low voltage input ranging 2.7Volts to 30 volts which cost only about a dollar, it provides a regulated output voltage ranging 1.2volts to 5 volts. The LDO is capable of providing a steady output voltage without being impacted by the change in the input voltage which minimizes the noise making a good candidate for our design. Since the power consumption plays a big role in our project, using this device would've been an issue for our design because of the heat dissipation due to the large difference between the input and the output voltages

The power dissipation of an LDO $P(\text{loss})$ is:

$$P(\text{loss}) = (V_{\text{in}} - V_{\text{out}}) I_{\text{out}}$$

$$\text{Efficiency} = V_{\text{out}}/V_{\text{in}}$$

This creates a reduction of efficiency by $V_{\text{out}}/V_{\text{in}}$

So, if our input voltage is 12V and output voltage which would be the voltage required by the MCU about 3.3V for the MSP430

$$\text{Efficiency reduction} = 3.3/12 = 0.275 = 27\%$$

The excessive of the power dissipation could also damage the LDO

The LDO works better when the output Voltage is closer in value to the input Voltage or if the power consumption is not top priority for the design.

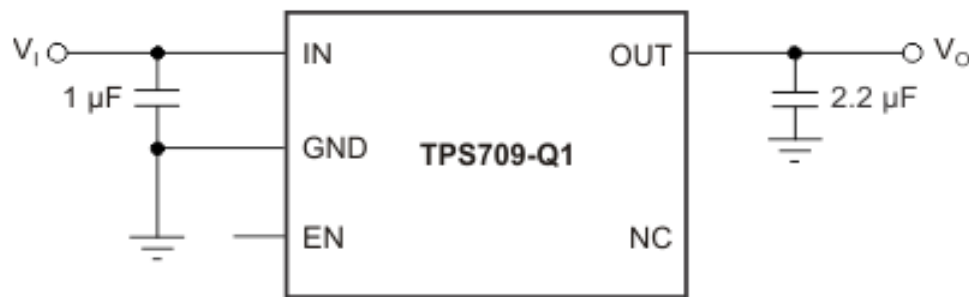


Figure 3.14 Low Dropout Voltage Regulator (LDO) TPS709-Q1

Image Source: [TPS709-Q1 data sheet, product information and support | TI.com](#)

3.1.8.5 Arduino

When it comes to choosing the right microcontroller, the size can play a big factor in reducing power consumption because each additional electrical device in the PCB consumes power to operate, therefore we needed to look at the smaller one that fits our project. Other methods used to lower the power consumption involve reducing the clock speed and enabling the low power mode. The table below shows the power consumption of different Arduino microcontrollers with different sizes using the three methods mentioned above.

Microcontroller	Reference 9V	Reduce Clock Speed 9V		Reduce Clock Speed and Operation Voltage 3.3V		Enable Low Power Mode 3.3V		Enable Low Power Mode 9V	
	Absolute	Absolute	Relative	Absolute	Relative	Absolute	Relative	Absolute	Relative
Arduino Nano	22.1 mA	18.5 mA	-16%	3.4 mA	-85%	3.4 mA	-84%	4.8 mA	-78%
Arduino Pro Mini 5V	14.6 mA	10.0 mA	-32%	3.7 mA	-75%	1.6 mA	-89%	3.2 mA	-78%
Arduino Pro Mini 3.3V	5.1 mA	3.8 mA	-25%	3.7 mA	-27%	1.6 mA	-69%	3.2 mA	-38%
Arduino Uno	98.4 mA	42.8 mA	-57%	11.6 mA	-88%	11.5 mA	-88%	27.9 mA	-72%
Arduino Mega	73.2 mA	61.8 mA	-16%	16.7 mA	-77%	11.9 mA	-84%	26.9 mA	-63%
Average Reduction			-29%		-70%		-83%		-66%

Table 3.3 Power Consumption of Different Arduino Microcontrollers

Image Source: [Guide to reduce the Arduino Power Consumption - DIYIO](#)

From these results we can see that the Arduino Pro mini is the most power efficient but the only problem with it is that it only has 6 analog inputs. Therefore, if this option had been chosen for the project, a multiplexer would have needed to be added to expand the number of analog inputs to accommodate the number of drum pads required for the OmniDrum. One example of a multiplexer that could have been integrated into the microcontroller was the Texas Instruments CD74HC4067 16-Channel multiplexer, which cost only \$1.05.

Source ([16 Channel Multiplexer - COM-00299 - SparkFun Electronics](#))

3.1.8.6 AC Power Adapter

A stable solution would be to use an AC to DC adapter, which converts the high voltage AC down to the level of DC voltage required by the OmniDrum. The AC adapter must have the same voltage and current specifications required for the OmniDrum kit with a clean supply to prevent electrical noise for a good sound quality.

3.1.8.7 Internal Power Regulation

Some components of the OmniDrum may require different voltages. In such cases, a voltage regulator circuit was needed. For example, where portions of a system (sensors or controllers) require a voltage of 5V while the main supply is at a voltage of 12V, a step-down voltage regulator could be used to convert voltage to the required level (a set ratio).

3.1.8.8 Protection and Safety

Fuses were added in our power circuit to prevent excessive current from damaging the circuit in case of a shorting or an overcurrent situation. The fuse was just a little larger than our total expected current rating of the system. We also used diodes in the power supply, which saved some of the sensitive electronics from getting damaged due to reverse polarity.

3.1.8.9 Energy Efficiency

Low Power Components: If your products are battery-powered, then you would want to use components that consume less power to prolong battery life. Using efficient amplifiers or LED lights that don't suck too much power can minimize energy use. **Sleep Modes:** If your drum kit has a microcontroller, then putting your peripheral unit(s) into the sleep mode whenever it isn't used helps to minimize power consumption on the drum.

3.1.8.10 Existing Product Examples

Most commercial electric drum kits, like Roland and Yamaha, use 9V or 12V common external power adapters which have certain current ratings (usually 500mA-2A), which forms a yardstick of reference to design your own.

3.1.9 Software Languages

The language that best suited the needs of our project was determined by many factors, such as hardware choices, software support, granularity, and the abstraction of hardware from software. The C language can be simple and powerful for developing at the lower level, having control over memory and device hardware. Code can also be developed to release some of this control. Java can use the Java Virtual Machine to abstract this level of granularity in our code. With the JVM, our program managed memory efficiently and provided interfaces to the hardware in a more organized manner. Both languages were widely adopted and had contrasting benefits. The more modern features of Java were applicable to some functions of our project while taking away some control from the program. Using C allowed for better integration with the operating system we chose. A portion of our program was developed in C to retain the desired level of control. The support available from these languages aided in reducing development time.

3.1.9.1 C

The C programming language is a fundamental tool for programming, offering a unique set of features that make it an excellent choice for special projects. Some explanations for why C is such a popular language include simplicity and low-level control of the program. C can be easy to use because of how simple it is. Other languages that include abstract concepts or complex syntax can take focus away from the goal of the program. An advantage of C is also the ability to provide direct access to hardware components, making it an excellent candidate for low-level programming on embedded systems.

This language does not have a lot of object-oriented features included. This lack of object-oriented features would mean developing our own functions or using premade libraries for some solutions. This gives a greater control to the programmer as they develop low-level code. It is a popular language for embedded and semiconductor programming and an excellent choice for our hardware development. Due to how long C has been available, it lacks some of the modern features common for object-oriented programming. An alternative solution for object-oriented projects or inheritance from modern languages would need to be created from scratch or sourced from a number of premade libraries. This would be necessary to achieve a similar result to the modern languages available.

With how long C had existed, there were many premade libraries available for developers to use. These libraries already included the necessary code to perform simple to complex tasks, which saved time for developers. Writing our

own algorithms or data structures from scratch took a significant amount of time, particularly when a program used multiple different variations of these types.

A key advantage is easy direct access to the hardware through C. This allows developers to adjust memory in different hardware that is attached to a C program. This language can provide direct access to registers, memory, and flags in an embedded microcontroller making it a great choice for developers. Code written in C requires compilation to machine code instructions. A separate computer would be needed to compile the code into an executable format before it is flashed onto the embedded microcontroller.

Developing code for our microcontroller with Arduino IDE allowed us to use the C language. This IDE supported a wide range of microcontrollers and had a large user base contributing even more support to devices. Embedded devices benefited from the libraries directly available from Arduino IDE. Common tasks such as input and output varied between microcontrollers, and having a collection of premade libraries through this IDE sped up development for our chosen hardware. A debugging tool was also included to support us during development. This debugging tool assisted with testing the program with breakpoints where we could find values for our variables. Arduino IDE included many features that aided in the swift development of our project.

The conclusions made were that the C programming language was excellent for direct hardware access or solutions to complex problems. Its simplicity made it a popular choice for programmers. The key advantages of its simple syntax, direct access to hardware components, and abundance of premade libraries offered a unique set of strengths that made it an excellent choice for specific projects using embedded systems.

3.1.9.2 Java

The Java programming language stands out as a versatile tool which is widely adopted across industries. It is a language with strong focus on object-oriented and class-based code. Some of the many advantages of Java are portability of code, organization, external libraries and tools, and official documentation.

Perhaps the greatest advantage of Java is its portability. Unlike other languages being specific to certain platforms and devices, Java's binary code can run on any device with a Java Virtual Machine. This means that Java programs can be easily installed on different environments. There is a partner component called the Java Development Kit which can be used during development and debugging. This kit includes a feature-rich set of tools for testing and finding bugs in a Java application.

The support for object-oriented programming helps to merge concepts of classes, objects, abstraction, polymorphism, encapsulation, and inheritance. Modern languages introduce many such concepts to developers to aid in creating organized code. These ideas can also produce greater resiliency to errors and protect from unauthorized access to data. There are built in security features

providing extra protection against a theoretical threat or vulnerability. Concepts of inheritance allow developers to reuse or recycle existing code in a way that can reduce the overall size of the code base. It facilitates the creation of organized code, making it easier to look through. In a large codebase with many components being added to the project the program can operate seamlessly when developed in an organized fashion.

Throughout the lifetime of Java there are many libraries, frameworks, and tools available. Adoption by industries has led to Java getting broad support. Industries that have chosen this language for their applications have an interest in finding bugs and improving the language overall. Many industries produce publicly available libraries to draw more interest in Java thereby expanding outside support of the language. It is not only enterprises that are interested in the language. Free source codes are available from countless FOSS projects which also share an interest in making the language all around better for everyone.

Java is an excellent choice for a versatile language that can handle complex projects. The portability alone was an attractive option. At scale this language can manage complicated applications efficiently. Even for small scale projects there is enough flexibility included to perform tasks with ease. A wide adoption through industries and devices, Java was positioned well as a leading programming language.

3.1.10 Sound Library

The drum kit had resources for media playback, which required a selection of sound files saved on persistent storage. The persistent storage medium was in the form of an external flash storage device. Our circuit board incorporated an external flash card reader for the system to mount and read this data. Software was a limiting factor to the various formats of filesystems and audio files that were supported. At the filesystem level, the drum kit followed a structured pattern to access all of these resources. Once the data became accessible to the device, our data was processed by the media codecs for playback. As with filesystems, there were many audio codecs that could be supported by the device. This support was simplified to a selected codec for quick implementation in the controller board software.

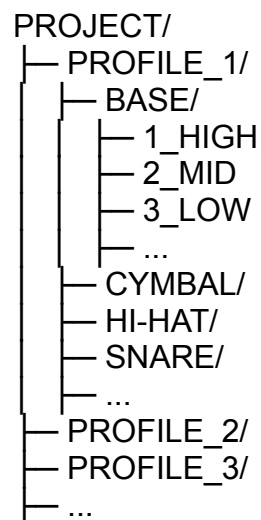
3.1.10.1 Directory Structure

Sound files were saved in external flash storage. The directory structure for these files did not use spaces in the directory paths or file naming. Each instrument mapped to the proper sound file on this device. Some devices were mapped to the same sound profile, such as two cymbal instruments or multiple snare drums. A directory structure was implemented such that a profile had a sound file mapped to each instrument installed. Multiple profiles were saved to

flash storage to cycle through using the controller board buttons. The modules of our drum kit consist of four distinct instrument types:

1. Cymbal
2. Hi-hat
3. Snare
4. Base

For each of the four instrument types, there were several strike levels that could have their own sound file. Depending on how hard an instrument was hit, the level of the strike was determined by the sensor's output. Within each profile, there were directories for each instrument, which contained the required sounds for each level of strike.



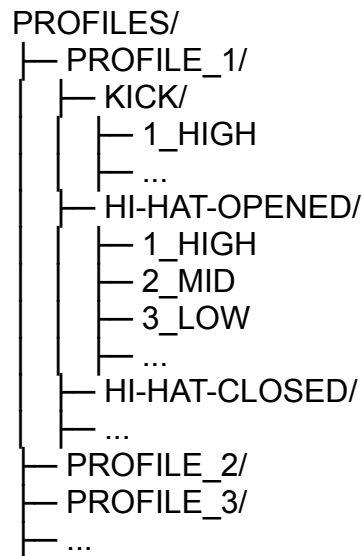
The PROJECT folder was located in the top directory of the device, meaning it was not within any other folder on the device. All profiles resided within this PROJECT folder. Only profile folders existed in this directory. If there were folders containing anything unrelated to the sound library, those folders could have caused unforeseeable errors. The number of profiles that existed here was no greater than 16 or 32, as users cycling through larger amounts may have had a poor experience. The naming convention for any custom profile folder did not need to adhere to any specific guideline, although it was greatly helpful to include a number either at the beginning or the end to arrange it in a specific order in the selection display. The numbers were unique, starting from 1 up to the maximum number of profiles chosen for implementation..

Within each profile folder, the default configuration was able to automatically assign instruments. The default configurations included instrument names such as BASE, CYMBAL, HI-HAT, and SNARE. The absence of any of these folders may have left the corresponding instrument unassigned unless other configuration data could be found, such as the profile.conf settings file,

which resided within its respective profile folder. The sound files within each instrument folder needed to be numbered starting from 1. The number for sound files should have appeared at the beginning, followed by a label, which could have been anything. The standard number of files for each instrument was three, although as few as one file was necessary for operation. Greater numbers of files could be included and would equally divide strike levels among them. The loudest strike was always numbered 1 and became softer as the count increased to 2, 3, 4, etc.

Senior Design II:

The software design for our profile creation was organized to handle different naming schemes when scanning through the SD card files. There were no specific standards for folder or file naming, other than the fact that our software scanned through the files in alphabetical order. Keeping in mind that we scanned our filesystem in alphabetical order, the naming of our directory structure was arranged in the order of the instruments listed below.



Note the order of the instruments for our kick pedal, opened hi-hat, and closed hi-hat. These three instruments are locked in this order and it is important to have folder names and file names in alphabetical order to match these positions. These instruments are interfaced with the kick pedal, hi-hat pedal, and the hi-hat sensor. The remaining five sensors can be assigned in any way the user wishes. Since the kick pedal is effectively a switch, only one sound effect can be used by that instrument. More than one sound effect is possible to have within the kick pedal folder though only the first sound effect in alphabetic order would be used.

3.1.10.2 Filesystem

The format that was most suitable for our integrated computer was the FAT32 filesystem. Nearly every device could use this format, allowing a user to change the contents stored on the external storage. One of the biggest restrictions of this format was the maximum partition size of 2TB at a sector size of 512 Bytes and a file size of 4GB. These restrictions were well beyond the needs of our storage device requirements. Contents in this storage device were only read from. Because we used this as read-only memory (ROM), the implementation of FAT32 support in software was greatly simplified. Even though we were only reading from the device, this did not mean that the device was only capable of reading through the software. Aside from these reasons, the legacy factor of FAT32 also provided a plethora of documentation and resources. Source code for other project implementations of this file format was widely available online if we wished to use a free software solution. The free library FATFS could be imported to handle filesystem detection and data transmission. With this library, our storage could be mounted, the directory structure parsed, and file operations conducted on FAT32 filesystems.

Extending support to other filesystems required different implementations, which could take focus away from developing the rest of the project. Personal computers at that time did not all agree on standard filesystem support, such as proprietary ones by Microsoft or Apple. NTFS from Microsoft and APFS from Apple were the common formats that each company used when formatting disks. These filesystems included more modern features for data storage. The features represented advanced and complex methods of ensuring data integrity and security. Adopting support for either format was at the discretion of the developer, as each company may have been content with supporting its own. What was common, though, was support for the older legacy formats. FAT and exFAT still had wide adoption throughout devices, from personal computers to smartphones and beyond. Because we were not interested in modern features such as checksumming, encryption, or compression, we could forgo implementing support for any additional filesystems.

3.1.10.3 File Format

The performance of our project was significantly impacted by the choices of which file format and codecs we chose to use. Several hardware constraints influenced the decision to use compressed audio over uncompressed audio formats. The time to read from the flash storage greatly increased when we opted for uncompressed data; conversely, if a compressed codec required too much processing from the integrated computer, our project could behave poorly. Codecs that implemented multiple compression level schemes introduced greater complexities in writing software.

The contents of any given sound file consisted of nearly a second of audio to play the sound of an instrument. For uncompressed audio, this duration resulted in a small amount of data. With this in mind, we considered using the

lossless and uncompressed audio codec of WAV. The relatively small sizes of our instrument sound files and minimal hardware requirements fit the constraints of our design. The drum kit hardware had minimal compute cycles and I/O available to handle decompressing other formats. Software requirements to implement this WAV format were simplified for these same reasons.

Audio buffering assisted in lowering I/O to the external storage by caching files from the currently selected profile closer to the processing cores. When a profile was active, we precached the relatively small sound files in memory, and with uncompressed audio, there was not a larger memory footprint from caching both the compressed data and the decompressed audio. The WAV format had minimal characteristics for how data was encoded, including details such as sample rate, channel count, and bit depth. A common configuration for WAV files included a 44.1 kHz sample rate, 16-bit depth, and stereo channel audio. For a one-second sound file in this format, the file size was approximately 90 kB. The operating system for the controller board was able to implement this common case at a minimum. The software included broad support for the WAV format as development continued or through the use of freely available libraries to provide greater support for the format.

3.1.10.4 SPI (Serial Peripheral Interface)

The SPI protocol was used for the communication between our external storage and microcontroller. The card was made available to the system through several software setups, such as initializing the card and placing it into SPI mode. Communication between the card and microcontroller was paired during the initialization steps. Once the device was ready to operate, it was necessary to locate the filesystem and partition table. Our external storage was formatted in the FAT32 filesystem and could be discovered through the FATFS library. Before the FATFS library could detect and mount our storage, the software needed to perform the SPI communication setup.

Input and output of data across SPI could be achieved at a block level, where a request command for 512 bytes of data could be retrieved. The storage device would respond with the requested block or blocks of data. The commands to request blocks could be for single blocks using CMD17 or continuous blocks using CMD18. Using a library such as FATFS could manage filesystem discovery and traversal. With this library, our read requests were achieved with simple file lookups.

While the system was running and our storage was removed, the software needed to unmount the filesystem. We wanted the filesystem to remount when the card was reinserted. This required us to perform the SPI setup on insertion and implement a monitoring task to check whether the card was inserted. Our monitoring task polled the SD card reader for a response and made our software aware of when there was no storage attached. The filesystem needed to be unmounted when removed, placing our project into a standby state where no file operations were performed.

The monitoring task can flag the system to standby without the storage attached and continue monitoring the SD card reader until it is reinserted. On reinsertion our card follows the setup of SPI again before being automatically handled by the FATFS library. With these steps in our software, we implemented the sound library through SPI using the FAT32 filesystem and resumed operation seamlessly if the storage was removed and reinserted while powered on.

3.1.11 Controller Board

This is the central component of the design that connects all peripherals together. The system software running on this board orchestrates the software interfaces to the sound library and instrument modules. These inputs are mapped together by the software so that an instrument corresponds to sound files. Users can remap these instruments for each profile as they cycle through the available resources of the sound library. Most, if not all, of the assignments are done automatically by the operating system. User intervention is kept to a minimum when selecting an active profile. The operating system also manages the output of both audio and MIDI data. All of these inputs and outputs are handled with minimal latency.

Senior Design II:

Our configuration of each sensor may be done by plugging in each sensor to one of the six ports on our device. Rather than having a software assignment of each sensor we can assign the sensor input ports to the profiles index. The profiles data structure contains all of the information that is needed for the specific instruments of our device such as the sound effects and MIDI code used. We allow the plug and play feature of our sensors to let the user arrange their sensor modules in whichever way they desire and assign each module when they plug into these ports. If a user wishes to reassign the instrument to use different sound effects they only need to change the port it uses.

3.1.11.1 Instrument Assignment

For each of the instrument modules, there is a dedicated port. The instrument assignment is set statically in software to these input ports on the controller board. Statically assigning the instrument to these ports simplifies coherence between the hardware layout and the operating system software. Implementing dynamic instrument assignment based on whichever module is plugged into a given port presents challenges, as there needs to be a method to identify each module.

An alternative to having the operating system detect which module is connected is to allow the user to control which input port corresponds to a particular module. The hardware requirements for this implementation include a display that shows the configurations of each port, a button to select and cycle through these ports, and a button to cycle through one of the four instrument

types for assignment. Improper configuration can cause issues between the types of modules, as the sensors for each module may have different calibration levels. The values read from one instrument module may differ from another instrument, such as cymbals and snares. Matching each port correctly is straightforward from a user perspective, regardless of these decisions.

Not every port needs to be used. If some ports remain unused, the instrument assignment for these ports can still be set, but they are nullified in software. There are no signals received from an unused port. Even when a port is unused, we can implement software that does not take any value until an instrument module is inserted. While the software runs, a user can plug and unplug these modules. The operating system remains aware of which plugs are active and which are disabled. Allowing each port to have an instrument assigned provides plug-and-play functionality. The operating system detects when a port changes between unused and used.

```
PROJECT/
├── PROFILE_1/
│   ├── BASE/
│   ├── CYMBAL/
│   ├── HI-HAT/
│   ├── SNARE/
│   └── ...
```

Expanding the number of instruments in a profile is as simple as adding another directory in the profile. While some modules can be assigned to share the same sound file, there are also ways to assign more unique combinations. If a user wishes to change the sound of one of their snare modules, they can do so using the selection controls on the controller board. The snare module can be assigned to one of the other available instrument sound files or to an added instrument placed in the profile. The display shows information about which port is selected and cycles through the directory names inside the active profile.

Senior Design II:

Each profile is expected to contain at least eight folders, one for each instrument. These folders are coded to map to our sensors and pedals in the following manner:

```
PROFILES/
├── PROFILE_1/
│   ├── KICK/
│   ├── OPEN-HAT/
│   ├── CLOSED-HAT/
│   ├── (SENSOR 2)
│   └── ...
```

| └─ (SENSOR 6)

Note that the assignment for sensor 1 is split to use both the open-hat and closed-hat folders. Whichever folder is used depends on the hi-hat pedal press.

3.1.11.2 Profile Selection

A standard profile with four distinct instruments and three levels of strike detection was approximately 1MB in size when using one-second WAV formatted files. This allowed for many profiles to be loaded onto our external storage device, although the number of profiles that could be cycled was limited to 16 unique profiles. The controller board had a display and button controls for navigating the available selections of profiles. There was a way to cycle through all available directories within the PROFILE folder, functioning with forward and back buttons. Although some profiles were customized to include greater or fewer instruments and sound files, a user had all instrument assignments made when switching profiles. This could be challenging when a user created a profile that deviated from the standard file structure, as some custom sound levels and instruments might have been present.

```
PROJECT/  
├─ PROFILE_1/  
├─ PROFILE_2/  
├─ PROFILE_3/  
└─ ...
```

In cases where a profile did not follow the standard file structure, the operating system kept the profile's settings when it was cycled back. A persistent method that allowed profiles to retain their assignment settings included a profile.conf file inside each given profile. Between power cycles, this persistent file could load the custom profile in the desired configuration. The user experience was considered important when evaluating how customizable any given profile could be.

In the absence of such configurations, the operating system made a best effort to assign a profile to a default configuration. The default configuration was set to PROFILE_1 with the standard file structure. If any instrument folder was missing from the standard profile structure, the system did not reassign automatically but rather left the instrument unassigned for the user to choose.

Senior Design II:

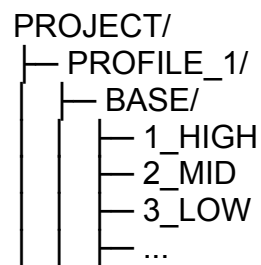
There are three status LEDs available on the device, each one is used to indicate what the active profile is. Although it is possible to include more profiles on the SD card and our software may be written to have more profiles, the current maximum number of profiles is set to three. These can be switched

between with the profile select button and will cycle through the available profiles and wrap back around to the first profile.

3.1.11.3 Strike Level Selection

This project implemented sensors to detect various levels of strikes. The standard configuration distinguished three levels of a strike and a level for no strike. Each of the four instrument modules implemented different sensing technologies that sent out varying values for the same forces applied to them. Calibration of these sensors provided the threshold levels needed to divide the sound files.

The lowest threshold was established between the point of the lightest strike detected and the lowest sensor reading. This boundary served as the dampening value to differentiate the lowest strike from bumps and knocks, ensuring no audio played from these bounds. Each sensor was hardened against crosstalk and bumps through software. The highest threshold was set just below the sensor values for the hardest strike on an instrument. When a sensor read beyond this threshold, the sound file for the high strike played.



With both of these thresholds determined, we assigned a number of strike levels that were available for the assigned instrument. Profile customization could add or reduce the number of strike levels present in their directory. In the case of only one strike level being present, we used the lowest threshold as the value our sensor could read to play that sound. If two levels were present, we used these two existing thresholds. As more levels of strike detection were present, the system evenly split between these two boundaries while adding more levels of strike detection. The limitation for how many divisions could be made was determined by the range of values a sensor could provide.

The operating system tracked how many levels of strike an instrument included by counting the number of files in that instrument's directory. The file naming structure needed to follow standard conventions for the correct strike level to be used.

Senior Design II:

Determining the strike level uses the value of a sensor and can dynamically choose which index of our profiles data structure to use. The sensor

values are 12-bit and have a lower level cutoff value. The remaining range of our sensor values can be split between the number of sound effects for that instrument. The software determines how much of the sensor range divides each index and iterates by this number until the sensor value is less than or equal to the current iteration. When this condition is true the index of the profile data that is needed is found. This is how the index is determined in the translation between a sensor strike and the profile index to use.

3.1.11.4 Operating System

Our integrated computer hardware was a limiting factor for the operating system. With minimal hardware specifications, the software that ran on the controller board needed to be minimal and light on compute cycles. Other requirements for the software included responsiveness and highly parallelized communication for input and output streams. This system monitored the array of input ports for sensor data, read data from external storage, processed audio codecs, multiplexed audio playback, and managed output data streams to MIDI and speakers.

The drum kit needed to have operations performed in real-time or exhibit the behavior of real-time responsiveness. Creating a real-time operating system from scratch proved challenging. Some freely available operating systems that could be used as the base of our system included FreeRTOS. Support for this lightweight embedded operating system was present across many popular microcontrollers. Using this as the foundation for our software allowed for more focus on the custom software of our project.

At startup, our software booted into the default profile, PROFILE_1. Precaching then brought sound files from external storage into RAM. The software identified the directory structure of the active profile and bound instruments to the relevant inputs. Scanning through a profile included case handling for any deviations from the standard structure, such as any added or missing files or folders.

The input signals were individually buffered to memory, where their values could be used for branching instructions. The operating system scheduler prioritized the task that monitored these buffer values. By buffering the sensor readings, we retained the values of multiple sensors and read back all buffers at once. This scheduled task was considered to have a soft time deadline. In the event of a soft deadline passing for a real-time operating system, the value was not wasted but resulted in a degradation of performance. If the task missed the target deadline to read the buffer, it could return to this task and retrieve the value still contained in the buffer. Buffers were overwritten when their contents were read from this task or if a new input value was detected from the sensor.

Multiple instruments were able to play over each other. When a number of instruments were struck in series, the sounds layered over each other. To enable continuous playback for each sound file, a multiplexer configured for stereo channel output was considered. All source files were downmixed to stereo. Audio re-encoding was implemented to transform audio with more than two channels

before sending it to the multiplexer. If a source was mono audio, the single channel was sent to both channels to retain consistent playback from both sides. As more sounds were directed to the multiplexer, their signals were added over each other to create this layered audio effect.

Even though FreeRTOS was a complete operating system, there were still edge cases that could cause errors for our project. Error handling was dealt with gracefully while the system was running. During software development, a detailed software flow chart was used to explain each case and branching direction as the software ran. A completed state machine diagram helped to minimize edge cases and reduce errors while the drum kit was running.

Senior Design II:

A FreeRTOS implementation has some downside to our project specifically with regards to the maximum tick rate of the OS being 1 kHz. Every tick of the OS is a point where the scheduler can determine which task is completed and waiting to be swapped with another process. Tasks may also be switched out on a tick by the scheduler that leaves one task partially through it's execution and allows another task to start or continue it's execution. There are concurrency issues that our project needs to be developed around so that shared data between tasks can be processed properly. Techniques are implemented in the software that ensures shared resources cannot be manipulated out of turn or ensures some processes are atomic.

The tick rate of 1 kHz could cause issues if our tasks are too small. A task that has less processing to complete can take cycles of our CPU waiting for another task to be swapped. A solution for ensuring that there are not too many small tasks created is to combine the smaller tasks together into one single process. This can take advantage of using more cycles of the CPU while the task is scheduled and prevent blocking of other processing. There are tasks that will benefit from having more time on the CPU to provide greater responsiveness to the device such as our audio mixing and buffering tasks. There is less of a need for this responsiveness for tasks that handle polling such as the polling of the kick pedal and profile select button. Although our ADC array would need to appear responsive, this task also can benefit from less time on the CPU while continuing to feel responsive. Software for the ADCs is designed to run at a rate where each ADC value is still gathered within a reasonable time duration that would be imperceptible to the user.

3.1.11.5 Control Panel

User control over the device was provided with a set of buttons and a display module that showed relevant information. The menu consisted of nested options for users to navigate through and select the configuration they desired. The most significant function for the control panel was cycling through the sound libraries included profiles. As a user cycled through the available profiles, the operating system handled some autonomous control. The operating system could manage certain assignments of a profile to the instrument modules. A user might have wished to reassign or customize how the active profile was set up. In this case, a user could navigate this control panel through options such as which instrument was assigned to an input.

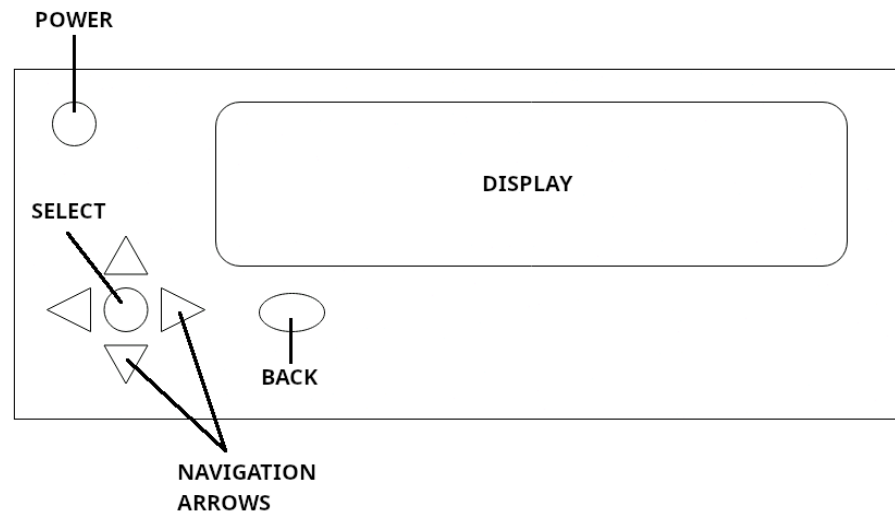
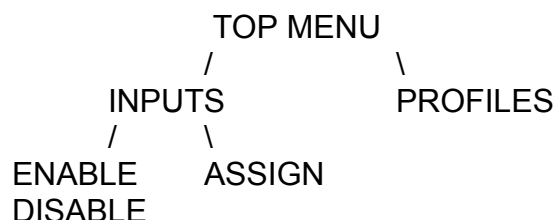


Figure 3.15 Control Interface and Display
By Jameson Palmer

The menu flow accounts for several submenus. As the user navigates further into any submenu the hierarchy of this navigation is designed with the given layers, or levels, below. Using the 'BACK' button will navigate this hierarchy up levels. The top level is actually displaying both the inputs and profile that is currently active.



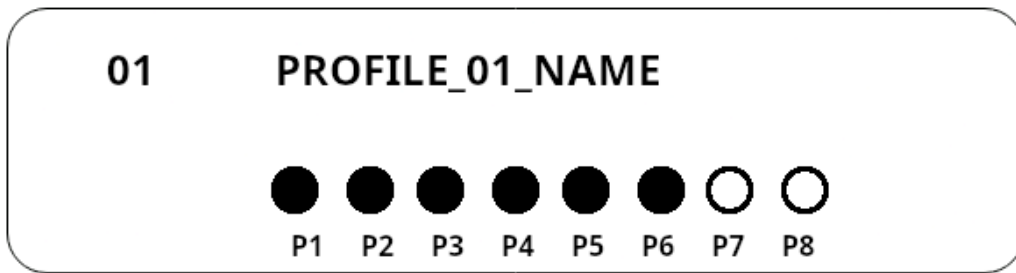


Figure 3.16 Top Menu Display
By Jameson Palmer

As the operating system pairs the sound library with instrument modules the status of these components will display along the screen. A status monitor is included to give a visual representation of the device configuration at a glance. A simplified approach to what information is displayed can help with ease of use and intuitive control. The display will show the input ports for our instruments and show which of these inputs are currently active. The arrow navigation buttons can highlight any one of these ports for additional options such as disabling an active port, reenabling a port, and changing which instrument a port is. For a port that does not have a wired connection the status should remain disabled.

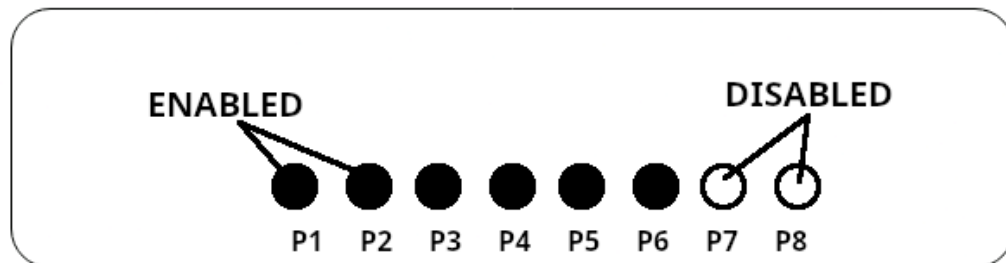


Figure 3.17 Input Port Status Monitor
By Jameson Palmer

Part of the customizability of our sound library allowed for including additional directories to represent more instruments. The standardized approach for a profile would have multiple instances of the same instrument to share the same copy of its sound files. Take the case of our kit having two snare drums connected. Our inputs would be assigned as snare drums and, because of this, use the snare drum sounds from the library. A custom profile may assign one or both of these instruments to use a different instrument in a profile. Custom instruments should be given appropriate folder names in the sound library as this is what is shown on screen when choosing a custom instrument. On the screen we navigate to any of our input ports and enter a submenu by using the 'SELECT' button. A list of available instruments are now shown for the user to

navigate. Hovering on an instrument and choosing 'SELECT' will reassign that input port. Choosing the 'BACK' navigation button closes the submenu.

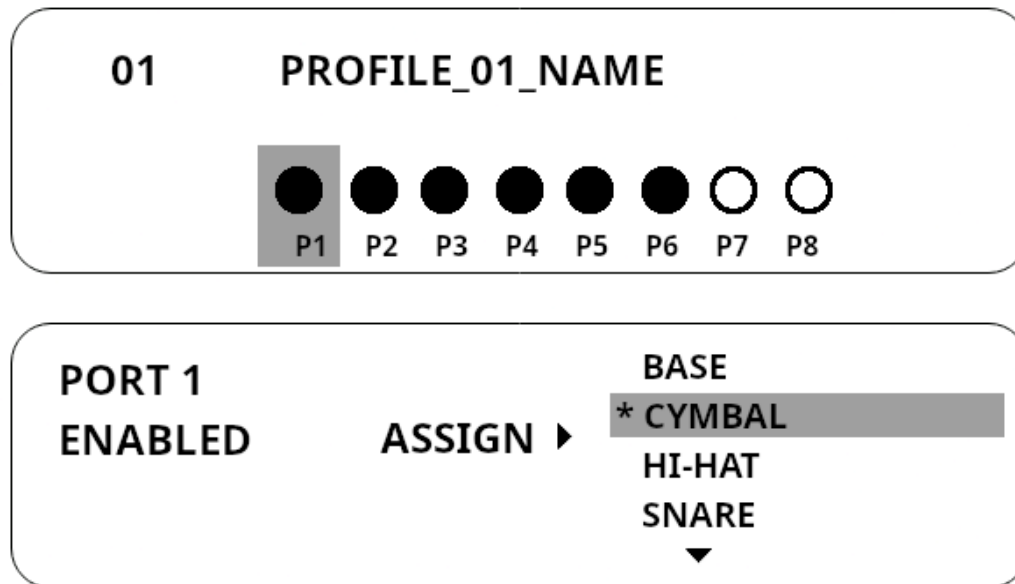


Figure 3.18 Input Port Assignment Submenu
By Jameson Palmer

Profile navigation is displayed on the second half of the screen. The folder name will display for the active profile and using the arrow navigation buttons can cycle through the included profiles in the sound library. Navigation will wrap around for the first and last profiles. Any number of customized profiles may be included into the sound library which may cause navigation to suffer as the number of profiles grows. An artificial restriction of 16 to 32 profiles is determined to round off edge cases and simplify our software implementation.

Senior Design II:

Although the device may benefit from a control panel scheme where a user can customize and configure the device with ease, our prototype is designed with the reasonable expectation that our instruments can be arranged and assigned by the sensor module ports. An instrument may be considered functionally disabled when the instrument port is not connected to a sensor. Even with nothing plugged in, the device will still process on that port by the ADC. If nothing is plugged in, the signal to the ADC is effectively grounded and will evaluate to zero or near-zero. The cutoff value of our software will prevent these low ADC readings from being processed as a strike just as if the sensor were connected. A user may change which sound effects are assigned to a sensor by connecting it to another port. All ports are statically assigned in accordance with the explanation in Instrument Assignment (see 3.1.11.1).

3.1.11.6 MIDI Data Output

We follow the common conventions of the MIDI code assignments when developing the software of our kit. The few instruments our design focuses around can be paired to these MIDI codes. The customizability of a profile brings a bigger challenge when developing for how these codes can be paired. A profile can include additional instruments that a user can assign. How does our software assign MIDI codes to such instruments? From a software perspective we may implement expanded support to the common MIDI drum values. A folder name would need to match any of the names of these expanded instruments to function this way.

MIDI codes can be changed on the users recording software. As long as we can assign distinct code values to any instrument we can differentiate that device. The idea that each device can be assigned a unique MIDI value allows the software to handle customized profiles more gracefully. Our supported codes can consist of a selection of common drum values and a set of reserved values to be used when no automatic pairing can be made. There should be a reserve code available for any of the plugged in modules in the most extreme case that no automatic pairing is made.

It should be noted that when a profile is selected it not only assigns the instrument from the sound library but also assigns the MIDI code. When the drum kit plays the instrument it plays both the sound and the MIDI data. This may occur even if the MIDI cable is not used. This can help simplify the running software as it doesn't need to monitor whether a cable is connected or not. This can grant the behavior of plug and play functionality without increasing the sophistication of software. We can be sure that MIDI data is transmitting when a cable is connected because this method will always be sending data.

Some instruments may be modified with foot pedals which are also part of the input array. A foot pedal modifier should not play until a strike on the related instrument is detected. When a strike on the instrument is made this foot pedal modifier will be checked before branching further. The flow of software can determine whether to use the sound file and MIDI code of the instrument with or without the modifier applied. Sound files and MIDI codes are applied to the foot pedal.

MIDI notes should have a duration relative to the duration of associated sound files. When parsing data in the media library the files contain metadata information on how long the file plays for. This can be retrieved and used as a timer for how long a MIDI note is played. This metadata will be collected and stored in a structure when the profile is selected. By keeping track of the current profile content and metadata in memory the software can access and keep track of the filesystem without having to retrieve information from storage. This metadata is not going to change while the device is running and storage is mounted so the information should be retrieved once per profile.

Senior Design II:

Each sound effect index of our profiles data structure contains the information needed for MIDI. This includes the MIDI code, channel, and duration. Every individual sound effect is able to be assigned uniquely however our prototype is designed to assign specific codes across the entire instrument. The device also separates the MIDI channel across each profile so for profile 1 if an instrument is struck it will play the code for that instrument on channel 1, profile 2 will send that same code but for channel 2, etc.

A duration attribute is possible to place for each sound effect, this would be possible to get from the WAV file header. This duration is in milliseconds and is used along with a hardware timer to determine which active MIDI notes are currently playing and when they expire. For the prototype design we set the duration to a small value of 50 milliseconds between the MIDI note on and off messages. The intention is to have the messages separated long enough for the external recording software to show the strike on its timeline. Having a delay too short could cause problems with the recording program. It is not strictly necessary to have a longer delay of our notes as the recording program itself can also be used to edit the note to be longer or shorter if the user wishes. The duration attribute being set to match a sound effect duration is more of a flair of our concept design.

Multiple notes can be played concurrently and expire independently through the use of a millisecond timer. Each tick of the timer is a millisecond period and with the current tick value we add the duration of the note. This value is the expiration tick which is compared at a reasonable rate with the current timer value. When the timer is at or past the value set for the expiration the software then sends the off message for that note. The software design is implemented in a way where each sound effect that is played would also have its corresponding MIDI note played. This 1:1 relation means that we have an array that holds the actively playing WAV files and an array that holds actively playing MIDI notes. A flag is set false when the WAV file or MIDI note is done playing and can be replaced with a new entry.

3.2 Part Selection

In this section, we focused on selecting the essential components for our project, ensuring that each part met the functional requirements and constraints of our design. The selection process involved evaluating various microcontrollers, audio output devices, amplifiers, sensors, and power supplies. Each component was assessed based on its specifications, performance capabilities, cost, and compatibility with other parts of the system. By carefully analyzing these factors, we aimed to create a well-rounded and efficient circuit that would deliver the desired functionalities while maintaining reliability and cost-effectiveness.

3.2.1 MCU

The first thing that we needed to decide on to begin forming our circuit was the microcontroller we wanted to use in our project. There were many different options, each with various onboard peripherals necessary for implementing the functionalities we desired. After conducting further research online, some possible options were found and are listed in the table below.

Model	STM32F405RG T6W LQFP64	STM32L562QE I6Q LQFP64	STM32G473R LQFP64
GPIO Pins	51	52	52
12-bit ADC Channels	16	16	26
External DACs	2	2	3
UART/USART/LP UART	2/4/0	2/3/1	2/3/1
SPI	3	8	4
Memory	1024 MB	512 KB	512 KB
Price	\$11.83	\$6.94	\$8.20

Table 3.1 Comparison of MCUs

The LQFP64 package size was chosen for all variants because it essentially guaranteed enough GPIO pins. A concern with the GPIO pins was that some core features overlapped. For example, in the STM32G473R, one of the SPI pins overlapped with a DAC output pin. This meant that we could not use SPI1 for the microSD card; fortunately, there was another pin with the same function on the LQFP64 package size, allowing it to work for our purposes. This was one reason why choosing a larger package size was beneficial. However, larger package sizes cost more and took up more space on the PCB. The LQFP64 package size provided a middle ground between these options.

Another benefit of using the LQFP64 package size was that the number of GPIO pins became less of a factor in choosing which chip to use. Among the chips listed, there was only one where the GPIO pin count was one less than the others, meaning that the GPIO Pins section of the table had a minimal impact on the choice of MCU.

The number of ADCs available on the chip significantly affected which chip was chosen. ADCs were to be used to measure values from our sensors to determine how hard they were hit and convert those measurements into values

usable by the MCU. We only needed about 6 ADCs, but as discussed earlier, pins could have overlapping capabilities, so having more was better to ensure that available pins were sufficient. In this category, the STM32G473R outperformed the other two by having 10 more ADC channels available for use.

Another important feature on our MCU was a good number of external DAC channels. Many MCUs had some combination of internal and external DACs. For our purposes, we specifically needed to consider external DACs. It should be noted that the terms DAC and DAC channels were used interchangeably here. In reality, many of these MCUs had one DAC with 2 DAC channels, functionally providing two separate output pins with independent audio channels. It was also crucial that these external DAC channels were buffered. Buffering allowed connections to outputs like amplifiers without worrying as much about pulling too much current or crosstalk between connected channels when combining signals. All of the DAC channels discussed in this section were buffered. Both the STM32F405RGT6W and the STM32L562QEI6Q had 2 external DAC channels, while the STM32G473R had 3 external DAC channels. While two channels could help create polyphony, they were not as effective as having three channels. With two channels, the likelihood of different sounds cutting each other off was much higher than with three channels, particularly in the case of a drum kit. Thus, the STM32G473R excelled compared to the other two choices.

The next item in the table was the UART/USART/LPUART count. To transfer MIDI information, we needed to utilize UART. USART in asynchronous mode could also function as a UART channel. Since only one channel was required, this goal was easily achievable. Each of the three microcontroller options could accomplish this, making this category somewhat irrelevant in our decision.

The following category in the table was the amount of flash memory available on the MCU. It should be noted that sound files would be stored on a microSD card, so their effect here was less relevant; however, we may still need to buffer our drum sounds. After some testing, it was found that a full drum profile without different sound files for each velocity level could easily be stored within 512 KB. Regardless, the STM32F405RGT6W won this category by having double the available flash memory compared to the other two.

The last category in the table was the price. The price difference between the choices was present, although not extreme. That said, the STM32F405RGT6W was significantly higher than the other options, meaning it needed to be exceptionally more useful to justify its selection. The STM32G473R, despite costing more than the STM32L562QEI6Q, provided another DAC output, which would be helpful when implementing the code. Given that the exact method of implementation was not known yet, it was best to choose an MCU with at least 3 DACs. There might be a way to combine the sound files digitally in real-time and only need a single DAC channel, but other methods of implementation might require playing the sounds on separate independent DAC channels. Since both the STM32G473R and the STM32L562QEI6Q shared the same amount of memory, but the STM32G473R

had an extra DAC channel, it seemed worth the slight price increase to use it over the others. It provided a good middle ground between memory and price, as well as an improvement in both ADC and DAC capability over the other two, making it the optimal choice for our project.

3.2.2 Audio Output Device

We needed to decide on which audio output device to use with our device to provide the user with auditory feedback while playing. The options are shown in the table below.

Model	HY7040DS083 0-H16.8	HYB30704R3W	SC700208-1
Price	\$1.75	\$2.00	\$4.10
Rated Power	3W	3W	3W
Rated Impedance	8Ω	4Ω	8Ω
Output SPL	93dB ±3dB	116dB ±3dB	100
Total Harmonic Distortion	10%	5%	NA
Dimensions	71mm x 41mm x 17mm	70mm x 31mm x 17mm	71mm x 41mm x 25mm

Table 3.2 Comparison of Audio Output Devices

The three options for an audio output device provided here were all loudspeakers. Piezo buzzers did not make good candidates due to their lack of audio quality, so only electrodynamic transducer speakers were considered. The rated power of the speakers was 3 watts; this value was appropriate for producing the volume of sound desired. The loudness was sufficient to fill a room without requiring too much current.

The next factor taken into consideration was the impedance of the speakers. Speakers with 8Ω impedance usually exhibited better audio quality than 4Ω speakers; however, they required more current to be driven properly. Since we were using 3-watt speakers, power was not a significant concern, allowing us to make the tradeoff for improved audio quality.

All of the speakers listed had similar dimensions, so regardless of the choice made, it would not make a difference. The output SPL of all the speakers was also loud enough for our needs. Anything over 80 to 85 dB SPL was considered plenty loud enough to fill an entire room. In terms of price, the HY7040DS0830-H16.8 was less than half the price of the SC700208-1.

Additionally, the HY7040DS0830-H16.8 had a desirable frequency response curve. In contrast, the SC700208-1 did not have any frequency response curve shown on its datasheet, nor did it list total harmonic distortion. This lack of information made it an unreliable choice, as there was no supporting data to confirm that it could produce good sound.

Given this information, it seemed that the best option to choose was the HY7040DS0830-H16.8. With its low price, it offered a good frequency response, an overall low power requirement, and sufficient loudness.

3.2.3 Amplifier

Now that we had decided on a speaker, we needed to choose the design for an amplifier based on that choice. Our chosen speaker had a nominal power rating of 3 watts and an impedance of 8Ω. Because of this, we needed to select an amplifier circuit that provided 3 watts to an 8Ω load. Fortunately, many designs for this type of circuit could be found online, complete with a parts list.

Model	TDA2003	PAM8403	LM386
Amplifier Class	AB	D	AB
Output Power	10W @ 4Ω	1.8W @ 8Ω	0.25W - 0.7W @ 8Ω
Supply Voltage	8V - 18V	2.5V - 5.5V	5V - 18V
Impedance Rating	1Ω - 8Ω	4Ω - 8Ω	4Ω - 32Ω
Lifecycle	Obsolete	Active	Active

Table 3.3 Comparison of Audio Speakers

The amplifier ICs mentioned above were all commonly used in amplifier circuits. They each had the ability to drive our loudspeaker, but there were differences between them that affected our circuit. The Amplifier Class category referred to the type of circuit used to amplify the signal. Class AB amplifiers had a conduction angle between 180° and 360°. These amplifiers were very common and inexpensive but tended to be inefficient. Class D amplifiers used pulse width modulation, where the conduction angle no longer played a significant role in the amplification process. Class D amplifiers tended to be more expensive but were much more efficient and produced less heat. For a low-power application like this, it was wiser to use a Class D amplifier; the added expense was minimal at the scale of a single IC and would keep us within our power constraints without requiring a large heatsink. This led us to lean towards choosing the PAM8403.

The output power of the amplifier had a somewhat related effect on how loud the amplifier could get; though these were not directly related, thinking about

it this way was practical in our situation. The output power of the TDA2003 was 10W at 4Ω, so it could be concluded that it would be around 5W for an 8Ω load. This output would exceed the power rating of our speaker and create unsafe conditions. Although the signal could be driven lower than 3W, it risked the possibility of being turned up too high and causing a fire. The PAM8403 could drive 1.8W at 8Ω when supplied with 5V according to the datasheet. Though this was not the maximum amount our speaker was rated for, it was still enough to drive the speaker to produce a significant amount of noise. The amplifier power rating being lower than the speaker power rating was acceptable as it could not overload the circuit and cause any danger. The LM386 had an output power of 250mW to 700mW at 8Ω, depending on the input voltage. Although this could drive our speaker, the inefficiency of a Class AB amplifier limited its performance and would likely be too quiet to be acceptable for our application.

The input voltage of the TDA2003 was higher than the required supply voltage of the other two options, and it would still output too much power to our speaker, making it undesirable at this point. The PAM8403 and the LM386 could both be powered with voltages near the supply voltage for our microcontroller. This meant we could have an overall input voltage suitable for our amplifier and then use an LDO to power the microcontroller, providing a win for both options in this category.

The PAM8403 and LM386 could also both drive the 8Ω impedance of the speaker we had chosen, making them equally capable in this regard. However, factoring in the output power and amplifier class, it seemed likely that the PAM8403 was the best choice.

Lastly, the lifecycle of the TDA2003 was obsolete, meaning that it was no longer being produced. There were listings online for sale, but it was possible that these could be counterfeit or damaged. It was not worth the risk, which made the PAM8403 and LM386 both favorable in this category as well.

Given everything we had examined, it made sense to choose the PAM8403. Its higher efficiency provided more output power than the other amplifiers at a low supply voltage. It was the closest choice to matching the wattage of our speaker without exceeding it and was readily available from many vendors.

3.2.4 Auxiliary Output Jack

In order to properly create an auxiliary output jack, we needed to construct a headphone amplifier circuit. These circuits have different characteristics than the loudspeaker amplifier due to the fact that headphones have a much higher impedance than loudspeakers. This also means that we needed to choose a different chip for amplification. Below were some options available to us:

Model	LM386	TS482	LM4915
Amplifier Class	AB	AB	AB
Output Power	325mW @ 6V 8Ω	67.5mW @ 32Ω	250mW @ 32Ω
Supply Voltage	5V - 18V	2V - 5.5V	2.2V - 5.5V
Impedance Rating	4Ω - 32Ω	16Ω - 32Ω	32Ω
Lifecycle	Active	Active	Active

Table 3.4 Comparison of Output Jacks

The amplifier chips mentioned above were all potential candidates for the headphone amplifier. All of the amplifier chips listed were Class AB, meaning that the amplifier class was irrelevant since it remained the same regardless of the choice. The lifecycle was also Active for all options, so there were no concerns regarding acquisition.

This left output power, supply voltage, and impedance rating as the key considerations. Starting with the impedance rating, the ideal impedance assumed was 32Ω. The amplifier options were all capable of driving this impedance. The designs of these amplifiers varied somewhat; the TS482 had the lowest output power at 32Ω. The LM4915 was noted for driving 250mW at 32Ω, whereas some calculations indicated that the LM386 would provide approximately 80mW at 32Ω. The difference between these two options lay in the circuit design. Although the LM4915 offered better output power, the LM386 featured a much simpler circuit design, requiring only a few components to function correctly. The amplifier for the headphones did not need to be as loud as the loudspeaker, and as long as it was audible to the user, it was deemed acceptable. Therefore, it was optimal to use the LM386 for its simplicity.

3.2.5 Sensor

We needed to decide which type of sensor we wanted to use in our implementation. These sensors had different ways of transmitting their information to the MCU, some of which required certain minor circuits to function correctly.

Model	Force	Velocity	Vibration
Pins	2	4	3
Variance Type	Resistive	I2C	Digital
Price	\$3.50	\$9.97	\$6.49/5pk
Required Connection	Analog IO Pin	I2C	Digital IO Pin
Sensitivity	High	Moderate	High

Table 3.5 Comparison of Sensors

The sensors listed above are all very different from each other in the ways that they transmit the information received. The velocity sensor in particular may be complicated to implement. The velocity sensor connects to the microcontroller through I2C. This is a bit more complicated to implement in our project due to the need for 6 independent inputs. Also, velocity may not be the best variable to measure; the drum pads will not be moving a ton when someone is playing despite receiving force and vibration.

This left us with the force sensors and the vibration sensors. The vibration sensor could have been useful because we could expect some vibration when a drum pad was struck. However, the issue was that it only outputs a digital high or low signal. This could have been problematic depending on the sensitivity of the sensor and might have caused false positives. It also did not provide any detail about how hard the sensor was struck, which would have prevented the implementation of velocity sensitivity.

On the other hand, the force sensors provided an analog output, as they simply decreased in resistance as pressure was applied. This made them easy to implement, given the number of ADC channels available on the MCU. It was also very simple to integrate, as the sensor only had two pins. Measuring force appeared to be the best way to receive input because it provided a direct measurement of how hard a drum pad was struck and was less susceptible to false positives.

It should be noted that the MCU could not measure the change in resistance directly, so a voltage divider needed to be created to implement a fluctuating voltage to be applied to the ADC pins.

3.2.6 Power Supply

For this section, we needed to decide on a way to provide power to our entire system. This required us to choose an initial input voltage and then also decide how to convert it to the other voltages we needed in our circuit. Looking

back at our circuit, it was apparent that we needed both 5V DC and 3.3V DC to supply different components. We also needed to be aware of the amount of current we would be drawing.

Looking at the datasheet for our chosen microcontroller, we saw that the maximum current into the MCU was around 150mA. The amplifiers were estimated to need around 500mA for the loudspeaker; the headphone amplifier was estimated to use a much lower amount of current — for this, we said 100mA. That was probably far more than what we would actually need to drive it, but this gave us a safety net as well. The force sensors would use negligible current, and all of the indicator LEDs were estimated to use 100mA altogether. MIDI output usually only required a couple of milliamps, and a microSD card could pull up to somewhere around 160mA.

This brought the total to around 1A at 5V.

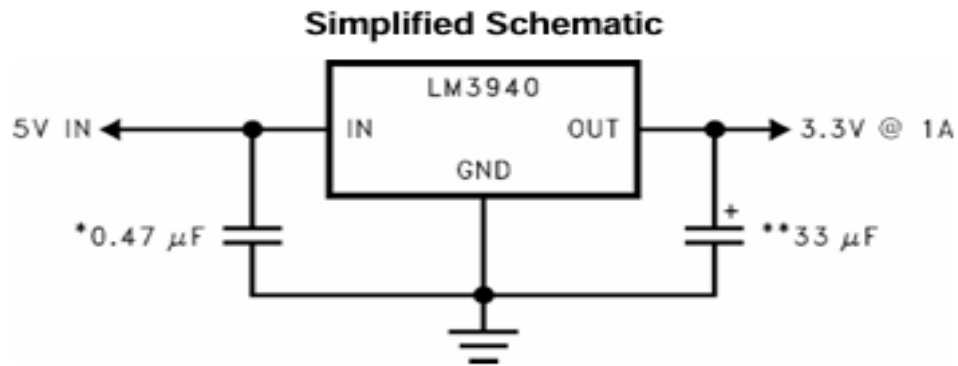
It was smartest for us to supply a main voltage equivalent to our highest required voltage. In this case, that was 5V. This was supplied by an off-the-shelf AC-DC converter. To be safe, we chose one that provided 1500mA.

The only other voltage value needed was 3.3V. We used an LDO to bring our 5V down to 3.3V without too much trouble. For this, we needed to select an LDO.

Model	BA33DD0WT	LM3940	LT1085
Input Voltage Range	3V - 25V	4.5V - 5.5V	2.55V - 30V
Output Current	2A	1A	3A
Output Voltage	3.3V	3.3V	1.25V to 28V
Price	\$3.12	\$1.62	\$10.13
Efficiency Estimate	65-85%	65-85%	60-80%

Table 3.6 Comparison of Power Supplies

This simple table aided in selecting the appropriate LDO for our application. Since the LDO outputted 3.3V, the current output was only relevant to the parts of our circuit that drew from the 3.3V line. This excluded the amplifier circuits, which resulted in a required current output much lower than that of the overall circuit. We did not need over 1A for this part to function correctly. We also received input from a stabilized 5V DC source, meaning that a wide range of input voltages was not particularly useful since our voltage was already at a steady level. This made the LM3940 appear very appealing due to its low cost. It was expected to fulfill all our needs regarding driving the 3.3V line



3.2.7 Development Environment: Languages and Repositories

The MCU of choice is the STM32 family microcontroller from STMicro. The manufacturer provides tools for developing with this controller on their website which includes an IDE and related testing software. STM32cubeMX is a companion software to the STM32cubeIDE which gives a graphical control of the components in the device. Setting up the pins of our controller can be done through this software which can help with connecting our peripherals. Initialization for our system can be generated into C code using this software. The initial configuration for embedded devices are often different due to hardware design and can be tricky to develop the proper setup procedure for different platforms. This provided software takes much of the complexity away with starting our development.

Although there is a provided IDE for our microcontroller we may also develop with VSCode or VSCodium, the open sourced variation of VSCode. Extensions may be installed to this program which contain the necessary resources to develop on the STM32 platform. Installing 'stm32-for-vscode' gives the needed functions to debug and compile code written for this family of microcontrollers. This extension may also work alongside the first party STM32cubeMX software. Code is expected to be developed in C language although C++ may also be supported.

The project behavior may have some complexity but overall each component in software can be simple enough to stick with C overall. This decision leaves most granularity of control with the programmer and unifies all aspects of the design into one language. Most of the low level code may be sourced from readily available libraries. Technologies that have existed for decades such as mounting and reading filesystems do not need to be made from scratch and are greatly discouraged in general. The C language can produce code that takes fewer cycles on the cores which can benefit the responsive feeling of the project. An emphasis on security is not necessary for this device as our use case does not involve personal or sensitive data in any way. Security is not an inherent aspect of the C language like it is with many modern languages today.

Hosting the project code throughout the duration of its development is important for all members to view the software's development. It is important to have a history of the project with tools such as 'diff' for easily visualizing changes to the source code. The repository should be secure in that only members of the project team may view, add, delete, or download these sources while it is developed. A choice of remote hosting is GitHub for how accessible it is to all members. It includes all of the important features our project needs and is historically the one most used by our team.

3.2.8 Filesystem

Software for this project is intended to perform limited file i/o operations on an external storage device. The intent is for a user to be able to load their storage on a personal computer to customize audio files for the device. A format that is widely adopted across popular operating systems should be chosen. This is the greatest restriction to our choice of filesystem. Other formats may include more advanced features although even the simplest option we are comparing with, FAT32, contains enough features for the purpose of the device. There are features of operating systems that can detect or prevent data corruption, reduce data size, or have greater file and partition capacity.

File sizes are not a restriction in our choice as we are expecting a common profile with all its content to be in the range of megabytes. The more modern formats we are comparing can have a single file as large as hundreds of gigabytes to hundreds of terabytes. The restriction of 4GB is more than enough from the FAT32 filesystem. Sparse files that contain long segments of 'empty' data are able to be reduced by modern filesystems and returning more of the flash storage back to the user. This feature could only slightly improve our device although common file sizes would range to a few hundred kilobytes in size.

Data corruption is a serious concern for any device that is meant to write and store exact data. Journaling is meant to help with recovering a device from corrupted states. The filesystem logs its contents over time to revert to an uncorrupted state. External storage devices that get removed too early is the most common cause of corruption. Without a method of dealing with corruption we must develop our own solutions for when the device receives bad data.

Case sensitivity involves whether a filesystem may handle uppercase and lowercase paths and files as distinct. Without case sensitivity we can force each directory and file to be made distinct. No path or file within may share the same name but with different capitalization. This benefits our software to not have case sensitivity as it can reduce confusion that capitalization may introduce. File and folder structure is meant to conform to a specific layout in order to be properly discovered by the device. Even though NTFS can have case sensitivity it is common for the Windows operating system to enforce naming that emulates no case sensitivity.

Format	Case Sensitive	Sparse Files	Journaling	OS Cross Compatible
FAT32	No	No	No	Yes
exFAT	No	No	Optional	Yes
NTFS	Yes	Yes	Yes	Yes*
APFS	Yes	Yes	Optional	No**

* Not all platforms support writing.

** Not all platforms support reading.

Table 3.7 Comparison of Filesystems

3.2.9 Audio Format

Audio codecs can impact data transfer time, memory usage and CPU cycles. Latency is introduced with codecs that involve additional processing such as compression. Before a codec can produce audio some formats require a range of data to look ahead before it can be decompressed. The transfer speed from our storage device can have a negative impact on responsiveness as file size increases. With higher compression the responsiveness is negatively affected by CPU load. Larger file sizes will also take a larger footprint in memory although codecs that reduce file size with compression will need additional space in memory for the decompressed data. Tradeoff for memory footprint is determined to be negligible because of this. The chosen audio formats to provide compatibility for this project are WAV for its low CPU overhead followed by Opus for a balance between CPU and data resources.

Format	Compression	CPU Processing	Size
WAV	No	Low	690 kbps
FLAC	Lossless	High	450 kbps
ALAC	Lossless	High	425 kbps
MP3	Lossy	High	160 kbps

AAC	Lossy	High	160 kbps
Opus	Lossy	Medium	80 kbps

Table 3.8 Comparison of Audio Formats

Note: Data size is approximated by 16-bit sample 44.1 kHz mono audio sources using a common high-quality configuration for each format.

4. Design Constraints & Standards

This chapter covers what we need to consider during the design process. There are certain guidelines we must follow when working with established standards to ensure compatibility between our device and third party devices that the user may connect. There are also natural constraints to the specific design context that we are in.

4.1 Industrial Standards

The following are industrial standards related to components used in our device. Following these standards allows our device to be predictable and consistent for a variety of users and use cases.

4.1.1 PCB Design Standards

To produce our PCB, we used the manufacturer JLCPCB. They were chosen due to our familiarity with the service and their past reliability. JLCPCB had certain guidelines that needed to be followed when creating a PCB design. These included using 1 - 2 oz copper for the outer layers of the board, as well as a minimum trace width of 0.127 mm. Other requirements specified that tracks had to be 0.127 mm apart from each other and 0.3 mm away from the edge of the board. Hole sizes needed to be within 0.3 mm and 3.6 mm, and vias had to be at least 0.3 mm with 0.5 mm pads.

Other than just the PCB guidelines of JLCPCB, there are other resources to consider when designing a PCB. IPC standards are relevant when designing PCBs to ensure a common understanding of their design and to make clear certain PCB design practices. We can reference IPC-2221 which has standards for generic PCB designs as well as reference to specific documents that discuss certain types of printed circuit structures in depth. IPC-2221 recommends paying attention to certain details in order to design for reliability. This includes choosing materials that are able to tolerate temperature, humidity, and vibration. Also choosing materials with structural strength and that properly conduct electricity and retain signal integrity.

The standard discussed considerations regarding the connection of components on the board. Trace width and trace spacing were addressed in the document in the context of selecting the correct trace sizes to handle the current required by the circuit. The dielectric material of the PCB was also important in controlling impedance and maintaining the integrity of high-speed signals. It mentioned practices for properly grounding the circuit and reducing electromagnetic interference (EMI). Further reading in the document highlighted thermal management and the necessity of using heatsinks for heat dissipation, especially when working with high-power components, as heat buildup could reduce the reliability and longevity of the device.

Related to all of these topics was component placement, which affected heat dissipation and EMI. A specific example of component placement that would be used later in this project was placing capacitors close to the voltage input pins of a microcontroller to ensure voltage stability. Positioning connected components near each other reduced potential interference issues encountered by the device and decreased the overall complexity of the PCB design, which could lower layer requirements and optimize costs.

IPC-2221 discussed the topics of mechanical design and manufacturing constraints of the PCB. It was important to design a board in such a way that it could be manufactured by the PCB manufacturer. Relevant constraints included minimum trace width, minimum trace spacing, minimum hole size, and the minimum distance between vias, among other things. These constraints were critical to pay attention to while designing the circuit because noticing them only after the design was completed and ready to be sent to the manufacturer could mean needing to go back and redesign major parts of the PCB.

Some final points described by IPC-2221 included environmental factors such as choosing materials that complied with environmental regulations, for example, RoHS compliance. This also applied to selecting durable materials and coatings that would last in harsh conditions. This consideration depended on the specific application of the device but was essential to take into account. Lastly, it described the importance of thinking ahead in terms of component placement regarding the assembly process as well as the troubleshooting process to ensure that these processes were not complicated and that components could be placed on the board without issues. Additionally, it emphasized creating a detailed BOM and including design notes, such as critical tolerances and requirements, to inform others who might have to work with the PCB design.

In conclusion, IPC-2221 was useful for us to reference as we went through the process of designing our PCB to ensure that we made decisions that would facilitate any future processes. We paid attention to the design requirements of JLCPCB to ensure they could manufacture our circuit board while selecting component placements that made assembly and troubleshooting of our PCB easier. Additionally, we made sure to choose reliable components to reduce the likelihood of our PCB failing in the future.

4.1.2 SPI Standard

The Serial Peripheral Interface (SPI) specification defined a form of communication between devices in which one acted as a master to multiple slave devices. This utilized a 4-wire connection between devices, with the pins functioning as the Chip Select (CS) pin, the Serial Clock (SCK, SCLK) pin, the Master-In-Serial-Out (MISO) pin, and the Master-Out-Serial-In (MOSI) pin. The functionalities of these pins and the usefulness of the SPI standard were discussed in this section.

SPI was a full duplex communication standard that was employed in our project to enable communication between the microcontroller and the microSD card that stored our sound files. It was a synchronous protocol that timed data transfers with clock pulses on the SCK pin. We went through the pins to explain the purpose and significance of each one.

The Serial Clock (SCK, SCLK) pin provided a clock signal to all devices connected to the master device to keep data transfer synchronized. The clock signal was sourced from the master device and sent as a square wave on a shared line. Devices following this clock signal conducted data transfers on either the rising edge or falling edge of the signal, depending on their specific properties.

The Chip Select or Slave Select (CS/SS) pin was used to choose which slave device the master device communicated with. Only one slave device could communicate with the master device at a time; otherwise, there would be communication issues. This pin was an active low pin, meaning that to enable a slave device for communication, the pin had to be driven low.

The Master-Out-Slave-In (MOSI) pin was used to transmit data from the master device to the slave devices. The line between all devices on the MOSI pin was shared; however, it only affected the slave device currently enabled by using the CS pin. The Master-In-Slave-Out (MISO) pin functioned as the pin that slave devices used to transmit data to the master device. This line was also shared among all devices. Only one slave device communicated over this line at a time, depending on which slave device was enabled. This line had no effect on disabled slave devices.

With these connections made, the master device began communications with the slave devices. This started with the master outputting the clock signal. The clock signal used depended on what the slave and master devices were capable of handling. The master device should have output a clock signal with a frequency within its own operating range but not exceeding the lowest maximum frequency that any one slave device could handle. It also needed to be greater than the lowest frequency that any slave device could handle. Once the master device was outputting a proper clock signal, the system then needed to drive the CS pin of any single slave device low to enable it for communication. After the correct slave device was enabled, the devices could begin sending digital bits across the MISO and MOSI pins according to the speed of the clock cycle. The bits were transmitted and received in series relative to each other and were read

in the order they were received. In SPI, there were no stop or start bits, and the data stream was continuous. The full duplex nature of SPI also allowed both devices to send and receive data simultaneously, making it a capable communication standard for many devices. The actual content of the messages was not dictated by the SPI standard; that depended on the specific devices being used and what data they expected to send and receive. The SPI standard simply defined the overall structure and format of the connections.

4.1.3 The MIDI Standard

MIDI (or Musical Instrument Digital Interface) is a music communication standard that has been around since 1983. To this day it is widely used and it can be found almost everywhere that you would need to digitally transmit note values and velocity values as well as numerous other functionalities. MIDI is an extremely capable communication standard that is also extremely simple to implement.

MIDI data is usually transferred through a cable with a male 5-pin DIN connector on either end. Though this connector has 5 pins, MIDI only uses 3 pins to transmit musical data. The innermost 3 pins are used to communicate data. In the case of MIDI output the central pin, pin 2, is connected to ground. Pin 4 is then connected to a +5V DC source with a 220 Ω resistor in series with it. Lastly Pin 5 is connected to the UART Tx pin of a microcontroller and sends digital signals across the cable in order to transmit information. The other two pins (1 and 3) are unused for the MIDI standard but can be used by other devices for custom purposes wherever necessary.

Messages in MIDI are sent over UART at a baud rate of 31250 Hz. The messages consist of 8-bit segments transmitted over the UART line to convey different properties about the note including when it begins, when it ends, the velocity of the note, what MIDI channel it is being transmitted on, and other variables such as MIDI Control Change messages that may be controlled by different knobs or faders on the MIDI controller being used by the user to change the characteristics of the sound. For our purposes, we are concerned with how to start a MIDI note, end the MIDI note, define the correct note value, and define the velocity with which the MIDI note should be played.

MIDI messages consist of different bytes which serve different purposes. The first byte in a MIDI message is the Status byte. This byte defines the type of message being transmitted. Following the Status byte are Data bytes which convey further information about the specific message defined by the Status byte. Once a Status byte is sent, the system will assume all subsequent Data bytes will be in the context of the most recent Status byte received. This means that if you are transmitting only Note messages you do not need to resend the Status byte for each note.

To send NOTE ON and NOTE OFF messages on MIDI channel 1, a Status byte must be sent as the following: 1001 0000. The most-significant 4 bits are what define the message as a NOTE ON status, and the least-significant 4 bits define the use of MIDI channel 1. MIDI channel 1 is the most commonly used

channel for general purposes, other channels have uses when using multiple MIDI instruments in the same system.

The next bytes in our NOTE ON message define the pitch and velocity of our note. The pitch byte contains a most-significant 0 bit followed by 7 bits to define the note value. MIDI note values range from 0-127, giving plenty more octaves than are available on conventional pianos. Referring to the earlier discussion in section 3.1.5.3, the note value for a kick drum would be 36. In this case the pitch data byte we would want to send would be as follows: 0010 0100. This would tell the system to begin a MIDI note of value 36.

The last byte in this message would be the velocity byte. This follows the same structure as the pitch data byte except the least-significant 7 bits define the velocity value and not the pitch. MIDI velocity values also follow a 0-127 scale where 0 is the least – or quietest – velocity, and 127 is the most -- or loudest -- velocity. For example if we wanted to transmit a velocity value of 100, we would send a message as follows 0110 0100.

This completes the entire NOTE ON message providing the receiving system with the information it needs to begin a MIDI note. The message in its entirety would be sent as 0x90 0x24 0x64. This simple message would be sent over the UART channel at 31250 baud to our MIDI output jack where it would be transmitted to a computer or other MIDI receiving device to begin a kick drum sound. Notice that Status byte messages always have a most-significant bit of 1, while Data byte messages always have a most-significant byte of 0.

The next step is to send a NOTE OFF message to end the MIDI note we just began. If we do not send a NOTE OFF message, the system will continue to assume the note is being “held down” similar to a key on a piano. This will create numerous issues for a person trying to record the MIDI information. In order to send a NOTE OFF message we need to send another Status byte to change our MIDI Running Status from NOTE ON to NOTE OFF. The NOTE OFF Status byte is as follows: 1000 0000. In this case the most-significant 4 bits define the message as NOTE OFF and the least-significant 4 bits define the message as being on MIDI channel 1.

After this follows another two Data byte messages. These messages define the pitch value of the note being turned off as well as the release velocity of that note respectively. The pitch message would be the same as it was for our NOTE ON message as a 0010 0100 or 0x24. The release velocity message is generally set to 0. Release velocity is a rarely used value for most MIDI devices, and for our purposes will be set to 0. So for this byte it is simply transmitted as 0000 0000 or 0x00.

The entire NOTE OFF message would then be seen as the sequence 0x80 0x24 0x00. This message would be sent momentarily after the NOTE ON message and the entire pair of messages would allow us to properly transmit a MIDI note to a computer or other device where it will be recorded and not hang because it is not properly closed at the end of the note’s duration.

This is about as advanced as our usage of the MIDI standard will get, however the standard itself has far more capability than just this. More advanced MIDI instruments would be able to transmit multiple notes on multiple channels

as well as Pitch Bend messages which continuously change and affect the pitch of all other notes in ways according to the receiving instrument. There is also something known as MPE or MIDI Polyphonic Expression which allows multiple notes that are being played simultaneously to be pitch bent or modulated all individually from one another. It is clear why MIDI as an open standard has stood the test of time and is still found everywhere in the digital music space today as it is an extremely capable standard while also being surprisingly simple. This is what makes it such a good choice for us to use in transmitting our note information.

4.2 Practical Constraints

The constraints discussed here relate to the context of our design process, and the limitations of our current situation in terms of the world we are in and the deadlines we have been given, as well as the limitations of our abilities.

4.2.1 Time

For this project one of the main and most restrictive constraints we faced is the constraint of time. This project comes with a strict deadline by which we had to meet certain requirements. In this time we were able to synthesize an entire project. A specific issue that was faced was finding a project that was complex enough to create an extensive report on, but also simple enough that it would be feasible within the semester. A way that we adapted to this was to find a base idea and expand upon it until it is of sufficient complexity.

Time became a very limiting constraint especially in regards to shipping times on certain products. When we learned that we needed something new or needed to purchase a replacement part, we had to wait a significant number of days before we received the part and were able to continue making progress. This is able to be remedied by paying for express shipping or faster shipping options so that we receive parts earlier, however this shifts the constraint over to cost which will be discussed later.

In the first half of the Senior Design course, a lot of time was spent researching how to do certain things and the methods to do them correctly. This took up a significant amount of the time that we spent on the project as it was a required step that could not be worked on later or pushed to the side. When we had an idea to do a certain thing a certain way, we then needed to research if our idea was possible and could be executed in the way that we thought it could. There was nothing particularly guiding our research except ourselves so the problem could sometimes be less about where to find the correct information and more so what the correct information even was. We needed to deep-dive into a list of different topics during our time researching to verify that the ideas we had were even along the correct line of thinking. If this were not an issue that was encountered during our research then we would be able to come to the

conclusion of how to do something much faster and we would be able to explore options much more quickly.

What helped in this area was reading several forum posts online as well as seeing videos of people doing similar things to what we wanted to do and hearing what they said were difficult issues they faced and how to overcome them. The internet was an extremely valuable resource for us to get research done.

Other than that, time is especially a constraint because although we budgeted a certain amount of time for something, we encountered problems with it during our work that made it take up a significantly larger portion of our time. Because of this we have had to give ourselves a safety net in terms of time spent researching or working on something to make sure that if a problem was encountered, there was some time set aside to remedy it while not causing other issues in our schedule.

One last issue that played a role in both the constraints of cost and is the PCB design. Getting the PCB produced takes time for both the production and shipping of it. For this reason it is generally beneficial to avoid having to get the PCB remade. If components needed to be changed or moved around on the board it would need to be remade, so we had put off having the PCB produced just yet until initial testing was over to reduce the number of iterations of the board that would be needed.

Overall, time is one of the prevalent constraints with this project and was the factor limiting us the most. Regardless, we have been able to make ample progress within the timeframe provided and are on track to have our researched design completed this semester. With the information we now have on what time estimates are realistic for which processes, it should be easier for us to determine our goals for the next semester

4.2.2 Economic

This project was funded by the members of the group and did not receive any external funding. Due to this fact, decisions had to be made carefully when choosing how to spend money to minimize the economic impacts of the project. Although a set cost was established for the bill of materials (BOM) for assembling the project, this did not take into account the costs of shipping, development tools, and potential mistakes made during development.

A specific example related to ordering parts from vendors such as mouser.com, digikey.com, or taydaelectronics.com was highlighted. When ordering from these vendors, after selecting the necessary components, a flat shipping cost was typically charged to the location, regardless of the order size unless the weight was particularly high. Consequently, the most economical option was to order all needed components in one order. However, once parts were ordered and tested or assembled in the project, it became apparent that additional components were needed beyond the initial estimates. Sometimes, these extra components were only a few small items costing less than a dollar in

total. Despite the low cost of these extra components, the shipping cost remained the same as during the first large order and would need to be paid again.

Such occurrences could happen multiple times, depending on how thoroughly research for that part of the project was conducted. To reduce the number of repeat orders, the team attempted to research each part of the project thoroughly and double-check that the bill of materials included everything required. This extended to ensuring that the types of components purchased had the correct specifications, determined by checking the datasheet and corroborating it with prior research to ensure compatibility with the rest of the project.

The economic constraint of the project was heavily linked to the time constraint as well. Conducting extra research and double-checking processes took additional time, which was a limited resource. This also included ensuring that the cost of the bill of materials met the budgeting goal. However, failing to do extra research could cost more time. As mentioned earlier, if additional parts needed to be ordered, that not only incurred extra costs that could have been avoided had everything been ordered together, but it also resulted in longer wait times for the new parts to arrive and be tested and incorporated into the project. The time expense required to test and integrate those new parts influenced decisions regarding the need for express shipping. It became clear that the constraints of time and economy were interconnected, necessitating thorough consideration in related decision-making.

Other economic issues faced included the risk of exceeding the projected budget for the bill of materials. Avoiding this was possible by cross-referencing multiple vendors to find the lowest prices for certain components. These decisions had to be weighed against the costs of placing multiple orders from different vendors, as shipping costs increased with more orders. Certain vendors were the only sources for specific components; in such cases, it was wise to purchase additional components from them as well, considering that shipping fees would already apply.

Overall, there were various strategies to consider for minimizing project costs, yet mistakes could still occur and be recognized only in hindsight. It was crucial to remain frugal in this situation, as funding for the project was provided solely by the group members. Time and economic factors interacted with one another and needed to be considered accordingly to ensure the best use of both time and money.

4.2.3 Manufacturability

Another constraint we faced while working on and researching this project was manufacturability. Our project needed to be something that could be manufactured with ease. This did not only refer to a PCB that could be produced by our manufacturer, but also to a design that we could easily create ourselves without too much struggle. If the project required an excessive amount of time for us to produce, or demanded tools that we did not have consistent access to, it

would not be deemed easily manufacturable and might not even have been feasible to produce.

We aimed to avoid situations like this by ensuring that our research and design could be executed with relative ease and did not require exotic production methods. This was accomplished by thinking ahead about our ideas and making sure they were realistic and within the scope of the semester. This tied back into the previous constraints of time and economics. Keeping our design simple benefited manufacturability, as well as time and economic factors. If our design was more manufacturable, it was also likely to be a more economical solution and required less time to produce. Our ideas needed to remain within what was reasonable for a team of three people to accomplish in a semester.

Although this was true, it was still possible for us to design something that could not be feasibly manufactured, even while staying within our economic and time constraints. This mistake was particularly easy to make if we did not conduct proper research on a topic or falsely assumed something that we should not have. We needed to be cautious to ensure that we thoroughly researched all of our topics while also considering how to execute them. The issue was that failing to do so could negatively impact our time and economic constraints. As discussed previously, additional research could be costly in terms of both time and economics. Therefore, ensuring manufacturability would take time regardless and could encroach upon the allotted research time needed to confirm the economic feasibility of our ideas. However, ensuring manufacturability could also help confirm that our project was economical and reduce the time spent on research and production. This highlighted that a significant factor in determining our overall efficiency was ensuring that we conducted the correct research and learned the necessary information sooner rather than later.

4.2.4 Health & Safety

Health and safety were lesser concerns for our product but still needed consideration. We aimed to ensure that our product did not pose any hazards to users. This meant designing a product that was safe to use and would not cause any danger to the user in any way. Possible dangers associated with our product included its shape, the risk of electricity, or the use of chemicals that were toxic to humans or could pose long-term health risks.

Designing for safety was not difficult; the risk of electricity was practically negligible because we used an AC/DC converter to supply power to the main board. Thus, there was no AC power that could be dangerous to the user after the converter. Furthermore, we did not plan to use any hazardous materials, such as paints or glues, which would pose issues for users. Lastly, the shape of the product was also not a concern. We ensured that any sharp corners were sanded down and that we utilized finished wood or another smooth material to construct the main enclosure.

4.2.5 Ethical

Another constraint of lesser concern was the constraint of ethics. The constraint of ethics was concerned with how the product would affect the end user or other people in society. Our specific product was seen as an ethically beneficial tool, as it allowed musicians on a limited budget to produce music more easily and with a better workflow than they would have otherwise achieved. This could have helped them finish projects that they otherwise wouldn't have completed, allowing their voices and music to be heard by a wider audience.

This point was considered ethically beneficial as it helped individuals achieve their aspirations. Ethics also extended to the content of their music and the messages it promoted. Generally, music did not promote an ethically negative message; however, if it did, that was beyond our control as authors and was not something we could remedy. This aspect was also discussed in the political section of this chapter. If users chose to create music that promoted a political message or agenda using our product as a tool in their music production process, that decision was entirely theirs and beyond our control.

4.2.6 Political

Political constraints did not have a significant effect on our project. There were not necessarily any political applications that it could have been used for. There were also no political intentions behind the design of any part of the project, and it was intended simply as a tool for amateur musicians to aid them in creating music. It was possible that it could have been used by musicians in the creation of songs that might push a political agenda or advocate for change, but this choice of use was entirely a decision of the end user and did not reflect the beliefs of any of the individuals who worked on the project. None of the authors of the project intended for it to be used in any political way, nor did we design it for that purpose.

4.2.7 Environmental

The environmental constraint concerned the use case of the product and what was feasible given the real-world conditions it would be subjected to. Our project was intended for indoor use at room temperature. Water was generally not a concern, even though a spill could damage the project; however, that was not something we intentionally designed for. One aspect we needed to be concerned with was our project's ability to withstand shock and vibration. Different components of the drum kit were going to be struck with drumsticks at a decent velocity. We did not design the project with the expectation that users would hit it as hard as they would at an actual live event, but rather as hard as they might during a relaxed practice setting.

The drum pad enclosures, which would be developed later, needed to be made with this in mind: they had to survive repeated hits while still providing reliable readings to our central processing unit. One factor that helped this central hub avoid excessive force was that the drum kit components were attached to it using only wires to transmit the data received from them. Not much

force and minimal vibration were transferred to this central hub, which remained within the specifications that the 3.5mm cable standard could tolerate.

Lastly, the foot pedals used with this system were designed for being pressed by the user's foot. The only concern was that the kick pedal might be pressed hard if the user applied the kind of force typically used at a live event. We hoped that the fact the kick drum sound effect played at a constant volume and was not velocity sensitive would help reduce the likelihood of this occurring.

4.2.8 Sustainability

Sustainability played a significant role in the previous section regarding the product's durability. The project was designed to ensure that it could withstand typical usage conditions effectively. The drum pads and cymbal enclosures used were capable of enduring strikes from drumsticks, provided they were not hit excessively hard. The design of the pads aimed to prevent damage to the sensors from regular use.

The cables utilized in the project were also expected to withstand multiple plug and unplug cycles, as long as this was done correctly. The connectors holding these cables were securely panel-mounted to the structure of the central processing unit's enclosure. This choice of cables and connectors, as opposed to hardwiring the components, was made to mitigate risks in high-stress situations. For example, if a component needed to be disconnected from the central hub due to an incident like something falling over, the components would disconnect safely, avoiding damage to hardwired connections and preventing separation from the PCB.

Additionally, it was essential to secure the cables connected to the force sensors to the enclosures housing them. This was necessary to ensure that any excessive force applied to those cables would be transferred to the enclosure rather than the sensors. The force sensors themselves were delicate and at risk of tearing or breaking if subjected to heavy stress. Damage to the sensors could lead to inaccurate readings and long-term issues, ultimately affecting the sustainability and reliability of the system.

5. Comparison of ChatGPT or Similar

ChatGPT was a great source and helped speed up some project tasks such as brainstorming ideas, summarizing, classifying, and theming primary research insights. However, the content generated by ChatGPT should always have been double-checked for accuracy.

Case Study 1:

Question "How can I optimize the power consumption of an electric drum?"

"Optimizing the power consumption of an electric drum involves improving circuit efficiency, component selection, and software optimization. Below are actionable steps:

1 Efficient Circuit Design

- **Use Low-Power Microcontrollers:** Select microcontrollers optimized for low power, such as ARM Cortex-M0/M4 or low-power MSP430 chips. Ensure they have sleep modes for when the system is idle.
- **Optimize Power Rails:** Use efficient DC-DC converters instead of linear regulators to reduce power loss in voltage regulation.
- **Component Selection:** Choose ICs with low quiescent currents and MOSFETs with low on-resistance to minimize losses.
- **Minimize Active Components:** Reduce the number of continuously powered components. Use multiplexers to share resources like ADC channels or GPIO pins.

2 Power Management Techniques

- **Implement Sleep Modes:** Design the system to sleep during inactivity and wake on specific triggers (e.g., striking the drum).
- **Dynamic Voltage and Frequency Scaling (DVFS):** Adjust the system clock and voltage dynamically based on performance requirements.
- **Control Peripherals:** Power down or disable unused peripherals when not in use.

3 LED and Display Efficiency

- **Use Low-Power LEDs:** If the drum uses LEDs for feedback or aesthetic purposes, opt for high-efficiency LEDs with low current requirements.
- **Optimize Display Power:** For any display interface, use energy-efficient technologies like e-paper or low-power OLEDs. Reduce refresh rates or dim the display during inactivity.

4 Sensor Optimization

- **Optimize Piezo Sensors:** Piezoelectric pickups can be energy-intensive when amplified. Use efficient op-amp designs or digital sensors with low power demands.
- **Threshold-Based Wake-Up:** Ensure the sensors only trigger processing when a significant event occurs, avoiding unnecessary processing.

5 Firmware Efficiency

- **Efficient Signal Processing:** Optimize algorithms (e.g., FFTs for sound analysis) to minimize computational overhead.
- **Event-Driven Programming:** Design the firmware to process events only when necessary instead of polling continuously.
- **Code Profiling:** Profile your code to identify and reduce energy-intensive tasks.

6 Testing and Measurement

- **Power Profiling:** Use tools like the Joulescope or an oscilloscope to measure power consumption during operation and optimize bottlenecks.
- **Iterative Design:** Test early prototypes and make incremental improvements to the design.

7 Housing Design Impact

- **Thermal Management:** Optimize the housing for heat dissipation to ensure components work efficiently without requiring additional power-hungry cooling systems.
- **Insulation:** Use materials that reduce power loss in environmental extremes.” (Chatgpt 2024)

In this case Chatgpt provided us with a more complete and organized answer that would satisfy our goal of the power consumption limit combining efficient circuit design, advanced power management, optimized firmware, and smart housing design which can largely reduce the power consumption of our electric drum, ensuring a good performance.

Google searches are more general and may not exactly refer to what we are looking for, requiring more time to search and verify the information and sometimes the information can be irrelevant to our question, but it provides us with different sources and websites of real world experiences that others share. Google’s answer for the same question gives us a list of websites about the power usage of different existing electric drums.

Asking gemini google which is the AI-powered from Google using the latest AI technology from google the same question.

“Here are some key considerations for optimizing power consumption when designing an electric drum kit:

1 Component Selection

- **Microcontrollers:** Choose low-power microcontrollers like those from the ARM Cortex-M series. These offer a good balance of performance and energy efficiency.
- **Sensors:** Opt for sensors with low power consumption, such as capacitive touch sensors or piezo sensors.
- **Power Amplifiers:** Select efficient Class-D amplifiers for audio output. These amplifiers convert electrical energy into sound with minimal power loss.
- **Display:** If a display is necessary, consider using low-power OLED or e-ink displays.

2 Power Management

- **Power Supply:** Design a power supply with high efficiency to minimize power loss during conversion.
- **Battery Management:** If battery-powered, implement efficient battery management systems to maximize battery life.
- **Power Switching:** Use power switches to selectively power down components when not in use.
- **Low-Power Modes:** Utilize low-power modes for the microcontroller and other components during idle periods.

3 Circuit Design

- **Minimize Current Leakage:** Pay attention to circuit design to reduce current leakage, especially in analog circuits.
- **Optimize Signal Routing:** Minimize the length of signal paths to reduce power consumption.
- **Use Efficient Digital Logic:** Employ low-power digital logic design techniques.
- **Software Optimization:**
- **Efficient Algorithms:** Use efficient algorithms to minimize processing power.
- **Task Scheduling:** Optimize task scheduling to reduce unnecessary power consumption.
- **Power-Saving Modes:** Implement power-saving modes in the firmware to reduce power consumption during idle periods.

4 Additional Considerations

- **Wireless Technology:** If wireless connectivity is required, choose low-power wireless protocols like Bluetooth Low Energy (BLE).
- **Environmental Factors:** Consider the operating environment and temperature range. High temperatures can increase power consumption.
- **User Interface:** Design a user interface that minimizes power consumption, for example, by using touch sensors instead of mechanical buttons.

By carefully considering these factors, you can design an electric drum kit that is both efficient and enjoyable to play.” (gemini.google.com 2024).

Compared to the google search engine, gemini google gives a more direct answer to our question, but focusing on different aspects than Chatgpt. Both systems provide important answers, but Chatgpt's concentrates upon the technicity of reducing power consumption through design and firmware optimization.

Case Study 2:

“What are the best microcontrollers for an electric drum project?”

“When selecting a microcontroller (MCU) for an electric drum project like your **OmniDrum**, consider factors such as real-time performance, input/output (I/O) requirements, audio processing capability, and power efficiency. Here are some of the best microcontroller options based on these criteria:

1. ESP32

Why it's great:

- Dual-core processor (240 MHz) for handling multiple tasks, such as input processing and audio output.
- Integrated Wi-Fi and Bluetooth for wireless connectivity (e.g., MIDI over Bluetooth).
- Plenty of GPIO pins, including ADC for reading piezo sensors or trigger inputs.
- Low power consumption in deep sleep modes.

Downside: Limited RAM (520 KB SRAM) may require optimization for complex audio tasks.

Best for: Wireless MIDI drums, basic audio synthesis, and cost-efficient designs.

2. STM32 (e.g., STM32F4/F7 series)

Why it's great:

- High-performance ARM Cortex-M processors (e.g., Cortex-M4 or Cortex-M7).
- Built-in DSP (Digital Signal Processing) for audio synthesis and filtering.
- Extensive GPIO and support for ADC/DAC and high-speed peripherals.
- Low-power modes available for efficient designs.
- **Downside:** Slightly steeper learning curve due to feature richness.
- **Best for:** Advanced audio processing and real-time MIDI implementations.

3. Teensy 4.1

Why it's great:

- ARM Cortex-M7 (600 MHz), extremely powerful for audio applications.
- Built-in support for the **Teensy Audio Library**, making audio processing, synthesis, and effects easy.
- Ample GPIO, including high-resolution ADC and DAC support.
- Compact form factor and easy to integrate.

Downside: Relatively higher cost.

Best for: High-performance audio drums with complex effects and real-time synthesis.

4. Raspberry Pi Pico

Why it's great:

- Dual-core ARM Cortex-M0+ processor (133 MHz).
- Affordable and readily available.
- Support for MicroPython or C/C++ for rapid prototyping.
- Decent GPIO and peripheral support for drum sensors and MIDI interfaces.

Downside: Less suitable for heavy-duty audio processing.

Best for: Budget-friendly designs and simpler electric drum setups.

5. Nordic nRF52 Series

Why it's great:

- Built-in Bluetooth Low Energy (BLE) for wireless MIDI.
- Ultra-low power consumption.
- Decent processing power for input handling and MIDI.

Downside: Limited audio processing capabilities compared to STM32 or Teensy.

Best for: Wireless, battery-powered MIDI drum kits.

Comparison Table:

Microcontroller	Processing Power	I/O Pins	Audio Features	Connectivity	Power Efficiency
ESP32	Medium	High	Basic	Wi-Fi, Bluetooth	Medium
STM32 (F4/F7)	High	Very High	Advanced	Optional (via modules)	High
Teensy 4.1	Very High	High	Advanced	Optional (via modules)	Medium
Raspberry Pi Pico	Medium	High	Basic	None	High
Nordic nRF52 Series	Medium	Medium	Limited	Bluetooth Low Energy	Very High

”

(ChatGPT, 2024)

In this case ChatGPT prides us with a list of different microcontrollers giving details about the strengths of each one of them depending on the project priorities, but again this requires a deeper research to learn more about each proposed MCU. After searching in different websites and learning about the properties of different Microcontrollers, **STM32G473R LQFP64** was chosen to be the best choice for our OmniDrum due to its balanced capabilities and cost ensuring the reliability and scalability for the project.

Case Study 3:

“What sensors are ideal for detecting drum hits accurately?”

“Detecting drum hits accurately in an electronic drum system like your **OmniDrum** typically requires sensors that can reliably capture force, vibration, or changes in pressure. Here are the most suitable types of sensors:

1. Piezoelectric Sensors

How They Work: Piezoelectric sensors generate a voltage when deformed or subjected to vibration. They are highly sensitive to impacts, making them the most common choice for electronic drums.

Advantages:

- High sensitivity to vibration and force.
- Compact and lightweight.
- Low power consumption.

Use Case: Ideal for detecting hits on drum heads, cymbals, and pads.

2. Force-Sensitive Resistors (FSRs)

How They Work: FSRs change resistance based on the force applied to them.

Advantages:

- Can differentiate between soft and hard hits.
- Easy to integrate and cost-effective.

Use Case: Suitable for detecting pressure and softer touches, like rim shots or light drum rolls.

3. Capacitive Touch Sensors

How They Work: These sensors detect changes in capacitance caused by proximity or physical contact.

Advantages:

- Ideal for very light touch-sensitive inputs.
- Can be used for multi-zone detection.

Use Case: Good for integrating extra control zones or surface touches (e.g., triggering effects).

4. Laser or Optical Sensors

How They Work: Infrared or laser-based sensors detect the position and speed of a striking object.

Advantages:

- Highly accurate.
- Can measure velocity for dynamic response.

Use Case: Can be used in advanced setups where drumstick motion needs to be tracked.

5. MEMS Accelerometers

How They Work: Measure acceleration caused by vibration or impact.

Advantages:

- Compact and lightweight.
- Can provide velocity and impact angle data.

Use Case: Useful for unique designs or multi-zone drums.

Recommended Approach for OmniDrum

For the best accuracy and responsiveness:

- **Primary Sensor:** Use **piezoelectric sensors** for detecting drum hits and differentiating between light and hard strikes.
- **Supplementary Sensor:** Add **FSRs** to enhance sensitivity for softer hits or pressure-based input.
- **Advanced Features:** Consider **optical sensors** for tracking motion or integrating with dynamic visual feedback.”(ChatGPT, 2024)

ChatGPT provided us with a great selection of the sensors that can be used for our OmniDrum explaining the use and the advantages of each case. Chatgpt suggests that we use primarily piezoelectric sensors. However, they may be more susceptible to environmental noise and crosstalk from other parts of the drum. After further research, we found that the force sensors are more ideal for

our project due to their ability to minimize the unwanted noise and crosstalk from other parts of the drum.

6. Hardware Design

This chapter covers the hardware design of our project. These topics range from the design of the PCB and why certain circuit components were chosen and placed how they are. It also covers the design of the PCB housing and how it will be adapted to connect to the rest of the peripherals.

6.1 Main Power Supply Circuit

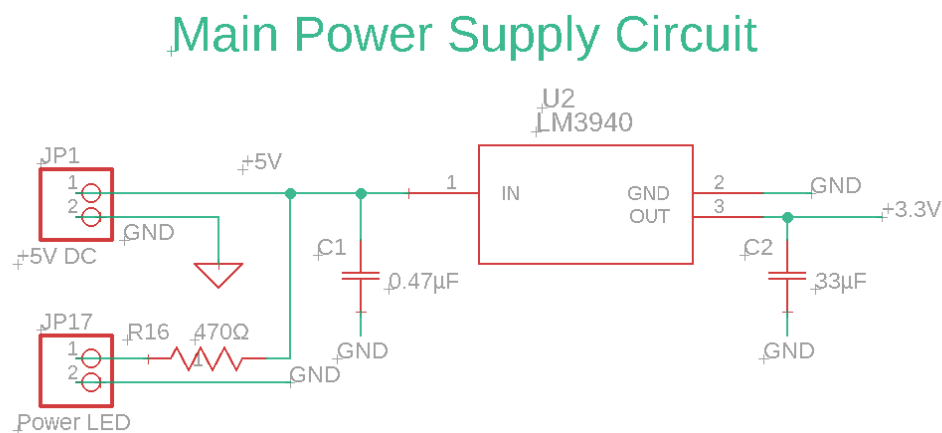


Figure 6.1 Main Power Supply Circuit Schematic
Image Source: Author

Shown here is the schematic for the Main Power Supply Circuit. The power will be provided at JP1 from a 5V AC/DC power supply plugged into a barrel jack which will be mounted on the outside panel of the unit. This will form our +5V line and our GND. The +5V line then goes down through a 470Ω resistor out to JP17. JP17 will power an LED that will be mounted near the DC barrel jack to indicate that the device is being powered. The +5V line will then go into the LM3940 LDO regulator which will output a voltage of 3.3V to power the MCU and other components. The input line of the LM3940 has a 0.47μF capacitor connected to it and ground as was recommended in the datasheet, there is also a 33μF capacitor on the output connected to ground. The 33μF capacitor is electrolytic with the negative side connected to ground, it was also recommended to have an ESR rating within a specific range, the ESR of this capacitor is 420mΩ.

6.2 STM32G4 Power Supply

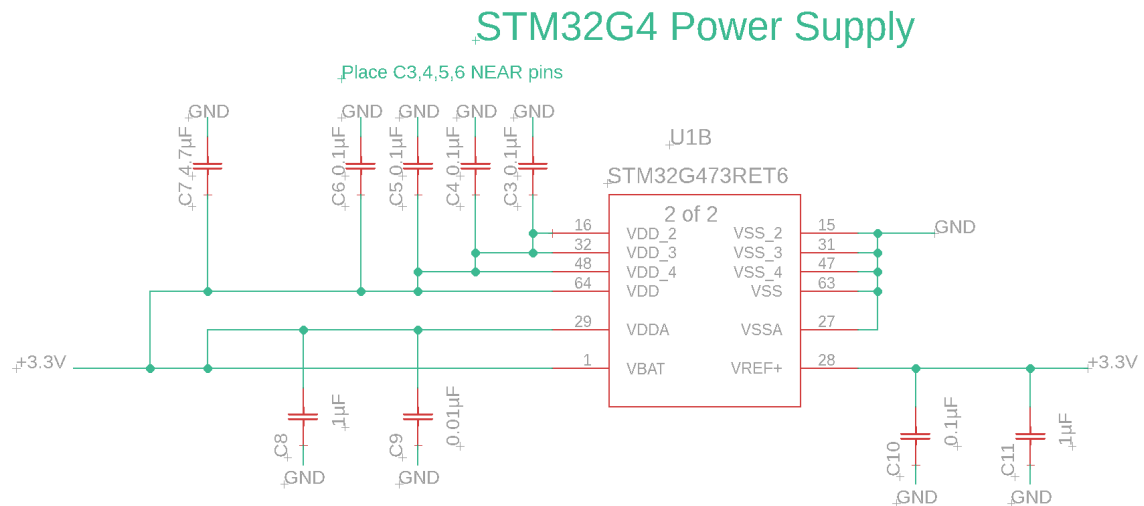


Figure 6.2 STM32G4 Schematic

Image Source: Author

This circuit shows the power supply scheme for the STM32G473RET6 microcontroller. The schematic splits the microcontroller into two separate parts: one for the power supply pins, and the other for the IO pins. Above is the part for the power supply pins. The input voltage for the microcontroller is +3.3V. Several capacitors are shown connected to ground around many of the pins; this practice was recommended in the datasheet for the microcontroller in order to improve stability. It should be noted that capacitors C3,4,5,6 are explicitly recommended to be placed as close to their respective pins as possible for the best results. Due to this, the components shown here were specifically chosen to be SMD/SMT components. The rest of the components on the board are through-hole components due to simplicity of removing and replacing through-hole components when testing and fabricating. Through-hole components however would not have been able to fit as closely to the pins as desired. The capacitors here are all of the 1206 package size.

The VDD pins provide power to the main internals of the STM32G473RET6 while that VDDA pin provides power specifically to the analog portions of the microcontroller. The VDDA pins were given a different series of capacitors for stabilization, but receive voltage from the same +3.3V line. The VREF+ provides an external voltage reference for the analog circuits. There is an internal voltage reference for the analog circuits as well, however it is best practice to power this one. The VBAT pin was recommended to be given direct +3.3V without any capacitors required to be near the pins. The VSS and VSSA pins are all connected directly to ground.

6.3 Sensor Array

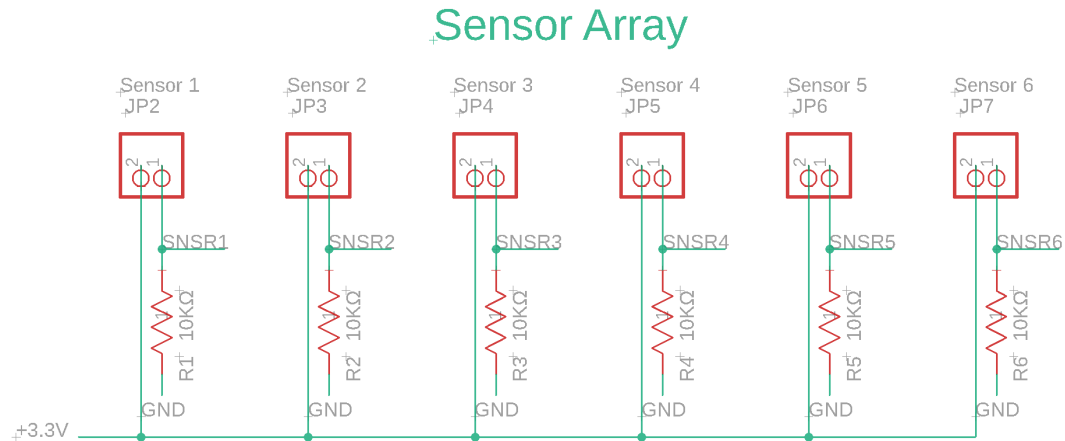


Figure 6.3 Sensor Array Schematic

Image Source: Author

The sensor array consisted of the six sensor circuits shown above. The schematic displayed JP2 - 7, which acted as pin headers for the individual sensors. These pins were connected to a series of panel-mounted 3.5mm TS jacks. The jacks featured two conductors and allowed for easy connection and disconnection of individual sensors to the main board. These force sensors provided measurements of how hard each component of the drum kit was being hit. The leads of the sensors were connected to a 3.5mm TS cable, providing a straightforward method to make the sensors removable from the device using existing connectors with the required number of conductors. This arrangement also made the sensors interchangeable in terms of which sensor controlled which instrument on the drum kit.

The force sensors acted as variable resistors with two leads. One of these leads was connected to an ADC pin on the microcontroller, with each sensor connecting to its own individual ADC pin. According to the STM32G4 datasheet, the 12-bit ADCs read voltage values from 0V to +3.3V and output a digital value from 0 to 4095. Since the ADCs could not sense a change in resistance, a voltage divider was necessary to enable the functionality of the sensors. To achieve this, a 10KΩ resistor was placed in parallel with the ADC pin that was connected to ground. When unpressed, the sensor was expected to be around 20MΩ. Applying +3.3V to the other lead of the sensor resulted in a voltage at the ADC pin of approximately 1.6mV, which was considered a near-zero digital output value from the ADC pin. Conversely, when the sensor was fully pressed, the resistance was near 0Ω. In this case, the voltage seen by the ADC pin was +3.3V, registering as a digital output value of 4095. This setup allowed for the utilization of the full range of the sensor as digital values from the ADC.

6.4 microSD IO

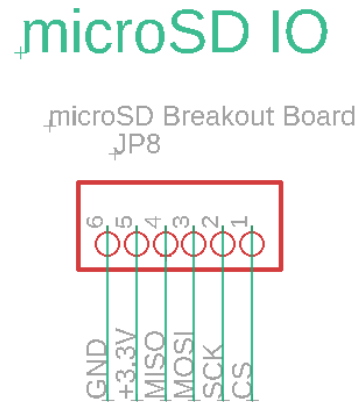


Figure 6.4 microSD IO Schematic

Image Source: Author

The microSD IO schematic shown above was very simple. The 6-pin header connected to a breakout board that contained a microSD port, allowing it to interface with the microcontroller. The GND and +3.3V were supplied by the main board, while the other 4 pins connected and communicated with the microcontroller over SPI.

6.5 Profile Select IO

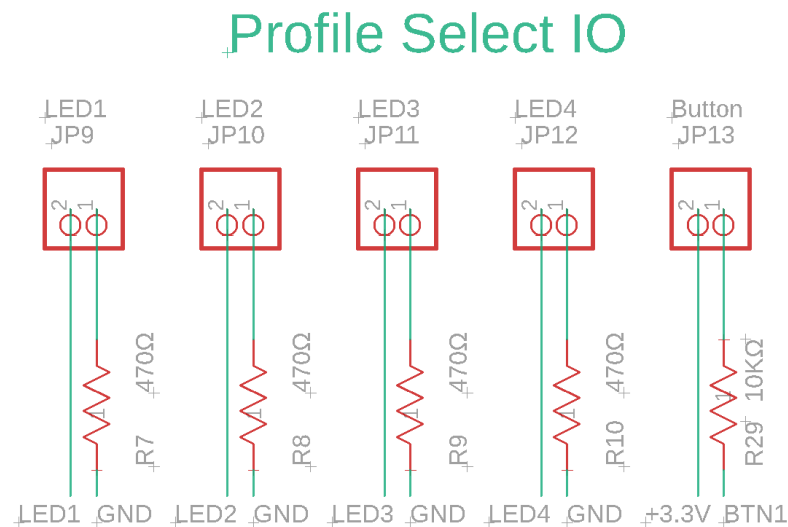


Figure 6.5 Profile Select Schematic

Image Source: Author

The Profile Selection IO facilitated the ability for the user to switch through different sound profiles of the drum kit. These sound profiles changed the style of sound output. The system defaulted to the first profile on startup, illuminating LED1. The button was set up similarly to the force sensors, but it functioned as a

digital input. The pin was configured as a digital input with a pull-down resistor. The button had two leads that, when pressed, bridged together. One of the pins was connected to the +3.3V line, while the other connected to a 10K Ω resistor that then linked to the GPIO pin. This configuration meant that normally the input pin was pulled low, but when the button was pressed, it was pulled high. To the system, this was seen as a trigger for the profile to switch. The button caused the system to rotate through the profiles in a round-robin formation. It was a momentary button that, when released, immediately returned to the open position. Therefore, the system switched profiles only when it first detected the higher pin voltage and did not switch profiles again until the button was released. If the button input bounced when pressed—meaning it unintentionally alternated quickly between high and low—those inputs were not considered for the profile switching. If this issue occurred, it could be addressed in the microcontroller's software.

When a new profile was chosen, an LED lit up to indicate which profile was currently selected. These LEDs were shown in the schematic and had a 470 Ω resistor in line with them to limit their current. The input voltage for these pins came from a GPIO that was set to an output pin in push-pull mode. Only one LED was illuminated at a time to signify which profile was currently active. Each LED pulled approximately 7mA from the output pin. According to the STM32G4 datasheet, output pins were rated to supply no more than 20mA. All output pins combined could not source or sink more than 100mA collectively. Because of this, it was important to minimize current usage, so having only one LED illuminated at a time was desirable. There were four LEDs, allowing for four separate profiles. This could have included three sound profiles and one MIDI output profile, or four sound profiles that all simultaneously output MIDI information and played the sounds associated with that profile.

The LEDs and the profile select button were panel mounted in a conspicuous location for the user, ensuring they were easily accessed and visible. The ports JP9 - 13 could be populated with pin headers if removable connectors were desired, or wires could be directly soldered from the LEDs and button to the PCB. The board clearly denoted which of the two pins was positive and which was negative, so that the LEDs could be oriented properly.

6.6 Pedal IO

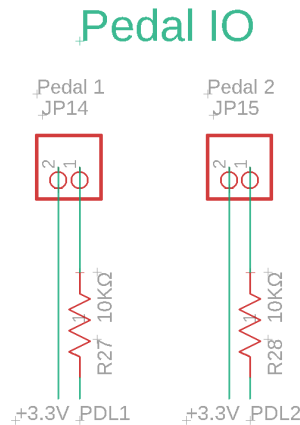


Figure 6.6 Pedal IO Schematic

Image Source: Author

The Pedal IO described above functioned similarly to the profile select button as a digital input. The pedals were connected to the +3.3V line and a GPIO pin set as a digital input with a pull-down resistor. When a pedal was unpressed, the digital input pin it was connected to read as low. When the pedal was depressed, it bridged the pins together, and the input pin read as high. The pedals were connected using panel-mounted ¼" TS jacks. Similar to the profile select button, a 10KΩ resistor was added in line with the pedal output for protection.

Most foot pedals used in musical applications had this connector. The connector featured two conductors and functioned as a switch. These foot pedals served as actuators for certain devices in a musical setting. A common example of usage was a foot pedal employed as a sustain pedal on a digital piano or MIDI controller. The pedal plugged into a ¼" TS jack on the digital piano or MIDI controller, and when pressed down, it sustained any notes being played. We used these foot pedals for a similar application that replicated the pedals used with drum kits.

The two pedals had different purposes. The first pedal simulated a kick pedal on a drum kit. When pressed down, the system used the pedal as an input trigger to play a kick drum sound in the same way the force sensors were used. The second pedal simulated the pedal on a hi-hat cymbal. This pedal did not trigger any sounds itself but was used as a check when the hi-hat force sensor was struck. If the hi-hat pedal was unpressed when the hi-hat force sensor was struck, the system played an open hi-hat sound. If the hi-hat pedal was pressed when the hi-hat force sensor was struck, the system played a closed hi-hat sound.

It should be noted that usually, the act of pressing the hi-hat pedal was used to create its own sound; however, as the pedal was a digital input in our case, there was no way to determine how loud the user would like the sound to be. The sound might have interfered with other sounds when playing if it was too loud. If the sound were too quiet, then it would have been generally ineffective in

6.7 MIDI Output

The schematic for the simple MIDI output that was incorporated in our project is shown above. The 3-pin header was connected to a female 5-pin DIN port on the side of the assembly. This port connected to an external MIDI-capable device to transmit the notes being played or to be recorded in a user's DAW. The MIDI output was powered by the +5V line and contained two 220Ω resistors. By design, a 220Ω resistor was placed across the +5V line to limit current. Additionally, a 220Ω resistor was placed on the UART transmitting line to protect it as well. These pins connected to the MIDI output port either through connectors and pin headers or wires directly soldered into the board.

Speaker Amplifier

The diagram shows a PAM8403DR speaker amplifier circuit. The chip is a U3 with pins 1-8 on the left and 9-16 on the right. The circuit includes a 10Ω resistor, a 10μF capacitor (C15), a 10μF capacitor (C12), a 1KΩ resistor (R11), and a speaker (SPKR). The output of the amplifier is connected to the speaker through a 10μF capacitor (C12).

Figure 6.8 Speaker Amplifier Schematic

Image Source: Author

The speaker amplifier utilized the PAM8403 to drive the internal speaker by amplifying the sound produced by the DAC pins of the microcontroller. The amplifier featured various peripheral components to enhance stability and ensure the proper functioning of the amplifier.

The audio signal initially arrived from the 'AMP' line located at the top left of the schematic. This line connected to all of the DAC pins (Pins 18, 19, 20) on the microcontroller. The DAC pins were buffered, protecting them from extraneous current and allowing for a direct connection to the amplifier inputs. This setup enabled the three pins to be tied together, allowing all audio signals to go to the same destination, regardless of which DAC was outputting. It also permitted the simultaneous use of multiple DACs to drive the amplifiers if needed.

In the schematic, the AMP pin connected to one lead of a potentiometer, represented as a 1x3 pin header. The potentiometer was mounted to the outside of the device assembly and connected to the main PCB through a removable connector or by soldering wires directly onto the board. This potentiometer acted as a volume knob for the speaker amplifier, enabling the user to adjust the volume to their preferred level. The potentiometer value was set at 10K Ω , selected based on the example circuit diagram used as a reference for this amplifier circuit.

The other lead on the volume knob potentiometer connected to an isolated ground bus. A 1 Ω resistor separated the ground of the audio input device from the ground of the power source for the amplifier, providing isolation. The example circuit schematic indicated that several nodes needed to connect to ground on the side of the 1 Ω resistor connected to the audio input. This isolated ground line was visible at the bottom of the speaker amplifier circuit schematic and connected to various other components. It should be noted that the ground from the power source and the ground from the audio input source were the same, as both were on the same board and not separate devices. Nonetheless, the resistor was included as a precaution.

The 'output' of the potentiometer connected to the main input of the amplifier. This configuration allowed the user to control how much of the input to the amplifier was influenced by the audio input source and how much was influenced by ground.

As the audio signal entered the input of the amplifier IC, it passed through a resistor and capacitor in series. This formed an RC timing circuit, which limited the maximum frequency entering the amplifier input, controlled the maximum gain, and prevented the amplifier from saturating. This portion of the circuit served multiple purposes, but the general idea was to protect the amplifier. Looking at the VDD pin, the +5V input passed through a network of resistors and capacitors before entering the PAM8403. The first two capacitors in this network connected directly to ground, with values of 1 μ F and 100 μ F respectively. These capacitors stabilized the input voltage of the amplifier, ensuring it operated

normally and was less affected by potential noise. After this, the input power passed through a resistor before entering the PAM8403 in parallel with a 10 μ F capacitor attached to the isolated ground plane for stability.

Above the VDD pin was the MUTE pin, which functioned similarly to the SHND pin discussed later. This pin allowed the amplifier to be muted by pulling it low, as it was an active low pin. When activated, the amplifier's output would be temporarily muted, but the amplifier itself remained powered on and active. For this application, there was no need for the MUTE pin, so it was left disconnected, as it was pulled high by a pull-up resistor.

The pin above was the PVDD_2 pin, a Power VDD pin shorted to another PVDD_1 pin on the opposite side of the chip. These pins provided supply voltage for the power side of the equipment, while the regular VDD pin powered the analog section. These pins were not relevant to the application and were left shorted together.

Above these pins were the speaker output pins, where the amplified audio signal was outputted. These pins connected to the internal speaker at JP19. The 1x2 pin header at JP19 connected to the internal speaker mounted on the side of the assembly, using either wires with removable connectors or soldered directly onto the main board. Between these two pins was the PGND or Power Ground pin, which was shorted to GND.

The other side of the PAM8403 was not extensively used in the application, as only one of its two channels was utilized. The input pin and both output pins on this side of the chip were left floating. The PGND pin on this side was still shorted to ground. There was another pin on the bottom left of the chip labeled NC, which was not connected to anything internally and served no purpose. It was tied to the isolated ground to reduce any potential EMI it could cause. The GND pin on this side was tied to the regular ground. The VREF pin was discussed in the following section.

The next pin discussed on the PAM8403 was the SHND pin, the 'shutdown' pin. The functions of the MUTE pin and the SHND pin were very similar. The SHND pin was an active low pin pulled high when not connected to anything. When pulled low, it shut down the bias circuitry of the amplifier chip, effectively turning it off. This was beneficial because, unlike the MUTE pin, it significantly reduced the chip's current draw. Due to this added advantage, this pin was employed to mute the internal speaker when headphones were plugged into the Phones jack of the device. The specifics of this operation were described in the next section. This pin connected to a microcontroller pin functioning as a push-pull digital output, controlling the muting of the speaker amplifier.

Lastly, the VREF pin played a critical role in enabling the shutdown feature. This pin connected to a capacitor, which was then linked to the isolated ground. The capacitor had a value of 0.1 μ F and determined how quickly the chip started up or recovered from the shutdown state. It also helped reduce noise from the power supply and minimized clicking or popping sounds that might be heard when entering or leaving the shutdown state.

6.9 Headphone Amplifier

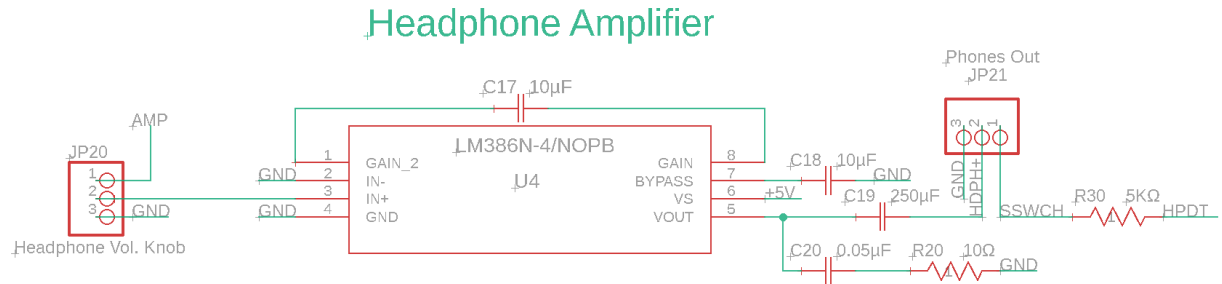


Figure 6.9 Headphone Amplifier Schematic
Image Source: Author

The headphone amplifier was the other component of the audio output system. This amplifier functioned differently than the speaker amplifier as it drove a much higher impedance device. The assumed impedance of a device plugged into the auxiliary output was 32Ω .

The signal flow for this portion of the circuit began with the 'AMP' node on the left side of the circuit. This connected to the JP20 pin header; this 1x3 pin header was connected to a potentiometer that was panel mounted to the outside of the assembly. This potentiometer was used as a volume control knob for the user to adjust the output volume of the headphone amp to a desirable level. The other lead of this potentiometer was connected to ground. The amplifier input was attached to the wiper of the potentiometer and changed the mix between the ground signal and the DAC output signal flowing into the input of the amplifier. The value of this potentiometer was chosen to be 10K Ω , as indicated in the typical application circuit shown on the LM386 datasheet. The specific circuit followed for this design was the typical application circuit that provided a gain of 200.

The power supply circuit of this amplifier was much simpler than that of the speaker amplifier. The VS pin on the amplifier was connected directly to the +5V line. The GND pin and the IN- pin were also both connected directly to ground.

The audio input from the potentiometer connected to the IN+ pin on the LM386 and was amplified with a gain determined by the gain pins. The pins GAIN and GAIN_2 on the chip were biased using a capacitor to set the value of the gain applied to the target signal. When not connected, the gain value of the LM386 was set to 20 by default. This was the lowest value and served as a default to protect the user from accidentally damaging a device connected to the output in cases where they ignored the GAIN pins.

In our specific case, the gain pins were connected to each other by connecting them to either side of a $10\mu\text{F}$ capacitor, as shown above the chip in the schematic. Setting the chip to a gain of 200 provided the maximum output power possible. We wanted the output power to be at its maximum because our intended audio device for this jack had a high impedance. From the datasheet, we observed that even at the maximum gain the LM386 could output, it would not

damage most headphones. However, users could plug in whatever they wanted into this jack, leaving the possibility of connecting a device incompatible with the amplifier output, which could potentially lead to damage.

The audio output of the amplifier came out of the VOUT pin. This pin was then connected to a capacitor leading towards the audio output jack, along with a capacitor and resistor in series going to ground. The function of the resistor and capacitor in series was to provide a load at higher frequencies, stabilizing the amplifier and preventing oscillations from occurring. If removed, the oscillations could have damaged the amplifier.

Moving towards the output channel, there was a 250 μ F electrolytic capacitor between the amplifier output and the speaker. The positive terminal of the capacitor faced the amplifier output pin while the negative terminal faced the speaker input. This capacitor was used to block DC voltage from the speaker, reducing distortion and protecting the speaker.

Finally, at JP21, there was a 1x3 pin header that connected to the panel-mounted internal speaker. The connection could be made either by wires with removable connectors or by wires that were soldered directly into the main board. Although the headphone output was a mono 1/4" TS jack, this connector had three pins instead of two. This configuration was designed to facilitate the muting of the internal speaker when something was plugged into the Phones output jack. The Phones output jack was a switched jack that allowed for the detection of when something had been plugged into it.

A switched mono jack had four pins; two of them were used as connections to handle the conventional use of the jack. The other two pins were normally-closed switches that connected to the other two pins respectively when nothing was plugged into the jack and opened when something was plugged in. This allowed for the detection of the jack being used without the need to rely on an external circuit. Since the switches were actuated physically, they opened when any device with a 1/4" connector was plugged in.

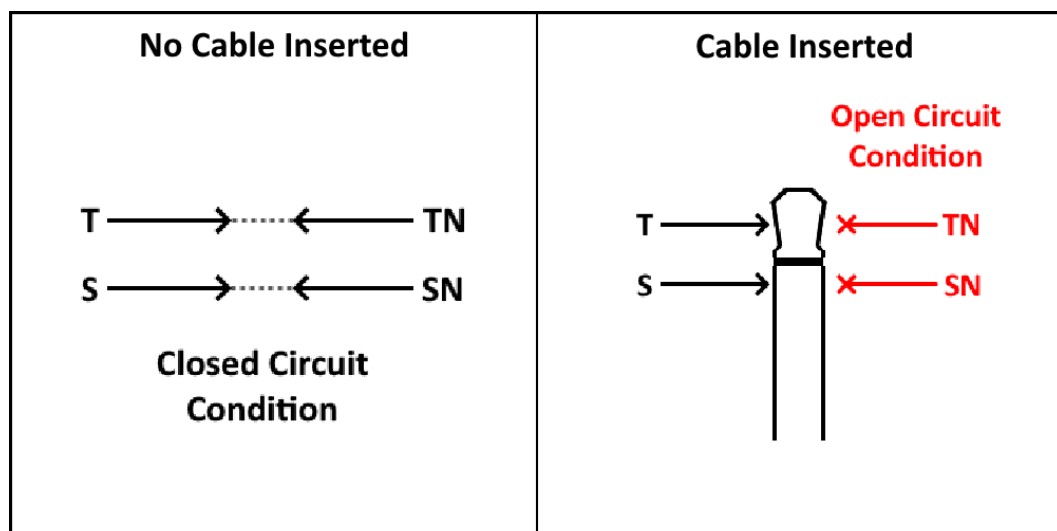


Figure 6.10 Switched Jack Functionality

Image Source: Author

The opening of the switches in the switched jack was used to control the state of the SHND pin on the speaker amplifier. In order to shut down the speaker amplifier, it was necessary to drive the SHND pin low. To achieve this, the pin needed to be grounded in some way. Connecting the pin to the sleeve switch pin (SN in Figure 6.10) allowed us to ground the SHND pin, but only when nothing was plugged into the Phones jack. When something was plugged into the Phones jack, the sleeve switch opened, and the SHND pin was no longer grounded. The internal pull-up resistor on the SHND pin then enabled the amplifier again. This configuration muted the speaker until headphones were plugged in, resulting in the headphones and internal speaker playing at the same time, which was the opposite of the intended functionality.

In fact, there was no way to directly use the switched pins on the Phones jack to pull the SHND pin on the speaker amplifier low. Somewhere in the flow of current, the voltage of the signal had to be inverted so that the headphones and internal speaker functioned in a mutually exclusive manner. This could have involved using an inverter IC; however, this simple functionality could be implemented in the software instead, using the components already on the board. To do this, it was necessary to use two GPIO pins set to digital input and digital output.

To make this functionality work, the sleeve switch pin of the Phones jack was first connected to a GPIO pin on the microcontroller. For this, Pin 42 was used. Pin 42 was then set as an input with a pull-up resistor. This way, when nothing was plugged into the Phones jack, the normally-closed switch on the jack connected the sleeve switch to ground, as the negative terminal of the headphone audio pair of wires was already tied to ground. When something was plugged into the Phones jack, the switch opened, and the input pin was no longer grounded. The internal pull-up resistor then pulled the input pin high. This meant that in the software, plugging something into the Phones jack appeared as Pin 42 going high.

It was noted that there was a 5K Ω resistor between the sleeve switch pin and Pin 42. This resistor acted as a current limiter so that the input pin did not short to ground and damage the microcontroller. It also ensured that it would not damage itself if the pin was set incorrectly. The value of this resistor was calculated to specifically limit the current to an acceptable level while keeping the voltage at Pin 42 low enough to be considered a digital low by the microcontroller; the values used for this calculation were based on the information in the STM32G4 datasheet.

To complete the amp shutdown functionality, another pin was needed as a digital output to drive the pin low. Pin 41 on the microcontroller was used for this purpose. This pin was set as a digital output in a push-pull configuration. Normally, it was intended to drive this pin high so that the amplifier remained active; however, the microcontroller pin could only deliver voltage up to 3.3V while the amplifier operated at 5V. This limitation was the reason a 1000 Ω

resistor was placed between the two pins. The resistor limited the current flowing to the output pin to only a few milliamps and kept the voltage at the shutdown pin high enough to register as enabled.

The software was then designed to drive Pin 41 low when it detected Pin 42 high. This effectively functioned as an inverter. In this case, the voltage divider formed by the 1000Ω resistor and the internal pull-up resistor experienced by the SHND pin drove the pin to a low enough voltage to disable the amplifier. The only additional task required to complete this functionality was to implement the logic in the software.

6.10 Overall Schematic

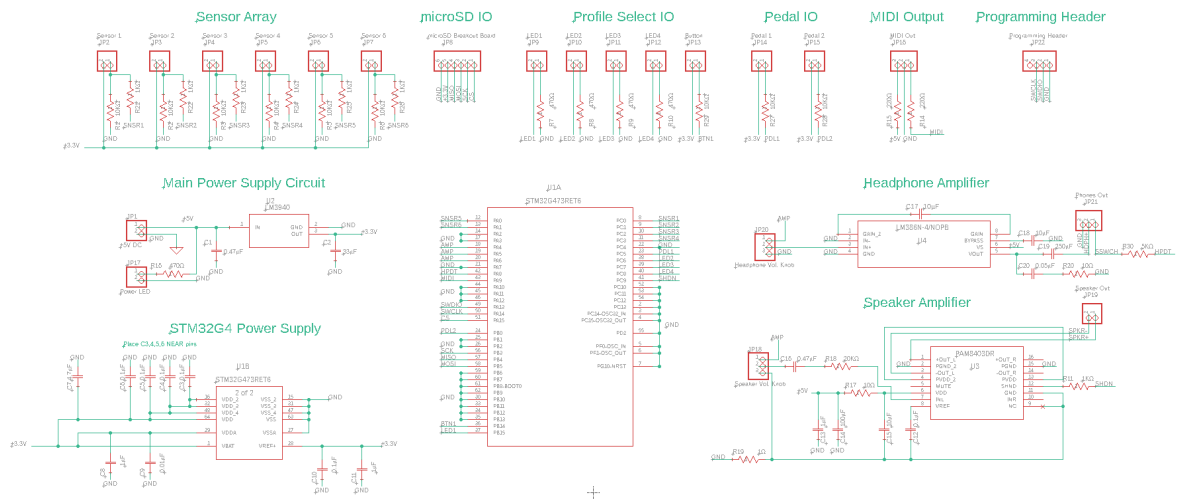


Figure 6.11 Overall Schematic
Image Source: Author

All together, these components created the overall schematic for our project. This was what was included on the main PCB. The components needed to be placed on the board according to their mounting style. The center of the image showed the IO pins of the microcontroller. Though it appeared as if nothing was connected, the pins had wires coming off them with the same labels as many of the inputs and outputs on other parts of the circuit. In this way, they were connected within the software but were not visually displayed in this manner to reduce clutter. All unused GPIO pins were shorted directly to ground to minimize any issues that might arise from leaving the pins floating.

All of the pin headers shown in the circuit schematic connected to some part of the general user IO. This included indicators such as LEDs, the profile select button, the headphone jack, and the sensor jack array. All of these components were panel mounted and not attached to the board. This approach provided us with more options on how to specifically customize the layout of our user interface so that it flowed well and was easy to use.

6.11 Structure & External IO

Top View

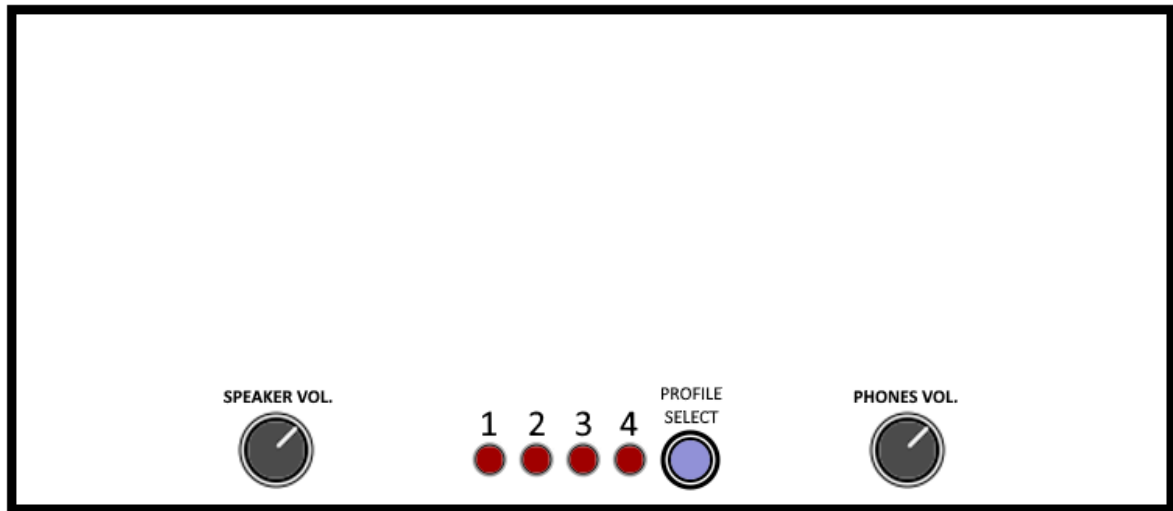


Figure 6.12 Top View of the Device

Image Source: Author

The image above provides a graphical representation of what the central processing hub looks like from above. It takes the form of a rectangular box with various I/O ports on its different faces. In this image, the bottom of the box faces the user when in position. On the side near the user, four different LEDs are visible, indicating which profile is currently selected. As discussed in earlier sections, only one of these LEDs is illuminated at a time to avoid pulling too much current from the microcontroller. These LEDs are on a circuit that limits their current draw to less than 10 mA. Labels above the LEDs clearly indicate which profile number each represents.

Nearby is a PROFILE SELECT button that allows the user to round-robin through the different profiles by pressing the button momentarily and then releasing it. This action also changes which profile LED is illuminated.

On opposing sides of the profile selection interface are two knobs, one for individually controlling either the volume of the internal speaker or the volume of the headphones. These knobs connect to the shafts of two 10 k Ω potentiometers linked to the amplifier circuits of the two audio output channels. This arrangement allows the user to adjust the volume for the two outputs separately to their liking. The PHONES VOL. knob is placed near the PHONES port, closer to it than the SPEAKER VOL. knob. The SPEAKER VOL. knob is located near the internal speaker, closer to it than the PHONES VOL. knob. Both knobs are clearly labeled, ensuring the user understands which knob controls what. They are positioned in a way that does not obstruct the user and minimizes the risk of accidental adjustments, protecting the user's hearing.

Front View

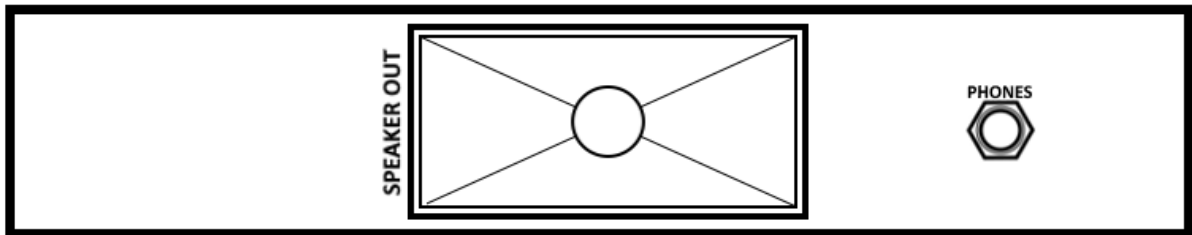


Figure 6.13 Front View of the Device

Image Source: Author

This image depicts the view from the front face of the device. In this case, the front face refers to the side that points towards the user, which is what they see when looking head-on at the device. This face includes only a few features, such as the speaker and the PHONES port.

The speaker is represented by the rectangular shape in the center, providing a rough estimation of what the internal speaker looks like. The final design may involve a grill or protective covering for the internal speaker to prevent damage from the user or any foreign object that could puncture or dislodge it. This protective grill allows sound from the speaker to escape the enclosure with minimal impact on overall sound quality.

A label appears on the left side of the internal speaker in the diagram above. This label is optional and does not need to be included in the final design, but it can be added if desired. Its purpose in the drawing is to indicate what the rectangular shape represents and to ensure proper positioning on the design.

The final feature visible on this face of the structure is the PHONES port. This port is a mono $\frac{1}{4}$ " TS jack where the user can plug in any external headphones they want to use when practicing or recording with the device, potentially in a quiet environment. Plugging into this port mutes the internal speaker and outputs audio solely through the PHONES port. The PHONES port should be located in a position that is easily accessible for the user and does not cause the headphone cable to interfere with their movement or bend in an undesirable manner. It should also be situated near the PHONES VOL. knob so that the user can easily identify which knob controls its volume.

Rear View

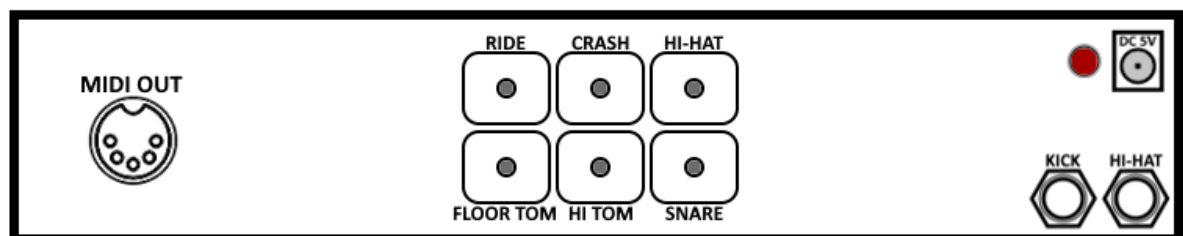


Figure 6.14 Rear View of the Device

Image Source: Author

This image depicts the rear view of the device, specifically the face that faces away from the user during operation. The majority of the I/O is located here, including the DC Power Input, the ¼" TS jacks for the Kick and Hi-Hat foot pedals, the array of 3.5mm jacks that connect the sensors to the microcontroller, and the MIDI OUT port used to send MIDI information to an external device.

We begin by discussing the DC Power Input jack. This is a DC barrel plug that powers the overall system. It is a 2.1/5.5mm jack with a positive center, mounted in the top corner of one side of the rear face of the enclosure. The jack is labeled "5V 1.5A," accompanied by a symbol depicting the positive center and negative sleeve of the port nearby, ensuring the user understands the correct specifications of the DC power needed for the device to function properly.

Near the DC Power Input is an indicator LED that turns on when the device is plugged in. This LED is positioned close to the power input to clearly indicate its purpose. It is directly powered by the DC Power Input and is not controlled by the microcontroller. Its illumination signifies that power is reaching the system and the PCB, rather than indicating the function of the microcontroller.

In the bottom corner (the bottom left corner when viewed from the front face) are two ¼" TS jacks designated for the Kick and Hi-Hat foot pedals. These jacks are located on the left side for the user, as the Hi-Hat is typically placed there in a conventional right-handed drum kit. The Kick port is positioned nearby for convenience and coherence with the design, with the Hi-Hat placed further left. These ports serve to signal the system when to play a kick drum sound and when to play an open or closed hi-hat sound file. When pressed, the connection bridges, and the microcontroller detects a high digital logic signal.

In the center of the rear face is the array of 3.5mm TS jacks that connect to various components of the drum kit. Six 3.5mm TS cables connect to these jacks, routing power to the sensors on all drum and cymbal enclosures. Each port is labeled according to the component of the drum kit it connects to, with their positions reflecting the usual layout of a conventional right-handed drum kit. Similar to the foot pedal jacks, the connections for these jacks bridge based on the force applied to the force sensor, allowing the microcontroller to read how hard the sensor is pressed by the drumstick.

Lastly, on the left side of the rear (the right side from the user's perspective) is the MIDI OUT jack. This female 5-pin DIN jack connects a MIDI cable to an external device capable of receiving MIDI information, such as an audio interface linked to a computer. This connection allows the MIDI information from the drum kit to be recorded in the user's DAW and utilized in their music production process.

7. Software Design

This section discusses the software design of our project as well as the structure of the code and the way that different functionality will be implemented on the microcontroller.

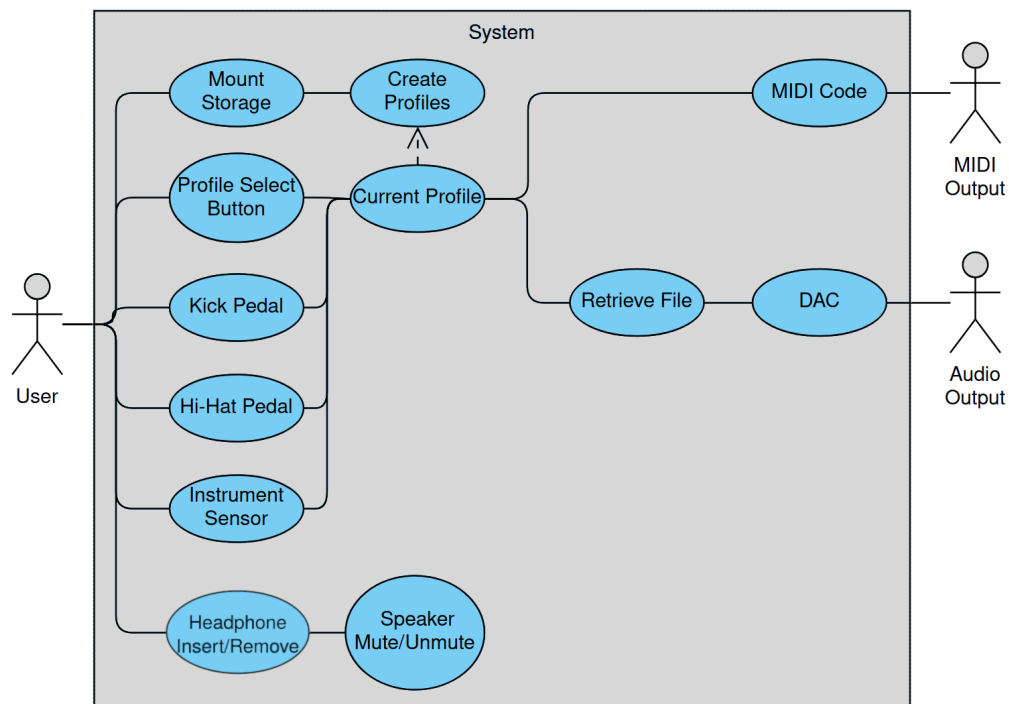


Figure 7.1 Software Use Case Diagram
By Jameson Palmer

A user interfaces with several modules of the device while several monitoring tasks are performed. A user action can be captured to registers where the task may iterate over these values. The captured value is used in branching behavior of the software to continue along the software flow. When the reading is passed to further process the task may continue monitoring the rest of the register values. Separating the monitoring tasks from these added processing steps is meant to increase parallelization giving greater responsiveness to our array of sensors. User interaction is the driving force for changing the state of the software at any given time. After initializing the state of the device a user can interact with the instrument modules or switch profiles. Upon switching profiles the state must reinitialize with the content associated with the given profile.

Cycling through the profiles available using the profile select button can activate the precaching scheme several times which can introduce high i/o usage and delay. The precaching scheme is part of the initialization steps whenever the state is changed. The last instance of the profile switch takes precedence over other instances of the cache to ensure maximum utilization of memory resources close to the cores. The last iteration of precache should evict most or all of the contents of other profile sound files to contain large portions or the entirety of the active profile content. It is expected for a user to not immediately need the response of the instruments when the profile switches. Transferring data from the

external storage is a significantly slower operation. When a user is switching profiles there is enough of a time window to perform access to external storage for the precache scheme. We expect the first usage of profile resources to be a cache miss and get transfers to external storage so a user would normally experience part of this delay regardless. Doing these i/o transfers as a batch will lead to fewer cache misses following the initialization.

The microcontroller cache policy should be set to most recently used caching schemes to give the greatest benefit of retaining useful data close to the cores. After precaching the microcontroller cache policy will determine contents of cache as the device is used with more frequent requests of data having a greater lifetime in cache. Data that is residing in cache but has gone stale or is not used as much as other rows of data will be evicted and the space is used for more frequently accessed rows. The entirety of a profile's audio data may not be placed inside of cache. The limited memory space of the microcontroller must share its resources with the running software, codex, buffers, and the cache space. Instances of data transfer from external storage may be mitigated by retaining beginning segments of files while a retrieval is sent for the remainder. Audio files are arranged in sequence with playback times so the beginnings of these files can have access through the cache in the time necessary to retrieve following sequences.

Senior Design II:

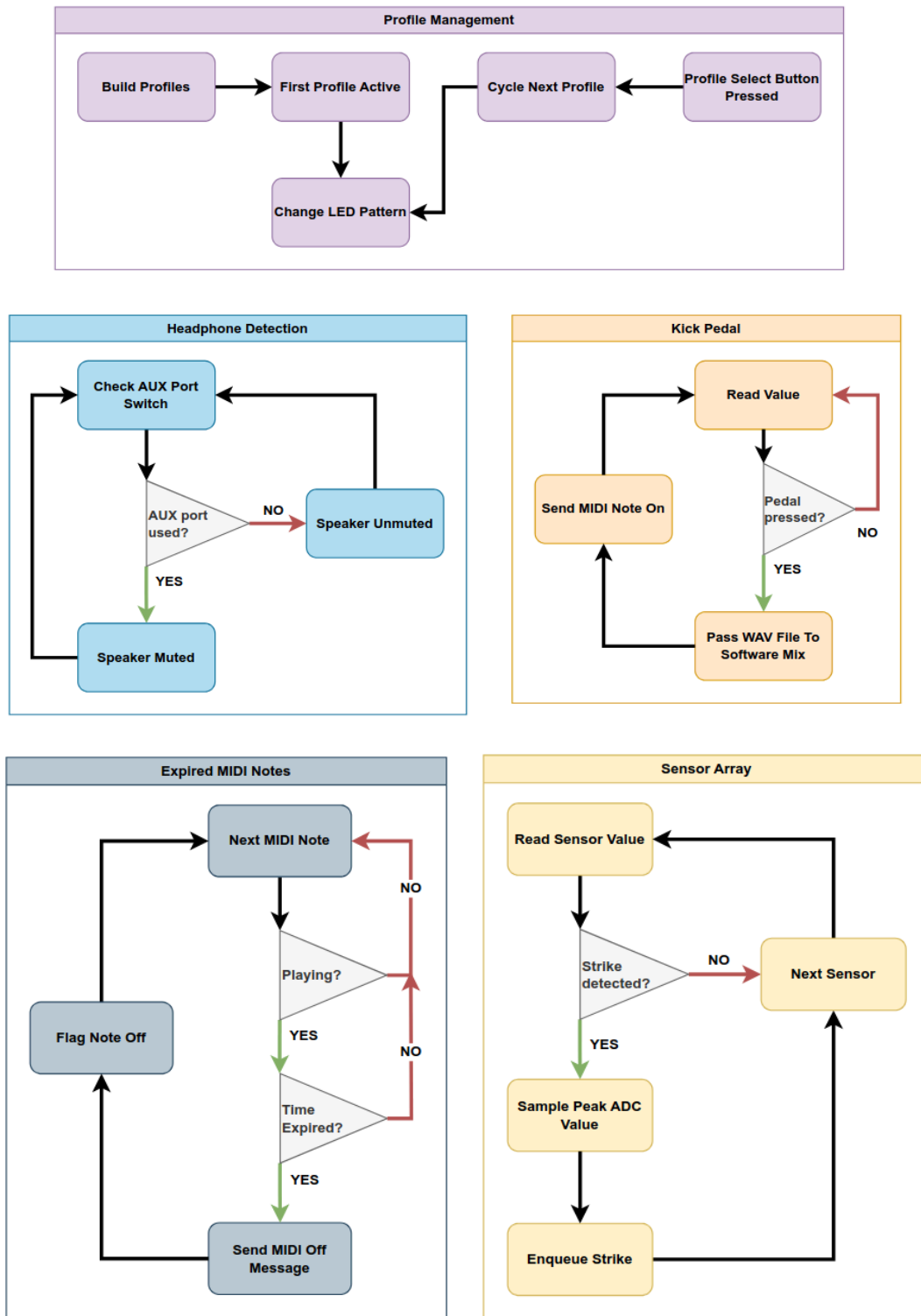


Figure 7.1a Software Tasks
By Jameson Palmer

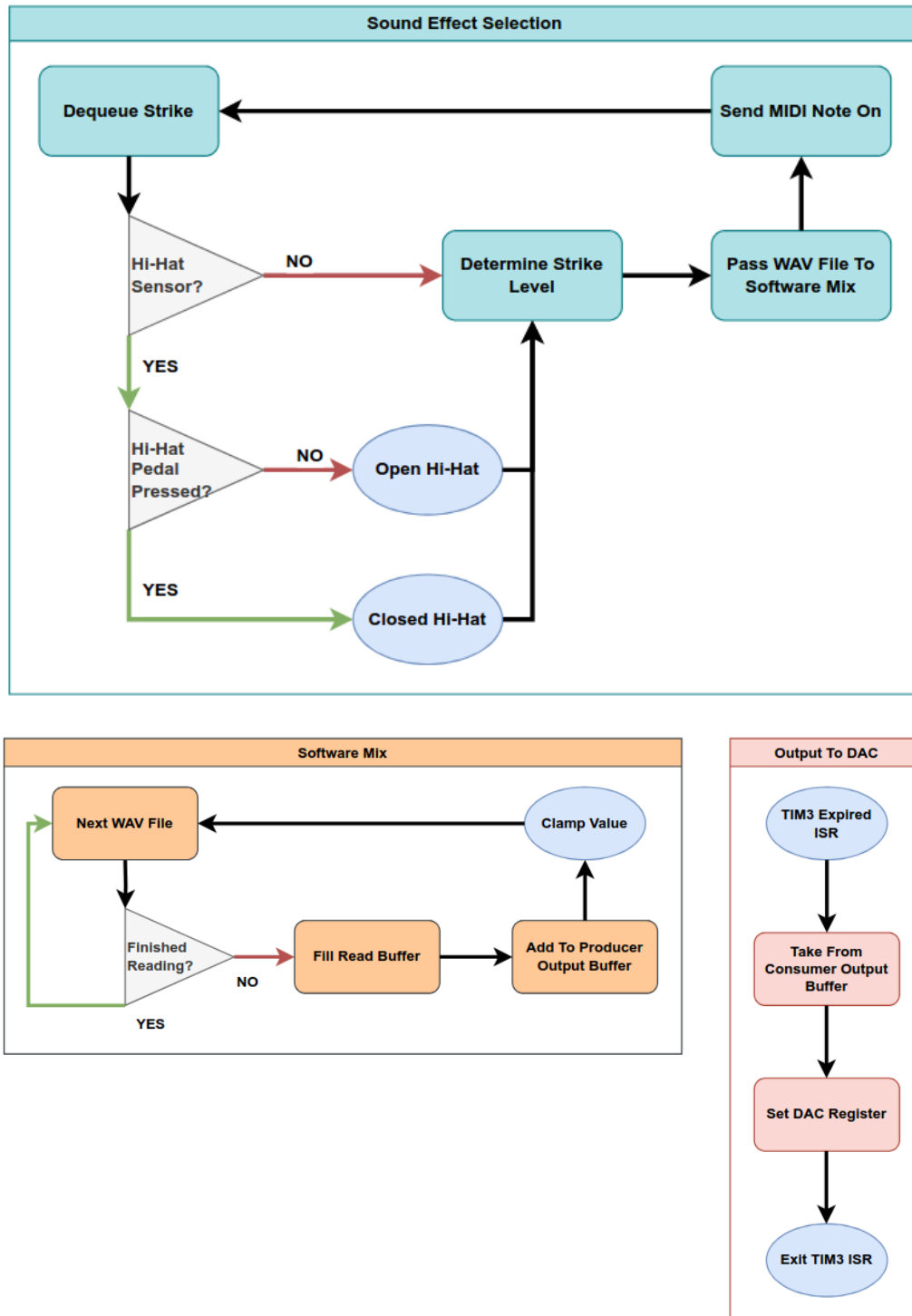


Figure 7.1b Software Tasks
By Jameson Palmer

7.1 External Storage Mounting and Parsing

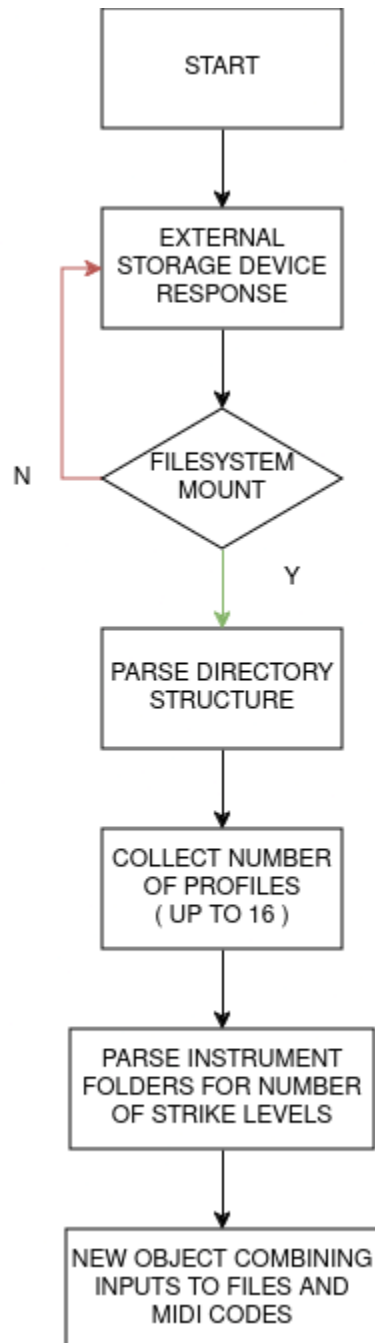


Figure 7.2 Procedure to Collect Profiles from External Storage
By Jameson Palmer

7.2 Monitoring of the Sensor Array

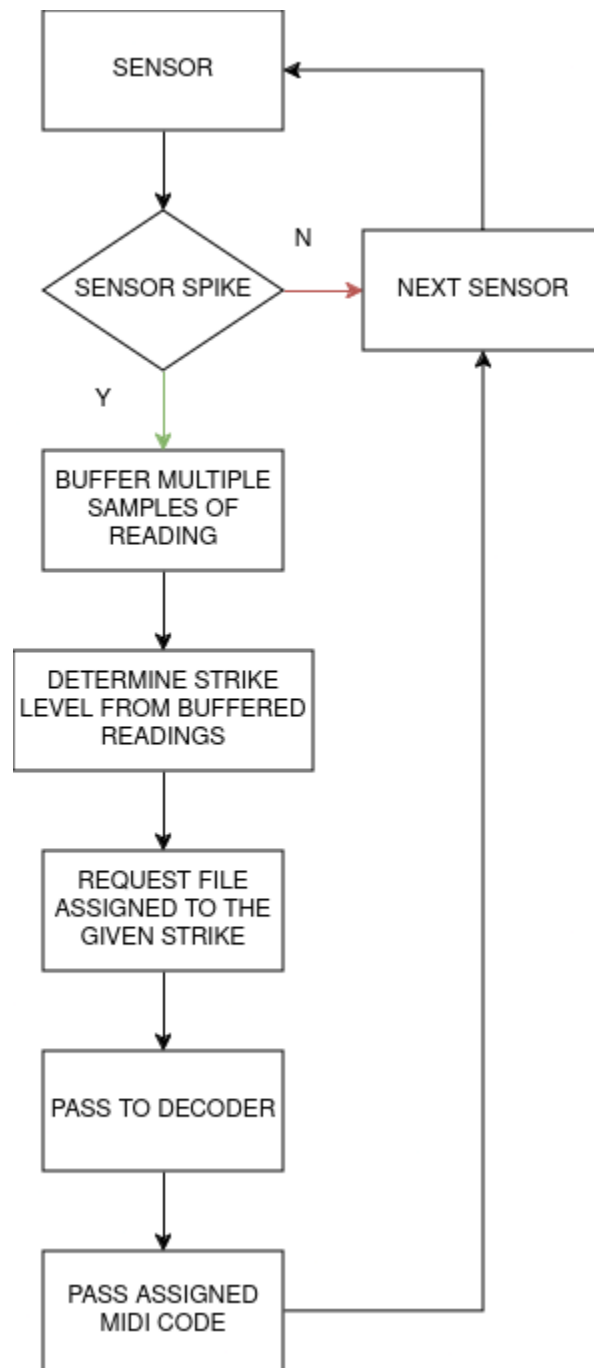


Figure 7.3 Procedure to Collect and Use Sensor Readings
By Jameson Palmer

7.3 Standby Mode, Idle

Without external storage available the device will not receive a response from the storage monitoring task. In the event the storage card is not inserted or removed the device will enter a standby mode until a response is received from an external storage device. Without storage the current mounted filesystem will be unmounted and tasks which are used for instruments will be suspended while the system awaits a response. Upon reinsertion the storage communication is then reestablished by the SPI initialization steps. The filesystem is then remounted and sent through the parsing stages. In the event a storage device is inserted that does not have a supported filesystem the software will await a properly formatted device to be present. Parsing and coursing the directories will initialize the state of our system again to ensure previous mounted data is no longer valid. A fresh state of the system is created before resuming suspended tasks.

7.4 Runtime

The procedure of our system's software flow when the system is ready begins with the regular polling of ADC sensor data to determine a strike above the `SENSOR_CUTOFF` value. When a strike is determined we have the sensor data translated into the appropriate index location of the audio resource for the particular sensor and strike value. Since this is dependent on both the presence of an SD card and the 'profiles' array being created. We use both of these as flags for our tasks as whenever the SD card is not connected or the 'profiles' structure is not filled with any data the dependent tasks do not perform any operations. With both the SD card file system mounted and the 'profiles' array containing at least one valid 'profile', the system is made ready with the first index of 'profiles' set as the active profile. A strike is detected on one of the six sensors of our ADC array and gets interpreted as the appropriate index of the active profile. The corresponding WAV file is made accessible and placed into one of eight slots in our software mix. The WAV files in our software mix are buffered and combined to one output buffer before sending the buffer to our DAC. Double-buffering is used to give a producer-consumer relationship between the software mix producer task and the DAC output consumer task. As one created buffer is being output the next buffer is created by our software mix.

7.4.1 System Startup

When the device is powered on, our controlling thread is able to run and set the state of our global flags 'bool sd_card_presence' and 'uint8_t num_profiles'. These flags must both be set to allow other tasks to operate. This controlling thread is responsible for the mounting of a FAT32 file system and creation of the profiles array. This task runs periodically to determine the state of

our SD card whether it is inserted, not inserted, was just inserted, and was just removed. If no FAT32 file system can be mounted the task will continue periodically attempting to mount until a provided FAT32 file system is mounted or an SD card that has a FAT32 file system is provided and mounted. This will set 'sd_card_presence = true'. Once a file system is able to mount, the root directory is checked for a PROFILES_FOLDER, by default the folder name is 'profiles'. This is the directory which contains all resources for the device to use. Follow the guide for creating and customizing these profiles to better understand how these resources are assigned. At least one profile must be created for the system to use and to set the global flag 'num_profiles' indicating the system is ready. With at least one profile created the system can select the first index of our 'profiles' array as the active profile. This is indicated by LED1. If more profiles are available, up to the maximum of four, the user may press the profile select button to cycle through them. Each LED indicates which profile is currently active.

7.4.2 Profile Switching

Parsing the storage will present the structure of our sound library. The number of profile folders available will be determined and the first 16 profiles that are discovered will be taken. A structure array will contain the available profiles in an object-like fashion where the current profile will cycle through the array and wrap around to the first index. Whenever the profile select button is pressed the index will increment and take the information for that index to the profile setup steps. Information in each index contains the pertinent information of how many strike levels per instrument and the associated files or MIDI codes. Assignments made from previous profiles are overwritten or set NULL where applicable. This is to ensure the system state is associated to one profile.

7.4.3 Strike Level Detection

Within each profile the information of how many sound files get associated with an instrument is saved. This value is used to divide the range of sensor values evenly and assign the sound file for each range. Calibration of sensor readings will be done to determine a MIN and MAX strike to find what range of sensor readings gets divided. A strike value passing a given threshold determines the level of strike that is played. To ensure our thresholds are appropriately placed the MAX strike should have an offset so the sensor readings may pass this threshold rather than have a MAX strike lie on this threshold. Without this offset a strike which should be detected at a higher level may be read as a strike from a level below.

Senior Design II:

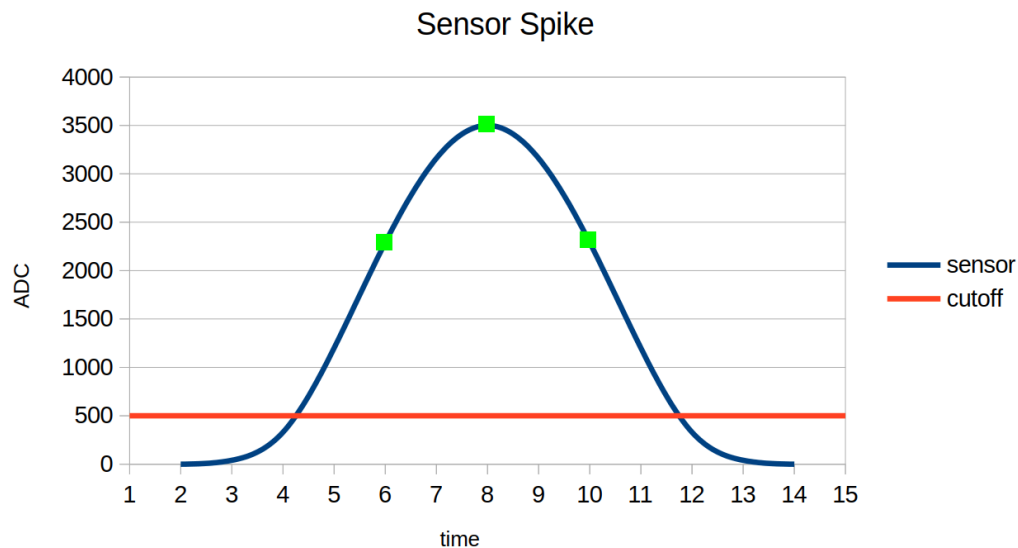


Figure 7.4.1 ADC sensor spike (centered)
by Jameson Palmer

When the ADC reading for any sensor goes above the cutoff threshold, samples are taken and evaluated with each other to determine the peak reading of that sensor. The sampling behavior is designed in a non-blocking manner such that any sensor that is in a spike does not prevent other sensors from also being read by the ADC and sampled themselves. Even when a spike is sustained, so a sensor is pressed down constantly, a peak value can be determined with a set number of samples to take. Once the set number of samples are taken, we determine the highest reading is the sample to use when sending the sensor data to the rest of the program. Even for sensor spikes that may be very short, so the number of samples aren't taken before the ADC value falls back below the cutoff, the device can take the peak value from the samples that could be taken.

Our sampling approach is designed to find which value of the ADC can best represent any individual hit to the sensor. This relates to whichever sound effect may be played. The design for our sampling approach is already non-blocking for every sensor which gives the same behavior as acoustic instruments, each instrument can operate independent of one another. The sampling approach for our instruments must then be designed to get the best value of any strike. The ADC must be read quickly enough across all sensors to be responsive. Each sensor should evaluate enough samples for the peak value. The peak value must be determined fast enough to not have a perceptible delay for the device output.

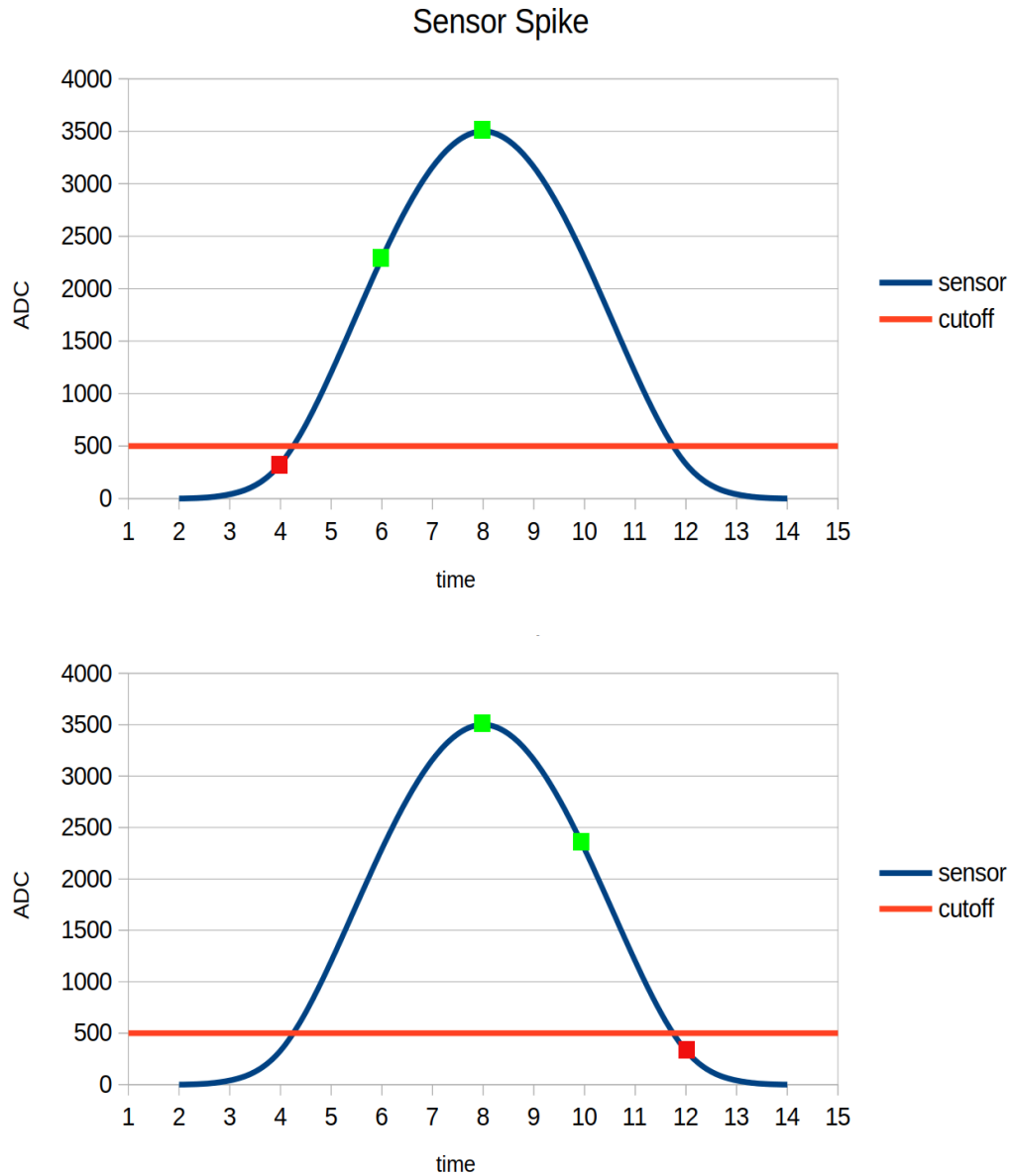


Figure 7.4.2 ADC sensor spike (TOP: leading, BOTTOM: trailing)
by Jameson Palmer

Both figures above represent our sampling strategy across the leading and trailing ends of a spike of the ADC values. Whether a sensor can begin sampling values along the leading or trailing ends we need our readings to cover and overlap along the center. Partially or fully covering the spike within our sampling window can generate an acceptable value for the instrument.

organized. This image only shows the relative locations of the components and not the exact traces between the components. The reason for this is because as research continues it is possible that we may realize that we have made an error and either need to add, remove, or relocate one or more components. This remains a possibility as we test different components of our system independently and it is a smart idea to save money and time completing testing before actually ordering the entire PCB and placing the components.

Nonetheless, this acted as a good example of the approximate dimensions of the board and also displayed the size of the components relative to each other. This allowed us to gain a good idea of how far apart the components needed to be from each other and how much space we required to fit all of the traces onto the board without issue.

Senior Design 2:

Below is the final PCB design for our project. The left side contains all of our digital components on their own ground plane. The right side contains all of our analog components on their own ground plane. These are separated to reduce noise and interference. The two ground planes connect over a 1 ohm resistor in the middle. There is a large AGND trace which crosses to the digital side to give our MCU a reference for the analog ground. There is also an analog power reference going to the MCU as well. The 6 blue lines at the top are our sensor inputs traveling to the ADC pins on our MCU.

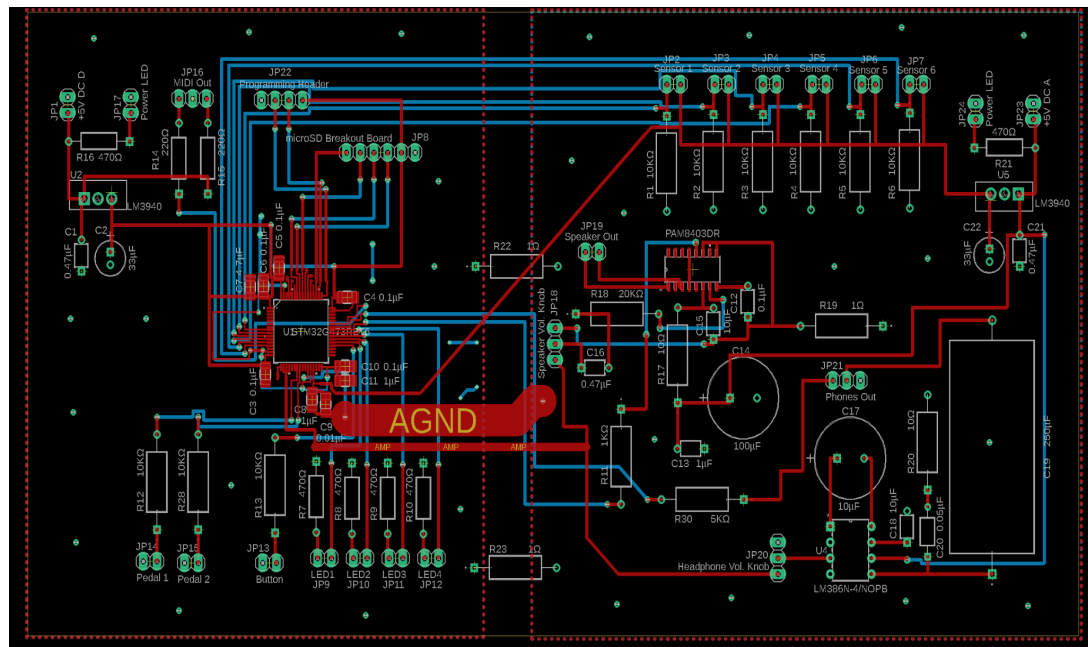


Figure 8.2 Final PCB Design
Image Source: Author

8.2 Enclosure Design

Once fabricated, the PCB was mounted and placed inside the enclosure. The general shape of the enclosure and position of the user IO had been decided, but the specific material used to construct the enclosure had yet to be determined. It was possible to use wood or 3D print the material, depending on which application was deemed best. This decision was to be made later, after testing was complete and the electrical design had been finalized. The method for labeling the user IO also had yet to be determined but was most likely to be done with a conventional label maker.

The enclosure that was used to hold the force sensors in the drum pads and cymbals had also yet to be decided. These enclosures needed to safely hold the sensors while allowing them to receive and transmit accurate readings as the user struck the drum kit components. Additionally, these enclosures needed to withstand the user's strikes during play, at least to a normal relaxed practice level of force. Lastly, they had to firmly hold the sensors in place so that pulling on the cables coming out of them did not damage, disconnect, or dislodge the sensors in any way.

8.3 Prototyping

Prototype construction for hardware was done using a basic breadboard and other available components to verify the function of major system components. This included components such as the speakers and the force sensors, among others. The speaker was tested using an existing amplifier circuit or by constructing one out of the available components on a breadboard, though this proved difficult as the amplifier ICs were SMD/SMT. The force sensors were tested by constructing a circuit that involved powering an LED and observing the LED turn on as the sensor was pressed down. The force sensors were further verified using a multimeter.

Other components were tested on a breadboard by constructing circuits that allowed for varying the voltage and current through them in specific ways, using indicators to ensure the desired effects were achieved. Some components were verified by measuring across them using different functions on a multimeter. Once it was confirmed that all components were working, they were also verified to work together appropriately.

9. System Testing & Evaluation

This section covers the overall testing of our device and how we ensured its functionality. It begins with the usual hardware and software testing and goes through the process to the final result ending with our future plans.

9.1 Hardware & Software Testing

In terms of testing, hardware was tested first to ensure that the parts ordered would function correctly when paired with the software. Since the software needed to be developed alongside at least some already functioning hardware to verify its correctness, it made sense to conduct major hardware testing initially. Some methods for this testing were described in the previous section. Hardware testing was also necessary to confirm that there was no need to reorder any components and that the circuit design was error-free, avoiding any risk of damage due to design flaws.

Both hardware and software design testing were conducted by subjecting the system to a variety of test states and situations. This approach ensured that the system did not damage itself, get stuck, or fail during any normal user scenario. Additionally, edge cases were a concern; it was essential to identify the possible edge cases of the system and test them to ensure that the system would still function properly if the user acted unexpectedly.

Software testing involved checking for edge cases as well as assessing the longevity of the program through extended use in various situations consecutively to ensure there were no issues, such as memory leaks, present in the code. This testing also verified that the software correctly interfaced with the hardware and that all external components were addressed appropriately. A specific example of something that needed to be tested was the force required to trigger sounds at varying levels of volume and velocity. Calibrating sounds based on how hard the user generally struck the drum pads and consequently pressed the force sensors needed to occur so that general force values could be categorized into different levels of velocity. This calibration had to be performed individually with each drum kit component, as a user might strike them with varying amounts of strength.

Senior Design 2:

Our prototype device was constructed out of black painted wood cut into the form of a box. Holes were cut in the side of the box to allow us to mount IO connections. The PCB was mounted to standoffs to keep it from contacting the interior sides of the box. Below is an image of the prototype.



Figure 9.1 Prototype Enclosure

Image Source: Author

In this image, the speaker and volume control knobs can be seen. Above the speaker are the indicator LEDs which tell the user which profile they currently have selected. The red button on the left side rotates through the built-in profiles. The jack on the left side is for the user to plug in their external audio device. Below is an image of another side of the device.



Figure 9.2 Prototype Enclosure Rear IO

Image Source: Author

This image shows the rear IO for our prototype. The multicolored cables coming out of the prototype are connected to the force sensors that we are using

to detect drum strikes. Next to that are two ¼" jacks that connect to the foot pedals used to control the kick drum and hi-hat. Finally next to that is the MIDI output jack which connects to the user's MIDI host. Below is an image of a top down view of the prototype with the top panel removed.

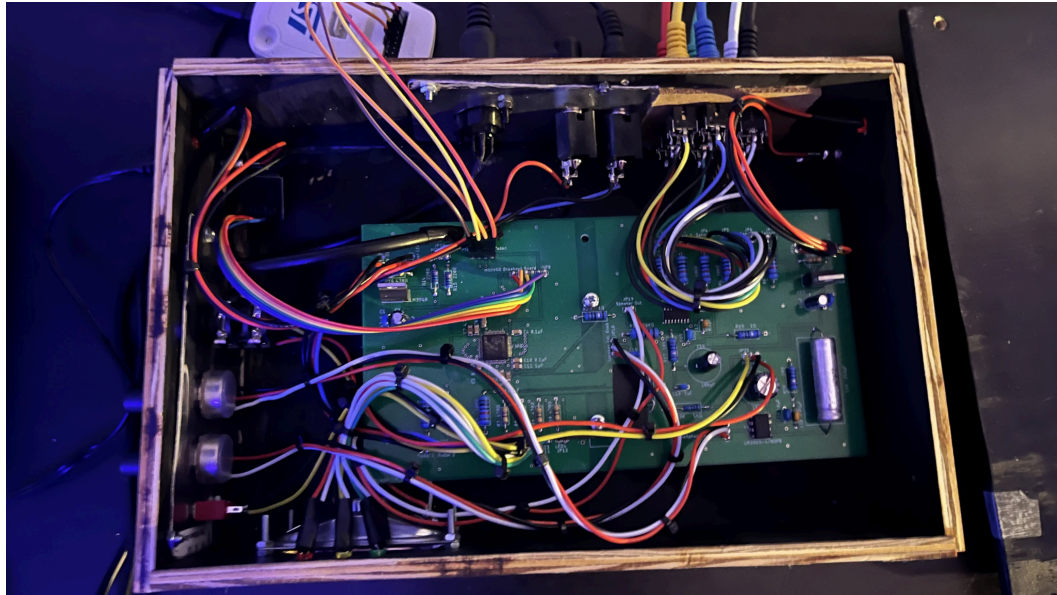


Figure 9.3 Prototype Enclosure Top View

Image Source: Author

This view of the device shows where we have mounted the PCB and how it is connected to the rest of the IO within the device. Also visible are the two power inputs for digital and analog power in the two top corners of the device, with their respective indicator LEDs. A final image below shows the sensors.



Figure 9.4 Force Sensor Peripherals

Image Source: Author

This image shows the force sensors used to detect drum strikes attached to the other end of the cables shown coming from the back of the device. These can be attached by the user to different surfaces to create drum pads so that the device can be played like an electric drum kit.

9.2 Performance Evaluation

Several aspects of the performance of our project needed to be evaluated, including the output volume from the speaker, the latency between striking the drum pad and hearing a sound, and the overall power consumption of the device. These individual factors could be tested in various ways and served as good indicators of the quality of our project and how well we executed our idea.

First, testing the speaker required the use of a decibel meter to verify that we reached the desired output volume. This volume needed to fill a room and remain usable and audible when other household devices were in use. While it was not intended for actual live events, it should have been sufficient for most household situations. Using a decibel meter from varying distances helped verify that the device performed loud enough for typical household use. For anything louder, the headphone jack could be used to bring sound directly to the user's ears.

The second aspect we tested was the latency from the moment a drum kit component was struck to the time a sound played from the device. We created a specific test fixture setup designed to strike the device and calculate the time between the impact on the drum pad and when a sound was picked up using a microphone. The intent behind minimizing latency was to prevent confusion for the user while playing. If there is a long delay between striking a note and hearing it, it can become disorienting for the player. A good simulation method involved using a MIDI controller with a DAW on a computer, digitally adding time between when a MIDI note was detected and when a sound played. Generally, latency over 100ms became noticeable and disorienting, while anything below that was less problematic. As long as we maintained latency within a comfortable range, it should not have been detectable by users. The goal of reducing latency focused more on achieving ease and comfort than reaching a specific performance metric.

The last testable metric we evaluated was the overall power consumption of the device. We used various devices for this testing. A digital multimeter measured the current passing through the DC power input of the device, which we multiplied by the +5V input power. Additionally, we utilized a device called a Kill-O-Watt to directly measure the wattage of power drawn from the wall before our AC/DC power converter. We did not expect to exceed our goal of using under 15W, as our overall circuit should not have consumed that much power. Nevertheless, this remained a valuable metric.

Senior Design 2:

While testing our design, we ran into some problems which caused us to have to redesign some of the project to provide functionalities to the user. The overarching problem with the design was our inability to get the microSD card to communicate over SPI. We tried different SPI breakout boards and despite our testing and manipulation of the code, the microSD card still refused to communicate properly with our microcontroller.

STM32 microcontrollers are known to be rather spotty with their SPI functionality and require particular connection characteristics. In light of this issue we decided that it would be easiest to move audio output to the user's device. Doing this also made it so that we no longer needed our amplifier circuits for the internal speaker and headphones. Instead we put the files on the user's device and triggered them with MIDI. Sadly this made us unable to test our audio output, but allowed us to focus on the implementation of other things, such as increasing the number of detectable velocity values.

9.3 Overall Integration

The end goal for the project was to create a drum kit consisting of a central processing hub and drum kit components as peripherals. Once powered, the microcontroller read incoming analog voltage signals from the force sensors, which rose when struck. When these signals rose to a certain threshold, the microcontroller polled the sensor and determined the value of the impact based on the maximum force value it detected. It then played a sound file with a velocity corresponding to the received voltage value. This sound was played through either an internal speaker or headphones that the user had plugged into the device. The device also outputs a MIDI signal for each note that was played. Foot pedals were used to determine when the kick drum was hit and whether the hi-hat was open or closed. A microSD card contained the sound files used by the program, and there were multiple different drum sound profiles so that the user could play sounds of their preferred style.

9.4 Future Plans

This was the first half of a two-semester project in which we completed the initial design of our device after coming up with an idea and conducting research to determine how to make it a reality. Our plans for the next semester involved testing our hardware and ensuring that it worked so we could get our PCB manufactured. Once we had the PCB, we would then develop the software to control our system, alongside the process of creating the enclosures and fixtures to hold the different parts of our system. Most of the hard work along the way consisted of figuring out how to implement solutions and troubleshooting any problems we encountered.

Senior Design 2:

If we had more time to continue working on this project, our goals would be to reevaluate the SPI situation and possibly adapt our board to work with ESP32, where we would most likely have better luck troubleshooting. We would have then also been able to test our internal speaker and amplifier circuits to determine if they work properly, such that the user could use the device standalone.

With more mechanical expertise, we could design a drum fixture that we could attach the sensor to in order to create a drum pad for the user. The issue currently is finding a way to suspend the sensor above something such that a strike would create a force against the sensor over a wider area. We would also have to create one for the cymbals. It may be easier to create a single design between both a drum and a cymbal shape.

10. Administrative Content

This section covers the Budget, Parts List, and Milestones for our project and what we were able to accomplish.

10.1 Budget Estimates

This project is being funded by the group members involved in designing and making it. The table below provides estimates for the cost of the materials required to complete the project. The project aims to create a low cost solution to the problem of creating a functional entry-level electric drum kit.

Part	Estimated Cost
PCB	\$30
Microcontroller	\$3
IO Jacks	\$3
Main Housing	\$20
Sensors	\$10
Pedals	\$15
Speaker	\$3
Amplifier Circuitry	\$3

Power Delivery Circuitry	\$5
Connecting Cables	\$15
Other Components (Indicators, Buttons)	\$3
Storage Device	\$5
Total	\$115

Table 10.1 Estimated Budget

Condensed BOM				
Component	Value	Notes	Qty.	Total Cost
Capacitor	0.01 μ F	SMD/SMT	1	\$0.10
Capacitor	0.1 μ F	SMD/SMT	5	\$0.50
Capacitor	1 μ F	SMD/SMT	2	\$0.17
Capacitor	4.7 μ F	SMD/SMT	1	\$0.27
Capacitor	0.05 μ F	Thru-Hole MLCC	1	\$0.36
Capacitor	0.1 μ F	Thru-Hole MLCC	2	\$0.36
Capacitor	0.47 μ F	Thru-Hole MLCC	2	\$0.62
Capacitor	1 μ F	Thru-Hole MLCC	2	\$0.66
Capacitor	10 μ F	Thru-Hole MLCC	1	\$0.54
Capacitor	10 μ F	Electrolytic	1	\$0.57
Capacitor	33 μ F	0.33 Ω - 0.5 Ω ESR Electrolytic	1	\$0.24
Capacitor	100 μ F	Electrolytic	1	\$0.26
Capacitor	250 μ F	Electrolytic	1	\$3.26
Resistor	1 Ω	Thru-Hole	1	\$0.03
Resistor	10 Ω	Thru-Hole	2	\$0.05

Resistor	220Ω	Thru-Hole	2	\$0.03
Resistor	470Ω	Thru-Hole	5	\$0.06
Resistor	10KΩ	Thru-Hole	10	\$0.50
Resistor	20KΩ	Thru-Hole	1	\$0.05
Potentiometer	10KΩ	Panel Mount	2	\$0.78
IC	PAM8403	PAM8403DR	1	\$1.13
IC	LM386	LM386N-4	1	\$1.52
Regulator	LM3940	TO-220 Package	1	\$1.62
Connector	6-Pin Male Header	None	1	\$0.08
Connector	6-Pin Female Header	None	1	\$0.07
Connector	4-Pin Male Header	None	1	\$0.05
Connector	Female DC Barrel Jack	Panel Mount	1	\$0.55
Connector	3.5mm TS Jack	Panel Mount	6	\$1.50
Connector	¼" TS Jack	Panel Mount	2	\$0.90
Connector	Female 5-Pin DIN	Panel Mount	1	\$2.00
Connector	Switched ¼" TS Jack	Panel Mount	1	\$0.49
Speaker	HY7040DS0830-H16.8	Panel Mount	1	\$1.75
Hardware	Potentiometer Knob	Volume Control	2	\$1.18
Hardware	3.5mm TS Cable	3ft Min.	6	\$20.00
Hardware	AC/DC Converter +5V DC 1.5A	+5V 1.5A Inner + Outer -	1	\$7.99
Hardware	microSD to SPI Breakout Board	6-Pin	1	\$2
Hardware	microSD Card	FAT32	1	\$10
Hardware	Momentary Button	Panel Mount	1	\$0.25
Hardware	Foot Pedal	None	2	\$15.00

Indicator	5mm/3mm LED	Thru-Hole	5	\$0.15
Sensor	MF01A-N-221-A01	None	6	\$23.70
MCU	STM32G473RET6	LQFP64 Package	1	\$8.20
Total				\$109.54

Table 10.3 Condensed BOM

10.3 Milestones

In order to stay on track our team has set certain milestones in accordance with what is due throughout the course. Designing and testing is hard to foresee accurately and will probably vary in reality compared to what is listed in this table. Hard deadlines should stay the same unless changed by the SD1 coordinators. These milestones cover only SD1 and up to the end of the Fall 2024 semester.

Time	Milestone
Week 3	Finish D&C document
Week 4	Meet with SD coordinators and begin Draft Paper
Week 5	Research specific parts, refine theoretical design
Week 6	Work on draft paper, select final components and begin design
Week 7-9	Testing components, rough PCB design, draft paper
Week 10	Submit first draft paper, continue testing/designing
Week 11-12	Submit draft paper revision
Week 13-14	Submit final draft
Week 15	Submit mini demo video

Table 10.4 Milestone Schedule

10.4 Work Distributions

Task	Primary	Secondary
Kit Assembly and Housing Design	Wylde	Soufiane
Instrument Sensor and Housing Design	Soufiane	Wylde
Sound Library Storage	Soufiane	Jameson
Connection between Sensors and Control Board	Soufiane	Wylde
Strike Level Detection	Jameson	Wylde
Sound Library Instrument Selection	Jameson	Soufiane
Sensor Instrument Assignment	Wylde	Jameson
Sound Library Mode Switching	Jameson	Soufiane
Auxiliary port passthrough	Wylde	Soufiane
MIDI cable data path	Jameson	Wylde
Control Board PCB Layout	Wylde	Jameson
Power Supply	Soufiane	Jameson
Control Board Software	Jameson	Wylde

Table 10.5 Work Distributions

Declaration: We hereby declare that we have not copied more than 7 pages from the Large Language Model (LLM). We have not utilized LLM for any purpose at this time.

11. Conclusion

Over the course of this paper, we moved from the concept of a project to conducting full research on how to execute it. We then selected the necessary hardware to create it and finally designed a circuit along with the concepts of a physical structure and software program to drive it. This process required extensive research and significant amounts of time to understand, and the project has yet to be proven functional. Although we accomplished a great deal of work on this project, we were still only halfway through the entire process and had a whole semester remaining to finalize and construct the design, as well as to reach a functional and presentable state.

In the next semester, we planned to work on our program and test it on the actual hardware we intended to use, as well as to complete the final PCB design and have it manufactured. During this process, we would determine how to construct the fixtures for the drums and the enclosures to hold the sensors in place to prevent damage or disconnection. This would take place over the course of several months, and we anticipated encountering some challenges along the way. With the knowledge we acquired over the past semester, we hoped to allocate our time more wisely, ensuring we had a safety net in case something went awry. Previously, we were uncertain about how much time to allocate to specific parts of the research; however, we had conducted an analysis of the constraints we needed to address and how they would affect us.

One of the most challenging aspects of the research and design process was determining which microcontroller to use for the project. We needed to cross-reference various types of microcontrollers and their capabilities to identify one with all the necessary features. This also involved researching how microcontrollers perform specific tasks. Once we understood how they executed those tasks, we could search for microcontrollers with the required specifications.

Another challenging aspect was figuring out how to safely input and output signals to and from the microcontroller. This primarily involved protecting the microcontroller from sourcing or sinking excessive current. This required a thorough examination of the datasheets and manuals for the microcontroller to determine its capabilities. We then designed circuits around these specifications to ensure our circuit operated within the defined limits. Even as we realized our circuit, we still needed to conduct real-world testing to confirm whether our calculations were accurate.

We predicted that the next semester would be easier than the previous one now that our idea was established. With a clear understanding of our goals, we had much more direction within the project and should have been able to outline our objectives and achieve them early on, rather than having to determine what steps were needed to realize our ideas.

This semester proved to be very informative in illustrating what it is like to work with other engineers, particularly those from different disciplines. Dividing tasks so that everyone could focus on their strengths was beneficial; however,

more issues arose when we needed to integrate these ideas. This experience taught us the importance of engineers having a baseline knowledge of other disciplines, facilitating communication and providing insight into what is feasible in each other's fields. Otherwise, one of us might have proposed an implementation idea and designed our part of the project around it, only to later realize that it was not a viable solution.

Overall, we were confident in how our project had developed and were prepared to enter the next semester to finalize our design while accomplishing what was necessary to verify our implementation. With our newfound knowledge, we believed we could organize ourselves effectively to complete the required tasks in a more timely manner.

Appendix

References:

42 Piezo Buzzer Images, Stock Photos, 3D Objects, & Vectors | Shutterstock, Shutterstock, www.shutterstock.com/search/piezo-buzzer.

Campbell, Scott. "Basics of the SPI Communication Protocol." *Circuit Basics*, circuitbasics.com, 14 Nov. 2021, www.circuitbasics.com/basics-of-the-spi-communication-protocol/.

EmbeTronicX. "SD Card Interfacing with STM32 - STM32 SPI." *YouTube*, YouTube, 3 Oct. 2022, www.youtube.com/watch?v=xbLkp_pTWbo.

"General MIDI Drum Notes." *MIDI Drum Chart*, zendrum.com, www.zendrum.com/resource-site/drumnotes.htm.

"Headphones v1.8 - Graph Tool." *RTINGS.Com*, rtings.com, www.rtings.com/headphones/1-8/graph/25500/raw-fr-l/audio-technica-ath-m50x/295.

"How to Interface SD Card Using SPI in STM32." *ControllersTech*, controllerstech.com, 24 Sept. 2024, controllerstech.com/sd-card-using-spi-in-stm32/.

"Lm386." Texas Instruments, <https://www.ti.com/lit/ds/symlink/lm386.pdf?ts=1732638357489>

"Lm3940." Texas Instruments, <https://www.ti.com/lit/ds/symlink/lm3940.pdf>

"PAM8403-247318." Diodes Incorporated, <https://www.mouser.com/ds/2/115/PAM8403-247318.pdf?srltid=AfmBOoq5JOmVQgSXmgj7009gIm42bYmWdBN9q8-EsHbHalkg35UzcRLQ>

Predictable Designs. "Tutorial: How to Design Your Own Custom Microcontroller Board." *YouTube*, YouTube, 15 May 2018, www.youtube.com/watch?v=tgUx0Gm7-RM.

Predictable Designs. "Tutorial - How to Design Your Own Custom Microcontroller Board (Part 2)." *YouTube*, YouTube, 22 May 2018, www.youtube.com/watch?v=jN6rc2Zd-h8.

"R/Askelectronics on Reddit: What Does This RC Series Do in This Amplifier? Does It Change the Sound If I Vary These Values?" *Reddit*,

reddit.com,
www.reddit.com/r/AskElectronics/comments/wjc8md/what_does_this_rc_series_do_in_this_amplifier/.

“RM0440 Rev 8, STM32G4 series advanced Arm-based 32-bit MCUs.”
STMicroelectronics, February 2024.

“AN3126 Rev 4, Audio and waveform generation using the DAC.”
STMicroelectronics, July 2020.

“PM0214 Rev 10, M4 MCUs and MPUs programming manual.”
STMicroelectronics, March 2020.

Saleem, Kiran. “Simple PAM8403 Audio Amplifier Circuit.” *Circuits DIY*,
circuitsdiy.com, 30 July 2024,
www.circuits-diy.com/simple-pam8403-audio-amplifier-circuit/.

Science Buddies. “How to Use a Force Sensor with an Arduino (Lesson #23).” *YouTube*, 25 Sept. 2023,
youtu.be/r7oWtcE6QQc?si=Yjr-jmNfqbtKnLAW.

self-study electrical engineer. “SD CARD READER With STM32.” *YouTube*,
4 Dec. 2020, youtu.be/fRfRwMyNE7A?si=XeSMHynNSSWSymXb.

“SPI (Serial Peripheral Interface).” *SPI (Serial Peripheral Interface) - UTAT Space Systems Documentation 1.0 Documentation*, UTAT Space Systems,
utat-ss.readthedocs.io/en/master/communication-protocols/spi.html.

“Stm32g473cb.” STMicroelectronics,
<https://www.st.com/resource/en/datasheet/stm32g473cb.pdf>.

“STM32L562E-DK.” *STMicroelectronics*,
www.st.com/en/evaluation-tools/stm32l562e-dk.html#st_all-features_sec-n
av-tab.

“The MIDI Specification.” *MIDI Specification*,
people.ece.cornell.edu/land/courses/ece4760/FinalProjects/s2000/selan/midi.html.

RTINGS Terms of Use:

3.2 ACCESSING THE MATERIAL

User is entitled to access and print, all material published on the Site, including without limitation documentation, text, pictures, sounds, graphics, articles, video's and audio clips (collectively the "Material") for User's personal private and non-commercial use only, provided that User does not:

- (a) download or print any Material in a systematic or regular manner to create a database (electronic or paper form);
- (b) remove any notices relating to the ownership of copyright or other intellectual property rights in the Material;
- (c) modify, translate, reverse engineer, reproduce, decompile, disassemble or create derivative works of any of the Material;
- (d) rent, lease, sub-license, loan, copy, commercially exploit or give or transfer any rights in the Material in any form, to any person or entity without RTINGS prior written consent.

RTINGS status as the owner of the Material must always be acknowledged. If User prints off or otherwise copies any part of the Site or Material in breach of these Terms, User's rights to use the Site will cease immediately and User must return or destroy any copies of the Material.

Appendix - Codes

“main.h” :

```
/* USER CODE BEGIN Header */
/**

*****
*****
* @file      : main.h
* @brief     : Header for main.c file.
*            : This file contains the common defines
of the application.

*****
*****
* @attention
*
* Copyright (c) 2025 STMicroelectronics.
* All rights reserved.
*
* This software is licensed under terms that can be found
in the LICENSE file
* in the root directory of this software component.
* If no LICENSE file comes with this software, it is
provided AS-IS.
*

*****
*****
*/
/* USER CODE END Header */

/* Define to prevent recursive inclusion
-----*/
#ifndef __MAIN_H
#define __MAIN_H
```

```

#ifdef __cplusplus
extern "C" {
#endif

/* Includes
-----
-----*/
#include "stm32g4xx_hal.h"

/* Private includes
-----*/
/* USER CODE BEGIN Includes */

/* USER CODE END Includes */

/* Exported types
-----
*/
/* USER CODE BEGIN ET */

/* USER CODE END ET */

/* Exported constants
-----*/
/* USER CODE BEGIN EC */

/* USER CODE END EC */

/* Exported macro
-----
*/
/* USER CODE BEGIN EM */

/* USER CODE END EM */

/* Exported functions prototypes
-----*/
void Error_Handler(void);

```

```

/* USER CODE BEGIN EFP */

/* USER CODE END EFP */

/* Private defines
-----*
/
#define Sensor_1_Pin GPIO_PIN_0
#define Sensor_1_GPIO_Port GPIOC
#define Sensor_2_Pin GPIO_PIN_1
#define Sensor_2_GPIO_Port GPIOC
#define Sensor_3_Pin GPIO_PIN_2
#define Sensor_3_GPIO_Port GPIOC
#define Sensor_4_Pin GPIO_PIN_3
#define Sensor_4_GPIO_Port GPIOC
#define Sensor_5_Pin GPIO_PIN_0
#define Sensor_5_GPIO_Port GPIOA
#define Sensor_6_Pin GPIO_PIN_1
#define Sensor_6_GPIO_Port GPIOA
#define Pedal_1_Pin GPIO_PIN_5
#define Pedal_1_GPIO_Port GPIOC
#define Pedal_2_Pin GPIO_PIN_0
#define Pedal_2_GPIO_Port GPIOB
#define Button_Pin GPIO_PIN_14
#define Button_GPIO_Port GPIOB
#define LED_1_Pin GPIO_PIN_15
#define LED_1_GPIO_Port GPIOB
#define LED_2_Pin GPIO_PIN_6
#define LED_2_GPIO_Port GPIOC
#define LED_3_Pin GPIO_PIN_7
#define LED_3_GPIO_Port GPIOC
#define LED_4_Pin GPIO_PIN_8
#define LED_4_GPIO_Port GPIOC
#define Amplifier_Pin GPIO_PIN_9
#define Amplifier_GPIO_Port GPIOC
#define Headphone_Pin GPIO_PIN_8
#define Headphone_GPIO_Port GPIOA

```

```

#define MIDI_Pin GPIO_PIN_9
#define MIDI_GPIO_Port GPIOA
#define SD_CS_Pin GPIO_PIN_15
#define SD_CS_GPIO_Port GPIOA

/* USER CODE BEGIN Private defines */

// configurable defines
#define PROFILES_FOLDER "profiles" // name of the root
folder containing the wav file assets
#define NUM_SAMPLES_FOR_PEAK 20 // MUST BE GREATER
THAN 0. (uint8_t value) how many samples to determine the
highest value of a strike.
#define SENSOR_MAX 4095 // highest value
reading from the 12-bit ADC
#define SENSOR_CUTOFF 1000 // lowest value
reading to count as a strike
#define WAV_FILES_MAX 8 // MUST BE GREATER
THAN 0. limit of files in the software mixer.
#define PROFILES_MAX 3 // MUST BE GREATER
THAN 0. limit of profiles to create.
#define BUFFER_SIZE 256 // MUST BE GREATER
THAN 0. for audio data, number of samples in each buffer.

// NOT CONFIGURABLE
#define SD_SPI_HANDLE hspi1 // needed for fatfs
SD card over SPI

/* USER CODE END Private defines */

#ifdef __cplusplus
}
#endif

#endif /* __MAIN_H */

```

“main.c” :

```
/* USER CODE BEGIN Header */
/**

*****
*****
* @file      : main.c
* @brief     : Main program body

*****
*****
* @attention
*
* Copyright (c) 2025 STMicroelectronics.
* All rights reserved.
*
* This software is licensed under terms that can be found
in the LICENSE file
* in the root directory of this software component.
* If no LICENSE file comes with this software, it is
provided AS-IS.
*

*****
*****

*/

/* USER CODE END Header */
/* Includes
-----
-----*/
#include "main.h"
#include "app_fatfs.h"

/* Private includes
-----*/
```

```

/* USER CODE BEGIN Includes */

#include "stdio.h"
#include "stdlib.h"
#include "stdint.h"
#include "string.h"
#include "stdbool.h"

/* USER CODE END Includes */

/* Private typedef
-----*/
/
/* USER CODE BEGIN PTD */

/* Audio file information structure */
typedef struct
{
    uint32_t SampleRate;      /* Audio sampling frequency
*/
    uint32_t NumChannels;     /* Number of channels:
1:Mono or 2:Stereo */
    uint32_t BitsPerSample;   /* Number of bits per
sample (16, 24 or 32) */
    uint32_t AudioLength;     /* Total length of useful
audio data (payload in Bytes) */
    uint32_t AudioStartAddr;  /* Relative start address
of audio payload */
    uint32_t RIFFchunksize;   /* The file header chunk
size */
    uint16_t FormatTag;       /* Audio file format: PCM =
1 */
    uint32_t ByteRate;        /* Number of bytes per
second (sample rate * block align) */
    uint16_t BlockAlign;      /* channels * bits/sample /
8 */
    uint32_t DataSize;        /* Audio data size */
} WAVE_FormatTypeDef;

```

```

// track opened files instead of using f_open() f_close()
many times
typedef struct OpenFile
{
    char full_path[256];           // full path to the file
    FIL* file_ptr;                 // file pointer (file
handle)
    uint32_t hash;                 // compare hash values
instead of strings
    struct OpenFile* next;        // pointer to the next open
file
} OpenFile;

// notes sent to MIDI output
typedef struct
{
    uint8_t midi_channel;         // separate profiles among
channels
    uint8_t midi_code;            // MIDI note (e.g., 60 for
middle C)
    uint8_t velocity;             // note velocity
    uint16_t duration;            // duration of the note (in
ms)
    uint16_t start_time;          // time when the note was
turned on (in ms or ticks)
    bool isPlaying; // true when initially added, false
after time expire
} midi_note_t;

// files in software mix
typedef struct
{
    FIL* wavFile;                 // file object for WAV file
access
    int16_t* buffer;              // buffer to hold file data
    uint32_t position;            // current read position in
the file

```

```

    bool isPlaying; // true when file is added to mixer,
false when EOF
} wav_file_buffer_t;

// place into the strike queue for translating to the
appropriate midi-wav pair
typedef struct
{
    uint8_t sensor_number;    // which sensor the reading
is for
    uint16_t peak_value;     // approximated result for
the highest ADC reading of the strike
} sensor_data_t;

// hold WAV file i/o information and its paired MIDI code
typedef struct
{
    char* filename;          // filename of a file
inside the instrument folder
    uint32_t start_addr;     // relative start address
of audio payload
    uint8_t midi_channel;    // MIDI channel
    uint8_t midi_code;       // MIDI code
assignment
    uint16_t duration;       // duration of the
audio (in ms)
} midi_wav_t;

// hold instrument data
typedef struct
{
    char* name;              // instrument folder name
inside the corresponding profile folder
    midi_wav_t* wav_files;   // array of midi-wav pairs
    uint8_t num_waves;       // number of valid WAV
files in this instrument (should be greater than 0)
} instrument_t;

```



```

// hold profile data
typedef struct
{
    char* name;                // profile folder name
    inside the "/profiles/" folder
    instrument_t* instruments; // array containing each
    instrument's data
    uint8_t num_instruments;   // number of instruments in
    this profile (should be 8 for valid profile)
    bool midi_code_used[128]; // track used MIDI codes
    for this profile (range 0-127)
} profile_t;

typedef struct {
    sensor_data_t buffer[WAV_FILES_MAX];
    uint16_t head;
    uint16_t tail;
    uint16_t count;
} PlayQueue_t;

// handle button press detection. useful for profile select
button and kick pedal switches.
typedef enum
{
    BUTTON_IDLE,    // no press, waiting for the button
    press
    BUTTON_PRESSED, // button has been pressed
    BUTTON_RELEASED // button is released
} ButtonState;

/* USER CODE END PTD */

/* Private define
-----
*/
/* USER CODE BEGIN PD */

```

```

// MIDI note constants
#define NOTE_ON  0x90  // Note On message type
#define NOTE_OFF 0x80  // Note Off message type

/* Error Identification structure */
#define VALID_WAVE_FILE          0    /** @brief Valid
Wave File TAG */
#define INVALID_RIFF_ID          1    /** @brief
Invalid RIFF ID */
#define INVALID_WAVE_FORMAT      2    /** @brief
Invalid Wave Format */
#define INVALID_FORMAT_CHUNK_ID  3    /** @brief
Invalid Format Chunk ID */
#define UNSUPPORTED_FORMAT_TAG   4    /** @brief Valid
Wave File TAG */
#define UNSUPPORTED_NUM_CHANNELS  5    /** @brief Valid
Wave File TAG */
#define UNSUPPORTED_SAMPLE_RATE  6    /** @brief Valid
Wave File TAG */
#define UNSUPPORTED_BITS_PER_SAMPLE 7    /** @brief Valid
Wave File TAG */
#define INVALID_DATA_CHUNK_ID    8    /** @brief Valid
Wave File TAG */
#define UNSUPPORTED_EXTRA_FORMAT_BYTES 9    /** @brief Valid
Wave File TAG */
#define INVALID_FACT_CHUNK_ID    10   /** @brief Valid
Wave File TAG */

#define LITTLE_ENDIAN            0
#define BIG_ENDIAN               1

/* Audio Parsing Constants */
#define CHUNK_ID                 0x52494646 /* correspond to the
letters 'RIFF' */
#define FILE_FORMAT              0x57415645 /* correspond to the
letters 'WAVE' */

```

```

#define FORMAT_ID          0x666D7420  /* correspond to the
letters 'fmt ' */
#define DATA_ID           0x64617461  /* correspond to the
letters 'data' */
#define FACT_ID            0x66616374  /* correspond to the
letters 'fact' */
#define WAVE_FORMAT_PCM    0x01
#define FORMAT_CHUNK_SIZE  0x10
#define BITS_PER_SAMPLE_8  8
#define BITS_PER_SAMPLE_16 16

#define WAV_HEADER_MAX_SIZE 200  /* 44 */
#define SAMPLE_RATE 44100        // 44.1 kHz sample
rate

#define NUM_SENSORS 6            // total number of
sensors connected to ADCs
#define DIRTY_FLAG 65535        // value of uint16_t max
since the 12-bit ADC max is lower

// #define TIM4_HALF_VALUE 181    // TIM4 must be set
to a 44100 Hz delay using APB1 clock freq.

/* USER CODE END PD */

/* Private macro
-----
-*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables
-----*/
ADC_HandleTypeDef hadc1;

DAC_HandleTypeDef hdac1;

```

```

SPI_HandleTypeDef hspi1;

TIM_HandleTypeDef htim3;
TIM_HandleTypeDef htim4;

UART_HandleTypeDef huart1;

/* USER CODE BEGIN PV */

FATFS FatFs;
// Fatfs handle
bool sd_card_presence = false; //
system status flag
ButtonState profile_button_state = BUTTON_IDLE;
    // state machine for button press (switch)
ButtonState kick_pedal_state = BUTTON_IDLE;
    // state machine for kick pedal (switch)
OpenFile* open_file_list = NULL;
// linked list to keep track of open files
profile_t* profiles = NULL;
    // array to hold all profile information
uint8_t num_profiles = 0;
    // how many profiles are built
uint8_t active_profile = 0;
    // the index used for profiles array
wav_file_buffer_t wavFiles[WAV_FILES_MAX];
// array to hold active files
midi_note_t active_notes[WAV_FILES_MAX];
// array to hold active notes

PlayQueue_t playQueue;
    // hold and transfer sensor data
bool playQueue_lock = false; // 0
= unlocked, 1 = locked

// Buffers for audio samples
int16_t audio_buffer_1[BUFFER_SIZE];
int16_t audio_buffer_2[BUFFER_SIZE];

```

```

int8_t switch_buffer = 0;                                // 0 =
buffer 1, 1 = buffer 2

uint16_t sensor_range = (SENSOR_MAX - SENSOR_CUTOFF);

//bool led_blinking = false;
// flag for LED blinking one time task

/* USER CODE END PV */

/* Private function prototypes
-----*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_ADC1_Init(void);
static void MX_DAC1_Init(void);
static void MX_SPI1_Init(void);
static void MX_TIM4_Init(void);
static void MX_USART1_UART_Init(void);
static void MX_TIM3_Init(void);
/* USER CODE BEGIN PFP */

// BARE METAL HELPERS
uint16_t ms_timer(void);
void ms_delay(uint16_t ms);
void Queue_Init(PlayQueue_t *q);
bool enqueue(PlayQueue_t *q, sensor_data_t data);
bool dequeue(PlayQueue_t *q, sensor_data_t *data);

// SOFTWARE MIXER
bool add_wav_file_to_mix(FIL* newFile, uint32_t start_addr);
bool reset_all_wavFiles(void);
void read_next_chunk(wav_file_buffer_t* wav_file_buffer);

```

```

// MIDI
void send_midi_note_on(uint8_t midi_channel, uint8_t
midi_code, uint8_t velocity);
void send_midi_note_off(uint8_t midi_channel, uint8_t
midi_code);
void start_midi_note(midi_wav_t* data, uint8_t velocity);
void reset_all_midi_notes(void);

// PROFILE MANAGEMENT
uint32_t ReadUnit(uint8_t NbrOfBytes, uint32_t BytesFormat,
uint8_t* pBuff, uint32_t BuffIdx);
uint32_t WavProcess_ParseHeader(uint8_t* pHeader,
WAVE_FormatTypeDef* pAudioFormatDataStruct);
void all_LEDs_on(void);
void all_LEDs_off(void);
void set_LEDs(void);
void set_active_profile(uint8_t profile_index);
void process_instrument_dir(char* instrument_path,
instrument_t* instrument, profile_t* profile);
void process_profile_dir(char* profile_path, char*
profile_name);
void create_profiles(void);
void create_profiles_midi_only(void);
void free_profiles(void);
uint32_t murmurhash3_32(char *key, uint32_t len, uint32_t
seed);
OpenFile* find_open_file(uint32_t hash);
FIL* add_open_file_handle(char* full_path);
void reset_all_open_file_handles(void);
char* get_full_path(uint8_t instr_index, uint8_t wav_index);
void translate_strike_to_profile(sensor_data_t strike);

```

```

// SD CARD
void sd_card_mount(void);
void sd_card_umount(void);
//bool sd_card_check_presence(void);

// ADC SENSOR ARRAY
//uint32_t read_ADC_value(uint32_t channel);

// TASK SEPARATION
void headphone_detect(void);
//void output_to_DAC(void);
void audio_mix(void);
void expired_notes(void);
void play_queue(void);
void profile_button(void);
//void sd_card(void);
void sensor_array(void);
void kick_pedal(void);

/* USER CODE END PFP */

/* Private user code
-----*/
/* USER CODE BEGIN 0 */

// NOTE: count resets after about one minute
uint16_t ms_timer(void)
{
    return (uint16_t)TIM3->CNT;
}

```

```

// implement bare metal delay
// NOTE: blocking busy-waiting function
void ms_delay(uint16_t delay)
{
    uint16_t start_time = ms_timer();
    uint16_t current_time;

    while (true) {
        current_time = ms_timer();
        if ((uint16_t)(current_time - start_time) >= delay) {
            break;
        }
    }
}

void Queue_Init(PlayQueue_t *q)
{
    q->head = 0;
    q->tail = 0;
    q->count = 0;
}

bool enqueue(PlayQueue_t *q, sensor_data_t data)
{
    while (__atomic_test_and_set(&playQueue_lock,
__ATOMIC_ACQUIRE)) {

        if (q->count == WAV_FILES_MAX) {
            __atomic_clear(&playQueue_lock, __ATOMIC_RELEASE);
            return false; // Queue full
        }
        q->buffer[q->head] = data;
        q->head = (q->head + 1) % WAV_FILES_MAX;
        q->count++;
        __atomic_clear(&playQueue_lock, __ATOMIC_RELEASE);
        return true;
    }
}

```



```

}

bool dequeue(PlayQueue_t *q, sensor_data_t *data)
{
    while (__atomic_test_and_set(&playQueue_lock,
__ATOMIC_ACQUIRE)) {

        if (q->count == 0) {
            __atomic_clear(&playQueue_lock, __ATOMIC_RELEASE);
            return false; // Queue empty
        }
        *data = q->buffer[q->tail];
        q->tail = (q->tail + 1) % WAV_FILES_MAX;
        q->count--;
        __atomic_clear(&playQueue_lock, __ATOMIC_RELEASE);
        return true;
    }

    // add a new wav file into the software mixer. files are
    opened and placed into the file handle
    bool add_wav_file_to_mix(FIL* newFile, uint32_t start_addr)
    {
        // try to take the mutex (wait for the lock)
        if (osMutexAcquire(wavFiles_mutexHandle,
osWaitForever) == osOK) {
            for (int i = 0; i < WAV_FILES_MAX; i++) {
                if (!wavFiles[i].isPlaying) {
                    // can add the file at this index
                    wavFiles[i].wavFile = newFile;

                    // verify the buffer exists, redundant
                    check
                    if (!wavFiles[i].buffer) {
                        //
                        xSemaphoreGive(wavFiles_mutex);
                        printf("Memory allocation error: NULL at
wavFile buffer %d\r\n", i);

```

```

        // attempt to allocate the buffer as a fix
        // NOTE: may indicate a greater problem if
we reach this code. may cause memory leak?
        wavFiles[i].buffer =
(int16_t*)malloc(BUFFER_SIZE * sizeof(int16_t));
        if (wavFiles[i].buffer == NULL) {
            printf("Memory allocation error:
wavFile buffer %d\r\n", i);
            while (true) {

                }
            }
        }

        // clear the existing buffer to 0 (silence)
in terms of signed int16_t
        memset(wavFiles[i].buffer, 0, BUFFER_SIZE *
sizeof(int16_t));
        // starting point for audio data after
header
        //          wavFiles[i].position =
0x50;

        wavFiles[i].position = start_addr;
        // allow mixing after everything else has
been set properly
        wavFiles[i].isPlaying = true;

        osMutexRelease(wavFiles_mutexHandle);
        return true;
    }
}
// array is full, failed to add
osMutexRelease(wavFiles_mutexHandle);
printf("error: add_wav_file_to_mix() could not add to
array. no spot available?\r\n");
}
return false;
}

```

```

// flag as not playing to mark all spots for overwrite.
useful between SD card mounts.
bool reset_all_wavFiles(void)
{
    // try to take the mutex (wait for the lock)
    if (osMutexAcquire(wavFiles_mutexHandle,
osWaitForever) == osOK) {
        for (int i = 0; i < WAV_FILES_MAX; i++) {
            wavFiles[i].isPlaying = false;
        }
        osMutexRelease(wavFiles_mutexHandle);
        return true;
    }
    return false;
}

// fill wavFile buffer with data from the wav file. useful
in software mix.
void read_next_chunk(wav_file_buffer_t* wav_file_buffer)
{
    FRESULT fres;
    size_t bytesRead;

    // move to the desired position in the file
    fres = f_lseek(wav_file_buffer->wavFile,
wav_file_buffer->position);
    if (fres != FR_OK) {
        printf("error: read_next_chunk() could not access file
for buffering.\r\n");
        wav_file_buffer->isPlaying = false; // stop playback
if there's an error or EOF
        // clear the entire buffer before letting it mix
        memset(wav_file_buffer->buffer, 0, BUFFER_SIZE *
sizeof(int16_t));
        return;
    }
}

```

```

        // read the next chunk into the buffer, overwriting
old data
        fres = f_read(wav_file_buffer->wavFile,
wav_file_buffer->buffer, BUFFER_SIZE * sizeof(int16_t),
&bytesRead);
        // if no data is read or fewer bytes than expected
assume end of file or the file is smaller
        if (fres != FR_OK || bytesRead == 0 || bytesRead <
BUFFER_SIZE * sizeof(int16_t)) {
            wav_file_buffer->isPlaying = false; // stop playback
if there's an error or EOF
            // attempt the last iteration again with a zeroed
buffer
            memset(wav_file_buffer->buffer, 0, BUFFER_SIZE *
sizeof(int16_t));
            fres = f_lseek(wav_file_buffer->wavFile,
wav_file_buffer->position);
            if (fres != FR_OK) {
                printf("error: read_next_chunk() could not access
file for buffering.\r\n");
                return;
            }
            // attempt to fill the buffer without checking
            f_read(wav_file_buffer->wavFile,
wav_file_buffer->buffer, BUFFER_SIZE * sizeof(int16_t),
&bytesRead);
            return;
        }

        // update the position in the file
wav_file_buffer->position += bytesRead;

        // if the position reaches the end of the file, mark
the wavFile as not playing
        // if (wav_file_buffer->position >=
f_size(wav_file_buffer->wavFile)) {
            // wav_file_buffer->isPlaying = false; //
Stop playback if there's an error or EOF

```

```

        //    }
    }

    // send MIDI "note on" message with the given code and
    // velocity values.
    void send_midi_note_on(uint8_t midi_channel, uint8_t
    midi_code, uint8_t velocity)
    {
        //    uint8_t midi_message[3] = { NOTE_ON, midi_code,
    velocity };
        uint8_t midi_message[3] = { (uint8_t)(NOTE_ON +
    midi_channel), midi_code, velocity };
        HAL_UART_Transmit(&huart1, midi_message, 3, 1000); //
    send "note on" message
    }

    // send MIDI "note off" message for the given code.
    // NOTE: most recording software need a "note off" for every
    // "note on". extra "note off" messages should be fine.
    void send_midi_note_off(uint8_t midi_channel, uint8_t
    midi_code)
    {
        uint8_t midi_message[3] = { (uint8_t)(NOTE_OFF +
    midi_channel), midi_code, 0 }; // Note Off message with
    velocity 0
        HAL_UART_Transmit(&huart1, midi_message, 3, 1000); //
    Send "Note Off" message
    }

    // start a MIDI note with duration (non-blocking).
    void start_midi_note(midi_wav_t* data, uint8_t velocity)
    {
        for (uint8_t i = 0; i < WAV_FILES_MAX; i++) {
            // find an empty slot to store the note
            if (!active_notes[i].isPlaying) {
                active_notes[i].midi_channel =
    data->midi_channel;
                active_notes[i].midi_code = data->midi_code;
            }
        }
    }

```

```

        active_notes[i].velocity = velocity;
        active_notes[i].duration = data->duration;
        active_notes[i].start_time = ms_timer(); //
current time in ticks

        // send "note on" message
        send_midi_note_on(data->midi_channel,
data->midi_code, velocity);
        active_notes[i].isPlaying = true;
        break;
    }
}

// clear all current notes. useful for profile switch or SD
card removed.
void reset_all_midi_notes(void)
{
    for (uint8_t i = 0; i < WAV_FILES_MAX; i++) {
        if (active_notes[i].isPlaying) {
            active_notes[i].isPlaying = false;
            send_midi_note_off(active_notes[i].midi_channel,
active_notes[i].midi_code);
        }
    }
}

/**
 * @brief Reads a number of bytes from a buffer and return
them
 * @param NbrOfBytes : number of bytes to read. This
parameter must be a
 *          number between 1 and 4.
 * @param BytesFormat : specifies the bytes endianness.
 *          This parameter can be one of the following
values: LITTLE_ENDIAN or
 *          BIG_ENDIAN
 * @param pBuff : memory address to read from.

```

```

    * @param BuffIdx : Buffer index.
    * @retval Bytes read from the buffer..
    */
uint32_t ReadUnit(uint8_t NbrOfBytes, uint32_t BytesFormat,
uint8_t* pBuff, uint32_t BuffIdx)
{
    uint32_t index = 0;
    uint32_t tmp = 0;
    uint32_t Idx = BuffIdx;

    if (BytesFormat == LITTLE_ENDIAN) {
        for (index = 0; index < NbrOfBytes; index++) {
            tmp |= pBuff[Idx++] << (index * 8);
        }
    }
    else {
        for (index = NbrOfBytes; index != 0; index--) {
            tmp |= pBuff[Idx++] << ((index - 1) * 8);
        }
    }

    return tmp;
}

/**
 * @brief Parse the wave file header and checks if it is
valid
 * @param pHeader: pointer to the wave file header table.
 * @param pAudioFormatDataStruct: Pointer to the wave
structure to be filled.
 * @retval 0 if wave file is valid, !0 else.
 */
uint32_t WavProcess_ParseHeader(uint8_t* pHeader,
WAVE_FormatTypeDef* pAudioFormatDataStruct)
{
    /* .WAV file format :
-----

```

	Endian	Offset	Length	Contents	
	big	0	4 bytes	'RIFF'	--
0x52494646					
	little	4	4 bytes	<file length - 8>	
	big	8	4 bytes	'WAVE'	--
0x57415645					

Next, the fmt chunk describes the sample format:

	big	12	4 bytes	'fmt '	--
0x666D7420					
	little	16	4 bytes	0x00000010	--
Length of the fmt data (16 bytes)					
	little	20	2 bytes	0x0001	--
Format tag: 1 = PCM					
	little	22	2 bytes	<channels>	--
Channels: 1 = mono, 2 = stereo					
	little	24	4 bytes	<sample rate>	--
Samples per second: e.g., 22050					
	little	28	4 bytes	<bytes/second>	--
sample rate * block align					
	little	32	2 bytes	<block align>	--
channels * bits/sample / 8					
	little	34	2 bytes	<bits/sample>	-- 8
or 16					

Finally, the data chunk contains the sample data:

	big	36	4 bytes	'data'	--
0x64617461					
	little	40	4 bytes	<length of the data block>	
	little	44	*	<sample data>	

 */


```

uint32_t Temp = 0x00;
uint32_t ExtraFormatBytes = 0;
uint32_t HeadTabIdx = 0;

/* Read chunkID, must be 'RIFF'
-----*/ // uint32_t
ChunkID; /* 0 */
Temp = ReadUnit(4, BIG_ENDIAN, pHeader, HeadTabIdx);
if (Temp != CHUNK_ID)
{
return(INVALID_RIFF_ID);
}
HeadTabIdx += 4;

/* Read the file length
-----*/
//uint32_t FileSize; /* 4 */
pAudioFormatDataStruct->RIFFchunksize = ReadUnit(4,
LITTLE_ENDIAN, pHeader, HeadTabIdx);
HeadTabIdx += 4;

/* Read the file format, must be 'WAVE'
-----*/ //uint32_t
FileFormat; /* 8 */
Temp = ReadUnit(4, BIG_ENDIAN, pHeader, HeadTabIdx);
if (Temp != FILE_FORMAT)
{
return(INVALID_WAVE_FORMAT);
}
HeadTabIdx += 4;

/* Read the format chunk, must be 'fmt '
-----*/ // uint32_t
SubChunk1ID; /* 12 */
Temp = ReadUnit(4, BIG_ENDIAN, pHeader, HeadTabIdx);
if (Temp != FORMAT_ID)
{
return(INVALID_FORMAT_CHUNK_ID);
}

```

```

    }
    HeadTabIdx += 4;

    /* Read the length of the 'fmt' data, must be 0x10
    -----*/ //uint32_t SubChunk1Size; /*
16*/   ???
    Temp = ReadUnit(4, LITTLE_ENDIAN, pHeader,
HeadTabIdx);
    if (Temp != 0x10)
    {
        ExtraFormatBytes = 1;
    }
    HeadTabIdx += 4;

    /* Read the audio format, must be 0x01 (PCM)
    -----*/ //uint16_t AudioFormat;
/* 20 */
    pAudioFormatDataStruct->FormatTag = ReadUnit(2,
LITTLE_ENDIAN, pHeader, HeadTabIdx);
    if (pAudioFormatDataStruct->FormatTag !=
WAVE_FORMAT_PCM)
    {
        return(UNSUPPORTED_FORMAT_TAG);
    }
    HeadTabIdx += 2;

    /* Read the number of channels: 0x02->Stereo
0x01->Mono -----*/ //uint16_t NbrChannels;
/* 22 */
    pAudioFormatDataStruct->NumChannels = ReadUnit(2,
LITTLE_ENDIAN, pHeader, HeadTabIdx);
    HeadTabIdx += 2;

    // OmniDrum implementation standard is mono channel
    if (pAudioFormatDataStruct->NumChannels != 0x01)
    {
        return(UNSUPPORTED_NUM_CHANNELS);
    }

```

```

        /* Read the Sample Rate
        -----*/
        //uint32_t SampleRate;      /* 24 */
        pAudioFormatDataStruct->SampleRate = ReadUnit(4,
        LITTLE_ENDIAN, pHeader, HeadTabIdx);
        HeadTabIdx += 4;

        // OmniDrum implementation standard is 44.1 kHz sample
rate
        if (pAudioFormatDataStruct->SampleRate != SAMPLE_RATE)
        {
            return(UNSUPPORTED_SAMPLE_RATE);
        }

        /* Read the Byte Rate
        -----*/ //
        uint32_t ByteRate;      /* 28 */
        pAudioFormatDataStruct->ByteRate = ReadUnit(4,
        LITTLE_ENDIAN, pHeader, HeadTabIdx);
        HeadTabIdx += 4;

        /* Read the block alignment
        -----*/
        //uint16_t BlockAlign;      /* 32 */
        pAudioFormatDataStruct->BlockAlign = ReadUnit(2,
        LITTLE_ENDIAN, pHeader, HeadTabIdx);
        HeadTabIdx += 2;

        /* Read the number of bits per sample
        -----*/ // uint16_t
        BitPerSample; /* 34 */
        pAudioFormatDataStruct->BitsPerSample = ReadUnit(2,
        LITTLE_ENDIAN, pHeader, HeadTabIdx);
        HeadTabIdx += 2;

        if (pAudioFormatDataStruct->BitsPerSample !=
        BITS_PER_SAMPLE_16)

```

```

    {
#ifdef SUPORT_8BIT_SAMPLES
        return(UNSUPPORTED_BITS_PER_SAMPLE);
#endif /* SUPORT_8BIT_SAMPLES */
    }

    /* If there are Extra format bytes, these bytes will
be defined in "Fact Chunk" */
    if (ExtraFormatBytes == 1)
    {
        /* Read the Extra format bytes here */
    }

    /* Read the Data chunk, must be 'data'
-----*/ //uint32_t
SubChunk2ID; /* 36 */
    while (ReadUnit(4, BIG_ENDIAN, pHeader, HeadTabIdx) !=
DATA_ID)
    {
        if (HeadTabIdx >= WAV_HEADER_MAX_SIZE)
        {
            return (INVALID_DATA_CHUNK_ID);
        }
        else
        {
            HeadTabIdx++;
        }
    }

    HeadTabIdx += 4;

    /* Read the number of sample data
-----*/ //uint32_t
SubChunk2Size; /* 40 */
    pAudioFormatDataStruct->AudioLength = ReadUnit(4,
LITTLE_ENDIAN, pHeader, HeadTabIdx);
    HeadTabIdx += 4;

```

```

        /* Set the data pointer at the beginning of the
effective audio data */
        pAudioFormatDataStruct->AudioStartAddr = HeadTabIdx;

        return(VALID_WAVE_FILE);
    }

void all_LEDs_on(void)
{
    // turn on all LEDs
    HAL_GPIO_WritePin(LED_1_GPIO_Port, LED_1_Pin,
GPIO_PIN_SET);
    HAL_GPIO_WritePin(LED_2_GPIO_Port, LED_2_Pin,
GPIO_PIN_SET);
    HAL_GPIO_WritePin(LED_3_GPIO_Port, LED_3_Pin,
GPIO_PIN_SET);
    HAL_GPIO_WritePin(LED_4_GPIO_Port, LED_4_Pin,
GPIO_PIN_SET);
}

void all_LEDs_off(void)
{
    // turn off all LEDs
    HAL_GPIO_WritePin(LED_1_GPIO_Port, LED_1_Pin,
GPIO_PIN_RESET);
    HAL_GPIO_WritePin(LED_2_GPIO_Port, LED_2_Pin,
GPIO_PIN_RESET);
    HAL_GPIO_WritePin(LED_3_GPIO_Port, LED_3_Pin,
GPIO_PIN_RESET);
    HAL_GPIO_WritePin(LED_4_GPIO_Port, LED_4_Pin,
GPIO_PIN_RESET);
}

// change LEDs between states by directly using the active
profile value. useful for profile change.
void set_LEDs(void)
{
    //    stop_blinking(); // redundant set

```

```

    all_LEDs_off();

    //    if (profiles == NULL) {
    //        num_profiles = 0; // redundant set just to
be safe
    //        active_profile = 0; // redundant set just
to be safe
    //    } else {
    switch (active_profile) {
    case 0:
    // only LED_1 on
    HAL_GPIO_WritePin(LED_1_GPIO_Port, LED_1_Pin,
GPIO_PIN_SET);
    break;

    case 1:
    // only LED_2 on
    HAL_GPIO_WritePin(LED_2_GPIO_Port, LED_2_Pin,
GPIO_PIN_SET);
    break;

    case 2:
    // only LED_3 on
    HAL_GPIO_WritePin(LED_3_GPIO_Port, LED_3_Pin,
GPIO_PIN_SET);
    break;

    case 3:
    // only LED_4 on
    HAL_GPIO_WritePin(LED_4_GPIO_Port, LED_4_Pin,
GPIO_PIN_SET);
    break;

    default:
    break;
    }
}

```

```

// switch to next valid profile in the profiles array.
useful for profile button pressed or after profiles setup.
void set_active_profile(uint8_t profile_index)
{
    if (num_profiles > 0) { // catch div by 0
        active_profile = profile_index % num_profiles;
        //          reset_all_open_file_handles();
        reset_all_midi_notes();
        set_LEDs();
    } else {
        active_profile = 0;
        all_LEDs_off(); // indicate not ready idle state (OFF)
    }
}

/**
 * @brief Process all WAV files in an instrument directory.
 * If the current profile exceeds the available midi codes
 * it reverts the current processed information and the
 * midi_wav_t array then returns an empty instrument_t for
 * process_profile_dir().
 */
void process_instrument_dir(char* instrument_path,
instrument_t* instrument, profile_t* profile)
{
    DIR dir;
    FIL fil;
    FILINFO fno;
    FRESULT res;    // result of FatFs operations
    UINT bytesRead; // number of bytes read
    uint8_t buffer[WAV_HEADER_MAX_SIZE]; // buffer to
hold the WAV file header (usually 44 bytes)
    WAVE_FormatTypeDef DecWavFmtStruct;

    // open the instrument directory
    if (f_opendir(&dir, instrument_path) != FR_OK) {
        return; // skip this instrument
    }

```

```

        // traverse all files in the instrument directory
        while (f_readdir(&dir, &fno) == FR_OK && fno.fname[0]
!= 0) {
            if (!(fno.fattrib & AM_DIR)) { // if it's a file

                // early fail condition. return to let the
profile be discarded.
                if (profile->midi_code_used[127]) { // no more
midi codes available
                    // FREE MEMORY BEFORE RETURN
                    for (uint8_t i = 0; i <
instrument->num_waves; i++) {
                        free(instrument->wav_files[i].filename);
                    }
                    if (instrument->wav_files) {
                        free(instrument->wav_files);
                    }

                    instrument->num_waves = 0; // set 0 to
indicate invalid instrument will not be added
                    f_closedir(&dir);
                    return;
                }

                char full_path[256];
                snprintf(full_path, strlen(full_path), "%s/%s",
instrument_path, fno.fname);

                // open the file in read-only mode
                res = f_open(&fil, full_path, FA_READ);
                if (res != FR_OK) {
                    printf("Error opening file: %d\r\n", res);
                    f_closedir(&dir);
                    return;
                }

                // read the first WAV_HEADER_MAX_SIZE bytes (WAV
header) into the buffer

```



```

        res = f_read(&fil, buffer, sizeof(buffer),
&bytesRead);
        if (res != FR_OK || bytesRead != sizeof(buffer))
        {
            printf("Error reading WAV header: %d\r\n",
res);
            f_close(&fil);
            f_closedir(&dir);
            return;
        }

        // close the file after reading
        f_close(&fil);

        // check if the file is a valid WAV
        if (WavProcess_ParseHeader(buffer,
&DecWavFmtStruct) == VALID_WAVE_FILE) {

            // allocate memory for new MIDI-WAV pair
            midi_wav_t* temp_wav_files =
realloc(instrument->wav_files,
            sizeof(midi_wav_t) *
(instrument->num_waves + 1));
            if (!temp_wav_files) { // handle realloc
failure
                // FREE MEMORY BEFORE RETURN
                for (uint8_t i = 0; i <
instrument->num_waves; i++) {

free(instrument->wav_files[i].filename);
                }
                if (instrument->wav_files) {
                    free(instrument->wav_files);
                }
                instrument->num_waves = 0; // set 0 to
indicate invalid instrument will not be added
                f_closedir(&dir);
                return;
            }
        }

```

```

    }
    instrument->wav_files = temp_wav_files;

    // store the filename and assign a MIDI
note
    // TODO: in the event a file name fails, it
    throws the entire instrument away. make only the file
    invalid.

    instrument->wav_files[instrument->num_waves].filename =
    strdup(fno.fname);
    if
    (!instrument->wav_files[instrument->num_waves].filename) {
    // handle strdup failure
        // FREE MEMORY BEFORE RETURN
        for (uint8_t i = 0; i <
instrument->num_waves; i++) {

    free(instrument->wav_files[i].filename);
        }
        if (instrument->wav_files) {
            free(instrument->wav_files);
        }
        instrument->num_waves = 0; // set 0 to
    indicate invalid instrument will not be added
        f_closedir(&dir);
        return;
    }

    // get the duration (ms) of audio data from
    the header parsing results
    uint32_t duration = 1000 *
DecWavFmtStruct.AudioLength /
        (DecWavFmtStruct.SampleRate *
DecWavFmtStruct.NumChannels * (DecWavFmtStruct.BitsPerSample
/ 8));

```

```

instrument->wav_files[instrument->num_waves].duration =
duration;

        // copy the starting address of audio data
from the header parsing results

instrument->wav_files[instrument->num_waves].start_addr =
DecWavFmtStruct.AudioStartAddr;

        // assign a unique MIDI code across the
profile
        // maximum number of WAV files in a profile
cannot exceed this condition
        for (uint8_t i = 24; i < 128; i++) { //
about 100 WAVs allowed in a profile
            if (!profile->midi_code_used[i]) {

instrument->wav_files[instrument->num_waves].midi_code = i;
                profile->midi_code_used[i] = true; //
mark the MIDI code as used in the profile
                break;
            }
        }
        instrument->num_waves++;
    }
}

    f_closedir(&dir);
}

/**
 * @brief Process all instruments in a profile
 */
void process_profile_dir(char* profile_path, char*
profile_name)
{

```

```

    DIR dir;
    FILINFO fno;

    profile_t* current_profile =
malloc(sizeof(profile_t));
    if (!current_profile) {
        printf("error: process_profile_dir() Memory allocation
failed"
               " on current_profile.\r\n");
        return;
    }

    current_profile->name = strdup(profile_name);
    if (!current_profile->name) {
        printf("error: process_profile_dir() Memory allocation
failed"
               " strdup on current_profile->name.\r\n");
        free(current_profile);
        return;
    }
    current_profile->instruments = NULL;
    current_profile->num_instruments = 0;
    // initialize MIDI code usage tracking for the profile
    memset(current_profile->midi_code_used, false,
sizeof(current_profile->midi_code_used));

    // open the profile directory
    if (f_opendir(&dir, profile_path) != FR_OK) {
        free(current_profile->name);
        free(current_profile);
        return;
    }

    // traverse all directories (instruments) in the
profile
    while (f_readdir(&dir, &fno) == FR_OK && fno.fname[0]
!= 0) {
        if (fno.fattrib & AM_DIR) { // if it's a directory

```

```

        char instrument_path[256];
        snprintf(instrument_path,
sizeof(instrument_path), "%s/%s", profile_path, fno.fname);

        // create new instrument structure
        instrument_t* new_instrument =
malloc(sizeof(instrument_t));
        if (!new_instrument) {
            printf("error: process_profile_dir() Memory
allocation failed"
                " on new_instrument.\r\n");
            break;
        }
        new_instrument->name = strdup(fno.fname);
        if (!new_instrument->name) { // handle strdup
failure
            printf("error: process_profile_dir() Memory
allocation failed"
                " strdup on
new_instrument->name.\r\n");
            free(new_instrument);
            break;
        }
        new_instrument->wav_files = NULL;
        new_instrument->num_waves = 0;

        process_instrument_dir(instrument_path,
new_instrument, current_profile);

        if (new_instrument->num_waves > 0) {
            // add valid instrument to the profile
            instrument_t* temp_instruments =
realloc(current_profile->instruments,
                sizeof(instrument_t) *
(current_profile->num_instruments + 1));

            if (!temp_instruments) { // handle realloc
failure

```

```

        printf("error: process_profile_dir() Memory
allocation failed"

               " realloc on
temp_instruments.\r\n");
        free(new_instrument->name);
        free(new_instrument);
        break;
    }
    current_profile->instruments =
temp_instruments;

current_profile->instruments[current_profile->num_instrument
s] = *new_instrument;
    current_profile->num_instruments++;
    } else {
        // free invalid instrument assuming no
files or failure already freed
        free(new_instrument->name);
        free(new_instrument);
    }
    // if the profile has 8 instruments, finish
creating it
    if (current_profile->num_instruments == 8) {
        break;
    }
}
}

f_closedir(&dir);

// only add profile if it has 8 valid instruments
if (current_profile->num_instruments == 8) { //
assuming exactly 8 instruments per profile
    profile_t* temp_profiles = realloc(profiles,
sizeof(profile_t) * (num_profiles + 1));
    if (!temp_profiles) { // handle realloc failure
        printf("error: process_profile_dir() Memory
allocation failed"

```

```

        " realloc on profiles.\r\n");
        free(current_profile->name);
        free(current_profile);
        return;
    }
    profiles = temp_profiles;
    profiles[num_profiles] = *current_profile;
    num_profiles++;
} else {
    // free profile data if not all instruments are valid
    for (uint8_t i = 0; i <
current_profile->num_instruments; i++) {
        free(current_profile->instruments[i].name);
        if (current_profile->instruments[i].wav_files) {
            for (uint8_t j = 0; j <
current_profile->instruments[i].num_waves; j++) {

                free(current_profile->instruments[i].wav_files[j].filename);
            }

            free(current_profile->instruments[i].wav_files);
        }
    }
    free(current_profile->instruments);
    free(current_profile->name);
    free(current_profile);
}

// begin initialization of profiles data structure. useful
for each mount of the SD card.
void create_profiles()
{
    DIR dir;
    FILINFO fno;

    // open the root directory containing profiles folder
    if (f_opendir(&dir, PROFILES_FOLDER) != FR_OK) {

```

```

        printf("warning: create_profiles() '%s' folder not
found."
               " No resources in root filesystem.\r\n"
               "attention: Read the standards for the
directory"
               " structure and try again.\r\n",
PROFILES_FOLDER);
    return;
}

    // for each sub directory start adding to the profiles
data structure
    while (f_readdir(&dir, &fno) == FR_OK && fno.fname[0]
!= 0) {
        if (fno.fattrib & AM_DIR) { // if it's a directory
            char profile_path[256];
            snprintf(profile_path, sizeof(profile_path),
"%s/%s", PROFILES_FOLDER, fno.fname);
            process_profile_dir(profile_path, fno.fname);
            // stop processing profiles if we've reached the
maximum allowed
            if (num_profiles >= PROFILES_MAX) {
                num_profiles = PROFILES_MAX; // redundant
set just to be safe
                break;
            }
        }
    }

    f_closedir(&dir); // close the profiles folder

    // print content of profiles data structure for
debugging
    for (uint8_t i = 0; i < num_profiles; i++) {
        profile_t* p = &profiles[i];
        // profile folder name and index number
        printf("Profile: %s, Index: %u\r\n", p->name, i);
        for (uint8_t j = 0; j < p->num_instruments; j++) {

```



```

        instrument_t* inst = &p->instruments[j];
        // instrument folder name and index number
        printf("\tInstrument: %s, Index: %u\r\n",
inst->name, j);
        for (uint8_t k = 0; k < inst->num_waves; k++) {
            // WAV file name, MIDI code, and index
number
            printf("\t\tWAV File: %s, MIDI Note: %u,
Index: %u\r\n",
                inst->wav_files[k].filename,
                inst->wav_files[k].midi_code,
                k );
        }
    }
}

```

```

// begin initialization of profiles data structure.
void create_profiles_midi_only()
{
    uint8_t temp_midi[8] = { 0x24, 0x2e, 0x2a ,0x28, 0x2d,
0x30, 0x31, 0x33 };
    uint8_t temp_num_profiles = 0;

    for (uint8_t i = 0; i < PROFILES_MAX; i++) { // create
3 profiles
        profile_t* current_profile =
malloc(sizeof(profile_t));
        if (!current_profile) {
            printf("error: process_profile_dir() Memory
allocation failed"
                " on current_profile.\r\n");
            continue;
        }
        current_profile->instruments = NULL;
        current_profile->num_instruments = 0;
    }
}

```

```

        for (uint8_t j = 0; j < 8; j++) { // create exactly 8
instruments
            instrument_t* new_instrument =
malloc(sizeof(instrument_t));
            if (!new_instrument) {
                printf("error: process_profile_dir() Memory
allocation failed"
                    " on new_instrument.\r\n");
                break;
            }
            new_instrument->wav_files = NULL;
            new_instrument->num_waves = 0;

            for (uint8_t k = 0; k < 5; k++) { // assign 5
codes for every instrument
                // allocate memory for new MIDI-WAV pair
                midi_wav_t* temp_wav_files =
realloc(new_instrument->wav_files,
                    sizeof(midi_wav_t) *
(new_instrument->num_waves + 1));
                if (!temp_wav_files) { // handle realloc
failure
                    // FREE MEMORY BEFORE RETURN
                    if (new_instrument->wav_files) {
                        free(new_instrument->wav_files);
                    }
                    new_instrument->num_waves = 0; // set 0 to
indicate invalid instrument will not be added
                    break;
                }
                new_instrument->wav_files = temp_wav_files;

new_instrument->wav_files[new_instrument->num_waves].midi_ch
annel = i;

```

```

new_instrument->wav_files[new_instrument->num_waves].midi_co
de = temp_midi[j];

        // quick generation of some short duration
        //          uint16_t duration =
(uint16_t)((rand() % 50) + 50 );
        //
new_instrument->wav_files[new_instrument->num_waves].duratio
n = duration;
        // fixed duration ( 100 ms )

new_instrument->wav_files[new_instrument->num_waves].duratio
n = 100;

        new_instrument->num_waves++;
    }

    if (new_instrument->num_waves > 0) {
        // add valid instrument to the profile
        instrument_t* temp_instruments =
realloc(current_profile->instruments,
        sizeof(instrument_t) *
(current_profile->num_instruments + 1));

        if (!temp_instruments) { // handle realloc
failure
            printf("error: process_profile_dir() Memory
allocation failed"
                " realloc on
temp_instruments.\r\n");
            free(new_instrument);
            break;
        }
        current_profile->instruments =
temp_instruments;

```

```

current_profile->instruments[current_profile->num_instrument
s] = *new_instrument;
        current_profile->num_instruments++;
    } else {
        // free invalid instrument assuming no
files or failure already freed
        free(new_instrument);
    }
}

    if (current_profile->num_instruments == 8) { //
assuming exactly 8 instruments per profile
        profile_t* temp_profiles = realloc(profiles,
sizeof(profile_t) * (temp_num_profiles + 1));
        if (!temp_profiles) { // handle realloc failure
            printf("error: process_profile_dir() Memory
allocation failed"
                " realloc on profiles.\r\n");
            free(current_profile);
            return;
        }
        profiles = temp_profiles;
        profiles[temp_num_profiles] = *current_profile;
        temp_num_profiles++;
    }
}

// show the contents of profiles
for (int i = 0; i < temp_num_profiles; i++) {
    printf("\r\n\t\tPROFILE: %d", i);
    for (int j = 0; j < profiles[i].num_instruments; j++)
    {
        printf("\r\nINSTRUMENT: %d", j);
        for (int k = 0; k <
profiles[i].instruments[j].num_waves; k++) {
            printf("\r\n\tCHANNEL: %u \tMIDI: 0x%X
\tDURATION: %u",

```

```

profiles[i].instruments[j].wav_files[k].midi_channel + 1,

profiles[i].instruments[j].wav_files[k].midi_code,

profiles[i].instruments[j].wav_files[k].duration
        );
    }
}

    // set global variable to show profiles created.
    num_profiles = temp_num_profiles;
}

// reset the profiles data structure. useful between SD card
mounts to properly track file contents.
void free_profiles()
{
    if (profiles != NULL) {
        // free each profile
        for (uint8_t i = 0; i < num_profiles; i++) {
            profile_t* profile = &profiles[i];

            // free each instrument in the profile
            for (uint8_t j = 0; j < profile->num_instruments;
j++) {
                instrument_t* instrument =
&profile->instruments[j];

                // free each WAV file in the instrument
                for (uint8_t k = 0; k <
instrument->num_waves; k++) {
                    free(instrument->wav_files[k].filename); //
free the filename of each WAV file
                }
            }
        }
    }
}

```

```

        free(instrument->wav_files); // free the
WAV file array
        free(instrument->name); // free the
instrument name
    }

    free(profile->instruments); // free the
instruments array
    free(profile->name); // free the profile name
}

// free the profiles array and set it to NULL
free(profiles);
}
profiles = NULL;
num_profiles = 0; // reset the profiles count
}

// determine hash from full_path string for faster compare
uint32_t murmurhash3_32(char *key, uint32_t len, uint32_t
seed)
{
    uint32_t h = seed;
    for (uint32_t i = 0; i < len; i++) {
        h ^= (key[i] * 0x5bd1e995);
        h *= 0x5bd1e995;
    }
    return h;
}

// find an open file by its full path
OpenFile* find_open_file(uint32_t hash)
{
    OpenFile* current = open_file_list;
    while (current != NULL) {
        if (current->hash == hash) {
            // file is already open
            return current;
        }
    }
}

```

```

    }
    current = current->next;
}
// file is not open
return NULL;
}

// add a specific file if it doesn't already have an open
file handle. used before adding a file to mix.
// NOTE: uses hash values instead of string comparison
FIL* add_open_file_handle(char* full_path)
{
    FRESULT fres;
    uint32_t hash_seed = 0x7fffffff; // large prime number
    uint32_t hash_res = 0;

    hash_res = murmurhash3_32(full_path,
strlen(full_path), hash_seed);

    // early return. check if the file is already open
    OpenFile* existing_file = find_open_file(hash_res);
    if (existing_file != NULL) {
        // file is already open, return its file pointer
        return existing_file->file_ptr;
    }

    // file is not open, so we open it and add to the list
    FIL* fil = malloc(sizeof(FIL)); // allocate memory for
the file handle
    if (!fil) {
        // failed to allocate memory for FIL
        printf("add_open_file_handle() error: Memory
allocation failed"
            " for file handle.\r\n");
        return NULL;
    }

    fres = f_open(fil, full_path, FA_READ);

```

```

        if (fres != FR_OK) {
            // cleanup if open fails
            printf("add_open_file_handle() error: f_open error
(%i) for file: %s\r\n", fres, full_path);
            free(fil);
            return NULL;
        }

        // add the new file to the tracking list
        OpenFile* new_entry = malloc(sizeof(OpenFile));
        if (!new_entry) {
            // cleanup if memory allocation fails
            f_close(fil);
            free(fil);
            return NULL;
        }

        // initialize the new entry
        strncpy(new_entry->full_path, full_path,
sizeof(new_entry->full_path) - 1);
        new_entry->file_ptr = fil;
        new_entry->hash = hash_res;
        new_entry->next = open_file_list; // insert at the
beginning of the list
        open_file_list = new_entry;

        // return the file pointer for the newly opened file
        return fil;
    }

    // close files currently used in wavFiles mix to free FATFS
handles.
    // useful between SD card mounts and changed active profile.
    // NOTE: use reset_all_wavFiles() before! all entries in
wavFiles array must have
    // isPlaying set false before calling this function.
    void reset_all_open_file_handles(void)
    {

```



```

    OpenFile* current = open_file_list;
    OpenFile* next_file;

    while (current != NULL) {
        // close the file
        f_close(current->file_ptr);
        next_file = current->next;

        // free the memory
        free(current->file_ptr);
        free(current);

        current = next_file;
    }

    // clear the list after all files are closed
    open_file_list = NULL;
}

// need full_path string for f_open(). used before opening
file then sending to mix.
// NOTE: only works on the active profile index!
char* get_full_path(uint8_t instr_index, uint8_t wav_index)
{
    static char full_path[256]; // static buffer for full
    path

    snprintf(full_path, 256, "%s/%s/%s/%s",
              PROFILES_FOLDER,
              profiles[active_profile].name,

    profiles[active_profile].instruments[instr_index].name,

    profiles[active_profile].instruments[instr_index].wav_files[
    wav_index].filename);

    return full_path;
}

```

```

// translate a sensor strike value to the corresponding
index of a file in the profiles array.
// open file handles help with using f_open and f_close once
per file.
// pass the file pointer to the software mix task and midi
code to the midi output task.
// NOTE: only works on the active profile index!
void translate_strike_to_profile(sensor_data_t strike)
{
    // no bounds checking on profiles array because if
    something went wrong it's YOUR problem :p

    uint8_t instr_index;                // the
appropriate sensor to instrument mapping
    uint8_t wav_index;                  // the
approximated index that corresponds to the sensor value
    uint8_t number_of_files;            // for dividing
sensor range so we can get range of each index
    uint16_t index_range;               // range between
file indexes in reference to ADC sensor values
    uint16_t temp_range;                // crawl in
increments of index_range and compare with sensor data
    midi_wav_t* translated_strike = NULL;

    instr_index = strike.sensor_number;

    // NOTE: hi-hat open (not pressed) instr_index = 1,
hi-hat closed (pressed) instr_index = 2
    /*
    *   instruments
    *   index          sensor      instrument
note
    *
    -----
    *   0              -          kick
pedal_1 doesn't need translation

```

```

        *      1              1          hi-hat open
pedal_2 is checked during translation
        *      2              1          hi-hat closed
pedal_2 is checked during translation
        *      3              2          -
        *      4              3          -
        *      5              4          -
        *      6              5          -
        *      7              6          -
        *
        * */

    if (instr_index == 0) {
        // kick pedal should not be sending data here,
        translate directly
        translated_strike =
&profiles[active_profile].instruments[0].wav_files[0];

        //    if (add_wav_file_to_mix(wav_file,
translated_strike->start_addr)) {
            //          start_midi_note(translated_strike, 127);
            //    }

        start_midi_note(translated_strike, 127);

        return;
    }

    if (instr_index == 1) {
        // hi-hat sensor strike
        GPIO_PinState hi_hat_pedal =
HAL_GPIO_ReadPin(Pedal_2_GPIO_Port, Pedal_2_Pin);
        if (hi_hat_pedal == GPIO_PIN_RESET) {
            // pedal pressed, closed hi-hat is instr_index =
2

            // expects the profile to have the 8 instruments
when built
            instr_index = 2;

```

```

    }
    } else {
        instr_index++;
    }

    number_of_files =
profiles[active_profile].instruments[instr_index].num_waves;

    if (number_of_files > 0) { // div by 0 check
        index_range = (uint16_t)(sensor_range /
number_of_files);
    } else {
        // div by 0 error
        // should not be possible to reach?
        printf("error: translate_strike_to_profile() div by
0,"
                " profiles initialization failed?\r\n");
        // while(true);
    }

    temp_range = (uint16_t)(SENSOR_CUTOFF + index_range);
// end boundary of first index
    while (wav_index < number_of_files) {

        if (strike.peak_value <= temp_range) {
            // current index value is the correct location
            break;
        }
        wav_index++;
        temp_range += index_range;
    }

    translated_strike =
&profiles[active_profile].instruments[instr_index].wav_files
[wav_index];

    if (!translated_strike) { // NULL pointer error

```

```

        printf("translate_strike_to_profile() error:
translated_strike NULL.\r\n");
        return;
    }

    // char* full_path = get_full_path(instr_index,
wav_index);

    // create open file handle if it doesn't exist
    // FILE* wav_file = add_open_file_handle(full_path);

    // if (!wav_file) { // NULL file pointer error
    //     // add_open_file_handle() can return NULL
    //     printf("translate_strike_to_profile()
error: wav_file NULL.\r\n");
    //     return;
    // }

    // get the velocity of the strike (12-bit ADC >> 5 ->
7-bit velocity)
    uint8_t velocity = (uint8_t)(strike.peak_value >> 5);

    // if (add_wav_file_to_mix(wav_file,
translated_strike->start_addr)) {
    //
start_midi_note(translated_strike->midi_code, velocity,
translated_strike->duration);
    // }

    start_midi_note(translated_strike, velocity);

}

// mount FATFS and test SPI communication with read write
operation.

```

```

// NOTE: this is for TESTING purposes. mount is done with
the periodic function sd_card_check_presence()
// and is called by the SD card monitor task.
void sd_card_mount(void)
{
    //    myprintf("\r\n~ SD card demo by kiwih
~\r\n\r\n");

    HAL_Delay(1000); // short delay is important to let
the SD card settle

    // some variables for FatFs
    // FATFS FatFs; //Fatfs handle
    // FatFs MAKE GLOBAL VARIABLE
    FIL fil;        //File handle
    FRESULT fres; //Result after operations

    // open the file system
    fres = f_mount(&FatFs, "", 1); //1=mount now
    if (fres != FR_OK) {
        printf("f_mount error (%i)\r\n", fres);
        while (true) {

        }
    }

    /* DEBUG TESTING
-----*/
    // Let's get some statistics from the SD card
    DWORD free_clusters, free_sectors, total_sectors;

    FATFS* getFreeFs;

    fres = f_getfree("", &free_clusters, &getFreeFs);
    if (fres != FR_OK) {
        printf("f_getfree error (%i)\r\n", fres);
        while (true) {

```

```

    }
}

// Formula comes from ChaN's documentation
total_sectors = (getFreeFs->n_fatent - 2) *
getFreeFs->csize;
free_sectors = free_clusters * getFreeFs->csize;

printf("SD card stats:\r\n%10lu KiB total drive
space.\r\n", total_sectors / 2);
printf("%10lu KiB available.\r\n", free_sectors / 2);

// Now let's try to open file "test.txt"
fres = f_open(&fil, "test.txt", FA_READ);
if (fres != FR_OK) {
printf("f_open error (%i)\r\n", fres);
while (true) {

}
}
printf("I was able to open 'test.txt' for
reading!\r\n");

// Read 30 bytes from "test.txt" on the SD card
BYTE readBuf[30];

// We can either use f_read OR f_gets to get data out
of files
// f_gets is a wrapper on f_read that does some string
formatting for us
TCHAR* rres = f_gets((TCHAR*)readBuf, 30, &fil);
if(rres != 0) {
printf("Read string from 'test.txt' contents: %s\r\n",
readBuf);
} else {
printf("f_gets error (%i)\r\n", fres);
}

```

```

        // Be a tidy kiwi - don't forget to close your file!
        fclose(&fil);

        // Now let's try and write a file "write.txt"
        fres = fopen(&fil, "write.txt", FA_WRITE |
FA_OPEN_ALWAYS | FA_CREATE_ALWAYS);
        if(fres == FR_OK) {
            printf("I was able to open 'write.txt' for
writing\r\n");
        } else {
            printf("f_open error (%i)\r\n", fres);
        }

        // Copy in a string
        strncpy((char*)readBuf, "a new file is made!", 20);
        UINT bytesWrote;
        fres = f_write(&fil, readBuf, 19, &bytesWrote);
        if(fres == FR_OK) {
            printf("Wrote %i bytes to 'write.txt'\r\n",
bytesWrote);
        } else {
            printf("f_write error (%i)\r\n", fres);
        }

        // Be a tidy kiwi - don't forget to close your file!
        fclose(&fil);

        // We're done, so de-mount the drive
        //   f_mount(NULL, "", 0);
    }

    // force unmount of the FATFS filesystem. useful when SD
    card is removed.
    // NOTE: DON'T FORGET OPEN FILE HANDLES ARE DEALT WITH
    SEPARATELY!
    void sd_card_umount(void)
    {
        FRESULT fres; // result after operations

```



```

        fres = f_mount(NULL, "", 0);
        if (fres != FR_OK) {
            printf("SD_card_umount() Failed to unmount
filesystem\r\n");
        }
    }

// periodically call this function to observe the state of
our filesystem's response.
// used in the SD card monitor task.
// NOTE: this is the actual mount function of our completed
system.
bool sd_card_check_presence(void)
{
    FRESULT fres; // result after operations
    fres = f_mount(&FatFs, "", 1);
    if (fres == FR_OK) {
        // device is still present and the filesystem is
mounted
        return true;
    } else {
        // device is not found or has been removed
        if (fres == FR_DISK_ERR) {
            printf("SD_card_check_presence() Disk error
occurred"
                " (possibly device removed).\r\n");
        } else if (fres == FR_NO_FILESYSTEM) {
            printf("SD_card_check_presence() No filesystem
found"
                " on the device.\r\n");
        } else if (fres == FR_NOT_READY) {
            printf("SD_card_check_presence() Device is not
ready"
                " (possibly removed or
unresponsive).\r\n");
        } else {
            printf("An unknown error occurred:"

```

```

        " SD_card_check_presence() (%i)\r\n",
fres);
    }
    return false;
}
}

/**
 * @brief Function implementing the defaultTask thread.
 * This determines the state of the headphones and
amplifier.
 * @param argument: Not used
 * @retval None
 */
void headphone_detect(void)
{
    /*
     * for the speaker to shut off when the headphones are
plugged in:
     * Pin 42 (PA8) needs to be set as digital Input with
Pull-Up
     * Pin 41 (PC9) needs to be set as digital Output with
Push-Pull
     * when Pin 42 is LOW, Pin 41 should output HIGH
     * when Pin 42 is HIGH, Pin 41 should output LOW
     */

    GPIO_PinState headphoneStateRead =
HAL_GPIO_ReadPin(Headphone_GPIO_Port, Headphone_Pin);

    if (headphoneStateRead == GPIO_PIN_RESET) {
        // headphones not detected, enable amplifier
        HAL_GPIO_WritePin(Amplifier_GPIO_Port, Amplifier_Pin,
GPIO_PIN_SET);
    } else {
        // headphones detected, disable amplifier

```

```

        HAL_GPIO_WritePin(Amplifier_GPIO_Port, Amplifier_Pin,
GPIO_PIN_RESET);
    }
}

/**
 * @brief Function implementing the outputToDACTask thread.
 * @param argument: Not used
 * @retval None
 */
//void output_to_DAC(void)
//{
//
//    /* Infinite loop */
//    while(true)
//    {
//        if (sd_card_presence && num_profiles > 0) { //
system ready
//
//        // Wait until the audio_mix_task() has filled the
next buffer
//        osSemaphoreAcquire(audio_semaphoreHandle,
osWaitForever);
//
//        // Determine which buffer is ready to be
processed
//        int16_t *current_buffer = (switch_buffer == 1) ?
audio_buffer_1 : audio_buffer_2;
//
//        // Send each sample to the DAC
//        for(int i = 0; i < BUFFER_SIZE; i++) {
//            while (TIM4->CNT <= TIM4_HALF_VALUE) {
//                // delay for first half sample time at
44.1 kHz with APB1 at 100 MHz
//            }
//            HAL_DAC_SetValue(&hdac1, DAC_CHANNEL_1,
DAC_ALIGN_12B_L, (uint32_t)current_buffer[i]);
//            while (TIM4->CNT >= TIM4_HALF_VALUE) {

```

```

//                                // delay for second half sample time
at 44.1 kHz with APB1 at 100 MHz
//                                }
//                                }
//
//                                // Once the buffer has been fully sent to the
DAC, signal that the next buffer can be filled
//                                switch_buffer = 0;
//
//
//                                } else {
//                                // set a s16le PCM "silent" sample (0) so clock
trigger on DAC produces "silence"
//                                HAL_DAC_SetValue(&hdac1, DAC_CHANNEL_1,
DAC_ALIGN_12B_L, 0);
//                                //                                // set a u16le PCM "silent" sample
(32768) so clock trigger on DAC produces "silence"
//                                //                                HAL_DAC_SetValue(&hdac1,
DAC_CHANNEL_1, DAC_ALIGN_12B_L, 32768);
//                                osDelay(1000); // system not ready, wait 1 second
//                                }
//                                }
//}

/**
 * @brief Function implementing the softwareMixTask thread.
The software mix takes signed 16-bit
 * audio samples (s16le) of multiple files and combines them
to one output sample. The output samples
 * are switched in hardware to u16le before conversion.
 * @note Supported format is 44.1 kHz mono s16le. THE DAC
MUST BE CONFIGURED 'SIGNED FORMAT ENABLED'.
 * @param argument: Not used
 * @retval None
 */
void audio_mix(void)
{

```

```

        // redundant. initialize isPlaying to false before
checking
        reset_all_wavFiles();

        /* Infinite loop */
        while(true)
        {
            if (sd_card_presence && num_profiles > 0) { // system
ready

                // Wait until the output_to_DAC_task() is done
with the current buffer
                while (switch_buffer == 1) {
                    //                osDelay(1); // Delay until
the other buffer is free
                }

                int16_t *current_buffer = (switch_buffer == 0) ?
audio_buffer_1 : audio_buffer_2;

                // initialize to 0 (silence) in terms of signed
int16_t
                memset(current_buffer, 0, BUFFER_SIZE *
sizeof(int16_t));

                // try to take the mutex (wait for the lock)
                if (osMutexAcquire(wavFiles_mutexHandle,
osWaitForever) == osOK) {
                    // mix audio data from all playing files
                    for (int i = 0; i < WAV_FILES_MAX; i++) {
                        if (wavFiles[i].wavFile != NULL &&
wavFiles[i].isPlaying) {
                            // fill the wavFile buffer with data
from the current wav file
                            read_next_chunk(&wavFiles[i]);

                            // mix the data into the output buffer

```

```

        for (int j = 0; j < BUFFER_SIZE; j++)
        {
            // sum the samples, temporarily
            using int32_t to avoid overflow
            int32_t mixed_sample =
            (int32_t)((int16_t)current_buffer[j] +
            wavFiles[i].buffer[j]);

            // if mixed value exceeds the
            16-bit signed range, clamp it
            if (mixed_sample > 32767) {
                mixed_sample = 32767;
            } else if (mixed_sample <
            -32768) {
                mixed_sample = -32768;
            }

            // convert to unsigned 16-bit
            PCM format (u16le)
            //
            uint16_t final_sample = (uint16_t)(mixed_sample + 32768);

            // store the mixed and clamped
            value as unsigned 16-bit in the output buffer
            //
            current_buffer[j] = final_sample;

            // store the mixed and clamped
            value as s16le in the output buffer
            current_buffer[j] =
            (int16_t)mixed_sample;
        }
    }
    osMutexRelease(wavFiles_mutexHandle);

```

```

        // Signal that the buffer is ready by
switching the current_buffer flag
        switch_buffer = 1;

        // Give the semaphore to allow
output_to_DAC_task to process the filled buffer
        osSemaphoreRelease(audio_semaphoreHandle);

    }
} else {
    osDelay(1000); // system not ready, wait 1 second
}
}
}

/**
 * @brief Function implementing the activeNotesTask thread.
MIDI notes are placed into the array
 * from another task and this task checks for expired timers
then sends the MIDI off message.
 * @param argument: Not used
 * @retval None
 */
void expired_notes(void)
{
    uint16_t current_time, start_time;

    current_time = ms_timer(); // get the current tick
count

    for (uint8_t i = 0; i < WAV_FILES_MAX; i++) {
        if (active_notes[i].isPlaying) {
            start_time = active_notes[i].start_time;

            // check if the note duration has passed
            if ((uint16_t)(current_time - start_time) >=
active_notes[i].duration) {

```

```

        // send Note Off for this note

send_midi_note_off(active_notes[i].midi_channel,
active_notes[i].midi_code);
        active_notes[i].isPlaying = false;
    }
}
}

/**
 * @brief Function implementing the playQueueTask thread.
Receive sensor readings and handle
 * their translation to the profiles array before passing to
the output tasks.
 * @param argument: Not used
 * @retval None
 */
void play_queue(void)
{
    // Wait for data to be available in the queue
    sensor_data_t data;
    while (dequeue(&playQueue, &data)) {
        // Successfully retrieved data from the queue
        //          printf("Sensor %d Peak Value:
%d\r\n", data.sensor_number, data.peak_value);
        translate_strike_to_profile(data);
    }
}

/**
 * @brief Function implementing the buttonTask thread. Polls
the GPIO and handles the state.
 * @param argument: Not used
 * @retval None
 */
void profile_button(void)
{

```



```

        // check the button state
        GPIO_PinState buttonStateRead =
HAL_GPIO_ReadPin(Button_GPIO_Port, Button_Pin);

        if (buttonStateRead == GPIO_PIN_RESET) {
            // pressing the button
            if (profile_button_state == BUTTON_IDLE) {
                // button is pressed (falling edge)
                profile_button_state = BUTTON_PRESSED;

                // handle button press event here
                printf("Profile select button pressed\r\n");

                // NOTE: something weird happened without having
another variable
                uint8_t change = (uint8_t)(active_profile + 1);
                if (change >= PROFILES_MAX) {
                    change = 0;
                }

                set_active_profile(change);
            }
            else {
                // not pressing button
                if (profile_button_state == BUTTON_PRESSED) {
                    // button is released (rising edge)
                    profile_button_state = BUTTON_RELEASED;

                    // handle button release event here

                }
                if (profile_button_state == BUTTON_RELEASED) {
                    // button not doing anything
                    profile_button_state = BUTTON_IDLE;
                }
            }
        }
}

```

```

/**
 * @brief Function implementing the sdCardTask thread.
Attempt two consecutive failures
 * before deciding the SD card is unavailable.
 * @note This task is responsible for the initialization
before the system is ready.
 * When the SD card is unavailable the flags are set to
tell other tasks the system
 * is not ready. Handles recovery when SD card is ejected
and brings the system back
 * to an idle state until the SD card is made available.
 * @param argument: Not used
 * @retval None
 */
//void monitor_sd_card_task(void *argument)
//{
//
//    static bool previous = false;
//
//    osDelay(1000); // short delay is important to let the
SD card settle
//    all_LEDs_on(); // initialize LEDs on to show not ready
state
//
//    /* Infinite loop */
//    while(true)
//    {
//        if (sd_card_check_presence()) {
//            if (!previous) {
//                // SD card was inserted, create profiles
structure
//                start_blinking();
//                create_profiles();
//                stop_blinking();
//                // show first profile is active
//                active_profile = 0;
//                set_LEDs();

```

```

//
//          // if no profiles were actually created
//          if (profiles == NULL) {
//              num_profiles = 0; // redundant set
just to be safe
//              all_LEDs_on(); // indicate system is
not ready from mount operation (ON)
//              // when profile select button is
pressed they will turn off
//          }
//
//          sd_card_presence = true; // flag SD card as
mounted
//      }
//      previous = true;
//      }
//      else {
//          if (!previous && sd_card_presence) {
//              // SD card was removed, set to idle state
//              num_profiles = 0; //
redundant set just to be safe
//              active_profile = 0; //
redundant set just to be safe
//              sd_card_presence = false; // flag to
stop using SD card
//              reset_all_midi_notes(); // force
stop all midi notes early;
//              reset_all_wavFiles(); // clear
the software mix items
//              reset_all_open_file_handles(); // close
files
//              sd_card_umount(); // force
unmount filesystem
//              free_profiles(); // allow
profiles structure to be rebuilt
//              all_LEDs_off(); //
indicate not ready idle state (OFF)
//          }

```

```

//          previous = false;
//          }
//
//          // the SD card is unmounted on two iteration
//          fails, maximum 0.5 second delay.
//          osDelay(250); // 0.25 seconds between checks
//      }
//}

/**
 * @brief Function implementing the sensorArrayTask thread.
 * Loops the array of sensors
 * as fast as necessary. Distinguish individual strikes as
 * values spike above SENSOR_CUTOFF.
 * A queue separates this task from the following stages to
 * allow faster polling.
 * @param argument: Not used
 * @retval None
 */
void sensor_array(void)
{
    /**
     *          Sensor      hadc1 rank      hadc1 channel
     * array index
     *
     -----
    -----
     *          1          1          6
     0
     *          2          2          7
     1
     *          3          3          8
     2
     *          4          4          9
     3
     *          5          5          1
     4
    
```

```

*           6           6           2
5
* */

    uint16_t current_adc_values[NUM_SENSORS];           //
current ADC value for each channel
    static uint16_t adc_peaks[NUM_SENSORS];           //
store peak value
    static uint8_t sample_counts[NUM_SENSORS];         //
track how many samples taken per sensor
    static bool spike_flags[NUM_SENSORS];             //
flag for when the sensor is above or below SENSOR_CUTOFF

    if (num_profiles > 0) { // system ready
        // generate readings for all sensors
        //      __HAL_ADC_CLEAR_FLAG(&hadc1, ADC_FLAG_EOS);
        // read ADC values from all channels
        for (uint8_t i = 0; i < NUM_SENSORS; i++) {
            HAL_ADC_Start(&hadc1);
            if (HAL_ADC_PollForConversion(&hadc1, 10) ==
HAL_OK) {
                current_adc_values[i] =
(uint16_t)HAL_ADC_GetValue(&hadc1);
            }
        }
        //      HAL_ADC_Stop(&hadc1);

        for (uint8_t i = 0; i < NUM_SENSORS; i++) {

            if (current_adc_values[i] > SENSOR_CUTOFF) {
                // sensor reading exceeds SENSOR_CUTOFF
                threshold, start monitoring the spike
                spike_flags[i] = true;

                if (sample_counts[i] < NUM_SAMPLES_FOR_PEAK
&& adc_peaks[i] != DIRTY_FLAG) {
                    // if not enough samples taken and the peak
                    value is not dirty

```

```

        if (adc_peaks[i] < current_adc_values[i]) {
            // during spike samples, track the
highest value
            adc_peaks[i] = current_adc_values[i];
        }
        sample_counts[i]++;
    }

    if (sample_counts[i] >=
NUM_SAMPLES_FOR_PEAK && adc_peaks[i] != DIRTY_FLAG) {
        // if enough samples taken and the peak
value is not dirty
        printf("Sensor %d Peak Value: %u\r\n", i +
1, adc_peaks[i]);

        // send the sensor data into the queue for
translation
        sensor_data_t strike;
        strike.sensor_number = i + 1; // sensor
array labeled starting with Sensor_1
        strike.peak_value = adc_peaks[i];

        enqueue(&playQueue, strike);

        sample_counts[i] = 0;        // reset sample
count
        adc_peaks[i] = DIRTY_FLAG; // set dirty
    }
} else {
    // sensor reading below SENSOR_CUTOFF
    if (spike_flags[i]) {
        if (sample_counts[i] > 0 &&
sample_counts[i] <= NUM_SAMPLES_FOR_PEAK && adc_peaks[i] !=
DIRTY_FLAG) {
            // if the value dropped while still
sampling and the peak value is not dirty
            printf("Sensor %d Peak Value: %u\r\n",
i + 1, adc_peaks[i]);

```

```

        // send the sensor data into the queue
for translation
        sensor_data_t strike;
        strike.sensor_number = i + 1;
        strike.peak_value = adc_peaks[i];

        enqueue(&playQueue, strike);
        sample_counts[i] = 0; // reset for the
next spike detection
    }
    spike_flags[i] = false;
    }
    adc_peaks[i] = 0; // reset or clear
DIRTY_FLAG from peak value
    }
}

    //      ms_delay_with_work(2); // 2 ms delay
between samples
    ms_delay(2); // 2 ms delay between samples

}

}

/**
 * @brief Function implementing the kickPedalTask thread.
 * @note Kick pedal is Pedal 1 connection on the device.
 * @param argument: Not used
 * @retval None
 */
void kick_pedal(void)
{
    // check the kick pedal state
    GPIO_PinState pedalStateRead =
HAL_GPIO_ReadPin(Pedal_1_GPIO_Port, Pedal_1_Pin);

    if (pedalStateRead == GPIO_PIN_RESET) {

```

```

// pressing the pedal
if (kick_pedal_state == BUTTON_IDLE) {
    // pedal is pressed (falling edge)
    kick_pedal_state = BUTTON_PRESSED;

    // handle pedal press event
    printf("Kick pedal pressed\r\n");

    midi_wav_t* translated_strike =
&profiles[active_profile].instruments[0].wav_files[0];
    if (!translated_strike) {
        // NULL index, should not be possible to
reach???
        printf("error: kick_pedal_task() null index
value!\r\n");
        return;
    }

    // char* full_path =
get_full_path(0, 0);

    // create open file handle if it doesn't exist
    // FIL* wav_file =
add_open_file_handle(full_path);

    // if (!wav_file) {
    // NULL file pointer
error
    // NOTE:
add_open_file_handle() can return NULL
    //
printf("kick_pedal_task() error: wav_file NULL.\r\n");
    //
    return;
    // continue;
    // }

```



```

        // if
        (add_wav_file_to_mix(wav_file,
        translated_strike->start_addr)) {
            // // no velocity
            variation for pedal (switch).
            //
            start_midi_note(translated_strike, 127);
            // }

            start_midi_note(translated_strike, 127);

        }
    } else {
        // not pressing pedal
        if (kick_pedal_state == BUTTON_PRESSED) {
            // pedal is released (rising edge)
            kick_pedal_state = BUTTON_RELEASED;

            // handle pedal release event here

        }
        if (kick_pedal_state == BUTTON_RELEASED) {
            // pedal not doing anything
            kick_pedal_state = BUTTON_IDLE;
        }
    }
}

/* USER CODE END 0 */

/**
 * @brief The application entry point.
 * @retval int
 */
int main(void)
{

```

```

/* USER CODE BEGIN 1 */

/* USER CODE END 1 */

/* MCU
Configuration-----
-----*/

/* Reset of all peripherals, Initializes the Flash
interface and the Systick. */
HAL_Init();

/* USER CODE BEGIN Init */

Queue_Init(&playQueue);

/* USER CODE END Init */

/* Configure the system clock */
SystemClock_Config();

/* USER CODE BEGIN SysInit */

/* USER CODE END SysInit */

/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_ADC1_Init();
MX_DAC1_Init();
MX_SPI1_Init();
MX_TIM4_Init();
MX_USART1_UART_Init();
if (MX_FATFS_Init() != APP_OK) {
Error_Handler();
}
MX_TIM3_Init();
/* USER CODE BEGIN 2 */

```

```

        HAL_TIM_Base_Start(&htim3); // start timer 3
        HAL_TIM_Base_Start(&htim4); // start timer 4 (TIM4
connected to APB1)

        // if (HAL_ADC_Start(&hadc1) != HAL_OK)
        // {
        //     Error_Handler();
        // }

        // // start the DAC
        // if (HAL_DAC_Start(&hdac1, DAC_CHANNEL_1) !=
HAL_OK)
        // {
        //     Error_Handler();
        // }

        all_LEDs_on(); // initialize LEDs on to show not ready
state

        // initialize before checking
        for (uint8_t i = 0; i < WAV_FILES_MAX; i++) {
            active_notes[i].midi_channel = 0;
            active_notes[i].midi_code = 0;
            active_notes[i].duration = 0;
            active_notes[i].start_time = 0;
            active_notes[i].isPlaying = false;
        }

        // create_profiles();
        create_profiles_midi_only();
        active_profile = 0;
        set_LEDs(); // show first profile is active

        if (profiles == NULL) { // if no profiles were
actually created
            num_profiles = 0; // redundant set just to be safe
            all_LEDs_on(); // indicate system is not ready from
mount operation (ON)

```

```

        // when profile select button is pressed they should
turn off
    }

    /* USER CODE END 2 */

    /* Infinite loop */
    /* USER CODE BEGIN WHILE */
    while (1)
    {
        if (num_profiles > 0) { // system ready
            headphone_detect();
            profile_button();
            kick_pedal();
            expired_notes();
            play_queue();

            sensor_array();
        }

        /* USER CODE END WHILE */

        /* USER CODE BEGIN 3 */
    }
    /* USER CODE END 3 */
}

/**
 * @brief System Clock Configuration
 * @retval None
 */
void SystemClock_Config(void)
{
    RCC_OscInitTypeDef RCC_OscInitStruct = {0};
    RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

    /** Configure the main internal regulator output
voltage

```

```

    */

HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1
);

    /** Initializes the RCC Oscillators according to the
specified parameters
    * in the RCC_OscInitTypeDef structure.
    */
    RCC_OscInitStruct.OscillatorType =
RCC_OSCILLATORTYPE_HSI;
    RCC_OscInitStruct.HSIState = RCC_HSI_ON;
    RCC_OscInitStruct.HSICalibrationValue =
RCC_HSICALIBRATION_DEFAULT;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
    {
        Error_Handler();
    }

    /** Initializes the CPU, AHB and APB buses clocks
    */
    RCC_ClkInitStruct.ClockType =
RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYCLK
        |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
    RCC_ClkInitStruct.SYCLKSource = RCC_SYCLKSOURCE_HSI;
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYCLK_DIV1;
    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

    if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct,
FLASH_LATENCY_0) != HAL_OK)
    {
        Error_Handler();
    }
}

/**

```

```

* @brief ADC1 Initialization Function
* @param None
* @retval None
*/
static void MX_ADC1_Init(void)
{

    /* USER CODE BEGIN ADC1_Init 0 */

    /* USER CODE END ADC1_Init 0 */

    ADC_MultiModeTypeDef multimode = {0};
    ADC_ChannelConfTypeDef sConfig = {0};

    /* USER CODE BEGIN ADC1_Init 1 */

    /* USER CODE END ADC1_Init 1 */

    /** Common config
    */
    hadc1.Instance = ADC1;
    hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV1;
    hadc1.Init.Resolution = ADC_RESOLUTION_12B;
    hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
    hadc1.Init.GainCompensation = 0;
    hadc1.Init.ScanConvMode = ADC_SCAN_ENABLE;
    hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
    hadc1.Init.LowPowerAutoWait = DISABLE;
    hadc1.Init.ContinuousConvMode = DISABLE;
    hadc1.Init.NbrOfConversion = 6;
    hadc1.Init.DiscontinuousConvMode = ENABLE;
    hadc1.Init.NbrOfDiscConversion = 1;
    hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
    hadc1.Init.ExternalTrigConvEdge =
ADC_EXTERNALTRIGCONVEDGE_NONE;
    hadc1.Init.DMAContinuousRequests = DISABLE;
    hadc1.Init.Overrun = ADC_OVR_DATA_OVERWRITTEN;
    hadc1.Init.OversamplingMode = DISABLE;

```

```

if (HAL_ADC_Init(&hadc1) != HAL_OK)
{
    Error_Handler();
}

/** Configure the ADC multi-mode
*/
multimode.Mode = ADC_MODE_INDEPENDENT;
if (HAL_ADCEx_MultiModeConfigChannel(&hadc1,
&multimode) != HAL_OK)
{
    Error_Handler();
}

/** Configure Regular Channel
*/
sConfig.Channel = ADC_CHANNEL_6;
sConfig.Rank = ADC_REGULAR_RANK_1;
sConfig.SamplingTime = ADC_SAMPLETIME_640CYCLES_5;
sConfig.SingleDiff = ADC_SINGLE_ENDED;
sConfig.OffsetNumber = ADC_OFFSET_NONE;
sConfig.Offset = 0;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
{
    Error_Handler();
}

/** Configure Regular Channel
*/
sConfig.Channel = ADC_CHANNEL_7;
sConfig.Rank = ADC_REGULAR_RANK_2;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
{
    Error_Handler();
}

/** Configure Regular Channel
*/

```

```

sConfig.Channel = ADC_CHANNEL_8;
sConfig.Rank = ADC_REGULAR_RANK_3;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
{
    Error_Handler();
}

/** Configure Regular Channel
*/
sConfig.Channel = ADC_CHANNEL_9;
sConfig.Rank = ADC_REGULAR_RANK_4;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
{
    Error_Handler();
}

/** Configure Regular Channel
*/
sConfig.Channel = ADC_CHANNEL_1;
sConfig.Rank = ADC_REGULAR_RANK_5;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
{
    Error_Handler();
}

/** Configure Regular Channel
*/
sConfig.Channel = ADC_CHANNEL_2;
sConfig.Rank = ADC_REGULAR_RANK_6;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
{
    Error_Handler();
}
/* USER CODE BEGIN ADC1_Init 2 */

/* USER CODE END ADC1_Init 2 */

}

```



```

/**
 * @brief DAC1 Initialization Function
 * @param None
 * @retval None
 */
static void MX_DAC1_Init(void)
{

    /* USER CODE BEGIN DAC1_Init 0 */

    /* USER CODE END DAC1_Init 0 */

    DAC_ChannelConfTypeDef sConfig = {0};

    /* USER CODE BEGIN DAC1_Init 1 */

    /* USER CODE END DAC1_Init 1 */

    /** DAC Initialization
    */
    hdac1.Instance = DAC1;
    if (HAL_DAC_Init(&hdac1) != HAL_OK)
    {
        Error_Handler();
    }

    /** DAC channel OUT1 config
    */
    sConfig.DAC_HighFrequency =
DAC_HIGH_FREQUENCY_INTERFACE_MODE_AUTOMATIC;
    sConfig.DAC_DMADoubleDataMode = DISABLE;
    sConfig.DAC_SignedFormat = ENABLE;
    sConfig.DAC_SampleAndHold = DAC_SAMPLEANDHOLD_DISABLE;
    sConfig.DAC_Trigger = DAC_TRIGGER_NONE;
    sConfig.DAC_Trigger2 = DAC_TRIGGER_NONE;
    sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;

```

```

        sConfig.DAC_ConnectOnChipPeripheral =
DAC_CHIPCONNECT_EXTERNAL;
        sConfig.DAC_UserTrimming = DAC_TRIMMING_FACTORY;
        if (HAL_DAC_ConfigChannel(&hdac1, &sConfig,
DAC_CHANNEL_1) != HAL_OK)
        {
            Error_Handler();
        }
        /* USER CODE BEGIN DAC1_Init 2 */

        /* USER CODE END DAC1_Init 2 */

    }

/**
 * @brief SPI1 Initialization Function
 * @param None
 * @retval None
 */
static void MX_SPI1_Init(void)
{

    /* USER CODE BEGIN SPI1_Init 0 */

    /* USER CODE END SPI1_Init 0 */

    /* USER CODE BEGIN SPI1_Init 1 */

    /* USER CODE END SPI1_Init 1 */
    /* SPI1 parameter configuration*/
    hspi1.Instance = SPI1;
    hspi1.Init.Mode = SPI_MODE_MASTER;
    hspi1.Init.Direction = SPI_DIRECTION_2LINES;
    hspi1.Init.DataSize = SPI_DATASIZE_8BIT;
    hspi1.Init.CLKPolarity = SPI_POLARITY_LOW;
    hspi1.Init.CLKPhase = SPI_PHASE_1EDGE;
    hspi1.Init.NSS = SPI_NSS_SOFT;

```

```

        hspi1.Init.BaudRatePrescaler =
SPI_BAUDRATEPRESCALER_2;
        hspi1.Init.FirstBit = SPI_FIRSTBIT_MSB;
        hspi1.Init.TIMode = SPI_TIMODE_DISABLE;
        hspi1.Init.CRCCalculation =
SPI_CRCCALCULATION_DISABLE;
        hspi1.Init.CRCPolynomial = 7;
        hspi1.Init.CRCLength = SPI_CRC_LENGTH_DATASIZE;
        hspi1.Init.NSSPMode = SPI_NSS_PULSE_ENABLE;
        if (HAL_SPI_Init(&hspi1) != HAL_OK)
        {
            Error_Handler();
        }
        /* USER CODE BEGIN SPI1_Init 2 */

        /* USER CODE END SPI1_Init 2 */

    }

/**
 * @brief TIM3 Initialization Function
 * @param None
 * @retval None
 */
static void MX_TIM3_Init(void)
{

    /* USER CODE BEGIN TIM3_Init 0 */

    /* USER CODE END TIM3_Init 0 */

    TIM_ClockConfigTypeDef sClockSourceConfig = {0};
    TIM_MasterConfigTypeDef sMasterConfig = {0};

    /* USER CODE BEGIN TIM3_Init 1 */

    /* USER CODE END TIM3_Init 1 */
    htim3.Instance = TIM3;

```

```

        htim3.Init.Prescaler = 16000-1;
        htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
        htim3.Init.Period = 65535;
        htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
        htim3.Init.AutoReloadPreload =
TIM_AUTORELOAD_PRELOAD_DISABLE;
        if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
        {
            Error_Handler();
        }
        sClockSourceConfig.ClockSource =
TIM_CLOCKSOURCE_INTERNAL;
        if (HAL_TIM_ConfigClockSource(&htim3,
&sClockSourceConfig) != HAL_OK)
        {
            Error_Handler();
        }
        sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
        sMasterConfig.MasterSlaveMode =
TIM_MASTERSLAVEMODE_DISABLE;
        if (HAL_TIMEx_MasterConfigSynchronization(&htim3,
&sMasterConfig) != HAL_OK)
        {
            Error_Handler();
        }
        /* USER CODE BEGIN TIM3_Init 2 */

        /* USER CODE END TIM3_Init 2 */

    }

/**
 * @brief TIM4 Initialization Function
 * @param None
 * @retval None
 */
static void MX_TIM4_Init(void)
{

```

```

/* USER CODE BEGIN TIM4_Init 0 */

/* USER CODE END TIM4_Init 0 */

TIM_ClockConfigTypeDef sClockSourceConfig = {0};
TIM_MasterConfigTypeDef sMasterConfig = {0};

/* USER CODE BEGIN TIM4_Init 1 */

/* USER CODE END TIM4_Init 1 */
htim4.Instance = TIM4;
htim4.Init.Prescaler = 0;
htim4.Init.CounterMode = TIM_COUNTERMODE_UP;
htim4.Init.Period = 362;
htim4.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
htim4.Init.AutoReloadPreload =
TIM_AUTORELOAD_PRELOAD_DISABLE;
if (HAL_TIM_Base_Init(&htim4) != HAL_OK)
{
Error_Handler();
}
sClockSourceConfig.ClockSource =
TIM_CLOCKSOURCE_INTERNAL;
if (HAL_TIM_ConfigClockSource(&htim4,
&sClockSourceConfig) != HAL_OK)
{
Error_Handler();
}
sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
sMasterConfig.MasterSlaveMode =
TIM_MASTERSLAVEMODE_DISABLE;
if (HAL_TIMEx_MasterConfigSynchronization(&htim4,
&sMasterConfig) != HAL_OK)
{
Error_Handler();
}
/* USER CODE BEGIN TIM4_Init 2 */

```

```

        /* USER CODE END TIM4_Init 2 */

    }

/**
 * @brief USART1 Initialization Function
 * @param None
 * @retval None
 */
static void MX_USART1_UART_Init(void)
{

    /* USER CODE BEGIN USART1_Init 0 */

    /* USER CODE END USART1_Init 0 */

    /* USER CODE BEGIN USART1_Init 1 */

    /* USER CODE END USART1_Init 1 */
    huart1.Instance = USART1;
    huart1.Init.BaudRate = 31250;
    huart1.Init.WordLength = UART_WORDLENGTH_8B;
    huart1.Init.StopBits = UART_STOPBITS_1;
    huart1.Init.Parity = UART_PARITY_NONE;
    huart1.Init.Mode = UART_MODE_TX;
    huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
    huart1.Init.OverSampling = UART_OVERSAMPLING_16;
    huart1.Init.OneBitSampling =
UART_ONE_BIT_SAMPLE_DISABLE;
    huart1.Init.ClockPrescaler = UART_PRESCALER_DIV1;
    huart1.AdvancedInit.AdvFeatureInit =
UART_ADVFEATURE_NO_INIT;
    if (HAL_HalfDuplex_Init(&huart1) != HAL_OK)
    {
        Error_Handler();
    }
}

```

```

        if (HAL_UARTEx_SetTxFifoThreshold(&huart1,
UART_TXFIFO_THRESHOLD_1_8) != HAL_OK)
        {
            Error_Handler();
        }
        if (HAL_UARTEx_SetRxFifoThreshold(&huart1,
UART_RXFIFO_THRESHOLD_1_8) != HAL_OK)
        {
            Error_Handler();
        }
        if (HAL_UARTEx_DisableFifoMode(&huart1) != HAL_OK)
        {
            Error_Handler();
        }
        /* USER CODE BEGIN USART1_Init 2 */

        /* USER CODE END USART1_Init 2 */

    }

/**
 * @brief GPIO Initialization Function
 * @param None
 * @retval None
 */
static void MX_GPIO_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStruct = {0};
    /* USER CODE BEGIN MX_GPIO_Init_1 */

    /* USER CODE END MX_GPIO_Init_1 */

    /* GPIO Ports Clock Enable */
    __HAL_RCC_GPIOC_CLK_ENABLE();
    __HAL_RCC_GPIOA_CLK_ENABLE();
    __HAL_RCC_GPIOB_CLK_ENABLE();

    /*Configure GPIO pin Output Level */

```

```

        HAL_GPIO_WritePin(LED_1_GPIO_Port, LED_1_Pin,
GPIO_PIN_RESET);

        /*Configure GPIO pin Output Level */
        HAL_GPIO_WritePin(GPIOC,
LED_2_Pin|LED_3_Pin|LED_4_Pin|Amplifier_Pin,
GPIO_PIN_RESET);

        /*Configure GPIO pin Output Level */
        HAL_GPIO_WritePin(SD_CS_GPIO_Port, SD_CS_Pin,
GPIO_PIN_RESET);

        /*Configure GPIO pin : PA6 */
        GPIO_InitStruct.Pin = GPIO_PIN_6;
        GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
        GPIO_InitStruct.Pull = GPIO_NOPULL;
        HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

        /*Configure GPIO pin : Pedal_1_Pin */
        GPIO_InitStruct.Pin = Pedal_1_Pin;
        GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
        GPIO_InitStruct.Pull = GPIO_PULLUP;
        HAL_GPIO_Init(Pedal_1_GPIO_Port, &GPIO_InitStruct);

        /*Configure GPIO pins : Pedal_2_Pin Button_Pin */
        GPIO_InitStruct.Pin = Pedal_2_Pin|Button_Pin;
        GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
        GPIO_InitStruct.Pull = GPIO_PULLUP;
        HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

        /*Configure GPIO pin : LED_1_Pin */
        GPIO_InitStruct.Pin = LED_1_Pin;
        GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
        GPIO_InitStruct.Pull = GPIO_NOPULL;
        GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
        HAL_GPIO_Init(LED_1_GPIO_Port, &GPIO_InitStruct);

```



```

    /*Configure GPIO pins : LED_2_Pin LED_3_Pin LED_4_Pin
*/
    GPIO_InitStruct.Pin = LED_2_Pin|LED_3_Pin|LED_4_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

    /*Configure GPIO pin : Amplifier_Pin */
    GPIO_InitStruct.Pin = Amplifier_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_OD;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(Amplifier_GPIO_Port, &GPIO_InitStruct);

    /*Configure GPIO pin : Headphone_Pin */
    GPIO_InitStruct.Pin = Headphone_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    HAL_GPIO_Init(Headphone_GPIO_Port, &GPIO_InitStruct);

    /*Configure GPIO pin : SD_CS_Pin */
    GPIO_InitStruct.Pin = SD_CS_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(SD_CS_GPIO_Port, &GPIO_InitStruct);

    /* USER CODE BEGIN MX_GPIO_Init_2 */

    /* USER CODE END MX_GPIO_Init_2 */
}

/* USER CODE BEGIN 4 */

/* USER CODE END 4 */

/**

```

```

    * @brief This function is executed in case of error
    occurrence.
    * @retval None
    */
void Error_Handler(void)
{
    /* USER CODE BEGIN Error_Handler_Debug */
    /* User can add his own implementation to report the
    HAL error return state */
    __disable_irq();
    while (1)
    {
    }
    /* USER CODE END Error_Handler_Debug */
}

#ifdef USE_FULL_ASSERT
/**
 * @brief Reports the name of the source file and the
    source line number
 *          where the assert_param error has occurred.
 * @param file: pointer to the source file name
 * @param line: assert_param error line source number
 * @retval None
 */
void assert_failed(uint8_t *file, uint32_t line)
{
    /* USER CODE BEGIN 6 */
    /* User can add his own implementation to report the
    file name and line number,
    ex: printf("Wrong parameters value: file %s on line
    %d\r\n", file, line) */
    /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */

```