High-Performance Audio (HIPA) Interface
Final Report / User Manual
EE486C Capstone Project

Team members

Robert Kahsin

Michael Garcia Ramirez

Zhenzhou Lu

Clients

Paul Flikkema

Richard Hofstetter

Graduate Teaching Advisor

Jordan Beverly

School of Informatics, Computing & Cyber Systems

Northern Arizona University

05/23/2023

Table of Contents

Table of Figures

List of Acronyms

- ADC - Analog to Digital Converter
- DAC - Digital to Analog Converter
- WSoC - Wireless System On Chip
- HIPA - High-Performance Audio
- PCB - Printed Circuit Board
- I2S - Inter IC Sound
- FH - High Cut off frequency
- FL - Low Cut off frequency
- PCM - Pulse Code Modulation
- dBFS - Decibels relative to Full Scale
- TI - Texas Instruments
- ASC - Analog signal conditioning

Introduction

The HIPA Interface capstone team will enclose within this report the technical approach specification used to accomplish the goals of the project, as well as general user maintenance instructions. The project's engineering concept for solving the proposed problem will be discussed and include an in-depth technical description of how to maintain the audio interface. After that, the team will discuss the design problems that have come with the original system and the test that can be used to confirm that the design follows the engineering requirements. Finally, this document will conclude with the future of the system and with proposed solutions and improvements.

Problem Statement

Forestry researchers need a portable audio interface that can collect audio recordings of harmful bark beetles within trees to help them further study their behaviors and weaknesses. This capstone project's goal is to design and manufacture a high-fidelity, low-cost, and portable audio recording system that can be used by ecologists.
This project's requirements were carefully considered in collaboration with the Forestry department at NAU to iterate on past audio interface development progress by improving the portability, cost, and fidelity of their current recording equipment.

Project Goals

Goals derived from the project problem statement are listed below:

- Record audio from trees, with active listening maintained
- Audio must be amplified and high-fidelity
- System must be low-cost
- System must be portable

These requirements are dictated by the general goals of the forestry department to accomplish their mission to quickly detect the insects within trees. These preliminary goals act as criteria for the team to determine if the project was a success or not.

Engineering Requirements

The engineering requirements as developed by carefully considering the project goals are listed below:

- Sample audio from trees
- Record sampled audio to SD card
- System dimensions smaller than 6″ x 6″ x 6″
- Noise floor of sampled audio less than -60 dBFS
- Analog anti-aliasing filter applied to sensor signal

In order to ensure compliance with the general goals described by the clients of the HIPA Interface project, the team developed the specified requirements list. The first requirement to sample audio from trees was accomplished by using a piezoelectric sensor that was able to generate an analog audio signal directly from the tree's surface.The second requirement to record the sampled audio was completed by integrating two systems together to route the audio to our constrained audio codec for digitization. The third requirement to reduce the overall system dimensions was easily maintained by considering the hardware connections between the different systems that were integrated. The fourth requirement to maintain a minimal noise floor was accomplished by ensuring each individual system component that interacted with the audio signal, whether hardware or software, was of the highest possible quality. The final requirement to incorporate an anti-aliasing analog filter was completed by simulating the proposed circuit extensively during the research and development phase of the project.

Constraints

The following list of parts acted as constraints that were required to be a part of the system by the project sponsors:

- Piezoelectric Disc Transducer
- TLV2772 - Op amp
- SGTL5000 - Stereo Codec
- STM32WL55JC - WSoC

These parts were assigned by the client to be used by the design team. The piezoelectric disc transducers were proposed as the original solution to keep the system from being destructive to the test fixture. The team had to create the analog anti-aliasing circuit with the operational amplifier that has been selected. For the process of digitization of the conditioned signal, the ADC and DAC of the specified stereo codec completed both functions. Finally, the WSoC was required to be used as the microcontroller of the design due to its radio frequency capabilities and ability to enhance the project far into the future.

Proposed Solution

The design developed for the HIPA interface consisted of four major system elements: a piezoelectric sensor as the microphone input, a combination amplifying and filtering circuit that conditions the input signal, a stereo codec for the analog to digital conversion and data routing, and a WSoC that handled the power distribution and system controls.

The sensor chosen by the stakeholders of the project were two piezoelectric disc transducers that generate the analog audio signal from the tree. Affixed to the disc transducer there is a small piece of wood with a single phonograph needle running through it to carry the signal from the tree without needlessly damaging it. The sensor's output is created from the stress imposed on the needle which induces a charge upon the parallel plates of the sensor, which is proportional to the vibrations in the wood. This differential charge output from the sensor can be then integrated, amplified, and filtered simultaneously using a charge mode amplifier circuit (the same analog signal conditioning circuit discussed earlier) to be measured as an instantaneous voltage by an ADC. The operational amplifier that will be used is the TLV2772 high slew rate amplifier.

The stereo codec which houses the ADC and DAC necessary to the audio routing and SD card saving of the sampled signal is the SGTL5000 chip that is conveniently integrated onto a proprietary Teensy AudioShield. Once the data has been converted into a digital format, it's then able to route that signal to an audio output like a headphone jack with the help of a DAC, or communicate through a serial protocol that can write data to an SD card for storage. The Teensy Audioshield will then need the

Teensy 4.1 controller board as a driver to allow the signal to be successfully routed through our integrated system of chips.
The WSoC was used to control the different configurable recording modes of the Teeny 4.1 controller board through the use of on-board buttons and LEDs. The WSoC was also used as a power regulator from the necessary lithium-polymer battery pack to the Teensy 4.1, AudioShield, and analog signal conditioning PCBs. This WSoC was ultimately underutilized by the project team due to the inclusion of the Teensy 4.1 controller board, which leaves a vast amount of processing power available to the team which is planned to enhance the HIPA Interface project moving forward.

Analog Signal Conditioning PCB

The analog circuit will be based on the charge amplifier design which requires a rail-to-rail op amp to be integrated with a feedback loop running from the inverting input to the output. Referencing the circuit depicted in Figure 1 below, the feedback loop will consist of a capacitor and resistor in parallel, which act as an integrator for the differential charge per time output from the sensor, as well as a power consumer which keeps the op amp from reaching its saturation point. There is also a voltage source of Vcc/2 which connects to the noninverting input of the op amp which acts as a DC bias offset which the amplified signal is centered around as Vout, which ensures compliance with the input voltage tolerances of the audio codec's ADC.

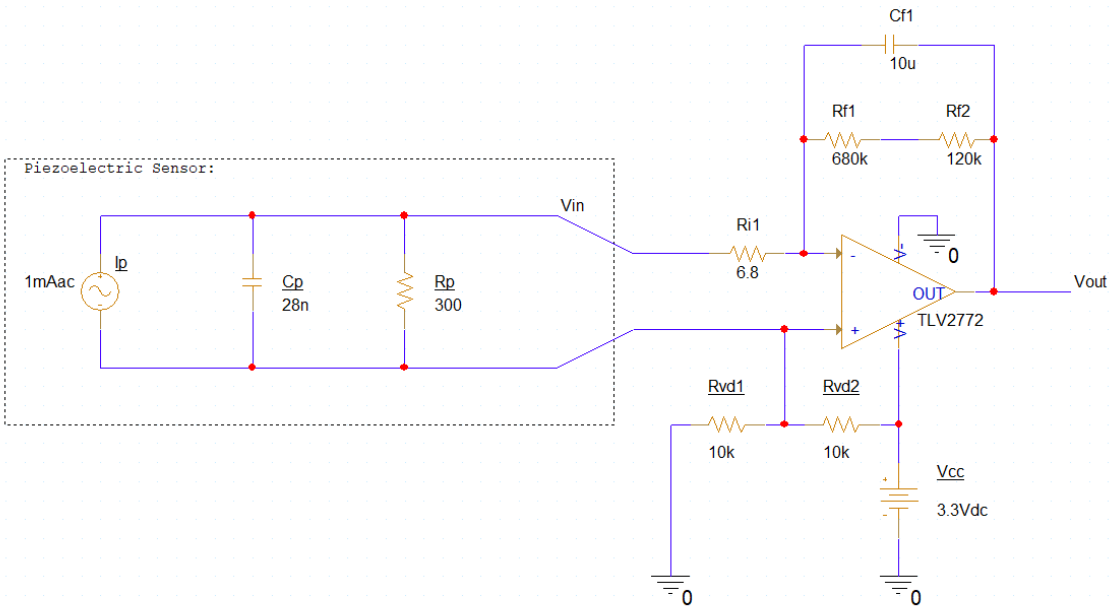


Figure 1: Charge Mode Amplifier Circuit

There are four equations listed below which must be considered when tuning the filtering, integrating, and amplification specification of the charge mode amplifier. Firstly in the output voltage equation, the Vout must be centered around the center point voltage of roughly 1.6V considering the allowance of the ADC (from GND to 3.6V maximum) to avoid losing data. Secondly in the high cutoff frequency equation, the high cutoff frequency must be set to 20kHz so that we filter out frequencies about this human audible range. Thirdly in the low cutoff frequency equation, we must set the low cutoff frequency to 20 Hz since this is the lower range of human hearing capability. Finally in the output gain equation, we must configure the gain to be as high as possible while prioritizing the previous filtering equations.

Output Voltage Equation: $V_o = -\frac{q_p}{C_f} + \frac{V_{cc}}{2}$

High Cutoff Frequency Equation: $f_H = -\frac{1}{2*\pi*R_f*C_f}$

Low Cutoff Frequency Equation: $f_L = -\frac{1}{2 * \pi * R_i * C_p}$

Output Gain Equation: $A_{db} = 20 * log_{10}(\frac{C_p}{C_f})$

This increasing priority is to condition the amplified signal to be compatible with the microcontroller sampling rate and the ADC analog input voltage swing allowance. This charge mode amplifier makes up the bulk of our proprietary PCB design, and due to having two channels on our specific op amp, we're able to record two duplicate recordings using two sensors at the same time. Both signals are fed into the audio codec, while the power supply is run from the WSoC microcontroller to the audio codec using jumper wires, and then routed from the Teensy 4.1 board to the charge mode PCB, as depicted below in Figure 2:

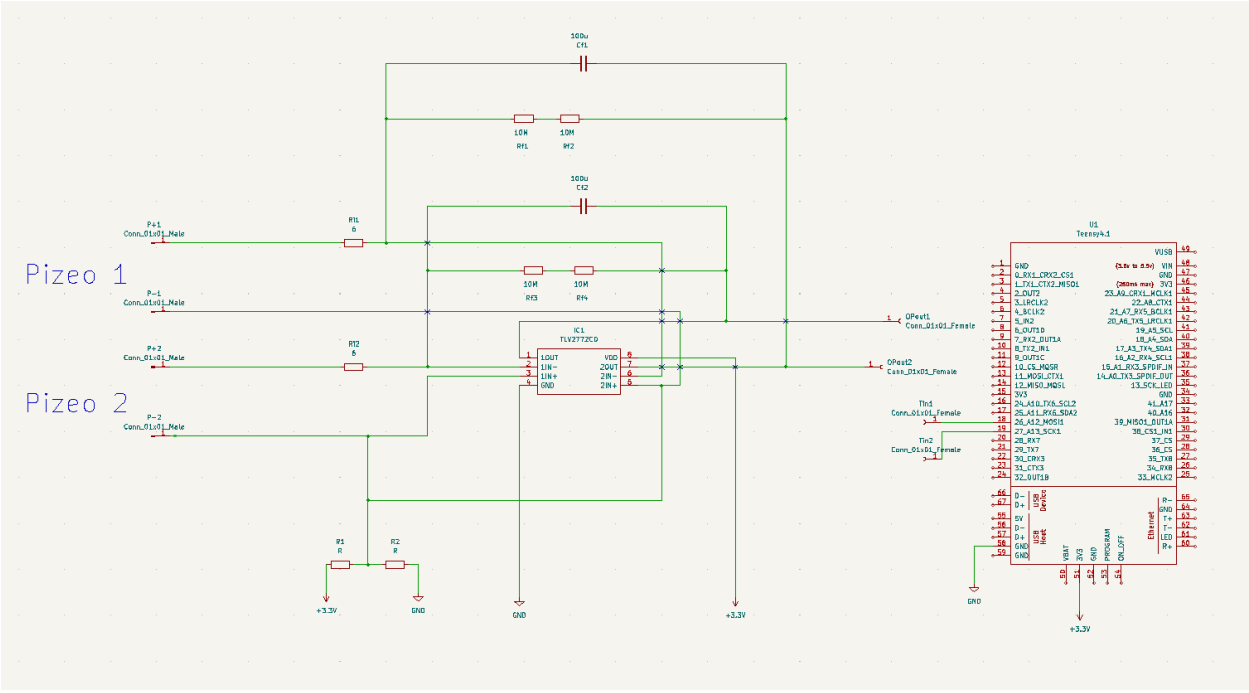


Figure 2: Charge Mode Amplifier PCB Schematic

Unfortunately, the power routing of the turn-two PCB design didn't align with the power supplies of the Teensy 4.1, so adaptations had to be made to ensure power is routed to both non-inverting op-amp input terminals.

Depicted below is the physical description of the PCB once it is printed and shipped to the team for further assembly. Once received, the analog signal conditioning circuit board is not populated with components, so manual soldering is necessary to prepare the board for further system integration.

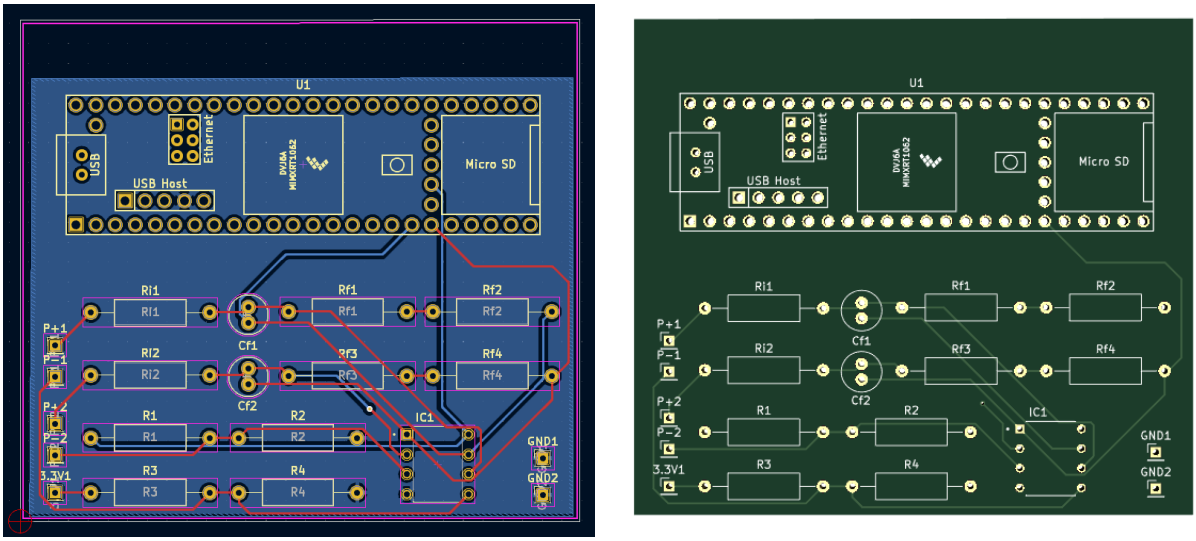


Figure 3: Analog Signal Conditioning PCB Physical Schematic

In order to solder the correct components into the correct locations on the PCB, please refer to the Figure 1 charge mode amplifier schematic. Present on the PCB are labels such as Ri1, Ri2, Cf1, Cf2, IC1, and many more which directly correspond to the components of the charge mode amplifier circuit. Since there are two channels being amplified by the amplifier, each component is present twice.

Some oddities of the analog signal conditioning PCB layout are that R1 through R4 are simply the voltage dividers used to supply Vcc/2 to the noninverting inputs on the op-amp, and are represented by Rvd1 through Rvd4 on the simulation schematic in Figure 1. Another is that instead of a single Rf resistor per channel, there is a series connected Rf1 and Rf2 which is necessary since there doesn't exist a 800kΩ resistor, so two in series are needed to add up to this desired value. Finally, the GND ports on the bottom right of the PCB are to be left unused, but were included initially due to the concern that GND connections may need to be supplied to the LINE_IN ports on the Audio Shield, but this ended up not being the case.

Below are listed the components necessary to solder the analog conditioning PCB together and their corresponding values:

- IC1 = TLV2772 Op-Amp
- Ri1 = 6.8Ω
- Ri2 = 6.8Ω
- R1 = Rvd1 = 10kΩ
- R2 = Rvd2 = 10kΩ
- R3 = Rvd3 = 10kΩ
- R4 = Rvd4 = 10kΩ
- Cf1 = 10μF
- Cf2 = 10μF
- Rf1 = 120kΩ
- Rf2 = 680kΩ
- Rf3 = 120kΩ
- Rf4 = 680kΩ

Software Implementation

The system runs on two different microcontrollers and requires two different IDEs: the Teensyduino IDE and STM32CubeIDE. The installation process for these environments will depend on the machine that is being implemented upon. [1][2] The code for the main drivers of these IDEs will be found in Appendix A-C. Using the Audio System Design tool provided by the Teensy development team, figure 4 was created as a solution to interface with the stereo codec and its peripherals. [3]

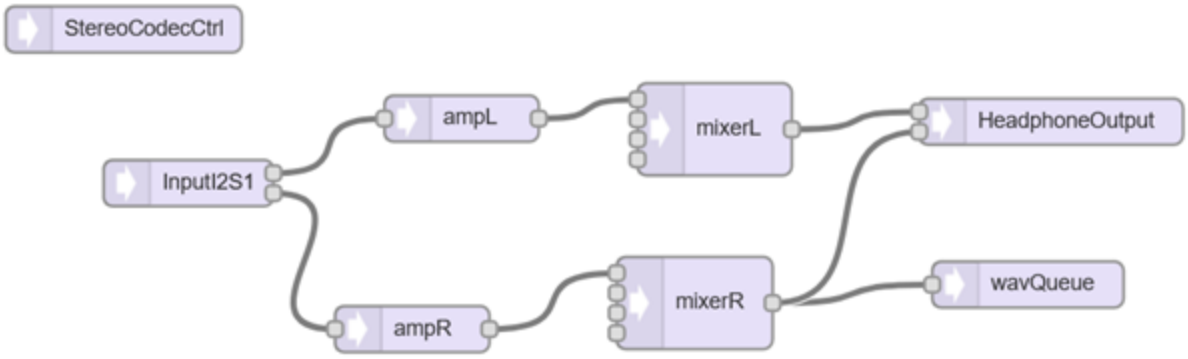


Figure 4: Audio system diagram with headphone output and recording queue

When the analog signal comes from the signal conditioning circuit, the input is isolated from the left and the right channel to the Line In port of the audio shield. The SGTL5000 will convert the signal into a 16 bit PCM digital signal that can be stored in the Teensy 4.1 audio memory block. The system uses the I2S communication protocol to send the signal throughout the audio system diagram. The digital amplifier can add analog gain before it is put into the mixer. It can also act as a switch for that specific channel to cut off the signal being input through. The headphone output takes the I2S output data and runs it through a DAC before it can be sent to the headphone peripheral on the audio shield. Appendix B gives the set up code necessary to configure the stereo codec that allows this process. This satisfies the playback requirement that set by the team.

Once that audio is stored on the Teensy 4.1 audio memory, it needs to be routed to an external memory so that it can be used for later use. Appendix B shows the process used to record a single audio data file into an SD Card. It extends the set up to include the initialization of the SD card before it can start recording. The audio data is stored into memory by the use of an audioQueue data structure because the queue data is first in first out. This will allow for buffers in the queue to accumulate the audio data and output it into the SD card in chronological order. The system uses functional programming so that the phases of start, stop, and continue can be implemented to the recording function. The startRecording function initializes the file in which the SD card is going to write to and starts the queue. Next, the continueRecording function will

populate the queue and then copy the memory from the queue to the file within the SD card. Finally, the stopRecording function will empty the queue buffers that remain and close the file so that the data stored within is saved. This completes the requirement of recording an audio sample.

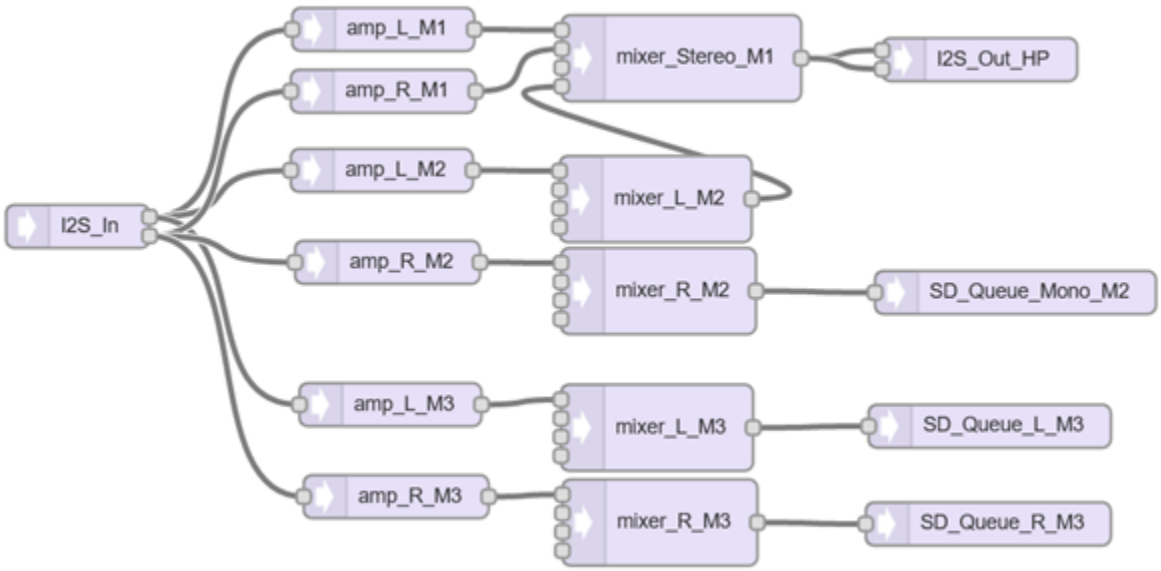


Figure 5: Audio system design for mode select

The three modes that have been designed are a stereo playback mode, a mono playback with mono recording mode, and a stereo recording mode. The separation of the modes into different mixers are to allow for the control signals that the WSoC is selecting be able to control the peripherals on the audio shield. Appendix A shows the user code that is run on the WSoC that creates the buttons that change between the mode function. The WSoC uses an interrupt service routine with the user buttons that are present on the nucleo board to send a voltage through the built in DAC that can be read by the Teensy 4.1 built in ADC. Appendix B then shows how the Teensy processor will read the data and determine what mode the stereo codec will operate. The amplifiers are used as switches to cut off the signals that are not going to be used when the mode is selected. With the expansion of Modes, we will need to have an iterative

the naming convention that the system shows will expand up to 1000000 files.

Cost Analysis

HIPA Interface BOM:

Part Description	Part Number	Supplier	Unit Price ($/unit)	Quantity (# units)
Piezo Transducer	B084KHH7B6 - 35mm	Amazon	0.37	2
Op-Amp	TLV2772CD	Digi-key	3.21	1
PCB	Analog Signal Conditioning PCB	OSH Park	12.00	1
WSoC	Nucleo Board WL55JC1	Mouser	42.00	1
Teensy 4.1	TEENSY41_NE Development Board	PJRC Store	29.60	1
Audio Shield	Audio Adaptor Board for Teensy 4.1	PJRC Store	14.40	1
Volume Knob	Tumbwheel Potentiometer for Audio Shield	PJRC Store	1.15	1
SD Card (A1 or A2)	MB-MD256KA/AM	Amazon	23.90	1
Battery Bank	Lithium-Polymer Battery	Amazon	35.95	1
3D Printed Packaging	3D Printed Packaging	MakerLab 3D Printing	19.40	1
Cf1 & Cf2 - Feedback Capacitors (10uF)	T322B106K006AT	Mouser	4.60	2
Ri1 & Ri2 - Input Resistors (6.8Ω)	HPCR0819AK6R8ST	Mouser	5.70	2
Rf2 & Rf4 - Feedback Resistors (120kΩ)	VR37000002373FR500	Mouser	0.59	2
Rf1 & Rf3 - Feedback Resistors (680kΩ)	PR02000206803JR500	Mouser	0.27	2
Rvd1 -> Rvd4 - Voltage Divider Resistors (10kΩ)	ROX9J10K	Mouser	1.51	4
			Total	210.71

Figure 6: Bill of Materials

The Bill of Materials records the necessary materials purchased by the team when creating the prototype. The model, supplier, price, and quantity of materials are shown in Figure 6. After determining the list of materials, the finance supervisor fills out the purchase request and submits it to the relevant departments of the college. In actual situations, for testing and emergency situations, the team purchased copies of the parts. If calculated at unit price, the total price is $210.71. Compared to the equipment that the client already has and costs $1300, our prototype clearly has a clear price advantage. In addition, there are still many opportunities to reduce costs in the future second-generation prototype development process. Such optional parts include the accessories that the team personally decided on such as the Battery Bank, Volume Knob, and SD card.

System Maintenance

System Assembly

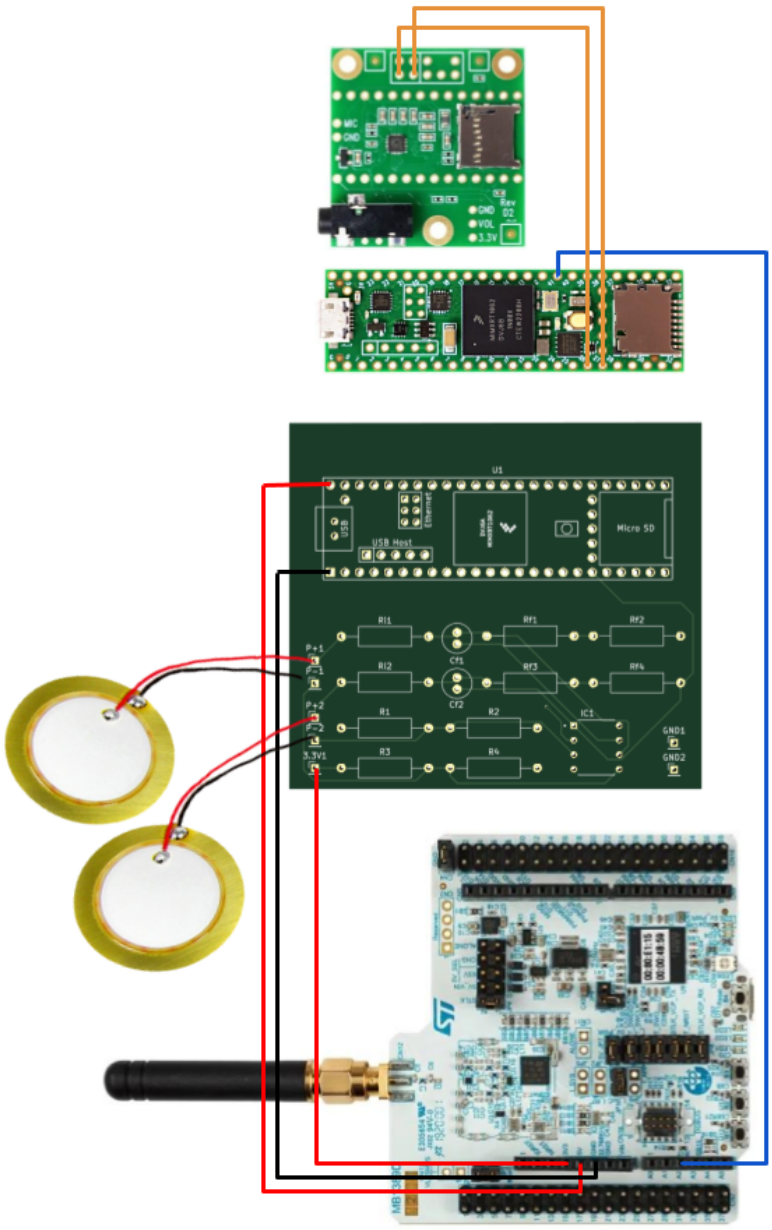


Figure 7: System Wiring Diagram

Depicted above the system wiring diagram for the HIPA Interface project. It is best practice to solder the many analog signal conditioning (ASC) PCB components on

before attempting to assemble the rest of the system. The ASC assembly instructions can be found in the Proposed Solution section of this document.

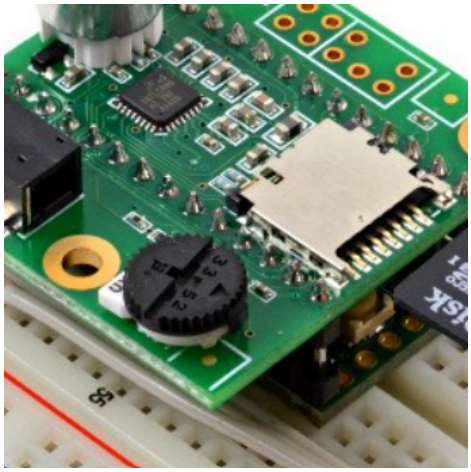
Firstly, the main rows of pinhole connections from the ASC board, Teensy 4.1, and Audio Shield must be soldered together inline. According to Teensy's documentation, it's advised to use header pins with 0.1 inch (2.54 mm) spacing and solder the connections to ensure positive connections.

Once these first three boards are soldered together, there are a number of additional jumper wires that must be soldered between the boards. There must be a jumper wire connection from pin 26 on the Teensy 4.1 to the LINE_IN_L pinhole on the Audio Shield, and from pin 27 on the Teensy 4.1 to the LINE_IN_R pinhole on the Audio Shield.

Unfortunately, the connections from the ASC board and the Teensy 4.1 don't actually route the power correctly to the various systems present, which is a discovered shortcoming of the PCB design. This means jumper wires for the 5V and GND connections to the Teensy needed to be soldered on, as well as a 3.3V connection directly to the ASC board. Following the above the wiring diagram, solder a red wire connection to the 5V Teensy 4.1 pin header sticking through the bottom of the ASC board. Then, solder a red wire connection through the bottom of the ASC board, through the bottom left pinhole. Finally, solder a black wire connection to the GND Teensy 4.1 pin header sticking through the bottom of the ASC board. Once these power connections have been made, they can simply be inserted into the corresponding WSoC female pin headers.

One more connection must be made via a jumper wire between the WSoC and the Teensy 4.1 boards. Solder a wire connection to the pin 41 header sticking out of the top of the Teensy 4.1 board, and then insert the other end of the jumper wire into the A2 female pin header on the WSoC. This connection enables the WSoC mode selection communication to be achieved.

Finally, to attach the piezoelectric sensor leads start by stripping the pre-soldered wire connections so there is enough exposed wire to be soldered to the ASC board pinholes. Tinning the newly exposed wires of the piezo will help with inserting them into the pinholes. Place the red wire from the piezo into the P+1 pinhole in the ASC board and solder the connection, followed by the black wire being soldered into the P-1 pinhole. Then repeat the last steps to connect the second piezo sensor to the P+2 and P-2 pinholes in the ASC board.



The volume knob (thumbwheel potentiometer) must be soldered in the correct spot onto the Audio Shield, as shown in the above depiction. The potentiometer must be inserted through the 3.3V, VOL, and GND pinholes with the thumbwheel facing away from the black auxiliary jack. Solder these three connections, and trim the excess pins if necessary.
Lastly, the power bank of choice must be connected via micro-USB cable to the WSoC in order for it to supply power to the rest of the system.

System Programming

The system will use two IDEs to install the programs for the Teensy 4.1 board and WSoC board. It is imperative that the version control for the systems are used for their respective boards. In addition to the IDEs, the team recommends downloading a file archiver to extract the compressed state. The Teensy 4.1 board requires the Arduino 2.x version. Instructions to download the Teensy Library are included. The WSoC board program is developed on the STM32Cube IDE version 11.2. Newer versions are available for the STM32 IDE version but will be able run the program.

The github code repository is provided by the team and includes two branches. Appendix D includes instructions on how to download the zip files of the software. For the WSoC board software, download the zip file from the WSoC branch. [4] For the Teensy board software, download the .ino file from the Teensy branch. [5]

Packaging

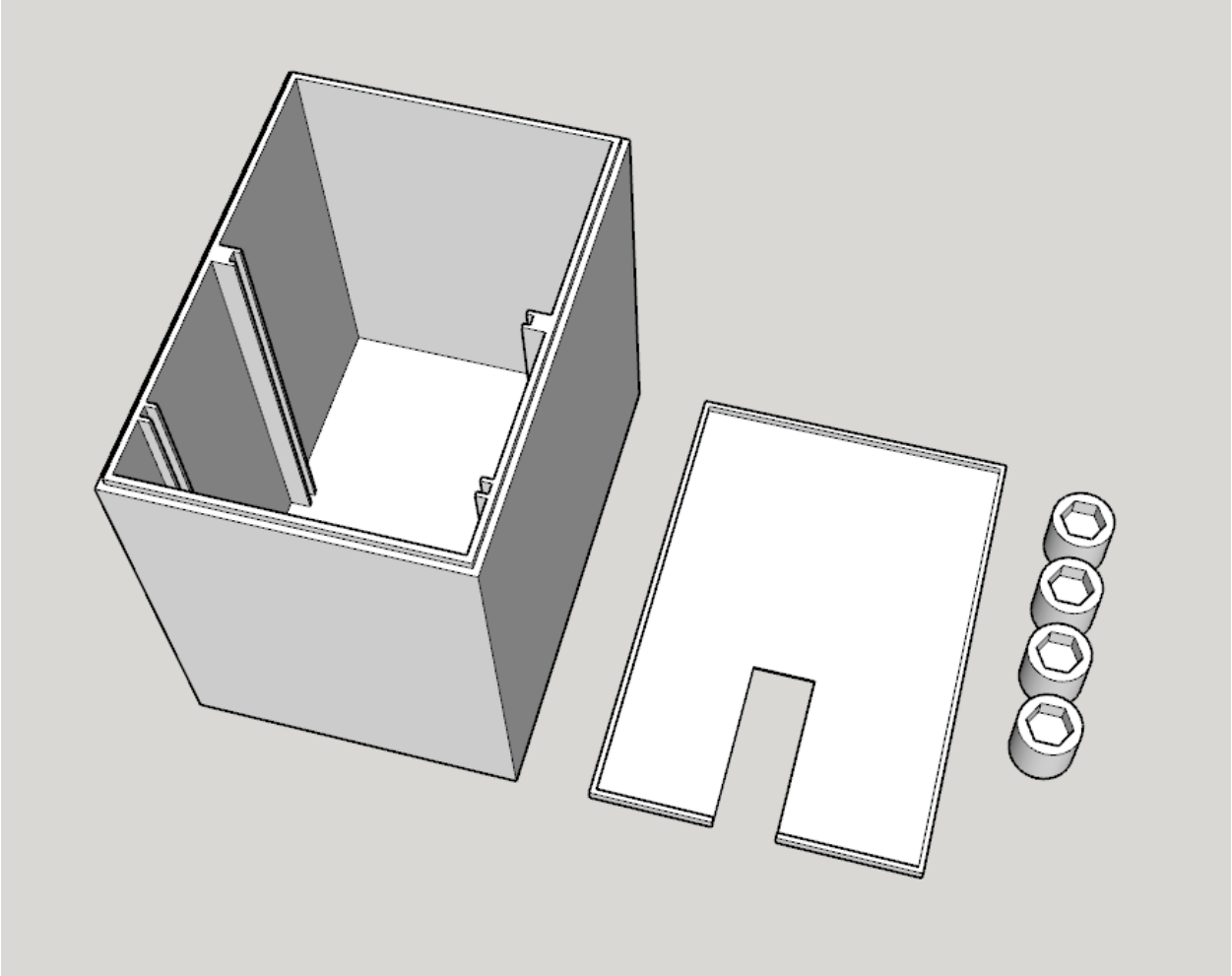


Figure 8: Packaging Design

The team designed a 3D printing package using SketchUp to accommodate the entire main circuit. As shown in Figure 8, two sets of drawers are added to the left and right inner walls to facilitate the insertion of WSoC and signal adjustment circuit boards. The front of the packaging is designed as a detachable cover, and there is a window in the middle and lower part of the cover, which is used to connect internal circuits with external power sources and piezoelectric sensors using cables.

In addition, four 1-inch high feet are added to the bottom of the packaging, which can to some extent reduce the impact of the natural environment on the system. In future plans, the team will use aluminum foil to cover the inner surface of the packaging,

which can greatly reduce the impact of electromagnetic noise from the natural environment on the system.

Recording Mode Instructions

The system will need to have the most up to date program for both the WSoC (STM IDE) and Teensy 4.1 (Teensyduino) boards respectively. Once the boards are manually programmed, the system will be in a standby state where power will be distributed but the audio is only being saved to memory. The Teensy Board set up code will wait until the SD card is input before it goes into the main function of the mode select program. Once the SD card is inserted the push buttons on the WSoC will control the states of the machine. It is recommended that the user always starts on mode 1 because this allows for the audio output to be routed to the headphones and they can hear that the sensors are properly inserted into the test fixture. Making a scratch on the wood has been the most reliable source of audio that can be distinguished by the project team and client. After confirming that the sensors are recording, the user is free to choose either mode 2 or 3 to start the recording process by pressing the respective button on the WSoC and waiting for the corresponding LED to flash. Mode 2 will allow the user to hear from headphones a mono signal from the left line input and record a mono signal from the right line input to the SD Card. This will produce one file that is in the next file sequence. Mode 3 will allow the user to record both inputs to the SD card into two respective audio files. The mode produces two files that are in the next file sequence or one file if the memory space is full. This mode is recommended when the device is being left to record unsupervised. When the user wishes to stop recording, then the user must push the mode one button but does not need to listen to the headphones.

File Conversion MATLAB Script

Once the files are put onto the SD card, they need to be encoded with the sample rate that the processor used to make the audio files. Appendix C shows a Matlab script that will do this function and convert the raw audio file into a wav format. It does this by simply identifying each raw file in a directory, encoding the sample rate of 44.1ksps and then saving these encoded files as the playable wav format.
The code will optionally prompt the user to insert the file name that needs to be analyzed and then output the waveform that the file had into the signal analyzer. The

signal analyzer tool baked into MATLAB is a powerful tool that can be configured to analyze the spectrum of the recorded audio samples.

SD Card Maintenance

The micro SD card must be inserted into the Teensy 4.1 controller board, not the Audio Shield. The Teensy 4.1 board's SD card reader is easier to access for the user, and that particular card reader must be hard coded within the Teensyduino program configuration.

The SD card that comes included with the working HIPA Interface prototype is a Samsung 256GB A2 rated card. This card was chosen due to the Teensy development team's documentation stating that Audio Shield works best with A1 or A2 rated SD cards. Any other card with the specified A1 or A2 ratings can be used instead, especially if more storage is desired.

The Teensy 4.1 program written by the project team supports up to 10000 recordings, with the limitation being the iterative naming conventions used in the program. This limitation could be improved in the future, but for a 256GB SD card the limitation is actually the storage capacity of the storage device.

The SD card will need to be emptied occasionally in order to stop the card from completely filling. In the event that a large capacity SD card is used and the Teensy 4.1 program hits the previously mentioned naming limit, the most likely occurrence is that the device would fail to record until the card is emptied. Thus, preemptively offloading the SD card recordings before the storage limits are reached is the best solution.

General Guidance

To provide some general guidance and warn of common causes of user-error, there are some points that should be gone over.

One point of concern is that the system cannot be configured if the SD card is not inserted at startup. This is because within the Teensy 4.1 program there is a setup function that ensures there is an SD card present before the infinite loop begins. Once the SD card is inserted the system will work correctly, but it is an easy component to overlook since it's in the back of the packaging.

Another aspect that is important to keep in mind is that the packaging is designed to keep the boards very close together, and that means they need to be placed into the box

in a specific order. It's essential that the WSoC is the first board to be removed from the packaging, and the last to be inserted back inside. Specifically, the power cable routing from the bottom of the ASC board and the WSoC will run into each other, and this is one way to break a soldering connection.
Finally, comprehensive documentation is available on the HIPA Interface webpage, along with other general information and status reports from the capstone team.
https://www.ceias.nau.edu/capstone/projects/EE/2023/HiPAInterface/

Testing

The PCB needed to be tested to ensure that the signal was being properly filtered. Once the signal is in the stereo codec, it must also output the certain frequency that was asked by the forestry researchers. Stated below are the unit tests that were implemented to ensure that the values worked in the full scale integration.

1. Analog signal conditioning PCB performance tests

The PCB components consist of an op-amp, two resistors, and one capacitor which are configured to induce a gain, apply a bandpass filter, and integrate the input analog signal for use with the ADC. This conditioning circuit also needs to ensure the output voltage swings within the tolerances of the ADC input, which limits our output from -0.3V to +3.6V. In order to tune the filtering of the circuit, the resistors and capacitor need to be carefully considered in accordance with two equations that describe the upper and lower cutoff frequencies. Once these components are selected to optimize the bandpass filter for human hearing ranges, finally the gain can be tuned. The gain is limited by the values selected for the filtering of the circuit, which is a higher priority in order to ensure the signal received by the ADC is not of a frequency bandwidth that's too wide to be sampled at 48kbps.
These three characteristics of the analog signal conditioning circuit (the output voltage swing, the filtering, and the gain) are all essential components of the final design. It's vital for the analog front-end circuitry to condition the input signal at the highest fidelity possible in order for the signal to be converted to a digital format effectively. Due to this being such an integral part of the overall system design, the three characteristics of this part of the system are crucial to be tested.
The tests can be performed on the fabricated analog amplifier PCB, and will be conducted using a signal generator and oscilloscope. The output swing can be tested

with any input signal from the signal generator regardless of its power, and also tested with a piezo to mimic the real-world system configuration. No matter what kind of signal the amplifier circuit receives, it should condition that signal to swing around 1.6V, and no higher or lower than the ADC input voltage tolerances of -0.3V to 3.6V. The filtering and gain of the signal can be tested simultaneously by injecting from the signal generator a frequency modulated sweep from 0Hz to 30kHz, at a 5mV peak-to-peak power output. This input signal sweep stimulates the frequency response of the amplifier circuit. By connecting the op-amp's output to an oscilloscope, one can see that the sweep depicts the upper and lower cutoff frequencies of the filter, as well as the gain within the unfiltered part of the output signal.

2. Stereo codec analog signal processing performance tests:

In order to ensure that the signal received by the audio codec is able to direct a signal with enough gain applied to be audible to a user wearing headphones, the audio codec must apply some amount of gain before the ADC. The method to optimize the gain applied by the audio codec is to first optimize the analog signal conditioning PCB components, and then iteratively apply more and more gain via the audio codec until it is appropriately audible.

The caveat to applying more and more gain using the audio codec is that at some point the team will have to optimize the gain to be both high-fidelity and audible to the average person using headphones. There may be some tradeoff between the perceived loudness of the signal routed to the headphones, and the quality of the recorded audio, but there are methods to separately apply gain to the headphone output, and not the recorded signal sent through the ADC.

Testers will need to use a pair of headphones to judge the appropriate gain to apply using the audio codec, and iteratively change it higher or lower between test cases. Once a gain is selected, it's important to evaluate if it negatively impacts the recorded signal in any way, and thus the path forward would be to route the signal to the headphones using a parallel gain block to that of the recorded signal. Optionally, a potentiometer can be used to control a "headphone volume" parameter within the audio codec, but getting the volume to an appropriate level for recording is essential.

3. Audio shield signal reception and storage tests

The stereo codec within the audio shield receives the audio signal through two different inputs: a line in and a mic input. For the testing purposes, the input will be used in stereo input because that allows for the signal to be read from two lines through the left

and right input. A waveform generator will send in a controlled voltage swing to these inputs at select frequencies. These frequencies will range from 20Hz to 20kHz; the lowest and highest bounds that are theoretically being filtered by the signal conditioning circuit. In addition to this, the generator will supply frequencies of 250 Hz, 500Hz, 1kHz and because they are within the range of most heard sounds that humans hear in music. These frequencies will also be played for the stereo codec analog signal processing performance test.

For the output of the stereo codec, it will be recorded to an empty microSD card from the Teensy 4.1 processor. The data should be saved into a .WAV file based on the signals given to the line in pins. The microSD card can then be read on a PC with an SD card reader using an adaptor and saved on that device. Once the storage shows that the .WAV file is stored on the device, then an audio editor program can open the file and determine whether the contents of the file match the input frequencies. The audio editor program chosen will be Audacity as it is free to use and open-source while being able to see the waveforms of the data.

Criteria for P/F

1. Analog signal conditioning PCB performance tests:

In order to test that the output of the analog amplifier circuit has a voltage swing that is within the tolerances of the ADC input, one must simply test any input signal at any power rating and any frequency. Under no circumstances should the output of the analog amplifier circuit swing at any level other than 1.6V to ensure compatibility with the ADC. Since this test is so simple to test for, it can be included in the procedures of the testing matrix that test for the filtering and gain performance of the circuit.
When it comes to testing the filtering and gain performance of the analog amplifier circuit, one must pay attention to the components used, and what version of the PCB circuit is being used. As described in the previous section, very small changes in the cable lengths of the PCB can mean that the resistor and capacitor components of the circuit must be reevaluated. The testing procedures for testing these aspects of the system must be iterative, with the goal of incrementally improving the filtering and gain specifications.
The goal of the filtering test is to ensure using an oscilloscope that the frequency response of the system confirms upper and lower cutoff frequencies of 20Hz and 20kHz.

If the frequency response of the system is near that of the desired cutoff frequencies, then the team can describe that outcome as a "Pass". The gain specification can only be enhanced to a certain extent, and is limited by the components chosen for the purpose of the filter. With that in mind, the team should still record the gain that is obtained during each iteration of the test procedures. The gain can be increased to an audible level using the capabilities of the audio codec, so a "pass/fail" isn't an appropriate way to measure the gain performance, but rather the goal is simply to make it as high as possible while prioritizing the filtering.

2. Stereo codec analog signal processing performance tests:

The stereo codec is capable of applying a gain to the received signal from the analog amplifier PCB circuit. The amount of gain that can be applied ranges in 10dB increments from 0dB to +60dB, so the tester will need to iteratively check what gain is appropriate to add to the signal to make it audible. The tester will start with a gain of 0dB, and incrementally increase the gain until it reaches a level that is appropriate for the average person to listen to.

The criteria for this testing is to incrementally change the gain and evaluate it on a preferential level, since human hearing has variance. The human perception of the audio is important to the customer, so personal judgment is inescapable in such a test case. So long as the fidelity of the recorded signal is not sacrificed in any capacity, the dependency that describes a certain gain level a "pass" is that it is at a comfortable volume for the average person to listen to.

3. Audio shield signal reception and storage tests

The existence of an audio file on the microSD card will be the first part of the pass/fail criteria. If no file exists, then the data was not properly encoded and will result in an overall failure. Properly uploading the code is critical to this criterion, so any attempt where this is the case will be invalidated. Once the audio file is uploaded to the PC, the audio editor will be able to determine the times at which the signal should be accounted for and what frequency is being displayed as the output. A pass scoring would require that the waveform shows all 5 waveforms ascending from 20Hz to 20kHz. In addition, the audio file should be listened to and noted if there is any noise coming from the signal. The exclusion of any of these frequencies should be marked on its specific column as pass or fail. If more than 3 of the signals are omitted, then the whole test should be considered a failure.

Results

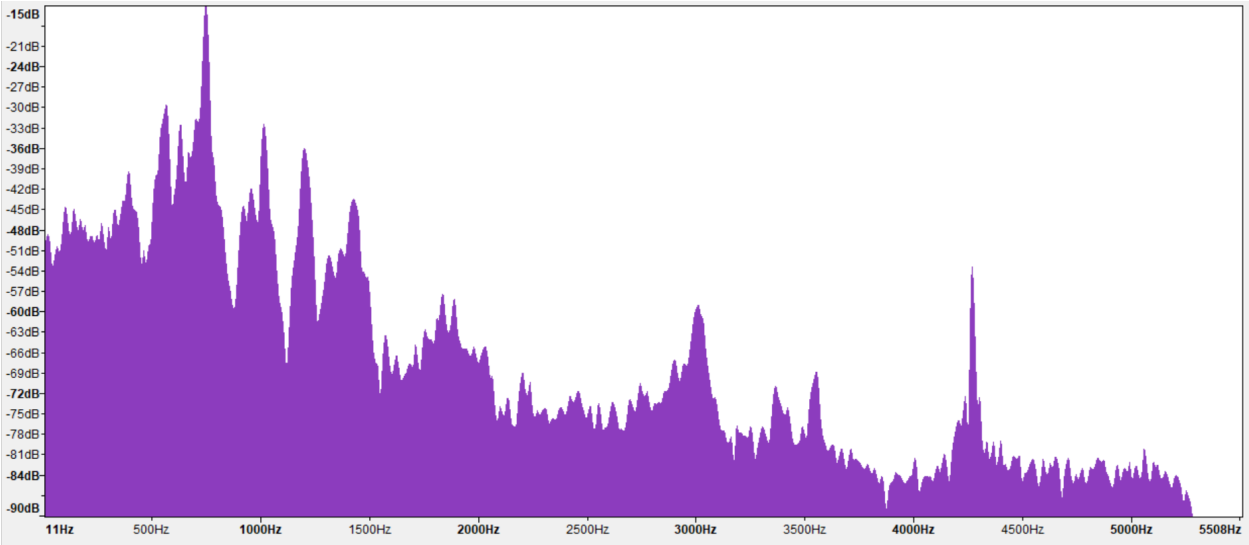


Figure 9: Frequency Response

In the frequency graph above is depicted the frequency response of the final system after recording a scratching sound on a tree. The recording was collected from a test fixture supplied to the team by the forestry department, and was a large tree sample they had collected recently.

Upon scratching the log, the audio was recorded by the HIPA Interface prototype and was run through a FFT to determine the frequency domain characteristics of the sample. With the frequency domain, we can determine the noise floor is substantially low enough for the rest of the signal to rise above, and the recording sounded clean to the human ear. There also exist spikes in the frequency response, indicating frequencies of interest should be possible to identify and learn more about in the future.

Conclusions

The goal of the HIPA Interface project to create a high-fidelity audio recording system in a portable package was achieved. All the requirements that were set by the Forestry department were met with the constraints that were put by the engineering client. The device is requiring more testing to bring it up to industry standards, but the groundwork has been established with this capstone group. The

For future investigation of the system, there is a lot of room for improvement and optimization. The constraints were proving to limit the ability of the audio system, but the engineers were able to design around them. While the WSoC was meant to be the main driver, the inclusion of the Teensy audio system proved to be a better solution and there was still use found for the WSoC. There remains lots of available processing power for future enhancements to the system with the WSoC given its radio capabilities and dual core processor. While the final design for the analog signaling PCB was only the second turn, there are points that can be added to improve the overall performance such as shielded cabling for the piezoelectric transducer, better routing for the analog signal, and even finding smaller passive components to shrink the dimensions. It would also be beneficial to retrofit a better way to supply power between the two boards and create a solid connection elsewhere.

The ultimate result from the project is to give the researchers a device that can be used to analyze the sounds of the forest and the team hopes that they will use this technology to better understand the environment. Since the audio device outputs the raw audio data, the graduate CS team should be able to build a substantial data set of files that can be used to test their machine learning algorithms. Once this process has been tested with definitive confidence values, the researchers can then automate the investigation process when surveying the surrounding forest.

Works Cited

[1] PJRC, "Audio System Design Tool for Teensy Audio Library," PJRC, 04 April 2017. [Online]. Available: https://www.pjrc.com/teensy/gui/index.html.

[2] PJRC, "Download Teensyduino, Version 1.58," PJRC, 04 04 2017. [Online]. Available: https://www.pjrc.com/teensy/td_download.html.

[3] STMicroelectronics, "Integrated Development Environment for STM32," STMicroelectronics, 06 March 2022. [Online]. Available: https://www.st.com/en/development-tools/stm32cubeide.html.

[4] M. Garcia Ramirez, "HIPA-Interface-Code WSoC Branch," 2023. https://github.com/mgr92300/HiPA-Interface-Code/tree/WSOC

[5] M. Garcia Ramirez, "HIPA-Interface-Code Teensy Branch," 2023. https://github.com/mgr92300/HiPA-Interface-Code/tree/Teensy

Appendix A - WSoC Main

```
/* USER CODE BEGIN PV */
int modeVar; // mode Select variable
uint32_t dacVolt;
uint32_t DAC_OUT[4] = {0, 1241, 2482, 3723};
/* USER CODE END PV */

int main(void)
{
  /* USER CODE BEGIN 1 */
    modeVar = 0;
    dacVolt = 0;

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* 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_TIM1_Init();
  MX_USART2_UART_Init();
  MX_DAC_Init();
```

```
  /* USER CODE BEGIN 2 */
  HAL_DAC_Start(&hdac, DAC_CHANNEL_1);
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {/* USER CODE BEGIN WHILE */
          /* USER CODE BEGIN 3 */
    if(modeVar != 0)
          {
                  HAL_DAC_SetValue(&hdac,DAC_CHANNEL_1,DAC_ALIGN_12B_R,
dacVolt);
                  HAL_Delay(1);
          }
     /* USER CODE END 3 */
  }
}

/* USER CODE BEGIN 4 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{

   if(GPIO_Pin == Button_Mode1_Pin)
          {
                  HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_15); // Toggle LED 1
                  for(int i = 0;  i<500000 ;i++);
                  HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_15); // end toggle
                  modeVar = 1;
          }
   if(GPIO_Pin == Button_Mode2_Pin)
          {

                  HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_9); // Toggle LED 2
                  for(int i = 0;  i<500000 ;i++);
                  HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_9); // end toggle
                  modeVar = 2;
          }
```

```
    if(GPIO_Pin == Button_Mode3_Pin)
        {
            HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_11); // Toggle LED 3
            for(int i = 0;  i<500000 ;i++);
            HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_11); // ed toggle
            modeVar = 3;
        }
    dacVolt = DAC_OUT[modeVar];
}
/* USER CODE END 4 */
```

Appendix B - Mode Final Teensy Driver Code

```
/*********************************************************************************
  IMPORTANT: Program Description                                          *
  Teensy_Program reads analog input from signal conditioning circuit, then reads *
  the desired audio routing mode selection from the WSoC buttons.          *
*********************************************************************************/

#include <Audio.h>
#include <Wire.h>
#include <SPI.h>
#include <SD.h>
#include <SerialFlash.h>

// GUItool: begin automatically generated code
AudioInputI2S       I2S_In;           //xy=71.49998474121094,212.99997997283936
AudioAmplifier      amp_L_M2;         //xy=283.2708053588867,173.2939395904541
AudioAmplifier      amp_L_M1;         //xy=284.50003814697266,75.99998950958252
AudioAmplifier      amp_R_M1;         //xy=284.50003814697266,118.99998092651367
AudioAmplifier      amp_R_M2; //xy=287.2708511352539,237.2939691543579
AudioAmplifier      amp_L_M3; //xy=289.2708511352539,336.29396629333496
AudioAmplifier      amp_R_M3; //xy=293.27083587646484,400.29393577575684
AudioMixer4         mixer_L_M2;       //xy=474.27075576782227,193.29395484924316
AudioMixer4         mixer_L_M3;  //xy=475.2708206176758,352.2939682006836
AudioMixer4         mixer_R_M2; //xy=476.2708168029785,257.2939682006836
AudioMixer4         mixer_R_M3; //xy=477.2708435058594,418.2939758300781
AudioMixer4         mixer_Stereo_M1;      //xy=492.49997329711914,95.99997520446777
AudioOutputI2S      I2S_Out_HP;   //xy=700.0454444885254,96.54545593261719
AudioRecordQueue            SD_Queue_R_M3; //xy=706.2708435058594,419.29393577575684
AudioRecordQueue            SD_Queue_L_M3;      //xy=707.2707214355469,351.29393768310547
AudioRecordQueue            SD_Queue_Mono_M2;
//xy=725.2707176208496,258.2938938140869
AudioConnection     patchCord1(I2S_In, 0, amp_L_M1, 0);
AudioConnection     patchCord2(I2S_In, 0, amp_L_M2, 0);
AudioConnection     patchCord3(I2S_In, 0, amp_L_M3, 0);
AudioConnection     patchCord4(I2S_In, 1, amp_R_M1, 0);
AudioConnection     patchCord5(I2S_In, 1, amp_R_M2, 0);
```

```
AudioConnection       patchCord6(I2S_In, 1, amp_R_M3, 0);
AudioConnection       patchCord7(amp_L_M2, 0, mixer_L_M2, 0);
AudioConnection       patchCord8(amp_L_M1, 0, mixer_Stereo_M1, 0);
AudioConnection       patchCord9(amp_R_M1, 0, mixer_Stereo_M1, 1);
AudioConnection       patchCord10(amp_R_M2, 0, mixer_R_M2, 0);
AudioConnection       patchCord11(amp_L_M3, 0, mixer_L_M3, 0);
AudioConnection       patchCord12(amp_R_M3, 0, mixer_R_M3, 0);
AudioConnection       patchCord13(mixer_L_M2, 0, mixer_Stereo_M1, 3);
AudioConnection       patchCord14(mixer_L_M3, SD_Queue_L_M3);
AudioConnection       patchCord15(mixer_R_M2, SD_Queue_Mono_M2);
AudioConnection       patchCord16(mixer_R_M3, SD_Queue_R_M3);
AudioConnection       patchCord17(mixer_Stereo_M1, 0, I2S_Out_HP, 0);
AudioConnection       patchCord18(mixer_Stereo_M1, 0, I2S_Out_HP, 1);
AudioControlSGTL5000        StereoCodecCtrl; //xy=81.5,20
// GUItool: end automatically generated code

// Teensy 4.1 SD pin configuration
#define SDCARD_CS_PIN     BUILTIN_SDCARD
#define SDCARD_MOSI_PIN  11
#define SDCARD_SCK_PIN   13

// Custom variables initialization
File wavFile;             // Create .wav file
File secondWavFile;             // For mode 3
int wsocMode = 0;      // Initialization of wsocMode selection variable
int teensyMode = 0;             // Initialization of teensyMode selection variable
float gainVal = 20.0;     // Gain level when amps activated later, manually tuned to confortable
level
AudioRecordQueue SD_Queue;  // Initialization og SD_Queue variable, used in start and stop
recording functions
bool recordFlag = false;

void setup() {
  Serial.begin(9600);
  AudioMemory(60);

  // Initialize all SD card queues & buffers
  SD_Queue_Mono_M2.freeBuffer();
```

```
  SD_Queue_L_M3.freeBuffer();
  SD_Queue_R_M3.freeBuffer();
  SD_Queue_Mono_M2.clear();
  SD_Queue_L_M3.clear();
  SD_Queue_R_M3.clear();

  // Initilaize all amp gains to zero (to be changed within inf loop)
  amp_L_M1.gain(0);
  amp_R_M1.gain(0);
  amp_L_M2.gain(0);
  amp_R_M2.gain(0);
  amp_L_M3.gain(0);
  amp_R_M3.gain(0);

  // Stereo Codec Control Initialization
  StereoCodecCtrl.enable();
  StereoCodecCtrl.inputSelect(AUDIO_INPUT_LINEIN);
  StereoCodecCtrl.muteHeadphone();

  // SD card Initialization (generated code)
  SPI.setMOSI(SDCARD_MOSI_PIN);
  SPI.setSCK(SDCARD_SCK_PIN);
  if (!(SD.begin(SDCARD_CS_PIN))) {  // stop here if no SD card
          while (!SD.begin(SDCARD_CS_PIN)) {
          Serial.println("Unable to access the SD card");
          delay(1000); // 1.0 sec delay
          }
  }
}

void loop()
{
  int knob = analogRead(A1);              // read the volume potentiometer position (analog input
A1)
  float vol = (float)knob / 1280.0; // calc A2 float
  StereoCodecCtrl.volume(vol);          // set 'vol' to A2 pot value

  // Watch for WSoC button presses
```

```
 int wsocMode = analogRead(A17);
 if((wsocMode > 200) && (wsocMode < 400))
 { // Recieved button 1 press signal
        teensyMode=1; // Set teensy to "Standby" mode, routing both CH1 & CH2 to HP output
 }
 else if((wsocMode > 500) && (wsocMode < 700))
 { // Recieved button 2 press signal
        teensyMode=2; // Set teensy to "Record/Listen" mode, routing CH1 to HP output and
CH2 to SD save
 }
 else if((wsocMode > 800) && (wsocMode < 1000))
 { // Recieved button 3 press signal
        teensyMode=3; // Set teensy to "Standby" mode, routing both CH1 & CH2 to SD save
 }

 // Watch for teensyMode state change, then set audio routing mode
 switch (teensyMode)
 {
        case 1: // Mode 1: CH1 & CH2 routed to HP
        StereoCodecCtrl.unmuteHeadphone();
        amp_L_M1.gain(gainVal);
        amp_R_M1.gain(gainVal);
        amp_L_M2.gain(0);
        amp_R_M2.gain(0);
        amp_L_M3.gain(0);
        amp_R_M3.gain(0);
        recordFlag = false;
        break;
        case 2: // Mode 2: CH1 routed to HP, CH2 routed to SD card
        StereoCodecCtrl.umuteHeadphone();
        amp_L_M1.gain(0);
        amp_R_M1.gain(0);
        amp_L_M2.gain(gainVal);
        amp_R_M2.gain(gainVal);
        amp_L_M3.gain(0);
        amp_R_M3.gain(0);
        setFile(2);
        recordFlag = true;
```

```
            recordingState(2);
            delay(10000);
            stopRecording(2);
            break;
            case 3: // Mode 3: CH1 & CH2 routed to SD card (seperate files)
            StereoCodecCtrl.muteHeadphone();
            amp_L_M1.gain(0);
            amp_R_M1.gain(0);
            amp_L_M2.gain(0);
            amp_R_M2.gain(0);
            amp_L_M3.gain(gainVal);
            amp_R_M3.gain(gainVal);
            setFile(3);
            recordFlag = true;
            recordingState(3);
            delay(10000);
            stopRecording(3);
            break;
            default: // teensyMode=0 upon startup
            break; // do nothing if mode not yet selected from WSoC
  }
}

//10second rec

void setFile(int mode)
{
  char wavFileName[] = "RECORDING_00000"; // create a new file
  int j = 1;                              //offset by 1 to get the next file for mode 3
  for (int i = 0; i < 1000; i++)
  { // Iterate file name
            wavFileName[10] = i/100 + '0';
            wavFileName[11] = ((i/10) % 10) + '0';
            wavFileName[12] = i% + '0';
            if (!SD.exists(wavFileName))
            { // only open a new file if it doesn't exist
            j += i;
            wavFile = SD.open(wavFileName, FILE_WRITE);
```

```
        }
  }
  if(mode == 3 && j < 999)
  {
        char secondFileName[] = "RECORDING_000";
        secondFileName[10] = j/100 + '0';
        secondFileName[11] = ((j/10) % 10) + '0';
        secondFileName[12] = j%10 + '0';
        secondWavFile =  SD.open(secondFileName, FILE_WRITE);
  }
}

void setFileWrite(int mode)
{
  //byte memLimit[1024^2];
  //while(memLimit [1024^2] != NULL)
  //{
  if (SD_Queue_Mono_M2.available() > 1 && mode == 2)
  {
        byte buffer[512];
        memcpy(buffer, SD_Queue_Mono_M2.readBuffer(), 256);
        SD_Queue_Mono_M2.freeBuffer();
        memcpy(buffer+256, SD_Queue_Mono_M2.readBuffer(), 256);
        SD_Queue_Mono_M2.freeBuffer();
        wavFile.write(buffer, 512); // write all 512 bytes to the SD card
        //memcpy(memLimit, buffer, 512);
  }
  else if(SD_Queue_L_M3.available() > 1 && SD_Queue_R_M3.available() > 1 && mode == 3)
  {
        //
        byte firstBuffer[256], secondBuffer[256];
        memcpy(firstBuffer, SD_Queue_L_M3.readBuffer(), 256);
        SD_Queue_L_M3.freeBuffer();
        wavFile.write(firstBuffer, 256); // write all 512 bytes to the SD card
        //memcpy(memLimit, firstBuffer, 256);
        //
        memcpy(secondBuffer, SD_Queue_R_M3.readBuffer(), 256);
        SD_Queue_R_M3.freeBuffer();
```

```
        wavFile.write(secondBuffer, 256); // write all 512 bytes to the SD card
        //memcpy(memLimit, secondBuffer, 256);
  }
}

void stopRecording(int mode)
{
  if(mode == 2)
  {
        SD_Queue_Mono_M2.end();
        if (SD_Queue_Mono_M2.available() > 0)
        {
        while (SD_Queue_Mono_M2.available() > 0)
        {
        wavFile.write((byte*)SD_Queue_Mono_M2.readBuffer(), 256);
        SD_Queue_Mono_M2.freeBuffer();
        }
        wavFile.close();
        }
  }
  if(mode == 3)
  {
        SD_Queue_L_M3.end();
        if (SD_Queue_L_M3.available() > 0)
        {
        while (SD_Queue_L_M3.available() > 0)
        {
        wavFile.write((byte*)SD_Queue_L_M3.readBuffer(), 256);
        SD_Queue_L_M3.freeBuffer();
        }
        }

        SD_Queue_R_M3.end();
        if (SD_Queue_R_M3.available() > 0)
        {
        while (SD_Queue_R_M3.available() > 0)
        {
        secondWavFile.write((byte*)SD_Queue_R_M3.readBuffer(), 256);
```

```
        SD_Queue_R_M3.freeBuffer();
        }
        }
        secondWavFile.close();
        }
}
```

Appendix C - Matlab Script

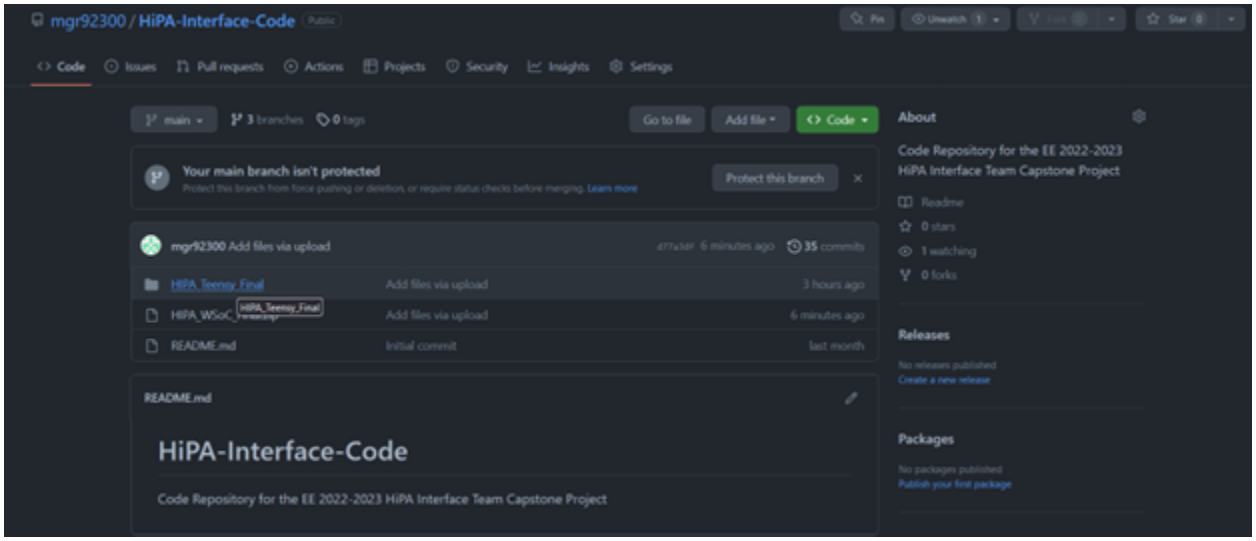
```
folder = 'E:\';                                % Set the folder containing the raw audio files
fileList = dir(fullfile(folder, '*.raw'));             % Get a list of all the raw audio files in the folder

for i = 1:length(fileList)                          % Loop over each file in the folder
   filename = fileList(i).name;                      % Get the filename of the current file
   [x, fs] = audioread(fullfile(folder, filename));    % Import the raw audio file and encode the
sample rate
   newFilename = strrep(filename, '.raw', '.wav');     % Save the audio data as a .wav file with the
specified sample rate
   audiowrite(fullfile(folder, newFilename), x, fs);
end

fprintf('RAW files successfully converted to WAV files. \n\n');
%recordingName = input('Enter file name: ', 's');               % Prompt command line for file
name
%x = audioread(recordingName);                              % Import the audio data
%signalAnalyzer(x, fs);                                 % Open the Signal Analyzer tool
```

Appendix D - Downloading the Software from Github

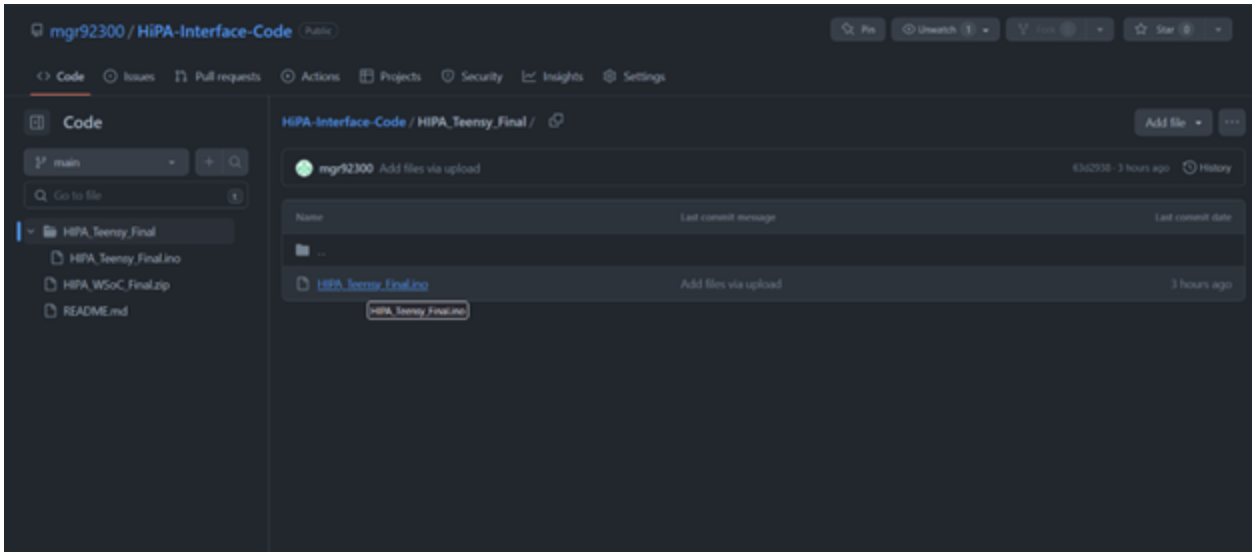
Search for the github repository that is cited in Works Cited that corresponds to the software that one is looking for. For the Teensy board, the file is a file directory. For the WSoC board, the file is a zipped file

Teensy board Download

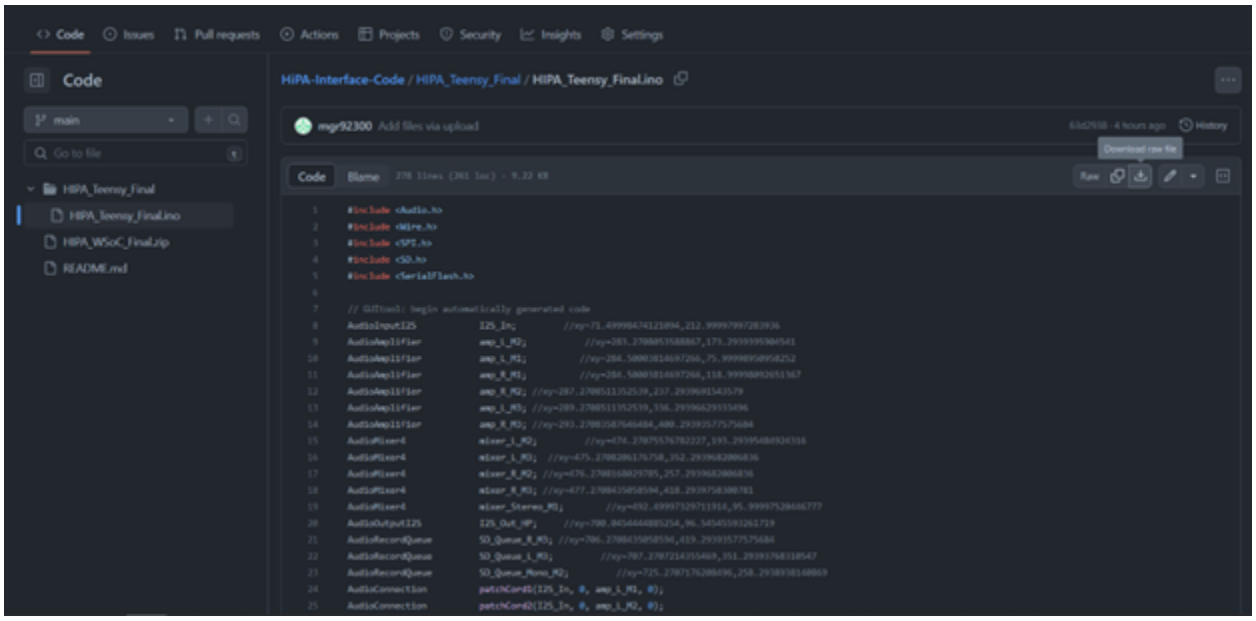
1. Once in the repository, open the HIPA_Teensy_Final folder



2. Open the HIPA_Teensy_Final.ino file

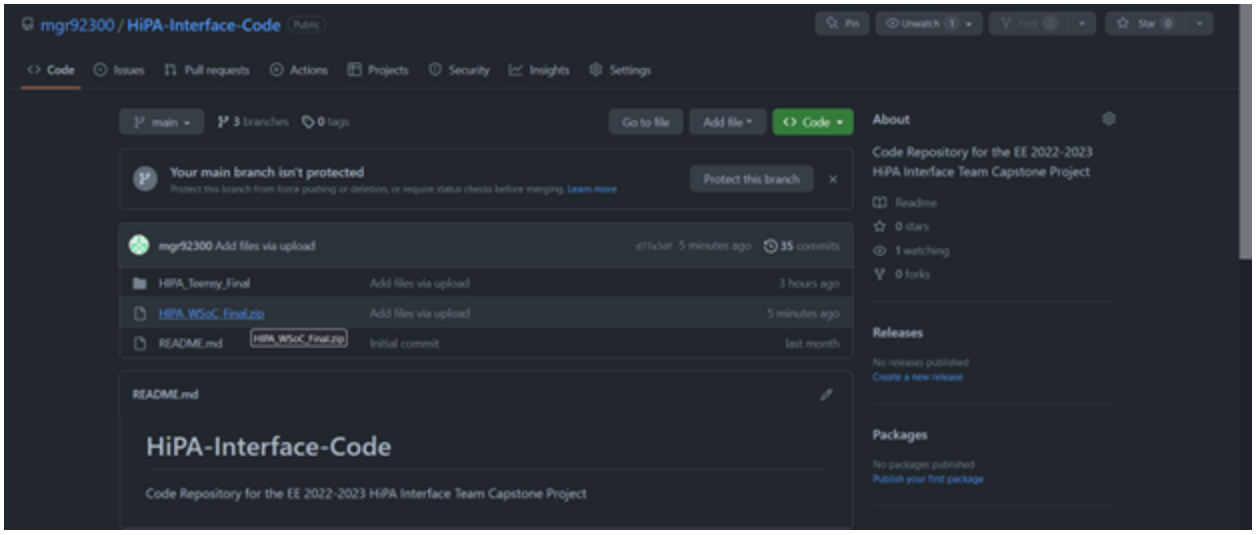


3. Click the download button on the top right corner of the window

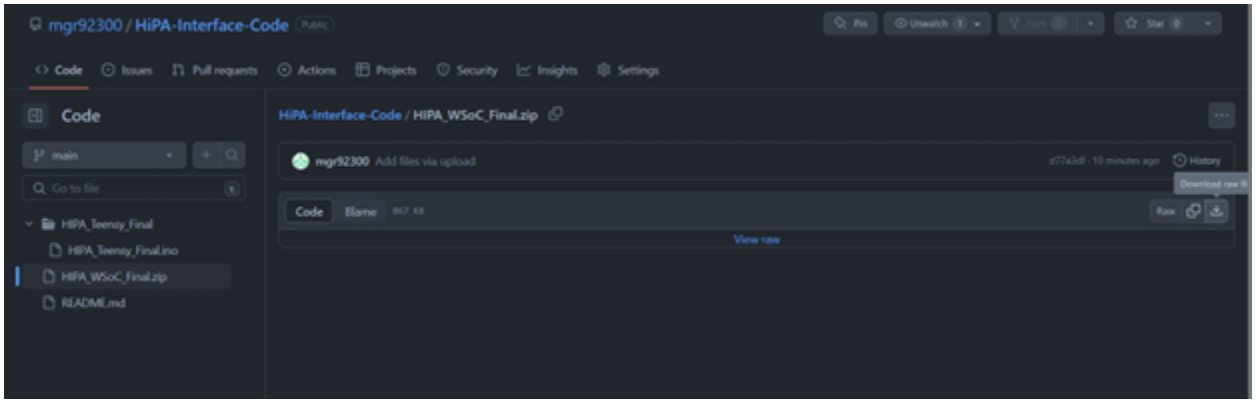


WSoC board Download

1. Once in the repository, open the HIPA_WSoC_Final.zip file

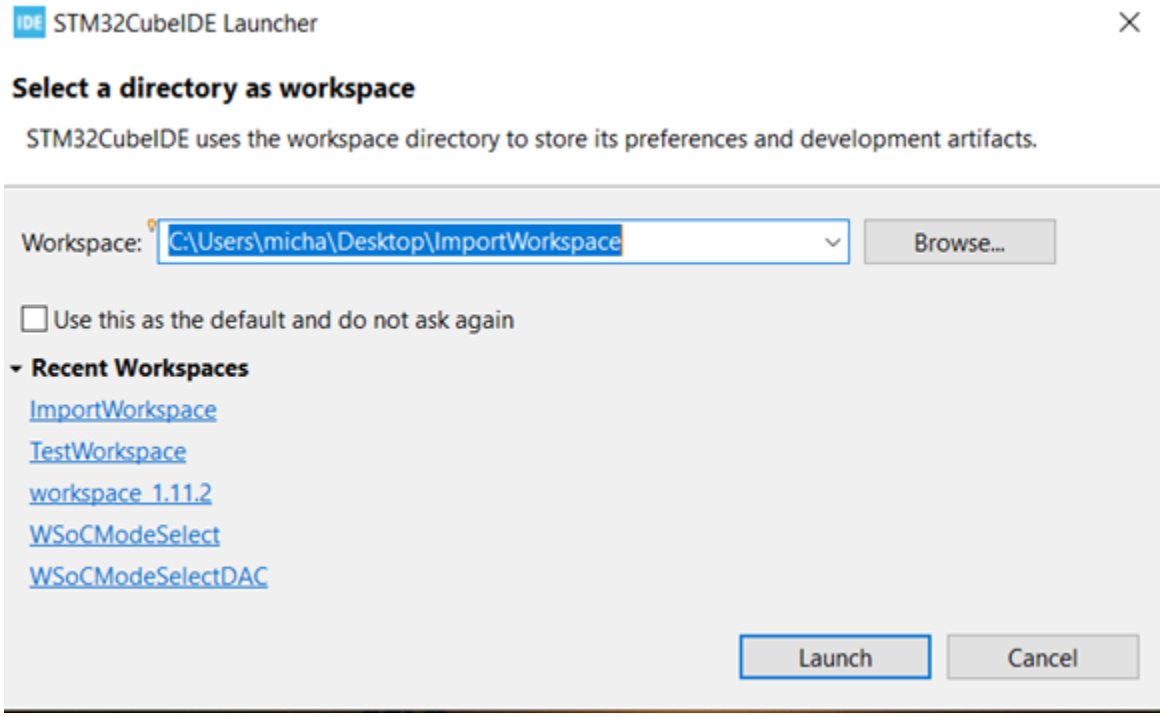


2. Click the download button on the top right corner of the window



Appendix E - Importing the WSoC Project

1. Open the STM32Cube IDE executable
2. Create a workspace as the default file directory then click Launch
    a. This workspace is a file directory that can host a multitude of projects from STM32 specific project files, C files, etc.

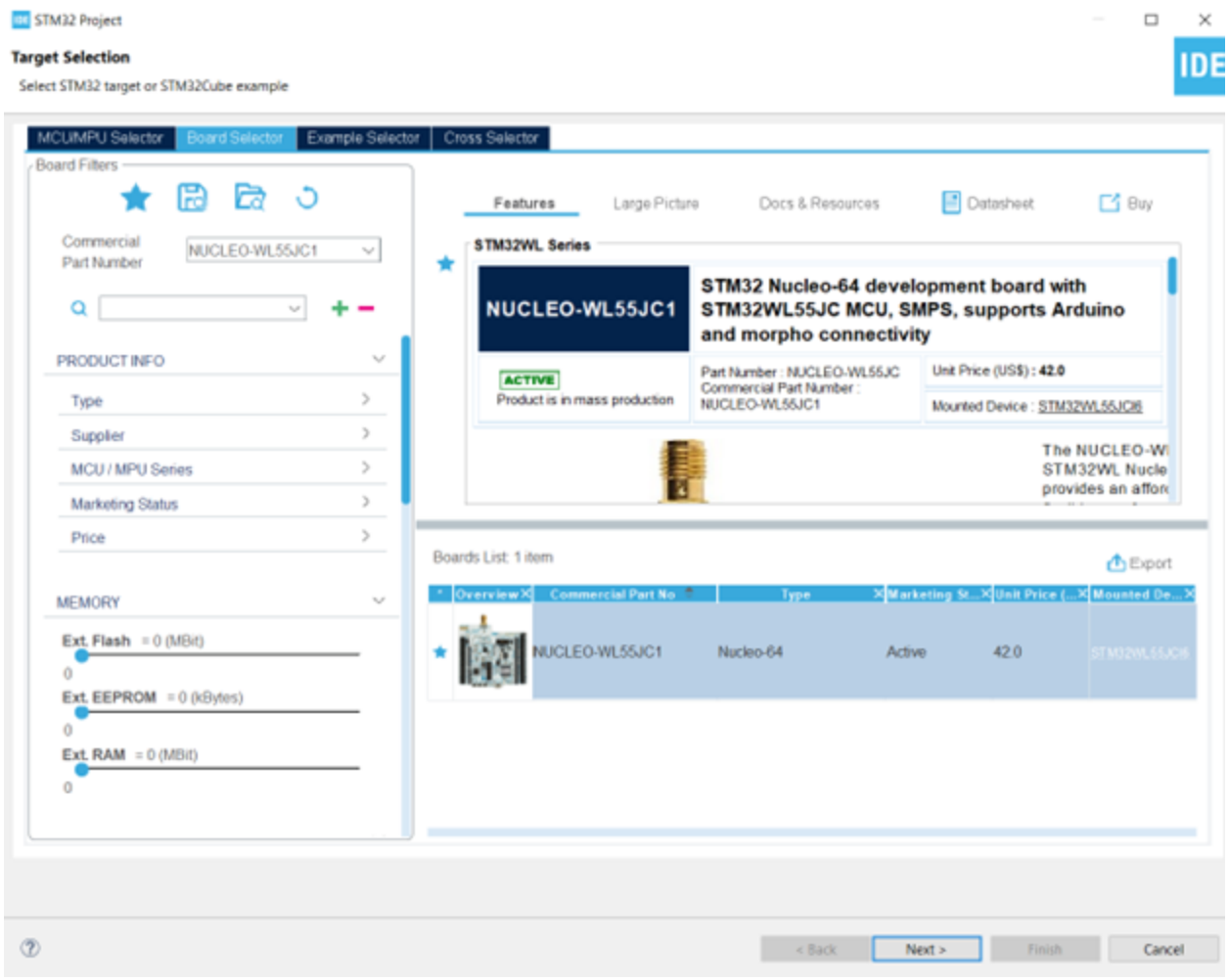


3. Close the Information Center to get to the main IDE interface

4. Follow the tabs on the toolbar -> File > New > STM32 Project

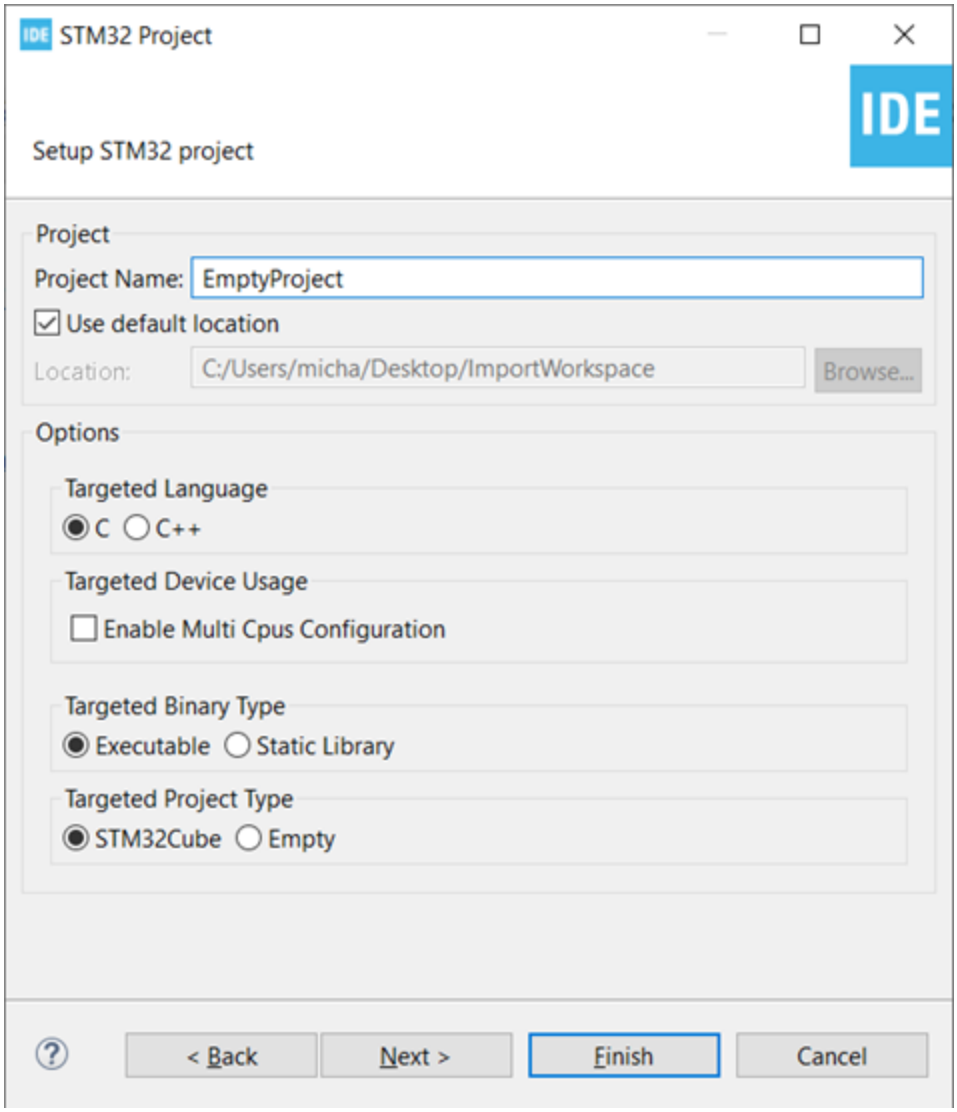
5. Select the target board in the board selector header

    a. For this Project it is the Nucleo-STM32WL55JC1 board



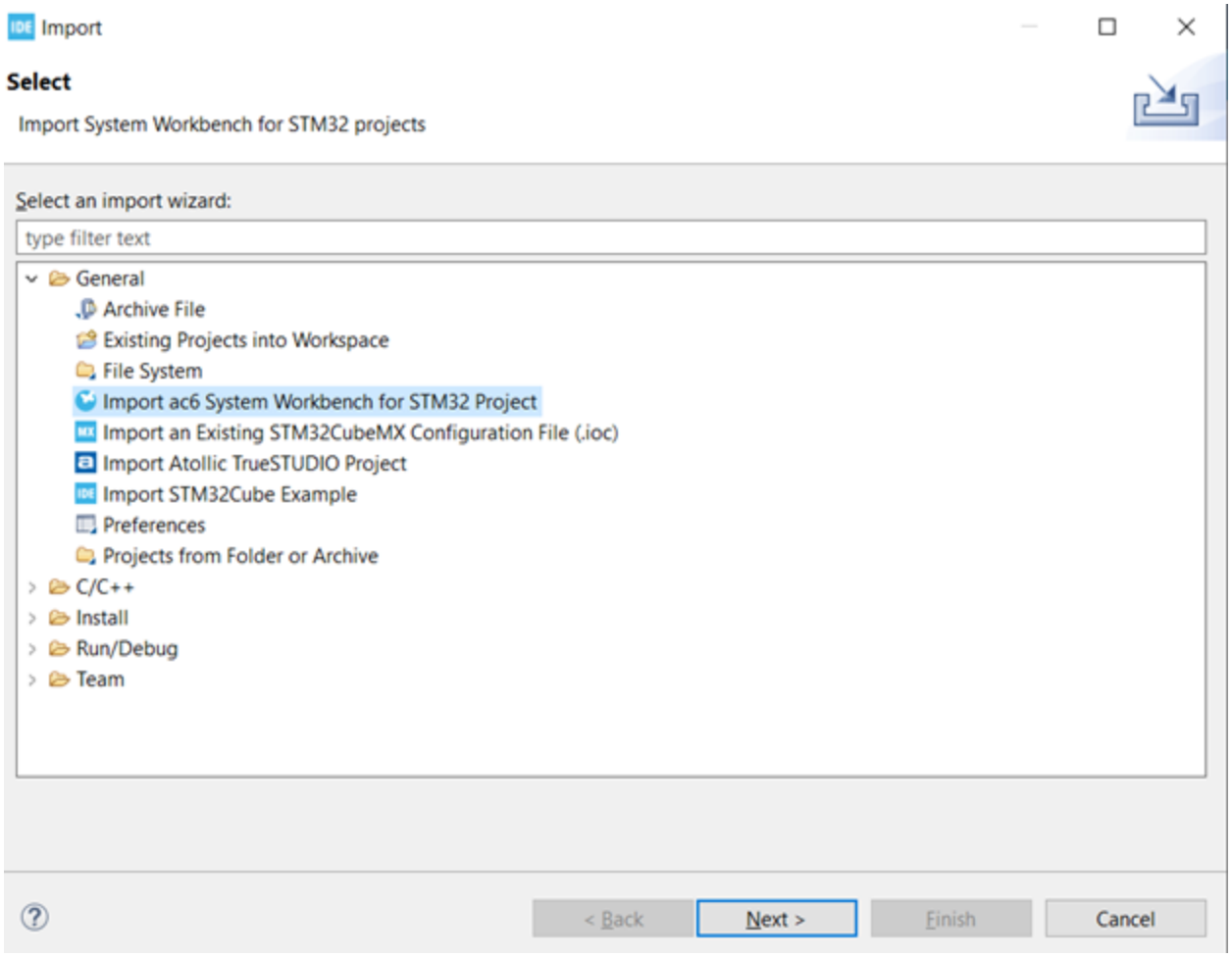
6. Create an empty project with whatever file name and hit finish

   a. The following options should be selected



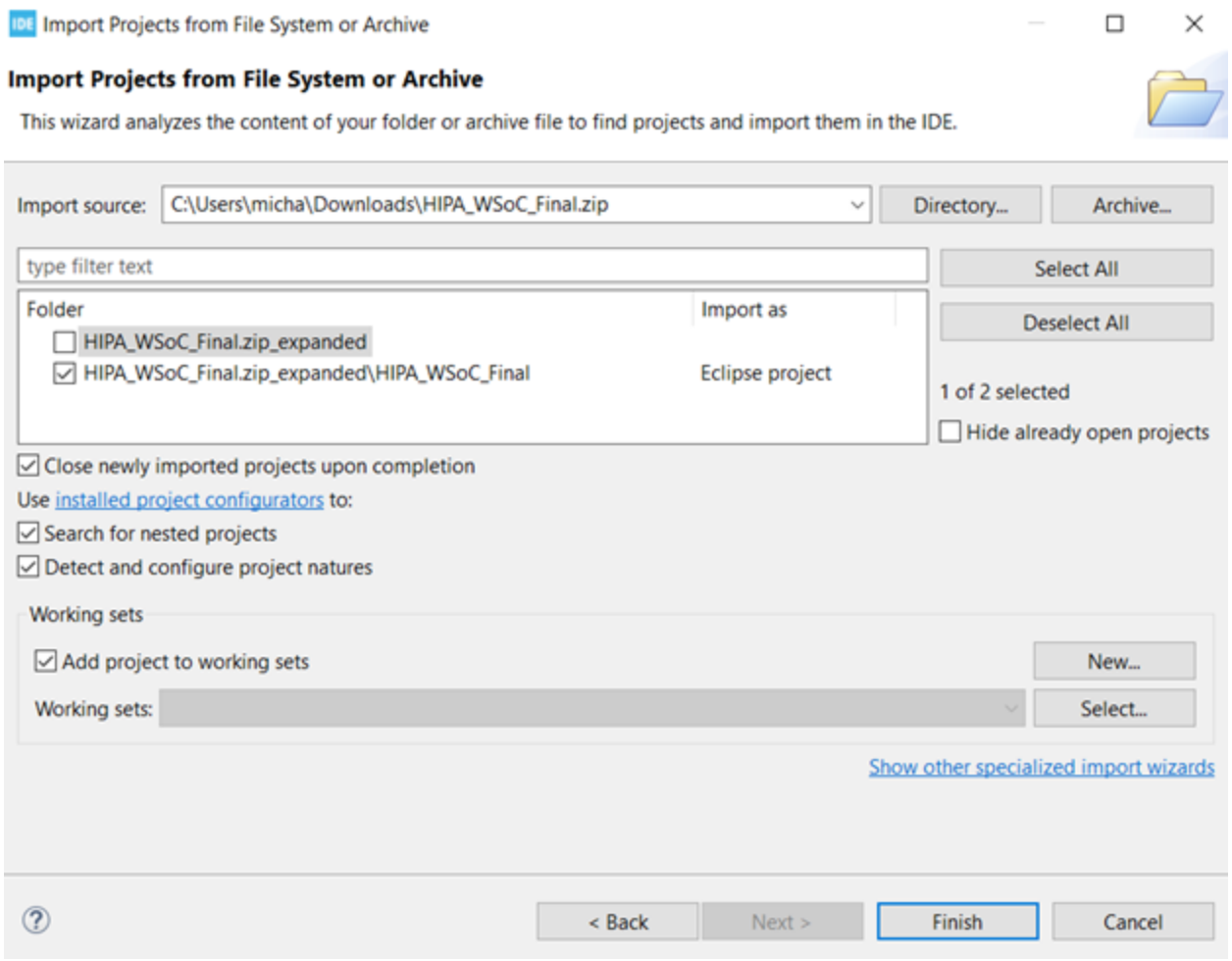
7. Follow the Tabs on the toolbar -> File > Import

8. Select the folder General > Import ac6 System Workbench for STM32 Project

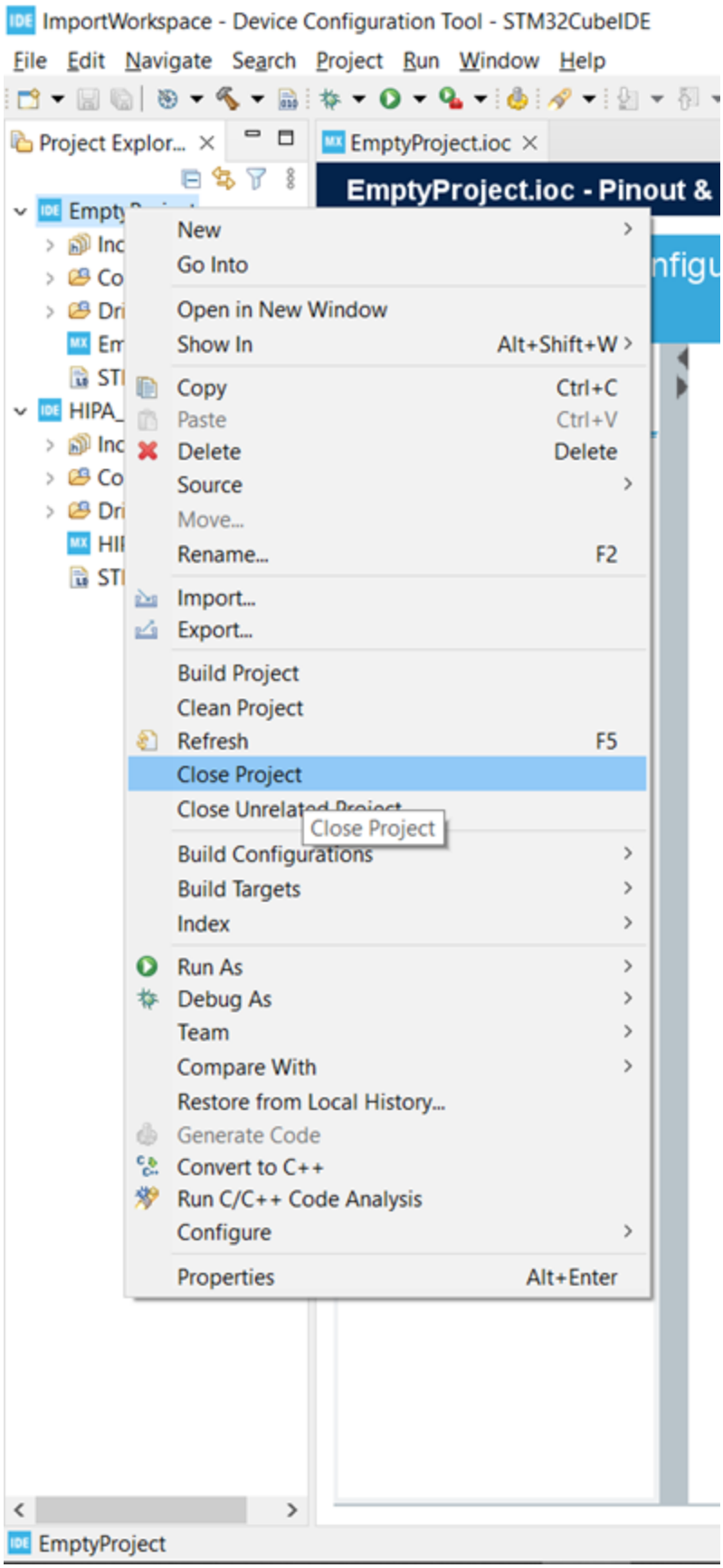


9. Select the Archive Button and select the directory where the downloaded zip file from Github is with respect to the WSoC as indicated in Appendix W

10. Select the options as seen in figure X and Click Finish



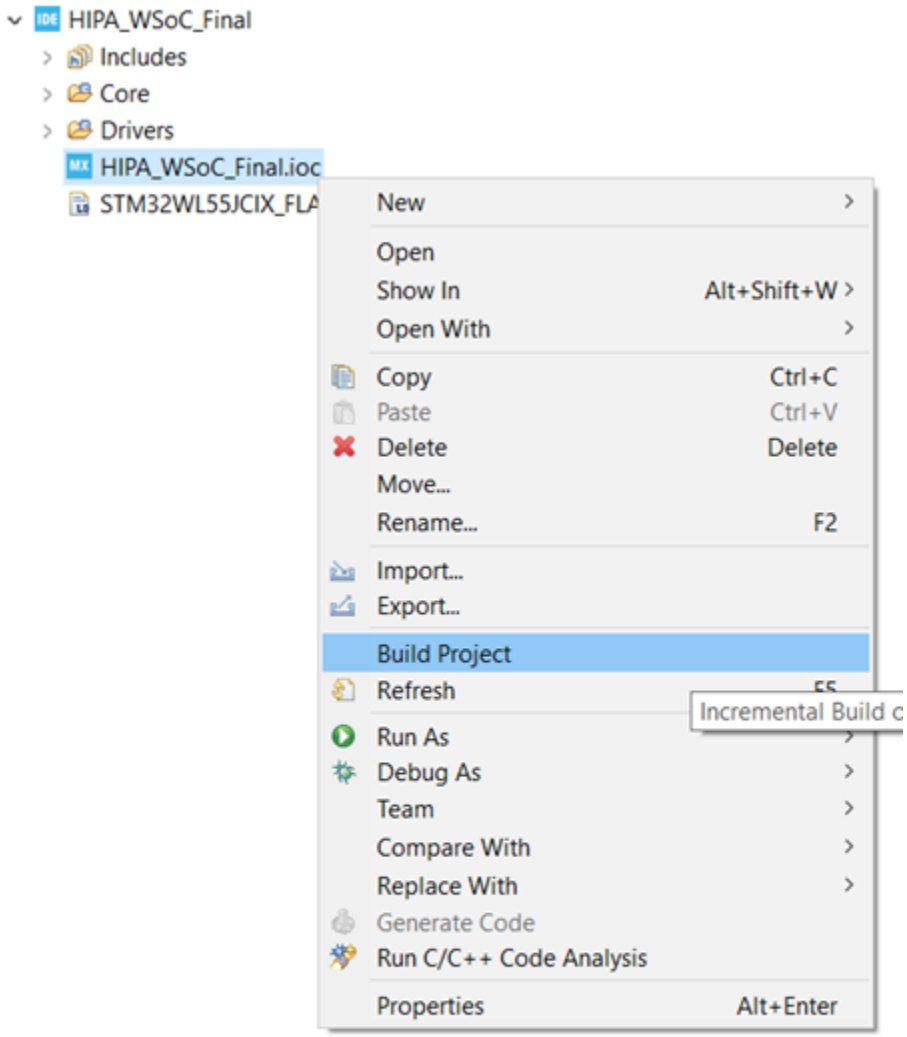
11. Close the empty project that was originally created


ImportWorkspace - Device Configuration Tool - STM32CubeIDE
File Edit Navigate Search Project Run Window Help
Project Explor...
EmptyProject.ioc
EmptyProject.ioc - Pinout &
New
Go Into
Open in New Window
Show In Alt+Shift+W
Copy Ctrl+C
Paste Ctrl+V
Delete Delete
Source
Move...
Rename... F2
Import...
Export...
Build Project
Clean Project
Refresh F5
Close Project
Close Project
Build Configurations
Build Targets
Index
Run As
Debug As
Team
Compare With
Restore from Local History...
Generate Code
Convert to C++
Run C/C++ Code Analysis
Configure
Properties Alt+Enter
EmptyProject

12. Double Click the HIPA_WSoC_Final Project on the Project Explorer window in the STM32Cube IDE and drop down the menu

13. Open the File Project File and Right click on the HIPA_WSoC_Final .ioc File

14. From the drop down menu, click on the build project option



This will build an executable that can be programmed onto the WSoC