Le Mans Université

SOL

Multi Audio Effect Processor

By
Guillermo Aramburo
Harishkumar Sivacoumar

September 25, 2022

Supervisors:
Antonin Novak
Bruno Gazengel
Manuel Melon

Acknowledgments

This project was achieved thanks to all the help we got from the different staff members and technicians that work in the university. At multiple times we were stuck with the implementation of the audio processor. Oliver Munroe and specially Antonin Novak were of great help to overcome any difficulties during the development of the multi-effect processor. Thanks to Bruno Gazengel and Manuel Melon for supervising the project and guiding us towards finding research regarding audio effects.

For the research and development of a plate reverb through physical modeling, we would like to thank Frédéric Ablitzer for showing us how to model a plate reverb through its mode shapes.

We would like to thank Alann Renault for helping us make our vector design a reality by engraving it by laser cutting to our enclosure.

Also, for the design and prototype testing of PCBs and external ADCs, Pierre Aranud Lecomte and Stephane Letourneur were extremely helpful by letting us work in the electronics lab and making sure our PCB designs were developed.

And finally, a special thanks to Jacky Maroudaye and his colleagues which include Eric Egon, Manuel Corvaisier, Lionel Guilemeau, and Hervé Mézière from the mechanical workshop who helped us with making the holes and knobs for the enclosure and once again Antonin Novak for always helping us debug and come up with better solutions for the code.

Le Mans Université

Contents

1 Introduction **1**

2 Hardware **2**
2.1 Electronic Hardware . . . 2
2.1.1 Audio Hardware . . . 2
2.1.2 User Interface . . . 3
2.2 Enclosure . . . 4

3 Software **6**
3.1 Audio Processor . . . 6
3.1.1 Main Page . . . 6
3.1.2 Custom Audio Processor . . . 9
3.1.3 Custom DSP Class . . . 9
3.2 Algorithms . . . 10
3.2.1 Plate Reverb . . . 10
3.2.2 Tremolo . . . 14
3.2.3 Bitcrusher . . . 15
3.2.4 Vibrato . . . 15
3.2.5 Delay . . . 16
3.3 User Interface . . . 16
3.3.1 Parameters . . . 16
3.3.2 Menu . . . 17
3.3.3 MIDI . . . 17

4 Measurements **18**

5 Conclusion **19**
5.1 Hardware . . . 19
5.2 Software . . . 20

6 Bibliography **21**

Appendix **22**
A Github Repository . . . 22
B Stereo Audio Processor . . . 22
C Audio Processor Header . . . 23
D Audio Processor CPP . . . 25
E D Audio Helper . . . 27
F ME_DSP Header . . . 27
G ME_DSP CPP . . . 29
H Tremolo . . . 35
I LFO . . . 36
J Bitcrusher . . . 37
K Schematic for drilling holes with CNC . . . 37
L Schematic for the main knob . . . 38

M User Interface Code 38
N Plate Reverb Implemented in Python 45
O Plate Reverb for Teensy 50

1 Introduction

An effect unit is an electronic device that alters the sound of an audio source through audio signal processing. They are called effect units because they apply audio effects. An audio effect, is signal processing focused for audio and mainly for artistic purposes. Effect units can be analog or digital as well as a combination of both. A multi effect unit refers to a unit capable to apply more than one audio effect either in parallel or series. Multi effect units are usually digital as the implementation through analog means is costly, more prone to noise and impractical compared to a digital implementation. For this project, the main goal is to implement an audio effect known as reverb. More specifically, a digital reverb capable of imitating the sound of a plate reverb, inside a multi-effect unit.

Reverberation is the summation of a direct sound and reflections of it. This audio effect can be usually heard in a room where the surfaces absorb and reflect the sound. These reflections are not perceived as discrete echoes as the delay between the direct sound and the reflections are shorter than 30 - 40 milliseconds which result in the psycho-acoustical effect of smearing of the sound, known as the Haas Effect. As the reflections travel and bounce from the different surfaces, the energy slowly decays due to the absorption of these surfaces which usually leads to an exponential decay in a pleasing sounding room.

This audio effect helps the listener have a sense of space and enhance the listening experience, making it an important factor to consider when designing a room for listening purposes. Prior to any technology, the characteristic reverberation of a space was set by its materials and geometry. The first alternative to obtain reverb by other means than a room was the plate reverb. The plate reverb consists of a metal sheet suspended (as to simulate the isolation from any damping outside the material) in which sound is fed through a directly coupled speaker and received by a coil pickup or contact microphone. Any sound fed to it would make the plate vibrate and most of the sound reflects inside the plate until it's dissipated. These reflections lead to a sound similar to a reverb yet with a few audible differences due to factors such as the plate being two dimensional, the speed of sound being faster and metal being a dispersive medium unlike air.

To choose or create an algorithm for the digital plate reverb, different methods will be considered such as physical modelling and classical reverb algorithms to simulate the desired effect. Furthermore, as the algorithm will be implemented in a multi-effect unit hosted inside a real-time embedded system, other simple audio effects such as tremolo and bitcrusher will be also implemented and detailed. Finally, the different methods will be carried out and compared along with it's performance in the embedded system working as a multi audio effect processor.

2 Hardware

This section consists of the electronic and mechanical hardware. The electronic hardware explains the components and connections necessary for the audio and user interface while the mechanical hardware explains the building and design of the enclosure and knobs.

2.1 Electronic Hardware

The main elements and connections of the electronic hardware are shown in figure 1.

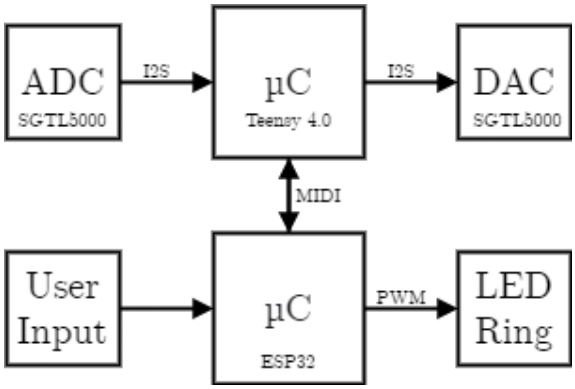


Figure 1: Overall schematic of electronics

2.1.1 Audio Hardware

The audio hardware consists of a micro-controller which exclusively takes care of the audio and only interprets the interrupts of the user input once they are fully processed. This approach was taken to ensure that the micro-controller's performance is focused towards the audio and not lost due to any processing required outside of it. The micro-controller chosen was a Teensy 4.0 as it has a native audio library that works for CD audio quality (16 bit, 44.1 KHz) and with this library, external ADC's and DAC's can be coupled through I2S avoiding the limitations of the internal converters of the micro-controller such as low bit resolution (12 bit).

For the audio conversion, the Teensy 4.0 Audio Shield was used which has a 16-bit resolution ADC and DAC which work through I2S. I2S is a communication protocol specifically made for audio which is very similar to I2C. It requires four signal connections: DATA (DIN or DOUT), Bit-Clock (BCLK), Word-Select (WS or LRCLK) and Signal Clock (SCLK). The data signal transmits the Pulse-Code Modulation (PCM) audio data, the bit clock transmits a pulse signal at 2 x f_s x n where f_s is the sample rate and n the bit resolution and the 2 is due to having two channels for stereo audio. The word-select pin is a pulse signal at the speed of the sample rate to indicate which channel's data is being transmitted. Finally, the signal clock is a pulse signal which operates at the required speed for all the previous signals to be acquired and processed, the frequency must be at a 2^n multiplication

of the sample rate, typically 256.

Effect units have certain standards. These standards are:

- The power supply voltage is generally 9V, sometimes 12V.
- The positive tip of the connector is reference and the sleeve is the positive voltage.
- The DC Jack allows for a battery to power the system if nothing is connected to it through an internal switch.
- There is no power switch, internal switches of the audio jack connectors are used to short the DC connector's reference with the ground of the effect unit. Therefore, the unit powers when an audio jack is connected to it.
- A three pole double throw switch is used as bypass so the effect unit is completely bypassed when the switch is active.

From these standards, only the first two were followed. Due to the system having analog and digital signals, grounding the digital signals directly to the audio jack was avoided as it could lead to noise. Furthermore, there was no time for testing lithium batteries and to power the system or to set an analog path for the master mix and volume control so the bypass switch was not considered.

Printed Circuit Boards (PCBs) were designed for the project. However, by the time the PCBs were done the manufacturing time wouldn't align with the due date for the project. Therefore, perfboards were used to assemble the circuit. A linear regulator of $5V$ was placed next to the Teensy and this same voltage line will be used to feed the circuit for the user interface explained in section 2.1.2. Quarter-inch connectors also known as female audio jacks were connected to the Teensy Audio Shield to feed and extract the processed audio. The UART connection also goes to the user interface circuit and its purpose is explained in 2.1.2.

2.1.2 User Interface

The hardware of the user interface consists of two potentiometers, touch sensors and three rotary encoders with built in switches for user inputs. For user feedback a ring of 24 RGB LEDs is used as shown in figure 1. All these components, are connected and processed by an ESP32 Pico-Kit v4 development board. The rotary encoders were implemented using the schematic recommended by the manufacturer in the datasheet and the encoder library created by Paul Stoffregen. The built in switches were configured with pull down resistors and conditioned through software to clean any noise by setting a delay between the reading of states. The touch sensors are inherent to the development board, which only require a single cable to be connected to each of the encoders' chassis. Finally the potentiometers don't required any hardware conditioning and the RGB LEDs were controlled with the FastLED audio library which changes each LED's color through a single cable with data sent as Pulse-Width-Modulation (PWM).

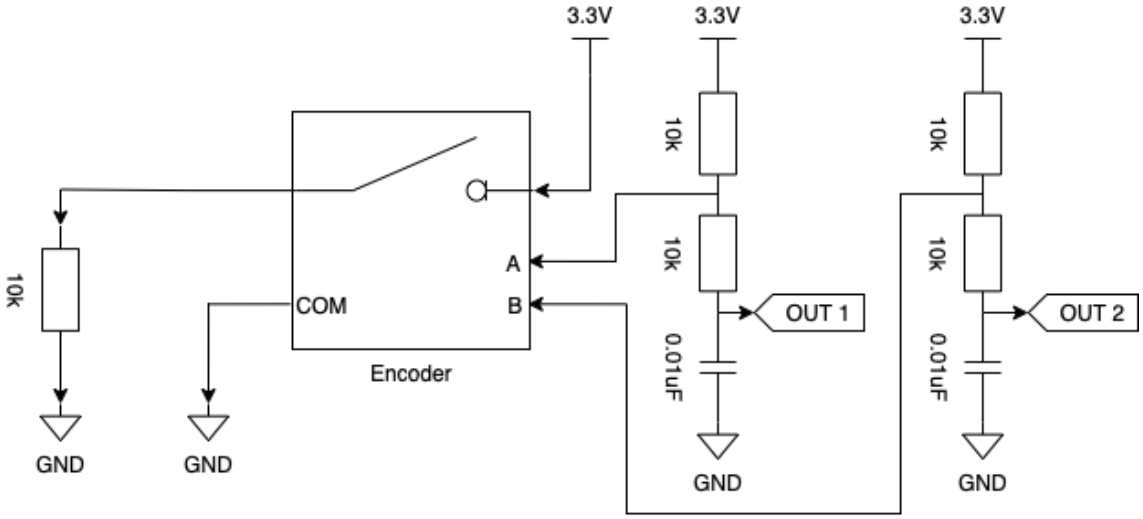


Figure 2: Encoder and built in switch schematic.

The ESP32 communicates to the Teensy 4.0 through the MIDI protocol. It's an asynchronous UART-based protocol made for electronic music instruments. The data given by the touch sensors and switches is interpreted solely by the ESP32. The communication between the Teensy and ESP32 is explained in section 3.3.3.

2.2 Enclosure

The design aesthetics for this project is inspired from Brüel & Kjær products as shown in figure 3 in terms of colors, layout, font for texts, knobs, etc.

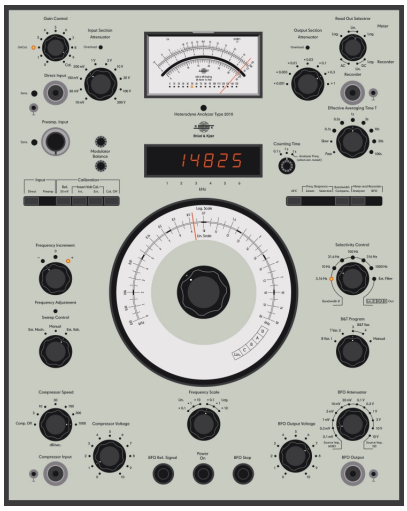


Figure 3: Brüel & Kjær Heterodyne Analyzer.
Drawn by Nenad Milosevic

The enclosure is the 1590DD manufactured by Hammond Manufacturing with the dimensions of 188mm x 120mm at the base. This particular enclosure was chosen as Hammond enclosures are a standard among guitar pedal hobbyists given their

enclosures are able to house and shield from radio frequencies all the electronic hardware while remaining compact and lightweight.

First, the design for the layout was done in order to finalise and verify the placement of the knobs, input, output ports and the DC input in order to provide an accessible user interface. This design (shown in figure 4), including the logo, were made using Adobe Illustrator. Situated at the center, is the main rotary encoder which will be used to select the effects surrounded by the LED ring. The two potentiometers at the top for controlling the volume and mix while the other two encoders are at the bottom which will help the user change parameters of a given effect. The input and output jacks are located at the top right corner.

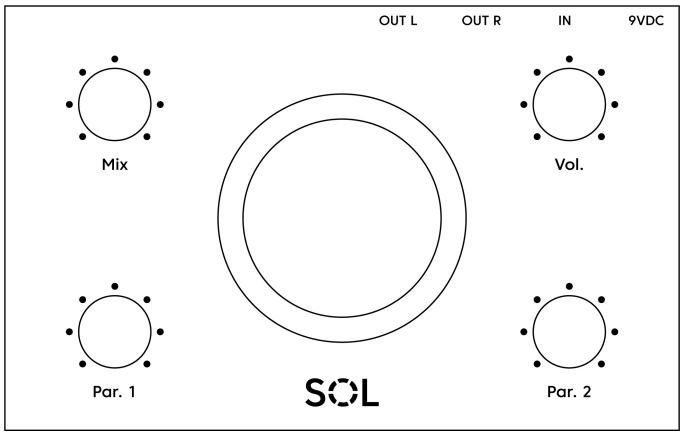
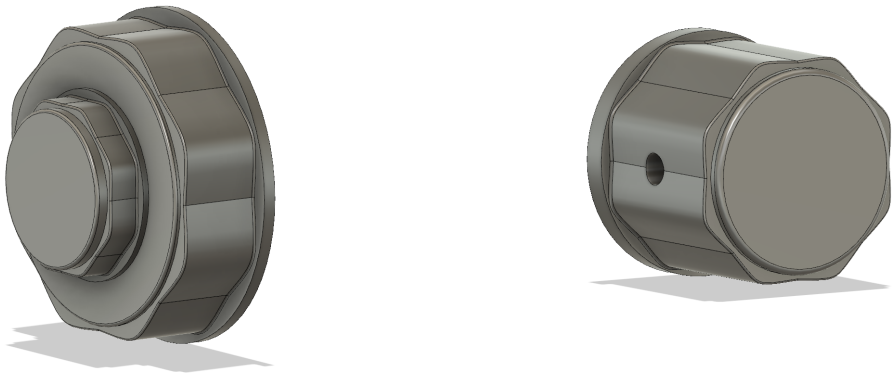


Figure 4: Design layout

With the layout completed, the next part is to create a schematic for the positions of the holes to be made for the components. A 3D model of the enclosure provided by the manufacturer was used in Fusion 360 software to create the schematic. Considering that the enclosure is not a perfect rectangular cuboid and the edges being curved, the best option to set the coordinates for the holes on the front face was to set the origin point at the center. The diameter for the holes corresponding to the encoders and potentiometers are set at 6.25mm and 11mm for the audio jacks. To hold the LED ring in place, an engraving with a depth of 1mm was done with the dimensions of the inner and outer diameter of the ring. The final schematic with the right dimensions (as shown in appendix L) was then sent to the technicians at the university who were able to drill the holes with the CNC machine.

The next part of adapting the Brüel & Kjær model is to design and build the knobs using the CNC machines. The knobs are made with aluminum in order to be conducting for the touch sensors. Two different knobs are required for the enclosure, a main knob for the encoder in the center and a secondary knob for the potentiometers and other encoders. The 3D design for the knobs was done in Fusion 360 CAD software. The main knob is designed in such a way that it looks like two knobs placed on each other as seen in figure 5a. The outer diameter of 52.2mm (the base) as it should be smaller than the inner diameter of the LED ring which is at

52.3mm. There are 8 curves on the side on the knob which represents the 8 possible effect selections. The knob holds on to the shaft of the components through a small M4 screw at the side of the knob. The secondary knob is a simplified version of the main one. The dimensions of both the knobs can be found in the appendix (ref).



(a) Main knob (b) Secondary knob

Figure 5: CAD design of the knobs

As for the paint, the Venom 6000 by Montana paint (figure 6) was chosen as it closely resembles the green on Brüel & Kjær products. First a metal primer spray is used as a pre-treatment which then helps the paint attach better to the surface. After coats of primer and then the green paint, a matt varnish is sprayed on the surface which serves as a protective coating over the paint.

Figure 6: Color used on enclosure

3 Software

3.1 Audio Processor

3.1.1 Main Page

As mentioned in the section 2.1.1, the micro-controller's audio library was used. By default, this library has defined processing blocks which can perform different types of operations such as Fast Fourier Transform (FFT) or even audio effects such as delay and reverb to name a few examples. Rather than using these blocks, a

custom processor was made. A custom processor allows to apply more algorithms which might not be included in the audio library such as the plate reverb algorithm. Furthermore, with this custom processor it's easier to have a control over when the interrupts due to parameters changing are being done and the algorithms are made to be able to work while having interrupts which might not be the case for all the algorithms predefined in the audio library.

The main page of the audio software takes care of initializing all the libraries, connections and variables necessary for the audio processing. First the libraries required for the Audio, MIDI and custom processor are declared

```
#include <Audio.h>
#include <SPI.h>
#include <Wire.h>
#include <SerialFlash.h>
#include "audio_processor.h" // Include custom audio processor
#include <MIDI.h>
```

Then, all objects and variables required are created. The SGTL5000 object corresponds to the codec inside the Teensy Audio Shield. The user interface variables are required to keep track of the current effect slot, current audio effect and normalized parameter value.

```
// Audio Objects
AudioControlSGTL5000 sgtl5000_1;
AudioInputI2S usb1;
AudioOutputI2S i2s1;
AudioProcessor myProcessor;

MIDI_CREATE_INSTANCE(HardwareSerial, Serial1, MIDI); // Create MIDI
    communication

// UI
uint8_t pos = 0; // Current effect slot
uint8_t fx = 0; // Current effect
float step;
```

Afterwards, the audio objects are connected.

```
// Connect ADC -> Processor -> DAC
AudioConnection patchCord1(usb1, 0, myProcessor, 0);
AudioConnection patchCord2(usb1, 1, myProcessor, 1);
AudioConnection patchCord3(myProcessor, 0, i2s1, 0);
AudioConnection patchCord4(myProcessor, 1, i2s1, 1);
```

Finally, the audio hardware and midi communication are enabled and the main loop consists of reading the data from user interface followed by calling the custom audio processor to perform the corresponding task. All code has been commented to ease reading comprehension. For this same reason, for the next sections most

code will only be placed in the appendix section and the logic behind the code will be explained rather than the lines of code. The full page of code can be found in appendix B or in the Github link provided in appendix A. The Github link is recommended as this project is constantly being improved.

```
void setup() {

  // --- audio
  AudioMemory(50);
    // Enable the audio shield, select input, and enable output
  sgtl5000_1.enable();
  sgtl5000_1.inputSelect(AUDIO_INPUT_LINEIN);
  sgtl5000_1.muteHeadphone();
  sgtl5000_1.unmuteLineout();
  sgtl5000_1.lineInLevel(0);
  sgtl5000_1.lineOutLevel(25);
  // --- serial
  MIDI.setHandleControlChange(processMIDI); // Function to call for midi data
  MIDI.begin(2); // MIDI channel to read data from
  myProcessor.changeEffect(0,1); // Set first effect slot to audio effect 1, which is
      Tremolo

}

void loop() {
  MIDI.read();
}

void processMIDI(byte channel, byte number, byte value)
{
    step = value * MIDI_TO_FLOAT; // Transform value to float number
        between 0 and 1
    switch(number)
    {
      case 0: // Change parameter 1
        myProcessor.changeParam(pos,number,step);
        break;
      case 1: // Change parameter 2
        myProcessor.changeParam(pos,number,step);
        break;
      case 2: // Change parameter 3
        myProcessor.changeParam(pos,number,step);
        break;
      case 3: // Change audio effect
        fx = value;
        myProcessor.changeEffect(pos,fx);
        break;
```

```
        case 4: // Change effect slot
          pos = value;
          break;
    }
}
```

3.1.2 Custom Audio Processor

As seen in section 3.1.1, the main purpose of the micro-controller is to call methods of the custom audio processor. This processor class inherits from the AudioStream class inside the Audio library. Inheriting from the AudioStream class allows to receive, process and output the audio taken from the Teensy Audio Shield. The audio library process the audio in bytes, because the audio samples are stored in 16-bit integers. These samples are inside a custom variable named "audio _block_t" which are arrays of 128 samples by default. The block size was reduced inside the inner workings of the audio library to 64 samples to improve latency.

The custom processor takes cares of processing every sample inside the audio block and setting interrupts for the MIDI data. In regards to audio processing, first, each sample is converted to a floating point number, then it's stored to a delay line. This delay line stores up to 1 second of audio and is used for audio effects such as vibrato and echo delay which don't require delay lines with internal feedback. This approach allows to create multiple instances of the aforementioned effects but it reduces the memory available for audio effects which require their own memory. After the sample is stored in the delay line, the sample is processed by 16 objects (8 per channel) of a custom class named "ME_DSP" which holds all the audio effect algorithms as private methods. Then, the sampled is denormalized, meaning that very small floating point numbers which may lead to miscalculations or noise are flushed to zero. Finally, the sample is converted back to an integer and when all the samples inside the audio block have been processed, these are streamed.

For the MIDI data, the processor sets an interrupt of the audio processing, calls the methods to change the parameter value or audio effect for the DSP object selected by the user interface and finally continues the audio processing. The code implementation for the header and cpp can be found in appendix C and D.

3.1.3 Custom DSP Class

For the custom ME_DSP class, all audio effects algorithms are implemented as private methods and these are explained in section 3.2. The ME_DSP class has four public methods (public means they can be called from outside the class). These public methods are:

- Initialize
- Process
- SetParam

- ParamUpdate

"Initialize" allocates the memory required for certain audio effects such as the plate reverb. This approach has to be taken because the volatile memory of the Teensy 4.0 is composed of two RAM of 512 KB. To store variables inside the second RAM, it is necessary to use functions which dynamically allocate the memory such as memset, malloc or calloc. Otherwise, all memory would be set only inside the first RAM which would lead to compilation errors, as the first RAM would be full. A better way of implementing the effect has been found, which is to declare each audio effect as a child class from the ME_DSP class and allow each effect to have a constructor and destructor. However, this approach has not been fully finished therefore only the finished code will be explained. For further information of the new method refer to the Github link.

"Process" is a switch statement which takes the current sample and applies the selected audio effect to the sample. The method "setParam" changes the parameter value. Every object of the ME_DSP class has memory for its own three parameters' values. "ParamUpdate" is a function to be called for audio effects which require the computation of coefficients dependent of the parameter value. This ensures the code to be more efficient as it keeps these values from being calculated for every sample. The header and cpp of ME _DSP can be found in appendix F and G.

3.2 Algorithms

All the DSP algorithms were implemented as private methods inside the ME_DSP class and to be modifiable by only three parameters in which the first parameter is always a mix control of the processed and unprocessed signal. Therefore, only the second and third parameters function will be detailed in the following explanations.

3.2.1 Plate Reverb

The plate reverb is the main effect of this project and also the most complex effect implemented for this project. The sound of a plate reverb can be digitally emulated through different means. The main methods would be physical modeling, convolution and filter algorithms. Physical modeling consists of obtaining a set of equations which are able to calculate the modes of the plate depending on it's boundaries, size and listening position. Afterwards, these modes can be applied to an input through convolution or numerical approximation. Convolution doesn't require physical modeling, it just requires an impulse response recording of a plate to apply its sound to any other input. The biggest problem with the previously mentioned methods is that these methods are only reliable for fixed parameters. Physical modeling is very CPU-intensive for modern computers so implementing this method inside a multi-effect processor would not be possible. Furthermore, convolution requires recordings. Therefore, to emulate the change of damping or size, many sample recordings would be required which would not fit in the memory of the micro-controller. The best approach is by filter algorithms.

Approximating reverb though filters was first mentioned by Manfred Schroeder in his paper "Natural Sounding Artificial Reverberation". Reverberation is approximated by a network of all-pass filter and comb filters. Comb filters are filters which create peaks and dips in the frequency response which resemble a hair comb, hence their name. This filter is commonly seen in rooms due to their modes. Therefore, this effect is audible in reverb as it occurs due to interference between a sound and a delayed copy of this sound. To recreate this filter discretely, the algorithm shown in figure 8 is used.

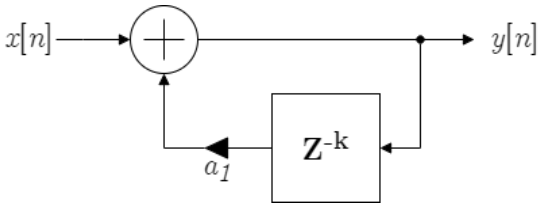


Figure 7: First order IIR comb filter.

Which corresponds to:

$$y[n] = x[n] + a_1 \cdot y[n-k]. \tag{1}$$

Where k is the delay in samples.

This filter consists of a first order IIR comb filter which is a single feedback network, it could also be a feed-forward network and it would lead to a similar result. The peaks and dips are distanced proportionally as the distance will be given by the delay which is equal to $k = \frac{n}{f_s}$, where n is the delay in samples and f_s the sample rate.

All-pass filters are filters that do not change the magnitude across frequency but only the phase, and consist of a feedback network along with a feed-forward network which can be seen as two comb filters joined together. This results in a sum of multiple delayed copies of the input. In other words, all-pass filters increase the amount of echoes, which is also a very fundamental aspect of reverberation. The higher the coefficient of an all-pass filter is, the higher the echo density will be. The algorithm used by Schroeder and this project is a first order IIR all-pass filter.

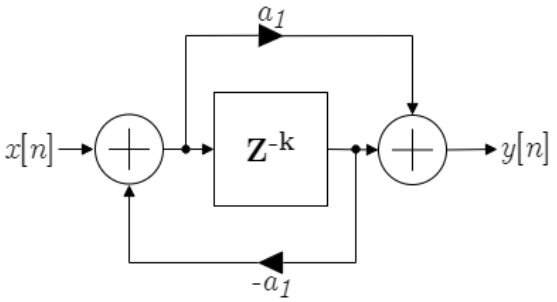


Figure 8: First order IIR all-pass filter.

Which corresponds to:

$$y[n] = x[n] \cdot a_1 + x[n-k] - a1 \cdot y[n-k] \tag{2}$$

Schroeder approximated reverb by three all-pass filters in series and four comb filters in parallel. By implementing the all-pass filters in series, the echo density increases quickly and each comb filters mimics the frequency response of a single reflection. The output of these filters is stored inside a buffer. The buffer is accessed at different points by prime numbers to minimize mutual periodicity and finally summed to set as the reverb output.

To achieve a reliable sound, research was conducted about known plate reverbs such as the EMT140. The main features of a plate reverb are:

- Instant high echo density, due to metal being a dispersive medium and sound traveling faster in metal, a high amount of reflections are perceived instantly.
- Frequency dependent decay time, as the frequency increases the decay of the plate decreases, above 10 Khz is less than one second.
- Constant resonance density over frequency. Which as shown, this is naturally achieved with IIR comb filters.

To achieve high echo density multiple all-pass filters in series are required with coefficients of high values (between 0.7 and 1). To add frequency dependence to the decay, first order low pass filters are required so high frequencies are attenuated before being fed back as a reflection. For these reasons, the plate reverb algorithm designed by Jon Datorro was implemented. Datorro's algorithm contained all the previously mentioned features of a plate reverb. Also, it's an algorithm which it's not processing heavy. Therefore, optimal for a multi-effect processor. The algorithm is shown in figure 9.

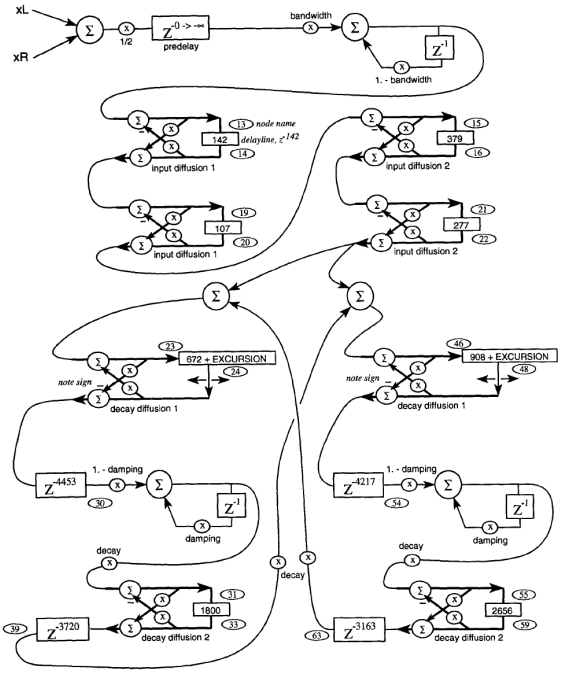


Figure 9: Jon Datorro's plate reverb algorithm.
Diagram taken from Jon Datorro's "Effect Design, Part 1: Reverberator and Other Filters" paper.

To properly implement Datorro's algorithm, most delays and coefficients need to be adapted to the sample rate of the Teensy 4.0, as this algorithm is set for a sample rate of $29761Hz$. To adapt the delays of all-pass filters and delay lines, the delay was converted from samples to seconds and then to samples for a sample rate of $44.1kHz$ in the following manner:

$$n = \frac{m \cdot 44.1}{29.761}. \tag{3}$$

Where n is the unrounded amount of samples for the $44.1kHz$ sample rate equivalent and m the original delay in samples.

For the low-pass filters, the algorithm corresponds to a first order IIR filter. Although the implementation of the bandwidth filter is different from the damping filter, both are the same type of filter just the coefficient adapts to give the same result. For the first filter network, the cutoff frequency corresponds to:

$$\alpha = 1 - e^{\frac{-2\pi f_c}{f_s}}. \tag{4}$$

Where α is the coefficient, f_c the cutoff frequency and f_s the sample rate. For the second filter network the cutoff frequency corresponds to:

$$\alpha = e^{\frac{-2\pi f_c}{f_s}}. \tag{5}$$

Lastly, the decay coefficient and memory size used in the original algorithm was chosen to be set by parameters. This allows for the RT60 of the reverb to change

dynamically and for the "size" of the plate to be modifiable. Changing the delay lengths leads to the perception of a smaller plate as the reflections occur and decay faster.
The reverb was implemented in the Teensy 4.0 and in python, in order to obtain the impulse response shown in figure 10. The implementation can be found in appendix N for python and appendix O for Teensy.

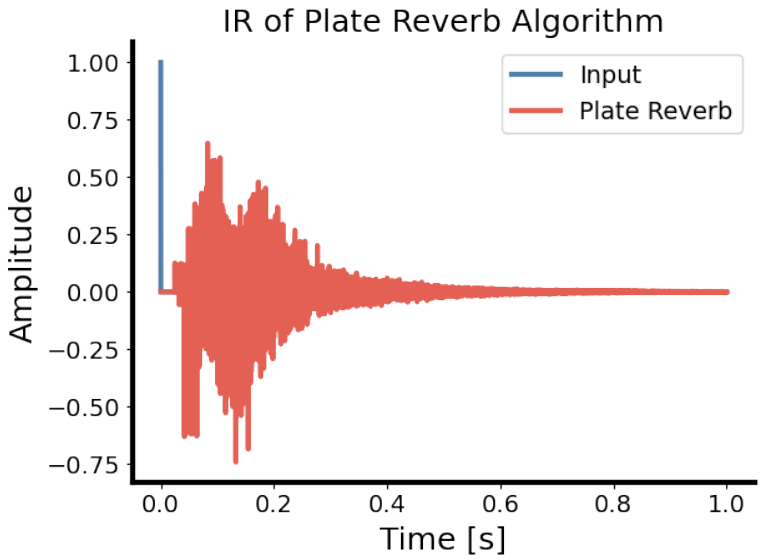


Figure 10: Jon Datorro's plate reverb impulse response.

For figure 10, the decay was set to 0.2, the delay lines were kept the same as the paper and the cutoff frequency of the IIR filters was set to 10kHz. Furthermore, the output was amplified by a gain of 10 just to show more clearly the exponential decay of the envelope of the reverb's impulse response. The small delay between the input and output is due to the reverb output being a sum of different delay taps. At the beginning all delay lines are filled with zeros so until the signal starts fillings these delay lines there will be no output. The python and c++ code for the algorithm can be found in appendix O and N.

3.2.2 Tremolo

Tremolo consists of an amplitude modulation of the input signal with a low frequency oscillator (LFO) which has an added offset to ensure it's values are only positive. The second parameter modifies the frequency of the LFO and the third parameter modifies the wave shape, which can be changed between sine, triangle, saw and square. The block diagram is shown in figure 11 and the implemented code in appendix H.

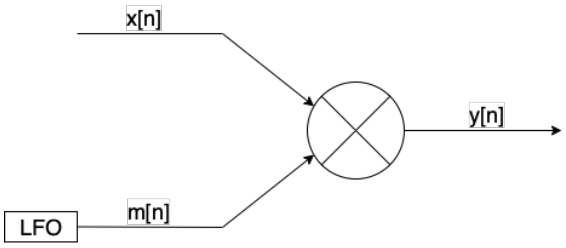


Figure 11: Tremolo

3.2.3 Bitcrusher

Bitcrusher is an audio effect that produces a type of distortion to the output. This effect can be achieved by using 2 methods. The first one being sample rate reduction and the second, resolution reduction. For this project, only the resolution reduction will be used. The amplitude resolution for digital signals is directly based on the number of bits used per sample. Reducing this resolutions means that the amplitude levels are being quantized to certain numerical values. This makes it so that by reducing the number of bits, the number of possible amplitude values is also reduced.

To build and demonstrate the bitcrushing method, first a 200Hz sine signal is used with the amplitude varying from -1 to 1. In Python and C++, there exists Numpy and Math libraries with a function called "ceil" which rounds up a number to it's ceiling value, in other words the closest integer greater than or equal to a given number. For example, the ceil of 0.2 is 1 and the ceil of -0.2 is 0.

To be able to round up multiple values between 0 and 1, the signal needs to be stretched out by multiplying it by the number of possible values desired (mentioned as mod in the code) and the relationship between the number of possible positive and negative amplitude values and the number of bits is $mod = 2^{nBits-1}$. The output is then divided by this value to bring it back to its original amplitude range. The output is also multiplied by a factor of 0.7 to avoid clipping. This method of bitcrushing is presented in appendix J.

3.2.4 Vibrato

Vibrato is the audio effect of adding small, quasi-periodic variations in the pitch of a tone. The figure 12 shows the block diagram of the vibrato. There are many ways of accomplishing this effect but the one implemented consists of having a time-varying delay line. This would be similar to reproducing a sound through a tape player while increasing and slowing the tape speed. When the tape speed increases so does the pitch of the recording.

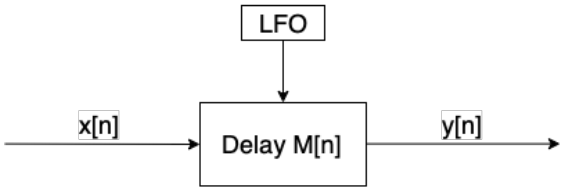


Figure 12: Vibrato

3.2.5 Delay

Delay, also known as echo, is the audio effect of repeating the sound source after a delay above the aforementioned Haas effect range. This effect emulates a sound source between two parallel boundaries far apart enough to hear the reflections individually until the sound is completely absorbed. This effect consists of a delay line with feedback. The feedback is multiplied by a number less or equal to one to ensure the effect is stable. Furthermore, clipping is added to ensure the signal can never exceed an absolute value larger than one. The block diagram is represented in figure 13.

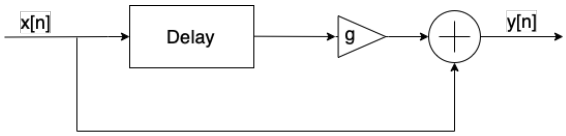


Figure 13: Delay

3.3 User Interface

Once again, rather than explaining the lines of code, the logic behind the code will be explained. First, the function of every parameter and afterwards the menu will be explained.

3.3.1 Parameters

- Mix: Controls the depth of the overall processed signal and original input signal,

$$y[n] = x[n] \cdot (1 - mix) + y[n] \cdot mix. \tag{6}$$

  Where $x[n]$ is the input, $y[n]$ the processed input and mix a parameter control which has a range of [0, 1].

- Vol: Controls the master volume of the output by multiplying a coefficient from 0 to 1 to the output.

- Main: Has three modes. The first mode controls the mix of individual effects. The same formula as equation 6 is used but this only controls the depth of a single effect which is chosen by the user. The second mode controls the effect

slot. An effect slot is the container which can store one single audio effect, as previously mentioned there are eight effect slots. The third mode is to change the audio effect on the previously selected effect slot. Mode 2 and 3 are further explained in section 3.3.2.

- Par. 1 and 2: These parameters change depending on the effect and they're explained for each audio effect in section 3.2.

3.3.2 Menu

The menu is always visible through the LED ring but it can only be modified in the aforementioned mode 2 and 3. To access the different modes the button internal to the encoder in the center must be pressed, when pressing this button a variable named "btnState" is changed, this variable dictates the mode of the main encoder. Furthermore, the encoders, knobs and switches all have memory variables which are checked every time their state is read. This approach is taken so the main functions are only called when there is a change to the parameters. This assures no overflow occurs for the LEDs or the MIDI communication.
For mode 1, all knobs call the same function, "changePar", which takes care of detecting if the change was positive or negative for each encoder. As there are eight effect slots, this requires to have a matrix of 8x3 to store all values of the encoders. The function changePar takes care of assigning the new value to the proper memory slot as a value from 0 to 127 for MIDI communication which is explained in section 3.3.3 and also transforms the value so it can be displayed by the LEDs.
For mode 2, the variable that saves the current effect slot is now modified by the main encoder thorugh the function "changePos". All other parameters work in the same manner. The value of the new effect slot it not sent to the audio micro-controller until mode 2 is released. The function "changePos" calls a function named "ledMenu" which interprets the effect slot position and current audio effects to properly display them in the LED ring.
For mode 3 the audio effect is changed. The main encoder value calls a function named "changeFx" which in contrary to mode 2, this function is called every time the value is changed. This allows the user to hear the effect before deciding to choose it. As the previous mode, this function internally calls "ledMenu". The code for the user interface can be found in section M.

3.3.3 MIDI

MIDI was used as the main communication protocol between audio and user interface as this protocol was designed for electronic music instruments. This protocol consists of three bytes. For this case only the Control Change scheme will be explained as it was the only one used:

- Channel: A value from 0 to 16 indicates the channel where date will be sent and read.
- Controller: A value that ranges from 0 to 127, indicates the controller which is being modified.

- Value: The value of the controller being modified. Also changes from 0 to 127.

Currently. the audio micro-controller only reads MIDI data from the ESP32. The ESP32 sends control change values for only 5 controllers which range in the following order:

- Controller 0: Value of parameter 1.
- Controller 1: Value of parameter 2.
- Controller 2: Value of parameter 3.
- Controller 3: Value of audio effect for current effect slot.
- Controller 4: Value of current effect slot.

The value of the parameters range from 0 to 127 and are normalized by the audio micro-controller. The audio effect and effect slot value does not any post-processing as they are integer numbers. The audio effect slot ranges from 0 to 5, 0 being no effect and the other numbers being one of the effects mentioned in section 3.2. The effect slot ranges from 0 to 7, which is the total amount of effect slots. The code implementation is inside the user interface code in appendix M.

4 Measurements

Measurements were done to test the reliability of the hardware done. First, the FRF measurement of the hardware was done, afterwards the Total Harmonic Distortion (THD).

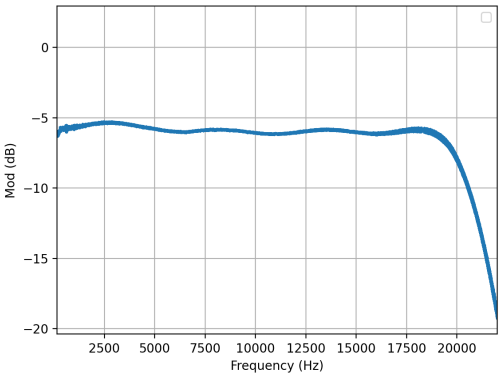


Figure 14: FRF of the hardware

The FRF was done for the hearing frequency range. As it can be seen in figure 14, The frequency response is not a straight line but its stable throughout the hearing range and it decays as it gets closer to it's Nyquist Frequency.

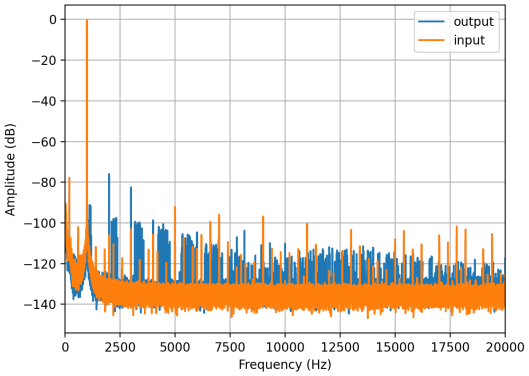


Figure 15: THD of the hardware

The percentage of total harmonic distortion of the system can be expressed theoretically by the following equation:

$$THD = \sqrt{\frac{U_2^2 + U_3^2 + U_4^2 + ... + U_n^2}{U_s^2}} * 100 \tag{7}$$

Using this equation, the THD calculated was of 0.012% which is extremely good considering the aforementioned measurement conditions.
Looking at figure 7, the Signal to Noise Ratio (SNR) can be estimated to be of 100 dB. This result is extremely good considering the measurement was done from the circuit implemented as a perfboard which means there are no ground planes and the circuit had no enclosure. Therefore, there was no protection from external electromagnetic frequencies. Implementing the circuit by proper made PCBs could lead to a further improvement in the SNR.

5 Conclusion

The implementation of a reverb and a processor capable of performing multiple effects in real time was successful. However, a lot of improvements could be made in all the aspects that conform the project. To begin with, finding research material for the project was a challenge as digital reverbs quickly became a commercial product rather than a research topic. Moreover, there is no paper or guideline for a project like this. Developing the concepts for the user interface and the stereo audio processor were done from scratch. The Teensy audio library was of great help but there is little documentation of projects which implement their own processor. Lastly, the improvements that could be done to the hardware and software will be discussed.

5.1 Hardware

The power supply designed for the project is based on a linear voltage regulator which ensures low noise but is not efficient and it's current limitations could lead

to malfunctions. A better design would be to implement a switching power supply as it could provide more current and it would be more efficient leading to less heat dissipated by the components. A buck converter power supply was designed but the voltage stability was not good enough for the project and time limitations made it not possible to continue prototyping for the power supply.

For the audio, the Teensy Audio Shield was chosen due to a lack of budget and materials for prototyping and building the PCB. Efforts were made to implement external ADc's and DACs. The implementation of a DAC with a breakout board was successful but not for an ADC. The PCM1808 was built in a custom breakout board, a market-available breakout board and by breadboard implementation. None of the aforementioned methods lead to successful results with the Teensy micro-controller. Further attempts should be made to have the chips implemented directly in a PCB and specially improve the filtering stage of the ADC, which uses a single passive filter with a cutoff frequency of $44.1kHz$ which is not optimal for this project. A higher order filter is required to ensure aliasing won't be an issue. For the purpose of this project, which it's main goal was to acquire knowledge in DSP. The Teensy 4.0 is a great choice but for a professional product, chips for specifically DSP purposes should be considered.

For the user interface, the potentiometers were set to be controlled exclusively with the ESP32 micro-controller. The potentiometers were meant to be set in the analog signal path as their purpose is to mix the input signal with the processed signal and control the master volume. However, as the audio is stereo, double-gang potentiometers were required which were out of the budget for the project. Furthermore, Voltage Controlled Amplifiers (VCAs) could allow the potentiometers stay in the digital section of the circuit yet control an analog signal path and still enable the values to be displayed through the LED ring. The LED ring is made of WS2812b LEDs and the model WS2813 could be more reliable.

5.2 Software

A lot of close attention was payed to memory management to ensure the effects could work at a low processing cost. However, optimization can still be performed. Currently, the DSP class has been made a parent class and all the audio effects a child class of the ME_DSP class. This ensures memory is used efficiently as effects that are not used do not take up memory. Furthermore, some functions work through float variables where it could rather be pointers, making the function call faster and less memory costly. For time-based effects no interpolation has been applied yet so audible artifacts are still a problem specially when changing the parameters.
The plate reverb implemented sounds great but there are more complex reverbs which are still inside the margin of the Teensy 4.0 processing power. Further effort could be done to implement more complex reverbs and audio effects which rely on Fast Fourier Transforms such as pitch-shifting.

6 Bibliography

- Reiss, J. D., & McPherson, A. Audio effects: Theory, implementation and application. CRC Press. 2014 .
- Dattorro, J. Effect design part 1: Reverberator and other filters. Stanford.Edu. Retrieved May 16, 2022.
- Tom Erbe Soundhack "designing the Make Noise Erbe-Verb" reverb design lecture (remastered). 2019, December 14.
- Antoine Chaigne, Jean Kergomard, Acoustique des instruments de musique; avec la collaboration de Xavier Boutillon, Jean-Pierre Dalmont, Benoît Fabre... [et al.], 2008.
- Daniel J. Inman. Englewood Cliffs (New Jersey), Engineering vibration : Prentice-Hall international. 1994.
- Distributed Mode Loudspeaker, Master 2 IMDEA Practicals, F. Ablitzer. 2020-21.

Appendix

A Github Repository

- Github Repository of project.

B Stereo Audio Processor

```
#include <Audio.h>
#include <SPI.h>
#include <Wire.h>
#include <SerialFlash.h>
#include "audio_processor.h" // Include custom audio processor
#include <MIDI.h>

// Audio Objects
AudioControlSGTL5000 sgtl5000_1;
AudioInputI2S usb1;
AudioOutputI2S i2s1;
AudioProcessor myProcessor;

MIDI_CREATE_INSTANCE(HardwareSerial, Serial1, MIDI); // Create MIDI
    communication

// UI
uint8_t pos = 0; // Current effect slot
uint8_t fx = 0; // Current effect
float step;

// Connect ADC - > Processor -> DAC
AudioConnection patchCord1(usb1, 0, myProcessor, 0);
AudioConnection patchCord2(usb1, 1, myProcessor, 1);
AudioConnection patchCord3(myProcessor, 0, i2s1, 0);
AudioConnection patchCord4(myProcessor, 1, i2s1, 1);

void setup() {

  // --- audio
  AudioMemory(50);
    // Enable the audio shield, select input, and enable output
  sgtl5000_1.enable();
  sgtl5000_1.inputSelect(AUDIO_INPUT_LINEIN);
  sgtl5000_1.muteHeadphone();
  sgtl5000_1.unmuteLineout();
  sgtl5000_1.lineInLevel(0);
  sgtl5000_1.lineOutLevel(25);
```

```
  // --- serial
  MIDI.setHandleControlChange(processMIDI); // Function to call for midi data
  MIDI.begin(2); // MIDI channel to read data from
  myProcessor.changeEffect(0,1); // Set first effect slot to audio effect 1, which is
      Tremolo

}

void loop() {
  MIDI.read();
}

void processMIDI(byte channel, byte number, byte value)
{
    step = value * MIDI_TO_FLOAT; // Transform value to float number
        between 0 and 1
    switch(number)
    {
      case 0: // Change parameter 1
        myProcessor.changeParam(pos,number,step);
        break;
      case 1: // Change parameter 2
        myProcessor.changeParam(pos,number,step);
        break;
      case 2: // Change parameter 3
        myProcessor.changeParam(pos,number,step);
        break;
      case 3: // Change audio effect
        fx = value;
        myProcessor.changeEffect(pos,fx);
        break;
      case 4: // Change effect slot
        pos = value;
        break;
    }
}
```

C Audio Processor Header

```
#ifndef audio_processor_h_
#define audio_processor_h_

#include "core_pins.h"
#include "AudioStream.h"
#include "ME_DSP.h" // custom DSP class
```

```
#include <stdlib.h>

class AudioProcessor : public AudioStream // inherit from AudioStream
{
public:

    AudioProcessor(void) : AudioStream(NUM_CHNLS, inputQueueArray)
    {
        for(k = 0; k < fxMax; k++) // Set buffer reading points for DSP objects
        {
          DSPL[k].initialize(&queue[0][0], &head, &test);
          DSPR[k].initialize(&queue[1][0], &head, &test);
        }
  }
    virtual void update(void); // Constants for float to void conversion
    const int resolutionDAC = 16;
    const int resolutionADC = 16;
    const float conversionConstADC = 1.0f/((1<<resolutionADC)-1);
    const float conversionConstDAC = (1<<resolutionDAC)-1;

    // Methods to change parameters inside DSP objects
    void changeParam(int fxPos, int numParam, const float _parameter);
    void changeEffect(int fxPos, int value);

private:

    inline void denormalize(float* sample); //flush small samples to zero

    audio_block_t *inputQueueArray[NUM_CHNLS]; // memory for incoming
        audio blocks

    // circular buffer
    float queue[NUM_CHNLS][BUFF_SIZE] = {};
    int head = 0; // writing index
    int test = 0; // compensation for effects that use global delay line

    // DSP Objects
    uint8_t fxNum = fxMax;
    // Create fxMax amount of DSP objects for each channel
    ME_DSP DSPL[fxMax];
    ME_DSP DSPR[fxMax];

    // variables for iterations
    uint8_t j = 0;
    uint8_t k = 0;
};
```

```
#endif
```

D Audio Processor CPP

```
#include "audio_processor.h"
#include <math.h>

void AudioProcessor::update(void)
{
    audio_block_t *blockL, *blockR;
    //float sample
    float sampleL, sampleR;

    // The data[] is an array of 16 bit integers representing the audio (blockL->
        data[i] is of uint16_t type)
    // (Note: The data[] array is always 32 bit aligned in memory, so you can
        fetch pairs of samples
    // by type casting the address as a pointer to 32 bit data (uint32_t))

    // obtain AUDIO_BLOCK_SAMPLES samples (by default 128)
    blockL = receiveWritable(0);
    blockR = receiveWritable(1);
    if (!blockL) return;
    if (!blockR) return;

    // audio processing
    for (j = 0; j < AUDIO_BLOCK_SAMPLES; j++)
    {

        // read the input signal
        sampleL = blockL->data[j] * conversionConstADC;
        sampleR = blockR->data[j] * conversionConstADC;

        // save sample to buffer
        queue[0][head] = sampleL;
        queue[1][head] = sampleR;

        test = head;
        // apply fx
        for (k = 0; k < fxNum; k++)
        {
            DSPL[k].process(&sampleL);
            DSPR[k].process(&sampleR);
        }
```

```
        // go to next buffer index
        head = (head + 1) % BUFF_SIZE;

        // denormalize
        denormalize(&sampleL);
        denormalize(&sampleR);

        // write output sample
        blockL->data[j] = sampleL * conversionConstDAC;
        blockR->data[j] = sampleR * conversionConstDAC;
    }

    // send buffer to output
    transmit(blockL, 0);
    transmit(blockR, 1);
    release(blockL);
    release(blockR);
}

inline void AudioProcessor::denormalize(float* sample)
{
    float absValue = fabs(*sample);
    if (absValue < 1e-15)
    {
        *sample = 0.0f;
    }
}

void AudioProcessor::changeParam(int fxPos, int numParam, const float
    _parameter) // change parameter for current effect slot
{
    __disable_irq();
    DSPL[fxPos].setParam(numParam, _parameter);
    DSPR[fxPos].setParam(numParam, _parameter);
    __enable_irq();
}

void AudioProcessor::changeEffect(int fxPos, int fx) // change effect for current
    effect slot
{
    __disable_irq();
    DSPL[fxPos].update(fx);
    DSPR[fxPos].update(fx);
    __enable_irq();
}
```

E D Audio Helper

```
#pragma once
// File to declare constants or any useful global function
#include <math.h>
// USER IO CONSTANTS
#define NUM_ENC 3
#define ENC_RESET 0.5f
#define MIDI_TO_FLOAT 0.00787401574 // 1 / 127

// General DSP
#define SEC 1
#define NUM_CHNLS 2
#define twoPI 2 * M_PI
#define PI_2 M_PI / 2
#define Fs 44100.0f
#define invFs 1 / Fs
#define fxMax 8
#define BUFF_SIZE 44100
```

F ME_DSP Header

```
#pragma once
#include "AudioHelper.h"

class ME_DSP
{
public:
  void initialize(float* buff, int* write, int* read);
  void update(int value);
  void process(float* sample);
  void setParam(int numParam, const float _parameter);

private:
  /* PROCESSOR HELPER */
  void reset();
  void paramUpdate();

  /* -- EFFECTS -- */
  void tremolo(float* sample);
  void bitcrush(float* sample);
  void vibrato(float* sample);
  void delay(float* sample);
  void reverb(float* sample);
  void clipping(float* sample, float thresh);
```

```
/* -- DSP HELPERS -- */
void LFO(float* out, float &phi, int wave);
void startVerb();
void endVerb();
void IIRfilter(float* sample, float gain, float* tap);
void allpass1(float* sample, float gain, float* tap, int delay);
void allpass2(float* sample, float gain, float* tap, int delay);
inline int positive_modulo(int i, int n);

/* -- PARAM HELPERS -- */
inline int map(float& x, int low, int high);

/* -- MEMORY -- */
int effect = 0;
float param[NUM_ENC] = { ENC_RESET };
float* buffer;
int* writeIndex;
int* readIndex;

float temp = 0.0f;
float mVar1 = 0.0f;
float mVar2 = 0.0f;
int mVar3 = 0;
int mVar4 = 0;

unsigned int i = 0;

unsigned long reverbIndex = 0;
// REVERB
float tempSample = 0;
float tempSample2 = 0;
float tempSample3 = 0;
float mPrev[3] = {};
float accumulator = 0;
int EXCURSION = 24;
float decay = 0.5f;
float decayDiffusion1 = 0.700f;
float decayDiffusion2 = 0.500f;
float inputDiffusion1 = 0.750f;
float inputDiffusion2 = 0.625f;
float bandwidth = 0.9995f;
float damping = 0.001f;
int size[8] = {998, 6598, 2667, 5512, 1345, 6249, 3936, 4687};
float* node13_14;
```

```
  float* node19_20;
  float* node15_16;
  float* node21_22;
  float* node23_24;
  float* node24_30;
  float* node31_33;
  float* node33_39;
  float* node46_48;
  float* node48_54;
  float* node55_59;
  float* node59_63;
};
```

G ME_DSP CPP

```
#include "ME_DSP.h"
#include <cmath>
#include <stdlib.h>

/* -- PROCESSOR -- */

void ME_DSP::update(int value)
{
  effect = value;
  if(effect == 6)
  {
    startVerb();
  }
  reset();
}

void ME_DSP::process(float* sample)
{
  switch (effect)
  {
    case 0:
        break;
    case 1:
        tremolo(sample);
        break;
    case 2:
        bitcrush(sample);
        break;
    case 3:
        vibrato(sample);
```

```
            break;
        case 4:
            delay(sample);
            break;
        case 5:
            clipping(sample, 1.0f − param[0]);
            break;
        case 6:
            reverb(sample);
            break;
        default:
            break;
    }
}

/* −− DSP EFFECTS −− */

void ME_DSP::clipping(float* sample, float thresh)
{
    if (*sample > thresh)
    {
        *sample = 1.0f − expf(−*sample);
    }
    else if(*sample < −thresh)
    {
        *sample = −1.0f + expf(*sample);
    }
}

void ME_DSP::tremolo(float* sample)
{
        // phase acts as f*n/Fs
        mVar1 += param[1] * 15 * invFs;
        // cal LFO for modulation
        LFO(&mVar2, mVar1, (int)param[2]*3);
        // dry / wet control
        *sample = *sample * (1.0f − param[0] + param[0] * mVar2);
        // reset phi
        if (mVar1 >= 1.0f)
        {
            mVar1 = 0.0f;
        }
}

void ME_DSP::delay(float* sample)
{
```

```
  mVar3 = param[1] * (BUFF_SIZE - 1); // calculate delay
  mVar4 = positive_modulo(*readIndex - mVar3, BUFF_SIZE); // set read
      index
  temp = *sample + param[2] * *(buffer + mVar4); // delay effect
  *sample = *sample * (1.0f - param[0]) + temp * param[0];
  clipping(sample, 0.8f);
  *(buffer + *readIndex) = *sample; // feedback
}

void ME_DSP::reverb(float* sample)
{
  mVar1 += 2 * invFs;
  //LFO(&mVar2, mVar1, 0);

  // input
  IIRfilter(sample, bandwidth, &mPrev[0]);
  allpass1(sample, inputDiffusion1, &node13_14[0], 210);
  allpass1(sample, inputDiffusion1, &node19_20[0], 159);
  allpass1(sample, inputDiffusion2, &node15_16[0], 562);
  allpass1(sample, inputDiffusion2, &node21_22[0], 410);

  // first tank
  tempSample2 = *sample + node59_63[(reverbIndex + 1) % (int) (param[1] *
      size[7])];
  allpass2(&tempSample2, decayDiffusion1, &node23_24[0], (int) (param[1] * size
      [0]));
  node24_30[reverbIndex % (int) (param[1] * size[1])] = tempSample2;
  IIRfilter(&node24_30[(reverbIndex + 1) % (int) (param[1] * size[1])], damping,
      &mPrev[1]);
  tempSample2 = param[2] * tempSample2;
  allpass1(&tempSample2, decayDiffusion2, &node31_33[0], (int) (param[1] * size
      [2]));
  node33_39[reverbIndex % (int) (param[1] * size[3])] = tempSample2;

  // second tank
  tempSample3 = *sample + node33_39[(reverbIndex + 1) % (int) (param[1] *
      size[3])];
  allpass2(&tempSample3, decayDiffusion1, &node46_48[0], (int)(param[1] * size
      [4]));
  node48_54[reverbIndex % (int) (param[1] * size[5])] = tempSample3;
  IIRfilter(&node48_54[(reverbIndex + 1) % (int) (param[1] * size[5])], damping,
      &mPrev[2]);
  tempSample3 = param[2] * tempSample3;
  allpass1(&tempSample3, decayDiffusion2, &node55_59[0], (int) (param[1] * size
      [6]));
  node59_63[reverbIndex % (int) (param[1] * size[7])] = tempSample3;
```

```
  // left output
  accumulator = 0.6 * node48_54[(reverbIndex + 394) % (int) (param[1] * size
      [5])];
  accumulator += 0.6 * node48_54[(reverbIndex + 4407) % (int) (param[1] * size
      [5])];
  accumulator -= 0.6 * node55_59[(reverbIndex + 2835) % (int) (param[1] * size
      [6])];
  accumulator += 0.6 * node59_63[(reverbIndex + 2958) % (int) (param[1] * size
      [7])];
  accumulator -= 0.6 * node24_30[(reverbIndex + 2949) % (int) (param[1] * size
      [1])];
  accumulator -= 0.6 * node31_33[(reverbIndex + 277) % (int) (param[1] * size
      [2])];
  accumulator -= 0.6 * node33_39[(reverbIndex + 1580) % (int) (param[1] * size
      [3])];

  *sample = *sample * (1.0f - param[0]) + param[0] * accumulator;

  reverbIndex++;
}

void ME_DSP::vibrato(float* sample)
{
  mVar1 += param[1] * 15 * invFs;
  LFO(&mVar2, mVar1, (int)param[2]*3);
  mVar3 = positive_modulo((int) *writeIndex - mVar2 * 150 * param[0],
      BUFF_SIZE);
  *sample = *(buffer + mVar3);
  *readIndex = mVar3;
}

void ME_DSP::bitcrush(float* sample)
{
    mVar1 = pow(2,map(param[1], 16, 2)-1);
    temp = ceil(mVar1 * *sample) / mVar1;
    *sample = *sample * (1.0f - param[0]) + temp * param[0];
}

/* -- DSP HELPERS -- */

void ME_DSP::LFO(float* out, float& phi, int wave)
{
    switch (wave)
    {
        case 0: // Sine
```

```
            *out = 0.5f + 0.5f * sinf(twoPI * phi);
            break;

        case 1: // Smooth Square
            if (phi < 0.48f)
                *out = 1.0f;
            else if (phi < 0.5f)
                *out = 1.0f - 50.0f * (phi - 0.48f);
            else if (phi < 0.98f)
                *out = 0.0f;
            else
                *out = 50.0f * (phi - 0.98f);
            break;

        case 2: // Triangle
            if (phi < 0.25f)
                *out = 0.5f + 2.0f * phi;
            else if (phi < 0.75f)
                *out = 1.0f - 2.0f * (phi - 0.25f);
            else
                *out = 2.0f * (phi - 0.75f);
            break;

        case 3: // Saw
            if (phi < 0.5f)
                *out = 0.5f - phi;
            else
                *out = 1.5f - phi;
            break;
    }
}

void ME_DSP::IIRfilter(float* sample, float gain, float* tap)
{
  *sample = *sample * gain + *tap * (1.0f - gain);
  *tap = *sample;
}

void ME_DSP::allpass1(float* sample, float gain, float* tap, int delay)
{
  *sample = *sample - gain * tap[(reverbIndex + 1) % delay];
  tap[reverbIndex % delay] = *sample;
  *sample = gain * *sample + tap[(reverbIndex + 1) % delay];
}

void ME_DSP::allpass2(float* sample, float gain, float* tap, int delay)
```

```
{
  *sample = *sample + gain * tap[positive_modulo(reverbIndex + 1 + (int)
      mVar2*400, delay)];
  tap[reverbIndex % delay] = *sample;
  *sample = - gain * *sample + tap[(reverbIndex + 1) % delay];
}

inline int ME_DSP::positive_modulo(int i, int n)
{
  return (i % n + n) % n;
}

/* -- PARAM HELPERS -- */

void ME_DSP::setParam(int numParam, const float _parameter)
{
    param[numParam] = _parameter;
    paramUpdate();
}

inline int ME_DSP::map(float& x, int low, int high)
{
  int out;
  out = (int) low + (high - low) * x;
  return out;
}

/* PROCESSOR HELPER */

void ME_DSP::startVerb()
{
  // REVERB
  tempSample = 0;
  tempSample2 = 0;
  tempSample3 = 0;
  accumulator = 0;
  EXCURSION = 24;
  decay = 0.98f;
  decayDiffusion1 = 0.700f;
  decayDiffusion2 = 0.500f;
  inputDiffusion1 = 0.750f;
  inputDiffusion2 = 0.625f;
  bandwidth = 0.9995f;
  damping = 0.001f;

  node13_14 = (float*) calloc(210,sizeof(float));
```

```
  node19_20 = (float*) calloc(159,sizeof(float));
  node15_16 = (float*) calloc(562,sizeof(float));
  node21_22 = (float*) calloc(410,sizeof(float));
  node23_24 = (float*) calloc(size[0],sizeof(float));
  node24_30 = (float*) calloc(size[1],sizeof(float));
  node31_33 = (float*) calloc(size[2],sizeof(float));
  node33_39 = (float*) calloc(size[3],sizeof(float));
  node46_48 = (float*) calloc(size[4],sizeof(float));
  node48_54 = (float*) calloc(size[5],sizeof(float));
  node55_59 = (float*) calloc(size[6],sizeof(float));
  node59_63 = (float*) calloc(size[7],sizeof(float));
}

void ME_DSP::initialize(float* buff, int* write, int* read)
{
  buffer = buff;
  writeIndex = write;
  readIndex = read;
}

void ME_DSP::reset()
{
  for(i = 0; i < NUM_ENC; i++)
  {
    param[i] = ENC_RESET;
  }
  temp = 0.0f;
  mVar1 = 0.0f;
  mVar2 = 0.0f;
  mVar3 = 0;
  mVar4 = 0;
}

void ME_DSP::paramUpdate() // change parameter values only
{
}
```

H Tremolo

```
void ME_DSP::tremolo(float* sample)
{
    for (i = 0; i < NUM_CHNLS; i++)
    {
        phi = lastPhi;
        // phase acts as f*n/Fs
```

```
            phi += param2 * 15 * invFs;
            // cal LFO for modulation
            LFO(&mod, phi, (int)param3*3);
            // dry / wet control
            *(sample + i) = *(sample + i) * (1.0f - param1 + param1 * mod);
            // reset phi
            if (phi >= 1.0f)
            {
                phi = 0.0f;
            }
        }
        lastPhi = phi;
}
```

I LFO

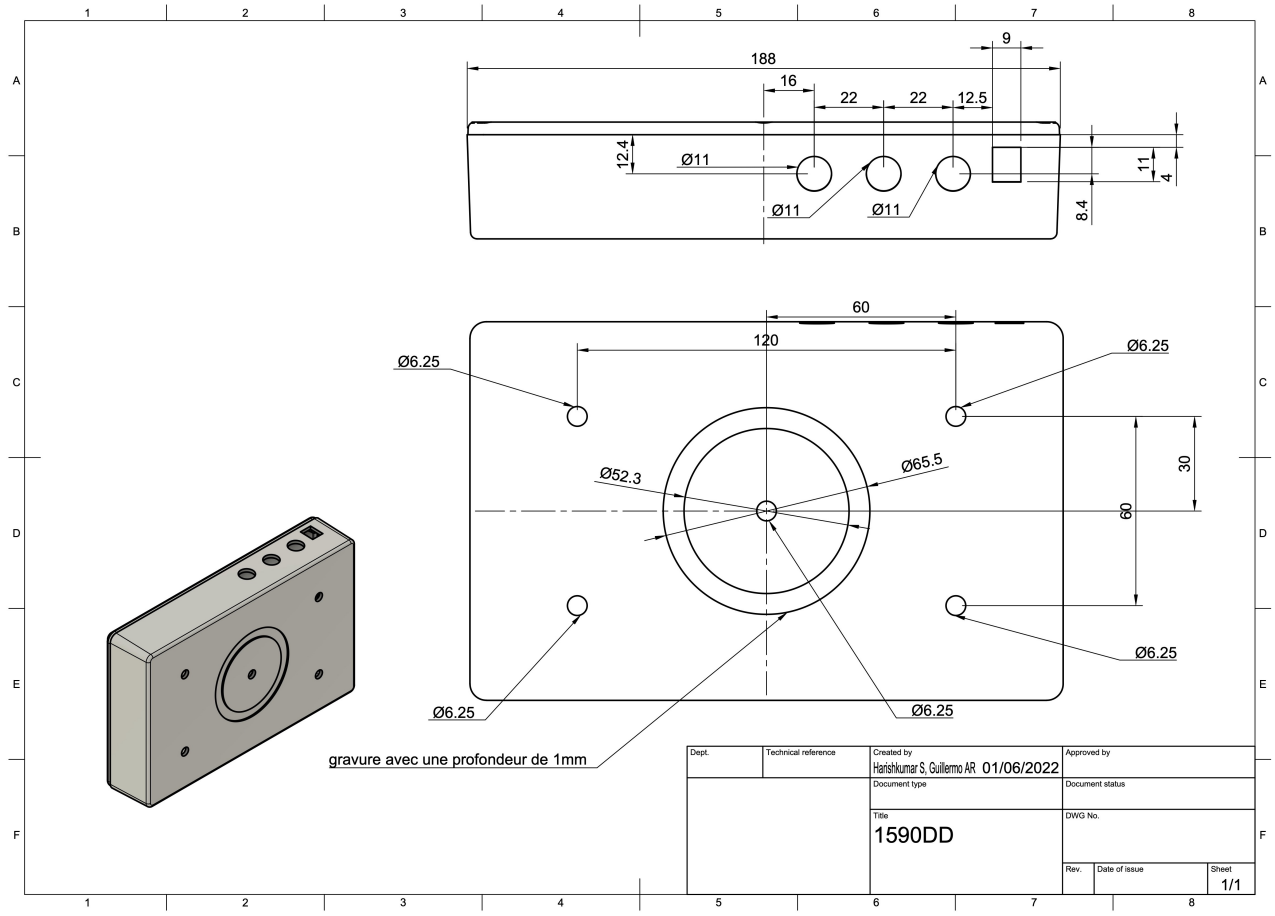
```
void ME_DSP::LFO(float* out, float& phi, int wave)
{
    switch (wave)
    {
        case 0: // Sine
            *out = 0.5f + 0.5f * sinf(twoPI * phi);
            break;

        case 1: // Smooth Square
            if (phi < 0.48f)
                *out = 1.0f;
            else if (phi < 0.5f)
                *out = 1.0f - 50.0f * (phi - 0.48f);
            else if (phi < 0.98f)
                *out = 0.0f;
            else
                *out = 50.0f * (phi -0.98f);
            break;

        case 2: // Triangle
            if (phi < 0.25f)
                *out = 0.5f + 2.0f * phi;
            else if (phi < 0.75f)
                *out = 1.0f - 2.0f * (phi - 0.25f);
            else
                *out = 2.0f * (phi - 0.75f);
            break;

        case 3: // Saw
```

```
            if (phi < 0.5f)
                *out = 0.5f - phi;
            else
                *out = 1.5f - phi;
            break;
    }
}
```

J Bitcrusher

```
void ME_DSP::bitcrush(float* sample)
{
    mod = pow(2,8-1);
    for (i = 0; i < NUM_CHNLS; i++)
    {
        *(sample + i) = 0.7 * ceil(mod * *(sample + i)) / mod;
    }
}
```

K Schematic for drilling holes with CNC

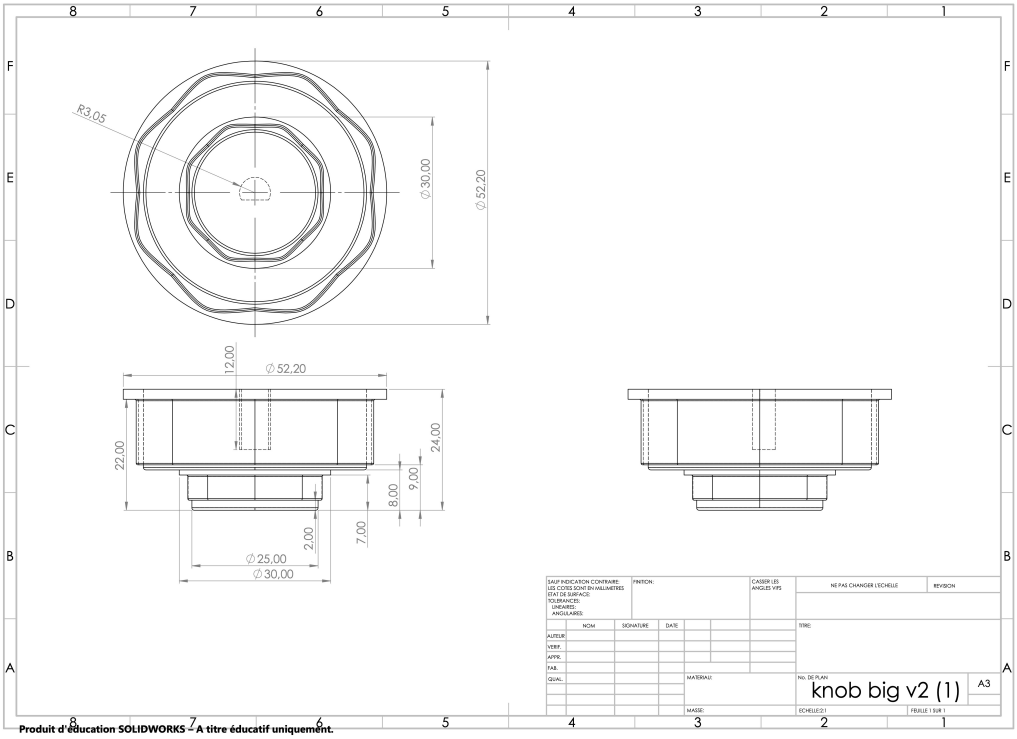


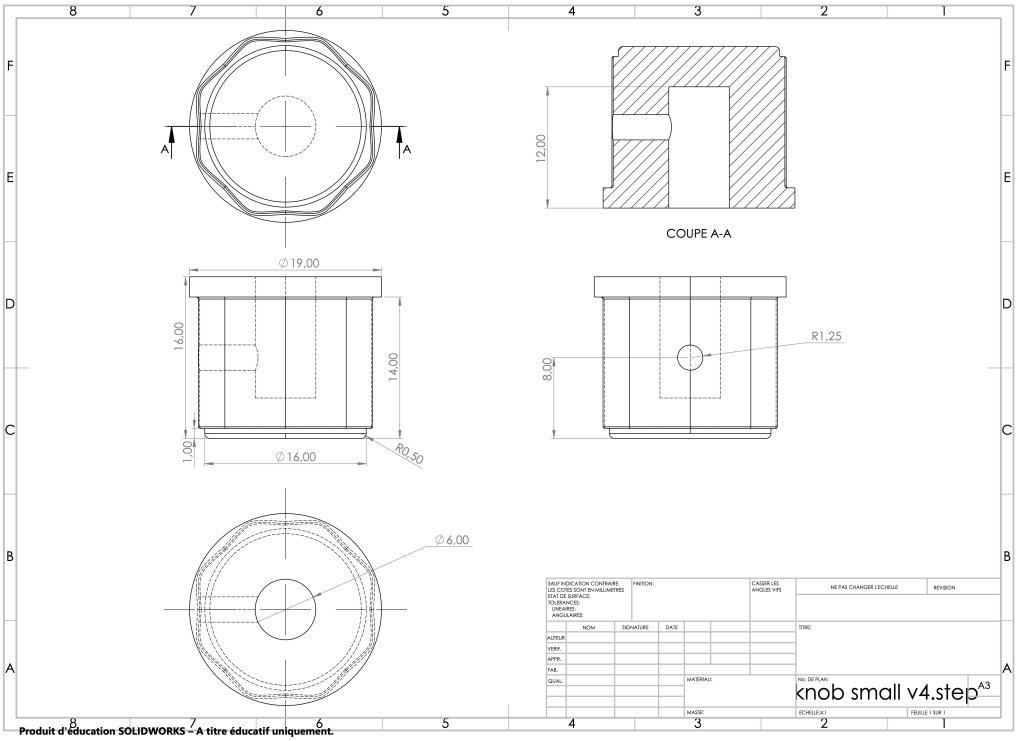
L Schematic for the main knob

L Schematic for the secondary knob

M User Interface Code

```
#include "AudioHelper.h"
#include <MIDI.h>
#include <Encoder.h>
#include "FastLED.h"

MIDI_CREATE_INSTANCE(HardwareSerial, Serial1, MIDI);

#define MIDI_CHNL 2
#define BTN1 25
#define NUM_LEDS 24
#define LED_PIN 4
int i;
int pos = 0;
int fx[MAXPOS] = {1, 0, 0, 0, 0, 0, 0, 0};
int mainBtn = 0;
```






```
int encDisplay[MAXPOS][NUM_ENC];
int step[NUM_ENC] = {};
int value2Send[MAXPOS][NUM_ENC] = { 64 };

unsigned long readTime;
```

```
int wait = 0;

// Encoder
Encoder enc1(23, 19);
Encoder enc2(22, 21);
Encoder enc3(5, 18);
int counter = 4;

// Touch Sensor
bool touchEvent = LOW;
bool touchWait = LOW;
unsigned long touchTime;

// LEDS
CRGB leds[NUM_LEDS];
CRGB menu[NUM_LEDS];
CRGB encValue[NUM_LEDS];
CRGB fxColors[MAXFX] = {CRGB(30, 10, 0), CRGB(30, 0, 0), CRGB(0, 30, 0),
    CRGB(0, 20, 30), CRGB(0, 0, 30)};
CRGB bright;

const int ledBlock = NUM_LEDS / MAXPOS;

void setup() {
  // BTNS
  pinMode(BTN1, INPUT);
  MIDI.begin();
  // LEDS
  FastLED.addLeds<WS2811, LED_PIN, GRB>(leds, NUM_LEDS).
      setCorrection( TypicalLEDStrip );
  for(i = 0; i < NUM_LEDS; i++)
  {
    menu[i] = fxColors[0];
  }
  for(i = 0; i < ledBlock; i++)
  {
    menu[i] = fxColors[1];
  }
  ledUpdate(1);
  //Serial
  Serial.begin(57600);
}

void loop() {
  // read encoder user input
  readEncoder( enc1.readAndReset(), 0 );
```

```
readEncoder( enc2.readAndReset(), 1 );
readEncoder( enc3.readAndReset(), 2 );

// read capacitive sensor
touch(touchRead(T9));

 // debounce
readBtn();
if(millis() > readTime + 500)
{
  wait = 0;
}

// if change detected, process and send user input
if(step[0] != 0)
{
  switch(mainBtn)
  {
    case 0:
      changePar(0, step[0]);
      Serial.print("Encoder 1:");
      Serial.println(value2Send[pos][0]);
      break;
    case 1:
      changePos();
      ledUpdate(1);
      Serial.print("pos: ");
      Serial.println(pos);
      break;
    case 2:
      changeFx();
      ledUpdate(1);
      Serial.print("fx: ");
      Serial.println(fx[pos]);
      break;
  }
  step[0] = 0;


}
if(step[1] != 0)
{
  Serial.print("Encoder 2: ");
  Serial.println(value2Send[pos][1]);
  changePar(1, step[1]);
  step[1] = 0;
```

```
  }
  if(step[2] != 0)
  {
    Serial.print("Encoder 3: ");
    Serial.println(value2Send[pos][2]);
    changePar(2, step[2]);
    step[2] = 0;
  }
}

void readEncoder(long newPosition, int i)
{
  if(newPosition != 0)
  {
    if(newPosition > 0)
    {
      counter++;
    }
    else
    {
      counter--;
    }
  }
  if(counter >= 8)
  {
    step[i] = 1;
    counter = 4;
  }
  if(counter <= 0)
  {
    step[i] = -1;
    counter = 4;
  }
}

void readBtn()
{
  if(digitalRead(BTN1) == HIGH && wait == 0)
  {
    wait = 1;
    if(mainBtn < 2)
    {
      mainBtn++;
    }
    else
    {
```

```
        mainBtn = 0;
      }
      readTime = millis();
      Serial.println("btn1");
    }
}

void changePar(int numParam, int value)
{
  //send param thru serial
  value2Send[pos][numParam] += (int) value*3.5;
  if(value2Send[pos][numParam] < 0)
  {
    value2Send[pos][numParam] = 0;
  }
  else if(value2Send[pos][numParam] > 127)
  {
    value2Send[pos][numParam] = 127;
  }
  else
  {
    MIDI.sendControlChange(numParam,value2Send[pos][numParam],
        MIDI_CHNL);
    // LEDS
    encDisplay[pos][numParam] = ceil(value2Send[pos][numParam] *
        NUM_LEDS * MIDI_TO_FLOAT);
    encLED(numParam);
    ledUpdate(0);
  }

}

void changeFx()
{
  fx[pos] += step[0];
  bound(&fx[pos],0,MAXFX); // circular scroll boundaries
  // send serial command to change fx
  MIDI.sendControlChange(3,fx[pos],MIDI_CHNL);
  // reset values
  value2Send[pos][0] = 64;
  value2Send[pos][1] = 64;
  value2Send[pos][2] = 64;
  // Display with LEDS
  ledMenu();
}
```

```
void changePos()
{
  // Set non active block to normal brightness
  for(i = pos * ledBlock; i < (pos + 1) * ledBlock; i++ )
  {
    menu[i] = fxColors[fx[pos]];
  }
  pos += step[0];
  bound(&pos,0,MAXPOS-1); // circular scroll boundaries
  // send new pos value thru serial
  MIDI.sendControlChange(4,pos,MIDI_CHNL);
  // Make LEDS of current position brighter
  ledMenu();
}

void ledUpdate(bool condition)
{
  if(condition == 0)
  {
    for(i = 0; i < NUM_LEDS; i++)
    {
      leds[i] = encValue[i];
    }
  }
  else
  {
    for(i = 0; i < NUM_LEDS; i++)
    {
      leds[i] = menu[i];
    }
  }
  FastLED.show();
}

void ledMenu() // Update LED memory for menu
{
  for(i = pos * ledBlock; i < (pos + 1) * ledBlock; i++)
  {
    bright.r = fxColors[fx[pos]].r * 3;
    bright.g = fxColors[fx[pos]].g * 3;
    bright.b = fxColors[fx[pos]].b * 3;
    menu[i] = bright;
  }
}

void encLED(int numParam) // Update LED memory for encoder value
```

```
{
  for(i = 0; i < NUM_LEDS; i++)
  {
    if(i < encDisplay[pos][numParam])
    {
      encValue[i] = fxColors[fx[pos]];
    }
    else
    {
      encValue[i] = CRGB::Black;
    }
  }
}

void touch(int sensor)
{
  //Serial.println(touchRead(T9));
  if(sensor < 32 && touchEvent == LOW && mainBtn == 0)
  {
    encLED(0);
    ledUpdate(0);
    touchEvent = HIGH;
    touchTime = millis();
    touchWait = LOW;

  }
  if(sensor >= 32 && touchEvent == HIGH)
  {
    touchTime = millis();
    touchEvent = LOW;
    touchWait = HIGH;
  }
  if(millis() > touchTime + 300 && touchWait == HIGH)
  {
    ledUpdate(1);
    touchWait = LOW;
  }
}
```

N Plate Reverb Implemented in Python

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jun 9 16:08:40 2022
```

```
@author: Guillermo
"""

import numpy as np
import matplotlib.pyplot as plt
import scipy.fft as fft
plt.close('all')

tempSample = 0.0
tempSample2 = 0.0
tempSample3 = 0.0
accumulator = 0.0

mPrev = np.zeros(3)

reverbIndex = 0.0
EXCURSION = 24
decay = 0.2
decayDiffusion1 = 0.7
decayDiffusion2 = 0.5
inputDiffusion1 = 0.75
inputDiffusion2 = 0.625
bandwidth = 1 - np.exp(-2*np.pi*10e3/44100)
damping = np.exp(-2*np.pi*10e3/44100)

size = [998, 6598, 2667, 5512, 1345, 6249, 3936, 4687]
node13_14 = np.zeros(210)
node19_20 = np.zeros(159)
node15_16 = np.zeros(562)
node21_22 = np.zeros(410)
node23_24 = np.zeros(size[0])
node24_30 = np.zeros(size[1])
node31_33 = np.zeros(size[2])
node33_39 = np.zeros(size[3])
node46_48 = np.zeros(size[4])
node48_54 = np.zeros(size[5])
node55_59 = np.zeros(size[6])
node59_63 = np.zeros(size[7])

reverbIndex = 0

def IIRfilter(sample, gain, tap):
    sample = sample * gain + tap * (1 - gain)
    tap = sample
    return [sample, tap]
```

```
def allpass1(sample, gain, tap, delay):
    sample = sample - gain * tap[(reverbIndex + 1) % delay]
    tap[reverbIndex % delay] = sample
    sample = gain * sample + tap[(reverbIndex + 1) % delay]

def allpass2(sample, gain, tap, delay):
    sample = sample + gain * tap[(reverbIndex + 1) % delay]
    tap[reverbIndex % delay] = sample
    sample = - gain * sample + tap[(reverbIndex + 1) % delay]

length = 1*44100
x = np.zeros(length)
y = np.zeros_like(x)
x[0] = 1.0

while(reverbIndex < length):
    tempSample = x[reverbIndex]
    # IIR Filter
    tempSample = tempSample * bandwidth + mPrev[0] * (1 - bandwidth)
    mPrev[0] = tempSample

    # allpass1
    tempSample = tempSample - inputDiffusion1 * node13_14[(reverbIndex +
        1) % 210]
    node13_14[reverbIndex % 210] = tempSample
    tempSample = inputDiffusion1 * tempSample + node13_14[(reverbIndex +
        1) % 210]


    # allpass1(&tempSample, inputDiffusion1, &node19_20[0], 159, reverbIndex);
    tempSample = tempSample - inputDiffusion1 * node19_20[(reverbIndex +
        1) % 159]
    node19_20[reverbIndex % 159] = tempSample
    tempSample = inputDiffusion1 * tempSample + node19_20[(reverbIndex +
        1) % 159]

    # allpass1(&tempSample, inputDiffusion2, &node15_16[0], 562, reverbIndex);
    tempSample = tempSample - inputDiffusion2 * node15_16[(reverbIndex +
        1) % 562]
    node15_16[reverbIndex % 562] = tempSample
    tempSample = inputDiffusion2 * tempSample + node15_16[(reverbIndex +
        1) % 562]

    # allpass1(&tempSample, inputDiffusion2, &node21_22[0], 410, reverbIndex);
    tempSample = tempSample - inputDiffusion2 * node21_22[(reverbIndex +
        1) % 410]
```

```
node21_22[reverbIndex % 410] = tempSample
tempSample = inputDiffusion2 * tempSample + node21_22[(reverbIndex +
    1) % 410]


## FIRST TANK ##
tempSample2 = tempSample + node59_63[(reverbIndex + 1) % size[7]]

# allpass2(&tempSample2, decayDiffusion1, &node23_24[0], size[0],
    reverbIndex);
tempSample2 = tempSample2 + decayDiffusion1 * node23_24[(reverbIndex +
    1) % size[0]]
node23_24[reverbIndex % size[0]] = tempSample2
tempSample2 = - decayDiffusion1 * tempSample2 + node23_24[(reverbIndex
    + 1) % size[0]]

node24_30[reverbIndex % size[1]] = tempSample2

# IIRfilter(&node24_30[(reverbIndex + 1) % size[1]], damping, &mPrev[1]);
node24_30[(reverbIndex + 1) % size[1]] = node24_30[(reverbIndex + 1) %
    size[1]] * (1 - damping) + mPrev[1] * damping
mPrev[1] = node24_30[(reverbIndex + 1) % size[1]]

tempSample2 = decay * tempSample2

# allpass1(&tempSample2, decayDiffusion2, &node31_33[0], size[2],
    reverbIndex);
tempSample2 = tempSample2 - decayDiffusion2 * node31_33[(reverbIndex +
    1) % size[2]]
node31_33[reverbIndex % size[2]] = tempSample2
tempSample2 = decayDiffusion2 * tempSample2 + node31_33[(reverbIndex +
    1) % size[2]]

node33_39[reverbIndex % size[3]] = tempSample2

## SECOND TANK ##
tempSample3 = tempSample + node33_39[(reverbIndex + 1) % size[3]]

# allpass2(&tempSample3, decayDiffusion1, &node46_48[0], size[4],
    reverbIndex);
tempSample3 = tempSample3 + decayDiffusion1 * node46_48[(reverbIndex +
    1) % size[4]]
node46_48[reverbIndex % size[4]] = tempSample3
tempSample3 = - decayDiffusion1 * tempSample3 + node46_48[(reverbIndex
    + 1) % size[4]]
```

```
    node48_54[reverbIndex % size[5]] = tempSample3

    # IIRfilter(&node48_54[(reverbIndex + 1) % size[5]], damping, &mPrev[2]);
    node48_54[(reverbIndex + 1) % size[5]] = node48_54[(reverbIndex + 1) %
        size[5]] * (1 - damping) + mPrev[2] * damping
    mPrev[2] = node48_54[(reverbIndex + 1) % size[5]]

    tempSample3 = decay * tempSample3


    # allpass1(&tempSample3, decayDiffusion2, &node55_59[0], size[6],
        reverbIndex);
    tempSample3 = tempSample3 - decayDiffusion2 * node55_59[(reverbIndex +
        1) % size[6]]
    node55_59[reverbIndex % size[6]] = tempSample3
    tempSample3 = decayDiffusion2 * tempSample3 + node55_59[(reverbIndex +
        1) % size[6]]

    node59_63[reverbIndex % size[7]] = tempSample3;

    ## OUTPUT ##
    accumulator = 0.6 * node48_54[(reverbIndex + 394) % size[5]]
    accumulator += 0.6 * node48_54[(reverbIndex + 4407) % size[5]]
    accumulator -= 0.6 * node55_59[(reverbIndex + 2835) % size[6]]
    accumulator += 0.6 * node59_63[(reverbIndex + 2958) % size[7]]
    accumulator -= 0.6 * node24_30[(reverbIndex + 2949) % size[1]]
    accumulator -= 0.6 * node31_33[(reverbIndex + 277) % size[2]]
    accumulator -= 0.6 * node33_39[(reverbIndex + 1580) % size[3]]

    y[reverbIndex] = accumulator;
    reverbIndex += 1

t = np.arange(length)/44100
plt.figure()
ax = plt.subplot(111)
font = {'family' : 'sans',
        'weight' : 'normal',
        'size' : 14}

plt.rc('font', **font)
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
plt.rcParams['axes.linewidth'] = 3
plt.title('IR_of_Plate_Reverb_Algorithm', fontsize = 18)
ax.plot(t,x, label = 'Input', color = (84/255, 127/255, 169/255), linewidth = 3)
```

```
ax.plot(t,y*9, label = 'Plate Reverb', color = (228/255, 96/255, 84/255),
    linewidth = 3)
ax.legend()
plt.xlabel('Time [s]', fontsize = 18)
plt.ylabel('Amplitude', fontsize = 18)
plt.tight_layout()
plt.show()
plt.savefig('demo.png', transparent=True)

Nfft = int(44100 / 2)
Y = fft.rfft(y,n = Nfft)
plt.figure()
plt.plot(np.abs(Y))
plt.show()
```

O Plate Reverb for Teensy

```
mVar1 += 2 * invFs;
 //LFO(&mVar2, mVar1, 0);

 // input
 tempSample = IIRfilter(*sample, bandwidth, &mPrev[0]);
 allpass1(sample, inputDiffusion1, &node13_14[0], 210);
 allpass1(sample, inputDiffusion1, &node19_20[0], 159);
 allpass1(sample, inputDiffusion2, &node15_16[0], 562);
 allpass1(sample, inputDiffusion2, &node21_22[0], 410);

 // first tank
 tempSample2 = *sample + node59_63[(reverbIndex + 1) % (int) (param[1] *
     size[7])];
 allpass2(&tempSample2, decayDiffusion1, &node23_24[0], (int) (param[1] * size
     [0]));
 node24_30[reverbIndex % (int) (param[1] * size[1])] = tempSample2;
 tempSample2 = IIRfilter(node24_30[(reverbIndex + 1) % (int) (param[1] * size
     [1])], damping, &mPrev[1]);
 tempSample2 = param[2] * tempSample2;
 allpass1(&tempSample2, decayDiffusion2, &node31_33[0], (int) (param[1] * size
     [2]));
 node33_39[reverbIndex % (int) (param[1] * size[3])] = tempSample2;

 // second tank
 tempSample3 = *sample + node33_39[(reverbIndex + 1) % (int) (param[1] *
     size[3])];
 allpass2(&tempSample3, decayDiffusion1, &node46_48[0], (int)(param[1] * size
     [4]));
```

```
  node48_54[reverbIndex % (int) (param[1] * size[5])] = tempSample3;
  tempSample3 = IIRfilter(node48_54[(reverbIndex + 1) % (int) (param[1] * size
      [5])], damping, &mPrev[2]);
  tempSample3 = param[2] * tempSample3;
  allpass1(&tempSample3, decayDiffusion2, &node55_59[0], (int) (param[1] * size
      [6]));
  node59_63[reverbIndex % (int) (param[1] * size[7])] = tempSample3;

  // left output
  accumulator = 0.6 * node48_54[(reverbIndex + 394) % (int) (param[1] * size
      [5])];
  accumulator += 0.6 * node48_54[(reverbIndex + 4407) % (int) (param[1] * size
      [5])];
  accumulator -= 0.6 * node55_59[(reverbIndex + 2835) % (int) (param[1] * size
      [6])];
  accumulator += 0.6 * node59_63[(reverbIndex + 2958) % (int) (param[1] * size
      [7])];
  accumulator -= 0.6 * node24_30[(reverbIndex + 2949) % (int) (param[1] * size
      [1])];
  accumulator -= 0.6 * node31_33[(reverbIndex + 277) % (int) (param[1] * size
      [2])];
  accumulator -= 0.6 * node33_39[(reverbIndex + 1580) % (int) (param[1] * size
      [3])];

  *sample = *sample * (1.0f - param[0]) + param[0] * accumulator;

  reverbIndex++;
}

void ME_DSP::vibrato(float* sample)
{
  mVar1 += param[1] * 15 * invFs;
  LFO(&mVar2, mVar1, (int)param[2]*3);
  mVar3 = positive_modulo((int) *writeIndex - mVar2 * 150 * param[0],
      BUFF_SIZE);
  *sample = *(buffer + mVar3);
  *readIndex = mVar3;
}

void ME_DSP::bitcrush(float* sample)
{
    mVar1 = pow(2,map(param[1], 16, 2)-1);
    temp = ceil(mVar1 * *sample) / mVar1;
    *sample = *sample * (1.0f - param[0]) + temp * param[0];
}
```

```
/* -- DSP HELPERS -- */

void ME_DSP::LFO(float* out, float& phi, int wave)
{
    switch (wave)
    {
        case 0: // Sine
            *out = 0.5f + 0.5f * sinf(twoPI * phi);
            break;

        case 1: // Smooth Square
            if (phi < 0.48f)
                *out = 1.0f;
            else if (phi < 0.5f)
                *out = 1.0f - 50.0f * (phi - 0.48f);
            else if (phi < 0.98f)
                *out = 0.0f;
            else
                *out = 50.0f * (phi - 0.98f);
            break;

        case 2: // Triangle
            if (phi < 0.25f)
                *out = 0.5f + 2.0f * phi;
            else if (phi < 0.75f)
                *out = 1.0f - 2.0f * (phi - 0.25f);
            else
                *out = 2.0f * (phi - 0.75f);
            break;

        case 3: // Saw
            if (phi < 0.5f)
                *out = 0.5f - phi;
            else
                *out = 1.5f - phi;
            break;
    }
}

float ME_DSP::IIRfilter(float sample, float gain, float* tap)
{
  return sample = *sample * gain + *tap * (1.0f - gain);
  *tap = *sample;
}

void ME_DSP::allpass1(float* sample, float gain, float* tap, int delay)
```

```
{
  *sample = *sample - gain * tap[(reverbIndex + 1) % delay];
  tap[reverbIndex % delay] = *sample;
  *sample = gain * *sample + tap[(reverbIndex + 1) % delay];
}

void ME_DSP::allpass2(float* sample, float gain, float* tap, int delay)
{
  *sample = *sample + gain * tap[positive_modulo(reverbIndex + 1 + (int)
      mVar2*400, delay)];
  tap[reverbIndex % delay] = *sample;
  *sample = - gain * *sample + tap[(reverbIndex + 1) % delay];
}
```