ELEX 7890 Capstone Project

Final Report

Multi-FX and Amp Modeler

1 Abstract

Did you know that musicians frequently spend thousands of dollars and countless hours managing complex setups of amplifiers, pedals, and other sound equipment? For gigging musicians, producers, and hobbyists alike, transporting multiple pieces of gear is not only costly but also challenging for consistent sound quality and reliability. As fourth-year electrical engineering students, we provided a more efficient, sustainable solution by developing a portable Multi-FX and Amp Modeler. This device integrates high-quality amp modeling and digital effects into a single, compact processor, significantly simplifying the process of sound customization for artists.

Our project, designed for both live and studio applications, will allow users to create custom sound profiles and manage them through a user-friendly touchscreen interface. By addressing logistical challenges and making professional-grade sound processing more accessible, this product not only supports musicians' creativity but also reduces the environmental impact of traditional analog setups.

Submitted as part of a BCIT electrical engineering Capstone project: 2023-2024		
Prepared by:	Student Number:	Submission Date:
Federico Sarabia	A00913638	05/13/2025
Spencer Arnold	A01282057	
Hassan Islam	A01248771	
Riley Olfert	A01280488	

Table of Figures

2 Executive Summary

This document presents the preliminary proposal for the Capstone project developed by our team. Guided by our capstone supervisor Victor Mendez and project mentor John Dian, we designed a Multi-FX and Amp Modeler, built as a versatile tool for gigging musicians, producers, and hobbyists who require high-quality sound processing in a portable format.

The device addresses critical issues faced by musicians, such as the high costs and complexity of transporting multiple amplifiers, pedals, and other sound equipment. By combining amp modeling and a wide range of digital effects, this project offers an all-in-one solution to meet users' needs for both reliability and sound quality. The touchscreen interface allows for intuitive effects customization, ensuring that users can quickly adapt to new environments and achieve consistent results.

Our initial research highlights the growing demand in the digital audio market, with the Canadian market for electronic instruments projected to reach US$169.7 million in 2024. The project will focus on developing a functional prototype with DSP (digital signal processing) algorithms, a custom PCB for low noise audio processing, and a housing with a selection of inputs, outputs, and elements for human interface including buttons, slider, and a touchscreen. Future development phases will include software updates to expand the effects library, enhancing the product's long-term value.

The device implements modern digital signal processing algorithms to achieve fast processing. Our work not only simplifies logistical challenges for musicians but also makes high-quality sound processing more accessible to a broader community of artists and hobbyists.

3 Acknowledgements

We would like to extend our sincere gratitude to the individuals who supported and guided us throughout the development of our Multi-FX and Amp Modeler Capstone project.

First and foremost, we thank our mentor, John Dian, whose technical expertise, consistent feedback, and mentorship were instrumental in shaping our project. His guidance helped us navigate the complexities of real-time DSP, embedded systems, and amplifier modeling with confidence and clarity.

We are also grateful to Victor Mendez, program head of Electrical Engineering, for fostering an environment that encourages innovation, collaboration, and excellence. His leadership ensured we had the tools and opportunities needed to succeed.

Special thanks to Katherine Golder, our communications mentor, whose support in documentation, report structure, poster design, and professional presentation helped us communicate our work clearly and effectively.

Finally, we would like to thank Connor Neidig, a local guitarist and session musician, for providing valuable insight into the needs of real-world users. His feedback on tone, usability, and interface design was essential in aligning our work with the expectations of our target audience.

To all who supported us, thank you for being part of this journey.

Introduction

The purpose of this report is to provide a detailed description of the various aspects of our Multi-FX and Amp Modeler digital signal processor project. This will allow for review by faculty of BCIT's School of Energy, students, our project mentor John Dian, other project stakeholders, and future capstone groups looking to inherit our work. This report will provide insights to our design process and decision making. Furthermore, the report will also highlight the challenges we encountered during the project and the steps taken to address each problem.

Throughout this document, any terms specific to digital signal processing (DSP) technology will be briefly explained as needed to ensure comprehension. This report is designed to be accessible to readers with general engineering or nontechnical backgrounds, while also providing enough depth for those with DSP expertise.

The Multi-FX and Amp Modeler project was developed from a desire to apply and deepen our understanding of modern DSP algorithms and printed circuit board design. This project spans eight months, divided into two semesters: the first semester focuses on research, and the second on prototyping the device. Most of the project work took place at BCIT's laboratories and project workspaces, where we made extensive use of the institution's engineering labs and technical resources. Our Multi-FX and Amp Modeler capstone also fulfills part of the graduation requirements for the Bachelor of Electrical Engineering program at BCIT. This project was originally proposed as part of our final year design course; this report follows from the initial project proposal submitted in September 2024.

This scope will cover the technical considerations and specifications for the development of our Multi-FX and Amp modeler. It includes the theoretical foundations and DSP concepts used in designing our algorithms and modeling techniques. Simulation and testing conducted in MATLAB will also be discussed. Additionally, the report details our hardware component selection, signal processing pipeline, firmware, and user interface design. Final chassis and power supply design are not discussed here, as they were not the project's focus.

4 Background and Motivation

Our end users are professional guitar-playing musicians and hobbyists who need a portable, all-in-one solution for playing classic guitar tones and effects. These users value high-quality sound modeling and seek to avoid the hassle of transporting bulky equipment like amplifiers, cabs, and pedals. They also value flexibility in customizing their signal chain. Whether performing at professional shows or playing at home, they require a device that delivers exceptional sound quality, flexibility, and ease of use. The following section details who the End-users are and their requirements.

4.1 End-Users and Stakeholders

4.1.1 End-Users

- **Gigging Musicians:** These users perform regularly and need a reliable, portable solution that reduces their equipment load while ensuring consistent sound quality at each venue. They require a device that is durable, easy to set up, and capable of producing high-quality, diverse sound profiles.

- **Producers:** Producers, particularly those in studio settings, benefit from a flexible device that integrates easily with digital audio workstations and offers a wide range of effects and amp models. This device enables them to achieve unique sounds without needing to invest in multiple physical amps and pedals, streamlining the recording process.

- **Hobbyists:** This group includes serious guitar and bass enthusiasts who seek professional-grade sound and customizable effects at home or for smaller gatherings. They prioritize affordability, ease of use, and flexibility to experiment with different tones and effects.

4.1.2 Stakeholders

- **Mentor/Instructor:** For the staff of BCIT's School of Energy, including Pejman Alibeigi, John Dian, Victor Mendez, and Katherine Golder.

4.2 Application and Need

Multi-FX processors and Amp modelers can help professional guitar-playing musicians and hobbyists alike from lugging around large guitar amplifiers, cabs, and guitar pedals. Our customers will use our device to make new and exciting rhythmic sounds and music. This could be in any context such as at a professional show or just at home.

The problem our product solves for our customers is that they will no longer need to carry around many guitar amps, pedals, and cabs but instead can use the multi-FX processor and amp modeler as an all-in-one device.

Transporting large amounts of equipment is often a big issue for musicians and hobbyists. Therefore, solving this problem in an elegant manner will help capture the Canadian electrical guitar market. With US$169.7m in revenue in 2024 in Canada in electronic instruments we can see that this market is has sufficient scale and demand to support our niche product tailored to musicians, hobbyists, and producers.

4.3 State of the Art

Similar devices aiming to provide users with effects processing and amplifier modeling exist in the market today. Modern devices e.g., the Line 6 Helix, Neural DSP, and Axe FX III implement user-interfaces, modern digital signal processors, and algorithms to implement real-time and low-latency[1] data streaming. This means that users can expect to hear their audio signal output, with all the processing, at almost the exact instance they play their instrument. The following section explores these products and highlights their current limitations and their solutions to low-latency digital signal processing.

Line 6 Helix

The Line 6 Helix Floor amp and effect processor utilizes modern DSP technology to provide the sound and feel of modern and vintage amps, cabs, and effects. It boasts a massive library of 106 amplifiers, and 273 guitar effects. The device implements an ADSP-21489KSWZ-3B, DSP specialized processor, allowing it to process CPU intensive effects in real time. High performance 32-bit/40-bit floating-point processor allows for fast mathematic operations. The device was released on the market in 2015 and has seen many software updates since then.

Figure 1: Line 6 Helix

Neural DSP Quad Cortex

The Neural DSP Quad Cortex is a 6-core floorboard amp modeler that simulates guitar effects pedals, amps, and cabinets. It has a built in 7” capacitive touch screen and a series of buttons and knobs to control the audio output. The Neural has a small form factor for better portability. The Neural has 4 SHARC®+ cores and 2 ARM Cortex-A5 cores both running at 500MHz. The device has support for Midi input and output as well as support for external pedals.

Figure 2: Neural DSP Quad Cortex

Axe FX III MK II

The Axe FX III makes use of a 16 core 500MHz microcontroller that allows for 16 channels of 48kHz audio (8 in, 8 out). It also has 72 effects and an 800x480 color display and a series of buttons and knobs to control the audio output. The Axe-Fx III is rackmount for 4U size racks for convenience.

Figure 3: Axe FX III MK II

[1] Latency: The delay between the audio at the input and output of a system.

4.4 Modern DSP Algorithms

Digital signal processing is a mathematically intense endeavour. Alongside elementary operations such as multiplication and addition; more complex operations are needed to perform signal filtering. These operations can be computationally heavy and if done naively can incur significant processing delays. In audio especially, processing needs to be fast such that the human senses cannot detect any latency between an input of a signal and its processed output. The following section introduces a commonly used mathematical operation called convolution and techniques used to perform the operation fast.

4.4.1 Convolution

Discrete convolution is a common mathematical operation in DSP. This operation is used often to apply filters to signals. This allows us to shape the frequency content of a signal, for example, reducing the high frequencies of a guitar signal. The following expression is the definition of convolution between two functions $\boldsymbol{f}$ and $\boldsymbol{g}$:

$$(f * g)[n] = \sum_{k=0}^{N-1} f[k]g[n-k]$$

Equation 1: Convolution equation

In general, convolution combines two functions to produce a third function which reflects how one function shapes the other. In our application, we are convolving an input signal with a filter/kernel[2] such that the input signal is modified based on the characteristics of that filter/kernel. Using this operation, we can calculate the effect a guitar cabinet, guitar amplifier, or even the room has on a guitar by finding their kernels—called the impulse response—and convolving them with the guitar signal.

Kernels that are created can have many coefficients, which if convolved naively using the expression above can cause large processing delays. As a result, fast algorithms are required which can reduce the number of computations and thus lower processing times. The following sections expand on techniques used to decrease processing time.

4.4.2 The Convolution Theorem and FFT

As described above, convolution in DSP often uses a signal and a kernel as its operands. This signal is measured (sampled) and stored as an array of values which represent its value at each time step. The following is how a signal would be represented in a digital system.

$$x_{\text{signal}}[n] = a_0[n] + a_1[n-1] + \cdots + a_{N-1}[n-N-1] = \sum_{k=0}^{N-1} a_k[n-k]$$

Equation 2: Input frame equation

In the expression above, n denotes a time step, and there are N total samples recorded. Each timestep is multiplied by a coefficient a which represents the value measured at that instant. A kernel would be stored in the same manner:

$$h_{\text{kernal}} = b_0[n] + b_1[n-1] + \cdots + b_{N-1}[n-N-1] = \sum_{k=0}^{N-1} b_k[n-k]$$

Equation 3: IR frame equation

To perform convolution, the digital signal processor would multiply each of the corresponding coefficients with each other and take their sum.

[2] A kernel is an array of coefficients that represent how a certain stimulus can affect the frequency characteristics of another signal.

$$a_0 \cdot b_0 + b_1 \cdot b_1 + \cdots + a_{n-1} \cdot b_{n-1} = \sum_{k=0}^{N-1} a_k \cdot b_k$$

Equation 4: Intermediary Convolution equation step

The operation above must be repeated for every time-step n of which there are a total of N. This means to convolve two arrays with N coefficients, would require $N \times N$ multiplies and $N \times (N-1)$ additions; this cost is referred to as a time complexity of $O(N^2)$. This quickly becomes demanding for kernels with many coefficients and so, we need a technique which reduces the time complexity to a lower degree.

To accomplish this, we take advantage of the Convolution theorem, which states that the Fourier transform of a convolution of two functions (or signals) is the product of their Fourier transforms. By applying the Fast Fourier Transform (FFT) to the signal and the kernel we can perform the $N \times N$ multiplies and $N \times (N-1)$ additions with only N multiplies and $N-1$ additions. There is a slight cost to this, which requires an initial computation cost of $O(N \cdot \log N)$ operations to convert to the frequency domain and convert back to time. But, as N grows, the transform cost becomes negligeable.

The following plot shows how the number of operations increase with respect to the signal length N

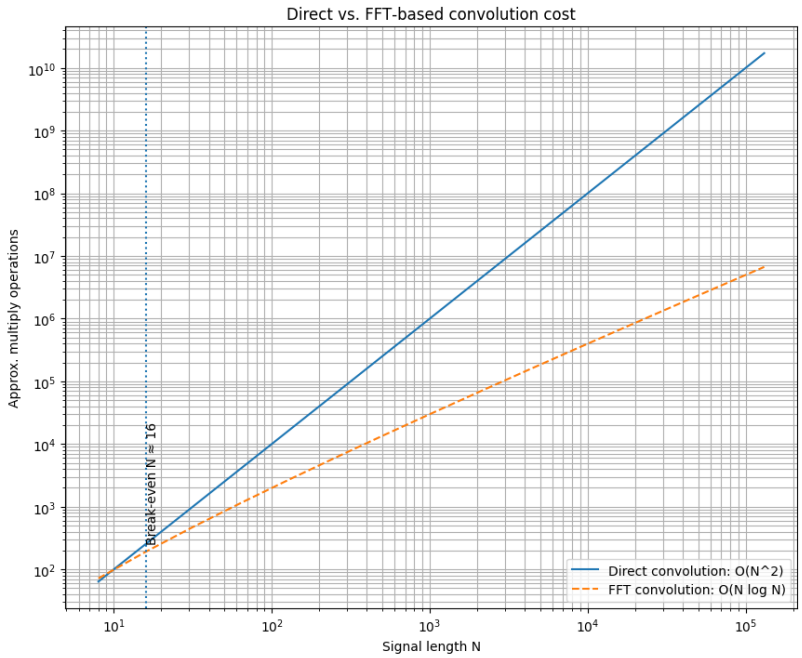


Figure 4: Direct vs FFT-based convolution cost graph

If we say an operation can be performed in 1us, then for $N = 10{,}000$ an FFT based implementation would take about 0.4 seconds, while direct convolution would take 100 seconds. The FFT-based convolution offers an improvement in performance for large values of N.

A necessary requirement for performing this operation without circular convolution[3] is to increase the sizes of the input array and kernel such that their new size is the sum of their original sizes subtracted by one.

[3] Circular convolution will introduce artifacts in the result that are undesirable.

$$L = M + N - 1$$

Equation 5 Convolution Length

In the expression above, M is the size of the input array, N is the size of the kernel, and $\boldsymbol{L}$ is the new size that both must be adjusted to; this is done by placing zeroes in the extra space that is created. This is referred to as zero-padding.

The second requirement is that the resulting array size must be a power of 2. This is important because it is what allows us to use the Fast Fourier Transform to reduce the complexity to $O(N \cdot \log N)$. Thus, L must be adjusted as follows:

$$L = 2^{\lceil \log_2 M+N-1 \rceil}$$

Equation 6: Length of power of 2 convolution

Once these conditions are met, we can successfully perform convolution using the frequency domain.

4.4.3 Uniform-Partitioned FFT (UPFFT) Over-lap Add (OVA) Convolution

In the previous section, we discussed the convolution theorem, which enables faster convolution by transforming operations into the frequency domain. We also addressed the requirements for applying this method correctly, notably the need to zero-pad both the input and kernel arrays. However, this introduces another practical issue: larger arrays increase latency because data acquisition (sampling) occurs at a fixed rate, independent of processor speed.

To illustrate this point, consider the following example: Sampling rate: $F_s = 48{,}000\ \text{Hz}$; Processing speed: $F_p = 500\ \text{MHz}$ (one output every $\frac{1}{500}\mu\text{s} = 2\ \text{ns}$); Array size: $N = 10{,}000$.

The total processing latency is given by:

$$\frac{N}{F_p} + \frac{N}{F_s} = 0.02\ \text{ms} + 20\ \text{ms} = 20.02\ \text{ms}$$

Equation 7: Sample latency calculation

This example demonstrates that despite having a very fast processor, the latency remains dominated by the sampling frequency. Thus, a technique that can reduce the array size and produce the same result is required.

This is what Uniform-Partitioned FFT (UPFFT) Over-lap add (OVA) convolution solves. UPFFT-OLA convolution cuts latency by chopping the impulse response into small partitions—each no larger than the real-time block (e.g., 128 or 256 samples)—and pre-computing their FFTs. When a new input block arrives, it is zero-padded to twice the block length, transformed to the frequency domain, and cached. That spectrum is multiplied by the next stored kernel partition, then inverse-transformed; the first half of the time-domain result becomes the current output, while the second half is added to the next input block to maintain continuity. The saved spectrum is reused with the following kernel partition, so the full convolution is assembled over successive blocks. This approach spreads the workload across many short FFTs, capping latency at one block while preserving the exact long-kernel response.

The following plot shows how the latency increases with as the array size N increases with a UPFFT-OLA block size of 256 compared to a single large FFT.

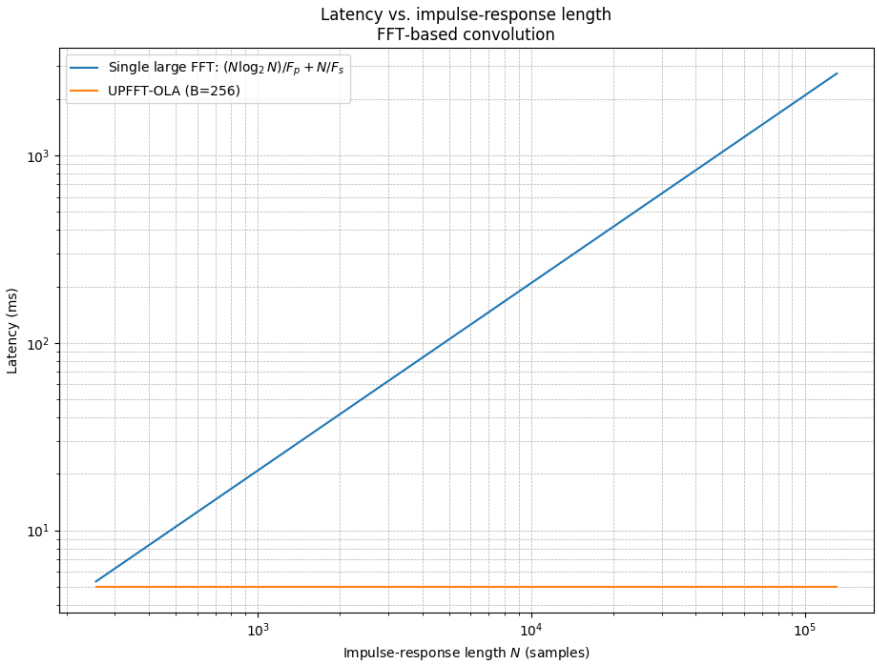


Figure 5: Latency vs Impulse response graph

A trade-off of UPFFT-OLA is that the system must store and pre-compute more values than the naïve implementation and thus uses more memory. But this is often a necessary trade-off to preserve real-time operation.

4.5 System Identification

System identification is the process of developing a mathematical model of a dynamic system purely from measured input–output data. Rather than starting from first-principles physical laws, you treat the system as a "black box" and infer its behavior by observing how it responds to known excitations. The following sections describe, and detail algorithms and techniques used for system identification.

4.5.1 Sine Swept Analysis

Sine-swept analysis is a system-identification technique used mainly for linear-systems that plays a sinusoid whose frequency smoothly changes—typically exponentially—from a low value f_1 to a high value f_2 over a fixed duration T.

$$f(t) = f_1 \cdot \left(\frac{f_2}{f_1}\right)^{\frac{t}{T}}, \quad 0 \le t \le T$$

Equation 8: Sine sweep analysis frequency term

The phase of the sinusoid is then defined as follows:

$$\varphi(t) = 2\pi \int_0^t f(\tau)d\tau = \frac{2\pi f_1 T}{\ln(f_2/f_1)}\left[\left(\frac{f_2}{f_1}\right)^{\left(\frac{t}{T}\right)} - 1\right]$$

Equation 9: Sine sweep phase equation

By recording how the system under test responds to this sweep and then mathematically “de-sweeping” (deconvolving) the recorded signal with the known sweep, we obtain the system’s impulse response in one measurement.

Compared with trying to excite a system with a single, ideal impulse (which in practice is very short, very loud, and difficult to generate cleanly), a sine sweep delivers its energy gradually and evenly across the entire frequency band. Because the sweep dwells longer at low frequencies (where speakers and rooms tend to be less efficient) and spends proportionally less time at high frequencies, you get a more uniform signal-to-noise ratio across the spectrum. The gradual frequency change also keeps peak levels lower—so you avoid overdriving amplifiers or exciting unwanted mechanical resonances.

4.5.2 Levenberg–Marquardt (LM) algorithm

The Levenberg–Marquardt (LM) algorithm is an iterative method for solving non-linear least-squares problems; it blends the fast convergence of the Gauss–Newton method with the robustness of gradient descent. At each iteration k there is a parameter vector x_k and residuals $\mathbf{r}(\mathbf{x}) = \mathbf{y}_{\text{meas}} - \mathbf{f}(\mathbf{x})$. Let $\mathbf{J}_k$ be the Jacobian matrix of $\mathbf{r}$ evaluated at $\mathbf{x}_k$. The LM update step is as follows:

$$\Delta \boldsymbol{x} = (\mathbf{J}_k^T \mathbf{J}_k + \lambda_k \mathbf{I})^{-1} \mathbf{J}_k^T \mathbf{r}(\mathbf{x}_k), \quad \mathbf{x}_{k+1} = \mathbf{x}_k + \Delta \mathbf{x}$$

Equation 10: Levenberg-Marquardt Algorithm

This step is repeated until convergence or until a cost function is reduced below a threshold This algorithm is used for system identification and curve-fitting for non-linear systems and was widely adopted as the main mechanism for characterising the Supro Coronado 1690T.

4.6 Key Literature Summary

The signal processing will be done on the STM32H743 microcontroller programmed in C using contemporary techniques such as those found in . We also are also modelling some amplifiers using block-oriented gray-box modelling and non-linear modelling techniques . We’re also using multipoint iterative matching .

4.6.1 Statista's Musical Instruments – Canada

We used this to evaluate the size and forecast of the Canadian musical instrument market. It provided insight into consumer demand trends and revenue potential, supporting our business case for launching a guitar multi-effects processor within the Canadian market.

4.6.2 Designing Audio Effect Plugins in C++ by Will Pirkle (2019)

This book served as a technical reference for implementing real-time DSP algorithms in a modular architectural style. It also informed the code structure and audio buffer handling used in the embedded firmware of the processor. The book gave a great technical overview of exactly what some of the were doing on and electrical with clipping and filtering allowing us to use that as a basis for building our effects in Matlab.

4.6.3 Gray-box modelling of guitar amplifiers (2017)

This paper was instrumental in the approach we took to modelling the amplifier. We were able to adapt the methods laid out in the paper to gather the data needed to build our amp model and to implement those models on our microcontroller.

4.6.4 M.J Kemp: "Analysis and Simulation of Non-Linear Audio Processes" [4]

The Sintefex paper was used as a reference during the amplifier and cabinet modeling phase. The concepts presented, particularly the technique of analyzing non-linear audio effects by capturing impulse responses at multiple input levels applying them through dynamic convolution. These methods guided our approach to modeling the non-linear behavior of the amplifier and the dynamic response of the cabinet.

4.6.5 Fractal Audio Systems' Multipoint Iterative Matching (2013)

The MIMIC technology paper was referenced during the amplifier modeling phase. Its methodology for capturing the dynamic response characteristics of analog amplifiers and applying signal matching within a DSP framework. This paper provided valuable industry standard perspectives. These concepts directly influenced our modeling the tonal and dynamic behavior of the Supro Coronado 1690T amplifier.

4.6.6 Reiss and McPherson's Audio Effects: Theory, Implementation and Application (2014)

We used this book to implement various effects, including delay, chorus, reverb, and phaser.

This book explores the foundational concepts of digital audio and advanced techniques for implementing a range of audio effects. By covering topics like delay lines, filters, modulation, and distortion. Alongside the concepts from [2], it functioned as a reliable reference for creating audio effects.

4.6.7 "Shimmer" Audio Effect by Zhang (2018)

We used the concepts within as reference material to model effects. Presented at Stanford's CCRMA, showcases the design and functionality of shimmer, often used in ambient music. It provides insight into building a distinct effect by layering reverberated audio with pitch shifting, contributing to unique sonic textures that can add value to a processor's feature set.

4.6.8 Physical Audio Signal Processing by Smith

This book was primarily used as an aid in modelling the Supro amplifier.

This book covers the mathematical principles necessary for implementing accurate physical modeling in audio software. This source informs the computational strategies needed for simulating physical properties of amplifiers and other analog devices within a digital environment, allowing for more faithful modeling of physical circuitry.

Combining these resources provides a robust framework for designing a multi-effects processor and guitar amp modeler that meets both technical and market needs. Integrating advanced DSP techniques, user-favored effects, and realistic amp modeling techniques—tailored to the increasing demand for digital music tools—will likely yield a product that appeals to both amateur musicians and professional artists.

4.6.9 F. Wefers: Partitioned Convolution Algorithm [10]

We used the partitioned convolution algorithm to help model the reverb effect and various cabinets. This book explores the concepts of three different classes of partitioned convolution algorithms for real-time systems. These algorithms (uniform and non-uniform partitioned filters, as well as unpartitioned filters) will act as a base point for our own implementation of partitioned convolution for effects with larger IR responses.

5 Scope of Work

This section outlines the scope of our work for the current capstone year. Our long-term objectives that may extend beyond the present scope will be outlined. We will also highlight the main capstone objectives, as well as our methods of management, execution, and our accomplishments. In addition, we will highlight any external collaborations and inputs to our project.

5.1 Longer-Term Objectives

The long-term objective is to develop a fully functional Multi-FX and Amp modeler, designed with portability and versatility in mind to meet the needs of musicians and producers in both live and studio environments. Therefore, the objectives for the final product should be as follows:

1. Process real-time audio with minimal latency to preserve audio fidelity.
2. Develop an intuitive user interface that enables seamless selection and configuration of audio effects.
3. Implement a comprehensive audio library featuring popular music effects with adjustable parameters.
4. Model a range of amplifiers, from classic analog designs to modern iterations
5. Fabricate a portable, low cost, and robust chassis with an efficient form factor for easy of transport.
6. Integrate Bluetooth connectivity, mobile app compatibility, and internet connectivity for data transfer.
7. Expand the effects library to include more specialized or customizable user-defined effects.

8. Design a customized mixed-signal PCB layout optimized for real-time audio processing.

These longer-term objectives are intended to ensure the product remains relevant and competitive in the evolving music technology market. By integrating modern features such as Bluetooth and internet connectivity, the system will become more versatile and future-proof, positioning it to meet the changing needs of users.

5.2 Capstone Project Objectives

The initial phase of this project focused on developing a functional prototype that showcases the core capabilities of the Multi-FX and Amp modeler. The ultimate objective for this capstone is to deliver a working unit that achieves the following:

1. Implementation of at least three real-time effects and guitar cabinets.
2. Model multiple guitar amplifiers using DSP techniques.
3. Establish serial and parallel processing of multiple effects within the signal chain
4. Design a customized codec PCB to handle analog to digital and digital to analog conversions, including anti-aliasing filtering at the input and reconstruction filtering at the output.

These goals were selected with careful consideration of the available time and resources for this project.

5.3 Project Management, Execution, and Accomplishments

This section provides an overview of project management and execution. Our strategies and tools used for effective planning, team coordination, and task delegation will be outlined. Key milestones and development phases are discussed, along with any challenges encountered and how they were addressed. Finally, the accomplishments achieved during this period are summarized to highlight the progress made towards meeting the project's objectives.

1. A working set of core audio effects, including Chorus, Delay, Phaser, and Reverb.
2. A digital model of the Supro Coronado 1690T amplifier and cabinet integrated into the system.
3. Functioning Serial processing architecture for chaining multiple effects
4. Fabricated a codec PCB for audio input/output conversion with integrated anti-aliasing and reconstruction filters.

Our original milestones were mostly met, although there were a few adaptations needed as the project progressed. The following is a table of milestones and their statuses:

Multi FX and Amp Modeler Milestones		
Task	Status	Completion Date
Codec PCB designed and fabricated	✓	1/30/25
Codec PCB system integration	✗	2/28/25
Effects Test Rig	✓	2/10/25
Effects research	✓	2/10/25
Amp and Cabinet Modelling	✓	2/10/25
Effects implementation into microcontroller	✓	3/10/25
System Pipeline (Serial)	✓	4/20/25
System Pipeline (Parallel)	✓	4/30/25
UI	✓	4/27/25

Table 1: Milestones

The initial proposal outlined the modeling of multiple guitar amplifiers and the full integration of a custom codec PCB. These two deliverables were revised due to unforeseen delays and technical constraints. As development progressed, it became clear that modeling multiple amplifiers would be infeasible within the available timeframe, given the attention required by other critical aspects of the project. As a result, the team chose to model a single amplifier, allowing for a better focus on quality across the rest of the system.

The codec PCB, although successfully designed and fabricated, was not integrated into the final product. Delays in board fabrication caused it to arrive well past the scheduled milestone. Attempting to integrate, test, and adapt the firmware for the board at that stage would have jeopardized the overall project timeline. Therefore, the team chose to populate the board for demonstration purposes only.

To manage our time effectively, we held weekly team meetings to assess progress, update one another, and reassign tasks as needed, along with biweekly meetings with our capstone mentor for guidance and accountability. A Gantt chart was used to outline major milestones and deadlines, helping us track our development phases. Github served as our primary tool for version control and collaboration on code, while Onedrive was used for sharing documentation, schematics, and other project files.

For budget management, we maintained a detailed collection of BOMs (bill of materials) in Excel, which tracked every component purchase and provide an overview of total expenditures. This allowed us to ensure that we remained within our allocated budget.

The following is a sample of our Gantt chart from January 13, 2025 to March 3, 2025:

TASK	ASSIGNED TO	PROGRESS	START	END
Codec	H			
Codec PCB	Hassan Islam	100%	1/13/25	2/3/25
FX Research	FRSH			
Delay and Chorus	Federico Sarabia	100%	1/13/25	2/10/25
Amp Model mapping function	Hassan Islam	100%	1/13/25	2/10/25
Phaser	Federico Sarabia	100%	1/13/25	2/10/25
AMP Model	Hassan Islam	100%	1/13/25	2/10/25
Fx Implementation Format	RSH			
Test Rig	Spencer/Hassan	100%	1/27/25	2/10/25
FX Implementation to uC	FH			
Reverb	Hassan/Spencer/Federico	100%	2/10/25	3/10/25
Amplifier	Hassan Islam	100%	2/10/25	3/10/25
Phaser	Federico Sarabia	100%	2/10/25	3/10/25
Cabinet	Hassan Islam	100%	2/10/25	3/10/25
FX Pipeline and UI	RS			
System Pipeline	Spencer Arnold	100%	2/10/25	3/10/25
UI	Riley Olfert	100%	2/10/25	3/10/25

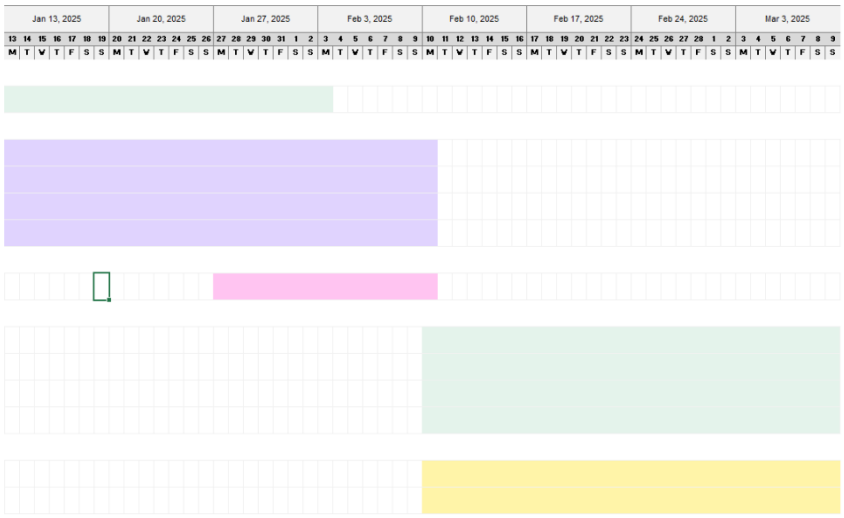


Figure 66: Team Gantt Chart

6 Project Implementation

6.1 High-Level Description

The Multi-FX Guitar Amp Modeler will provide users with the ability to process an **instrument**-level output signal (e.g., from an electric guitar) with customizable effects. Users can apply time-based effects (e.g., reverb) and frequency-based effects (e.g., pitch shifting) via the **Effects Bus**. Each effect includes adjustable parameters that can be tweaked to the user's preference and allows the user to arrange effects in any order. The Multi-FX Guitar Amp Modeler has one instrument level **line-level** balanced output These it to be used in many common scenarios including, live-concert venues, studios, etc. The Multi-FX Guitar Amp Modeler a user with a **LCD Touchscreen User Interface**, which is used to control the Effects bus. Adjustable parameters include the number of effects instantiated, effect types, effect parameters, effect order, and which input/output option to utilize.

The following block diagram illustrates the high-level system architecture of the Multi-FX and Amp Modeler system:

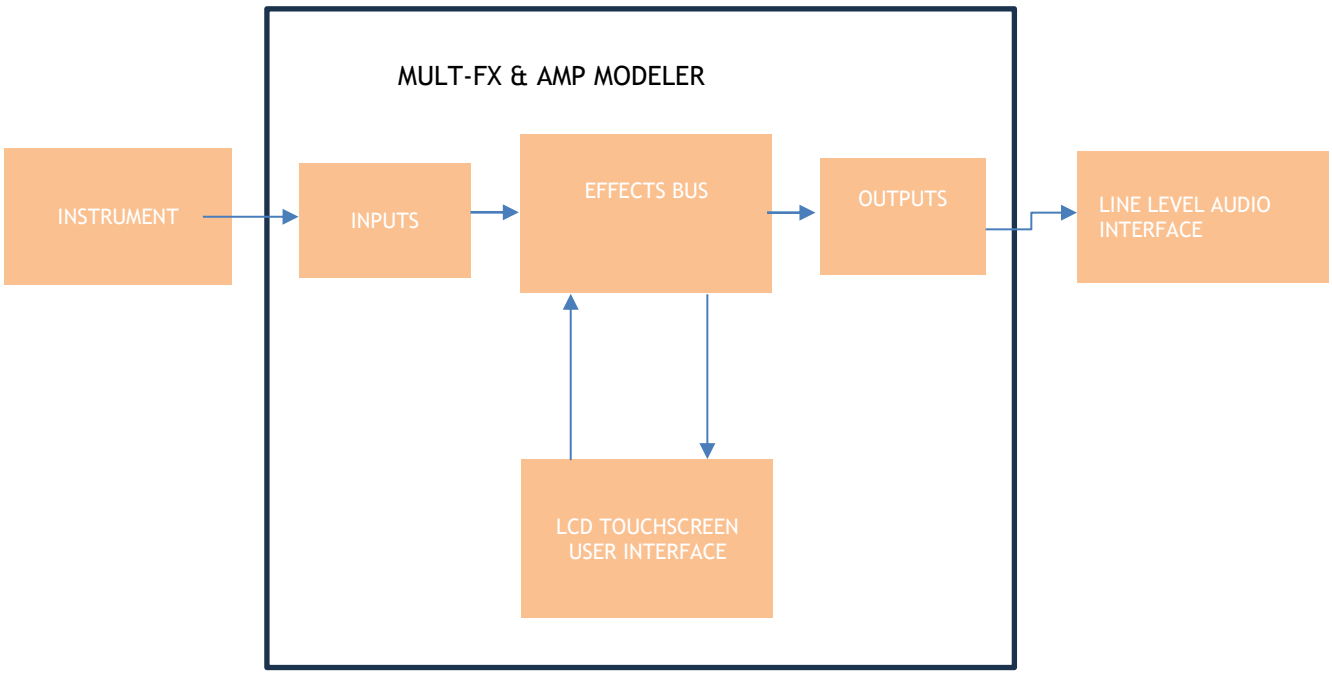


Figure 77: Multi-FX and Amp Modeler Block Diagram

The Effects Bus is the central processing unit for applying various audio effects to the instrument signal. Each box labeled "Effect" represents an individual effect module that can be applied to the signal

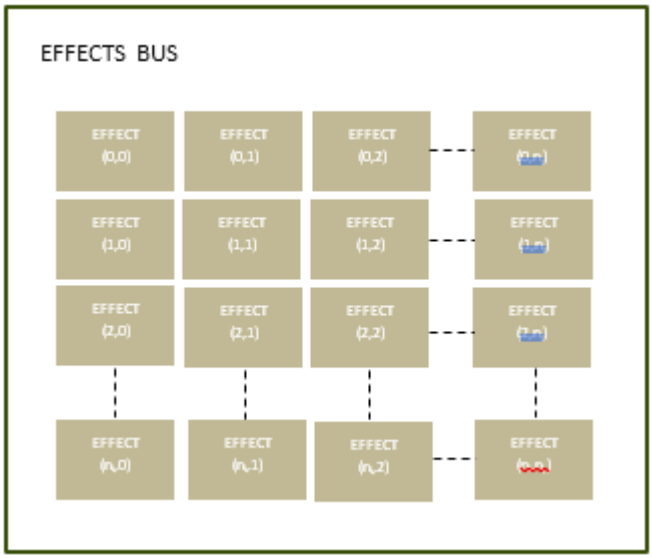


Figure 88: Effects bus diagram

The notation inside each box (e.g., Effect (0,0), Effect (0,1)) indicates the position of the effect in a matrix structure, where each row and column represent a separate processing path. This arrangement allows multiple effects to be configured and processed in sequence or parallel, depending on the desired signal flow. The signal can move across the Effects Bus to apply different effects, and each effect's parameters can be customized by the user. Not all effects blocks need to be utilized.

6.2 Design and Implementation

6.2.1 Latency

In real-time audio processing, latency refers to the delay between the audio input and the corresponding audio output. If this delay exceeds the maximum acceptable latency, then it will become noticeable to the human ear and will negatively impact the responsiveness and usability of the system. To prevent this, all audio processing must be completed within a strict time frame, which is influenced by factors such as sampling rate, buffer sizes, processing, and optimizations.

Research suggests that the upper bound for imperceptible latency in musical performance is approximately 30 ms on average. However, sensitivity varies among individuals, with highly trained or experienced musicians able to detect latencies as low as 20 ms, while less experienced performers may tolerate delays up to 40 ms without perceiving disruption [11].

Therefore, our most critical constraint is maintaining a system latency below 30 milliseconds. Every subsequent design decision, whether related to software architecture, signal processing, hardware selection, or system interfacing, has been made with this constraint in mind.

While 30 ms represents our upper bound for imperceptibility, we aim to minimize latency further and approach the lower bound of 20 ms to ensure a seamless and responsive experience for even the most discerning users.

6.2.2 Memory

Memory is another fundamental constraint in real-time audio processing, directly impacting both the quality and flexibility of the system. Certain effects such as convolutional reverbs and detailed amplifier models rely on large impulse responses (IRs), which require significant memory allocation. Beyond, individual effects, overall memory availability dictates how many effects can be chained simultaneously the complexity of the processing pipeline, and the system's ability to maintain audio fidelity without overlaps or artifacts.

Additionally, the chosen memory allocation scheme, whether static, dynamic, or pooled, affects not only memory usage but also access times. Therefore, careful consideration of our system's memory usage at both the hardware and software levels are critical to achieving low-latency performance and maintaining high audio fidelity.

6.2.3 Hardware Design and Implementation

With latency and memory as key constraints, our hardware selection was driven by the need to meet both demands. We required a system with sufficient processing speed to minimize latency and ample memory to support complex effects and efficient allocation strategies.

During the preliminary design phase, one of our first major decisions was selecting the appropriate processing platform: a microcontroller or an FPGA. Each one presented distinct advantages. FPGAs offered extremely low latency, greater parallelism, and high configurability at the hardware level. However, microcontrollers provided significantly easier development, faster prototyping, and broader support for integrated modules such as ADCs, DACs, and communication interfaces. After evaluation of our project constraints, such as cost, development time, and performance requirements, we opted to use a microcontroller. A microcontroller provided a more practical balance of affordability, simplicity, and processing power to meet our latency and memory requirements.

Our microcontroller of choice is the STM32H7, with detailed justifications provided in the following section.

6.2.3.1 Microcontroller Selection: STM32H743

Our microcontroller of choice is the STM32H7 series, specifically the STM32H743, which offers a max clock speed of 480 MHz. While higher-performance microcontrollers do exist, they often come at a significantly higher cost. The STM32H743 provided a well-balanced solution, offering sufficient processing speed to meet our latency requirements while remaining cost-effective. In addition to its speed, it includes up to 1MB of internal RAM, 2MB of flash memory, dual DACs, high-speed ADCs, extensive GPIO availability, and external memory support. These integrated features make it well-equipped to handle the latency, memory, and interfacing demands of our system.

6.2.3.2 External Audio Output Amplifier and Reconstruction Filter.

To ensure high-quality output, the system was designed with a dedicated audio amplifier stage coupled with a reconstruction filter. The amplifier provides the drive strength after the DAC output for external load such as headphones or speakers while preserving signal integrity. The reconstruction filter, implemented as a low-pass filter, attenuates the high-frequency components and quantization noise introduced during the digital-to-analog conversion process. This combination was chosen to improve overall audio fidelity, minimizing distortion and producing a cleaner analog signal. Special attention was given to component selection and layout to reduce noise issues.

The following figures show the schematic for the amplifier system and the reconstruction filter, where Vcc is the 12V supply and V_uC is supplied by microcontroller:

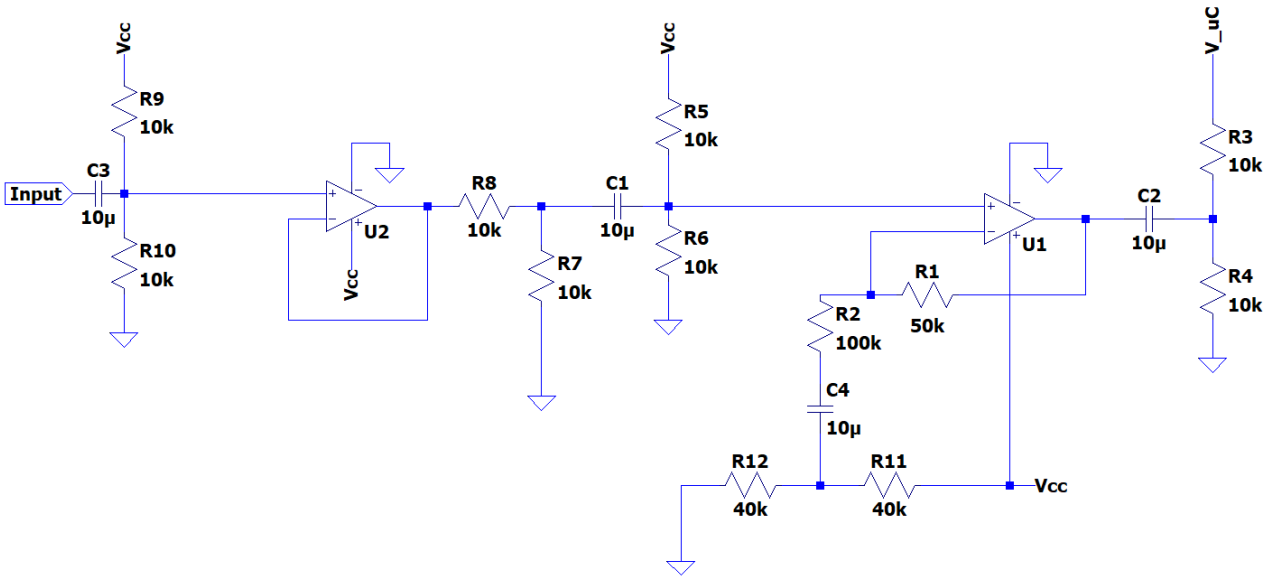


Figure 9: Audio Amplifier Schematic

The schematic of the reconstruction filter can be found in Appendix A.

6.2.3.3 Audio Codec Design and Outcome

At the early stages of development, a customized codec PCB was planned for integration into the final design. This board included its own anti-aliasing filter, ADC, DAC, and reconstruction filter all integrated into the PCM3070 codec processor. This was intended to operate as a standalone module to be interfaced with the main processor. The design phase proceeded on schedule and was completed by the end of December 2024. However, due to delays in fabrication and shipping, the board did not arrive until early April 2025. By that time, development and testing of the effects and amplifier modeling were already well underway. Further attempts to populate, debug, and integrate the codec would have risked significantly delaying the overall project. As a result, the decision was made to populate the board for display purposes only, allowing it to be showcased and potentially used by future teams.

6.2.3.4 User-Interface Hardware

For our user interface hardware, we used an ESP32-8048S070 as these displays are cheap and easy to program. Furthermore, the touchscreen aspect allows an interface most people are familiar with. Priced at about $35 this was a good choice for our project.

6.2.4 Software Design and Implementation

This section provides an overview of our software design, including key considerations and implementation strategies. Topics covered include our simulation approach, programming language selection, software architecture, and reference sources used to implement effects and amplifier models.

6.2.4.1 MATLAB

MATLAB served as our primary platform for testing and simulation during the early stages of the project. It was used to validate the feasibility of our planned audio effects and amplifier models. To simulate a real-time environment, we set a standard of segmenting audio files into frames, processing each frame with the effect algorithm, and playing each frame back in real time, while the fully processed output was also saved for reference. This simulation standard allowed us to test algorithm performance, assess audio quality, and identify which effects were viable for implementation in the final product.

6.2.4.2 C Programming Language

Choosing a programming language was a critical decision in the development process. Early in the project we considered three options: C, C++, and Python, each offering its own strengths. Python stood out for its ease of use and extensive library support, making it ideal for rapid prototyping and high-level development. However, it was ruled out due to its lack of low-level hardware control and the high overhead associated with running Python in embedded environments. C++ provided support for object-oriented programming, including classes, inheritance, and polymorphism, which could have benefited the modularity and scalability of our code. But, C++ also introduces large overhead, memory footprints, and increased complexity.

C, on the other hand, offered a well-understood, lightweight, and efficient foundation for real-time embedded systems. While it lacks object-oriented features of C++ and the vast libraries of Python, we found that we could still achieve modular design using structs, pointers, and careful abstraction. This approach allowed us to mimic object-oriented behavior without additional overhead. Furthermore, while not as extensive as Python, there are still libraries available in C.

Ultimately, C was chosen for its balance of performance, familiarity, low-level hardware access, and the ability to implement structured and scalable code without large overhead.

6.2.5 Effects Simulation

We chose four effects to simulate during the feasibility phase: chorus, delay, phaser, and reverb. Each algorithm was implemented based on the concepts outline in [6]. Each effect was selected with future scalability in mind. Delay was implemented first due to its fundamental nature and role as a building block for more complex effects such as chorus and phaser. Reverb was chosen next, as it shares similar memory and processing requirements as amplifier modeling, particularly in terms of convolution between impulse responses and input signals. MATLAB was used to simulate these effects following a standardized real-time simulation framework we developed:

1. The input audio file was segmented into frames of size N
2. Each frame was processed using the corresponding effect algorithm
3. Every effect parameter including frame size, sampling rate, and mix must be variable
4. Each processed frame was played back in real time upon completion

This allowed us to evaluate the real-time feasibility of each algorithm with respect latency, memory usage, and audio fidelity. By the end of the simulation phase, we had functional MATLAB implementations for each effect, which served as a reference models for their subsequent C-based equivalent.

6.2.5.1 Delay

Delay is a fundamental audio effect with a wide range of applications. At its core, a delay plays back an audio signal after a specified period. In its simplest form, delay can be implemented using the following expression, where $\boldsymbol{n}$ is the current sample and $\boldsymbol{N}$ is the delay length in samples:

$$y[n] = x[n] + g * x[n - N]$$

Equation 11: Delay Formula

Here, $\boldsymbol{y}[\boldsymbol{n}]$ is the output signal, $\boldsymbol{x}[\boldsymbol{n}]$ is the input signal, and $\boldsymbol{g}$ is the gain applied to the delay signal. This is known as a feedforward delay, where the current output depends on the input signal and a delayed version of it. The following is a block diagram representation of the formula.

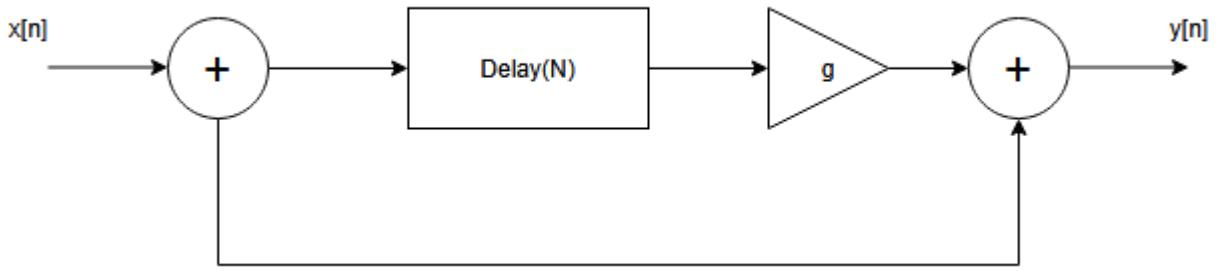


Figure 810: Feedforward Delay Block Diagram

A more commonly used variation is the **feedback delay**, which incorporates a previous (specified by **N**) output into the current output. In conjunction with the feedback, this creates echoes that gradually decay. The feedback delay is described by the following equation:

$$y[n] = g_{FB}y[n-N] + x[n] + (g_{FF} - g_{FB})x[n-N]$$

Equation 12: Feedback Delay Formula

In this configuration:

- $\boldsymbol{g_{FB}}$ is the feedback gain
- $\boldsymbol{g_{FF}}$ is the feedforward gain
- $\boldsymbol{y[n-N]}$ is the delayed output
- $\boldsymbol{x[n-N]}$ is the delayed input

The following is the block diagram representation of the feedback delay system:

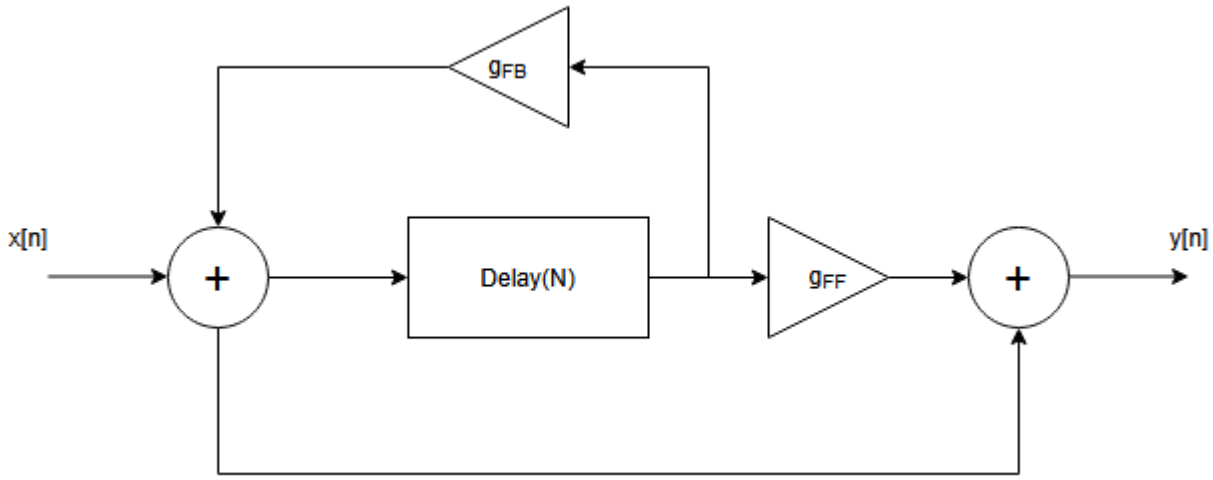


Figure 911: Feedback Delay Block Diagram

As will be seen in the Chorus and Phaser sections, delay serves as a fundamental building block for both effects.

6.2.5.2 Chorus

Chorus in and audio effect that simulates the sound of multiple instruments or voices playing the same part in using but with slight variations in timing and pitch. It works by mixing the original signal with one or more delayed and slightly pitch-modulated copies. This creates the sense of multiple performers playing together, resulting in a richer and more spatially dynamic sound.

To recreate this effect digitally, the system uses a modulated delay line controlled by a **Low Frequency Oscillator (LFO),** as shown in the diagram. The LFO varies the delay time $\boldsymbol{M(n)}$ in real time with a sine wave, causing the delay the fluctuate smoothly above and below a central value. The depth parameter controls the frequency range of the LFO, while **rate** adjusts how often the LFO frequency varies. Together, depth and rate determine the extend and speed of the delay modulation, which produces the subtle pitch variations of the chorus effect. A **mix factor** $\boldsymbol{G}$ is applied to the delayed signal before it is mixed with the dry input. The mix parameter determines the balance between the dry and the modulated (wet) signals, affecting how prominently the chorus effect is heard in the output $\boldsymbol{y[n]}$.

The following formula and block diagram illustrates the implementation of Chorus. The delay block is the feedback delay outlined in the previous section.

$$y[n] = x[n] + G * x[n - M(n)]$$

Equation 13: Chorus Formula

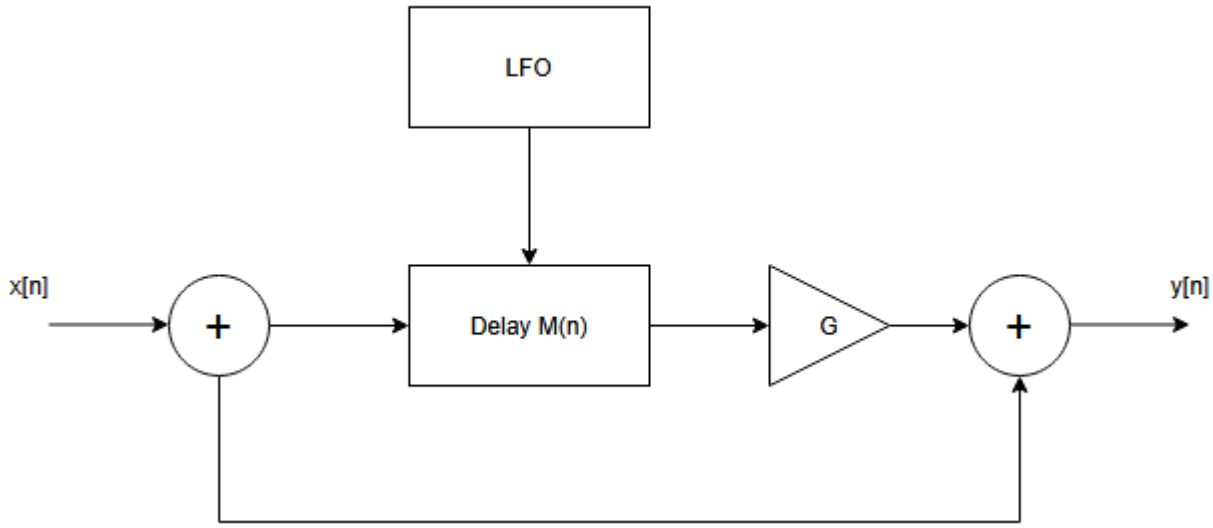


Figure 1012: Chorus Block Diagram

While there are variations of chorus that attenuate certain harmonics or alter phase relationships to achieve smoother sounds, this implementation was chosen for its simplicity and ease of implementation.

6.2.5.3 Phaser

Phaser is an audio effect that creates a sweeping, swirling sound by introducing phase shifts to the input signal. It works by passing the input signal through multiple **all-pass filters**, which introduces a change in the signal's phase without altering the magnitude. These phase-shifted signals are mixed with the original input, creating frequency-dependent constructive and destructive interference, producing notches in the spectrum that move over time.

The following block diagram and equation illustrates the first order all-pass filter:

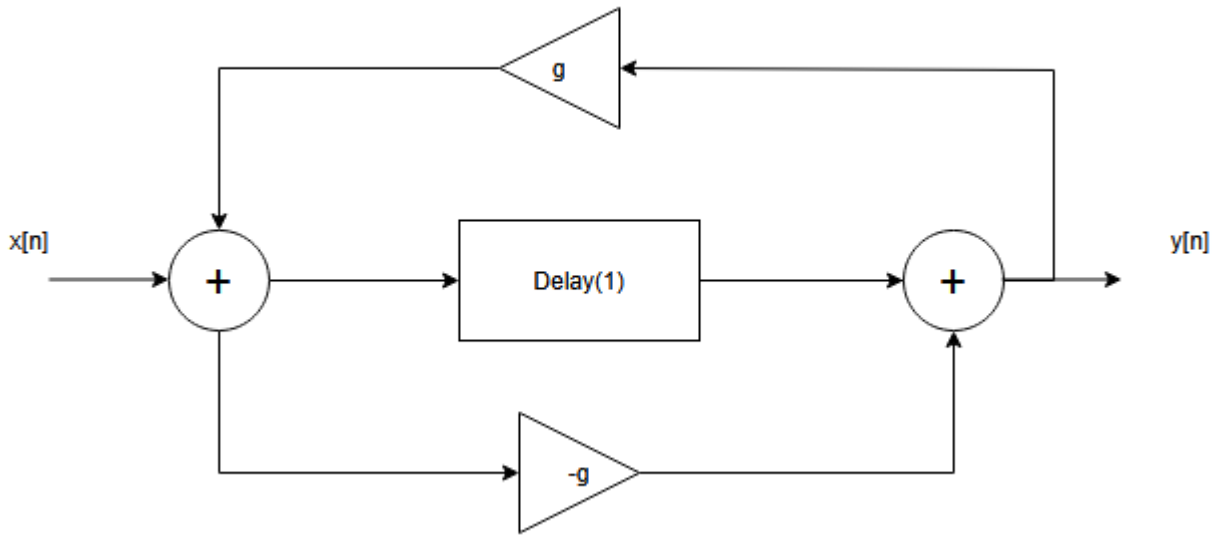


Figure 1113: All Pass Filter Block Diagram

$$y[n] = x[n-1] - g * x[n] + g * y[n-1]$$

Equation 14: All-Pass Filter Equation

In this implementation, the all-pass filter was constructed using a delay-based structure with a fixed one-sample delay. Even a single sample delay introduces frequency-dependent phase shifts, which creates notches in a phaser's spectrum. By chaining multiple delay blocks, referred to as **stages**, the overall phase shift is enhanced, resulting in deeper and more pronounced notches. The position and intensity of these notches are controlled by the parameter **g** which is influenced by the **rate** and **depth** of the Low Frequency Oscillator (LFO) (See Section 6.2.5.2 Chorus for an explanation of rate and depth).

The following is the block diagram for the implementation of Phaser.

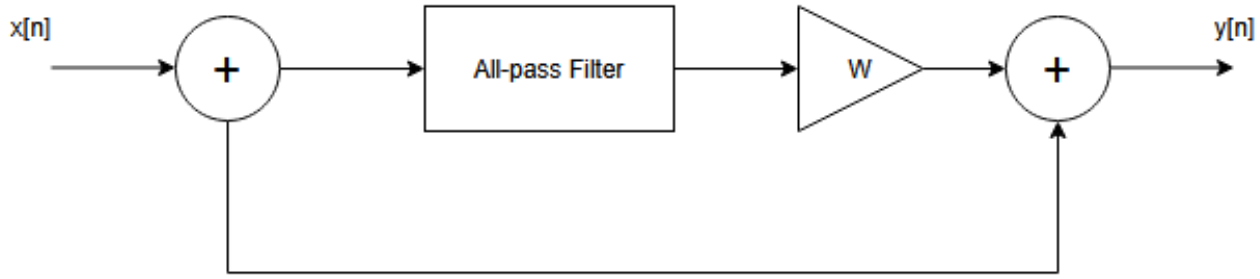


Figure 1214: Phaser Block Diagram

In this implementation, the **Weight** (W) parameter is used to control the intensity of the notches created by the phaser. When the original input signal is summed with the phase shifted signal from the all-pass filter, W determines how strongly the two signals interact. By adjusting this parameter, the user can emphasize or soften the notches in the frequency spectrum, allowing for customization of the phaser's tonal character.

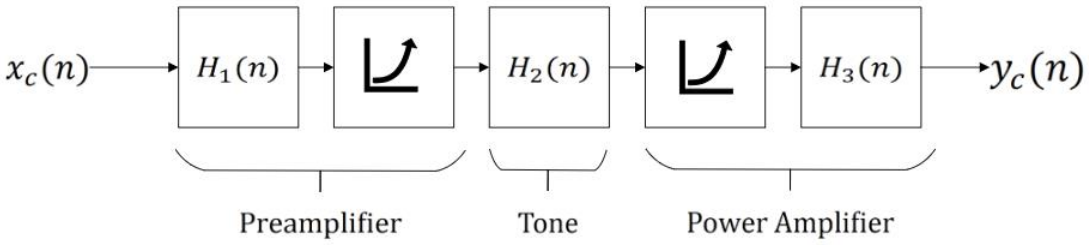
6.2.5.4 Reverb

Reverb is an audio effect that simulates the sound profile of a space (i.e. A concert venue hall). In audio processing, reverb adds a sense of space, depth, and realism by replicating how sound behaves in rooms, halls, or other acoustic environments. It is typically achieved by convolving the dry signal with an **impulse response (IR)**, which is a recording that characterizes how a specific space responds to a sharp sound.

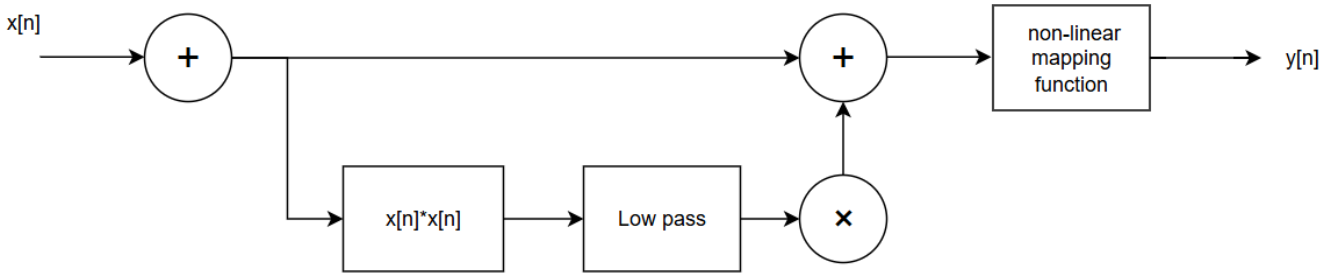
However, convolution with long IRs is computationally expensive, especially in real-time applications. A direct time-domain convolution has a complexity of $\boldsymbol{O(N^2)}$, where N is the length of the IR and the input signal. To address this, we used Uniform Partitioned FFT Overlap-Add method (refer to section 4.4.3).

6.2.6 Amplifier modeling

To implement and capture the linear and non-linear characteristics of the Supro Coronado 1690 T amplifier, we used a Block-oriented gray-box modeling approach. This technique offers a pragmatic bridge between detailed circuit emulation and purely data-driven techniques. In the method presented, a guitar-amplifier signal path is decomposed into repeating linear–nonlinear pairs: an input filter $H_1(n)$, a pre-amp waveshaper, a tone-control filter $H_2(n)$, a power-amp waveshaper, and an output transformer filter $H_3(n)$.



Linear sections are realized as FIR responses, while nonlinearities use either a polynomial mapping function with envelope-based bias shifting (pre-amp) or asymmetrically tuned hyperbolic tangents (power-amp). For the preamp, we are restricting the polynomial to the first forty harmonics and providing adjustable pre/post gains plus dry-wet mixing which keeps computation light yet expressive enough for a wide range of amplifier voices.



For the linear section, two exponential sine sweeps (low-level and high-level) isolate small-signal behavior and the portion suppressed by clipping, yielding initial input and output filters. The exponential sine sweep is defined such with a starting frequency $f_1 = 20\ \text{Hz}$ and ending frequency $f_2 = 20{,}000\ \text{Hz}$. These frequencies were chosen as they are near the limits of the human hearing range.

A 256-tap FIR in the tone stack is then refined with Levenberg–Marquardt to match the residual frequency response. With the linear blocks frozen, a staged optimization tunes nonlinear parameters: envelope matching sets coarse gains; single-tone, multi-tone, and finally recorded guitar passages drive successive passes that minimize a spectrogram-based error weighted by the inverted threshold of hearing. This gradual tightening of the cost function helps the solver avoid local minima and converge on perceptually plausible parameters.

6.2.6.1 User-interface

To do our user interface, we used C++, Arduino Libraries, and PlatformIO. This made for a speedy development and makes our touchscreen easy to program as the availability of graphics libraries makes the development easy. To interface between the touchscreen and the microcontroller that processes the signal, we had the touchscreen transmit a shared list of nodes that are stored as a directed acyclic graph on the touchscreen's ESP32 microcontroller. The following snippet of code shows how each node in the graph gets stored. We store the nodes in a C++ vector of nodes.

```
struct Block {
        uint16_t              id;          // unique (>0 & <0xFFFF); Input=0, Output=0xFFFF
        NodeType              type;
        int16_t               x, y;        // *top-left* corner (keep integer)
        uint16_t              w, h;        // dimensions (80×40)
        std::vector<uint16_t> outs;        // adjacency list (dst block IDs)
        std::vector<String> params;
    };
```

To keep it acyclic a cycle guard is implemented (see snippet below) that runs a depth-first search every time a someone tries to connect two blocks and if the search can reach the proposed source node again we reject the edge and cycles never appear.

```
bool createsCycle(uint16_t from, uint16_t to) {
        std::vector<uint16_t> stack{ to }, seen;
        while (!stack.empty()) {
            uint16_t cur = stack.back(); stack.pop_back();
            if (cur == from) return true;
            if (std::find(seen.begin(), seen.end(), cur) != seen.end()) continue;
            seen.push_back(cur);
            const Block* b = findBlock(cur);
            if (!b) continue;
            stack.insert(stack.end(), b->outs.begin(), b->outs.end());
        }
        return false;
    }
```

We use this in our edge constructor by checking if an edge is already present then making an edge only if createsCycle indicates it to be safe.

```
void connectBlocks(uint16_t from, uint16_t to) {
        Block* src = findBlock(from);
        Block* dst = findBlock(to);
        if (!src || !dst) return;
```

```
    // already wired?
    if (std::find(src->outs.begin(), src->outs.end(), to) != src->outs.end()) return;

    // no loops allowed
    if (createsCycle(from, to)) return;

    // ── NEW fan-in guard ──────────────────────────────────────
    int limit = 1;
    //if (dst->type == NodeType::Mixer)   limit = 2;
    if (dst->type == NodeType::Input)  return;     // never feed the Input block

    if (incomingCount(to) >= limit) return;             // refuse, already full
    // ──────────────────────────────────────────────────────────

    src->outs.push_back(to);
}
```

To draw on the touchscreen we used Arduino graphics library functions such as drawLine, fillCircle, fillScreen, etc and made our own display library based on these more primitive functions to draw complex shapes.

In order to communicate between the touchscreen and the STM board we used UART to communicate both ways using a common protocol. We would send each node line by line, for example if we sent "N2;E2;P3,50,50;I254" this would show node 2 as being an Amp with parameters 3, 50, and 50, and being sent as the output signal. Additionally, the STM is also able to tell the touchscreen when the configuration uses too much memory and this is displayed for the user when this happens.

6.2.6.2 System pipeline architecture

The original vision of the product included both serial and parallel processing of effects and amplifiers. But the complexity of building a high-quality and efficient pipeline proved greater than expected. As a result, the design was revised to create a streamlined serial effects bus to prioritize audio fidelity and system performance.

Our system pipeline was built using a system of interconnected nodes in a graph structure assembled by the user on a connected touch screen. We built a string representing the nodes, the assigned effect, any parameters the effect takes, and inputs to the node. This string was then sent over UART to our processor board and decoded.

The user configuration is stored as a directed acyclic graph data structure, meaning that all node connections had a strict directionality, output to input, no internal loops could be made, and the nodes could connect in a mesh structure. Since we had this specific type of graph structure, we were able to implement a variation on Kahn's sorting algorithm to determine processing order. This was accomplished by performing a depth first topological sort, which involved looking for the processing nodes with no input connections when the system inputs were ignored and adding them to the stack. We then add any nodes found in the first pass to the ignore mask and repeat the process. Once the process is done, we have an order that ensures that all the nodes will be processed in an order so that every node has its respective inputs processed before they're required.

Once we have the effect order we can build our nodes using our effects handlers. When the effect handlers are initialized and given an effect, several state buffers based on the effect called are allocated and the parameters are passed. The effect nodes are then called in the precalculated order until the display sends a new configuration which would restart the node routing and initialization.

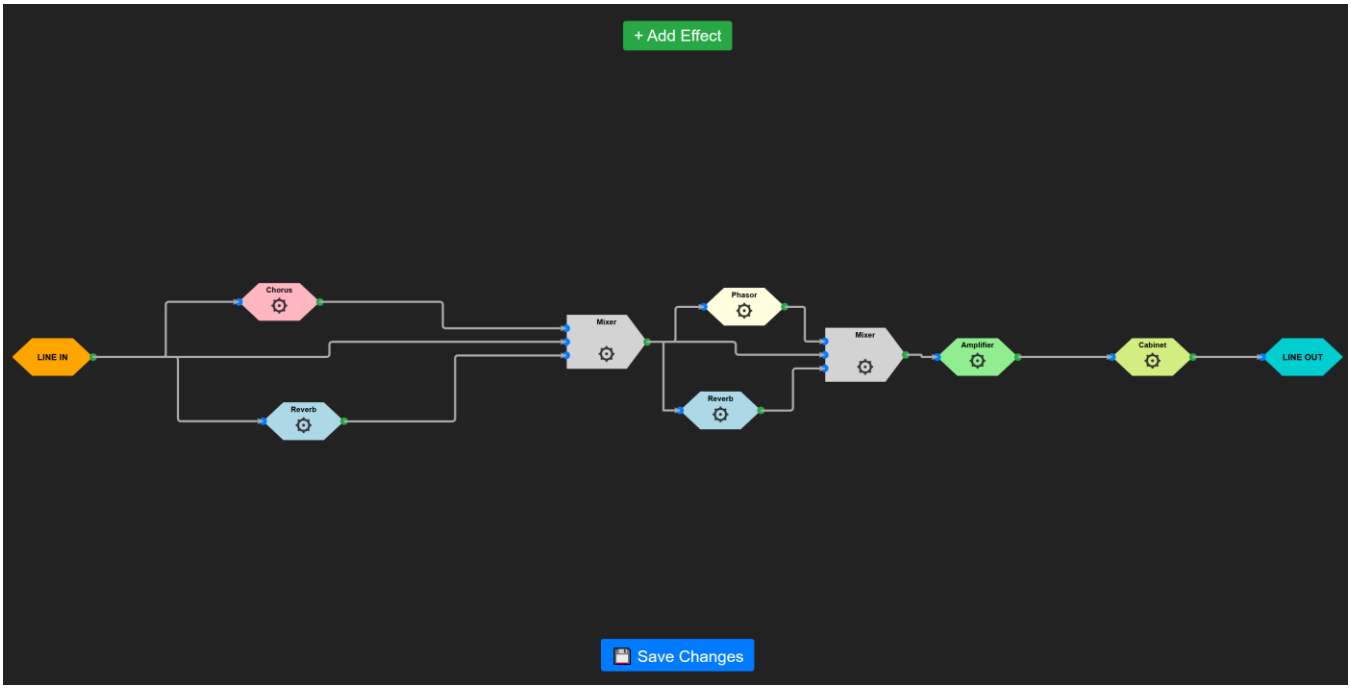


Figure 15: Node Pipeline Example

The overall node pipeline then calls the FX handler in the precalculated order to process the effects, then stores the processed audio to be retrieved by later nodes.

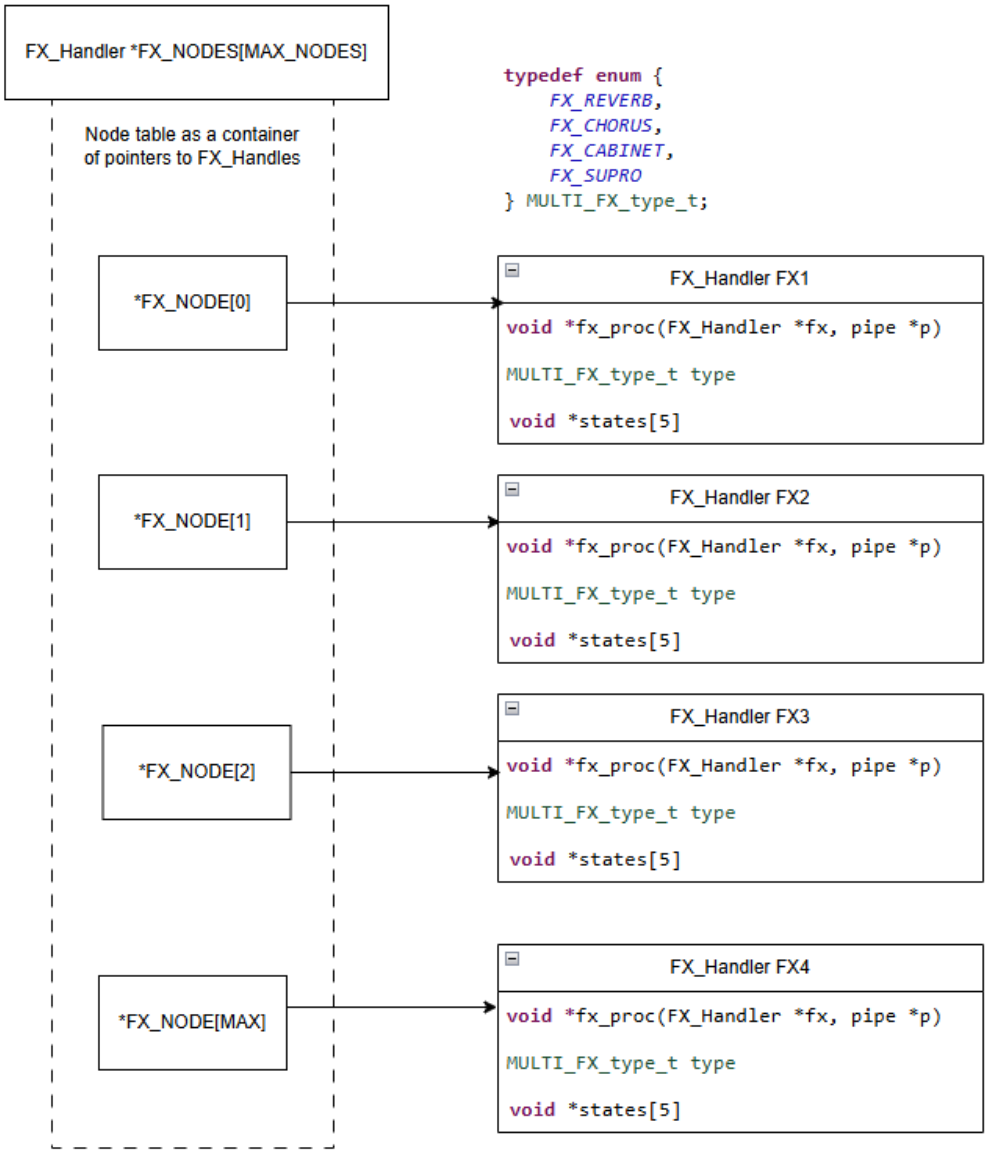


Figure 16: FX Handler Pipeline

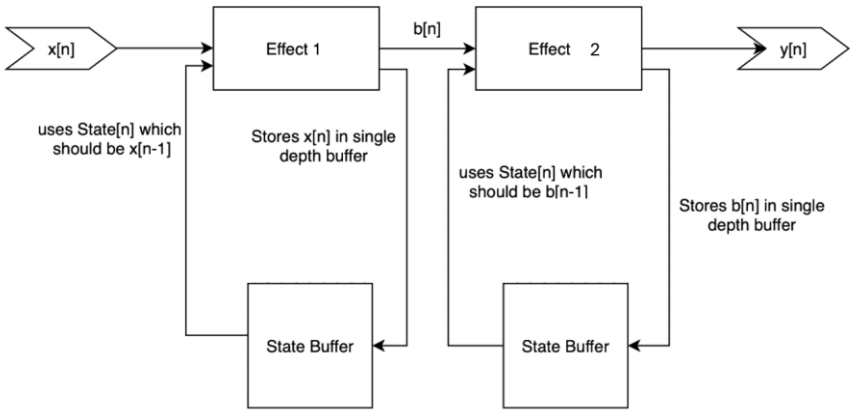
6.2.6.3 Effects Handlers

The effects handler (`FX_HANDLER_t`) is a lightweight, generic wrapper that encapsulates each audio effect's state pointers, parameter count, and two callback hooks—init and process—so that new effects (reverb, cabinet sim, Supro model, etc.) can be instantiated and chained at runtime without changing the core engine. At startup, each handler calls its init function to grab its pre-assigned buffers from the static memory pools and configure any algorithm-specific structures (FIR filters, biquads, lookup tables). During the audio callback, the engine iterates through an array of active `FX_HANDLER_t` objects in a fixed sequence, invoking each handler's `process(self, &pipe)` method so that buffers flow from one effect to the next. This modular design decouples memory management and scheduling from DSP code and makes it trivial to add, remove, or reorder effects. The following code-snippet shows the `FX_HANDLER_t` struct definitions and an example effect struct.

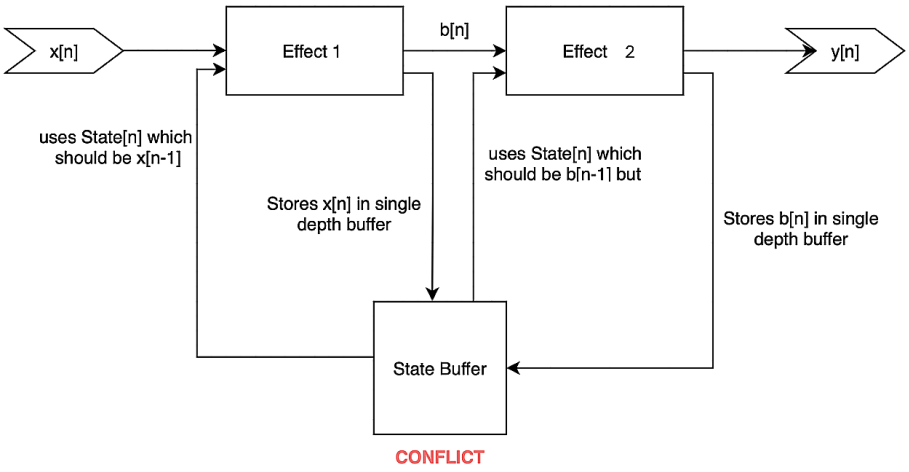
```
//-----------------------------------------------------------------------------
// Generic FX handler
//-----------------------------------------------------------------------------

typedef struct FX_HANDLER_t {
    void*    states[MAX_FX_STATES];        // pointers into static memory pools
    uint32_t num_params;                   // number of tunable parameters
    void   (*init)(struct FX_HANDLER_t*);  // allocate & configure state at startup
    void   (*process)(struct FX_HANDLER_t*, struct pipe*); // per-block DSP
} FX_HANDLER_t;



//-----------------------------------------------------------------------------
// Three-stage Supro tube-amp model
//-----------------------------------------------------------------------------
typedef struct {
    float*            state;    // envelope follower & scratch buffers
    fir_t*            fir1;     // preamp FIR (if used)
    fir_t*            fir2;     // mid-stage FIR (if used)
    fir_t*            fir3;     // power-amp FIR (if used)
    void (*process)(FX_HANDLER_t*, struct pipe*);
} supro_simulation_f32;
```

Each effect module maintains its own state buffers and internal variables because each algorithm relies on history, filter memory, or dynamic parameters that must persist independently between audio blocks: the convolution reverb needs per-partition overlap buffers and FFT scratch space to accumulate and add delayed reflections without cross-talk; the cabinet simulator holds a separate impulse-response pointer and, if using FIR convolution, its own delay-line and coefficient memory; and the Supro model tracks envelope-follower history, biquad filter states, and nonlinear saturation internal registers for each of its three stages. By giving each `FX_HANDLER_t` its own state pointers into the static pools, the engine ensures that time-domain continuity, frequency-domain overlap, and dynamic feedback all remain localized to each effect—preventing data corruption, avoiding unintended coupling between modules, and guaranteeing deterministic, real-time performance.

The following block diagram shows two effects with their own state buffers. These two effects do not collide as they do not access the history of another effect.



The following block diagram shows a conflict condition if effects were sharing the same memory location for their state buffers. The history for each effect would be corrupted by other effects in the signal chain and result in the incorrect output.



6.2.6.4 System Memory Management

The system uses two large, statically allocated memory pools—one in fast on-chip DTCM for audio buffers and one in external SRAM for FFT scratch space—and services all runtime allocation requests via simple bump-pointer allocators, thus combining the flexibility of “dynamic” allocation with the predictability of static allocation: at startup each pool’s head pointer is reset to zero, and each call to:`_static_mem_alloc(size, align)` or `_dctm_static_mem_alloc(size, align)` computes an aligned offset, checks against the fixed pool size, advances the head by the requested size, and returns a pointer into the preallocated array; because there is no `free()` function common in dynamic allocation methods like `malloc()` and `calloc()`, there is zero fragmentation, every allocation costs only an integer alignment and bounds check O(1) time, alignment requirements are guaranteed, and overflow simply traps in a minimal error handler—yet callers can still request buffers at run time without any heap overhead or unpredictability.

6.2.6.5 Cost Summary

The total project cost was $557.12, with the majority of expenses driven by hardware required to support individual development and testing. To ensure efficiency and avoid delays, each team member was equipped with their own setup to prevent hardware contention during implementation.

The largest costs in the project came from the following purchases:

1. Custom PCB fabrication and shipping - $133.92
2. 2x NUCLEO-144 STM32H7S3L8 development boards with STLINK Programmer - $119.57
3. 3x STM32H743IIT6 Development board - $ 105.45

These purchases allowed for streamlined development, minimized delays, and uninterrupted workflow.

6.3 Results, Testing, and Verification

6.3.1 Proof of concept: Reverb in ELEX 7820

Reverb in C was developed as the final project for the ELEX 7820 course, it served as a strong proof of concept for our design ideology. Numerous issues were encountered during development, including memory limitations, long impulse response (IR) handling, timing delays, latency issues, and degraded sound quality. These challenges forced us to explore more advanced and efficient processing techniques.

To manage the long IR required for reverb, we implemented partitioned convolution, which breaks the IR into smaller segments to reduce memory and processing load. Additionally, we explored both Overlap-Add and Overlap-Save for our block-based convolution. We settled on Overlap-Add due to its simplicity over Overlap-Save.

We also adopted loop unrolling to reduce execution time and improve performance. To stay within memory limits, we segmented and manually allocated memory buffers rather than relying on dynamic allocation, ensuring our reverb implementation remained real-time.

This was implemented on the Texas Instruments TMS320F8379D, a high-performance microcontroller from the C2000 series. This phase of development exposed us to significant performance bottlenecks and provided valuable insight into what we may encounter in the future. The solutions and optimizations discovered here were later applied to the final product.

6.3.2 Effects implementation in C

The MATLAB simulations proved highly effective. Prototyping each effect in MATLAB allowed for rapid iteration and testing, with most algorithms directly translatable into C with minimal changes and functional issues. This significantly accelerated transition from simulation to real-time embedded implementation.

However, several challenges were encountered during the conversion process. First, loop optimizations were necessary to meet our two constraints: latency and memory, especially for large buffers or repetitive operations. Techniques such as loop unrolling were applied to reduce instruction overhead and execution speed.

Second memory access and storage optimizations were critical. The team needed to create a memory implementation that allowed for multiple copies of an effect to be created without overlapping parameters while minimizing memory footprint. Therefore, our solution was to create a stack buffer where parameters can be stored with known addresses, allowing for deliberate memory usage and quick access.

6.3.3 Product Testing

Product testing was done throughout the project timeline.

6.3.3.1 MATLAB Testing

The first phase of product testing involved simulating each core component, audio effects, amplifier model, and cabinet model within MATLAB. The primary goal was to verify that all selected components could be realistically and accurately modelled in MATLAB before moving towards a real-time implementation.

All four audio effects, delay, chorus, phaser, and reverb were individually implemented using the algorithms and structures outlined in sections **6.2.5.1** through **6.2.5.4** and **6.2.6.** Each effect was tested with variable user parameters (e.g. feedback gain, LFO rate, and wet/dry mix) to ensure that the expected behavior was met and the effects remained stable. These simulations confirmed the validity of each algorithm and ensured

high audio fidelity. A live guitar audio signal was used as the test input for all simulations, providing a realistic signal source to evaluate audio quality.

The Supro Coronado 1690T amplifier and cabinet were modeled using a block-oriented gray-box modeling approach (Section 6.2.5.5). The MATLAB testing used the same framework as the effects but with a stronger emphasis on audio accuracy. The same live guitar input was used to test the amplifier and cabinet models. Particular attention was given to how the system responded to changes in dynamics, playing technique, and tone to ensure the modeled output remained faithful to the characteristics

The resulting audio files were saved and later used as reference material during the real-time implementation verification stage, ensuring consistency across simulation and real-time audio.

6.3.3.2 System Pipeline Testing and Implementation

During this phase of testing, the focus was on verifying the integrity of the real-time processing pipeline. Initial tests were conducted using a simple sine wave as the input signal, with no effects applied. This allowed for confirmation that the audio data could be fed through the pipeline properly without introducing artifacts, latency, or instability.

Once basic functionality was established, each effect was implemented into the pipeline one by one. This approach ensured that any issues could be isolated to a specific stage or module.

A significant challenge came up when integrating reverb and cabinet simulation stages, both of which rely on convolution with long impulse responses (IRs). These components placed a high demand on system memory, exceeding the limits of the available resources. The issue was resolved by limited the number of active IR and cabinet profiles available, thus reducing memory usage. Furthermore, the system uses two statically allocated memory pools, one in fast on-chip DTCM for audio buffers and one in external SRAM for FFT scratch space, managed by bump pointer allocators. This provided the flexibility of dynamic allocation with the predictability and efficiency of static memory, eliminating fragmentation and ensuring constant-time allocation without heap overhead.

6.3.4 Product Verification

The fully integrated system was verified through live signal testing. Each effect was individually tested using a live guitar input and compared directly against its MATLAB simulation counterpart to ensure consistent behavior and accuracy. Parameters such as modulation rate, mix, and gain were adjusted to confirm that embedded implementation properly reproduced the simulated results.

Following individual verification, effects were chained together in various orders with different parameter combinations to evaluate the system's robustness under real-time usage. This helped identify issues such as unintentional clipping, distortion, noise, or other audio artifacts introduced by hardware and software interactions.

Two notable noise-related issues were encountered and resolved during this phase:

1. Software Noise Suppression: Noise was observed in the output when no audio input was present. To address this, a signal power threshold was implemented in software. For each incoming sample block, the average signal power was calculated by taking the square root of the sum of squares of the samples and dividing it by the total number of samples. If this value fell below a threshold, the entire block was zeroed out, effectively silencing low-level background when no input signal was present.

2. Hardware Grounding Issue: Unwanted electrical noise was present in the output even with the Software Noise Suppression. The source was trace to an ungrounded USB port, which was acting as an antenna and picking up electrical interference. Properly grounding the unused USB input resolved the issue

Presenting quantitative data for audio quality remains inherently difficult. Unlike electrical or timing characteristics, audio effects and tonal qualities are perceptual in nature and rely heavily on subjective listening and comparison. The complexity of sound profiles, particularly when dealing with dynamic input signals like live guitar, makes it challenging to represent the full scope of system performance using conventional plots or numeric summaries. As such, while rigorous testing was performed throughout development, it cannot be fully conveyed in this report due to the subjective nature of audio perception, which can only be meaningfully assessed through listening and is inherently difficult to quantify with objective metrics. Instead, the validation was conducted through live evaluations, comparisons to the reference material, and careful tuning of parameters. Furthermore, with the help of musicians and a team member

holding a diploma in audio engineering, we were able to verify the validity and accuracy of the system through listening tests.

6.4 Use of Engineering Tools

6.4.1 MATLAB

As stated in Section 7.2.4.1, MATLAB was used as our primary simulation and prototyping environment. It was chosen for its versatility, extensive toolset, and ease of use for algorithm development and signal visualization. Throughout the simulation phase, we leveraged both its basic matrix operations and more advance capabilities, such as audio signal plotting, FFT analysis, and frame-based processing/

Although MATLAB offers a wide range of built-in toolboxes and modules for audio DSP, we deliberately avoided relying on high-level functions or black-box modules. Our goal was to implement all algorithms manually using basic operations to maintain a closer resemblance to how they would be eventually written in C.

However, MATLAB introduced some notable limitations during simulation. First, memory usage and performance are dependent on the host's PC's specifications, making it difficult to predict how the system would behave on resource-constrained microcontrollers. Second, MATLAB is highly optimized behind the scenes, using JIT (Just-In-Time) compilation, multithreading, and vectorized execution that are not directly replicable in C on our STM32H743. As a result, many performance bottlenecks emerged later, such as, memory overflow, latency issues, and inefficient loop structures. These had to be identified and resolved during the C implementation phase.

6.4.2 CUBE IDE

After we built the effects in MATLAB to a level that we were comfortable with, we moved on to building the final project in ST Micro's Cube IDE. This development environment was chosen since it provided the most direct support for the microcontroller, we selected in section 6.2.3.1. Since this is the tool built by the manufacturer of our microcontroller, it naturally gave us the most in depth for configuration and debugging. The software is based on Eclipse IDE, the same base software that Code Composer Studio, that we have previously used for course work uses, thereby starting us off with a comfortable level of familiarity.

The provided tools for configuring the hardware abstraction language allowed us to quickly and efficiently set up clock timings and peripherals without needing to directly make register level calls which allowed us to place more of our attention on processing algorithms and data pipelining. This abstraction also made it easier to migrate our code to another board from the same family part way into development when some limitations present on our original choice of microcontroller became evident.

The built-in debug tools were invaluable as they allowed us to directly view memory addresses while the firmware was running which let us diagnose exactly which parts of our processes were not behaving as expected so we could correct it.

6.4.3 Bugera PS1 Power Soak Passive Power Attenuator

In our experimental setup, we employed the Bugera PS1 Power Soak—a passive power attenuator—between the amplifier's speaker-output and the recording interface to permit operation at elevated volume settings while limiting the actual power delivered to the speaker. This approach preserves the amplifier's characteristic tube-driven saturation and dynamic response, ensuring that the nonlinear behavior under high-drive conditions remains authentic.

6.4.4 Focusrite Scarlett 8i6

In our measurement setup, the Scarlett 8i6's dual high-impedance inputs let us simultaneously record both the amplifier's input reference (via a DI feed of the swept-sine excitation) and its speaker output (via a mic or line capture). By driving the amp with a 20 Hz–20 kHz exponential sine sweep and recording at 24-bit/192 kHz with unity preamp gain, we preserved the full dynamic range of both signals—making use of the 8i6's A-weighted noise floor of approximately –128 dBu and >110 dB dynamic range. This ensured that even the lowest-level harmonic and intermodulation components produced by the tube stages remained well above the noise floor.

6.4.5 Digilent Analog Discovery 2

The Analog Discovery 2, or AD2, was an incredibly valuable tool for diagnosing issues and finding system parameters. We utilized the spectrum analyzer to find the noise floor of our analog filtering circuitry and the logic analyzer to simulate and read encoded data over I2C and UART.

6.4.6 LTSpice

Used to simulate designs for analog filtering. This included the reconstruction and anti-aliasing filter.

7 Conclusions and Recommendations for Future Work

With the Multi-FX and Amp Modeler we were able to successfully demonstrate a working proof of concept for a real-time, embedded audio processor tailored to musicians and producers. The system achieved our core Capstone objectives, the implementation of real-time DSP effects, the modeling of a guitar amplifier and cabinet, and the development of a functional touchscreen-based UI with a flexible signal-routing architecture.

Despite challenges such as delayed hardware deliveries and the complexity of real-time audio system integration, the project delivered a modular and scalable platform. The STM32H743 proved capable of handling real-time effects with low latency, and the implemented memory management system ensured deterministic performance without fragmentation. Furthermore, the effect chain's use of a directed acyclic graph (DAG) structure allowed for a dynamic and reconfigurable audio pipeline, opening the door to a broad range of applications beyond those initially implemented.

One of our major accomplishments was the successful modeling of the Supro Coronado 1690T amplifier using block-oriented gray-box techniques. This involved advanced system identification and the implementation of dynamic convolution, demonstrating the system's capacity to replicate nonlinear analog behavior. Effects such as reverb, chorus, delay, and phaser were simulated and implemented using efficient DSP techniques, with careful consideration given to latency, memory, and audio fidelity.

However, several features initially planned, including the integration of the custom codec PCB, could not be fully implemented within the time constraints. We also ran into memory constraints with the number of effects we could process at once and the number of effects, amplifiers, and cabinets.

If we were to continue the project, we could greatly improve the noise floor and audio quality by interfacing with the codec and if we were to add external flash and ram we could increase the number of individual effects saved on the device and the number of active nodes at once. To create more effects, we would benefit from creating an AI based neural processing to speed up the effect model training process.

References

[1] Statista, "Musical Instruments - Canada | Statista Market Forecast," 2024. [Online]. Available: https://www.statista.com/outlook/cmo/toys-hobby/musical-instruments/canada#revenue. [Accessed 01 11 2024].

[2] Line 6, "Line 6 | Helix Family | Guitar Multi-Effects Processor," Line 6, [Online]. Available: https://line6.com/helix/. [Accessed 13 05 2025].

[3] Neural, "Quad Cortex - Neural DSP," Neural, [Online]. Available: https://neuraldsp.com/quad-cortex. [Accessed 13 05 2025].

[4] Fractal Audio Systems, "Axe-Fx II XL+ - Fractal Audio Systems," [Online]. Available: https://www.fractalaudio.com/p-axe-fx-ii-preamp-fx-processor. [Accessed 13 05 2025].

[5] W. Pirkle, Designing Audio Effect Plugins in C++: For AAX, AU, and VST3 with DSP Theory, Focal Press, 2019.

[6] F. Eichas, S. Moller and U. Zolzer, "BLOCK-ORIENTED GRAY BOX MODELING OF GUITAR AMPLIFIERS," 9 09 2017. [Online]. Available: http://dafx17.eca.ed.ac.uk/papers/DAFx17_paper_35.pdf. [Accessed 01 11 2024].

[7] M. J. Kemp, "Analysis and Simulation of Non-Linear Audio Processes," 1999. [Online]. Available: http://www.sintefex.com/docs/appnotes/dynaconv.PDF. [Accessed 01 11 2024].

[8] Fractal Audio Systems, "Multipoint Iterative Matching," 04 2013. [Online]. Available: https://www.fractalaudio.com/downloads/manuals/axe-fx-2/Fractal-Audio-Systems-MIMIC-(tm)-Technology.pdf. [Accessed 01 11 2024].

[9] J. D. Reiss and A. McPherson, Audio Effects: Theory, Implementation and Application, CRC Press, 2014.

[10] J. Zhang, "“Shimmer” Audio Effect:," *Center for Computer Research in Music and Acoustics (CCRMA), Stanford University,* p. 5, 2018.

[11] J. O. SMITH, "PHYSICAL AUDIO SIGNAL PROCESSING," [Online]. Available: https://www.dsprelated.com/freebooks/pasp/. [Accessed 01 11 2024].

[12] Grandview Research, "Guitar Market Size, Share & Trends Analysis Report, 2030," Grandview Research, [Online]. Available: https://www.grandviewresearch.com/industry-analysis/guitar-market-report. [Accessed 01 11 2024].

[13] F. Wefers, "Partitioned convolution algorithm," [Online]. Available: https://publications.rwth-aachen.de/record/466561/files/466561.pdf. [Accessed 02 11 2024].

8 Appendix A: Reconstruction Filter Schematic