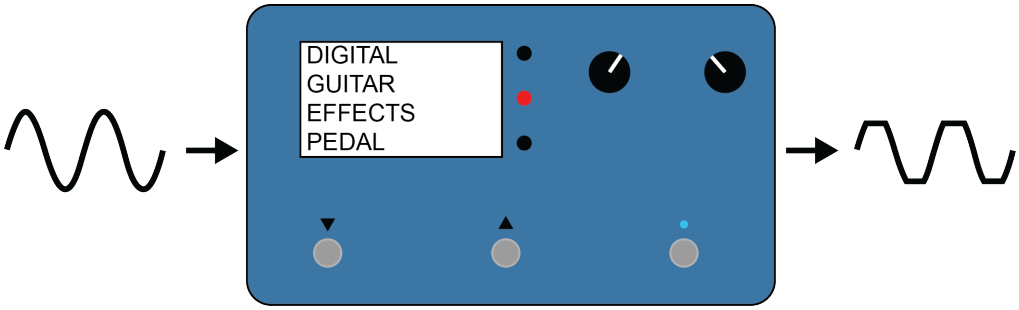
Digital Guitar Effects Pedal

by

Ian Lang



Senior Project

ELECTRICAL ENGINEERING DEPARTMENT

California Polytechnic State University

San Luis Obispo

June 2018

Abstract

The Digital Guitar Effects Pedal assists guitarists in creating music by implementing several useful functions. The pedal takes an analog input signal from an electric guitar, performs digital operations on it, and outputs a modified analog signal to an external guitar amplifier. Pedal functions include: an easy-to-use instrument tuner, a looper which records and plays back music segments, a tap tempo mode allowing easy synchronization with other instruments, and various guitar effects (distortion, echo, and vibrato, among other examples). The pedal user interface displays the current selected function, and allows easy switching between effects.

Acknowledgements

I would like to thank a number of people for their contributions to the success of this project. A huge thank you to my advisor, Dr. Bridget Benson, for her support and advice through the past two quarters. I would also like to thank Dr. David Braun for assisting in senior project preparation, and making sure my writing in the report remains sufficiently lard free. Thank you to Claire Leyba and Cassidy Sargent for your continued moral support, and to Brice Turnbull and Billy Gottenstrator for your guitar playing expertise, and to all my friends and family, without whom this project could never have been accomplished.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

Since the beginnings of the electric guitar, guitarists have sought out different guitar effects to change or improve the sound of their instrument. Before the development of fast, affordable microcontrollers, all guitar effects were created using analog methods. Some effects, such as distortion and overdrive, arose from guitarists pushing the limitations of the vacuum tubes in their amplifiers. Other effects, such as delays or vibrato, used analog circuitry and guitar pedals using these circuits remain popular today [1]. A guitar pedal is the common term used to describe a device that takes an analog signal from a guitar, adds effects to the signal, and outputs the signal to an amplifier. The guitarist usually activates the pedal's effect using their foot, allowing them to play the guitar while changing effects. Most analog guitar pedals use only a mix of operational amplifiers, resistors, and capacitors in their creation, and while this simplifies the design of these devices, it makes it difficult to add multiple different effects into a single enclosure because of the space required for all the components.

The first guitar pedal containing transistors appeared in 1962 as the Maestro Fuzz Tone pedal, and in the 1960's and 1970's the number of available guitar effects pedals greatly increased, with effects such as chorus, wah-wah, and phase pedals becoming available [2]. The first digitized guitar effect pedals did not appear until 1980's, however these units have since become much more common due to widespread availability of powerful microcontrollers.

With the advancements made in modern microcontrollers, implementing guitar effects in the digital domain becomes much more feasible. Algorithms exist which can implement a wide variety of effects, including loudness effects, time effects, pitch effects, and timbre effects [3]. With a digital guitar pedal, the easy collection of multiple effects into a single pedal becomes possible, making it easier for guitarists to select exactly the effect they desire. In addition, opportunities arise to add new functions only possible in the digital domain, such as delays greater than 1 second, an accurate digital tuner, or a looping function to continuously replay segments of music.

This project aims to create an affordable digital guitar pedal with high quality effects. The pedal contains an accurate tuner, a looping function, and several guitar effects. The implemented guitar effects include distortion, delay, echo, vibrato, flanger, and chorus [3][4].

Chapter 2 explores the requirements and specifications the Digital Guitar Effects Pedal must meet to compete with other commercially available pedals.

Chapter 2: Requirements and Specifications

Determining the requirements and specifications that the pedal should satisfy involves analyzing different customer needs. A discussion of potential customers appears in appendix A, and analysis of their needs generates a set of marketing requirements, appearing at the bottom of table I. These marketing requirements drive the generation of engineering specifications, which follow the guidelines set forward in IEEE 1233. The specifications are implementation free, bounded, complete, unambiguous, verifiable, and traceable [5]. Table I below shows the engineering specifications and marketing requirements for the Digital Guitar Effects Pedal.

Table I – Digital Guitar Effects Pedal Requirements and Specifications

Marketing Requirements	**Engineering Specifications**	**Justification**
1	Final production cost $<$ \$100	Most commercial digital guitar pedals retail for over \$250, so maintaining market viability requires low production costs.
2	Tuner reports frequency of a pure sine wave from 60 Hz-350 Hz within +/- 5 cents	Any variation in tuner accuracy should remain inaudible to users. Research shows the average person cannot perceive a difference in pitch within +/- 5 cents, and modern pitch detection algorithms can achieve 1 cent accuracy [6].
3	Looping function can record and play back over 20 s of sound (at 44.1 kHz sample rate and 16-bit resolution)	Recording duration should allow user to record several measures.
4	Analog-digital and digital-analog conversion occur with at least 44.1 kHz sample rate and at least 16-bit resolution	44.1 kHz and 16-bit resolution represents "CD Quality," necessary for high quality sound and to reduce aliasing effects.
4	Audio input has $>$ 100 kΩ input impedance measured from 20 Hz-20 kHz	Electric guitars can have output impedances ranging up to 10 kΩ, so the pedal requires high input impedance to prevent signal attenuation [7].
4	Audio output has $<$ 100 kΩ output impedance measured from 20 Hz-20 kHz	Guitar amplifiers expect electric guitar inputs which have up to 10 kΩ output impedance, so the pedal should have an comparable output impedance [7].

4	The total harmonic distortion (THD) of the output signal from the input signal with no effects added should be $< 1\,\%$ over the audible range (20 Hz-20 kHz)	Minimal unintended output distortion in the signal preserves sound quality, and any distortion should remain inaudible to the user.
5	Device dimensions should not exceed $20\,\text{cm} \times 20\,\text{cm} \times 6\,\text{cm}$	Smaller dimensions make the pedal easier to transport.
5	Device weight should not exceed 1 kg	Lower weight makes the pedal more portable.
6	Device audio input and output compatible with standard 1/4" audio cables	Electric guitar outputs, guitar amplifier inputs, and other guitar effect pedals all use 1/4" jacks, so this device must also maintain compatibility with this standard [8].
6	Device powered from external 9 VDC power supply rated at or under 500 mA	Most other guitar effects pedals operate on 9 VDC, so this device should follow this standard allowing for users to provide their own power supplies if desired.
6	Power supply connection compatible with standard 5.5×2.1mm barrel plug with center negative polarity	Standard power supplies for guitar pedals use 5.5×2.1mm barrel plugs with a center negative polarity, so this product should accept this standard power input [8].
7	The device follows the standards described in UL 60065	Following UL 60065: Standard for Audio, Video and Similar Electronic Apparatus ensures product safety and prevents user injury [9].

Marketing Requirements

1. Cheap
2. Accurate tuner
3. Contains looping function
4. High quality sound
5. Portable
6. Easily interfaces with guitars, amplifiers, and other effects pedals
7. Safe

These requirements and specifications affect the device functional decomposition, explored in chapter 3.

Chapter 3: Functional Decomposition

This chapter decomposes the project into functional blocks and explains the operation of these blocks in relation to overall system functionality. This chapter contains both level 0 and level 1 functional decompositions.

3.1 Level 0

The Digital Guitar Effects Pedal takes an input audio signal and outputs a modified audio signal, with user input determining the output characteristics. Figure 1 below shows the level 0 system block diagram.

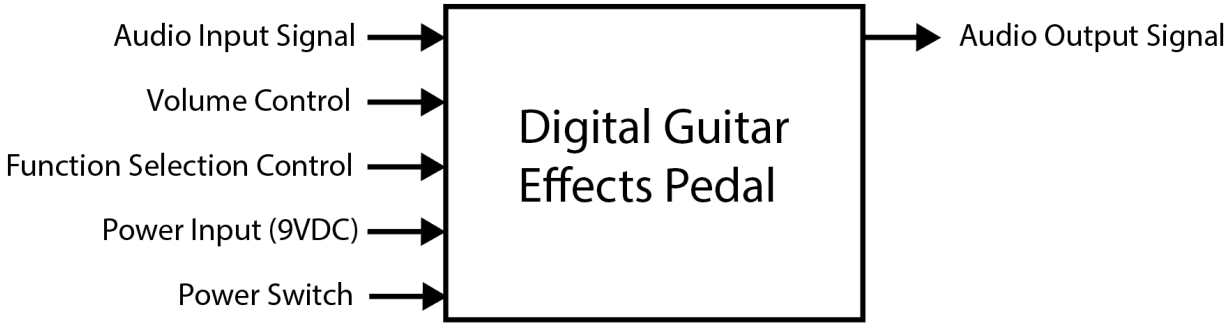


Figure 1 – Level 0 System Block Diagram

Table II shows the level 0 functional requirements, derived from the level 0 block diagram and specifications. The expected audio input and output voltage levels arise from the electric guitar pickup output level, which could reach up to 1 Vrms [7].

Table II – Level 0 Functional Requirements

Module	Digital Guitar Effects Pedal
Inputs	• Audio input signal: input from electric guitar, 1 Vrms max. • Volume control: volume continuously variable from no volume to maximum volume. • Function selection control: interface to select active functions applied to the input signal. • Power input: 9 V DC power supply, rated 500 mA. • Power switch: can interrupt power to device to toggle device operation.
Outputs	• Audio output signal: output to external guitar amplifier, 1 Vrms max.
Functionality	The device should take an audio signal from an electric guitar, digitally add effects to it (distortion, delay, and vibrato, among other examples), and output the original signal plus any effects added to an external guitar amplifier. The user controls the desired effects using an interface on the pedal exterior. The user can adjust the volume continuously from no volume to maximum volume. The user can also toggle device power using a switch.

3.2 Level 1

The level 1 block diagram in figure 2 expands upon the level 0 block diagram and explains device inner functionality. The input audio signal passes through an analog-digital converter, and a microcontroller then processes the digital signal before sending it to a digital-analog converter. A power supply block takes the input power and converts it to the necessary system power rails. The microcontroller communicates with system memory to store and retrieve data as necessary.

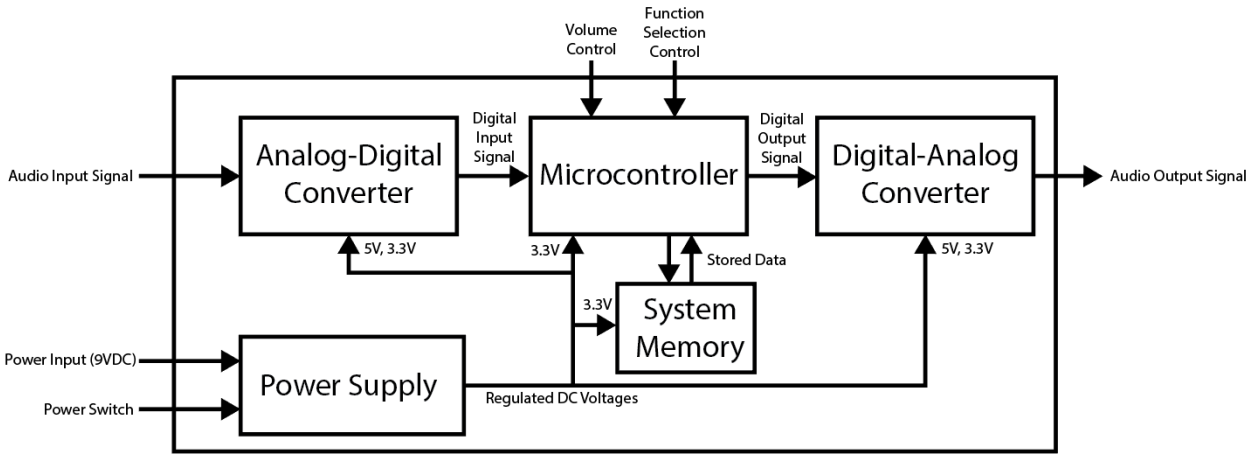


Figure 2 – Level 1 System Block Diagram

Table III shows the functional decomposition for the analog-digital converter (ADC) block, which

converts the input audio signal into a digital signal readable by the microcontroller.

Table III – Analog-Digital Converter Functional Requirements

Module	Analog-Digital Converter (ADC)
Inputs	• Audio input signal: input from electric guitar, 1 Vrms max. • 5 VDC: regulated power rail from internal power supply for ADC reference voltage and active filters, 10 mA max. • 3.3 VDC: regulated power rail from internal power supply for ADC power, 1 mA max.
Outputs	• Digital input signal: Digitized audio input signal from the ADC. The ADC communicates the digital signal to the microcontroller over a serial peripheral interface (SPI) bus.
Functionality	The analog-digital converter takes the input audio signal and samples it at regular intervals to convert it to a digital signal which the microcontroller can then modify. Based on the specifications in table I, the ADC must sample faster than 44.1 kHz and with at least 16-bit resolution. This block filters the analog audio signal input before converting it to digital to reduce noise and prevent aliasing. The ADC uses a reference voltage of 5 V to allow sufficient headroom to prevent signal clipping. The ADC draws its power from a 3.3 V rail to allow communication with the microcontroller which also runs on 3.3 V.

Table IV shows the functional decomposition for the microcontroller unit (MCU) block, which controls all system functions and performs operations on the digital audio signal. The STM32F446 microcontroller from STMicroelectronics has a fast clock speed and a floating-point unit, making it ideal for the digital operations required in this device [10].

Table IV – Microcontroller Functional Requirements

Module	Microcontroller
Inputs	• Digital input signal: Digitized audio input signal from the ADC, communicated over SPI. • Volume control: Volume continuously variable from no volume to maximum volume. • Function selection control: interface to select active functions applied to the input signal. • 3.3 VDC: regulated power rail for microcontroller power, 200 mA max. • Stored data: interface to system memory to store data, communicated over SPI.
Outputs	• Stored data: interface to system memory to retrieve stored data, communicated over SPI. • Digital output signal: modified digital audio signal for the DAC, communicated over SPI.
Functionality	The microcontroller takes the digital audio input signal and performs operations on it according to the functions selected by the user. The user can adjust the signal amplitude continuously, and the microcontroller can store the input audio signal in system memory for later recovery. The microcontroller outputs the modified digital signal to the DAC.

Table V describes the functional requirements for the digital-analog converter (DAC), which takes the output digital audio signal and converts it to a usable analog output signal.

Table V – Digital-Analog Converter Functional Requirements

Module	Digital-Analog Converter (DAC)
Inputs	• Digital output signal: modified digital audio signal from the microcontroller, communicated over SPI. • 5 VDC: regulated power rail from internal power supply for DAC reference voltage and active filters, 10 mA max. • 3.3 VDC: regulated power rail from internal power supply for DAC power, 1 mA max.
Outputs	• Audio output signal: output to external guitar amplifier, 1 Vrms max
Functionality	The DAC takes the digital audio signal from the microcontroller and transforms it back into an analog signal, then outputs it to an external guitar amplifier. Like the ADC, the DAC must sample faster than 44.1 kHz and have at least 16-bit resolution. The DAC requires the same power rails as the ADC.

Table VI shows the functional decomposition for the stored memory block, which the microcontroller can communicate with to store data.

Table VI – Stored Memory Functional Requirements

Module	System Memory
Inputs	• Stored data: Data from the microcontroller, communicated over SPI. • 3.3 VDC: regulated power rail from internal power supply for memory power.
Outputs	• Stored data: Data from memory, communicated over SPI.
Functionality	System memory stores data from the microcontroller, which may include segments of audio. The system memory must transmit and receive data at speeds greatly exceeding 44.1 kHz to prevent microcontroller internal memory overflow. The memory size must allow storage of enough audio data to meet the specifications.

Table VII describes the requirements for the power supply, which provides the necessary voltage rails for system operation.

Table VII – Power Supply Functional Requirements

Module	Power Supply
Inputs	• Power input: 9 V DC power supply, rated 500 mA. • Power switch: Can interrupt power to device to toggle device operation.
Outputs	• 5 VDC: regulated power rail, 20 mA max. • 3.3 VDC: regulated power rail, 200 mA max.
Functionality	The power supply must take the 9 V DC input and transform it into the necessary system voltages. The system requires 5 V and 3.3 V rails, which the power supply can generate using either linear regulators or switching power supplies, depending on system needs. Users can toggle device operation using a power switch, which interrupts system power.

Chapter 4: Design

The design of the Digital Guitar Effects Pedal consists of two distinct aspects: hardware and firmware. Hardware relates to the physical electronic components that provide the desired functionality when assembled together. Firmware describes the code programmed into the microcontroller which controls the different electronic components and allows for user interaction.

4.1 Hardware

Hardware development on this project consists of creating a schematic to describe the connections between the different components, and then manufacturing a printed circuit board (PCB) to connect everything. This project favors the use of individual components instead of pre-made modules because it allows greater design flexibility and optimization opportunities. This section describes the design process for each aspect of the circuit. Some subsections include figures with fragments of the schematic for easier reference, and a full schematic is included in appendix C. A complete parts list appears in appendix E. Creation of the schematic and layout uses KiCAD, a free and open-source electronic development environment.

The schematic uses two different ground references: an analog ground represented by the GNDA net and a digital ground represented by the GND net. Any device in the audio signal path uses analog ground as its reference, which connects to the digital ground in only one location, preventing coupling of digital noise into the analog circuitry.

Microcontroller

At the heart of the Digital Guitar Effects Pedal lies the microcontroller unit (MCU), which controls the operation of every device in the circuit. The MCU selected is the STM32F446RC from STMicroelectronics [10]. This MCU runs at a maximum clock speed of 180 MHz, allowing many instructions per audio sample. It contains a floating point unit (FPU), which allows fast computations of complex DSP algorithms like Fast Fourier Transforms (FFTs). The RC version of the STM32F446 contains 256 kB flash memory and 128 kB SRAM, a sufficient amount to implement the various desired features. The 64-pin LQFP device package is selected, since 64 pins is enough for implementing all circuit functions while allowing for more manageable hand-soldering than the 100-pin version.

Programming the MCU takes advantage of the available Joint Test Action Group (JTAG) interface. Programming the MCU from a computer requires using the ST-LINK/V2, an in-circuit debugger/programmer from STMicroelectronics which provides a USB to JTAG interface [11]. The ST-LINK/V2 normally requires a 20-pin ribbon cable connector to program the circuit, however, section 4.2 of [11] describes an option for a 10-pin interface using a proprietary Tag-Connect cable and adapter. To save space on the PCB, the design uses the 10-pin interface as described in the Tag-Connect manual [12], however the ST-LINK/V2 attaches to the PCB using jumper wires instead of the actual Tag-Connect cable. The JTAG standard normally requires external pull-up resistors, however the MCU provides embedded pull-up and pull-down resistors to reduce the number of external parts [13].

A 3.3 V rail powers the MCU, and 0.1 µF capacitors adjacent to each of the four MCU power pins provide power supply decoupling. A separate 10 µF capacitor placed close to the package provides additional decoupling.

To provide a stable reference clock for the MCU, the circuit contains an external 25 MHz crystal, the ABM3-25.000MHZ-D2Y-T [14]. The crystal connects to the MCU as recommended in [15]. Figure 3 shows the fragment of the schematic containing the MCU.

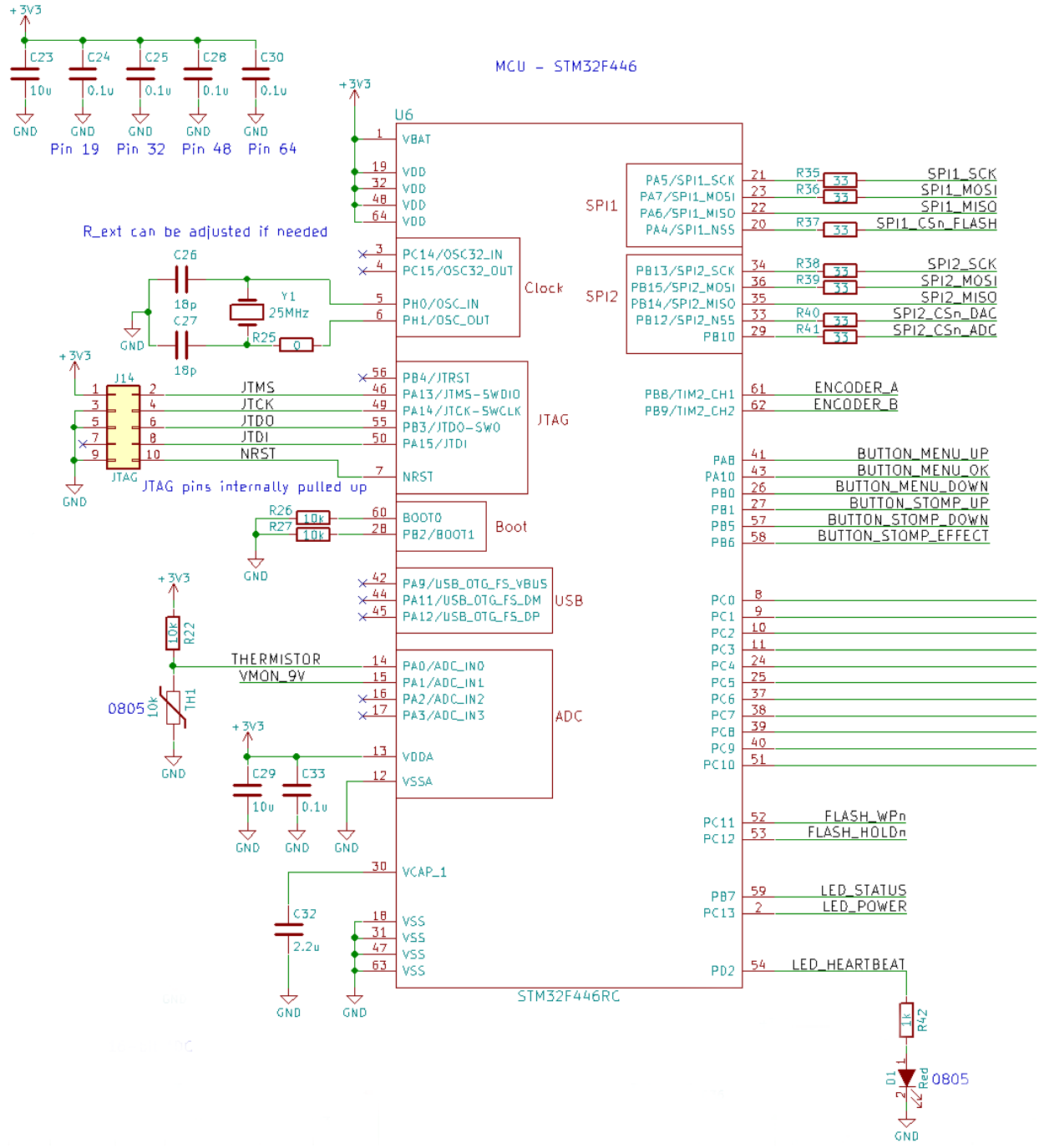


Figure 3 – MCU Schematic

The circuit contains a 10 kΩ thermistor attached to one of the MCU's ADC pins, allowing the circuit to detect the ambient temperature to prevent overheating. The design uses two serial peripheral interface (SPI) buses, SPI1 for communication with the flash memory chip and SPI2 for communication with the ADC and DAC. The high-speed data lines are series terminated with 33 Ω resistors to reduce signal reflections, and pull-up resistors on chip select lines ensures devices remain deactivated by default. A surface mount LED attached to one of the GPIO pins acts as the device's heartbeat. The MCU firmware manually switches it on and off at a regular interval, and if the LED stops blinking, something unexpected has happened with the operation of the code.

Power Supply

As noted in table VII in the functional decomposition, the circuit requires both 5 V and 3.3 V rails to power the analog and digital components. The 3.3 V rail powers the digital circuitry, including the MCU, the analog to digital converter (ADC), digital to analog converter (DAC), the Flash memory, and the LCD display. Table VIII below shows the worst case supply current for each device on the digital rail.

Table VIII – Digital Power Supply Current Requirements

Component	**Digital Supply Current**
MCU	100 mA [10]
ADC	Negligible [16]
DAC	240 µA [17]
Flash Memory	25 mA [18]
LCD Display	20 mA [19]

Based on the above requirements, a converter for the digital rail needs to source at least 150 mA. Since this rail is only provides digital power and not precise analog references, the design can use a DC-DC switching buck converter, which saves power over using a linear regulator. The buck converter chosen is the TPS560200, which has an input voltage range of 4.5 V to 17 V, an adjustable output voltage, and a 500 mA continuous output current capacity [20]. This buck converter has a high switching frequency of 600 kHz, allowing for smaller external components to achieve the same performance as slower converters. Figure 4 below shows the buck converter schematic.

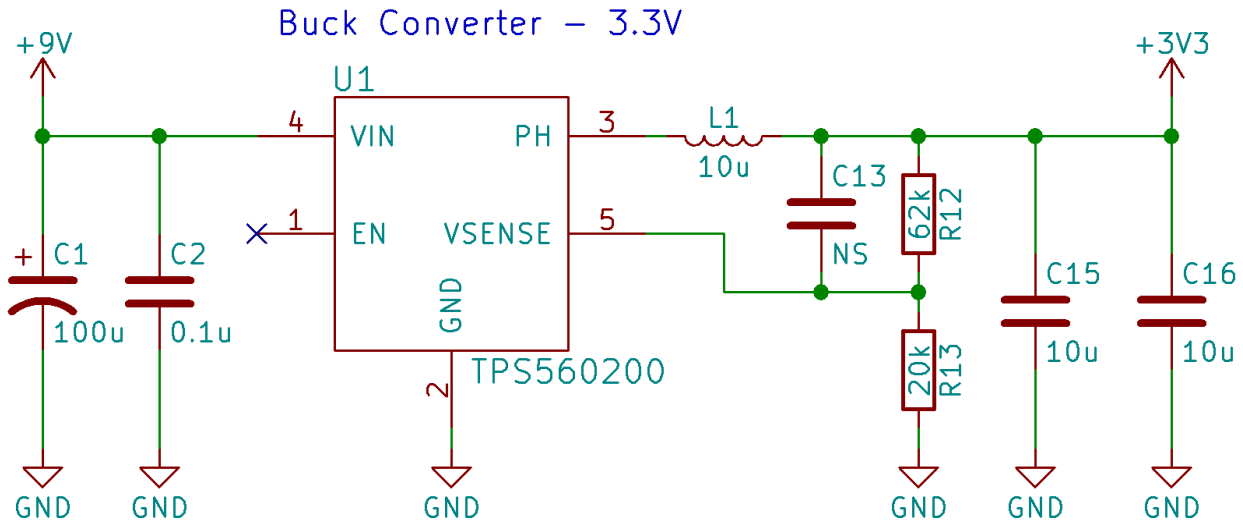


Figure 4 – Buck Converter Power Supply Schematic

The buck converter circuit in the schematic is based directly on the typical application circuit from the datasheet [20], with output voltage set to 3.3 V. The enable pin is left floating to permanently enable the converter. The design closely follows the recommended PCB layout to give the best performance.

The 5 V rail powers the analog circuitry in the device, and so requires there to be less noise on the power rail than the digital circuits. Because of this, excess switching noise makes a buck converter not an ideal choice, since the switching noise interferes with the precise analog references required by the ADC and DAC. The design instead uses a low drop-out linear regulator (LDO), since these components usually have excellent line and load regulation and do not introduce additional switching noise. Linear regulators dissipate more power than switching converters, however the current requirements of the analog device, seen in table IX below, remain low enough to prevent excessive power dissipation. At max current draw, the linear regulator should only dissipate 120 mW.

Table IX – Analog Power Supply Current Requirements

Component	Analog Supply Current
ADC	4.5 mA [16]
DAC	45 µA [17]
Op-amps	4 × 6.5 mA [21]

The design uses the LP2951 LDO, which has an input range up to 30 V and a continuous output current capacity of 100 mA [22]. Figure 5 shows the LDO circuit, based on the typical application circuit from the datasheet.

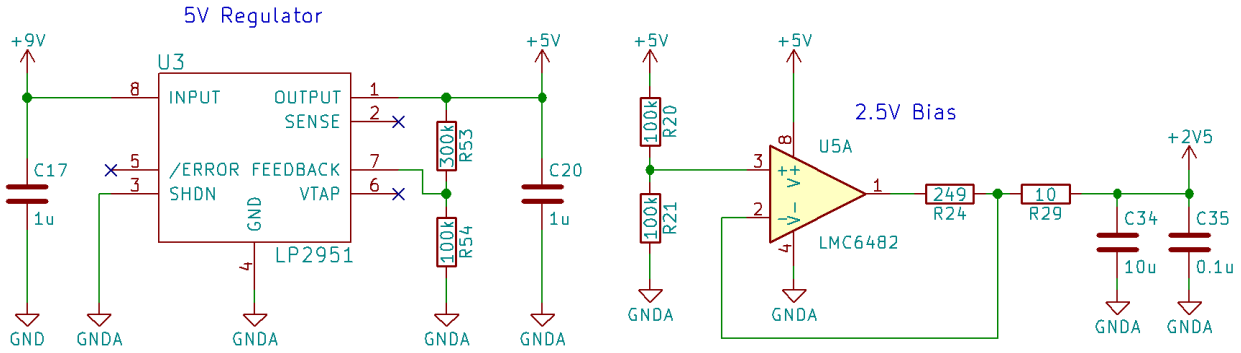


Figure 5 – LDO Regulator Schematic

A buffered voltage divider creates a 2.5 V bias voltage for the active filters.

Analog to Digital Converter

The analog to digital converter (ADC) takes the audio signal from the electric guitar and converts it to a digital binary representation. The ADC chosen must meet the sampling rate specification of 44.1 kHz and the resolution specification of 16 bits. The ADS8319 meets both of these specifications, with a max 500 kSPS sampling rate and 16-bit resolution [16]. This ADC also has separate digital and analog power supplies allowing for communication at 3.3 V and conversion at 5 V. The ADC communicates over SPI with a max frequency of 30 MHz. Figure 6 below shows the section of the schematic covering the ADC and input filter.

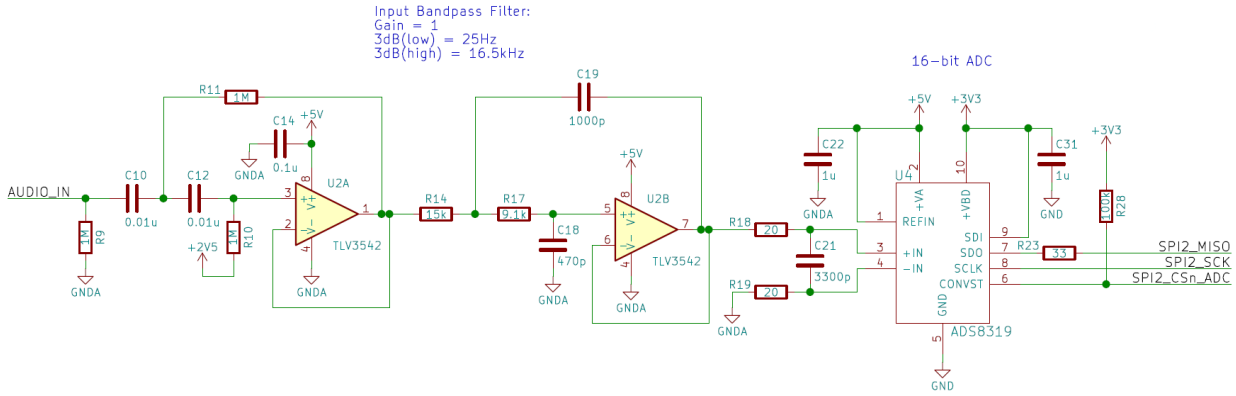


Figure 6 – ADC Schematic

The input audio signal passes through a 4th-order unity gain Sallen-Key band-pass filter with cutoff frequencies of 25 Hz and 16.5 kHz. This preserves the signal over the audible range, while eliminating signals past the Nyquist frequency which would cause aliasing. The active filters use the TLV3542 dual operational amplifier, which has 200 MHz gain-bandwidth, high slew rate, and rail-to-rail input and output [21]. The amplifiers before the ADC require high slew rates because the ADC input presents a capacitive load, which must charge fast enough to allow for accurate ADC conversion. After the active filters, the signal passes through a passive RC filter with a cutoff

frequency of 12 MHz as recommended in the ADS8319 datasheet. Components in the audio signal path use the analog ground reference to isolate them from digital noise arising from the high-frequency communication interfaces and digital power supplies. The 5 V rail powers the analog circuitry, and the ADC uses the same voltage as its reference. The ADS8319 operates in "3 wire CS mode without busy indicator," selected by tying SDI to +VBD.

Digital to Analog Converter

The digital to analog converter (DAC) takes a digital binary representation of the audio waveform and converts it back into an analog signal. The DAC must meet the same sampling rate and resolution specifications as the ADC to preserve the signal integrity. The design uses the DAC8551 which has 16-bit resolution, a settling time under 10 µs, and communicates over SPI at up to 30 MHz [17]. Figure 7 below shows the schematic for the DAC.

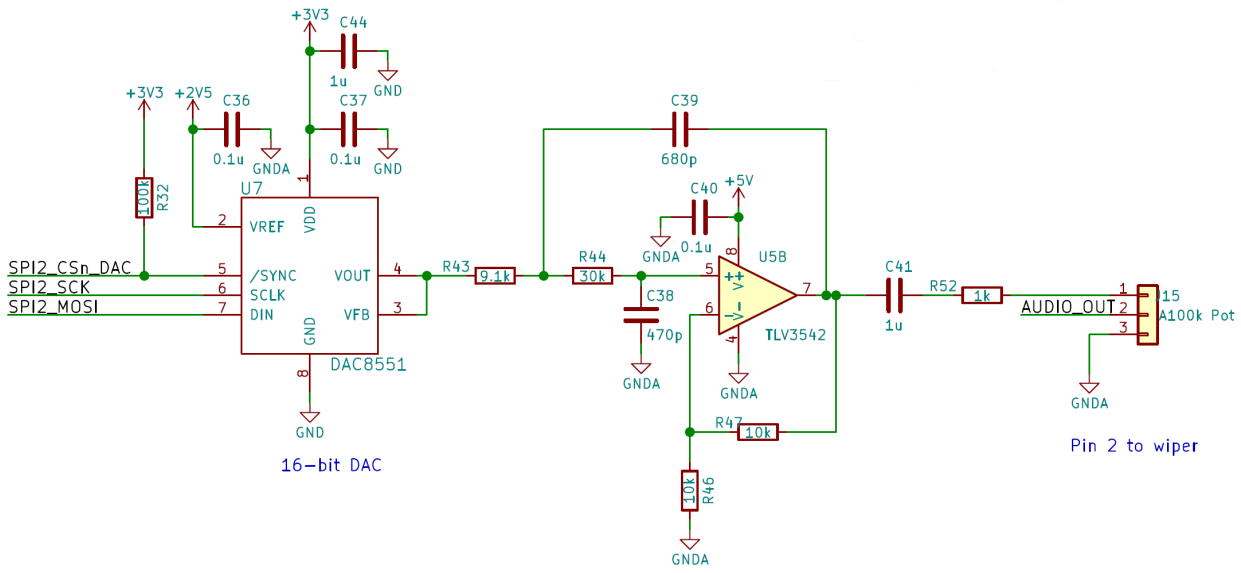


Figure 7 – DAC Schematic

A second order low-pass filter follows the DAC output to act as a reconstruction filter, smoothing the output signal and removing some of the quantization effects. The design uses a corner frequency of 17 kHz for the filter to preserve signals in the audible range. The DAC cannot generate voltages greater than its digital supply, so the DAC uses the 2.5 V rail as its reference voltage instead of the 5 V rail. This essentially cuts the signal amplitude in half, so the reconstruction filter has a gain of 2 to bring the signal back to its original amplitude.

A 1 µF capacitor AC couples the audio signal to remove the DC bias from the DAC output, and the signal then passes through a 100 kΩ logarithmic taper potentiometer. The potentiometer allows the user to adjust the signal amplitude, and since the human ear detects volume logarithmically, a logarithmically tapered potentiometer gives linear volume change when rotated. 100 kΩ output potentiometers reflect the standard on most guitar pedals, so even though they increase output impedance significantly, the pedal should still operate normally. A 1 kΩ resistor is placed in series with the output to protect the op-amps from sourcing excess current if the output becomes shorted.

Flash Memory

The pedal's looper function requires the storage and recall of audio data. Since the audio data is 16-bits at 44.1 kHz, one second of data composes 88.2 kB. The MCU only contains 256 kB of internal flash memory, enough to store around 3 seconds of data. This does not meet the recording length specification, so the pedal requires an external memory chip. The design uses flash memory since it provides the best compromise between cost and speed. The W25Q128JV-DTR 128 MB flash memory chip has enough storage to hold over 24 minutes of audio data [18]. The chip communicates over SPI at up to 50 MHz, allowing rapid data transfer. The flash chip communicates with the MCU using the SPI1 bus, while the ADC and DAC use the SPI2 bus. This allows the MCU to carry out communication with the flash memory independent of the audio signal processing.

Liquid Crystal Display

The pedal should display the currently selected effect so the user can adjust different parameters. The pedal uses the NHD-0420H1Z-FSW-GBW-33V3 4x20 character liquid crystal display (LCD) since it operates from a 3.3 V rail and displays enough characters to convey the necessary information [19]. The display communicates with the MCU using an 11-pin parallel interface, with 8 pins for data and 3 pins for operation selection. A small 10 kΩ linear taper potentiometer controls the contrast of the display.

User Interface

To allow the user to change different options, the pedal uses 3 small momentary push-buttons and 3 large momentary footswitches. The pushbuttons allow control of different menu options, and consist of 'Up,' 'Down,' and 'OK' buttons. The footswitches are durable enough for users to press them with their feet, so options can be changed while still playing the guitar. The footwsitches consist of 'Up,' 'Down,' and 'Effect' buttons. The original design contains an RC lowpass filter after each button to debounce the output, however for reasons described in section 4.2, the final product opts for firmware debouncing instead. The buttons use the MCU's internal pull-up resistors on the GPIO ports to reduce external components.

The pedal uses a rotary encoder allowing the user to easily adjust different effect parameters. The MCU's timer peripherals contain functionality that allow them to decode the encoder output, so the encoder's A and B outputs attach to the TIM2 CH1 and CH2 GPIO pins on the MCU [23]. The encoder outputs pass through an RC filter with a time constant of 1 ms to debounce the signals.

Input and output 1/4" audio jacks allow attaching external audio cables, and the grounds from the jacks connect to the analog ground plane through ferrite beads to isolate the ground reference from radio-frequency (RF) noise picked up on the cables.

A 3PDT power switch turns on the pedal, with one pole switching power and the other two poles shorting the input to the output with the pedal off. This implements a "true bypass" function, so sound can pass through the pedal when not powered. Figure 8 below shows a representation of the 3PDT switch underside, and shows the function of each connected wire. The right of the image shows how the pins map to the actual switch functionality.

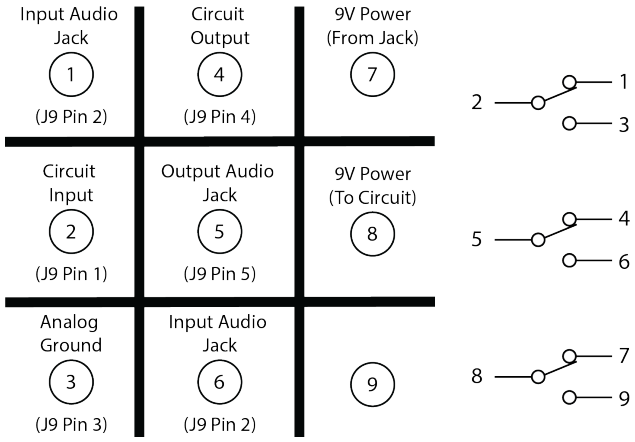


Figure 8 – Power Switch Connections

The pedal uses two external LEDs; one LED indicates the power status and the another indicates the status of the effects.

PCB Layout

Due to the complexity of the design and extensive use of surface mount components, the creation of a PCB becomes necessary. The pedal uses a 4-layer PCB as recommended by STMicroelectronics for designs containing any STM32 microcontrollers [13]. All components and integrated circuits reside on the top layer, along with most of the signal routing. Any excess signal routing resides on the bottom layer. The middle layer under the top layer contains solid ground planes for both analog and digital grounds, and the other middle layer contains solid power planes for 3.3 V and 5 V rails. This scheme provides a low impedance path to power or ground from anywhere on the PCB. The PCB separates analog and digital circuitry into their own sections on the board. Each section has its own power and ground plane to reduce the coupling of noise from the digital circuitry to the sensitive analog circuitry. The analog section contains the the analog ground and 5 V rail, and the digital section contains the digital ground and 3.3 V rail. The analog and digital ground planes connect at a single location, with a 0 Ω jumper. The original design called for a ferrite bead to connect the ground planes for additional noise suppression, however testing revealed the ferrite causes complications with the high speed current draw from the ADC.

Since noise can easily interfere with the crystal oscillator, the design implements special layout considerations as recommended by STMicroelectronics [15]. The layout isolates the digital ground plane directly underneath the crystal from the rest of the ground plane, and a guard trace surrounds the crystal to stop high frequency signals from coupling across the PCB surface.

The PCB uses 0.25 mm traces for all signal routing and 0.4 mm traces for power routing. All vias are 0.6 mm diameter with 0.3 mm drill diameter for signals and 0.4 mm drill diameter for power. The final board dimensions are 78 mm × 38.2 mm. Appendix D contains images of each layer of the PCB layout.

Device Assembly

The PCB is fabricated using OSHPark, then components are hand-soldered on. Figure 9 below shows the front and back of the fabricated PCB. Figure 10 shows a close up of the assembled PCB, and figure 11 shows an overview of the assembled system.

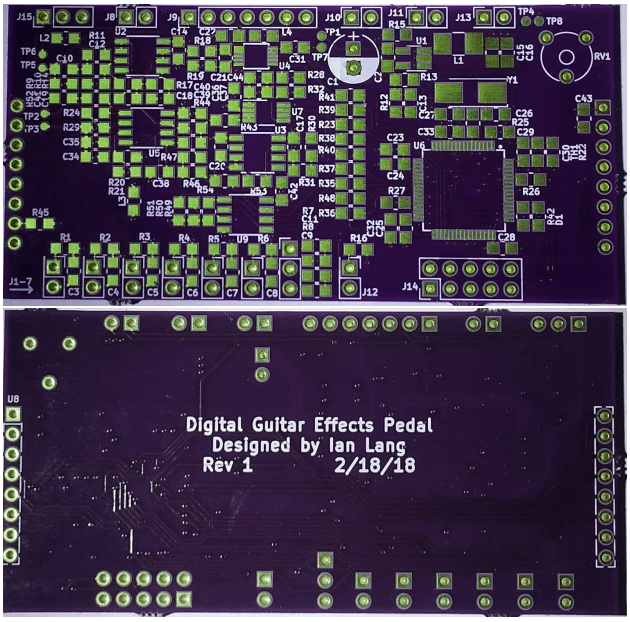


Figure 9 – PCB Front (Top) and Back (Bottom)

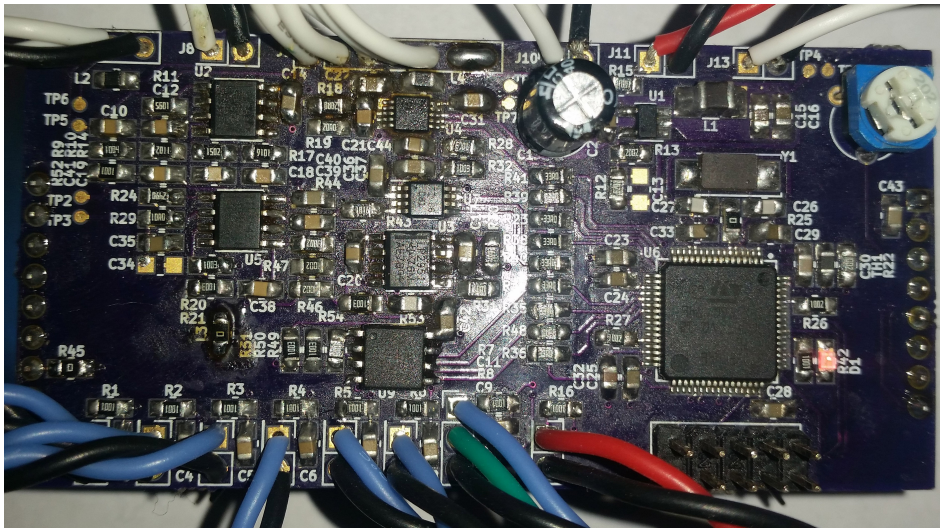


Figure 10 – Assembled PCB

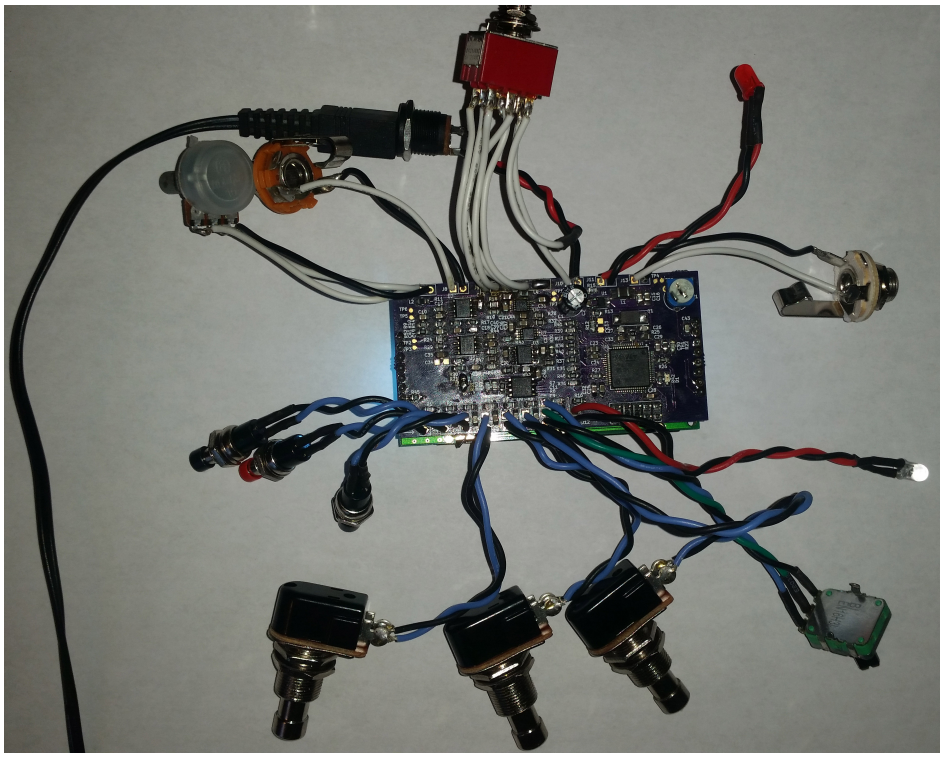


Figure 11 – Assembled System

To increase the durability of the pedal and provide some resistance to external noise, the electronics are enclosed in an aluminum diecast enclosure, the 4S1590DD from Mammoth Electronics. The enclosure has dimensions of 188 mm × 119.5 mm × 33 mm, allowing the spacing of the footswitches so the user can easily operate them with their feet. The switches, audio jacks, and dials attach to the enclosure through holes drilled through the aluminum, and the LCD rests in a rectangular hole cut from the top. Figure 12 shows the top of the final pedal in its enclosure, and figure 13 shows the enclosure back with the audio and power jacks.

Figure 12 – Completed Device in Enclosure - Top

Figure 13 – Completed Device in Enclosure - Back

The three footswitches at the bottom of figure 12 are arranged (from the left): 'Up,' 'Down,' and 'Effect.' The three pushbuttons next to the LCD screen are arranged (from the top): 'Up,' 'OK,' and 'Down.' The left dial is the rotary encoder and the right dial controls the volume. The back of the enclosure in figure 13 contains (from the left): the input jack, the power jack, the power LED, the power switch, and the output jack.

4.2 Firmware

After assembling the hardware and verifying that it functions, the next step involves writing firmware to control all the peripherals and give the device its functionality. The integrated development environment (IDE) used to develop the code is *System Workbench for STM32*, which supports all STM32 microcontrollers and automatically downloads specific driver packages. The firmware sets the microcontroller's SYSCLK frequency to 160 MHz, the peripheral PCLK frequency to 40 MHz, and the timer clock frequency to 80 MHz.

Developing the firmware involves coding all the peripheral drivers from scratch, since this gives the most flexibility in designing the program and allows optimization for speed where appropriate. The firmware uses a finite state machine to control program flow, consisting of five different states which can switch depending on input from the user. The states are: Default State, Adjust State, Tuner State, Looper State, and Tap Tempo State.

In all states, the microcontroller continuously samples the audio input using the ADC and outputs data to the DAC. A hardware timer triggers a software interrupt at 44.1 kHz which starts the conversion of the ADC. After the minimum conversion time has elapsed, the program initiates communication over the SPI2 bus to get the data from the ADC. Upon receiving the data, the program applies any active effects to the signal. A second software interrupt is generated in the middle of the timer's count, triggering the program to open communication over the SPI2 bus to send the last processed sample to the DAC. All these functions run using only interrupts generated by hardware timers, so the program never waits for communications to complete, and can resume other tasks in the meantime.

The user interacts with the firmware through the buttons, which attach to GPIO interrupts so the code does not have to constantly poll them. The program tracks both the pressing and releasing of the button, and only registers and processes the button press when it releases. This allows secondary functions when the button remains pressed for a longer period. The original design for the external debouncing RC filters works on button down presses; however, upon releasing the button, the capacitor must charge up through the internal pull-up resistor, which is too slow to accurately detect quick button presses. For this reason, the design opts to remove the external filter, and the firmware debounces the buttons internally by ignoring subsequent button presses for a set amount of time after detecting a press.

Select code fragments appear in appendix F. Due to the length and quantity of device peripheral driver functions, only functions directly related to core program flow appear in the appendix. Comments in the code explain how different devices functions operate.

Default State

In the default state, the device shows the currently selected effect on the LCD screen, as well as any parameters that affect the operation of the effect. The user can toggle the effect by pressing the 'Effect' footswitch, which also activates a blue LED to indicate the status of the effect. The user can change the currently selected effect by pressing either the 'Up' or 'Down' footswiches, or by rotating the rotary encoder. Changing to another effect does not deactivate the current effect, allowing the simultaneous application of multiple effects. Pressing the small 'Up' or 'Down' menu buttons cycle through the effect parameters, with an arrow indicating the parameter currently in focus. Pressing the small red 'OK' button switches to the adjust state, allowing the user to change the currently selected parameter. Figure 14 shows an example of the default state, with an effect named Effect 2 which has two parameters, Param 1 and Param 2.

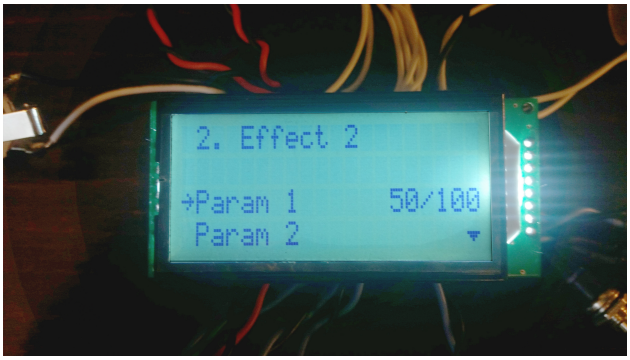


Figure 14 – Default State GUI Example

Adjust State

In the adjust state, the pedal functions similarly to the default state, except the rotary encoder now changes the value of the currently selected parameter. The LCD screen displays the selected parameter, as well as a bar graph which visually indicates the parameter value. Pressing the 'Up' and 'Down' menu buttons cycle through the effect parameters, and pressing the 'OK' menu button again returns the device to the default state.

Tuner State

The device enters the tuner state by holding down the 'Down' footswitch for over a second. The tuner should accurately detect the fundamental frequency of the input signal and display the corresponding musical note on the LCD so the user can tune their instrument. The simple method for determining frequency of the signal involves taking the Fourier transform, however to get even 1 Hz resolution in the frequency spectrum would require a 44,100 length fast Fourier transform (FFT) operation, which would require excessive computational effort and take too much time.

The tuner determines the fundamental frequency using a method called the normalized squared difference function (NSDF), a detailed explanation of which appears in [6]. The basic principle involves taking the auto-correlation of the input signal, which for a periodic signal gives peaks at every period of the fundamental frequency. The algorithm normalizes the auto-correlation result to give consistent amplitudes regardless of input signal amplitude, allowing a simple peak finding algorithm to find the correct fundamental frequency and isolate it from any harmonics. This method allows better than 5-cent accuracy in the frequency range occupied by open guitar strings (60 Hz - 350 Hz), and can determine the frequency of an input signal containing only two full periods of the the fundamental frequency [6]. This allows for shorter windows, greatly reducing the computation time. The tuner uses a window size of 1024 samples.

The code calculates the NSDF $n'(\tau)$ using equation 4.1 seen below:

$$n'(\tau) = \frac{2r'_t(\tau)}{m'_t(\tau)} \tag{4.1}$$

where $r'_t(\tau)$ represents the auto-correlation function and:

$$m'_t(\tau) = \sum_{j=t}^{t+W-\tau-1} (x_j^2 + x_{j+\tau}^2) \tag{4.2}$$

with x_t as the input samples and W as the length of the window used [6]. The firmware calculates the auto-correlation using the built in correlation function (`arm_correlate_f32()`) in the Cortex Microcontroller Software Interface Standard (CMSIS) library, which takes advantage of the microcontroller's floating-point unit to perform a fast calculation. The code calculates $m'_t(\tau)$ and $n'_t(\tau)$ incrementally using a method described in [6], which initially sets both x^2 terms equal to $r'_t(0)$ (the auto-correlation result at $\tau = 0$) then subtracts off specific x_t^2 terms as τ increases. The exact algorithm appears in Appendix F in the `Audio_Tuner()` function.

After obtaining the NSDF function, a simple peak picking algorithm determines the fundamental frequency. The fundamental frequency represents the first major peak after the first positive going zero crossing, so the algorithm cycles through the NSDF function (which starts at 1) and looks for the first negative going zero crossing, then looks for the first positive going zero crossing. The algorithm then determines the maximum of the function before the next negative going zero crossing, and the index of the detected peak represents the period of the fundamental frequency expressed in number of samples.

The tuner records a number of past results to obtain a better average value. The function sorts the previous values and only looks at values around the median to throw away outlying data, and then averages those samples to get a more accurate result. The function then calculates the note on the MIDI scale from the average period using equation 4.3, and the screen displays the note to the user [6].

$$note = \frac{log_{10}(f_s * 16/(T_{average} * f_{ref}))}{log_{10}(\sqrt[12]{2})} \tag{4.3}$$

The above equation uses f_{ref} to calculate the note, which for most modern music is 440 Hz. The user can adjust this value if desired by pressing the 'Up' and 'Down' menu buttons. f_s is the sampling rate of 44.1 kHz.

Looper State

The device reaches the looper state by holding down the 'Up' footswitch for over a second. The looper records the input data stream and saves it in the external flash memory chip. Upon finishing recording, the pedal retrieves the data and plays it back in a loop. This allows the user to record several measures of a backing track to later play over. The 'Effect' footswitch controls most of the looper functionality, and pressing it once when the looper is empty starts the recording. Pressing it again stops the recording and immediately begins playback. Further presses start and stop playback at will. Holding the 'Down' footswitch for over a second erases the current recording, displaying a message on the screen while the operation executes.

The flash memory chip allows programming in segments of 256 bytes called pages, and must erase each page before programming can occur. Since each audio sample requires 16 bits, each write instruction saves the previous 128 samples. Communication with the flash chip occurs over the

SPI1 bus, at a speed of 20 MHz. The code can read data from the flash chip at any location and for any length, but for simplicity the firmware reads 128 samples at a time. Before each write and erase operation, the device must transmit a write enable instruction to the flash chip, and after the instruction the chip remains in a busy state until the operation completes. After the erase instruction, the program periodically reads the chip's status register to determine when the erase operation completes.

Because each instruction may consist of multiple separate operations, the program implements a simple first-in first-out (FIFO) buffer to store future instructions until the program becomes ready to execute them.

Tap Tempo State

Holding down the 'Effect' footswitch for over a second places the pedal in tap tempo mode, which allows the adjustment of a global 'Tempo' parameter that some effects can utilize. Some effects sound best when synchronized to the beat of the music, so the tap tempo function allows the user to easily set the tempo. The pedal automatically determines the tempo from the rhythmic tapping of the 'Effect' footswitch, saving the user from having to figure out the tempo manually.

This function simply uses a hardware timer to measure the time between button presses. The code saves the previous five button press intervals and averages them together to find the frequency in beats per minute (BPM). The code saves this value in a variable accessible by the different effects, so any effects that use it becomes synchronized automatically.

Audio and Effects

The basic structure of the audio involves a large circular buffer to store past samples for use in various effects. The circular buffer is a section of memory where the code places samples one after the other, and wraps around to the beginning of the buffer when the end is reached. The design sets the circular buffer size to 88.2 kB, enough to store exactly one second of audio data. The MCU only contains 128 kB of RAM, so the audio buffer fills the majority of the available memory space. The effects can access any sample within the buffer by subtracting the desired index from the current index value.

The user can toggle the activation of each available effect from the default and adjust states. The program sequentially checks each effect and calls the function that executes the effect if activated. This allows for the activation of multiple effects if desired, though in practice this does not always work since when added together some effects take too much time to process and fail to complete before the next audio sample appears. Each effect has its own set of parameters that adjust the effect operation.

Code listings for each effect appear in appendix F.

Distortion: The distortion effect remains a staple in many rock and metal songs. The pedal creates the distortion effect digitally by first amplifying the signal and then clipping off the top and bottom of the waveform. The program uses an exponential function to accomplish this, as seen in equation 4.4, where G represents the gain applied to the signal x, and $sgn(x)$ represents the sign (+1 or -1) of the input signal [24].

$$y_t = sgn(x) * (1 - e^{sgn(x)*G*x}) \tag{4.4}$$

However, calculating the exponential function using the built in C `exp()` function requires excessive computation time, and the processing time exceeds the available time between audio samples. To overcome this, the design manually implements a Taylor series approximation to ensure that processing completes within the allotted time. Equation 4.5 shows the operation which approximates the exponential function to a good degree.

$$y_t = 1 + \sum_{n=1}^{N} \frac{(-sgn(x) * G * x)^n}{n!} \tag{4.5}$$

The function uses $N = 20$ which provides a good balance of computation speed and accuracy over the expected input range. The effect has two parameters, gain and boost, which the function multiplies together to give the gain G. After calculating the output signal from equation 4.5, the function divides the output value by the boost parameter to reduce the volume of the signal. The user can play with different combinations of the boost and gain parameters to give different sounds. At high gain values, the Taylor series approximation breaks down and the audio quality becomes significantly impaired, so the design sets maximum values of the parameters to provide the largest range of usable signal.

Delay: The delay effect simply produces a replica of the input signal but delayed in time. The function accomplishes this using a basic FIR filter structure, described by equation 4.6:

$$y(n) = x(n) + G * x(n - L) \tag{4.6}$$

where G represents the relative amplitude of the delay and L represents the delay length. The effect's 'Level' parameter describes the relative amplitude G, and ranges from 0 - 1. The 'Tempo' parameter changes the delay length, translated from its value in BPM to an index to get the correct audio sample. The function allows the user to change the 'Tempo' parameter using the tap tempo function, allowing the delay duration to match up exactly with the beat of the music.

Echo: The echo effect appears very similar to the delay effect, except that multiple delayed copies of the input appear in multiples of the delay time. Equation 4.7 below describes the echo:

$$y(n) = x(n) + G * x(n - L) + \frac{3}{4}G * x(n - 2L) + \frac{9}{16}G * x(n - 3L) + ... \tag{4.7}$$

The 'Delay' parameter sets the delay time in units of ms. The 'Level' parameter changes the initial echo amplitude G, and subsequent echos decrease in amplitude by subtracting off the previous sample's gain value divided by 4.

Vibrato: The vibrato effect describes the slight variation in time of a musical note's frequency, normally created on a guitar by gently rocking the fretting hand back and forth. The pedal creates this effect digitally by using a periodically varying time delay and only listening to the delayed

signal. This essentially creates a Doppler shift in the signal which varies the signal's frequency slightly [24]. Equation 4.8 shows the creation of this effect:

$$y(n) = x(n - D * (1 + sin(2\pi ft)) \tag{4.8}$$

D represents the depth of the effect, controlled by the 'Depth' parameter, adjustable between 0 ms and 2 ms. Higher depth values correspond to more frequency deviation and a more pronounced effect. The 'Speed' parameter sets the frequency of the modulation f, adjustable between 1 Hz and 11 Hz.

Flanger: The flanger effect creates a "swooshing" sound in the output. The effect is almost identical to the vibrato effect except for the addition of the original signal to the delayed signal, as shown in equation 4.9:

$$y(n) = x(n) + x(n - D * (1 + sin(2\pi ft)) \tag{4.9}$$

Chorus: The chorus effect attempts to replicate the sound of multiple musicians playing the same notes, simulated by creating copies of the input signal that vary slightly in frequency and time. This effect uses the same principle as the vibrato and flanger effects, with this particular implementation using five copies of the input signal. Each delayed signal has the same frequency and depth values, but they differ in phase to ensure each signal is audible at the output. The frequency of oscillation is also slower than the other effects, adjustable between 1 Hz and 3 Hz.

Chapter 5: Testing

While developing the firmware, several unexpected minor issues arose. One problem involved the ADC getting poorer resolution than expected. The original design called for a ferrite bead to connect the analog and digital ground planes, however testing revealed this adversely affects the ADC's ability to obtain a steady voltage reading. This possibly occurs because the ADC draws current in rapid pulses when performing conversion, which the ferrite bead may interfere with. Replacing the ferrite with a 0 Ω jumper helped increase the ADC resolution, but did not solve the entire problem. Originally, the design used the LMC6482 operational amplifier for the active filters. However, this op-amp has poor performance when driving capacitive loads such as the ADC input, so oscillations occurred in the active filters, reducing the effective resolution. Changing the op-amps to the TLV3542 reduced the oscillations since that amplifier easily drives large capacitive loads and Texas Instruments recommends it for use with ADCs [21].

One oversight in the original design involved using the power switch to disconnect the negative lead of the power supply from circuit ground to turn the device off. This only works correctly if no other ground references exist elsewhere in the circuit, and in this circuit the input and output audio jack sleeves serve as paths to ground, turning on the device even with the power switch off. The final design fixed this by having the power switch toggle the positive power supply connection instead of the negative.

Testing of the design involves ensuring that all specifications are met. Table X reviews the stated specifications and checks if they have been satisfied by the final product.

Table X – Specifications and Test Results

Specification	Met?	Explanation
Final production cost < $100	Maybe	The device is not optimized for mass production yet, however total BOM cost remains under $100, so this specification could be met if assembly costs remain low.
Tuner reports frequency of a pure sine wave from 60 Hz-350 Hz within +/- 5 cents	Yes	Testing with various sine waves at frequencies within 60 Hz-350 Hz revealed consistent readings within 5 cents of the actual frequency.
Looping function can record and play back over 20 s of sound (at 44.1 kHz sample rate and 16-bit resolution)	Yes	Ensured by design.
Analog-digital and digital-analog conversion occur with at least 44.1 kHz sample rate and at least 16-bit resolution	Yes	Ensured by design.
Audio input has > 1 MΩ input impedance measured from 20 Hz-20 kHz	Yes	Ensured by design, final design should have 500 kΩ input impedance.

Audio output has $< 100\,\text{k}\Omega$ output impedance measured from 20 Hz-20 kHz	Yes	Ensured by design.
The total harmonic distortion (THD) of the output signal from the input signal with no effects added should be $< 1\,\%$ over the audible range (20 Hz-20 kHz)	Maybe	Not specifically tested, however individual components in the design support low distortion and no audible distortion recognized during testing.
Device dimensions should not exceed $20\,\text{cm} \times 20\,\text{cm} \times 6\,\text{cm}$	Yes	Final design dimensions: $18.8\,\text{cm} \times 12\,\text{cm} \times 5.5\,\text{cm}$
Device weight should not exceed 1 kg	Yes	Final design weight: 0.579 kg
Device audio input and output compatible with standard 1/4" audio cables	Yes	Ensured by design.
Device powered from external 9 V DC power supply rated at or under 500 mA	Yes	Ensured by design.
Power supply connection compatible with standard 5.5x2.1mm barrel plug with center negative polarity	Yes	Ensured by design.
The device follows the standards described in UL 60065	Yes	Product is inherently safe to use.

Figure 15 below shows an oscilloscope capture of the pedal output when provided with a 440 Hz sine wave at 600 mV peak-to-peak. The top yellow trace shows the input and the bottom green trace shows the output after passing through the pedal's audio signal path. The output copies the input exactly, showing that the pedal manages to convert the signal from analog to digital and back to analog without any distortion.

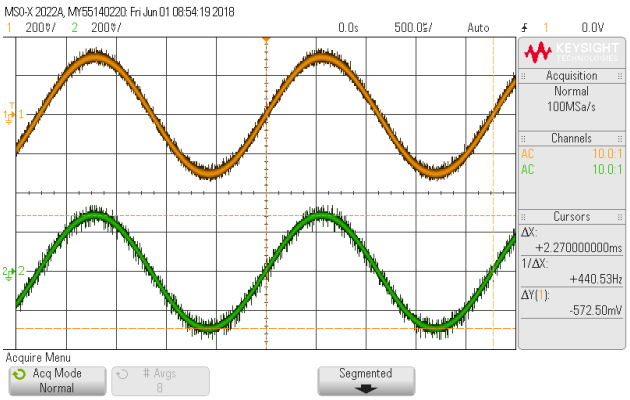


Figure 15 – Oscilloscope Capture of Input (Top) and Output (Bottom)

As seen above, significant noise exists at the output, which becomes audible in the output signal. Figure 16 below shows a zoomed in view of the output noise with the output potentiometer turned all the way down (output directly connected to ground). This indicates that the noise most likely arises from noise on the ground planes, possibly caused by the digital circuitry and high speed communications.

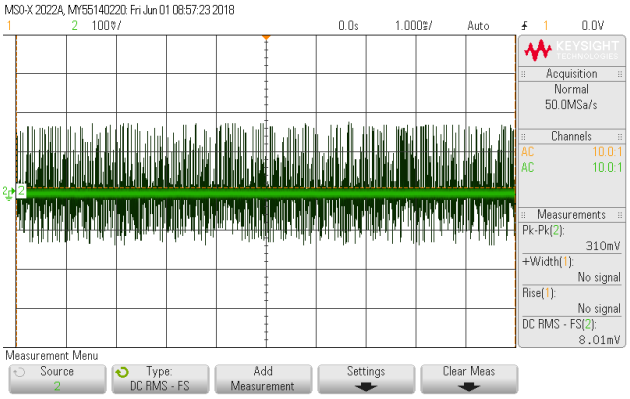


Figure 16 – Oscilloscope Capture of Output Noise

An oscilloscope capture of the distortion effect appears in figure 17. The input signal (top, yellow) is a 440 Hz sine wave at 600 mV peak-to-peak, and the output waveform (bottom, green) shows the distorted output. The output shows the top and bottom of the input signal clipped off, giving audible distortion, and proving the distortion effect works as intended.

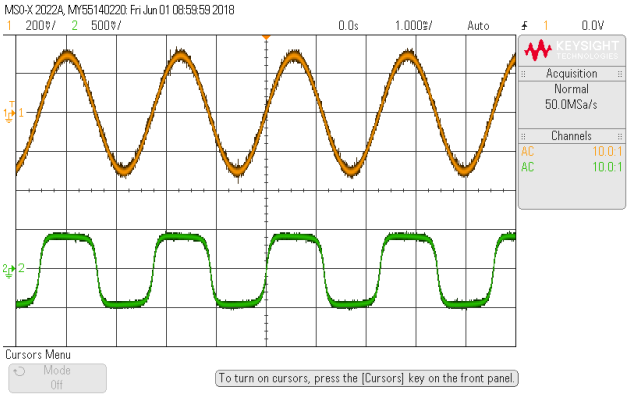


Figure 17 – Distortion Effect Oscilloscope Capture with Input (Top) and Output (Bottom) Traces

Testing the tuner function involves providing the pedal with sine waves of various frequencies and checking the reported frequency value against the true value. Testing revealed the tuner remains accurate within 5 cents for input signals ranging from 60 Hz-350 Hz, proving the tuner works as expected.

Testing of other pedal functions and effects relied mostly on listening to the output to determine if the audio signal appears as expected. Testing revealed the looper can record and play back large segments of music smoothly and without audible distortion, and the tap tempo function can accurately determine the tempo when tapping the 'Effect' footswitch. Every effect sounded similar to the expected output, proving that the pedal satisfies its main purpose of adding digital guitar effects to the audio signal.

Chapter 6: Conclusion and Future Work

Overall, this projects ended up being an incredible success. The final product meets the original specifications and accomplishes everything that was desired of it. The six effects programmed onto it at the project completion all sound good and their quality approaches the performance of commercially available guitar pedals. The looper function works perfectly and allows the user to record a good amount of music and play it back seamlessly. The tuner can accurately report the musical note played and allows easy tuning of the instrument. The user interface gives the necessary data to the user and allows clean and straightforward switching between different functions, effects, and parameters.

Despite the successes of this project, future versions of the pedal could improve on a number of features. Better layout and more careful arrangement of components could reduce the presence of excess noise at the output. The external flash memory chip cannot erase and program fast enough to keep up with the real time audio processing, so future versions could replace the flash memory chip with faster memory such as SRAM. This would allow the program to locate the audio buffer in the external memory. Internal storage limitations currently limit the audio buffer size to only one second, which could be greatly increased with more external RAM.

Replacing the 4x20 character LCD screen with a screen containing individually addressable pixels would allow the screen to display additional information in an aesthetically pleasing manner. This would allow custom icons for each effect and function, making the entire pedal appear more professional. This project did not explore this option due to the additional complexity of coding the graphics for the screen, but future versions can take advantage of the additional display freedom to improve the user interface.

In conclusion, the Digital Guitar Effects Pedal is a very fun device to play with, and can serve a useful purpose to guitarists everywhere. Future versions can improve the product even further, and eventually this device may even become commercially available.

Chapter 7: References

[1] A. McCabe, "Voices within the music: A brief history of guitar effects," *npr.org*, Dec. 13, 2014. [Online]. Available: https://www.npr.org/2014/12/13/370361269/voices-within-the-music-a-brief-history-of-guitar-effects, [Accessed: Nov. 17, 2017].

[2] G. Stevens, "A brief history of the guitar effects pedal," *guitaradventures.com*, Aug. 4, 2017. [Online]. Available: http://www.guitaradventures.com/guitar-effects-pedals-history, [Accessed: Nov. 17, 2017].

[3] V. Verfaille, U. Zölzer, and D. Arfib, "Adaptive digital audio effects (A-DAFx): A new class of sound transformations," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 14, no. 5, pp. 1817–1831, Sep. 2006.

[4] S. W. Smith, *The Scientist & Engineer's Guide to Digital Signal Processing*, 2nd ed. San Diego, CA: California Technical Publishing, 1999. [Online]. Available: http://www.dspguide.com/pdfbook.htm, [Accessed: Oct. 16, 2017].

[5] *IEEE Std 1233, 1998 edition*. DOI: 10.1109/IEEESTD.1998.88826.

[6] P. McLeod and G. Wyvill, "A smarter way to find pitch," in *Proceedings of the International Computer Music Conference*, (Sep. 2005), Barcalona, Spain, pp. 138–141. [Online]. Available: http://miracle.otago.ac.nz/tartini/papers/A_Smarter_Way_to_Find_Pitch.pdf, [Accessed: Oct. 15, 2017].

[7] T. Studer, "Electric guitar output voltage levels," *Tom's Guitar Projects*, Dec. 31, 2014. [Online]. Available: http://tomsguitarprojects.blogspot.com/2014/12/electric-guitar-output-voltage-levels.html, [Accessed: Nov. 9, 2017].

[8] C. C. Adams, D. V. Curtis, J. A. Brieske, *et al.*, "Standalone electronic module for use with musical instruments," U.S. Patent 7 678 985, Mar. 16, 2010. [Online]. Available: https://www.google.com/patents/US7678985, [Accessed: Oct. 15, 2017].

[9] *Standard for Audio, Video and Similar Electronic Apparatus, UL 60065*, Edition 8, Sep. 20, 2015.

[10] STMicroelectronics, "ARM® Cortex®-M4 32b MCU+FPU, 225DMIPS, up to 512kB Flash/128+4KB RAM, USB OTG HS/FS, 17 TIMs, 3 ADCs, 20 comm. interfaces," STM32F446xC/E datasheet, Feb. 2015, [Revised Sep. 2016]. [Online]. Available: http://www.st.com/resource/en/datasheet/stm32f446rc.pdf, [Accessed: Oct. 15, 2017].

[11] ——, "ST-LINK/V2 in-circuit debugger/programmer for STM8 and STM32," UM1075 User manual, Apr. 2011, [Revised Mar. 2016]. [Online]. Available: http://www.st.com/content/ccc/resource/technical/document/user_manual/65/e0/44/72/9e/34/41/8d/DM00026748.pdf/files/DM00026748.pdf/jcr:content/translations/en.DM00026748.pdf, [Accessed: Jan. 10, 2018].

[12] Tag-Connect, "TC2050-ARM2010 ARM 20-pin to TC2050 Adapter," [Online]. Available: http://www.tag-connect.com/Materials/TC2050-ARM2010.pdf, [Accessed: Jan. 10, 2018].

[13] STMicroelectronics, "Getting started with STM32F4xxxx MCU hardware development," AN4488 Application note, Jun. 2014, [Revised Dec. 2016]. [Online]. Available: http://www.st.com/content/ccc/resource/technical/document/application_note/76/f9/c8/10/8a/33/4b/f0/DM00115714.pdf/files/DM00115714.pdf/jcr:content/translations/en.DM00115714.pdf, [Accessed: Jan. 10, 2018].

[14] Abracon Corporation, "Miniature ceramic smd crystal," ABM3-25.000MHZ-D2Y-T datasheet, Sep. 2013. [Online]. Available: https://abracon.com/Resonators/abm3.pdf, [Accessed: Jan. 10, 2018].

[15] STMicroelectronics, "Oscillator design guide for STM8AF/AL/S and STM32 microcontrollers," AN2867 Application note, Jan. 2009, [Revised May 2017]. [Online]. Available: http://www.st.com/content/ccc/resource/technical/document/application_note/c6/eb/5e/11/e3/69/43/eb/CD00221665.pdf/files/CD00221665.pdf/jcr:content/translations/en.CD00221665.pdf, [Accessed: Jan. 10, 2018].

[16] Texas Instruments, "ADS8319 16-Bit, 500-kSPS, Serial Interface, Micropower, Miniature, SAR Analog-to-Digital Converter," ADS8319 datasheet, 2007. [Online]. Available: http://www.ti.com/lit/ds/symlink/ads8319.pdf, [Accessed: Oct. 15, 2017].

[17] ——, "DAC8551 16-bit, Ultralow-Glitch, Voltage-Output Digital-to-Analog Converter," DAC8551 datasheet, Apr. 2005, [Revised June 2017]. [Online]. Available: http://www.ti.com/lit/ds/symlink/dac8811.pdf, [Accessed: Oct. 15, 2017].

[18] Winbond Electronics Corporation, "3V 128M-Bit Serial Flash Memory with Dual/Quad SPI & QPI & DTR," W25Q128JV-DTR datasheet, Jan. 2015, [Revised Nov. 2016]. [Online]. Available: http://www.winbond.com/resource-files/w25q128jv%5C%20dtr%5C%20revb%5C%2011042016.pdf, [Accessed: Oct. 15, 2017].

[19] Newhaven Display International, "Character Liquid Crystal Display Module," NHD-0420H1Z-FSW-GBW-33V3 datasheet, Nov. 2017. [Online]. Available: http://www.newhavendisplay.com/specs/NHD-0420H1Z-FSW-GBW-33V3.pdf, [Accessed: Jan. 20, 2018].

[20] Texas Instruments, "TPS560200 4.5-V to 17-V Input, 500-mA Synchronous Step-Down Converter With Advanced Eco-Mode^{TM}," TPS560200 datasheet, Sep. 2013, [Revised Feb. 2016]. [Online]. Available: http://www.ti.com/lit/ds/symlink/tps560200.pdf, [Accessed: Jan. 20, 2018].

[21] ——, "TLV354x 200-MHz, Rail-to-Rail I/O, CMOS Operational Amplifiers for Cost-Sensitive Systems," TLV3542 datasheet, Oct. 2016. [Online]. Available: http://www.ti.com/lit/ds/symlink/tlv3542.pdf, [Accessed: Apr. 19, 2018].

[22] ——, "LP295x Adjustable Micropower Voltage Regulators with Shutdown," LP2951 datasheet, Apr. 2006, [Revised Nov. 2014]. [Online]. Available: http://www.ti.com/lit/ds/symlink/lp2951.pdf, [Accessed: Jan. 20, 2018].

[23] STMicroelectronics, "STM32F446xx advanced ARM®-based 32-bit MCUs," RM0390 Reference Manual, Mar. 2015, [Revised July 2017]. [Online]. Available: http://www.st.com/resource/en/reference_manual/dm00135183.pdf, [Accessed: Oct. 15, 2017].

[24] X. Amatriain, D. Arfib, J. Bonada, *et al.*, *DAFX: Digital Audio Effects*, U. Zolzer, Ed. New York: John Wiley & Sons, 2002. [Online]. Available: http://www.music.mcgill.ca/~ich/classes/dafx_book.pdf, [Accessed: Oct. 16, 2017].

[25] T. L. Beauchamp and J. F. Childress, *Principles of Biomedical Ethics*, 6th ed. Oxford: Oxford University Press, 2008.

[26] "IEEE Code of Ethics," *ieee.org*, 2017. [Online]. Available: `https://www.ieee.org/about/corporate/governance/p7-8.html`, [Accessed: Nov. 9, 2017].

[27] M. Malko, J. Kovacevic, R. Peckai-Kovac, *et al.*, "Implementation of digital audio effects for electric guitar on DSP platform," in *19th Telecommunications Forum Proceedings of Papers*, (Nov. 22–24, 2011), Belgrade, Serbia, pp. 1099–1102.

[28] R. Ford and C. Coulston, *Design for Electrical and Computer Engineers*. McGraw-Hill, 2007.

Appendix A: Senior Project Analysis

1. Summary of Functional Requirements

   The Digital Guitar Effects Pedal takes an analog input audio signal from an electric guitar, performs digital operations on it, and outputs a modified analog audio signal to an external guitar amplifier. The digital operations performed by the pedal include standard guitar effects such as distortion, delay, and vibrato. The pedal can also record and play back segments of music continuously, as well as tune the attached instrument. Users select the desired pedal function from a simple user interface on the device. Further discussion of functional requirements appears in chapter 3.

2. Primary Constraints

   The pedal's processor speed represents the primary system constraint. The pedal must operate fast enough to sample the input audio stream, process and modify it, and output the modified audio stream. To maintain audio quality, the sample rate should exceed 44.1 kHz, so processor speed must greatly exceed this to allow time for data transfer and processing. Audio quality represents another important constraint; the input analog-digital converter and output digital-analog converter must have 16-bits of resolution, and converter data transfer speeds must allow for fast communication with the main processor. Maintaining audio quality also requires careful use of filtering and noise reduction techniques. A summary of the system constraints and specifications appears in chapter 2.

3. Economic

   Manufacturing this product requires the use of fabrication and assembly facilities, which operate using human capital. Assembling and testing this product utilizes human capital by creating jobs for workers. Development time represents another source of human capital, as people need to design and program the product. Funding the project consumes financial capital, as described in the paragraph below. The product's constituent parts represent manufactured capital, and companies such as ST Microelectronics and Texas Instruments create the parts used in the design. Creating these parts also consumes natural capital, as IC fabrication requires the use of silicon and other natural materials.

   Project costs mostly accrue during initial development; however, development can continue indefinitely to improve existing effects and add new features. Project benefits begin to accrue once initial development completes and customers can purchase the product. The project is self-funded, with reimbursement for parts up to $200 available from Cal Poly's EE department. The EE department also provides the test equipment needed to verify proper pedal operation. The development period should last until June of 2018, when a final product should become available. Once the project completes, customers may begin purchasing the product, but product development and support can continue for the product's lifetime. The

product's manufacturing lifetime could last for several years, at which point development may begin on an updated product. The individual products should have lifetimes over 10 years, allowing users to continue using it for a long time.

4. If manufactured on a commercial basis:

   The pedal could have sales around 10,000 units per year. The pedal manufacturing cost should remain below $100, as stated in the specifications in table I. A final purchase price of $200 is competitive with other commercial pedals. This gives an estimated profit of $1M per year. Assuming the national average electricity cost of 12 cents per kilowatt-hour, and assuming 2 hours of daily device use, the pedal costs 40 cents per year to operate at maximum power consumption. This figure should decrease since the pedal's normal operating power consumption represents a small fraction of the maximum power.

5. Environmental

   Manufacturing this product requires the use of various integrated circuits, whose fabrication has substantial environmental impacts. The IC fabrication process consumes vast amount of chemicals and uses substantial electric power, which deplete the earth's natural resources. The integrated circuits chosen for this project should originate from companies that strive to lessen their environmental impact on the world. The finished product does not require many natural resources to operate; from section 4 above, the product should consume less than 3 kilowatt-hours per year at maximum power, equivalent to running a typical desktop computer for 15 hours.

   The project directly impacts the environment in the use of electricity to design and test the product. Shipping required parts also impacts the environment through transportation fuel requirements. Improper disposal of this product by users could also constitute an environmental concern, and the design should consider the environmental impacts of disposal when selecting parts. This project should not directly impact other species, but waste from product fabrication may adversely affect the local ecosystem and therefore other species.

6. Manufacturability

   The pedal requires fabrication of a PCB to support the pedal electronics. PCB fabrication should not present any major difficulties for modern manufacturing facilities. The pedal enclosure contains the PCB, input and output jacks, power input, and the user interface. Manufacturing of the enclosure requires hand soldering and wiring, so the design should consider ease of installation when planning product layout. Possible issues in assembly arise from connecting different wires to the main PCB in the wrong location. The design should ensure that the wire connection locations are clearly marked and the design should utilize multiply different connectors to reduce the risk of incorrect installation. The device's small size should allow for easy packaging and shipping.

7. Sustainability

   Maintaining the completed system only requires electricity, as users should not need to interact with the internal electronics. Wear from normal use may inhibit proper device functionality, however the device design should prevent mechanical failures for a normal device lifetime. Device manufacturing consumes natural resources which do not return when the product is disposed, but the design should consider this and attempt to limit the environmental impact. Future designs may improve the device by making it easier and more sustainable to manufacture. Upgrading the design poses problems in development because of firmware rewriting, but the upgraded version may still copy many aspects of the original design.

8. Ethical

   This project follows the doctrine of Ethical Principlism as closely as possible [25]. The main stakeholders in the project include: the project creators, the end users, manufacturers, Cal Poly, other effects pedal makers, and the guitarist community. Creating this effects pedal gives more autonomy to most stakeholders, except for other effects pedal makers, since it may force them to create new products to compete. End users gain autonomy by having more options to choose from when purchasing, manufacturers have more autonomy since they now have more financial capital to expand their business, and Cal Poly has more autonomy since they can show this project as evidence that they create good engineers. The project strives towards non-maleficence for the reasons stated in section 9 below, by protecting the health and safety of its users. The project attempts to not harm its stakeholders in any way. The project strives toward beneficence by providing a useful product for the guitarist community and end users. The project follows an idea of justice and treating all stakeholders equally, further discussion of which follows in section 10.

   This project adheres to the IEEE code of ethics wherever possible [26]. The project follows aspect 1, making decisions consistent with the health, safety, and welfare of the public, by ensuring that all proper safety measures are taken to protect the users. Further discussion of this appears in section 9 below. The project follows aspect 2, avoiding conflicts of interest, by remaining open and transparent about device limitations to prevent false advertising. The design follows aspect 3, honesty in stating claims or estimates, by undergoing many experiments and reporting the results fairly. The project strives to follow aspect 4 by rejecting all forms of bribery. The project report satisfies aspect 5, improving the comprehension of technology, by documenting the entire design process so future students can learn from this project. The project satisfies aspect 6, improving technical competence, by experimenting and attempting new ideas. The project undergoes multiple design reviews and receives feedback from colleagues, which follows aspect 7, accepting honest criticism of technical work. The project attempts to treat all persons fairly (aspect 8) for the reasons described in section 10 below, however the nature of the product prevents some customers from benefitting as much as others. The project avoids injuring others (aspect 9) for the reasons stated in section 9 below. The project follows aspect 10, assisting colleagues in their professional development and adhering to the code of ethics, since other project team can assist in design reviews, and this project hopefully can help other teams fulfil their goals.

9. Health and Safety

   This product presents a minimal health and safety risk to users, as the product operates from a low voltage power supply (9 VDC) and users should not have to interact with the internal electronics. The product design should consider various fault protection strategies such as internal temperature sensing and fuses to prevent internal shorts and dangerous operating conditions. All inputs and outputs should have electro-static discharge (ESD) protection to prevent unintentional damage to the product. The product should consider electromagnetic interference (EMI) issues, and contain proper filtering and shielding to prevent unwanted emissions. The device should follow UL 60065: Standard for Audio, Video and Similar Electronic Apparatus to ensure user safety [9].

10. Social and Political

    The direct stakeholders of this project include: the project creators, the end users, various manufacturers, and Cal Poly. The project creators benefit since they learn valuable skills by developing the different project aspects. The end users benefit since they can use the product to improve their guitar playing, and maybe become professional musicians. The harm to end users comes from the cost of the product, which they must pay before they can see any benefits. The various manufacturers benefit since the product requires the use of different components purchased from them, which increases their profits. Cal Poly benefits from this project since having successful senior projects reflects well on the entire school, and shows they create good engineers. The indirect project stakeholders may include: friends and neighbors of customers, manufacturers of other guitar pedals, and the general guitarist community. Since the product generates an audio output, neighbors of end users may become annoyed at excessive audio noise levels. The responsibility to use the product in a courteous manner rests mainly in the hands of the users, since they can adjust the volume to prevent neighbors becoming annoyed. The manufacturers of other guitar pedals become affected by this product since it introduces competition to the market, which might force them to adapt and create new products. The guitarist community benefits from more guitar pedals on the market, since it gives them more choices when choosing one for their needs.

    Most end users benefit equally from the product, if they already own a guitar and amplifier. Users with higher quality guitars may have a different experience with this product than users with lower quality guitars, however all users may access all product features regardless of gear quality. Users who do not own guitars and amplifiers cannot fully utilize this product, however this product's marketing should clarify the need for the guitar and amplifier, hopefully preventing this source of inequality. The product should work for users all over the world, though the product interface focuses on the English-speaking community. A future product upgrade may add multiple languages to the interface to allow for a more worldwide market.

11. Development

    Developing the pedal requires a knowledge of PCB design software and a knowledge of digital signal processing. Schematic capture and board layout can be accomplished using KiCAD, an open source schematic editor. Firmware creation is accomplished using System Workbench for STM32, a free integrated development environment which allows easy programming of the

microcontroller chosen. Implementation of the digital effects requires research and testing to ensure proper and efficient operation. The digital effect algorithms originate from various papers and books on the topic of audio effects [4][24][27]. Implementation of the tuner also requires extensive research [6]. Interfacing with flash memory constitutes another design challenge [18].

Appendix B: Project Planning

This appendix displays the original project plan as it appeared in the final report for EE 460: Senior Project Preparation, delivered on November 17th, 2017. A discussion follows which analyzes to what degree this project plan matched the actual development process.

Gantt Chart

Figure 18 shows the project development schedule in Gantt chart format. The chart contains 3 sections, one for each senior project class: EE460, EE461, and EE462. The EE460 section contains the overall project planning stage, with a final project planning report due at the end of the quarter. Hardware design should occur during EE461, with time allotted for two hardware design-build-test iterations. An intermediate status presentation occurs near the middle of the quarter, and a status report and demonstration occur near the quarter's end. The bulk of firmware design should occur during EE462, with time allotted to write a senior project report draft near the middle of the quarter. Firmware design should complete by the time of the Senior Project Expo, and the final senior project report completes soon after.

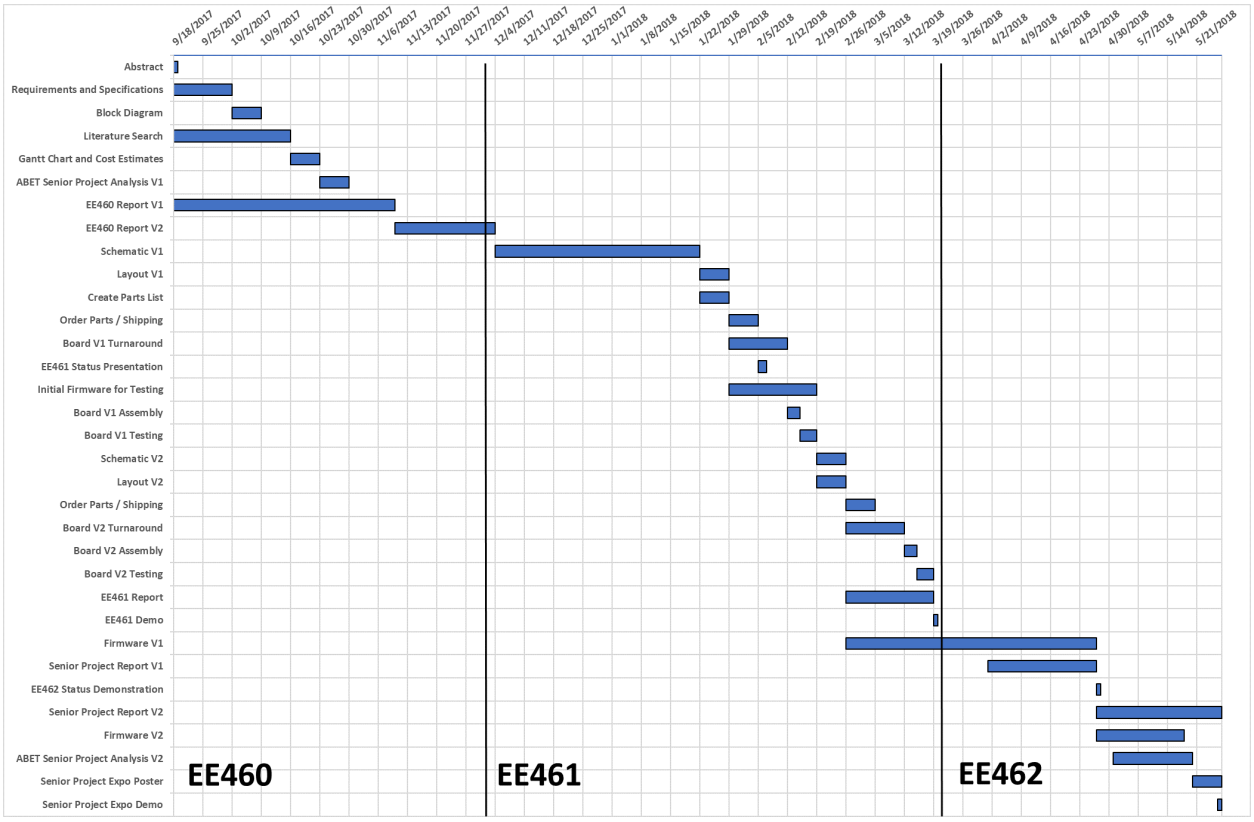


Figure 18 – Original Project Plan Gantt Chart

Cost Estimates

Labor and materials contribute most to overall project cost. Estimated materials costs derive from

current best estimates of required parts and number of board revisions. The labor costs derive from estimates of required hours spend, calculated using a weighted average of a realistic (tr), optimistic (to), and pessimistic (tp) estimates, according to the PERT formula provided in [28].

$$t_e = \frac{4 * t_r + t_p + t_o}{6} \tag{B.1}$$

Labor cost equates to project time at $50 an hour, which considers an hourly rate and a small amount of overhead since the school provides most required test equipment.

The planning stage, which comprises most of EE460, should take 42 hours (40 hours realistic, 30 hours, optimistic, 60 hours pessimistic), which at the stated hourly rate should cost $2100.

The hardware design stage, which occurs mainly during EE461, should take 67 hours (60 hours realistic, 40 hours optimistic, 120 hours pessimistic). This gives a cost of $3300. The hardware design stage consists of schematic design, board layout, and mechanical design. Costs may increase if the design requires more revisions.

The firmware design stage, which occurs mainly during EE462, should take 90 hours (80 hours realistic, 60 hours optimistic, 160 hours pessimistic), giving a cost of $4500. Firmware design duration depends on difficulties encountered during development, as well as the number of desired additional features.

Board fabrication costs and part costs comprise the total project materials cost. Assuming two board revisions gives an estimated board fabrication cost of $80. Total parts cost includes all necessary integrated circuits, such as the microcontroller, DAC, ADC, and memory IC. The STM32F777 microcontroller costs around $10, the LTC1864 ADC costs $13 per chip, the DAC8551 DAC costs $7 per chip, and the W25Q128JV-DTR flash memory IC costs $3 per chip. A suitable LCD display for the user interface could cost $30 for a single prototype unit. Assuming the purchase of two of each chip, and adding on miscellaneous parts required such as resistors and capacitors, the parts cost becomes around $160.

Table XI below summarizes the project cost estimates.

Table XI – Project Plan Cost Estimates

	Item	**Expected Cost**
Labor	Planning	$2100
	Hardware Design	$3300
	Firmware Design	$4500
	Manufacturing/Testing	$1100
Materials	Board Fabrication	$80
	Parts	$160
	Total:	$11,300

Comparison to Actual Development Process The overall project plan remained on track

over the project duration, and the project completed before the final deadline. The hardware development only required a single design-build-test cycle as the majority of the hardware worked the first time. The hardware development did continue partway into EE462 when it should have completed at the end of EE461, however this did not hinder firmware development as work could continue simultaneously. The estimate of development time turned out to be optimistic, with actual development time taking around 250 hours throughout the entire project. Project material costs remained lower than expected, due to only needing to fabricate a single PCB and getting several free samples of components.

Appendix C: Schematic

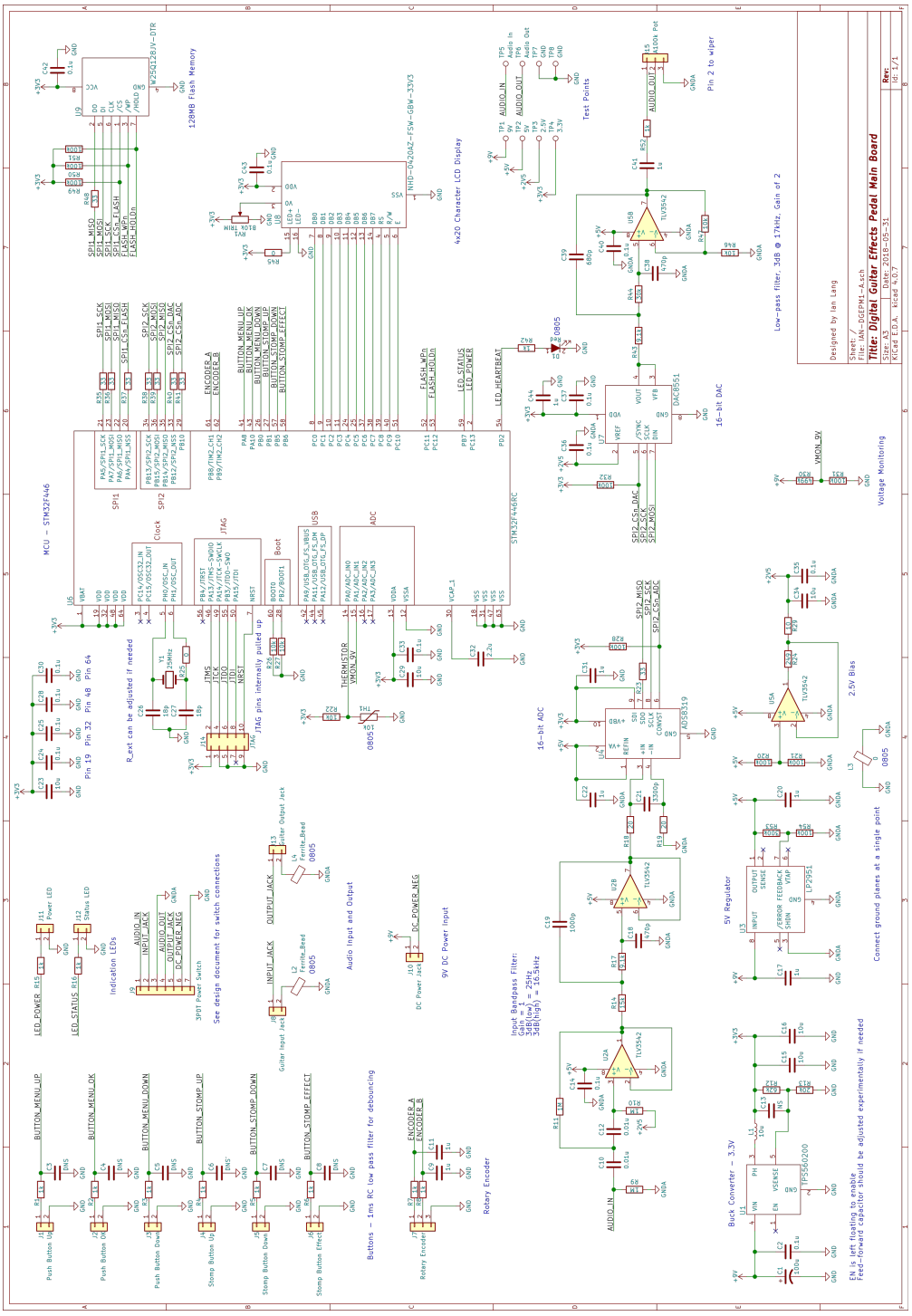


Figure 19 – Complete Schematic

Appendix D: PCB Layout

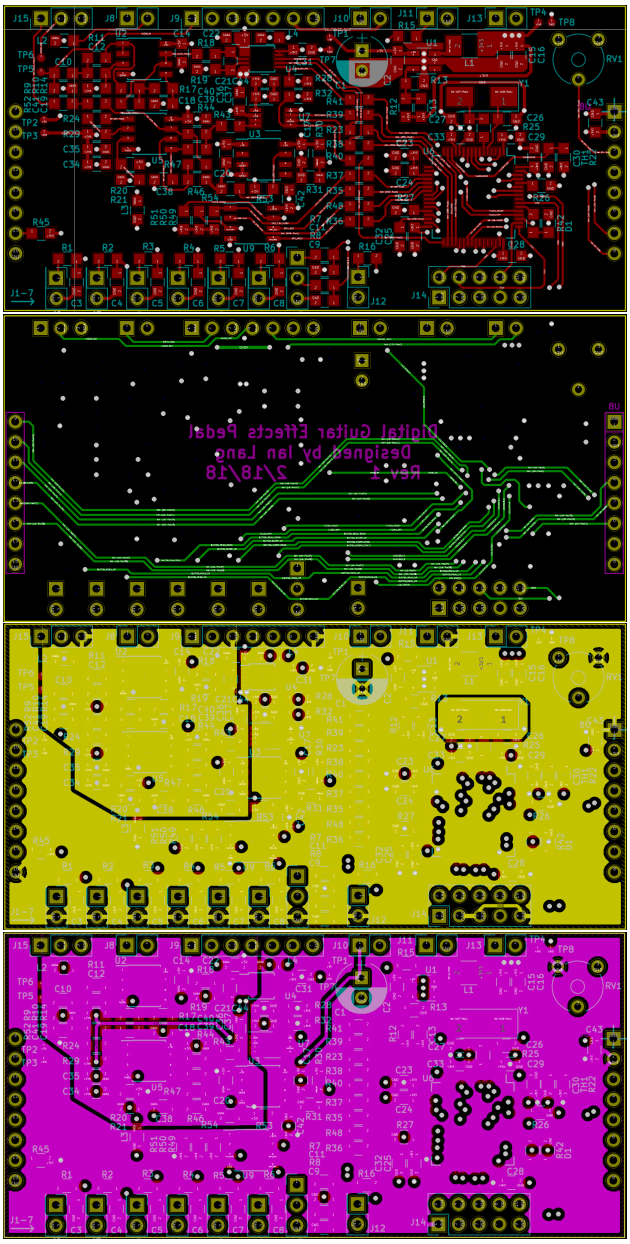


Figure 20 – PCB Layout (Layers From Top: Front, Back, Ground, Power)

Appendix E: Parts List and BOM

#	Part Name	Manufacturer	MFR Part Number	Quantity	Unit Price	Price	Purchase Place
1	STM32F446RC Microcontroller	ST Microelectronics	STM32F446RCT6	1	7.15	7.15	Digikey
2	ADS8319 16-bit ADC	Texas Instruments	ADS8319IDGST	1	10.84	10.84	TI.com
3	DAC8551 16-bit DAC	Texas Instruments	DAC8551IDGKR	1	6.27	6.27	TI.com
4	128 Mbit Flash Memory	Winbond	W25Q128JVSIM TR	1	2.46	2.46	Digikey
5	TPS560200 Buck Converter	Texas Instruments	TPS560200DBVR	1	0.96	0.96	TI.com
6	LP2951 5V Reg	Analog Devices	LP2951DR	1	0.6	0.6	Digikey
7	4x20 Character LCD	Newhaven Display	NHD-0420H1Z-FSW-GBW-33V3	1	19.1	19.1	Digikey
8	TLV3542 Op amp	Texas Instruments	TLV3542IDR	2	2.06	4.12	TI.com
9	0805 Resistors	Various	Various	50	0.01	0.5	Digikey
10	0805 Capacitors	Various	Various	40	0.1	4	Digikey
11	5mm LEDs (Blue and Red)	Various	Various	2	0.05	0.1	Tayda Electronics
12	25MHz Crystal	Abracon	ABM3-25.000MHZ-D2Y-T	1	0.69	0.69	Digikey
13	Rotary Encoder	TT Electronics	EN16-H20AF15	1	1.22	1.22	Digikey
14	SPST Stomp Switch	PIC	PBS-24B-4	3	1.97	5.91	Tayda Electronics
15	SPST Pushbutton Switch	PIC	PBS-105	3	0.25	0.75	Tayda Electronics
16	100k Potentiometer	ALPHA	A100K 17mm	1	0.5	0.5	Tayda Electronics
17	Knobs	Tayda Electronics	TYMF-B00	2	0.5	1	Tayda Electronics
18	2x40 0.1” Header	GTK	2x40 Pin Header Strip	1	0.21	0.21	Tayda Electronics
19	1/4” Jacks	Tayda Electronics	PJ699	2	0.87	1.74	Tayda Electronics
20	DC Power Jack	Tayda Electronics	DC Power Jack 2.1mm	1	0.16	0.16	Tayda Electronics
21	Power Switch	Tayda Electronics	Mini Toggle Switch 3PDT On-On	1	0.72	0.72	Tayda Electronics
22	Aluminum Enclosure	Mammoth Electronics	4S1590DD	1	8.6	8.6	Mammoth Electronics
					Total	77.6	

Appendix F: Selected Firmware Code

```c
#include "main.h"

int main(void)
{
    Initialize_Peripherals();   // Call initialization functions for each peripheral
    // Erase Flash memory
    FLASH_Erase_Blocks_IT(LOOPER_STARTING_FLASH_ADDRESS, LOOPER_MAX_FLASH_ADDRESS);
    LED_Blink_Heartbeat(HEARTBEAT_LED_PERIOD_MS);    // Set up heartbeat LED
    Audio_Enable();

    uint8_t systemState = SYSTEM_STATE_DEFAULT;      // Initialize the state machine

    //Main program loop, calls specific functions for different system states
    while (1)
    {
        switch(systemState)
        {
        case SYSTEM_STATE_DEFAULT:
            systemState = Run_State_Default();
            break;
        case SYSTEM_STATE_ADJUST:
            systemState = Run_State_Adjust();
            break;
        case SYSTEM_STATE_TUNER:
            systemState = Run_State_Tuner();
            break;
        case SYSTEM_STATE_LOOPER:
            systemState = Run_State_Looper();
            break;
        case SYSTEM_STATE_TEMPO:
            systemState = Run_State_Tempo();
            break;
        }
    }
}

/*
 * Initializes all peripherals required by the Digital Guitar Effects Pedal
 */
void Initialize_Peripherals(void)
{
    HAL_Init();                             // Reset of all peripherals
    SystemClock_Config();                   // Configure the system clock
    Audio_Init(TIMER_CLK_FREQ_MHZ);         // Initialize audio processing functions
    Delay_Init(TIMER_CLK_FREQ_MHZ);         // Initialize internal delay functions
    SPI1_Init();                            // Initialize SPI1 for external flash memory
    SPI2_Init();                            // Initialize SPI2 for ADC and DAC communication
    ADC_Init(TIMER_CLK_FREQ_MHZ);           // Initialize the ADC
    DAC_Init();                             // Initialize the DAC
    FLASH_Init(TIMER_CLK_FREQ_MHZ);         // Initialize the external flash memory
    LEDS_Init(TIMER_CLK_FREQ_MHZ);          // Initialize indication LEDs
    Encoder_Init(TIMER_CLK_FREQ_MHZ);       // Initialize encoder
    Buttons_Init(TIMER_CLK_FREQ_MHZ);       // Initialize buttons
    Display_Init(TIMER_CLK_FREQ_MHZ);       // Initialize display
}

uint8_t Run_State_Default(void)
{
    static uint8_t initialized = 0;             // Store if state has been initialized
    uint8_t nextSystemState = SYSTEM_STATE_DEFAULT;

    // Initialize the display if state has not been previously initialized
    if(initialized == 0)
    {
        Display_Clear();    // Clear the entire display

        // Line 1
        Display_Write_Number(Effect_Get_Current_Number(), 2, 0x01,
                             DISPLAY_ALIGN_RIGHT);
        Display_Write_Char('.', 0x02);
        Display_Write_String(Effect_Get_Current_Name(), 0x04);

        // Line 2
        Display_Write_String(Effect_Param_Get_Previous_Name(), 0x41);
        // Only draw up arrow if previous parameter exists
        if(strlen(Effect_Param_Get_Previous_Name()) > 0)
            Display_Write_Char(0x01, 0x53);

        // Line 3
        Display_Write_Char(0x7E, 0x14); // Draw right pointing arrow
        Display_Write_String(Effect_Param_Get_Current_Name(),  0x15);
        Display_Write_Number(Effect_Param_Get_Current_Value(), 3, 0x27,
                             DISPLAY_ALIGN_RIGHT);

        // Line 4
        Display_Write_String(Effect_Param_Get_Next_Name(), 0x55);
```

```
87          // Only draw arrow if next parameter exists
88          if(strlen(Effect_Param_Get_Next_Name()) > 0)
89              Display_Write(0x00, 1, 0x67);
90
91          // Turn on the status LED if the current effect is active
92          if(Effect_Get_Current_Activated_Status())
93              LED_Status(1);
94          else
95              LED_Status(0);
96
97          initialized = 1;    // Signify that state has been initialized
98      }
99      // Check if user has interacted with front panel, and process the button press
100     // or encoder rotation
101     if(Front_Panel_Event())
102     {
103         uint16_t buttonStatus = Front_Panel_Button_Status();    // Get status of buttons
104         switch(buttonStatus)
105         {
106         case BUTTON_STATUS_MENU_UP_PRESS:
107             Effect_Param_Previous();                // Go to previous parameter
108             initialized = 0;                        // Reinitialize screen
109             break;
110         case BUTTON_STATUS_MENU_OK_PRESS:
111             nextSystemState = SYSTEM_STATE_ADJUST;  // Change to adjust state
112             break;
113         case BUTTON_STATUS_MENU_DOWN_PRESS:
114             Effect_Param_Next();                    // Go to next parameter
115             initialized = 0;                        // Reinitialize screen
116             break;
117         case BUTTON_STATUS_STOMP_DOWN_PRESS:
118             Effect_Next();                          // Change to next effect
119             initialized = 0;                        // Reinitialize screen
120             break;
121         case BUTTON_STATUS_STOMP_DOWN_HOLD:
122             nextSystemState = SYSTEM_STATE_TUNER;   // Change to tuner state
123             break;
124         case BUTTON_STATUS_STOMP_UP_PRESS:
125             Effect_Previous();
126             initialized = 0;                        // Reinitialize screen
127             break;
128         case BUTTON_STATUS_STOMP_UP_HOLD:
129             nextSystemState = SYSTEM_STATE_LOOPER;  // Change to looper state
130             break;
131         case BUTTON_STATUS_STOMP_EFFECT_PRESS:
132             if(Effect_Get_Current_Activated_Status())   // Toggle effect activation
133             {
134                 Effect_Deactivate_Current();
135                 LED_Status(0);                      // Turn off LED if effect off
136             }
137             else
138             {
139                 Effect_Activate_Current();
140                 LED_Status(1);                      // Turn on LED if effect on
141             }
142             break;
143         case BUTTON_STATUS_STOMP_EFFECT_HOLD:
144             nextSystemState = SYSTEM_STATE_TEMPO;   // Switch to tempo state
145             break;
146         default:
147             // Change effect depending on direction of encoder rotation
148             if(Front_Panel_Encoder_Rotation() == ENCODER_COUNTERCLOCKWISE)
149             {
150                 Effect_Previous();
151                 initialized = 0;                    // Reinitialize screen
152             }
153             else if(Front_Panel_Encoder_Rotation() == ENCODER_CLOCKWISE)
154             {
155                 Effect_Next();
156                 initialized = 0;                    // Reinitialize screen
157             }
158             Front_Panel_Reset_Encoder_Rotation();
159             break;
160         }
161         Clear_Front_Panel_Event();                  // Signify button press registered
162         Reset_Front_Panel_Button_Status();          // Reset button status
163     }
164     if (nextSystemState != SYSTEM_STATE_DEFAULT)    // Detect state transitions
165     {
166         initialized = 0;    // Reinitialize next time state is entered
167     }
168     return nextSystemState;
169 }
170
171 uint8_t Run_State_Adjust(void)
172 {
173     static uint8_t initialized = 0;         // Store if state has been initialized
174     uint8_t nextSystemState = SYSTEM_STATE_ADJUST;
175
176     // Initialize the display if state has not been previously initialized
177     if(initialized == 0)
178     {
```

```
179         Display_Clear();    // Clear the entire display
180
181         // Line 1
182         Display_Write_Number(Effect_Get_Current_Number(), 2, 0x01, DISPLAY_ALIGN_RIGHT);
183         Display_Write_Char('.', 0x02);
184         Display_Write_String(Effect_Get_Current_Name(), 0x04);
185
186         // Line 2
187         Display_Write_Char(0x7E, 0x40); // Draw right arrow
188         char* paramName = Effect_Param_Get_Current_Name();
189         // Place name at center of screen
190         Display_Write_String(paramName,  0x49 - strlen(paramName)/2);
191
192         // Line 3
193         Display_Write_Number(Effect_Param_Get_Current_Value(), 4, 0x1C,
194                              DISPLAY_ALIGN_LEFT);
195         Display_Write_Number(Effect_Param_Get_Current_Min_Value(), 2, 0x14,
196                              DISPLAY_ALIGN_LEFT);
197         Display_Write_Number(Effect_Param_Get_Current_Max_Value(), 4, 0x27,
198                              DISPLAY_ALIGN_RIGHT);
199
200         // Line 4
201         // Draw bar graph of parameter value
202         Display_Write_Bar_Graph(0x54, 20, Effect_Param_Get_Current_Value(),
203                 Effect_Param_Get_Current_Min_Value(),
204                 Effect_Param_Get_Current_Max_Value());
205
206         // Set status LED if current effect is activated
207         if(Effect_Get_Current_Activated_Status())
208             LED_Status(1);
209         else
210             LED_Status(0);
211
212         initialized = 1;            // Signify that state has been initialized
213     }
214     // Check if user has interacted with front panel, and process the correct button press
215     // or encoder rotation
216     if(Front_Panel_Event())
217     {
218         uint16_t buttonStatus = Front_Panel_Button_Status();
219         switch(buttonStatus)
220         {
221         case BUTTON_STATUS_MENU_UP_PRESS:
222             Effect_Param_Previous();                 // Go to previous parameter
223             initialized = 0;                        // Reinitialize screen
224             break;
225         case BUTTON_STATUS_MENU_OK_PRESS:
226             nextSystemState = SYSTEM_STATE_DEFAULT; // Return to default state
227             break;
228         case BUTTON_STATUS_MENU_DOWN_PRESS:
229             Effect_Param_Next();                    // Go to next parameter
230             initialized = 0;                        // Redraw screen
231             break;
232         case BUTTON_STATUS_STOMP_DOWN_PRESS:
233             nextSystemState = SYSTEM_STATE_DEFAULT; // Return to default state
234             break;
235         case BUTTON_STATUS_STOMP_DOWN_HOLD:
236             nextSystemState = SYSTEM_STATE_TUNER;   // Change to tuner state
237             break;
238         case BUTTON_STATUS_STOMP_UP_PRESS:
239             nextSystemState = SYSTEM_STATE_DEFAULT; // Return to default state
240             break;
241         case BUTTON_STATUS_STOMP_UP_HOLD:
242             nextSystemState = SYSTEM_STATE_LOOPER;  // Change to looper state
243             break;
244         case BUTTON_STATUS_STOMP_EFFECT_PRESS:
245             if(Effect_Get_Current_Activated_Status())   // Toggle effect activation
246             {
247                 Effect_Deactivate_Current();
248                 LED_Status(0);                      // Turn off LED if effect off
249             }
250             else
251             {
252                 Effect_Activate_Current();
253                 LED_Status(1);                      // Turn on LED if effect on
254             }
255             break;
256         case BUTTON_STATUS_STOMP_EFFECT_HOLD:
257             nextSystemState = SYSTEM_STATE_TEMPO;   // Change to tempo state
258             break;
259         default:
260             // Change parameter value depending on rotation of encoder
261             if(Front_Panel_Encoder_Rotation() == ENCODER_COUNTERCLOCKWISE)
262                 Effect_Param_Decrease_Current_Value();
263             else if(Front_Panel_Encoder_Rotation() == ENCODER_CLOCKWISE)
264                 Effect_Param_Increase_Current_Value();
265             Front_Panel_Reset_Encoder_Rotation();
266             // Only update the current value to avoid having to redraw the entire screen
267             Display_Erase_Area(0x1C,4);
268             Display_Write_Number(Effect_Param_Get_Current_Value(), 4, 0x1C,
269                                  DISPLAY_ALIGN_LEFT);
270             // Update bar graph
```

```
271             Display_Write_Bar_Graph(0x54, 20, Effect_Param_Get_Current_Value(),
272                     Effect_Param_Get_Current_Min_Value(),
273                     Effect_Param_Get_Current_Max_Value());
274             break;
275         }
276         Clear_Front_Panel_Event();
277         Reset_Front_Panel_Button_Status();
278     }
279     if (nextSystemState != SYSTEM_STATE_ADJUST)      // Detect transitions out of state
280     {
281         initialized = 0;    // Reinitialize next time state is entered
282     }
283     return nextSystemState;
284 }
285
286 uint8_t Run_State_Tuner(void)
287 {
288     static uint8_t initialized = 0;      // Store if state has been initialized
289     uint8_t nextSystemState = SYSTEM_STATE_TUNER;
290
291     // Initialize the display if state has not been previously initialized
292     if(initialized == 0)
293     {
294         Audio_Tuner_Activate(); // Changes the audio processing to the tuner state
295         // Turn on status LED depending on whether output has been turned on
296         if(Audio_Tuner_Get_Output_Status())
297             LED_Status(1);
298         else
299             LED_Status(0);
300         Display_Clear();        // Clear the entire display
301
302         // Line 1
303         Display_Write_String("Tuner", 0);
304         Display_Write_String("Ref = ", 0x09);
305         Display_Write_Number(Audio_Tuner_Get_Reference_Freq(), 3, 0x0F,
306                              DISPLAY_ALIGN_LEFT);
307         Display_Write_String("Hz", 0x12);
308
309         // Line 2
310
311         // Line 3
312         Display_Write(0x00, 1, 0x1D); // Down arrow
313
314         // Line 4
315         // Initialize bar graph to half length
316         Display_Write_Bar_Graph(0x54, 20, 50, 0, 100);
317
318         initialized = 1;            // Signify that state has been initialized
319
320
321     }
322     // When audio buffer is full, start the tuner to calculate the correct note
323     if(Audio_Tuner_Get_Ready_Flag())
324     {
325         Audio_Tuner();  // Run the tuner function
326         uint8_t note;   // Variable to store detected note
327         int8_t cents;   // Variable to store number of cents
328         char* noteName = Audio_Tuner_Get_Note(&note, &cents);   // Get the tuner result
329
330         // Write the note name to the screen
331         Display_Erase_Area(0x49,2);
332         Display_Write_String(noteName,0x49);
333
334         // Update the bar graph depending on the cents variable
335         Display_Write_Bar_Graph(0x54, 20, (uint16_t)(cents + 50), 0, 100);
336
337         // Write the number of cents to the screen
338         Display_Erase_Area(0x21,3);
339         // Draw negative sign if cents < 0
340         if(cents < 0)
341         {
342             Display_Write_Char('-', 0x21);
343             cents = -1 * cents;
344         }
345         Display_Write_Number((uint16_t)cents, 2, 0x22, DISPLAY_ALIGN_LEFT);
346         Audio_Tuner_Reset_Ready_Flag();
347     }
348     // Check if user has interacted with front panel, and process the correct button
349     // press or encoder rotation
350     if(Front_Panel_Event())
351     {
352         uint16_t buttonStatus = Front_Panel_Button_Status();
353         switch(buttonStatus)
354         {
355         case BUTTON_STATUS_MENU_UP_PRESS:
356             Audio_Tuner_Increase_Reference_Freq();
357             Display_Erase_Area(0x4F,3);
358             Display_Write_Number(Audio_Tuner_Get_Reference_Freq(), 3, 0x0F,
359                                  DISPLAY_ALIGN_LEFT);
360             break;
361         case BUTTON_STATUS_MENU_DOWN_PRESS:
362             Audio_Tuner_Decrease_Reference_Freq();
```

```
363             Display_Erase_Area(0x4F,3);
364             Display_Write_Number(Audio_Tuner_Get_Reference_Freq(), 3, 0x0F,
365                                  DISPLAY_ALIGN_LEFT);
366             break;
367         case BUTTON_STATUS_STOMP_DOWN_HOLD:
368             nextSystemState = SYSTEM_STATE_DEFAULT; // Return to default state
369             break;
370         case BUTTON_STATUS_STOMP_EFFECT_PRESS:
371             // Turn audio output on and off, and update status LED
372             if(Audio_Tuner_Get_Output_Status())
373             {
374                 Audio_Tuner_Disable_Output();
375                 LED_Status(0);
376             }
377             else
378             {
379                 Audio_Tuner_Enable_Output();
380                 LED_Status(1);
381             }
382             break;
383         default:
384             break;
385         }
386         Clear_Front_Panel_Event();
387         Reset_Front_Panel_Button_Status();
388     }
389     if (nextSystemState != SYSTEM_STATE_TUNER)      // Detect transitions out of state
390     {
391         Audio_Tuner_Deactivate();
392         initialized = 0;    // Reinitialize next time state is entered
393     }
394     return nextSystemState;
395 }
396
397 uint8_t Run_State_Looper(void)
398 {
399     static uint8_t initialized = 0;     // Store if state has been initialized
400     static uint8_t eraseInProgress = 0; // Store if an erase operation is ongoing
401     uint8_t nextSystemState = SYSTEM_STATE_LOOPER;
402
403     // Initialize the display if state has not been previously initialized
404     if(initialized == 0)
405     {
406         // If an erase operation is in progress,
407         // indicate that on the screen and continue to check
408         if(FLASH_Get_Busy_Flag())
409         {
410             // Only update the screen on the first check
411             if(eraseInProgress == 0)
412             {
413                 Display_Clear();    // Clear the entire display
414                 Display_Write_String("Looper", 0);
415                 Display_Write_String("Erase in progress...", 0x14);
416                 eraseInProgress = 1;
417             }
418         }
419         else
420         {
421             Display_Clear();        // Clear the entire display
422             Display_Write_String("Looper", 0);
423             if(Audio_Looper_Get_Flash_Status() == 0)
424                 Display_Write_String("Ready to record", 0x14);
425             else
426             {
427                 Display_Write_String("Hold ", 0x14);
428                 Display_Write_Char(0x00, 0x19);
429                 Display_Write_String("To Erase", 0x1B);
430             }
431             eraseInProgress = 0;
432             // Blink LED if playback is active
433             if(Audio_Get_State() == AUDIO_STATE_PLAYBACK)
434                 LED_Blink_Status(500);
435             else
436                 LED_Blink_Status(0);
437             initialized = 1;            // Signify that state has been initialized
438         }
439     }
440     // Check if user has interacted with front panel, and process the correct button
441     // press or encoder rotation
442     if(Front_Panel_Event())
443     {
444         uint16_t buttonStatus = Front_Panel_Button_Status();
445         switch(buttonStatus)
446         {
447         case BUTTON_STATUS_STOMP_DOWN_HOLD:
448             Audio_Looper_Stop_Record();                 // Stop recording if in progress
449             Audio_Looper_Stop_Playback();               // Stop playback if in progress
450             Audio_Looper_Delete();                      // Delete recording
451             LED_Blink_Status(0);                        // Turn off blink
452             initialized = 0;                            // Reinitialize screen
453             break;
454
```

```
455         case BUTTON_STATUS_STOMP_UP_HOLD:
456             nextSystemState = SYSTEM_STATE_DEFAULT;          // Return to default state
457             break;
458         case BUTTON_STATUS_STOMP_EFFECT_PRESS:
459             switch(Audio_Get_State())
460             {
461             case AUDIO_STATE_DEFAULT:
462                 if(Audio_Looper_Get_Flash_Status() == 0)    // If flash memory available
463                 {
464                     Audio_Looper_Start_Record();           // Start recording
465                     LED_Status(1);                         // Status LED on
466                 }
467                 else
468                 {
469                     Audio_Looper_Start_Playback();         // Start playback
470                     LED_Blink_Status(500);                 // Blink LED
471                 }
472                 break;
473             case AUDIO_STATE_RECORD:
474                 Audio_Looper_Stop_Record();
475                 LED_Status(0);                             // Status LED on
476                 Audio_Looper_Start_Playback();
477                 LED_Blink_Status(500);
478                 initialized = 0;                           // Reinitialize screen
479                 break;
480             case AUDIO_STATE_PLAYBACK:
481                 Audio_Looper_Stop_Playback();
482                 LED_Blink_Status(0);                       // Turn off blink
483                 break;
484             }
485             break;
486             default:
487                 break;
488         }
489         Clear_Front_Panel_Event();
490         Reset_Front_Panel_Button_Status();
491     }
492     if (nextSystemState != SYSTEM_STATE_LOOPER)          // Detect transitions out of state
493     {
494         if(Audio_Get_State() == AUDIO_STATE_RECORD)
495             Audio_Looper_Stop_Record();                    // Stop recording
496         LED_Blink_Status(0);                               // Turn off LED
497         initialized = 0;    // Reinitialize next time state is entered
498     }
499     return nextSystemState;
500 }
501
502 uint8_t Run_State_Tempo(void)
503 {
504     static uint8_t initialized = 0;          // Store if state has been initialized
505     uint8_t nextSystemState = SYSTEM_STATE_TEMPO;
506
507     // Initialize the display if state has not been previously initialized
508     if(initialized == 0)
509     {
510         Display_Clear();    // Clear the entire display
511
512         // Line 1
513         Display_Write_String("Tap To Set Tempo", 0);
514
515         // Line 3
516         Display_Write_String("Tempo = ", 0x14);
517         Display_Write_Number(Effect_Get_Tempo(), 3, 0x1E, DISPLAY_ALIGN_RIGHT);
518         Display_Write_String("BPM", 0x1F);
519
520         LED_Blink_Status(60000/Effect_Get_Tempo()); // Blink status LED in time with tempo
521         initialized = 1;            // Signify that state has been initialized
522     }
523     // Check if user has interacted with front panel, and process the correct button
524     // press or encoder rotation
525     if(Front_Panel_Event())
526     {
527         uint16_t buttonStatus = Front_Panel_Button_Status();
528         switch(buttonStatus)
529         {
530         case BUTTON_STATUS_MENU_UP_PRESS:
531             Effect_Increase_Tempo();
532             initialized = 0;                          // Reinitialize screen
533             break;
534         case BUTTON_STATUS_MENU_DOWN_PRESS:
535             Effect_Decrease_Tempo();
536             initialized = 0;                          // Reinitialize screen
537             break;
538         case BUTTON_STATUS_STOMP_DOWN_PRESS:
539             Effect_Decrease_Tempo();
540             initialized = 0;                          // Reinitialize screen
541             break;
542         case BUTTON_STATUS_STOMP_UP_PRESS:
543             Effect_Increase_Tempo();
544             initialized = 0;                          // Reinitialize screen
545             break;
546         case BUTTON_STATUS_STOMP_EFFECT_PRESS:
```

```
547             Effect_Calculate_Tempo();
548             initialized = 0;                        // Reinitialize screen
549             break;
550         case BUTTON_STATUS_STOMP_EFFECT_HOLD:
551             nextSystemState = SYSTEM_STATE_DEFAULT; // Return to default state
552             break;
553         default:
554             // Change tempo depending on encoder rotation
555             if(Front_Panel_Encoder_Rotation() == ENCODER_COUNTERCLOCKWISE)
556             {
557                 Effect_Decrease_Tempo();
558                 initialized = 0;
559             }
560             else if(Front_Panel_Encoder_Rotation() == ENCODER_CLOCKWISE)
561             {
562                 Effect_Increase_Tempo();
563                 initialized = 0;
564             }
565             Front_Panel_Reset_Encoder_Rotation();
566             break;
567         }
568         Clear_Front_Panel_Event();
569         Reset_Front_Panel_Button_Status();
570     }
571     if (nextSystemState != SYSTEM_STATE_TEMPO)       // Detect transitions out of state
572     {
573         LED_Blink_Status(0);     // Turn off LED
574         initialized = 0;         // Reinitialize next time state is entered
575     }
576     return nextSystemState;
577 }
578
579
```

```
1  #include "clk_init.h"
2
3  /*
4   * System Clock Configuration
5  */
6  void SystemClock_Config(void)
7  {
8
9      RCC_OscInitTypeDef RCC_OscInitStruct;
10     RCC_ClkInitTypeDef RCC_ClkInitStruct;
11
12     /**Configure the main internal regulator output voltage
13      */
14     __HAL_RCC_PWR_CLK_ENABLE();
15
16     __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);
17
18     /**Initializes the CPU, AHB and APB busses clocks
19      */
20     RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
21     RCC_OscInitStruct.HSEState = RCC_HSE_ON;
22     RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
23     RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
24     RCC_OscInitStruct.PLL.PLLM = 15;
25     RCC_OscInitStruct.PLL.PLLN = 192;
26     RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
27     RCC_OscInitStruct.PLL.PLLQ = 2;
28     RCC_OscInitStruct.PLL.PLLR = 2;
29     if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
30     {
31         //_Error_Handler(__FILE__, __LINE__);
32     }
33
34     /**Initializes the CPU, AHB and APB busses clocks
35      */
36     RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
37             |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
38     RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
39     RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
40     RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
41     RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV4;
42
43     if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_5) != HAL_OK)
44     {
45         //_Error_Handler(__FILE__, __LINE__);
46     }
47
48     /**Configure the Systick interrupt time
49      */
50     HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000);
51
52     /**Configure the Systick
53      */
54     HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);
55
56     /* SysTick_IRQn interrupt configuration */
57     HAL_NVIC_SetPriority(SysTick_IRQn, 0, 0);
58
```

```
    // Enable peripheral clocks
    RCC->AHB1ENR |= (   RCC_AHB1ENR_GPIOAEN |   // GPIOA
                        RCC_AHB1ENR_GPIOBEN |   // GPIOB
                        RCC_AHB1ENR_GPIOCEN |   // GPIOC
                        RCC_AHB1ENR_GPIODEN |   // GPIOD
                        RCC_AHB1ENR_DMA1EN  |   // DMA1
                        RCC_AHB1ENR_DMA2EN  );  // DMA2
    RCC->APB1ENR |= (   RCC_APB1ENR_SPI2EN  |   // SPI2
                        RCC_APB1ENR_TIM6EN  |   // TIM6
                        RCC_APB1ENR_TIM7EN  |   // TIM7
                        RCC_APB1ENR_TIM2EN  |   // TIM2
                        RCC_APB1ENR_TIM3EN  |   // TIM3
                        RCC_APB1ENR_TIM12EN |   // TIM12
                        RCC_APB1ENR_TIM13EN |   // TIM13
                        RCC_APB1ENR_TIM14EN );  // TIM14
    RCC->APB2ENR |= (   RCC_APB2ENR_TIM1EN  |   // TIM1
                        RCC_APB2ENR_SYSCFGEN|   // SYSCONFIG
                        RCC_APB2ENR_TIM9EN  |   // TIM9
                        RCC_APB2ENR_TIM10EN |   // TIM10
                        RCC_APB2ENR_TIM11EN |   // TIM11
                        RCC_APB2ENR_SPI1EN);    // SPI1
}
```

```
void Process_Audio(void)
{
    *pInputBuffer = ADC_Get_Result();   // Save the input signal in buffer
    switch(audioState)
    {
    // In default state, run input through effects and send it to output
    case AUDIO_STATE_DEFAULT:
        *pOutputBuffer = Apply_Effects(pInputBuffer);
        break;
    // In tuner state, send the input directly to the output if output flag is set high
    case AUDIO_STATE_TUNER:
        if(tunerOutput)
            *pOutputBuffer = *pInputBuffer;
        break;
    // In record state, apply effects to input then save them to looper buffer
    case AUDIO_STATE_RECORD:
        *pOutputBuffer = Apply_Effects(pInputBuffer);
        *pLooperBuffer = *pOutputBuffer;
        pLooperBuffer++;    // Increment buffer pointer
        if(pLooperBuffer >= pLooperBufferEnd)   // If end of buffer
        {
            pLooperBuffer = pLooperBufferStart; // Wrap buffer pointer to beginning
            // Write the buffer to flash memory at current flash address
            FLASH_Page_Write_IT((uint8_t*)pLooperBufferInstructionStart, looperCurrentFlashAddress);
            looperCurrentFlashAddress += FLASH_PAGE_SIZE;   // Increment flash address
            // Check if too much data has been stored in flash
            if(looperCurrentFlashAddress > LOOPER_MAX_FLASH_ADDRESS)
            {
                Audio_Looper_Stop_Record();
            }
        }
        break;
    // In playback state, add the input data to the data from the looper
    case AUDIO_STATE_PLAYBACK:
        // Combine signals
        *pOutputBuffer = Apply_Effects(pInputBuffer) + *pLooperBuffer - DC_BIAS;
        pLooperBuffer++;    // Increment buffer pointer
        if(pLooperBuffer >= pLooperBufferEnd)   // If end of buffer
        {
            pLooperBuffer = pLooperBufferStart; // Wrap buffer pointer to beginning
            // Read data from flash memory into the buffer
            FLASH_Page_Read_IT((uint8_t*)pLooperBufferInstructionStart, looperCurrentFlashAddress);
            looperCurrentFlashAddress += FLASH_PAGE_SIZE;   // Increment flash address
            // Loop flash address around if it overflows
            if(looperCurrentFlashAddress >= looperEndFlashAddress)
            {
                looperCurrentFlashAddress = LOOPER_STARTING_FLASH_ADDRESS;
            }
        }
        break;
    }

    pInputBuffer++;                             // Increment buffer pointer
    if(pInputBuffer >= pInputBufferEnd)
    {
        pInputBuffer = pInputBufferStart;       // Wrap pointer around
        tunerReady = 1;                         // Indicate that tuner can calculate value
    }
}

void Audio_Tuner(void)
{
    // Change buffer pointers to ensure they have their own memory space
    pTunerInputBuffer = (float32_t*) pInputBufferEnd + 1;
    pTunerResultBuffer = pTunerInputBuffer + TUNER_BUFFER_SIZE + 1;

    // Convert all samples in buffer to 32bit floats
    int i;
```

```
69      for(i = 0; i < TUNER_BUFFER_SIZE; i++)
70      {
71          pTunerInputBuffer[i] = (float32_t)pInputBuffer[i];
72      }
73
74      // Take the mean of the samples to get the DC bias voltage
75      float32_t dcBias;
76      arm_mean_f32(pTunerInputBuffer, TUNER_BUFFER_SIZE, &dcBias);
77
78      // Subtract the DC bias from each sample
79      arm_offset_f32(pTunerInputBuffer, -1 * dcBias, pTunerInputBuffer, TUNER_BUFFER_SIZE);
80      // Perform an autocorrelation function
81      arm_correlate_f32(pTunerInputBuffer, TUNER_BUFFER_SIZE, pTunerInputBuffer,
82                        TUNER_BUFFER_SIZE, pTunerResultBuffer);
83      // Autocorrelation result gives symmetric output, only want result starting from center
84      pTunerResultBuffer += TUNER_BUFFER_SIZE - 1;
85
86      // Initialize the terms of the m'(t) calculation to r'(0)
87      float32_t mt_term1 = pTunerResultBuffer[0];
88      float32_t mt_term2 = pTunerResultBuffer[0];
89
90      uint8_t peakFinderState = 0;
91      uint16_t temp;
92      float32_t maxNSDF;
93      float32_t maxNSDFPeriod;
94
95      // Cycle through entire result buffer and calculate the NSDF incrementally
96      for(i = 0; i < TUNER_BUFFER_SIZE; i++)
97      {
98          // Calculate the NSDF
99          pTunerResultBuffer[i] = 2 * pTunerResultBuffer[i] / (mt_term1 + mt_term2);
100
101         // Subtract the appropriate x^2 from terms of m'(t)
102         mt_term1 -= pTunerInputBuffer[TUNER_BUFFER_SIZE - 1 - i] *
103                     pTunerInputBuffer[TUNER_BUFFER_SIZE - 1 - i];
104         mt_term2 -= pTunerInputBuffer[i]*pTunerInputBuffer[i];
105
106         // Algorithm to find peaks of NSDF
107         switch(peakFinderState)
108         {
109         case 0:     // Detect first negative going zero crossing
110             if(pTunerResultBuffer[i] < 0)
111             {
112                 peakFinderState = 1;
113                 temp = i;
114             }
115             break;
116         case 1:     // Detect first positive going zero crossing
117             if(pTunerResultBuffer[i] > 0 && (i - temp) > PEAK_FINDER_BUFFER)
118             {
119                 peakFinderState = 2;
120                 maxNSDF = pTunerResultBuffer[i];
121                 temp = i;
122             }
123             break;
124         case 2:     // Detect peak
125             if(pTunerResultBuffer[i] > maxNSDF)
126             {
127                 maxNSDF = pTunerResultBuffer[i];
128                 maxNSDFPeriod = i;
129             }
130             // Detect second negative going zero crossing
131             if(pTunerResultBuffer[i] < 0 && (i - temp) > PEAK_FINDER_BUFFER &&
132                     maxNSDF > TUNER_NDSF_THRESHOLD)
133             {
134                 peakFinderState = 3;
135             }
136             break;
137         }
138     }
139
140     // Check to see if detected peak is a valid result
141     if(maxNSDF > TUNER_NDSF_THRESHOLD && maxNSDFPeriod < 750 && maxNSDFPeriod > 50)
142     {
143         *pTunerResultPeriod = maxNSDFPeriod;     // Save the calculated period in buffer
144         pTunerResultPeriod++;                    // Increment buffer pointer
145         // Wrap pointer around if overflow
146         if(pTunerResultPeriod > pTunerResultPeriodEnd)
147             pTunerResultPeriod = pTunerResultPeriodStart;
148
149         // Use sorting algorithm to place previously recorded periods in order
150         int i;
151         int j;
152         float32_t temp;
153         float32_t sortedPeriod[TUNER_RESULT_AVERAGES];
154         arm_copy_f32(pTunerResultPeriodStart, &sortedPeriod[0], TUNER_RESULT_AVERAGES);
155         for(i = 0; i < TUNER_RESULT_AVERAGES - 1; i++)
156         {
157             for(j = i + 1; j < TUNER_RESULT_AVERAGES; j++)
158             {
159                 if(sortedPeriod[j] < sortedPeriod[i])
160                 {
```

```
                    temp = sortedPeriod[i];
                    sortedPeriod[i] = sortedPeriod[j];
                    sortedPeriod[j] = temp;
                }
            }
        }

        // Calculate the average period, using only the values around the median
        float32_t averagePeriod;
        arm_mean_f32(&sortedPeriod[0] + TUNER_RESULT_BUFFER,
                     TUNER_RESULT_AVERAGES - TUNER_RESULT_BUFFER * 2, &averagePeriod);
        // Calculate the detected note on the MIDI scale
        detectedNote = log10(CALIBRATED_TUNER_SAMPLE_RATE * 16 /(averagePeriod *
                                tunerReferenceFreq)) / NOTE_CALCULATION_DENOMINATOR;
    }
}
```

```
void Effect_Calculate_Tempo(void)
{
    static uint16_t previousTimerPeriods[TEMPO_NUM_AVERAGES] = {0}; // Save previous values
    static uint8_t index = 0;

    // Get value from timer and convert to period in ms
    uint16_t timerPeriod =  (uint32_t)TIM3->CNT * TIM3_PRESCALE / (TIMER_FREQ_MHZ * 1000);
    TIM3->EGR |= TIM_EGR_UG;            // Reset timer
    TIM3->SR &= ~TIM_SR_UIF;            // Reset flag
    TIM3->CR1 |= TIM_CR1_CEN;           // Enable timer

    // Check if period is within limits
    if((timerPeriod < TEMPO_MIN_PERIOD_MS) || (timerPeriod > TEMPO_MAX_PERIOD_MS))
        return;

    // Save period in buffer
    previousTimerPeriods[index] = timerPeriod;
    index++;
    if(index >= TEMPO_NUM_AVERAGES) // Wrap index around if overflow
        index = 0;

    // Calculate the average period from previous values
    uint8_t i;
    uint32_t averagePeriod = 0;
    for(i = 0; i < TEMPO_NUM_AVERAGES; i++)
    {
        averagePeriod += previousTimerPeriods[i];
    }
    averagePeriod = averagePeriod / TEMPO_NUM_AVERAGES;
    uint16_t newTempo = 60000 / averagePeriod;  // Calculate the tempo in BPM
    // Update tempo variable if valid
    if((newTempo > TEMPO_MIN_VALUE_BPM) && (newTempo < TEMPO_MAX_VALUE_BPM))
    {
        tempo = newTempo;
    }
}
```

```
uint16_t Apply_Effects(uint16_t* pInputData)
{
    // Cycle through available effects and call their corresponding function
    uint8_t i;
    uint16_t outputBuffer = *pInputData;
    for(i = 0; i < numEffects; i++)
    {
        if(effects[i].activated)
        {
            outputBuffer = effects[i].effect(&outputBuffer, &effects[i].params[0]);
        }
    }
    return outputBuffer;
}
```

```
static uint16_t distortion(uint16_t* pInputData, Effect_Param_TypeDef* effectParams)
{
    // Save parameters
    float32_t gain = effectParams[0].value;
    uint16_t boost = effectParams[1].value;

    // Get input value as float from -1 to 1
    float32_t x = (*pInputData - DC_BIAS)/(float32_t)33000;

    float32_t sign = (x > 0) - (x < 0); // Extract sign of number
    float32_t temp;
    arm_mult_f32(&x,&gain,&temp,1); // Use FPU to multiply input by gain
    temp = temp*(-1)*sign*boost;
    float32_t accumulator = 1;      // Initialize accumulator
    float32_t currentProd = 1;

    // Use Taylor series approximation for exponential function
    int i = 0;
```

```
19      for(i = 1; i <= 20; i++)
20      {
21          arm_mult_f32(&currentProd, &temp, &currentProd,1);  // Use FPU
22          currentProd = currentProd/i;
23          accumulator += currentProd;
24      }
25
26      float32_t outputData = sign*(1 - accumulator)/boost; // Calculate output data
27      return (uint16_t)(outputData*33000 + DC_BIAS);  // Convert back to integer for output
28  }
29
30  static uint16_t delay(uint16_t* pInputData, Effect_Param_TypeDef* effectParams)
31  {
32      uint16_t delay = (uint32_t)60 * FSAMPLE / tempo;     // Get delay in ms
33      uint16_t level = effectParams[1].value;              // Get level parameter
34      // Calculate the delay value
35      uint32_t delayValue = (uint32_t)*Audio_Get_Previous_Sample(delay) * level
36                            / effectParams[1].maxValue - DC_BIAS;
37      return *pInputData + delayValue;     // Add delayed sample to current sample
38  }
39
40  static uint16_t echo(uint16_t* pInputData, Effect_Param_TypeDef* effectParams)
41  {
42      // Calculate parameters
43      uint32_t delay = (effectParams[0].value * FSAMPLE) / 1000; // Get delay in ms
44      uint32_t currentDelay = delay;
45      uint16_t decay = (uint32_t)effectParams[1].value;
46      uint16_t outputData = *pInputData;
47      uint16_t numEchos = 0;
48
49      // Only calculate at max 10 echos to keep calculation short
50      while(numEchos < 10)
51      {
52          // If echo length is too long for buffer, return from function
53          if(currentDelay > AUDIO_BUFFER_SIZE)
54              break;
55          // Get delayed sample and add it to current sample
56          uint16_t* pEcho = Audio_Get_Previous_Sample(currentDelay);
57          outputData += (decay * (*pEcho - DC_BIAS)) / effectParams[1].maxValue;
58          currentDelay += delay;  // Increment delay to get next echo
59          decay -= decay/4;       // Increase decay to make next echo quieter
60          numEchos++;
61      }
62      return outputData;
63  }
64
65  static uint16_t vibrato(uint16_t* pInputData, Effect_Param_TypeDef* effectParams)
66  {
67      static float32_t t = 0; // Keep track of time variable for sine wave
68      // Frequency ranges from 1 - 11Hz
69      float32_t frequency = (float32_t)effectParams[0].value / 10 + 1;
70      // Depth ranges from 0 - 2ms
71      float32_t depth = (float32_t)effectParams[1].value * FSAMPLE / 50000;
72
73      // Use FPU to calculate sine
74      float32_t modulation = arm_sin_f32(2 * M_PI * frequency * t);
75      // Calculate index using sine wave result
76      uint16_t delay = (uint16_t)(1 + depth + depth * modulation);
77
78      // Ensure argument of arm_sin_f32 is from 0 to 2pi
79      t += 1/(float32_t)FSAMPLE;
80      if(t >= 1/frequency)
81          t = 0;
82
83      // Only return the delayed signal
84      return *Audio_Get_Previous_Sample(delay);
85  }
86
87  static uint16_t flanger(uint16_t* pInputData, Effect_Param_TypeDef* effectParams)
88  {
89      static float32_t t = 0; // Keep track of time variable for sine wave
90      // Frequency ranges from 1 - 11Hz
91      float32_t frequency = (float32_t)effectParams[0].value / 10 + 1;
92      // Depth ranges from 0 - 2ms
93      float32_t depth = (float32_t)effectParams[1].value * FSAMPLE / 50000;
94
95      // Use FPU to calculate sine
96      float32_t modulation = arm_sin_f32(2 * M_PI * frequency * t);
97
98      // Calculate index using sine wave result
99      uint16_t delay = (uint16_t)(1 + depth + depth * modulation);
100
101     // Ensure argument of arm_sin_f32 is from 0 to 2pi
102     t += 1/(float32_t)FSAMPLE;
103     if(t >= 1/frequency)
104         t = 0;
105
106     // Return the current sample added to the delayed sample
107     return *pInputData + *Audio_Get_Previous_Sample(delay) - DC_BIAS;
108 }
109
110 #define CHORUS_EFFECT_SCALE_FACTOR 2
```

```
111 static uint16_t chorus(uint16_t* pInputData, Effect_Param_TypeDef* effectParams)
112 {
113     // Start each sine wave at a different point
114     static float32_t t[5] = {0, 1/8.0, 1/6.0, 1/4.0, 1/2.0};
115     // Frequency ranges from 1 - 3Hz
116     float32_t frequency = (float32_t)effectParams[0].value / 50 + 1;
117     // Depth ranges from 0 - 2ms
118     float32_t depth = (float32_t)effectParams[1].value * FSAMPLE / 50000;
119 
120     int32_t outputData = *pInputData;
121 
122     // Cycle through different modulated delay lines
123     uint8_t i;
124     for(i = 0; i < 5; i++)
125     {
126         // Calculate delay from sine value
127         float32_t modulation = arm_sin_f32(2 * M_PI * frequency * t[i]);
128         uint16_t delay = (uint16_t)(1 + depth + depth * modulation);
129         t[i] += 1/(float32_t)FSAMPLE;
130         if(t[i] >= 1/(frequency))
131             t[i] = 0;
132         outputData += *Audio_Get_Previous_Sample(delay);
133     }
134     // Add all delayed samples together and divide by a scale factor to reduce volume
135     return (uint16_t)((outputData - 6 * DC_BIAS) / CHORUS_EFFECT_SCALE_FACTOR + DC_BIAS);
136 }
137 
```