Updated Project Proposal
User-Extensible Digital Effects Pedal

Group 31: Gerard Gallagher

January 29, 2023

Abstract

This is the updated project proposal for the UE-DEP pedal. The chapters correspond to the numbered sections in the template document. I have included some Appendices to provide context for how audio plugins work and why they are important.

This project will implement a digital effect pedal capable of loading and utilizing one or more user-supplied digital effects in the form of VST plugins. The main objective of this project is to produce a playable guitar effect pedal where the user can supply effects in the form of digital audio plugins. Previous attempts at such a pedal have been oriented towards lo-fi enthusiasts and DSP students who are comfortable coding their effects or significant portions of their system by hand. My design will attempt to make the approach as painless as possible for musicians and non-engineers, while retaining such "hackability" under the hood for audio engineers and DSP students.

This document is written with LyX, a LaTeX-based word processor. LyX allows for *version control*. Like LaTeX, it has excellent support for mathematical typesetting. If an editable format is required, send me an email[1].

[1]gg232@njit.edu

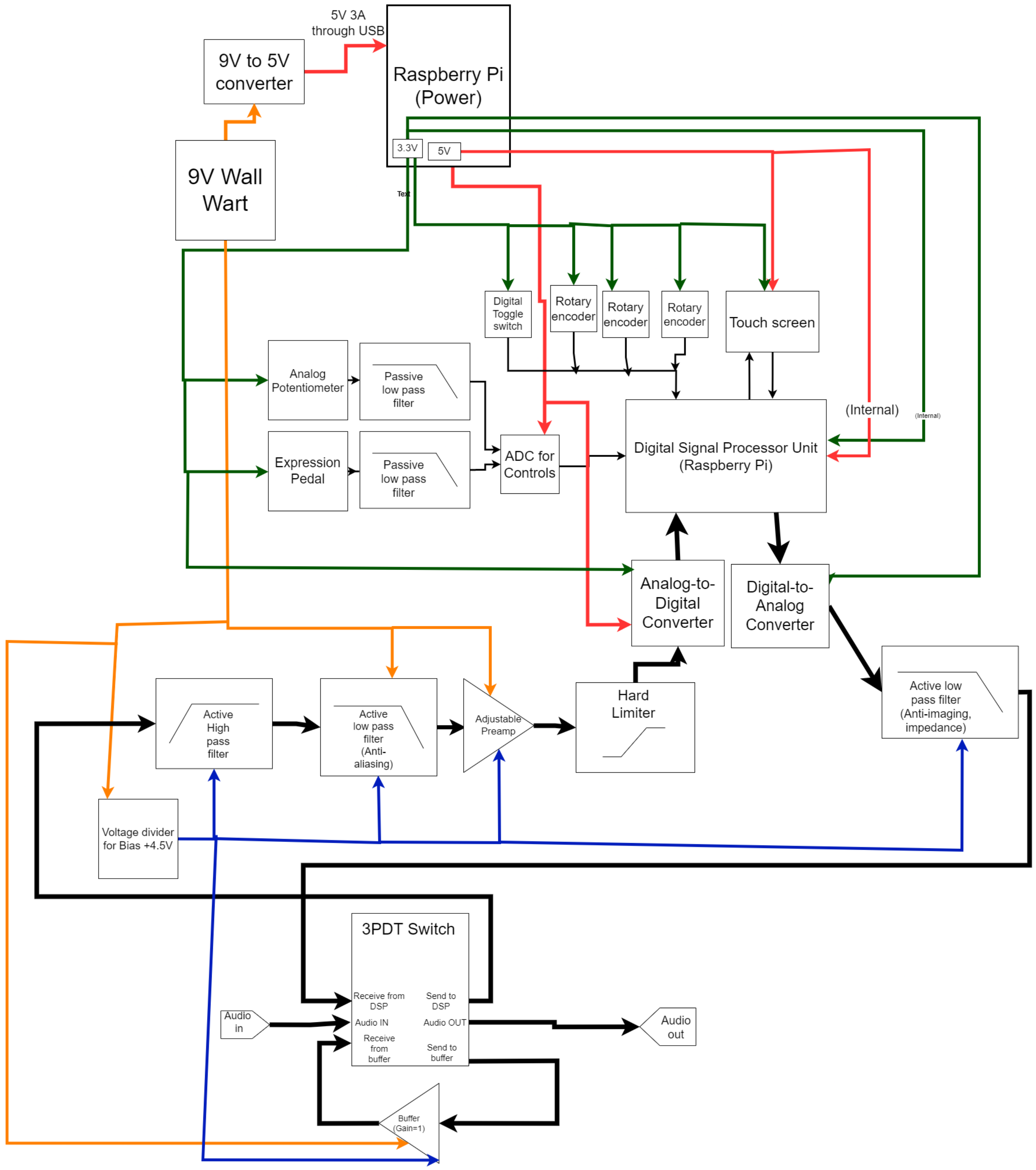
Contents

Chapter 1

Changes since Approval

For the sake of time, I have decided not to include MIDI input and output directly. MIDI control can come from the control bank, as some or all of them will receive MIDI events. MIDI control could still be used if the controlling device is capable of MIDI over USB. Most devices have this capability.

1.1 System Block Diagram



For the sake of readability, the power and bias wires have been colored. Specifically, 5V power is red, 3.3V power is green, and 4.5V bias for the op-amps are blue. Signal and control wires are black; thick wires are audio signals (digital or analog), and the thin ones are control signals. For the rotary encoders, these will be completely separate GPIO inputs, but the diagram was getting cluttered.

1.2 Technical Approach

The approach will be to use a Raspberry Pi 4 as a DSP unit, where the DSP algorithm is effectively specified and possibly modulated by the user

Although the algorithms used by the user are in general nonlinear[1], the device needs to be as linear as possible within its operating range. Colloquially, this translates to the requirement that the device imparts "no color of its own" upon audio signals.

The pedal will be designed to use a sampling rate of 44.1kHz. Higher sampling rates could be used, but the design will consider the 44.1kHz case only[2]. The Raspberry Pi's I2S hardware will provide the audio clock, and because it is generated by the Pi's microprocessor, it is controlled by the Pi and can be set through software. The ADC and DAC can both handle frequencies as low as 8kHz and as high as 384kHz. Preliminary tests have shown that the Raspberry Pi 4B can handle processing several effects with a sample frequency of 44.1kHz.

The preamp will be adjustable for the user by a potentiometer mounted to the exterior of the device. Although most electric instruments have a volume knob, I would like to give the user the ability to specify the gain needed (by ear + clipping LED) when the instrument's volume knob is at maximum gain, so that they can set the headroom of the preamp-ADC combo to suit their playing style and particular instrument. The circuit will be based on a non-inverting opamp circuit.

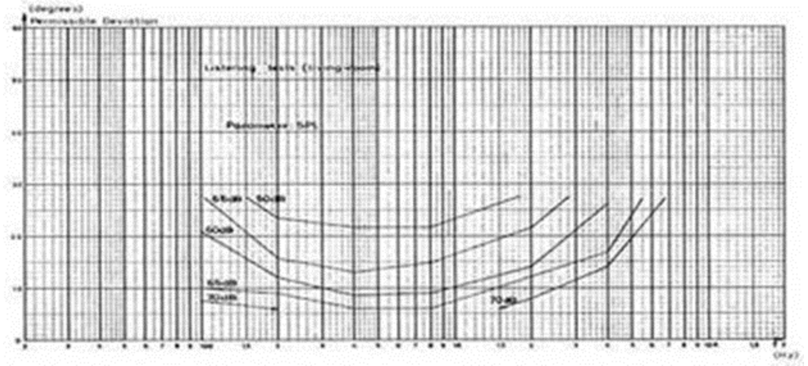
I have used the terminology "flat" several times without further specification. This will now be done. In signals and systems, a flat frequency response is one where the frequency response is constant up to infinity, and the phase response is perfectly linear up to infinity . This is impossible to build, but even if we allow for responses that are flat only in the passband, this is still a stringent specification for my system to respond to. The frequency response is known to affect the tonal quality of a sound, and it is through the idea of "balancing frequencies" that filtering is taught to audio engineers in recording schools. However, the subject of the audibility of phase response is controversial and steeped in misinformation. Basically, "reasonable" amounts of phase distortion are not audible when applied to regular music in typical settings. It can be heard if applied to sufficiently chosen test signals, and in anechoic chambers. The seminal paper on the issue, "On the Audibility of Midrange Phase Distortion in Audio Systems" by Lipshitz, et. al. , states the following four findings (emphasis is mine):

1. Even quite small midrange phase nonlinearities can be audible on suitably chosen signals.
2. Audibility is far greater on headphones than on loudspeakers.
3. Simple acoustic signals generated anechoically display clear phase audibility on headphones.
4. On normal music or speech signals phase distortion appears not to be generally audible, although it was heard with 99% confidence on some recorded vocal material.

[1]The human ear is "ridiculously" sensitive to nonlinearities in frequency (magnitude) response. Music production as a practice evolved in tandem with electrical engineering. In very brief: the sound of music was historically shaped by the circuits that transmitted the music and their elements. To this end, guitarists in the modern day seek out effects with terms like "diode clipping" or "tube distortion" because these nonlinearities, initially considered parasitic imperfections, were adopted by musicians as part of their art.

[2]As a producer, I typically choose 44.1kHz or 48kHz. Because the range of human hearing is below 20kHz, there is no musical use for frequencies any higher than that. Increaing the sampling frequency increases the size of a recorded .WAV file (holding all other variables, including time and input signal, constant). Effects that require oversampling can perform digital upsampling. Consequently, choosing a frequency far above 44.1kHz doesn't make sense. 48kHz makes math easier. Using a higher sample rate can reduce noise, but dithering can decorrelate the noise to make it "less intrusive". Practically, the noise is already either too low to be audible, or there is some other noise in the signal chain drowning it out. High-end studios might benefit from using a higher sample rate, but for the average producer, it is unnecessary.

The focus of my design is guitar signals. Although it is plausible that a user might plug a microphone into my design, vocal performances are not the focus of the project, and there already exist several better-adapted units for vocalists. To be more quantitative about this, I cite the following graph published as part of the study by Hansen and Madsen:



Predictably, the tolerance for phase distortion is lowest around the audio midrange, where human voices are center, and increases with frequency. As amplitude level increases, the tolerance also decreases across the board, but still seems to approach 30 degrees at some point in the audible band as frequencies fall to zero or rise to 20kHz. A caveat of this study is that it was conducted at "reasonable" levels, and guitar amplifiers are invariably "unreasonably loud," specifically much louder than the levels participants were subjected to in this study. Additionally, it was tested with sine waves.

The short version of the above is that phase distortion should be kept low, but a little bit towards the limits of the audible band is acceptable. The more noticeable problem will be, in my view, the frequency response of my filters, which needs to be as flat as possible in the passband. Passband ripple has a distinctive (and unpleasant) sound.

1.2.1 Input Signal Chain

Because the input filters are active filters with cutoffs separated by (far more than) an octave, it is permissible to design them independently and then cascade them, as opposed to designing a narrowband asymmetrical bandpass. I mulled over adding a notch at 60Hz to deal with ground hum, but decided against it for two reasons. Firstly, a digital notch could be trivially implemented by the user through an EQ plugin. Secondly, it is plausible that a guitarist might want to play notes near 60Hz, and even a properly designed notch could intrude in that space. A better solution is a toggleable ground lift.

The filters are unity-gain active filters. The input low-pass needed to be steep, whereas the requirements of the high-pass are more relaxed overall. As discussed below, a 3rd-order Butterworth low-pass was required. To do this with the fewest opamps (one), I chose a Sallen-Key 3rd-order low-pass[3]. The single-opamp biquad topologies (Sallen-Key and multiple feedback) are both insensitive enough to component changes, but sensitive to changes in *gain.* The Sallen-Key topology in particular is unstable for gains greater than three ($3\frac{\text{V}}{\text{V}} \approx 9.54\text{dB}$), which is inadequate for guitars with weak pickups. Therefore, it makes the most sense to me to make the preamp stage separate, and the filters unity gain.

The input signal needs to be biased in each stage to some positive DC voltage because the opamps are running off a single 9V supply. Biasing around 4.5V will give the signal lots of headroom to avoid clipping in the negative direction with an equal headroom in the positive direction. Most importantly, the "middle" of the opamp's operating range is where it is most linear.

The gain stage will be followed by a hard limiter circuit to protect the ADC. The single-ended ADC has a full-scale value of 5V (when $2V_{REF} = 5\text{V}$) First, there will be a diode clamper circuit to force the entire waveform to be positive. Then, there will be a diode clipper to clip off any voltages above 5V. The preamplified guitar signal will distort if it goes above 5V, and it will *audibly* clip. However, the ADC is safe. It is up to the guitarist to leave themselves enough headroom when they set the preamp gain.[4]

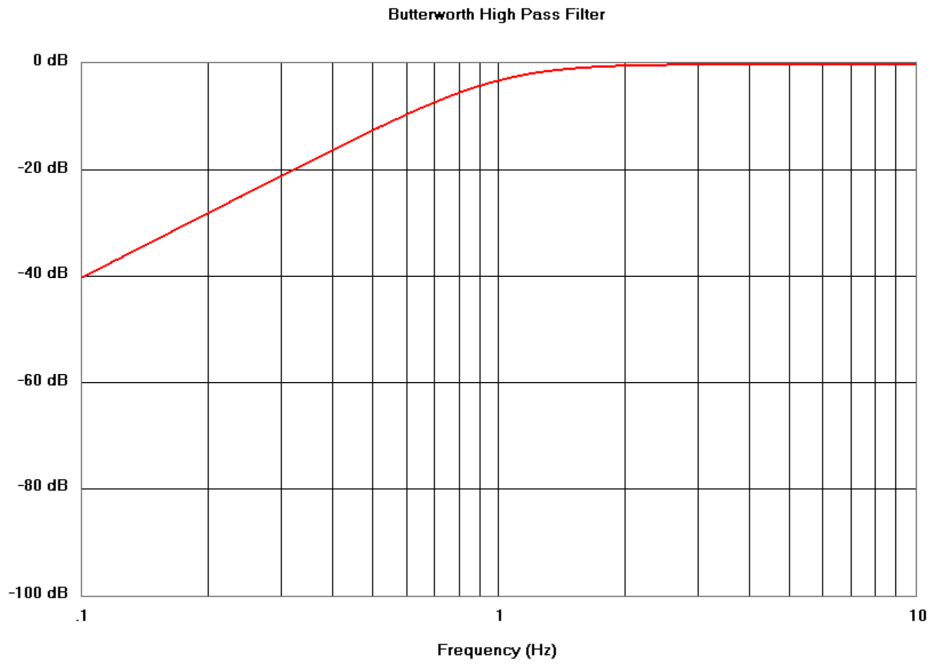
[3]Really, a "Sallen-Key 3rd-order filter" is a passive (1st-order) RC filter cascaded with a Sallen-Key second-order filter. Thus, it inherits some of the qualities of both filters.

[4]How does a guitarist know if they have enough headroom? Typically, they don't, at least not exactly. Gain knobs are

The filter op-amps will (tentatively) be NE5532 op amps. I need to confirm that they can work with a single supply.

1.2.1.1 High-pass filter

The purpose of the input low-pass filter is to block DC and filter out inaudible (sub-20Hz) garbage. To that end, the filter will be a unity-gain Sallen-Key high-pass filter with a cutoff of about 20Hz, slightly lower preferred. Because the Butterworth is flatter towards the bottom of the passpand, it is preferred over a Bessel characteristic.



I picked the filter characteristic based on Keck Taylor's freely available Analog Filter Program [1]. This program plots normalized frequency responses for a variety of filter characteristics, and gives filter coefficients and other data to implement the filter.

1.2.1.2 Low-Pass Filter

The purpose of the input low-pass is primarily to ensure that aliasing will not occur. Secondarily, it reduces high-frequency noise.

The criteria for the low-pass filter will be based on the following observation: an audio signal can be normalized to the interval $[0, 1]$. A single-ended quantizer with a bit depth of N bits can represent 2^N values[5]. Assuming uniform quantization, the size of the smallest change that can be represented is $\frac{1}{2^N} = 2^{-N}$. Converting to decibels, this means that the required stopband gain is $20\log 2^{-N}\text{dB} = -20\left(\log 2\right)N\text{dB} \approx -6.0206N\text{dB}$. The ADC and DAC can be operated with a bit depth of 24 bits, so the gain would have to be -144.49dB. This is a steep requirement. However, this does not account for the effective number of bits, which is a reduction of the usable dynamic range of the quantizer. The formula in this case[6] is $ENOB \approx \frac{SNR-1.76}{6.0602}$, which gives a minimum stopband attenuation of $20\log 2^{-ENOB}\text{dB} \approx (SNR - 1.76)\,\text{dB} = 99 - 1.76 = 97.24$dB. This is still rather steep. In the previous analysis, the architecture of the ADC and DAC were not considered. Both use $\Sigma\Delta$ modulation to oversample the signal, then decimate it. Practically, there is effectively a steep low-pass built into the filter.

For the ADC, the following graph is clipped from the datasheet:

typically set to a point of distortion, then "backed off" until it stops distorting for the loudest input, then backed off a bit more for good measure. Consequently, most electric guitarists should be able to use the gain control without much fuss. This is how they will likely set the gain on their amplifier too.

[5]The ADC is single-ended, so a sign bit is not needed.

[6]Technically, it should be $ENOB \approx \frac{SINAD-1.76}{6.0602}$, but SINAD was not given.

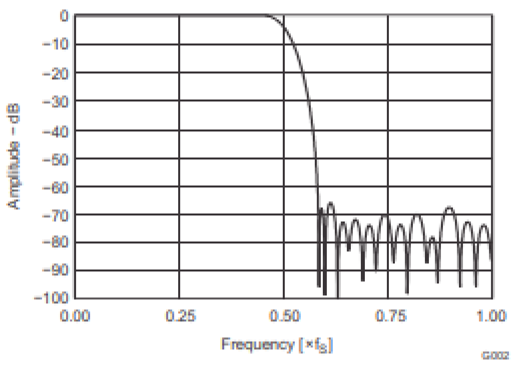
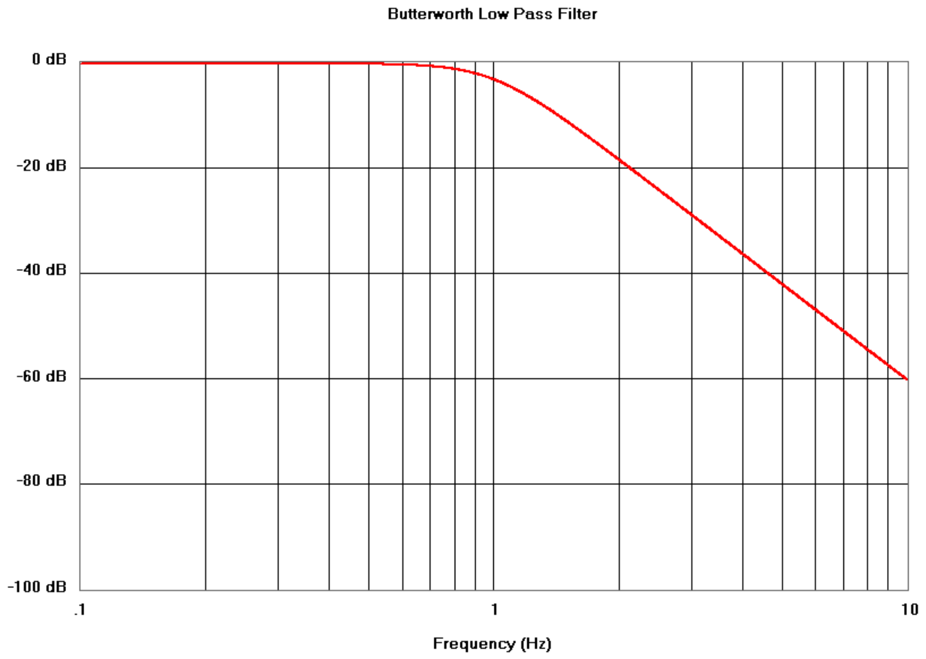


Figure 8. Decimation-Filter Frequency Response Stop-Band Attenuation Characteristics

Note that the graph was measured with an output sampling frequency of 48kHz, so it will slightly different for any other sample rate, but by plotting one curve it is implied that this curve is appropriate to use for any f_S. Choosing $f_S = 44.1\text{kHz}$, at 20kHz $\approx 0.45 f_S$, the attenuation is almost zero. At about $0.6 f_S \approx 26.4\text{kHz}$, the attenuation soars to 70dB, and the frequency response of higher frequencies seems to stay below that. Thus, $(97.24 - 70)\,\text{dB} = 27.24\text{dB}$. This is steep, but obtainable. However, at the input, the full range of hearing is not needed. A frequency-normalized 3rd-order Butterworth low-filter reaches about 28.5dBat $3Hz$, implying that a denormalized filter reaches that attenuation at $f_{3\text{dB}}$. If $f_{3\text{dB}}$is chosen to be $\frac{(44.1 \div 2)}{3}\text{kHz} = 7.35\text{kHz}$ or less, then the low-pass filter cascaded with the ADC's effective filter should ensure that aliasing is kept below the LSB. This is much higher than the electric guitar's typical frequency range, but it is plausible that a user might insert a nonlinear element before the input, specifically a distortion pedal. Anecdotally, presence controls on high-gain amplifiers typically boost a frequency in the region of 5kHz after the clipping stage, so the 7.35kHz figure aligns with my experience.



The Butterworth characteristic was chosen over a Bessel characteristic because the Bessel filter could not "reach" the required attenutation "in time", e.g. it is not steep enough when order is varied. Butterworth characteristic are often chosen for linear EQ plugins because they are steep and their phase response is "good enough". The other characteristics are chosen to optimize other qualities than transient response and passband frequency response, so they were ruled out. I tried a transitional Bessel-Butterworth, but the transition coefficient had to be set so close to 1 (pure Butterworth) that it was not worth the hassle to use the transitional design.

1.2.1.3 Input ADC

The ADC will be a breakout board based on the Texas Instruments PCM1801. This IC is a single-ended 24-bit sigma-delta ADC designed for cheap embedded audio applications. It requires both 5V power for the analog section, and 3.3V from the digital section. The input can swing between 0V and 5V max. This shows that it is a single ended ADC, which means that all prior op-amp circuits need to be biased. Additionally, all op-amps prior to the ADC need to be "comfortable" working with a single-ended supply voltage of 9V and signals up to 5Vpp. [2] It outputs digital words in I2S or left-justified format. For this project, I will be using the I2S format.

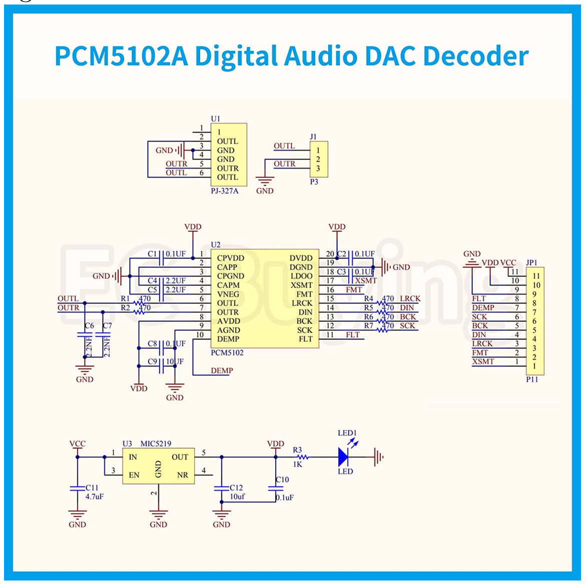
The DAC will be a breakout board based on the Texas Instruments PCM5102A. This IC is a single-ended sigma-delta DAC designed for cheap embedded audio applications. Unlike the ADC, this one only needs 3.3V for both sections.

Both the ADC and DAC are stereo devices, meaning that there are actually two inputs and outputs. Stereo operation is unnecessary for most instruments. Guitar is inherently monophonic, although some guitar reverbs and ping-pong delay effects take advantage of stereo audio. In order for the input to be stereo, a user needs to use a stereo effect before going into my pedal. Although this is plausible, it would make a lot more sense to use my pedal to implement that effect. Stereo reverb plugins are a dime a dozen, as are ping-pong delays and any modulation effect, which stereo operation is often used for. Thus, the input will be sent to one of the ADC inputs, and the other will be left unconnected. On the other hand, the outputs will be sent as stereo outputs. Unfortunately, that means the low-pass filter and output impedance matching will have to be matched.

1.2.2 Output Signal Chain

1.2.2.1 DAC

The DAC will be a breakout board based on the Texas Instruments PCM5102. It can accept data with bit depths of 16-, 24-, or 32-bits. Data is transmitted with the i2S protocol. The sellers of the board were kind enough to offer a schematic of what is on their board. This schematic is pasted below.



The important things to note are the output high-pass filter and the onboard headphone amplifier. The cutoff for their first-order high-pass[7] is $f = \frac{1}{2\pi RC} = \frac{1}{2\pi R_1 C_6} = \frac{1}{2\pi(470\Omega)(2.2\times10^{-9}F)} \approx 153992\text{Hz}$. This is well

[7]The capacitor and resistor subscripts are with respect to this diagram only. Also, both left and right channels have one filter each with the same circuits.

above the audible range. Unless the sampling frequency is lower than 154kHz $\times$ 2 = 308kHz, this filter does little to help with aliasing. Therefore, an additional high-pass is needed.

The headphone amplifier works out of the box and interfaces with the Pi without a fuss so far.

1.2.2.2 Output Filter

The output filter needs to remove aliased frequencies from the DAC output. Using similar reasoning as for the ADC, as it uses sigma-delta modulation in its architecture, the filter will be a 3rd-order Sallen-Key low-pass filter with cutoff just above 20kHz.

1.2.3 Bypass circuit

The bypass circuit will be a simple 3PDT footswitch wired so that one set of poles inserts the circuit, and the other passes the signal through a buffer. Buffered bypass vs true bypass (active vs. passive circuit, respectively) is a sore topic in the guitar world. Based on how I see my pedal being used, I would like it not to load other elements down the line. Therefore, a unity-gain non-inverting opamp buffer will be put between input and output. If there is enough time, I will drop in a toggle-switch to bypass the buffer circuit. [8]

1.2.4 Power

After some closer reading of the Raspberry Pi's specs, there is absolutely no way that a standard guitar pedal power supply will be sufficient to power the Raspberry Pi 4. These supplies, even the nicest ones, can typically only offer a few hundred mA of current. Keep in mind, the guitarist might want to drive other electronics on the pedalboard, such as a tuner, noise gate, or unique analog pedal for which no digital simulation exists. The Raspberry Pi 4 requires a whopping 3A of current, albeit at only 5V. Moving to a smaller model of Pi will not fix the problem. For comparison, the Electrosmash Pedal-Pi uses the Raspberry Pi Zero, which still requires 1A of current, and all other components essentially inherit all the power from the Raspberry Pi. Additionally, my understanding is that, according to Raspberry Pi documentation, the safest way to power the device is through the USB port, which is a 5V connection.

The system will be controlled primarily with the touchscreen, but secondarily through various knobs and switches external to the device. Right now, I have planned three digital knobs (rotary encoders), a footswitch toggle (separate from the bypass footswitch), an input for an expression pedal, and one sampled analog knob. Although designs vary wildly, to the point that there is not even a standard pinout, the basic principle of operation of an expression pedal is that a voltage is placed across a potentiometer operating as a voltage divider, whose action is controlled by the lever position. (There will also be at least an extra discrete resistor to limit the flow of current.) The Raspberry Pi will supply the input voltage and a MicroChip MCP3008 ADC (separate from the audio ADC) will convert the signal to SPI for the Raspberry Pi. This ADC is only 10 bits with a maximum of 200kHz sample frequency, although the limitations of SPI make such high rates impractical for the Pi. From experience, expression pedal outputs are nearly DC. If a user rocks the pedal back and forth 32 times in a second, that would yield a frequency of 32Hz, which allows for a Nyquist rate of 64Hz. Rounding up 100 Hz provides a guard band for aliasing. Note that this is a generous specification; a user who needs to oscillate the pedal will be better off doing so using software or an external MIDI CC pedal. The initial design will accommodate the pinout of a specific expression pedal, although the goal will be to implement a multi-position toggle switch that switches between the set of a few of the most common ones.

The touchscreen will be a Waveshare 4-Inch screen. The video output will be sent through HDMI, and control from the touch screen will be sent to the Pi via SPI. The touch screen requires both 3.3V power and 5V power.

[8] The type of bypass in a pedal is not something that a guitarist would want to change during a performance. It only needs to be a small toggle switch, not a full footswitch.

1.2.5 Software

The operating system will be the newest version of Raspberry Pi OS. Raspberry Pi OS is a derivative of Debian, which itself is a distribution of GNU/Linux. In general, music computers do not have real-time operating systems, even those at the highest-end studios, for the simple reason that Windows is not a real-time operating system. The average latency of Windows with a powerful enough CPU can be optimized nearly to zero, but it cannot be given an upper bound for *all* operations to finish. Practically, this means that Windows can choose not to prioritize the audio thread and allow it to miss a sample, which cerates clicks and pops that can damage speakers and the eardums of listeners. It is a bit difficult to quantify how powerful a CPU needs to be, but it depends heavily on the number and type of effects being processed. That being said, it has been possible for over a decade to play instruments through a typical office computer.

All versions of the GNU/Linux kernel include the ALSA (Advanced Linux Sound Architecture) API (application programming interface). When properly configured, it provides kernel-level support for sound devices to interact with software. It can interface with multiple input and output devices, as well as provide support for MIDI synthesis. To control and use inputs provided by ALSA, a JACK (JACK Audio Connection Kit) audio server must be started. It is from here that the user can set inputs and outputs, and where sample rate and bit depth can be controlled, if those bit depths are available. JACK is, from experience, indispensable for low-latency audio work. All that is left from here is the VST host. Several valid choices for VST host exist, and the one which I have chosen for now is the Carla audio plugin host. Carla is a free and open source modular plugin host. It can run plugins in VST format as well as those in the LADSPA, DSSI, LV2, AU, and JSFX formats. When supplied with the correct drivers, architecture emulation, and Windows API emulation, it can run VST plugins compiled on Windows and GNU/Linux, for x86 or x64 architectures. Specifically, a software called WINE (WINE Is Not an Emulator) is used to emulate Windows API calls. To use a Windows program on GNU/Linux, one typically installs the program through WINE, then runs it through WINE every time it needs to be used. The Raspberry Pi 4B uses a CPU with an ARM64 instruction set architecture. Consequently, x86 and x64 emulation is required. The software to provide this is Box86 and Box64, respectively.

Additionally, I am looking to modify the source code of Carla slightly to add in dedicated bypass switches. Right now, effects can be bypassed with MIDI CC inputs or by setting their respective "Wet" knob to zero, but in my view, there is nothing so helpful as a bypass switch. Additionally, I would like to see if it is feasible to implement my controls without sending MIDI CC inputs, which might somehow get sent to instruments further down the chain. Modifying the program is perfectly acceptable, as the program is licensed under GNU GPL-2.0 license. Basically, so long as the derivative work inherits the freedoms the license originally granted and states that it is modified, it is legal to copy, modify, and redistribute the file, possibly for money. (Any code I write or modify will be made freely available to the public the moment I think it is safe for another user to use.)

To make turning on the pedal a "turn-key operation," Raspberry Pi OS needs to be programmed to do several things upon startup. Firstly, ALSA needs to be set to take input from and send output to the Raspberry Pi's I2S ports. Next, it would be helpful to boot into a desktop environment. This way, in case anything goes wrong, the user has a "friendlier" desktop environment from which they can either reboot or reopen closed programs. Next, a JACK server needs to be started that takes in I2S input and spits out I2S output. Finally, a plugin host such as Carla needs to be opened with settings that make sense. Most obviously, Carla needs to be pointed to the JACK server, but it might make sense to load in a default session where the input is sent directly to output with an "identity operator" plugin that does nothing to demonstrate to the user how to add in a plugin. What will likely happen is that I will set up a default project file for Carla to load that has all the desired settings and plausibly some (bypassed) plugins ready to be enabled.

1.2.6 Control Bank

1.2.6.1 Sampled Analog Knob and Expression Pedal

1.2.6.1.1 Justification According to the datasheet [3], the SINAD of the control ADC is 61dB. The effective LSB of the 10-bit MCP3008 ADC is $2^{-ENOB}V_{REF} \approx 2^{-\frac{SINAD-1.76}{6.0602}}V_{REF} \approx 2 \times 10^{-3}V_{REF}$. If $V_{REF} = 5$V, then the LSB corresponds to a change of 10mV. Potentiometers for musical instruments usually have 270 degrees of rotation. Comparing the rotation to the number of levels yields the smallest rotation the ADC should be able to “see”: $\frac{270°}{2\times10^3} = 0.135°$. In contrast, the rotary encoders can only see $\frac{360°}{20} = 18°$. To control software that emulates an analog unit, the sampled knob can provide very fine control.

The potentiometer must be linear because the plugin designers usually hard-code a logarithmic taper into their plugins. An audio taper on top of that would make the knob behave differently from its GUI counterpart. Other than the particular resistor, the circuit will be the same as the expression pedal circuit.

1.2.6.1.2 Implementation These circuits will be simple variable voltage dividers with current-limiting series resistors connected to a channel of the Adafruit MCP300 10-bit ADC[3]. The ADC has 8 channels, so the control circuits could later be duplicated to add more controls. It uses a SPI interface, of which the Pi has three.

For all variable-voltage controls, a driver will be written that converts SPI inputs to MIDI or OSC data. Driver will likely be written in Python for simplicity, but there are more details in the next section.

1.2.6.2 Push-button control and rotary encoders

This will be a typical pullup-pulldown, ut with two extra additions. The first will be a capacitor to smooth out the bouncing effect. If these controls were linked to a plugin knob through software, a bounce would sound like someone is violently turning the knob. If this is, for example, a volume knob, this makes the control unusable. To eliminate the bouncing effect, I will use an inverting Schmitt trigger IC, specifically a 74HC14 because I have one on hand. This will add a hysterisis to the circuit. Practically, this means that the output will only change when the signal is “serious” about settling high or low, and it will not change if it is corrupted by noise during the transition.

I will initially write the driver software using the official Raspberry Pi GPIO libraries, which interface with Python. The drivers will accept GPIO signals and output MIDI or OSC data. For real-time audio *processing*, Python is unacceptably slow. For non-audio tasks like control or GUI rendering, Python is usually fine, especially for simple tasks. If Python is too slow, the driver software will be rewritten in C++ with the `pigpio` library. Whatever language is used, this program will be started automatically every time the Pi starts.

Chapter 2

Preliminary Work

As far as the software is concerned, support for *Linux-compiled* plugins in x64, x86, and ARM64 architectures has been implemented. VST and JSFX format plugins have been tested, but Carla should be able to handle any plugin in any reasonable format designed for Linux. Limited support for 32-bit Windows has been achieved with WINE and VSTHost. The DAC has been hooked up in both hardware and software.

I also did a lot of research into RTOS's and their usage in audio. What I found is the Elk Audio OS, a derivative of Linux with a real-time cokernel for audio processes and built-in VST support. They explicitly support the Raspberry Pi. Basically, the task of optimizing Linux for audio purposes has already been completed by people far more qualified than I to do so. However, it is not *accessible*: it has no GUI, because it is intended for embedded, often headless operation. Therefore, if for some reason the Pi's performance is garbage with Raspberry Pi OS, then the project can be moved to Elk Audio OS as a last resort. Unfortunately, I would have to reinstall Box86, Box64, and WINE, which is already a painful process on Raspberry Pi OS. Additionally, I would have to install a desktop environment and dependencies for all the above. The point of this paragraph is to indicate that while a RTOS would increase performance, the 8GB RAM and 1.5GHz CPU of my Raspberry Pi 4 should be enough to run plugins. If more computing power is needed, then I will use a different board.[1]

[1]Unfortunately, any better microcomputer would make this device prohibitively expensive for most people, although I could afford it for the prototype.

Chapter 3

Components

Things I have

- Raspberry Pi 4 Model B
- ADCs (PCM1808 and MCP3008)
- Resistors (1/4W and 1/2W, 0-1Mohm)
- Electrolytic Capacitors Film
- Capacitors
- Linear and audio-taper potentiometers and plastic knobs
- Op amps
- Touch screen
- M-Audio EX-P expression pedal
- DAC (PCM5102)
- XLR-1/4in combination jacks
- Ordinary 1/4in jacks

Things I need

- Case

Chapter 4

Updated Project Schedule

Assume Week 0 is that which starts on 01/17/2023 and ends on 01/23/2023.

Milestone	Week
Finalize design of input filters, input preamp, output post-amp, and output filter	3
Build input and output circuitry	4
Implement touch screen	5
Finalize audio software	6
Design and build control network, including software	8
Design and build expression inputs	10
Drill holes in case, place everything into the case	12

Appendix A

Introduction to Music Production

A.1 Audio terminology and standard workflow

Once a song is written and the performers know how to play it, the song is recorded. Typically, this is achieved by having the band play their parts while their instruments are recorded by a set of microphones. There is usually at least one microphone per "instrument", and several for the drum kit. Unless the engineer sums the outputs of the microphones before the conversion to digital (which might commit serious phase issues to "tape"; so it is never done), each microphone will record to one track of a multitrack recorder.

Historically, a multitrack recorder was literally a bank of reel-to-reel tape recorders, which were summed in the analog domain. This practice survives in a different form: the "multitrack metaphor" in the design of digital audio workstations. If the entire band is recorded as a single performance, then all the microphone inputs will be sent to distinct ADC's, which will convert the audio signals to a multichannel digital signal in a specially designed low-latency audio interface. (Basically, the audio interface is an audio input or output where a driver that circumvents the operating system's default driver is used.) Unless the engineer permanently sums the tracks in the digital domain, each track will be recorded to a separate uncompressed digital audio file.

Despite all the steps involved in this process, this can be easily done in real time. Actually, we often add real-time effect plugins to the tracks, and we also send the digital audio back out to the performers where, after an obligatory DAC, they use this audio to monitor their performance.

In short, audio tracks are recorded and converted to digital individually. These audio files are processed by either analog gear or digital plugins, then digitally summed with weights decided by the engineer. The process of preparing the tracks for summing and balancing the weights is called mixing. After mixing, the track is taken to a professional with nice analog gear and a deep understanding of audio for mastering, which prepares the final recording for distribution.

This is not some idealized, theoretical concept of audio production. In fact, this is the typical workflow of a modern budget studio. In the modern day, almost no information is lost from coversion between digital and analog, or vice versa, even in low end consumer devices. Basically, so long as it is "designed for audio," it is a safe bet that the ADCs and DACs are practically transparent. It would take a spectacularly incompetent engineer to release an audio output that sounds "digital" today.

For home studios and bedroom bands, the process is modified slightly. Because consumers typically lack the hardware and inputs to record an entire band, the band might take turns recording parts with one or two microphones or direct inputs. The first band member, typically the drummer, records to a click track, which ensures a consistent tempo. Then, each band member records on top of the previous musicians work in a new track.

Higher-end studios do use analog gear because they can afford to do so. However, the vast majority of processing will still be in the digital domain because, if used competently, digital systems can sound just as good as analog ones.

A particularly helpful example is the linear equalizer. Assuming a perfect performance and everything else is perfect, a real track needs, at the bare minimum, high-passes and low-passes to get rid of low- and high-frequency garbage and help separate the different instruments. Although nearly linear equalizers can be built in the analog domain, they are expensive and difficult to design. Digital linear equalizers are a dime a dozen, almost perfectly linear, and use trivial CPU. Crucially, an engineer can insert as many concurrent real-time instances of a digital EQ plugin as they desire. An analog equalizer will usually have two stereo inputs, so to use it on more than two tracks, an engineer needs to convert the tracks back to analog, play them through the equalizer, and rerecord and convert the tracks back into the digital domain. Although this process is not worth the effort for an equalizer, it can make good sense to rerecord guitar amps and vintage dynamics processors, especially if there is no plugin emulation for the particular analog element. Still, for analog gear that has a plugin counterpart, it is much easier to use the digital version. A common occurance is when an engineer is handed a vocal track recorded in an untreated room, and as a result has unwanted resonance corresponding to the room modes. A linear EQ is typically dialed in to provide a narrow cut at the resonant frequency. This preserves the character of the vocal track while removing the room mode. Then, the engineer can process the track further without having to worry about the room mode.

Except for a handful of purists, audio engineers will reach for whatever tool provides the best sound for the job. In high-end studios, there is just as much chance of the engineer reaching for a piece of analog gear as there is of the engineer adding a premium plugin to the channel strip in their DAW. It is in fact access to the latter that budget studios will often advertise as their selling point.

The reason why all this is relevant for my project is that in the early 2000's, bands that could afford it began to use high-end laptops running mixing software as effect processors. For example, Periphery is a progressive metal band that inserts digital effects into their guitar signal chain to give their guitar sound an "electronic" timbre for small parts of the song. Because they do most of their effect processing digitally using the same signal chain as they used in the studio, their live performances sound very close to their albums.

My concept is to create a dedicated device to process guitar signals using existing audio plugins for live usage. One goal is to allow a musician to "drag-and-drop" their signal chain into my box and play through it in real time. Another is to create a platform where makers can write and test audio DSP algorithms in context with the rest of their hardware signal chain.

A.2 Subjectivism in digital audio

The thesis of this section is that while my pedal should be able to produce "correct", "good-sounding" outputs in a way that can be emprically measured, there will always be a class of people who intrinsically reject digital audio technology and cannot be convinced (even with evidence) that my pedal can help them.

In philosophy, *subjectivism* is a group of related doctrines that view human knowledge in some field of endeavor as purely subjective, and that no objective truth about the field exists. The result is that an individual's subjective experience is valued over empirical evidence, possibly including that which contradicts an individual's beliefs. For example, an artistic subjectivist would view that there is no objective way to determine whether a piece of art is "good", and that the quality of art is only intrinsically determined by the viewer. This application of subjectivism is not controversial; in contrast, moral subjectivism has been the center of centuries of heated debate. Consequently, we cannot simply drop or accept subjectivism without some degree of consideration for the underlying field.

In the sciences, it is widely accepted that there exists some objective truth.[1] Specifically, there are laws of physics, and they can be accurately described by Newton's laws and the associated mathematical equations. This is the position taken in this project: that audio phenomena can be objectively measured[2]. A consequence of this viewpoint is the rejection of the existence of audio phenomena that cannot possibly be measured or quantified[3] Unfortunately, a large subset of people, particularly those who are the most

[1]At a bare minimum, there is something so common to all our experiences that it ought be treated as objective reality. For example, almost everyone knows that releasing an apple in the air will cause it to fall towards the ground. Therefore, the existence of gravity as a phenomenon is effectively an objective truth, although its mechanism is not understood by all people.

[2]Whether or not the quality of a sound *as music* can be objectively measured is separate from measuring physical audio phenomena, and irrelevant for the project.

[3]To be clear, whether or not an individual is personally competent enough to measure the phenonmenon is not what is being considered. It is whether a measurement instrument of suitable precision and a competent operator could verify that the phenomenon exists. Measurability is a necessary but insufficient quality for existence. For example, the 21 grams experiment "measured" the weight of the soul to be 21 grams, yet the sample size was too small to conclude anything about the value of the weight or the correlation between the weight differential and the existence of the soul.

vocal about the importance of “quality” sound systems, are *audio subjectivists.* Consequently, these people are inherently distrustful of scientifically minded people who might make measurements that invalidate their opinions.

The problem with audio subjectivism is that there actually are phenomena that are difficult (but not impossible) to measure with standard equipment. The human ear is a ridiculously sensitive instrument, and it can detect very small vibrations. The ear can detect very small nonlinearities in the frequency response of an otherwise linear audio system compared to other fields in signal processing. Musicians with their ear training make that instrument even more sensitive. They have probably been told several times that something they can clearly hear does not exist by someone who claims to be an authority figure or domain expert. How often is such an interaction followed by whipping out an SPL meter and checking that the sound exists? Rarely, if ever. The problem of debunking audio subjectivism is that the concerns of its adherents *are valid and likely based in reality.* Unfortunately, it might be a reality that is mathematically complicated or driven by a poorly understood mechanism.

The continued usage of analog music gear in place of digital implementations is one of the most important examples of audio subjectivism in the modern era. One of the reasons that analog gear has persisted amongst musicians is precisely because of the variability of analog parameters. Two nominally equivalent analog pedals might sound subtley different. Consequently, there is a sense of uniqueness and individuality that comes with owning an analog pedal. For vacuum-tube-based gear, this is especially true, as vacuum-tube gear will audibly dull over the life of the tube, which can be noticed over a period of a few weeks of heavy usage. Quantifying the variability of these parameters is a difficult and subtle topic that is not of interest to most people. Through a misuse of Occam’s razor, because the unmeasureable, intrinsic “analog sound” seems just as reasonable to a musician unaquainted with electronics, “analog sound” is a simpler explanation and therefore the one worth adopting. For musicians this is mostly harmless, but audio designers must understand what makes analog gear sound good in order to compete with it.

For musicians who consider the authenticity of their analog gear to be a part of their art, no bit depth or sample frequency will ever be high enough to make the switch to digital worthwhile, because their pedal is metaphysically unique. Because of this, many musicians reject digital gear and cannot be convinced otherwise.

For musicians who can at least tolerate a part of their signal chain being digital, Shannon’s sampling theorem provides most of the answer. For linear processing, sampling at 44.1kHz ensures that the entire audible range is captured with about 2kHz as a buffer. Because the $\operatorname{sinc}(\cdot)$ interpolation function in the sampling theorem violates causality, some approximation, in my pedal a low-pass filter, is used to remove high-frequencies copies, or “aliases”, of the output signal after the DAC. If the filter is set to to be flat for all audible signals (which it is), then the process of ADC or DAC should not color the signal. For nonlinear processing, digital oversampling can be (and usually is) used. This is usually baked into the particular audio plugin that needs it. For practical purposes, anything that can be done in the analog domain can be done in the digital domain without creating any artefacts, so long as the plugin designer is competent. (Really, we can achieve an arbitrarily good approximation, so good that the artefacts are inaudible.) This does not account for the idiosyncratic behavior of analog devices. These need to be added to digital audio systems manually if they are desired.

The issues arise for plugins that attempt to model specific pieces of analog gear. The problem is to optimally discretize a nonlinear circuit model. Although the problem is theoretically simple, the actual implementation requires the plugin designer to choose function approximations that yield good results around the operating point in the fastest possible time. Although the feasibility of analog modeling plugins is considered a solved problem, there is still a lot of research to be done in testing fast nonlinear approximations.

However, analog emulating plugin technology has come a long way. Even ten years ago, emulations of the major distortion pedals, amplifiers, and matched speaker cabinets were indistinguishable from their analog versions in the mix. This is still very much the case for the most popular plugins.

This is not a hypothetical scenario. All sorts of wild pseudoscience has cropped up around audio system design, particularly in the "audiophile" community where things like "passive preamps" are regularly sold to people supersitious about the intrinsic worth of analog electronics. The experiences of artists and consumers of art are based upon their subjective evaluations of the media they consume. As I have demonstrated above, it is not much of a stretch to think that audio systems design is also a subjective field, since it is adjacent to music.

Of course, as I hope to demonstrate, audio system design is mostly an objective field driven by data and physical princples. Objectively good electronic system design, combined with subjective decisions about the user interface and intentional "imperfections" in the signal chain, are the ingredients that yield a quality sound system.

In summation, for those who absolutely cannot tolerate a single piece of digital gear in their signal chain:

> "You can lead a horse to water, but you can't make it drink."

I hope to establish their trust in the future by releasing the code for this project and as much of my audio work as possible as free and open-source software, to show them that there's nothing to hide. For everyone else, I hope my pedal serves as a useful tool in the future.

A.2.1 TL;DR

Shannon's Sampling Theorem ensures that any audible signals can be represented as a digital stream and reconstructed without losing a significant amount of information. Although the system should sound nearly transparent to a trained musician, there is nothing that can be done to convince analog purists to use my pedal, as their analog pedals are a part of their art.

Appendix B

Introduction to Audio Plugins

B.1 The purpose of the SimpleEQ plugin.

The SimpleEQ plugin was initially written as a way to pass the time while waiting for the ball to drop on New Years' Eve. The reason why I have brought it into the project is because it is a specific example of an audio plugin whose processing I fully and unambiguously understand, because I wrote it. This plugin really is the bare minimum that the system should be able to handle. It will serve as the first plugin to be tested, and it (or a version of it) will be included in the final version. Because I have all the project files, and because I wrote it with the JUCE framework, it is a trivial[1] matter to cross-compile the project for any target operating system and architecture. It's GitHub repo can be found here.[2]

B.1.1 Principle of operation

Audio is processed in blocks that are determined by the plugin host. This sample block size can usually be set by the user, and it is made as small as the CPU can handle before the block size exceeds the amount of time it takes to process the block.

The DSP algorithm is a cascade of generic 2nd-order IIR Butterworth filters and a resonant peak. The Butterworth filters are "low cut" and "high cut" (high-pass and low-pass respectively[3]). The plugin generates filter coefficients based on the frequency and steepness settings stored in the plugin's state, and it transforms one set of coefficients to the corresponding low-cut coefficients. The resonant peak is generated based on the frequency, gain, and Q parameters stored in the plugin's state. The plugin convolves an audio buffer with each filter response in cascade.

While this occurs, the GUI operates on a separate thread. The GUI looks for changes to its parameters, then sends any changes to the plugin's state. The new state will take effect when the next audio buffer is processed. The plugin host can also change parameters through *automation* if the user so chooses. For example, the gain of the peak filter can be set to respond to the amplitude of the audio signal.

The program structure, where audio processing and user control are handled by different threads and communicate through an external plugin state, is common to almost all plugins in all formats. Any plugin written with the JUCE framework begins with an automatically generate template of this structure. Consequently, these are the functions a plugin host, and consequently my hardware platform, needs to be capable of supporting.

[1]Almost trivial. There is a different fork of JUCE for Raspberry Pi. However, as explained in Appendix C.4, the system can support x86 and x64 GNU/Linux compiled plugins, so it will be unnecessary to compile for ARM64. Thus, the normal version of JUCE could be used.

[2]In case the URL doesn't work: https://github.com/gg232/SimpleEq

[3]In audio, the "cut" terminology is typically preferred because that is why an audio mixing engineer typically applies a high/low-pass filter: to remove or "cut out" low frequencies. In this section, the "cut" terminology will be used, but in the rest of the UE-DEP project, the "pass" terminology is used.

B.2 What is an audio plugin?

An audio plugin is a piece of software designed to extend the signal processing functionality of a DAW (Digital Audio Workstation - music producing software). For example, SimpleEQ is (as of this submission) a linear equalizer plugin. However, nonlinear effects like distortion, dynamic compression, modulation, and real-time convolution are possible and commonplace.

The rest of this section is low-level details, and can be skipped on first reading.

A plugin host is any program capable of running at least one format of standard audio plugins. For the purpose of this project, an "audio plugin host" will refer to a DAW that is not necessarily designed to permanently record input or output to a file. The primary plugin host will be Carla.

The motivation behind plugin standards is that in C++, a programmer cannot simply move classes[4] out of a file and expect it to compile. For this reason, there needs to be a software interface between the host and the code contained in the plugin to allow the pieces of software to communicate. The various plugin standards have each found their own unique ways around that problem. It is up to the developers of audio hosts and DAWs to ensure that their hosts can load whatever plugin formats they claim to support. However, most hosts, especially those geared towards budget-conscious musicians, support VST at least.

B.2.1 What exactly is a *VST* Plugin?

VST is a specific group of plugin format standards defined by Steinberg. Practically, it establishes a program architecture where the audio processor and control structure are separate. It also establishes a uniform system for the plugin and host to communicate. Crucially, it enforces Microsoft's Component Object Model, a set of restrictions on C++ classes that allows low-level code compiled with different compilers to communicate.

There are three generations of VST: the original VST, VST2, and VST3. Plugins are rarely released in VST2 anymore. For practical purposes, supporting VST3 is equivalent to supporting all VSTs. There is also a distinction between plugins that operate on audio signals and those that operate on MIDI signals. Because these are usually used as instruments controlled by an external keyboard or MIDI source, they are called VST instruments, or VSTi. So long as the host can accept MIDI input, supporting VST3 is equivalent to supporting VSTi's within the VST3 standard.

For a more in-depth look at VST, and the differences between VST2 and VST3, see [4].

B.3 Conclusion

According to Wikipedia:

"A simulation is the imitation of the operation of a real-world process or system over time"

In this sense, an audio plugin is not *intrinsically* a simulation. This fact is the principle that drives this project. For example, the SimpleEQ plugin is not a simulation of an equalizer; it *is* an equalizer. Running a piece of audio through this pedal will affect the frequency response according to the indicated filter characteristic. The only practical difference between using the plugin with ADC and DAC and using an (ideal) analog equalizer with the same transfer function is that the digital one introduces a time delay of 2^N samples.[5] The memory locations the plugin host assigns *are* where the process of mixing music and modifying audio data actually occurs.

In discrete event dynamic systems, a system is said to simulate another system if all the equivalent transitions of the simulated system can happen in the simulation subject to any sequence of events that can occur in the simulated system, up to an equivalence relation. A relation is inherently binary. If the simulated system never existed, then the "simulation" system is not a simulation.

[4]Classes and other data structures are absolutely necessary to program a serious audio plugin. C would be unable to handle object like GUI sliders or processor classes, so at least C++ is necessary.

[5]2^N represents the size of the audio buffer in samples. Practically, the audio buffer is chosen to be as small as the CPU can handle. It is set by the plugin host. Other plugins can introduce their own latency depending on their programming, but the latency caused by the audio buffer is only applied once. Typical values are between 16 and 1024 samples. For a 48kHz sample rate, it is not advised to use a buffer larger than 48 samples $\left(= 48\text{k}\frac{\text{samples}}{\text{s}} \times 1\text{ms}\right)$if the CPU allows, although guitarists can typically put up with slightly higher values than musicians who play other instruments.

That being said, *many* audio plugins are (informally) simulations of some piece of analog gear. For example, the TSE808 plugin is a model of a specific Tube Screamer pedal. Most of the plugins I use are models of specific pieces of analog gear. However, this is not the case for all plugins. As a counterexample, all the filters in SimpleEQ are discrete-time IIR versions of the 2nd order Butterworth transfer function. These filters could be realized in the analog domain by designing Butterworth filters using a filter coefficient table and cascading the indivdual filter sections, but *this plugin was not derived from an already existing analog circuit.* The goal was to implement the discrete time filter in the digital domain. Still, the frequency responses and nonlinear system functions of analog gear still provide inspiration for plugin algorithms because these are the sounds to which listeners have been acclimated.

Appendix C

Operating Systems and How They Affect This Design

This appendix is a vast oversimplification of how the existence of several operating systems influenced the design of the system. The facts are recounted in broad terms as they are relevant to the project.

The operating system is a piece of software that manages other applications and provides common services for them. Examples of operating systems include Windows 10, Mac OS X, Debian, and Raspberry Pi OS. A crucial component of any operating system is its *kernel*, which manages low-level tasks. For example, the common element amongst all GNU/Linux operating systems is their dependence on a version of the Linux kernel. For this reason, different Linux-based operating systems are often called distributions. For example, Raspberry Pi OS is a modified Debian distribution.

C.1 The relation between Operating Systems and Processor Architectures

There are many processor architectures, but the ones that are important for this project are x86, x64, and ARM64. x64 is the "typical" architecture for 64-bit desktop computers using Intel or AMD CPUs. x86 is the older 32-bit version of the Intel/AMD architecture. ARM64 is the 64-bit version of the ARM architecture typical of mobile and embedded devices.

GNU/Linux is the term for the group of FOSS operating systems that use the Linux kernel. For the purpose of audio work, the source code of a "Linux" plugin will compile on any GNU/Linux OS so long as all its dependencies are installed. However, depending on the complexity of the plugin, pre-compiled binaries might not work for all Linux systems. In that case, developers will typically release a bunch of binaries compiled for the "mainstream" Linux-based operating systems. People using specialized GNU/Linux distributions can often get away with using the mainstream binaries if their distribution was forked from one of the main ones. (For example, Debian binaries will usually run on RPi OS because RPi OS was forked from Debian). It is up to each developer of a GNU/Linux operating system to release separate binaries of their operating system for each processor architecture they wish to support. In the case of RPi OS, the OS is available for x86, x64, and ARM64 architectures, as well as 32-bit ARM.

Windows is a proprietary, closed source operating system. Most versions of Windows can be built for x86 and x64 architectures. Special versions of Windows exist for ARM architectures, but these are never used for audio work. For the purposes of this project, Windows software can be built for either the x86 or x64 architectures.

For the highest-end studios, Mac OS X is the operating system of choice because of its integration with ProTools and other Avid products, and the status that can be broadcasted by owning an expensive Apple computer. However, for even remotely budget-conscious studios that typically cannot afford Avid and Apple products, Windows is the operating system of choice for audio production. (Anecdotally, it is currently my choice for audio production.) Although it is the absolute worst OS for low-latency audio work out of the box, most manufacturers supply Windows drivers to circumvent the OS's native audio manager. Additionally, most audio programs (including ProTools) support Windows systems. Most people use Windows on their office and gaming computers anyway, so the transition to their audio PC is more seamless.

C.2 Raspberry Pi 4 is ARM64-based

The Raspberry Pi 4 has a CPU with an *ARM64* processor architecture.

The importance of this fact cannot be understated. The reason why the RPi 4 was chosen was because it has excellent documentation, it is accessible for makers and musicians, and because it has 8GB of RAM and a fast processor. I could not find a comparable x86 microcomputer with a similar amount of RAM or comparable processor. In order to support the vast majority of audio plugins, there needs to be *software* support for x86 and x64[1] system calls.

When I graduated from recording school (2016), professional audio on Linux was simply impractical. Our curriculum included no mention of Linux. The main issue with audio on Linux at that time was a lack of audio programs comparable to those available for Windows and OS X. Even back then, the problem of latency was mostly solved at the system level with the integration of ALSA into Linux and the development of JACK for low-latency audio routing. WINE did exist and had decent support for modern video games and office applications, but the technology was not quite ready for pro audio.

Windows emulation has come a very long way since then. As if to demonstrate to the world how far such emulation had come, in 2020 the Valve Corporation released a game console based around a heavily modified distribution of Arch Linux. This is aimed at mainstream consumers looking to play games *developed for PC* on a Windows system. So far, the system has been quite successful. It demonstrates that real-time operations comparable to those expected of a powerful desktop PC are possible on Linux.

For this project, the important development is the release of the Carla plugin host. Carla can be built with bridges to WINE and x86/64 emulators to allow the host to use plugins compiled for Windows.

C.3 How Computer Programs Work

The workflow to implement a computer program is as follows:

1. Write the code in a programming language as a text file.
2. Compile the source code into assembly language.
3. Link together assembly code and libraries into an executable.

The assembly language in Step 2 depends on the processor's instruction set architecture. Step 1 could be dependent on the operating system if the programmer uses APIs that are unique to a platform. Steps 2 and 3 are usually abstracted away by the IDE, which is preconfigured by default to compile for the host operating system.

An audio plugin is a compiled binary file with functions that are called by the audio host as they are needed. Because the binary file is compiled for a target platform, all plugin binaries are at least restricted to run on the processor architecture for which the programmer chose to compile it. Usually, x86 and x64 versions of plugins are offered as precompiled binaries. Because the plugin contains GUI data, the compliled plugins usually rely on operating-system specific API calls unless a unifying framework like JUCE is used. (Even when using JUCE, the plugin needs to specifically compiled for each target platform, although there usually are no meaningful changes to the source code. The change is in the linked libraries provided by JUCE.) JUCE is increasingly adopted by audio developers.

[1]64-bit support is prioritized over 32-bit support because 32-bit operating systems are going the way of the dinosaur. Many plugin developers have stopped releasing 32-bit binaries for their plugins altogether.

C.4 The Approach for Supporting Audio Plugins

The support of audio plugins is the same as support for software in general, so I will discuss the topic in general terms.

Consequently, there are eight possible classes of software to support:

1. Compiled for ARM
   (a) 64-bit
      i. Windows (unheard of)
      ii. GNU Linux (Raspberry Pi)
   (b) 32-bit
      i. Windows (unheard of)
      ii. GNU Linux (Raspberry Pi w/ 32-bit OS)
2. Compiled for Intel/AMD
   (a) 64-bit
      i. Windows (most common)
      ii. GNU Linux (rare but increasing)
   (b) 32-bit
      i. Windows (common in the past)
      ii. GNU Linux (rare, unlikely for new plugins)

The easiest class to support is ARM64-GNU Linux. This includes plugins compiled for Raspberry Pi directly. 32-bit ARM binaries are supported on 64-bit Raspberry Pi OS. Regardless, I am not aware of any plugin manufacturer who actually releases ARM binaries.

Next, we have the two classes of 32- and 64-bit "Intel" architectures with a GNU Linux target OS. These are currently supported. This is accomplished by the box86 and box64 libraries that translate x86 and x64 system calls (respectively) into ARM. Carla is aware of these libraries, and consequently can load plugins compiled for GNU Linux in any architecture.

Support for 32-bit Windows plugins has been suboptimally implemented by installing a 32-bit Windows plugin host with ASIO drivers that runs in WINE. However, my goal is to compile Carla with the Windows plugin bridges. These would fully support x86 and x64 Windows plugins. Contingency plans for the various types of plugins are detailed in the Plan section.

At a bare minimum, the pedal can currently run GNU Linux plugins compiled for any "mainstream" target architecture. This opens the door for a variety of FOSS plugins. Paid and proprietary plugins are typically released for Windows.

The focus will be on implementing VST-formatted plugins. This is the most common format, and it is the format-du-jeur of consumer-grade (and consumer-priced) audio plugins. If Carla is used, it will be easy to support the FOSS plugin formats for GNU Linux.

I have no intent to support AAX plugins because developers need to pay Avid for the rights to develop software in their format. Every other plugin format is free to develop. Frankly, I have no use for AAX-only plugins as a musician or audio developer, and I advise that audio engineers avoid AAX-only plugins for any purpose.

Appendix D

Git, GitHub, Version Control, and their Role in Maintaining This Project

Very briefly, my project will be hosted on GitHub. Project progress can be monitored by checking through the edit history of the project's GitHub repository.

This Appendix is included so I can discuss Git concepts clearly with my adviser or anyone else interested in my project.

D.1 An explanation of Git

Version control software is a tool that tracks the changes to a group of files. For example, if a programmer adds code to a project that breaks the project, then saves the project and publishes it to the internet, then the project will be broken. To fix the error, the programmer has to look through the source code of the project and remove the error directly. For complex projects, it might not be obvious which code was broken. If several programmers are involved in the project, it will be difficult to point out how one programmer changed the project.

Git is the most popular version control software. It is available for practically any modern operating system. Git works by tracking the changes to files and subdirectories within a directory[1] by using a *tree*[2] representation. Because source code files are typically plain text files with unique file extensions, Git can track the changes in source code files and other compressed text files. It cannot track changes in binaries such as executables, .docx files, and .png files, but it can tell if changes have been made, and it can revert those files back to their previously saved versions. Source files (or anything a programmer wants Git to track) are typically saved to online *repositories* (or *repo* for short).

To compile a program from source, the first step is typically to clone the Git repository, or copy the files and subdirectory[3] structure to a local directory of choice.

GitHub is Microsoft's online repository host. Whenever the source code of a project is released online, the code actually resides on the servers owned by the repository host (in this project's case, GitHub). GitHub is a searchable website where most Git features are available for viewing.

To edit a Git repository hosted on GitHub, a programmer fetches the repo from GitHub, then works on the contents of the repo on their local machine. Once the programmer is ready to commit a change to the online repository, it is pushed to GitHub using a unique security code so that only people who are allowed to change the repo can do so.

Because Git stores a tree structure, a programmer can create a branch within the original project and work on it independently from the main branch. This allows a programmer to make drastic, possibly

[1]On Windows systems, a directory is usually called a *folder*, and a subdirectory is called a *subfolder*.

[2]In graph theory, a tree is a minimal set of branches such that, by traversing only the branches in the tree, all nodes in the graph can be reached. Basically, Git implements this graph, and allows the user to traverse the graph by either committing data or reverting previous commits (undo).

[3]A subdirectory (relative to a parent directory) is a directory contained within another directory.

program-breaking changes in their branch without affecting the main branch. Practically, this is leveraged by programmers who designate a "main" branch as the most important or "outward facing" branch that people can use, and a developement branch (or several) where changes can be implemented. Once a development branch is debugged to work, it can be *merged* back into the main branch through Git's merge command. This allows the programmer to see the changes and where they occur, and to approve or reject individual changes.

D.2 What files can be managed by Git?

For any text file with any file extension (or none), the creation of, changes within, and deletion of files can be fully tracked. This is why the original versions of all documents are written in LyX[4]: .lyx files are uncompressed text files, so all changes can be tracked with Git. These files can be easily converted to .tex, .docx, or .odt.

For any other file (e.g. a compressed binary, or just "binary" for short), Git can track if the file was changed, and what was its binary content for the last commit. Practically, this means that versions of binaries can be stored, but that it cannot tell the programmer exactly which lines were changed. Individual changes can not be approved or rejected, but the programmer can still choose to accept the changed binary or revert to the old version.

Explicitly, the following file types are compiled binaries:

- Multisim files
- Word documents (.docx)
- Open Document Type documents (.odt)
- Executables
- Dynamic linked libraries (.dll's)

The following file types are plain text files who changes can be fully tracked:

- Ordinary text files (.txt)
- GitHub markdown files (.md)
- LyX documents (.lyx)
- TeX and LaTeX documents (.tex)

D.2.1 What files can be hosted on GitHub?

GitHub can host any file that Git can track with one crucial exception: files must be *under 100MB* for users who have not paid for large file storage. Basically, hosted files should be small. There is no stated size limitation for entire repos, but GitHub support suggests that they stay under 1GB total size.

D.3 Conclusion

I will be using GitHub to disclose my progress, release the project files and documentation to the public, and *maintain a history of my changes.*

The typical workflow for the project will be to fetch the copy onto whatever computer I happen to be using, then create a local WIP (work in progress) branch. All new work will take place on this branch. If I need to interrupt my work or switch computers, the local WIP branch will be pushed to the GitHub repo. When the WIP branch has reached a point that I am comfortable committing the changes permanently, the WIP branch will be merged into the main branch.

[4]LyX is a "manuscript editor" that generates LaTeX code in a "friendlier" environment resembling a standard word processor.

The GitHub repo for this file is located at the following hyperlink[5]. All code, documentation, and assets that I can host will be posted there.

[5]In case the hyperlink is broken: https://github.com/gg232/UE-DEP

Bibliography

[1] “Free analog filter program,” https://www.kecktaylor.com/, accessed: 2023-01-22.

[2] “Ti pcm1808 datasheet,” https://www.ti.com/lit/ds/symlink/pcm1808.pdf.

[3] “Mcp3008 adc datasheet,” https://www.ti.com/lit/ds/symlink/pcm1808.pdf.

[4] “Fabian renn giles - under the hood of vst2, vst3, au, auv3 and aax,” https://youtu.be/swVqdbhfkkE?t=2227.