INSTITUTO TECNOLÓGICO DE AERONÁUTICA

Eduardo Henrique Ferreira Silva

DIGITAL PROCESSING OF AUDIO SIGNAL WITH RASPBERRY

Trabalho de Graduação
2019

Curso de Engenharia de Computação

Eduardo Henrique Ferreira Silva

DIGITAL PROCESSING OF AUDIO SIGNAL WITH RASPBERRY

Orientadores

Prof. Dr. Manish Sharma (ITA)

ENGENHARIA DE COMPUTAÇÃO

São José dos Campos
Instituto Tecnológico de Aeronáutica

2019

Dados Internacionais de Catalogação-na-Publicação (CIP)
Divisão de Informação e Documentação

Ferreira Silva, Eduardo Henrique Ferreira Silva
Digital Processing of audio signal with Raspberrry / Eduardo Henrique Ferreira Silva
São José dos Campos, 2019.
48f

Trabalho de Graduação – Engenharia de Computação – Instituto Tecnológico de Aeronáutica, 2019. Orientador: Dr. Manish Sharma

1. Digital Signal Processing. 2. Audio Processing. 3. Raspberry. I. Instituto Tecnológico de Aeronáutica. II. Digital processing of audio signal with Raspberry

REFERÊNCIA BIBLIOGRÁFICA

FERREIRA SILVA, Eduardo Henrique. **Digital processing of audio signal with Raspberry**. 2019. 48f. Trabalho de Conclusão de Curso. (Graduação em Engenharia de Computação) – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSÃO DE DIREITOS

Eduardo Henrique Ferreira Silva
Rua H8A, 117, Campus do DCTA
12228-460, São José dos Campos - SP

DIGITAL PROCESSING OF AUDIO SIGNAL WITH RASPBERRY

Essa publicação foi aceita como Relatório Final de Trabalho de Graduação

Eduardo Henrique Ferreira Silva
Autor

Prof. Dr. Manish Sharma (ITA)
Orientador

Prof. Dr. Inaldo Capistrano Costa
Coordenador do Curso de Engenharia de Computação

São José dos Campos, 20 de novembro de 2019

Dedico esse trabalho aos meus pais, que abriram mão de muito para me dar o que nunca tiveram, a Guilherme, que cresceu tanto mesmo eu estando longe, a Diogo, que se foi cedo demais, e à amiga, hoje distante, que me aturou nas fases mais dificeis da jornada

Agradecimentos

A execução desse trabalho não seria possível sem a incrível colaboração dos professores, colegas e amigos que fizeram parte dessa jornada. Acima de tudo, agradeço aos meus pais e irmão, pelo suporte em cada segundo dos últimos 7 anos.

"I’m just a musical prostitute, my dear”.

(Freddie Mercury)

Resumo

O processamento digital de sinais de áudio encontra aplicações em diversas áreas como telecomunicações, acústica e produção musical. Em particular, equipamentos relacionados a aplicações de alta fidelidade no meio artístico possuem alto valor agregado e são, via de regra, de código fechado. Isso dificulta o acesso a essas tecnologias e impões barreiras a seu desenvolvimento. O presente trabalho apresenta o desenvolvimento de uma plataforma portátil e opensource com foco em processamento de alta qualidade em tempo real, baseada em um computador single-board Raspberry. Os resultados desse desenvolvimento serão avaliados através da aplicação de efeitos como eco, reverberação, distorção e modulações a sinais de instrumentos elétricos de corda. Os usos da plataforma não ficam, entretanto, restritos ao meio artístico, podendo ser utilizada também em campos como Processamento de Linguagem Natural (PLN).

Abstract

Digital signal processing has applications in several areas, such as telecommunication, acoustics and musical production. In particular, equipment related to artistic applications have high added value and are, as a rule, closed source code. This hinders access to these technologies and imposes barriers to their development. The present dissertation presents the development of a portable, open-source platform that focuses on high definition, real-time processing, based on a Raspberry single-board computer. The results will be evaluated by the application of effects such as delay, reverb, distortion, and modulation to electric stringed instruments. The uses of this platform, however, are not restricted to the artistic field, being utilizable in areas like Natural Language Processing (NLP)

Table of Figures

Table of Abbreviations and Acronyms

ADC	Analog-Digital Conversion
ALSA	Advanced Linux Sound Architecture
DAC	Digital-Analog Conversion

Table of Symbols

n_{buffer}	ADC and DAC buffers size, measured in frames
n_{frame}	ADC and DAC frame resolution, measured in bits

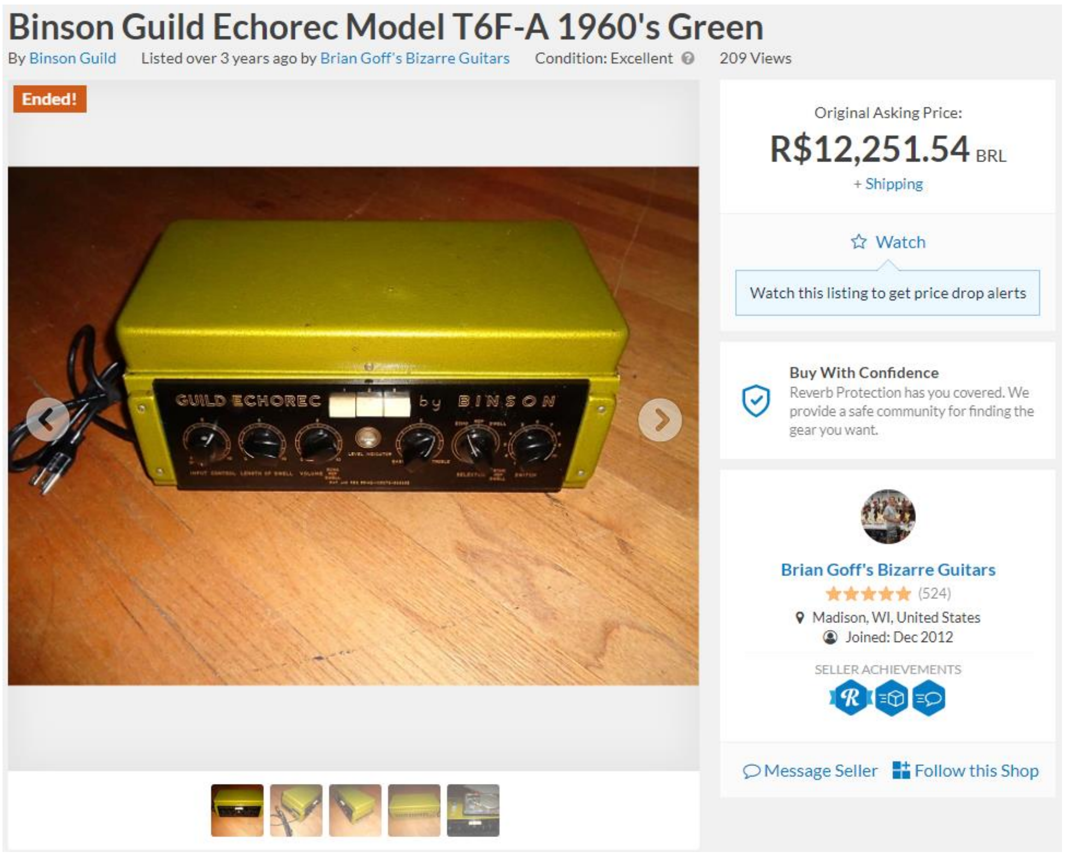
Table of Contents

1 Introduction

Audio processing is intrinsically related to every modern audio consumption. In the musical field, for example, virtually every sound heard in a record was processed in some way: voices are equalized to sound more exciting, effects like distortion and delay are added to guitars and the whole song is made louder using compression and limiting.

By the 60’s, all the above technics were well developed to audio application, mainly using technologies such as vacuum tubes and magnetic tapes. The analog processors that were originally used to produce these effects are nowadays considered *vintage* and highly sought after.

For example, signal processors that added echo to the sound, like the 60’s *Binson Echorec*, are now sold used for more than US$12.250 [1].


Binson Guild Echorec Model T6F-A 1960's Green

By Binson Guild Listed over 3 years ago by Brian Goff's Bizarre Guitars Condition: Excellent 209 Views

Original Asking Price: R$12,251.54 BRL + Shipping

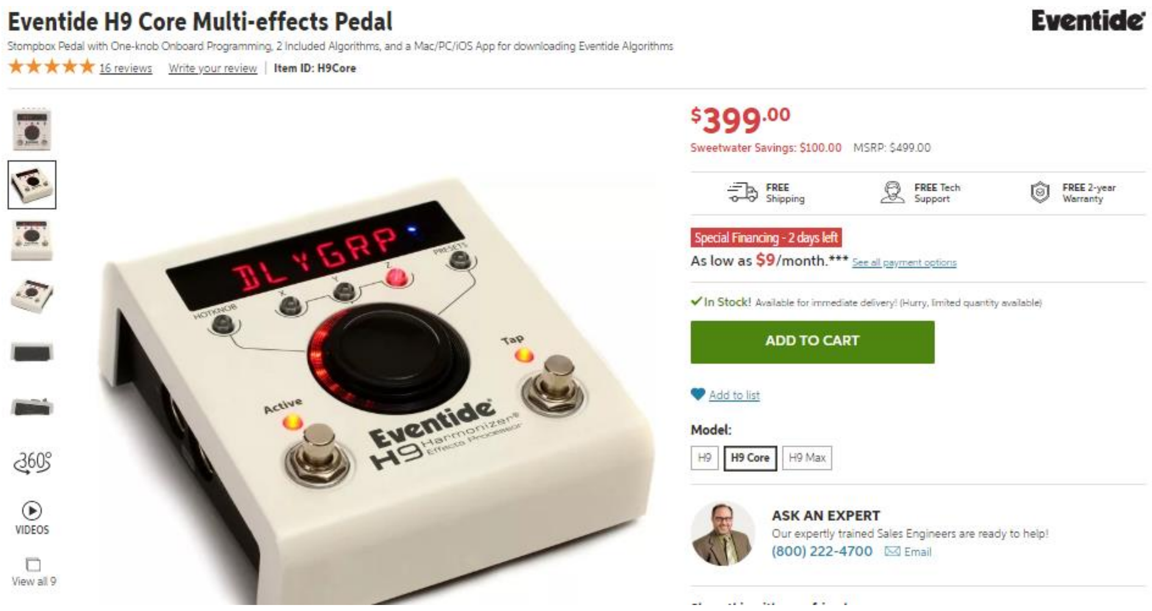
Watch this listing to get price drop alerts

Buy With Confidence
Reverb Protection has you covered. We provide a safe community for finding the gear you want.

Brian Goff's Bizarre Guitars (524)
Madison, WI, United States
Joined: Dec 2012
SELLER ACHIEVEMENTS

FIGURE 1 - Binson Echorec vintage audio processor

Motivated by the high demand for vintage audio processors, a whole industry developed around simulating those pieces of equipment digitally. Merging the analog sound characteristics and the practicality of digital, brands like *Eventide* started developing high-end digital processors, such as the *Eventide H9* [2], that promise to emulate equipment like the *Binson Echorec* and others for a fraction of their prices.



Eventide H9 Core Multi-effects Pedal

Stompbox Pedal with One-knob Onboard Programming, 2 Included Algorithms, and a Mac/PC/iOS App for downloading Eventide Algorithms

★★★★★ 16 reviews Write your review | Item ID: H9Core

$399.00

Sweetwater Savings: $100.00 MSRP: $499.00

FREE Shipping FREE Tech Support FREE 2-year Warranty

Special Financing - 2 days left

As low as $9/month.*** See all payment options

✓ In Stock! Available for immediate delivery! (Hurry, limited quantity available)

ADD TO CART

Add to list

Model: H9 H9 Core H9 Max

ASK AN EXPERT

Our expertly trained Sales Engineers are ready to help!

(800) 222-4700 Email

FIGURE 2 - Eventide H9 audio effects processor

1.1 Motivation

Although alternatives like the Eventide H9 are much more affordable than the original processors, they are still economically inviable for the clear majority of consumers. In the Brazilian market, for example, its prices go up to R$3.000 [3].

Another essential problem with these modern commercial signal processors is that they are all closed-source. Once you buy the hardware, you'll not be able to reprogram it, even though they can do it, since companies sell extra algorithms, with different sounds or effects.

Open-source audio signal processor options do exist. One example is the Electrosmash Pedal PI, a Raspberry Pi Zero based audio signal processor. The major problem with this platform is its sound quality is a limiting factor for professional uses: it has a resolution of 12 bits, while modern commercial application uses, at least, 24 bits, per audio sample, per channel.

On the other side of the spectrum, there are options like the MOD Duo, that has open-source code and high-fidelity sound specifications. On the other hand, its price can be as high as that of closed-source options, making it inviable as well.

1.2 Objective

In this context, the objective of this work is to create a low cost, portable digital audio processor, focusing on high definition sound quality, by using the single-board computer Raspberry Pi 3.

To make this task feasible and testable, the main focus will be to process signals form electric stringed instruments such as electric guitar and bass guitar, and add well known and used effects, such as delay, reverb, distortion, and modulations.

By making the source code and the hardware project open, the aim is to facilitate the development of applications that demand professional audio definition, whether in the scientific or artistic fields.

1.3 Outline

The present work is structured in the following way:

- Chapter 1: introduces the context, the objective and the approach of the work;
- Chapter 2: describe the main tools and equipment used in the development;
- Chapter 3: describe the construction and implementation of the processor;
- Chapter 4: discusses the obtained results;
- Chapter 5: concludes the work and presents further continuation proposals;

2 Development Tools

In this chapter, the main tools and equipment used in the development of the platform will be discussed.

2.1 Hardware

2.1.1 Raspberry Pi 3

The first component chosen was the Raspberry Pi 3 [4], which will be the core of the audio processor. It is a single-board computer, so the final audio processor will be self-contained, enabling the user to program it without the use of an additional computer.

Some other specifications guided that choice:

- A Quad-Core 1.2GHz CPU should provide enough processing to power the platform, even allowing multiple algorithms running at once;
- The ready-to-use HDMI and USB ports provide an easy interface to monitor, keyboard and mouse, making its programming easier;
- The official supported operational system, Raspbian, is a Debian variation and, as so, is light, fully open source and widespread;
- The Raspberry Pi Foundation assures production and support for this model at least until 2022.

FIGURE 3 - Raspberry Pi 3 Single-board computer

2.1.2 Audio Injector Raspberry Pi Sound Card

Since electric string instruments produce a low amplitude analog signal, two steps are necessary before manipulating it digitally: preamplification and analog-digital conversion (ADC).

Once manipulated by the Raspberry Pi 3, the digital signal needs to be converted again to an analog signal (DAC)

To handle these steps, the solution that better fitted the project's time and cost criteria was to use a pre-assembled interface, such as the *Audio Injector Raspberry Pi Sound Card* [5].

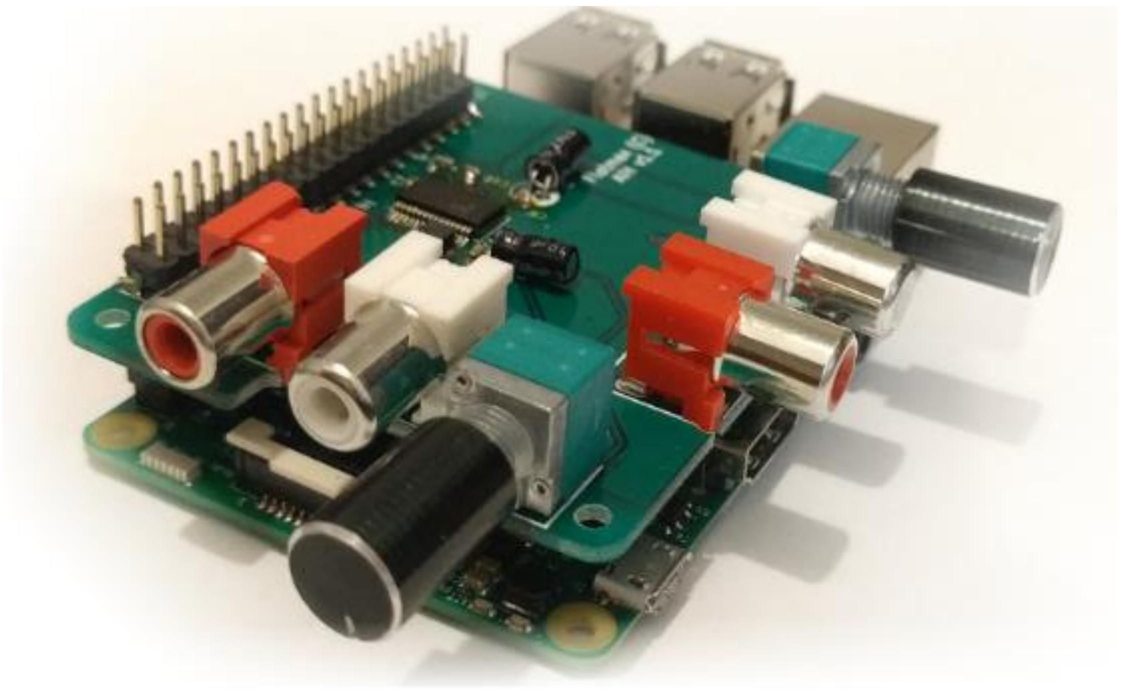


FIGURE 4 - Audio Injector Raspberry Pi Sound Card

The following arguments contributed to the choice for this interface:

- It has a 96 kHz maximum sample rate and 32-bit resolution, that is more than enough for professional audio applications;
- It has built-in input gain control, that allows the ADCs full converting range to be used;
- Including conversion and taxes, the final importing price of this component should be less than R$150, that is not an unfeasible price.

- It is driven by ALSA, Advanced Linux Sound Architecture (discussed later), so it has smooth integration with the Raspberry Pi 3 Operational System.

2.2 Software

2.2.1 Advanced Linux Sound Architecture (ALSA)

The Advanced Linux Sound Architecture [6] is a software framework that provides audio functionalities to Linux based Kernels.

The Audio Injector interface already has implemented drivers to proceed with the communication with Raspberry, and these drivers are implemented using ALSA, so the integration between the interface and the single boarded computer will be facilitated.

Another fundamental ALSA component is alsa-lib, a user-space library, programmed in C, and that will simplify the algorithm's implementation since it provides user-ready high-level functionalities such as ring buffers and mixers.

2.2.2 gtkIOStream

gtkIOStrem [7] is a C++ library that facilitates the usage of ALSA by interfacing it with a handful of other useful software tools.

For example, it implements audio clients for playing, recording and processing of digital audio, interfaces with matrix computation libraries and implements some neural network functionalities.

The main usage of the gtkIOStream library in the present work will be to facilitate the usage of ALSA and alsa-lib, since both of these software is very low level, but since it will be already fully installed in the end product it will be a powerful tool to future improvements.

2.2.3 jack

jack [8] is a cross-platform API, compatible with C++, that enables real-time digital audio between program and even platforms. It’s one of the main libraries interfaced by *gtkIOStream*, and with both together it’s possible to process the audio signal in a very efficient, intuitive and accessible manner.

Some very popular audio applications, such as Audacity, a free cross-platform multitrack audio editor, and Blender, an open-source 3D content creation suite, use *jack* in their source code.

The usage of both *gtkIOStream* and *jack* is fundamental to allow to final project to be fast and intuitive, making it easier for future users to modify an improve the platform.

3 System design and implementation

3.1 Signal chain

With the components previously discussed, it is viable to construct the following signal chain:

1. The audio interface captures instrument signal, amplifies it and the analog-digital conversion proceeds;
2. The communication between the de audio interface and the Raspberry is carried by the ALSA Drivers, so the audio information is available in the user programming space.
3. An application powered by the gtkIOStream and jackd libraries will manipulate the sound digitally, adding the desired effects.
4. The processed digital audio signal is sent back to the Audio Injector interface, which proceeds with the digital-analog conversion.
5. The processed analog signal can be then amplified by an instrument amplifier and the effect can be heard.

The following figure illustrates the planned signal chain:

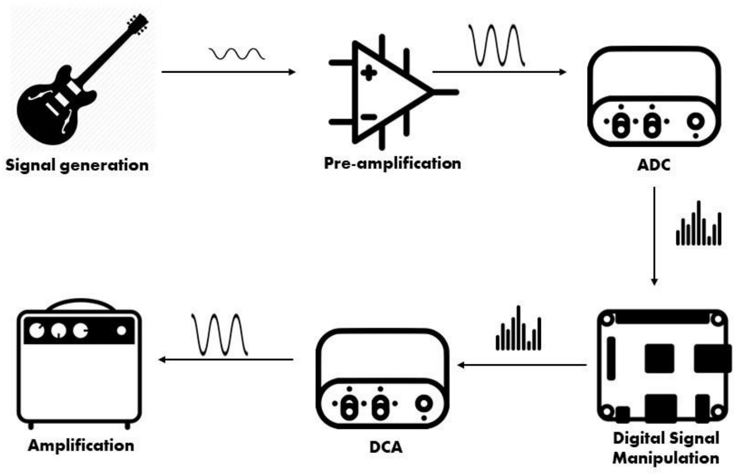


FIGURE 5 – Planned signal chain

3.2 Dependencies installation

To power the platform, two major dependencies are required: the Raspbian operational system and the gtkIOStream package. Both require installation and compilation. In the gtkIOStream package, the installation process is complex, but there is a tutorial in *Appendix B – Dependencies Installation*, all the process is detailed.

The full compilation of the package enables the platform to have a faster execution time since no more package compilation is required when the pedal is stated.

3.3 Source code

With the Raspibian operational system installed and all other dependencies compiled, it was possible to start developing the actual pedal's source code.

The two main concerns behind the development of the pedal's source code are that it needs to be fast and easy to use. This led to the decision of using *C++* as the programing language since it's fast, accessible and has seamless integration with all other components.

Another decision made to improve performance and ease of use was to write all the source code in one file, *Raspedal*.c, that is fully transcribed in *Appendix B – Source Code*.

This file can be divided into three parts:

- The global environment, where all effect functions and control variables are;
- The *JackFullDuplex clas*s, that contains the *processAudio* method, which calls all the active effects;
- The *main* function, that initializes every control variable and implements the user interface.

3.3.1 Global environment

The global environment is where all effect functions are defined. In addition, all variables utilized in the audio processing are defined in the global environment as well.

This allows a fast and organized code editing, simple understanding and prevents the use of complex object-oriented programing, that would be prohibitive to a lot of users.

On the other hand, defining all these functions and variables in the global scope requires a lot of attention is aspects such as variable naming and usage. Repeated variable names could cause bugs and malfunction, but weighing the pros and cons of this approach, it is still positive to the final product.

All effect functions will be further explained in the following topic.

3.3.2 The *JackFullDuplex* class

The *JackFullDuplex* is a class implemented by the gtkIOStream package that enables the user the program to make a full-duplex communication with the interface buffers. In this type of communication, both the receptor and the transmitter can send information to each other simultaneously. This communication works as follows:

- Every time the ADC buffer reaches its maximum capacity (n_{buffer} frames), the running programming is interrupted, and the *JackFullDuplex* class's method *processAudio* is called.
- The class receives an array *in[n_{buffer}]*, which consists of a series of float point values ranging from -1 to +1. These are the normalized values of each ADC frame measure.
- The *processAudio* method than iterates over the *in[n_{buffer}]* array, copies it to the *out[n_{buffer}]* array, and call all the active effects.
- The active effects modify the *out[n_{buffer}]* array.
- Once every value of the *out[n_{buffer}]* array is written, the class informs the DAC that it can perform its conversion.
- The DAC copies the *out[n_{buffer}]* to its own buffer and then generates the corresponding analog signal.

3.3.3 *main* function

Finally, the *main* functions have three responsibilities:

- First, it initializes all control and effect variables, attributing the correct initial value to them;
- Second, it initiates an instance of the *JackFullDuplex* class, create the necessary audio ports and start the audio processing client;
- Third, it prints and operates the user interface.

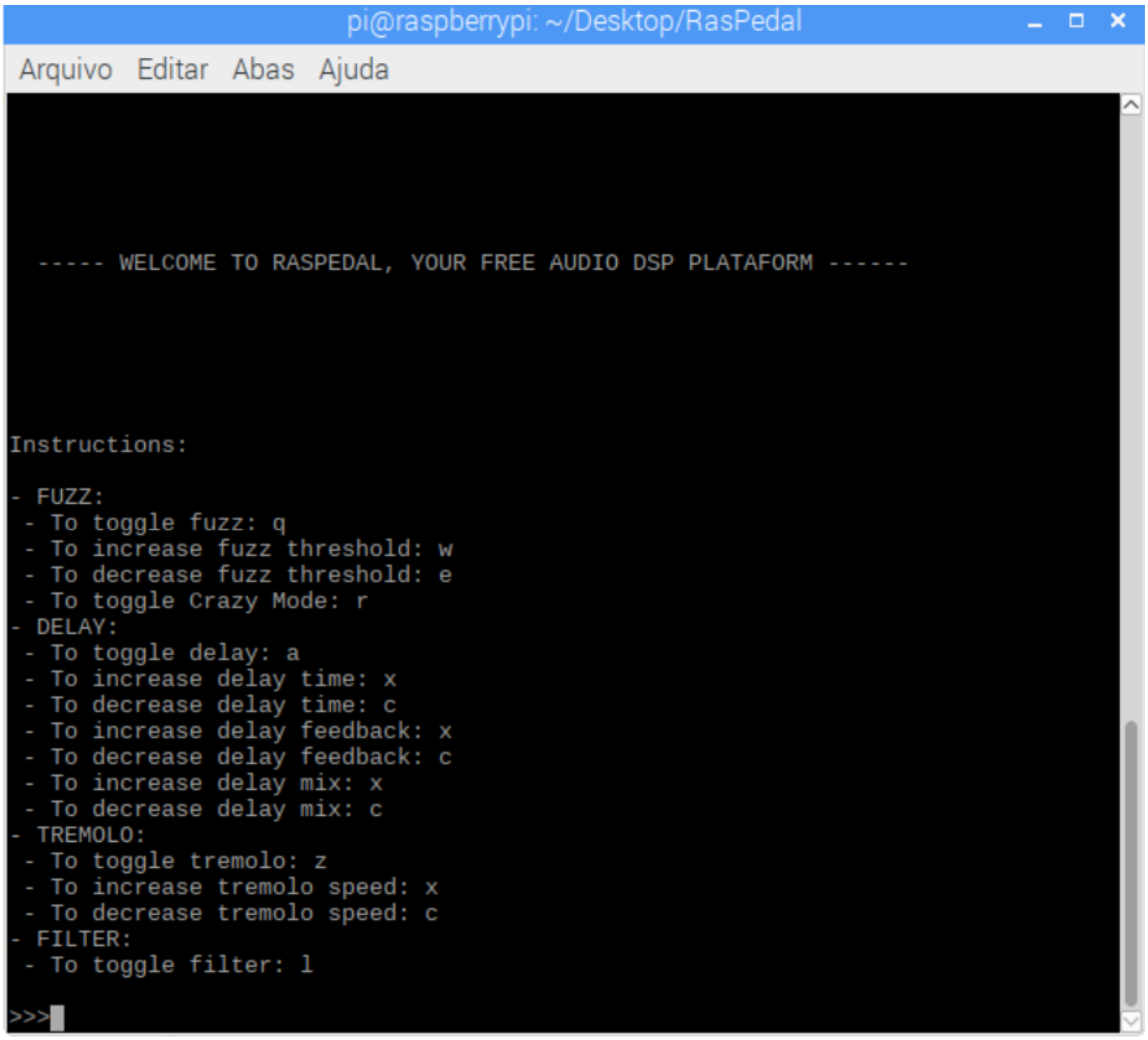
By these three steps, the main function enables the programmed effects to run correctly while the user makes changes to the audio parameters.

3.4 Interface

To enable the user to manipulate the sound parameters while playing, a user interface was made

It consists of a welcoming text, followed by the instruction. Each possible parameter changing is mapped to a specific key so that it's easy to manipulate the sound with only one hand. This is fundamental since it's expected that the user will be manipulating a musical instrument while using the platform.

The following figure shows the actual interface.



```
  ----- WELCOME TO RASPEDAL, YOUR FREE AUDIO DSP PLATAFORM ------




Instructions:

- FUZZ:
 - To toggle fuzz: q
 - To increase fuzz threshold: w
 - To decrease fuzz threshold: e
 - To toggle Crazy Mode: r
- DELAY:
 - To toggle delay: a
 - To increase delay time: x
 - To decrease delay time: c
 - To increase delay feedback: x
 - To decrease delay feedback: c
 - To increase delay mix: x
 - To decrease delay mix: c
- TREMOLO:
 - To toggle tremolo: z
 - To increase tremolo speed: x
 - To decrease tremolo speed: c
- FILTER:
 - To toggle filter: l

>>>
```

FIGURE 6 – Raspedal's user interface

The main goal of this interface is to use a small quantity of processing power while still enabling the user to shape the sound properly. All command keys can be changed easily in the *main* function.

In the next chapter, all implemented effects will be further explained.

4 Effects

Once the programming environment was ready, it was possible to start developing audio effects to test the functionality of the platform. The effects developed where:

- Fuzz, an intentional non-linear distortion effect;
- Delay, a time-based effect;
- Tremolo, a modulation effect;
- Low pass filter, a linear distortion effect.

To facilitate development, all function follows the same logic: they take the audio signal present in the *out[n_{buffer}]* array, modify it and store it again in the same array. This way, there is no need to always be the same function starting the signal chain, effect order can be changed and future users already have a framework to follow when developing new effects.

The next topics will explain each one of the effects and how they were implemented.

4.1 Fuzz

The fuzz was one of the first guitar effects ever developed using solid state compenents, back in the '1960s [9]. Before fuzz, distorted guitar sounds were achieved by saturating valve amplifiers. By the nature of valves, this signal distortion is gradual, fluid. It's called soft clipping. The fuzz, in a different way, is done using transistors and diodes, that affect the audio signal more radically and abruptly. This way of distortion, on the other hand, is called hard clipping.

The following figure [10] represents the difference between both types of distortion.

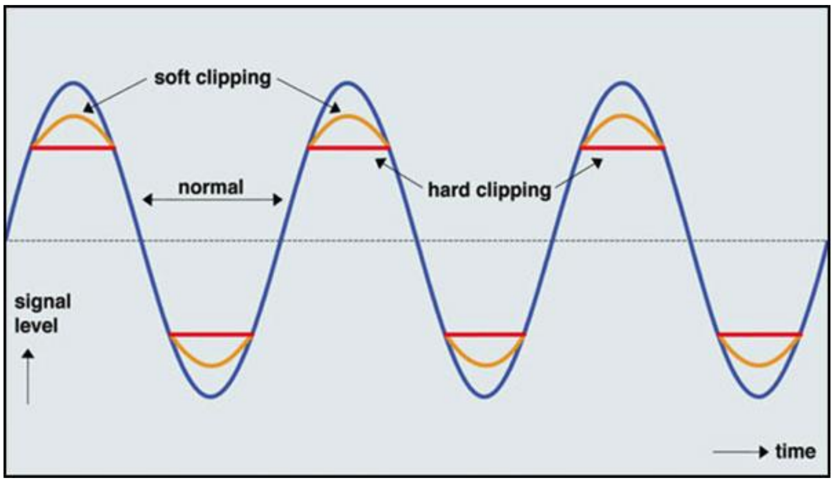


FIGURE 7 - The difference between soft and hard clipping

Besides the hardness of the clipping, different distortion mechanisms may be symmetrical or asymmetrical. In FIGURE 6 example, since the distortion is applied in both positive and negative signal cycles, the clipping is considered symmetrical. If it had occurred in only one cycle, it would be an asymmetrical clipping.

Another parameter in fuzz implementations is the threshold above which the signal is attenuated or flattened.

In the present fuzz implementation, *add_fuzz()*, hard symmetrical clipping is used, and the distortion threshold is controlled by the user.

4.2 Delay

Delay is the effect of the repetition of the signal in a constant time interval after it's generated. It can be done using a handful of technologies: magnetic tape, bucket brigade circuits, digital IC components or via software.

Usually, the main part of a delay circuit is a mechanism that delays the signal by a determined amount of time. By itself, it would not generate the desired effect since it's capable of only one delayed repetition and the original signal is lost. So, the following circuit can be used to turn a single delay mechanism into the desired effect.

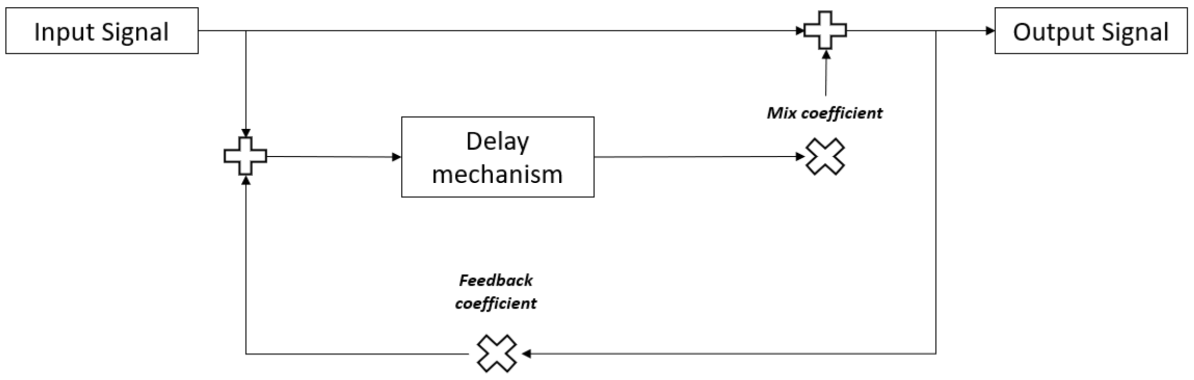


FIGURE 8 - Delay circuit diagram

The effect generation occurs in the following way: the raw input signal is duplicated, and one copy is sent to the delay mechanism. This mechanism then generates an identical signal delayed by a time constant *delay time*. This delayed signal is then multiplied by a *mix coefficient*, which determines how much of the delayed signal will be heard in the final signal. After that, it's summed to the other copy of the original signal.

To enable multiple repetitions, there is usually a feedback circuit. This feedback circuit sends the output signal to the input of the delay mechanism, after multiplying it by a *feedback coefficient*. This feedback coefficient determines how many repetitions of the sound will be heard: if the coefficient is zero, there will ever be only one repetition of the signal; if it's greater than 0 and less than 1, there will be a finite quantity of repetitions; and above 1, infinite repetitions

In the present delay implementation, *add_delay()*, the user can control *delay time*, *mix coefficient* and *feedback coefficient.*

4.3 Tremolo

Tremolo is a modulation effect that impacts on the amplitude of the signal. This causes a ripple effect to the sound. Classic guitar amplifier, such as the Vox AC30, famously used by George Harrison and Brian May, and the Fender Super Reverb have takes on the tremolo effect built-in.

To a better understanding of how it works, an analogy with a radio AM modulation is helpful.

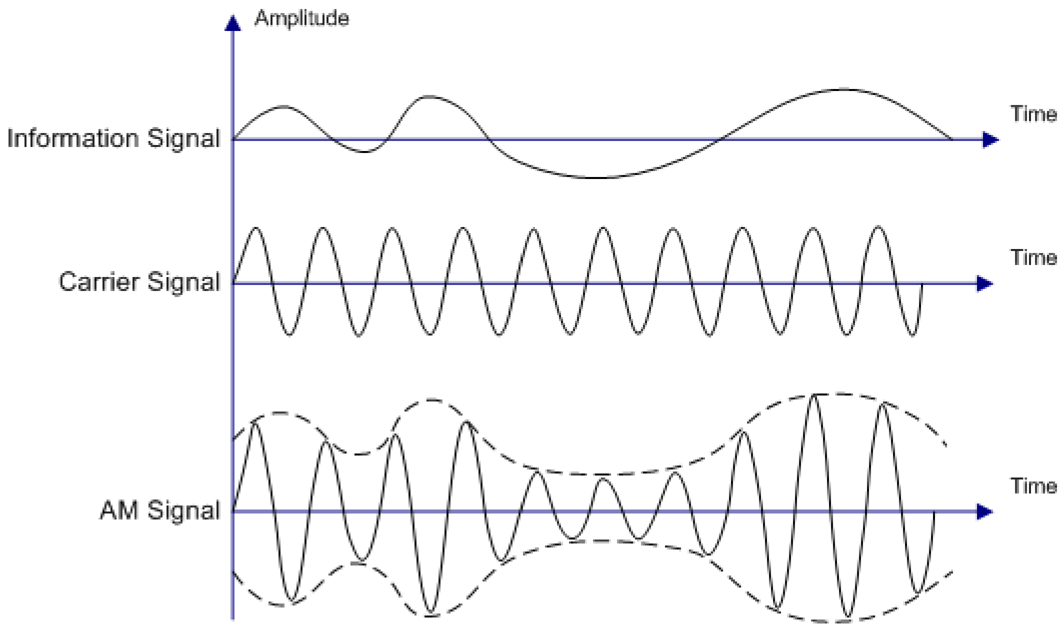


FIGURE 9 - Operation of an AM modulation

By analogy to the FIGURE 8, in a tremolo effect circuit, the input signal would take the place of the AM *carrier signal* and a cyclic signal, such as a sine or a triangular wave, would be used as the *information signal*.

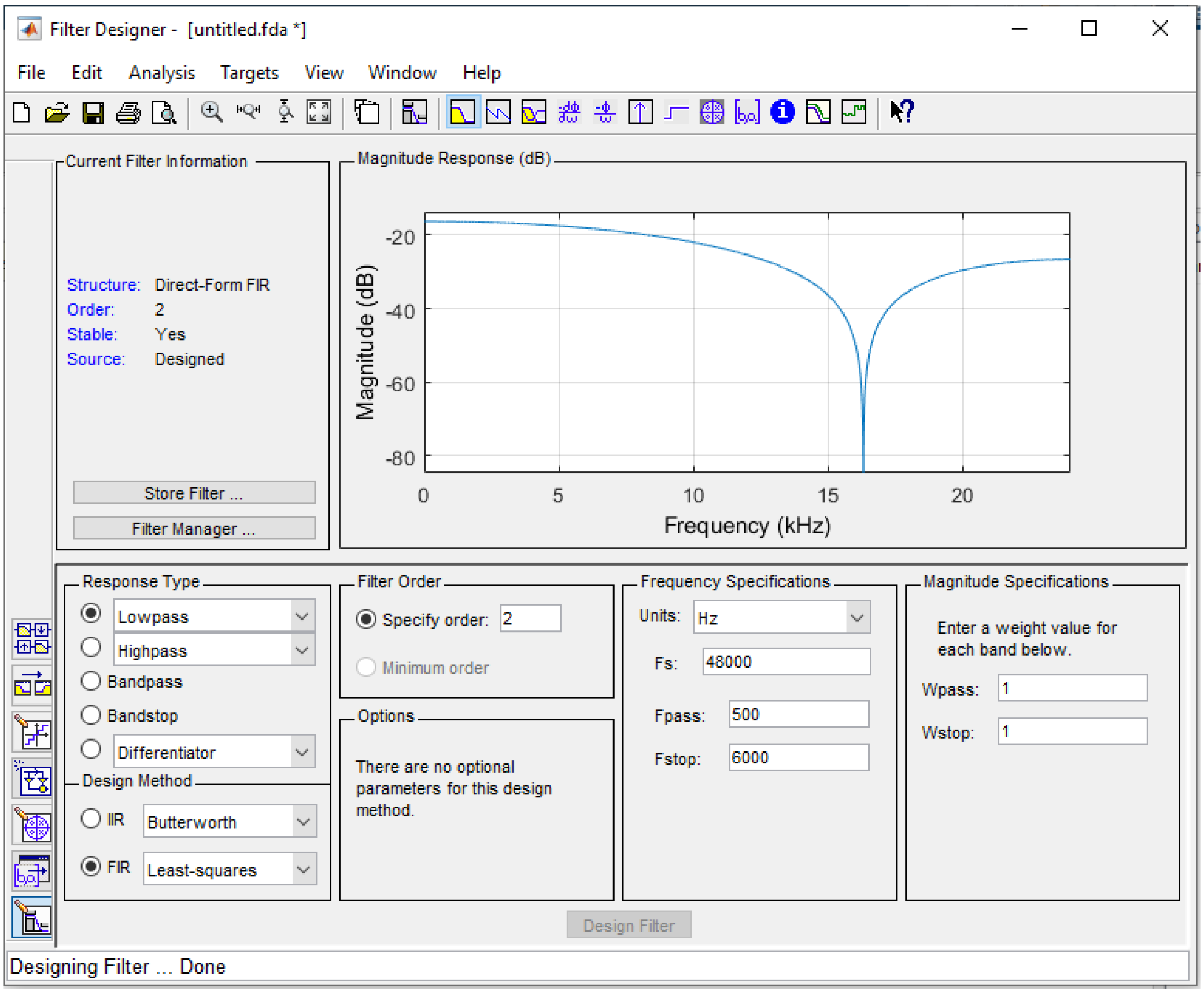
In the present delay implementation, *add_tremolo()*, a triangular wave is used as the cyclic signal and the user can control its frequency.

4.4 Filter

The filter effect is a linear distortion effect. That said, it consists of a variation in phase and/or amplitude of the signal without adding new frequencies. This intensity of this variation can usually depends on the frequency: higher frequencies can be effected more than lower frequencies, for example.

In audio application, filters are usual in amplifier equalization circuits and guitars tone controls.

The filter implementation was made using the finite impulse response method. Basically, each sample, after being processed by the filter, will be a linear combination between itself and a number of previous samples. The coefficients of this linear combination were calculated with the help of the *Matlab* software. The following figure shows the frequency response of the calculated filter:



Current Filter Information

Structure: Direct-Form FIR
Order: 2
Stable: Yes
Source: Designed

Response Type

[✓ Lowpass]
Highpass
Bandpass
Bandstop
Differentiator

Design Method

IIR Butterworth
[✓ FIR] Least-squares

Filter Order

[✓ Specify order:] 2
Minimum order

Options

There are no optional parameters for this design method.

Frequency Specifications

Units: Hz
Fs: 48000
Fpass: 500
Fstop: 6000

Magnitude Specifications

Enter a weight value for each band below.
Wpass: 1
Wstop: 1

Designing Filter ... Done

FIGURE 10 - Filter effect specifications

The resulted effect is a low pass filter that reaches -20dB at approximately 7,5KHz. Tonally speaking, it removes the higher frequencies, making the signal darker.

The real filter does not achieve the specifications present in FIGURE 10 because the filter order (number of previous samples present in the linear combination) was limited to achieve maximum performance.

5 Discussion and Implications

To achieve its goals, the digital audio processor should be able to provide professional performance and programmability at a low cost. Therefore, the following topics will be taken into consideration to evaluate whether the equipment has achieved its goals.

5.1 Latency

Latency is the lag between when the audio signal enters the system when it emerges. In an audio system, a possible contributor to latency could be ADC, digital signal processing time, DAC and others.

Once the main focus of the processor is to deal with the real-time instrument signal, the latency should not be felt by the player. For video applications, for example, the typical latency is between 15ms and 30 ms [11].

In the final product, there is no perceptible lag, such that it does not spoil the player's experience.

5.2 Sound Quality

When discussing digital sound quality, there are two main factors: sample rate and bit depth. The sample rate is the frequency at which the analog audio sample will be read. Bit depth, on the other hand, is the number of bits used to write the information read in every sample.

The Nyquist theorem states that for a signal of maximum frequency f, a sampling of frequency higher than $2f$ will contain all the information of the original signal [12]. Since the upper limit of the human hearing is 20 kHz [13], any sample rate above 40 kHz will be enough. That being said, the initially chosen sample rate was 44.1 kHz, that is a commercial standard for music recording and playback, and is used, for example, in CD's. Since there was no signal of lag, a higher resolution, 48 kHz, was utilized.

The bit rate is also based on commercial standards. Professional live and recorded sound uses 24 bits of resolution, which is already higher than CD standards, which uses 16 bits. The mains focus was 24 bits, but if latency becomes a problem, 16 bits are still a viable option without noticeable quality loss.

The focus was a bit resolution of 24 bits, and 16 bits were still considered. But since lag didn't show up as a limitation, a higher bit resolution was chosen: 32 bits. This resolution is usual in high-end studio and live equipment and emphasizes the power of the platform.

With this bit resolution, the platform needs to precess 32*48000 float numbers per second. Since it didn't struggle to process this amount of information, there is no reason to go back to smaller resolutions or sampling rates, since it could result in an audible quality loss and decreased signal-to-noise ratio. [14]

The only problem related to sound quality is a crackling sound when engaging the tremolo effect, and eventually the filter. There is not variation on the lag perception when the crackling occurs.

5.3 Estimated costs

To make the platform accessible, the production cost is one of the main concerns. All component choices so far have taken this into account. Considering the Raspberry and the Audio Injector Audio interface, the costs so far are bellow R$300. Values up to R$600 are considerate valid.

5.4 Replication and reutilization easiness

During all the development process, a lot of major decisions were based in the easiness of reutilization. Some of these decisions are:

- All software utilized is open-source and the hardware is available in major retail sites, such as eBay;
- The code structure is aimed to be simple, in only one file;
- All effect's code follows the same structure, that can be used by the user.

In addition, there is a GitHub repository [15] with all necessary archivers and instructions. Besides the possible hardware importation time, all replication process can be done in less than one day, allowing DSP application to be utilizing the platform.

5.5 Qualities and limitations

By the end of the platform development, its qualities and limitations can be discussed. To illustrate the processing power of the platform, an equivalent signal chain was simulated with commonly used guitar pedals.

FIGURE 11 - Equivalent signal chain built with guitar effects

In the pedalboard above, we have the following pedals:

- Dunlop Fuzz Face, a fuzz effect pedal (~ R$ 1.000 [16]);
- Boss Tremolo, a tremolo effect pedal (~ R$560 [17]);
- MXR Carbon Copy Delay, a delay effect pedal (~ R$1.100 [18]);
- Boss Equalizer, a pedal with filter capabilities (~ R$ 430 [19])

With the platform in its current development stage, all the effects above can be simulated digitally with high end grade convertion rate and sample resolution. Besides that, the platform is versatile enough to simulate more and completely different effects, mainly because of its open-source code.

Comparing the processing power of the platform with the vintage effect units that motivated its development, in the delay effect for example, while a original Binson Echorec had maximum delay time of 300ms [20], the platform actualy have a 10s maximum delay time, and with focused optimizartion it coul be even more.

Apart of the guitar effects environment, the platform can be used to other scientific purposes, such as natural language processing and other DSP application, since its already interfaced with specific vector, algebra and machine learning libraries. Above that, the plataform is self contained, and the user can use it without being stuck to another computer.

All the above said, one of the platform main strengthes is its price. Even the simple and acssessible pedalboard represented in FIGURE 11 costs more than 10 times the initial price of the platform, that could even be optimized with further sourcing.

About its limitations, some effects implementations ,like the tremolo, are presenting minor sound discrepancies. The source of this discrepancies is not performance related, since there is no preceptive lag time, but has not yet being pointed. Meantime, this minor discrepancies does not affect the product capabilities as a low-budget, open-source digital signal processing platform, with professional audio quality.

6 Conclusions

The Raspberry based digital audio processor fill a gap between low-quality open-source options and the high-fi closed-source ones, with a reasonable price. It allows both engineering students and artists to prototype audio applications with flexibility and proper professional audio quality.

Even though the present audio processing application are representative, the main development mote was to make the product a platform that could be used and further developed by others and, in this sense, the product have a lot of features that facilitates that:

- Simple code structure;
- Replicable effect implementation framework;
- Already implemented interface with major DSP and machine learning libraries;
- Fully documented replication procedure.

Thus, the platform will facilitate the development of future application and help other artists of scientists to prototype and implement their ideas.

Reference

[1] REVERB. **Binson Guild Echorec Model T6F-A**: sale ad. Available at: <https://reverb.com/item/2354037-binson-guild-echorec-model-t6f-a-1960-s-green/>. Accessed on: 20, November 2019.

[2] SWEETWATER. **Eventide H9 Core Multi-effects Pedal**: sale ad. Available at: <https://www.sweetwater.com/store/detail/H9Core--eventide-h9-core-multi-effects-pedal>. Accessed on: 20, November 2019.

[3] MERCADO LIVRE. **Eventide H9 Core Harmonizer **novo****: sale ad. Available at: < https://produto.mercadolivre.com.br/MLB-1277901317-eventide-h9-core-harmonizer-novo-_JM?quantity=1&variation=40723330490#position=1&type=item&tracking_id=67508767-5c52-4aeb-8e68-f1852bffd2e3>. Accessed on: 20, November 2019.

[4] RASPBERRY ORGANIZATION. **RASPBERRY PI COMPUTE MODULE 3+**: datasheet. Available at: <https://www.sweetwater.com/store/detail/H9Core--eventide-h9-core-multi-effects-pedal>. Release on 1, January 2019.

[5] AUDIO INJECTOR. **STEREO RASPBERRY PI SOUND CARD**: Main page. Available at: < http://www.audioinjector.net/rpi-hat>. Accessed on: 20, November 2019.

[6] ALSA PROJECT. **ALSA PROJECT – THE C LIBRARY REFERENCE**: Main page. Available at: < https://www.alsa-project.org/alsa-doc/alsa-lib/index.html>. Accessed on: 20, November 2019.

[7] GTKIOSTREAM. **GTKIOSTREAM**: Main page. Available at: < http://gtkiostream.flatmax.org/>. Accessed on: 20, November 2019.

[8] JACKAUDIO. **JACK AUDIO CONNECTION KIT**: Main page. Available at: < https://jackaudio.org/>. Accessed on: 20, November 2019.

[9] SUNPOWER ELETRONICS. **A BRIEF HISTORY OF GUITAR EFFECTS PEDAL**: Available at: < https://www.sunpower-uk.com/news/a-brief-history-of-guitar-effects-pedals-infographic/Accessed on: 20, November 2019.

[10] SWEETWATER. **BOOST, OVERDRIVE, DISTORTION & FUZZ PEDALS – WHAT'S THE DIFFERENCE?** Available at: < https://www.sweetwater.com/insync/boost-overdrive-distortion-fuzz-pedals-whats-the-difference/>. Accessed on: 20, November 2019.

[11] Sara Kudrle; et al. (July 2011). "Fingerprinting for Solving A/V Synchronization Issues within Broadcast Environments". Motion Imaging Journal.

[12] Cutnell, John D. and Kenneth W. Johnson. Physics. 4th ed. New York: Wiley, 1998: 466.

[13] Proakis, Digital communications, 3th ed. New Jersey, 1996: 763

[14] Pohlman, Ken C. (1989). Principles of Digital Audio 2nd Edition. SAMS. p. 60.

[15] GITHUB. **RASPEDAL GITHUB REPOSITORY** Available at: < https://github.com/eduardohferreiras/RasPedal>. Accessed on: 20, November 2019.

[16] SUBMARINO. **PEDAL DUNLOP JDF2 FUZZ FACE:** sale ad. Available at: < https://www.submarino.com.br/produto/10490973/pedal-dunlop-jdf2-fuzz-face-cod-1111?WT.srch=1&acc=d47a04c6f99456bc289220d5d0ff208d&epar=bp_pl_00_go_g35175&gclid=Cj0KCQiA5dPuBRCrARIsAJL7oej_X3tD8HdqZGoGodK5xhT7MOErwVubsXl3cE6YSX0tEICOJBTiOroaAvBEEALw_wcB&i=57803ebeeec3dfb1f87e9304&o=560c656f6ed24cafb528ab13&opn=XMLGOOGLE&sellerId=4112118000162&sellerid=4112118000162&wt.srch=1>. Accessed on: 20, November 2019.

[17] SUBMARINO. **PEDAL BOSS TR-2 TREMOLO:** sale ad. Available at: < https://www.submarino.com.br/produto/217777558/pedal-boss-tr-2-tremolo?WT.srch=1&acc=d47a04c6f99456bc289220d5d0ff208d&epar=bp_pl_00_go_g35175&gclid=Cj0KCQiA5dPuBRCrARIsAJL7oejJsmUesi9Hls5N2k7jgphvdO_jCx0-ZnkNEzNpQlejV7c23ywPL08aAvYSEALw_wcB&i=573fedcceec3dfb1f8034a52&o=5d7007126c28a3cb50953195&opn=XMLGOOGLE&sellerId=10755289000102&sellerid=10755289000102&wt.srch=1>. Accessed on: 20, November 2019.

[18] SUBMARINO. **PEDAL MXR CARBON COPY ANALOG DELAY DUNLOP:** sale ad. Available at: < https://www.submarino.com.br/produto/26449606/pedal-mxr-carbon-copy-analog-delay-dunlop?WT.srch=1&acc=d47a04c6f99456bc289220d5d0ff208d&epar=bp_pl_00_go_g35

175&gclid=Cj0KCQiA5dPuBRCrARIsAJL7oeixXUHsBkb7QskRsCZAU0zgtNWQG6sj04htf89dhwGHqZUMxXIQ9MAaApWLEALw_wcB&i=5a17b52feec3dfb1f88ef2d0&o=5995e5dceec3dfb1f85557ac&opn=XMLGOOGLE&sellerId=10885874000127&sellerid=10885874000127&wt.srch=1>. Accessed on: 20, November 2019.

[19] AMERICANAS. **PEDAL BOSS GE-7 EQUALIZER GE7:** sale ad. Available at: < https://www.americanas.com.br/produto/16826008/pedal-boss-ge-7-equalizer-ge7?WT.srch=1&acc=e789ea56094489dffd798f86ff51c7a9&epar=bp_pl_00_go_im_todas_geral_gmv&gclid=Cj0KCQiA5dPuBRCrARIsAJL7oehGCw5NTguUlPJ8uNxNsvfwk5klggfq09QZ99c2xPdbvpisMCLJ2wgaAuTWEALw_wcB&i=5cb7eb3449f937f6259ca3ba&o=590caf98eec3dfb1f83f5696&opn=YSMESP&sellerId=28591923000184&sellerid=28591923000184&wt.srch=1>. Accessed on: 20, November 2019.

[20] CATALINBREAD. **CATALINBREAD ECHOREC MANUAL**. Available at: < https://www.catalinbread.com/manuals/ECHORECmanual.pdf>. Accessed on: 20, November 2019.

Appendix A – Dependencies Installation

1. To install the Raspbian OS, follow the instruction in: https://www.raspberrypi.org/documentation/installation/installing-images/README.md

2. To install all other dependencies, run the following commands:

   ```
   sudo apt install libsox-dev libeigen3-dev libfftw3-dev libasound2-dev libjack-jackd2-dev sox libgtk2.0-dev

   sudo apt-get install git g++ autoconf libtool

   git clone https://github.com/flatmax/gtkiostream.git

   cd gtkiostream

   ./tools/autotools.sh

   ./configure --disable-octave

   make -j2 (This command can take about one hour to run)

   sudo make install
   ```

3. Reboot the unit.

4. Download and install the .deb.tar.gaz file in: http://forum.audioinjector.net/viewtopic.php?f=5&t=3

5. Run:

   ```
   audioInjector-setup.sh

   alsactl --file /usr/share/doc/audioInjector/asound.state.MIC.thru.test restore

   alsactl --file /usr/share/doc/audioInjector/asound.state.RCA.thru.test restore
   ```

6. Enjoy! :)

Appendix B – Source Code

```
#includ
e
<JackCl
ient.H>
           #include <iostream>
           #include <new>
           #include <cmath>
           #include <string>

           using namespace std;

           #include <unistd.h> // for sleep

           //Parametro Fuzz
           float volume_fuzz;
           float teto_fuzz;
           float pteto_fuzz;
           float octave_on;
           void add_fuzz (jack_default_audio_sample_t *out, int j)
           {
                   teto_fuzz = pow(2.0 , pteto_fuzz)/pow(2.0 , 32.0);

                   if(out[j] > teto_fuzz){
                           out[j] = teto_fuzz;
                   }
                   else if(out[j] < -1*teto_fuzz){
                           out[j] = octave_on*teto_fuzz;
                   }

           }

           //Paramentros Delay
           float *cauda_delay;
           int tamanho_cauda;
           float delay_time;
           int counter;
           float mix;
           float feedback;
           void add_delay (jack_default_audio_sample_t *out, int j)
           {
                   cauda_delay[counter] = out[j] + feedback*cauda_delay[counter];
                   counter ++;
```

```
        counter = counter % tamanho_cauda;

        out[j] = (out[j] + mix*cauda_delay[counter]);
}

//Parametros Tremolo
int contador_tremolo; //aprox 1 segundo entre os zeros
int velocidade_tremolo;
int subindo;
void add_tremolo (jack_default_audio_sample_t *out, int j)
{
        out[j] = out[j]*(0.1 + 1.0*contador_tremolo/velocidade_tremolo);
}

//Parametros Filtro
float *filtro;
int counter_filtro;
int tamanho_filtro;
void add_filtro (jack_default_audio_sample_t *out, int j)
{
        filtro[counter_filtro] = out[j];
        out[j] = 0.2308*out[j] + 0.1929*filtro[(counter_filtro -
1)%tamanho_filtro] + 0.1929*filtro[(counter_filtro - 2)%tamanho_filtro] +
0.2308*filtro[(counter_filtro - 3)%tamanho_filtro];
        counter_filtro ++;
        counter_filtro = counter_filtro % tamanho_filtro;
}

//Variaveis de controle

int fuzz_on;
int tremolo_on;
int delay_on;
int filtro_on;

class JackFullDuplex : public JackClient {

        //processAudio é chamada a cada 1024 samples (0,021 segundo)
    int processAudio(jack_nframes_t nframes) {

                if (outputPorts.size()!=inputPorts.size()){
                        cout<<"Different input and output port count, don't know
how to copy. In port cnt="<<inputPorts.size()<<" output port
count="<<outputPorts.size()<<endl;
                        return -1;
                }
                jack_default_audio_sample_t *out = (
jack_default_audio_sample_t* ) jack_port_get_buffer ( outputPorts[0], nframes
); // the output sample buffer
        jack_default_audio_sample_t *in = ( jack_default_audio_sample_t* )
jack_port_get_buffer ( inputPorts[0], nframes ); // the inputs sample buffer

        for (uint j=0; j<nframes; j++){

                        out[j]= in[j];
                        if(fuzz_on == 1) add_fuzz(out, j);
                        if(tremolo_on == 1) add_tremolo(out, j);
                        if(delay_on == 1) add_delay(out, j);
                        if(filtro_on == 1) add_filtro(out, j);
                        out[j] = 2*out[j];

                }

                //Contado base do tremolo
                if(subindo == 1){
                        contador_tremolo ++;
                        if (contador_tremolo == velocidade_tremolo) subindo = 0;
                }
                else {
                        contador_tremolo --;
                        if (contador_tremolo == 0) subindo = 1;
                }
                return 0;
    }
};

int main(int argc, char *argv[]) {

        //Inicialização e paramentros do Fuzz
        pteto_fuzz = 25.0;
        octave_on = -1.0;

        //Inicialização e paramentros do Tremolo
        contador_tremolo = 0;
        velocidade_tremolo = 3;
        subindo = 1;

        //Inicialização e Parametros do Delay
        counter = 0;
        delay_time = 0.3;
        mix = 0.25;
        feedback = 0.5;

        tamanho_cauda = (int)floor(delay_time*48000);
        cauda_delay = new float[tamanho_cauda];
        for (uint j=0; j<tamanho_cauda; j++){
               cauda_delay[j] = 0;
        }

        // Inicializaçãoe parametros do Filtro
        tamanho_filtro = 4;
        filtro = new float[tamanho_filtro];
        for (uint j=0; j<tamanho_filtro; j++){
               filtro[j] = 0;
        }
        counter_filtro = 0;

        fuzz_on = 0;
        tremolo_on = 0;
        delay_on = 0;
        filtro_on = 0;


        //-------------------------------------------------------------------
-----------------//


    JackFullDuplex jackClient; // init the full duplex jack client

    // connect to the jack server
    int res=jackClient.connect("jack full duplex client");
    if (res!=0)
        return JackDebug().evaluateError(res);

    //cout<<"Jack : sample rate set to : "<<jackClient.getSampleRate()<<"
Hz"<<endl;
    //cout<<"Jack : block size set to : "<<jackClient.getBlockSize()<<"
samples"<<endl;

        // create the ports
    res=jackClient.createPorts("in ", 1, "out ", 1);
    if (res!=0)
        return JackDebug().evaluateError(res);

    // start the client connecting ports to system ports
    res=jackClient.startClient(1, 1, true);
    if (res!=0)
```

```
         return JackDebug().evaluateError(res);



       cout<<"\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\
n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n";
       cout<<"
----- WELCOME TO RASPEDAL, YOUR FREE AUDIO DSP PLATAFORM ------
\n\n\n\n\n\n\n";
       cout<< "Instructions:\n\n";
       cout << "- FUZZ:\n";
       cout << " - To toggle fuzz: q\n";
       cout << " - To increase fuzz threshold: w\n";
       cout << " - To decrease fuzz threshold: e\n";
       cout << " - To toggle Crazy Mode: r\n";
       cout << "- DELAY:\n";
       cout << " - To toggle delay: a\n";
       cout << " - To increase delay time: x\n";
       cout << " - To decrease delay time: c\n";
       cout << " - To increase delay feedback: x\n";
       cout << " - To decrease delay feedback: c\n";
       cout << " - To increase delay mix: x\n";
       cout << " - To decrease delay mix: c\n";
       cout << "- TREMOLO:\n";
       cout << " - To toggle tremolo: z\n";
       cout << " - To increase tremolo speed: x\n";
       cout << " - To decrease tremolo speed: c\n";
       cout << "- FILTER:\n";
       cout << " - To toggle filter: l\n\n";


       char comando;
       while (1)
       {
              cout << ">>>";
              cin >> comando;

              //Comandos Fuzz

              if(comando == 'q'){
                     if(fuzz_on == 1){fuzz_on = 0; cout << "Fuzz off!\n";}
                     else if(fuzz_on == 0){fuzz_on = 1; cout << "Fuzz
on!\n";}
              }

              if(comando == 'w'){
```

```
                        if(pteto_fuzz >= 31.0){cout << "Fuzz threshold is
already maximized!\n";}
                        else {pteto_fuzz = pteto_fuzz + 1.0; cout << "Fuzz
threshold raised!\n";}
                }

                if(comando == 'e'){
                        if(pteto_fuzz <= 1.0){cout << "Fuzz threshold is already
minimized!\n";}
                        else{ pteto_fuzz = pteto_fuzz - 1.0; cout << "Fuzz
threshold decreased!\n";}
                }

                if(comando == 'r'){
                        octave_on = octave_on*-1.0;
                        if(octave_on > 0){cout << "Crazy Mode on!\n";}
                        else if(octave_on < 0){cout << "Crazy Mode off!\n";}
                }

                //Comandos Delay

                if(comando == 'a'){
                        if(delay_on == 1){delay_on = 0; cout << "Delay off!\n";}
                        else if(delay_on == 0){delay_on = 1; cout << "Delay
on!\n";}
                }

                if(comando == 's'){
                        if(delay_time >= 10.0){cout << "Delay time is already
maximized!\n";}
                        else {
                                delay_time = delay_time + 0.1; cout << "Delay
time raised!\n";
                                tamanho_cauda = (int)floor(delay_time*48000);
                                cauda_delay = new float[tamanho_cauda];
                                for (uint j=0; j<tamanho_cauda; j++){
                                        cauda_delay[j] = 0;
                                }
                        }
                }

                if(comando == 'd'){
                        if(delay_time <= 0.11){cout << "Delay time is already
minimized!\n";}
                        else {
```

```
                delay_time = delay_time - 0.1; cout << "Delay
time decreased!\n";
                tamanho_cauda = (int)floor(delay_time*48000);
                cauda_delay = new float[tamanho_cauda];
                for (uint j=0; j<tamanho_cauda; j++){
                    cauda_delay[j] = 0;
                }
            }
        }
        if(comando == 'f'){
            if(feedback >= 0.99){cout << "Delay feedback is already
maximized!\n";}
            else {feedback = feedback + 0.05; cout << "Delay
feedback raised!\n";}
        }
        if(comando == 'g'){
            if(feedback <= 0.005){cout << "Delay feedback is already
minimized!\n";}
            else {feedback = feedback - 0.05; cout << "Delay
feedback decreased!\n";}
        }
        if(comando == 'h'){
            if(mix >= 0.99){cout << "Delay mix is already
maximized!\n";}
            else {mix = mix + 0.05; cout << "Delay mix raised!\n";}
        }
        if(comando == 'j'){
            if(mix <= 0.005){cout << "Delay mix is already
minimized!\n";}
            else {mix = mix - 0.05; cout << "Delay mix
decreased!\n";}
        }


        //Comandos Tremolo

        if(comando == 'z'){
            if(tremolo_on == 1){tremolo_on = 0; cout << "Tremolo
off!\n";}
            else if(tremolo_on == 0){tremolo_on = 1; cout <<
"Tremolo on!\n";}
        }
        if(comando == 'x'){
            if(velocidade_tremolo == 1){cout << "Tremolo speed is
already maximized!\n";}
```

```
                        else {velocidade_tremolo = velocidade_tremolo - 1; cout
<< "Tremolo speed raised!\n";}
                }
                if(comando == 'c'){
                        if(velocidade_tremolo == 100){cout << "Tremolo speed is
already minimized!\n";}
                        else {velocidade_tremolo = velocidade_tremolo + 1; cout
<< "Tremolo speed decreased!\n";}
                }

                //Comandos Filtro

                if(comando == 'l'){
                        if(filtro_on == 1){filtro_on = 0; cout << "Filtro
off!\n";}
                        else if(filtro_on == 0){filtro_on = 1; cout << "Filtro
on!\n";}
                }

        }



    return 0;
}
```