Cristóvão Diniz Trevisan, Victor Volochtchuk de Araujo

Guitar Digitizer

Curitiba, Brazil

2017, v-0.0.1

Cristóvão Diniz Trevisan, Victor Volochtchuk de Araujo

Guitar Digitizer

Project presented to the Electronics Academic Department as graduation material for the course of Electronic Engineering at UTFPR

Federal University of Technology - Paraná – UTFPR

Electronic Engineering

Graduation Program

Supervisor: Gustavo Benvenutti Borba

Curitiba, Brazil

2017, v-0.0.1

Cristóvão Diniz Trevisan, Victor Volochtchuk de Araujo
Guitar Digitizer/ Cristóvão Diniz Trevisan, Victor Volochtchuk de Araujo. – Curitiba, Brazil, 2017, v-0.0.1-
51 p. : il.

Supervisor: Gustavo Benvenutti Borba

Graduation Final Project – Federal University of Technology - Paraná – UTFPR
Electronic Engineering
Graduation Program, 2017, v-0.0.1.

1. Hexaphonic Guitar 2. Digitizer 3. MIDI 4. Pitch Detection I. Guitar Digitizer II. Gustavo Benvenutti BorbaIII. Federal University of Technology - Paraná IV. Electronic Engineering

CDU 02:141:005.7

Cristóvão Diniz Trevisan, Victor Volochtchuk de Araujo

Guitar Digitizer

Project presented to the Electronics Academic Department as graduation material for the course of Electronic Engineering at UTFPR

Project Approved. Curitiba, Brazil, January 14, 2018:

Gustavo Benvenutti Borba
Supervisor

Professor
Invited 1

Professor
Invited 2

Professor
Invited 3

Professor
Invited 4

Curitiba, Brazil
2017, v-0.0.1

This work is dedicated to those who supported our way through engineering course, even more when ourselves tried to give up.

Acknowledgements

We give our special thanks to Eng. Mikhail Anatholy Koslowski, who gave us both intellectual (since he once used an equipament with the same purposed) and physical support (with compenents importation and equipaments). Also to our advisor who was present even in the moment that the initial idea came to life, in a situation nobody else would have stayed.

Abstract

Guitar are one of most popular instruments today, but there is one big disadvantage to use it: there is no good and affordable way to digitize it's music. The biggest problem with this is the cost to annotate music, as it needs to be done by manually. This project tries to build one such system, starting from passive hardware (hexaphonic pickup) to modern signal processing (pitch detection), attempting to produce a cheap and effective equipment for guitar music annotation by means of generating MIDI format data.

Key-words: guitar. digitizer. MIDI. pitch. detection. hexaphonic.

Resumo

Violões e guitarras estão entre os instrumentos mais populares da atualidade, mas existe uma grande desvantagem em os utilizar: não há um meio barato e eficaz para digitalizar sua música. O grande problema com isso é o alto custo para transcrever partituras, que atualmente é um processo manual. Esse projeto tenta construir um sistema com esse propósito, criando desde sensores passivos (captador hexafônico) até processamento digital de sinais moderno (detecção de nota), visando um produto barato e eficaz para anotação musical através da geração de dados no format MIDI.

Key-words: guitarra. digitalizador. MIDI. nota. detecção. hexafônico.

List of Figures

List of Tables

List of abbreviations and acronyms

ADC	Analog to Digital Converter
API	Application Programming Interface
AWG	American Wire Gauge
DIY	Do It Yourself
DMA	Direct Memory Access
DOM	Document Object Model
fft	Fast Fourier Transform
GUI	Graphical User Interface
IC	Integrated Circuit
IDE	Integrated Development Environment
ifft	Inverse Fast Fourier Transform
MIDI	Musical Instrument Digital Interface
npm	Node Package Manager
PCB	Printed Circuit Board
UI	User Interface
USB	Universal Serial Bus
n.d.	No Date

List of symbols

Ω	Ohm resistance unit
μC	Microcontroller
V	Volts
mV	milivolts

Contents

1 Introduction

With the advance of technology, music - and musical instruments - have also evolved to use it's advantages. They are a lot of use cases, the most noticeable ones being music annotation and creation (through electronic instruments, also known as synthesizers). All of the modern musical software and hardware use the same format to communicate, called MIDI.

To translate music playing it is needed to know which note is being played at a given time. This make it very easy to translate instruments that have separated keys for each note (the most noticeable one being piano) to MIDI, but very hard do the same for instruments like the guitar, that have a single output for multiple notes (each string has about 15 notes). The guitar is also a harmonic instrument, as it can play multiple notes simultaneously, which makes it's digitization even harder.

There are already a few commercial solutions for this, but not a very performant and cheap one. Recently a new pure software solution was released at a reasonable price which works very well for live MIDI playing, but not enough for music annotation. There are also a few hardware solutions available at the market, which perform well, but are very expensive. Table 1 shows the most relevant solutions in the current market.

Table 1 – Market Solutions

Name	Price (U$D)	Usage Complexity	Live Performance	Annotation Performance
Roland GK3 + GhostHexpander + GI20	700	Hard	High	High
Godin Freeway + GhostHexpander + GI20	800	Hard	High	High
Jam Origin - Audio to MIDI	100	Easy	High	Low/Medium
Migic	40	Easy	Medium	Low

Source: authors

2 Hardware

2.1 Pickup

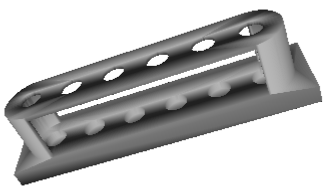
Electric guitars pickups are usually built by wrapping copper wire around magnets. The working principle is based on the variation of magnetic field, created by the string vibration. The vibration frequency of the string induces an electric signal on the output of the pickup (WALLACE, 2004) (NAVE, n.d.).

The designed pickup used the following main components:

- Base to assembly the set up magnet+coil
- 6 magnets
- Copper wire to wrap the magnets
- Cover to attach the set up on the guitar

After some studies it was decided to build the pickup base using a 3D printer, due the availability and reduced cost of this project. The model is showed on Figure 1 and was projected using the AutoCAD software.

Figure 1 – 3D pickup base project



Source: Authors

There were two materials available to print the model, *PLA* (Polylactic Acid) (3D. . . , 2013) and *ABS* (Acrylonitrile Butadiene Styrene) (3D. . . , 2013). It was decided to print the model on PLA because it attends the requisites of robustness of the project, is faster to print and have a lower cost when compared to the other materials. The built part is showed on Figure 2

With the pickup base ready, the coils (Figure 4) were dimensioned. The area of a turn was considered as a square with side dimensions equal to the wire diameter, and using that it was estimated the wire diameter as in Equation 2.2.

Figure 2 – 3D pickup base

Source: Authors

$$Turns = \frac{Area\ between\ magnets}{Area\ of\ each\ turn} = \frac{w * l/2}{d^2} = \frac{5mm * 12mm/2}{d^2} = 1000 \tag{2.1}$$

$$d = \sqrt{\frac{w\ *\ l}{2\ *\ Turns}} = \sqrt{\frac{5mm * 12mm}{2 * 1000}} \cong 0.17mm \cong 34AWG \tag{2.2}$$

Equation 2.1 was used with 1000 turns as a reference, based on a similar project found at research which uses that number (HEXAPHONIC. . . , n.d.). Using the area between 2 magnets (5mm wide and 12mm length) and the estimated area of a wire turn it is possible to calculate the wire diameter, as in Equation 2.2 - resulting in a diameter of 0.17mm, that can be converted to 37AWG, approximately.
This diameter is only feasible with an automatized process. The present project was built by manually wrapping copper wire around each magnet (Figure 3), so only 500 turns were feasible.
After mounting the coils, as showed on Figure 4 it was performed a test to verify the real voltage value induced by the variation of the magnetic field. It was used the LM 741 (LM741. . . , 2015) with a gain of 10, because it wasn't possible to verify the value on the coil output. With this simple test circuit it was possible to verify that the output value on the IC was 8mV, this value indicate that the output value on the coil is around 1mV.

After testing with one coil, the complete assemble was made, as showed on Figure 5 and Figure 6.

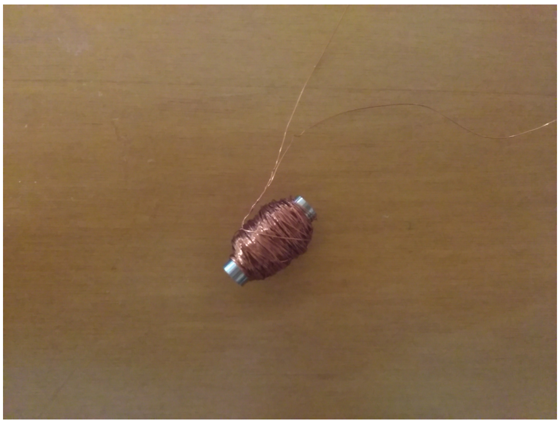
2.2 Amplifier Circuits

Amplifiers circuits (AMPLIFIER. . . , n.d.) are circuits which increase an electrical input signal according to a specific transformation function (gain) for each different topology. The circuit for small input signals is normally composed by an operational amplifier,

Figure 3 – Magnets used on the project

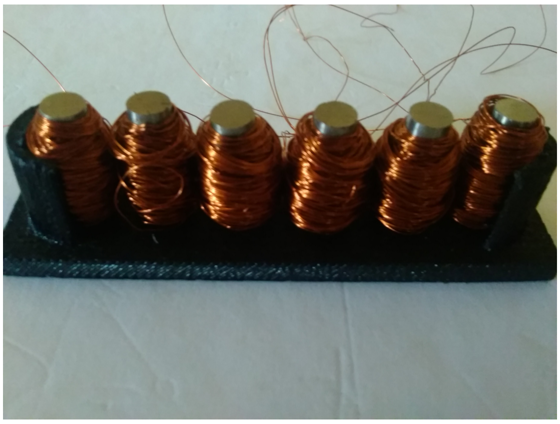
Source: magnets (n.d.)

Figure 4 – Coils assembled



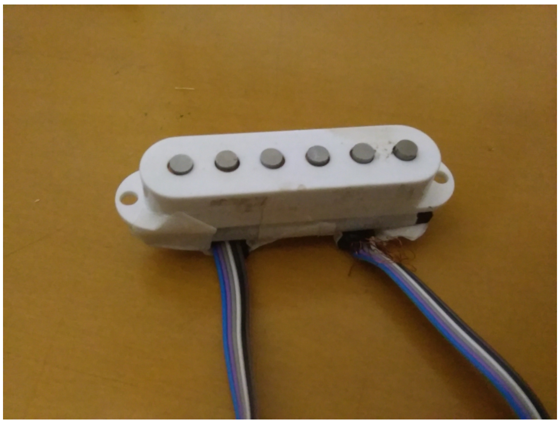
Source: Authors

Figure 5 – Coils assembled



Source: Authors

Figure 6 – Coils assembled



Source: Authors

resistors, trimmer potentiometers to adjust the gain and capacitors for frequency filtering. It was required by the project an amplifier with a gain of at least 1000, beacuse, as it was mentioned on section 2.1 the tested output coil presented the value around 0.1 mV and the circuit needs a signal of around 1V to work as expected on the AD converter, that can be powered by a sigle 5V supply, provided by the USB port. The circuit should be cheap (and thus compact) when compared to the existent systems which usually need multiple complex modules to perform the same action as the proposed in this project.
It was researched some types of amplifiers circuits in Milmann e Halkias (1981) and in Carter (2000). After verifying some amplifier topologies it was selected one circuit which attend to this project's requirements. This chosen circuit uses the Texas Instruments INA 326 Instrumental Amplifier.This topology can be supplied with a single 5V source and can reach the desirable gain without distortion on the desired frequency range (human audible frequencies) with a single amplifier per channel.

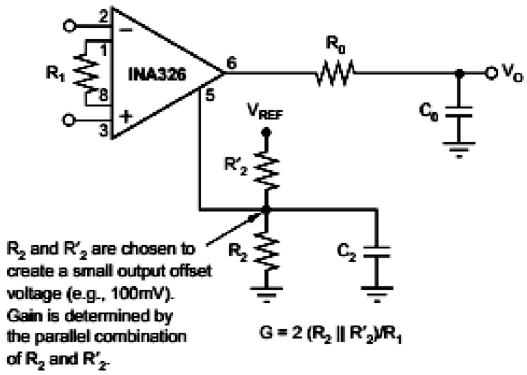
2.2.1 INA 326 Project

The project using the INA 326 IC started by researching the *component datasheet* (INA. . . , 2004) and the *supplier catalogue* (CARTER, 2000). In these documents it was verified the circuit topology Figure 7 which provides the desired gain for the pickup signal respecting this project's initial requests.
This recommended circuit's gain is obtained by Equation 2.3 which is provided by the *component datasheet* (INA. . . , 2004):

$$G = 2 * \frac{(R_2||R_2')}{R_1} \tag{2.3}$$

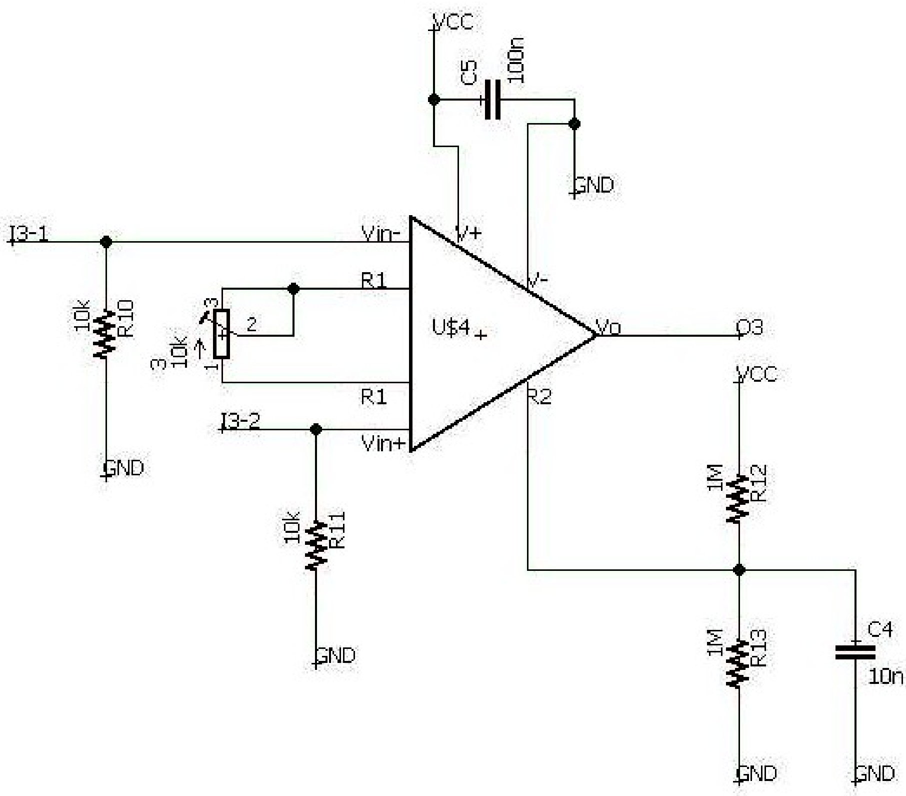
Both the R_0 resistor and the C_0 capacitor were excluded because they were not relevant for project requirements. The values for the remaining components were calculated and the circuit was build to test it's results for the desired application. After verifying the circuit does work properly, giving the desirable value on the output and without distortions on single supply method, the schematic Figure 8 was developed using the software CadSoft Eagle Professional 7.6.0.

Figure 7 – INA 326 topology



Source: INA... (2004)

Figure 8 – INA 326 Schematic Circuit



Source: Authors

It was decided to use trimmer potentiometers on the amplification circuit to regulate the gain for each channel. The desired gain range was achieved by selecting the component values showed at Figure 8. The complete schematic is just a replication of Figure 8 for each channel, and can be seen at Figure 25.

A PCB was then built, using the same software described on the schematic modeling, resulting in the board seen at Figure 9. It has two layers, with track width of 15 mils. It was assembled on FR4 dual layer copper board, using both through-hole and surface-mount technologies. This choice was made due the facility of the assembly of the through-hole components and the availability of the IC only in surface (SOP-8) encapsulation. The generated gerber files were sent to an internal manufacturer at UTFPR, which gave the board seen at Figure 10.

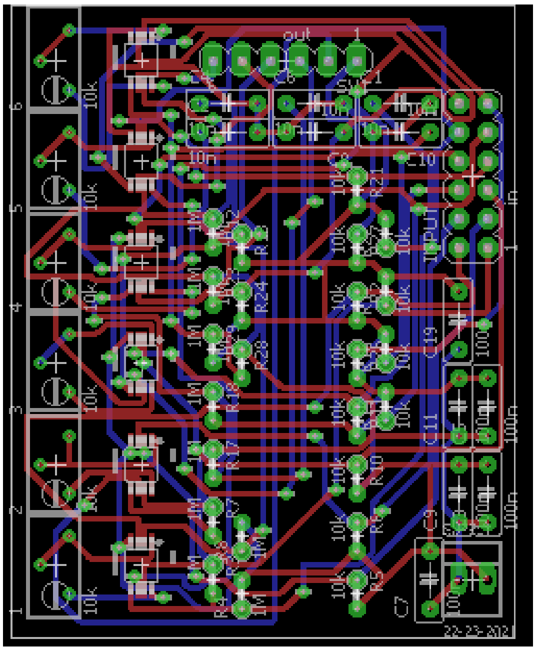
The component list for this board is as in Table 2.

Table 2 – INA Board BOM

Name	Quantity	Value
INA 326	6	
Trimmer Potentiometer	6	10kΩ
Ceramic Capacitor	6	10nF
Electrolytic Capacitor	6	100nF x 50V
Resistor	12	10kΩ
Resistor	12	1MΩ
Pin bar	1	6 positions
Pin bar	1	12 positions dual track
Pin bar	1	2 positions

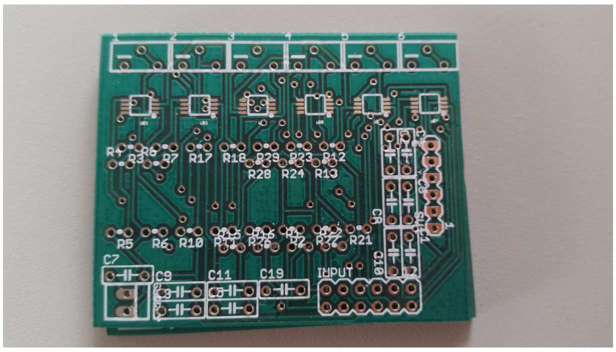
Source: Authors

Figure 9 – Projected INA PCB



Source: Authors

Figure 10 – PCB Project



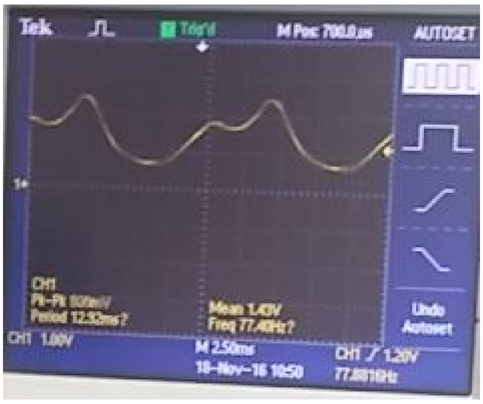
Source: Authors

After soldering all the components the assembled PCB was as showed on Figure 11, then it was performed some bench tests to verify the functionality of the amplifier circuit. The result showed that the circuit attends the required functionality, amplifying the signal from a peak-to-peak of about 1mV to 1.5V for the entire range of audible frequencies, proving that the circuit is working perfectly and attending the demands of the project. After the bench test the system was connected to the pickup to verify if the guitar signal would be amplified as needed for the conversion process. The result was satisfactory and attended well the purpose of the project, as showed on Figure 12 which shows the resultant electric wave when one string is played.

Figure 11 – PCB Board

Source: Authors

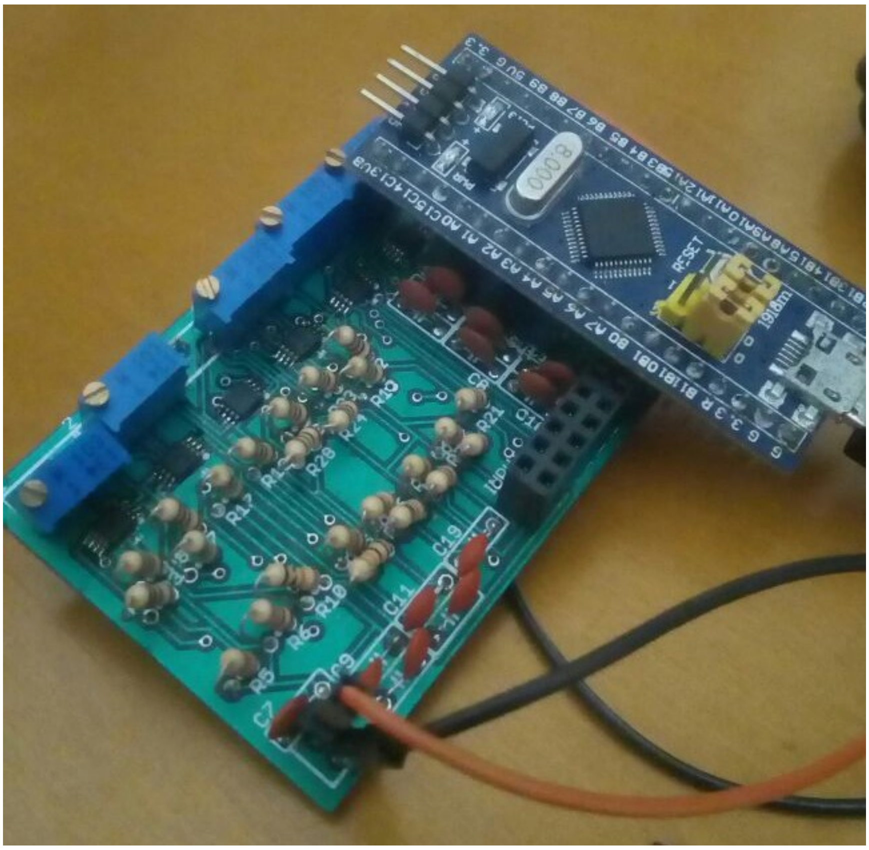
Figure 12 – Amplified pickup signal with INA circuit



Source: Authors

After perform all the functionality tests it was attached the amplifier circuit to the microprocessor circuit, as showed onFigure 13.

Figure 13 – Captation system



Source: Authors

3 Firmware

3.1 Specifications

The firmware is basically an analog sampler, all it has to do is sample six analog channels, add a header (to identify the beginning and check continuity) and send it through USB. Lets start by listing the requirements for the hardware.

3.1.1 Requirements

a) Super cheap

b) 6 analog channels (more precision is better)

c) High sample rate (at least 10 kHz for each channel, but ideally 44 kHz or more)

d) Fast USB support, to send the data with headers

Based on this requirements the minimum transfer speed can be calculated, let's consider that a header will be set (SENT?) for every 252 samples (42 for each channel) and it has 4 bytes (3 of identification - to assure it is the header and not some data - and a counter). Previewing the worst case, each sample is 2 bytes long. The transfer rate given by Equation 3.1.

$$transfer\ rate = \left(\frac{channels * \frac{bytes}{sample} * \frac{samples}{package} + header\ size}{\frac{samples}{package}} \right) * f_s [B/s] \tag{3.1}$$

Considering that f_s has to be somewhere between 10 kHz and 50 kHz the transfer rate numerical result is given by Equation 3.2.

$$transfer\ rate = \left(\frac{6 * 2 * 252 + 4}{252} \right) * f_s = 12.0159 * f_s = 120.159 - 600.794 [kB/s] \tag{3.2}$$

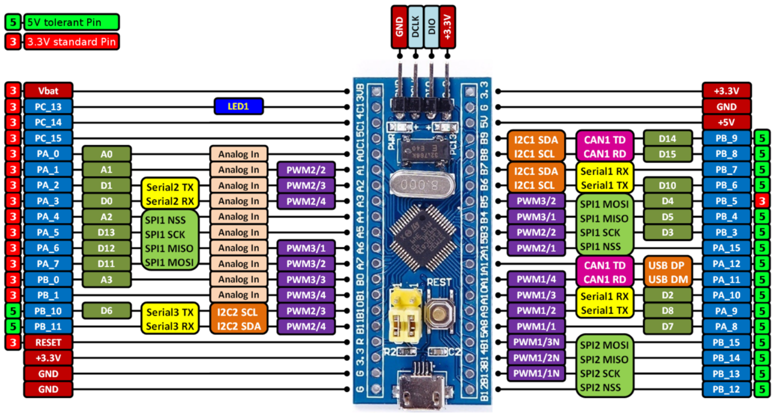
USB transfer speed is usually referred in Mbps, which gives a range between 961.27 and 4806.34 Mbps. This is too high for serial communication (typical max of 1 Mbps) so we need to add a requirement for raw USB support, which allows bulk transfer that can a have transfer rate up to 12Mbps (USB full-speed standard).

We still need to choose or exact sample rate (f_s), but first let's select which hardware will be used.

3.1.2 Microcontroller Selection

There are too many microcontrollers that fit our requirements, but the most popular and cheap one is clearly the ARM from ST called STM32F103C8T6 (STM32F103XX..., 2015), and that is why it was selected.
It has eight 12 bits ADC inputs (with 2 parallel channels), DMA for the ADCs and USB full-speed support. It's also relatively fast (72 MHz clock, 32 bits architecture). All this for under 2 U$D in a developing board from China (the actual μC is under 0.2 U$).
The board an it's pinout can be seen in Figure 14.

Figure 14 – STM32F103C8T6 Board



Source: Hudak (n.d.)

3.1.3 Sample Frequency

Usage of the ADCs for the selected μC can be optimized by using continuous sampling mode in conjunction with DMA (STM32F103XX..., 2015, ch. 11). In this mode the sample frequency is controlled by a register that sets the convolution time, which gives more precision the more ADC clock cycles (longer) it takes to sample.
The first variable chosen for this setup is the ADC clock, which is set by dividing the μC clock by 2, 4, 6 or 8. The ADC also has a maximum clock of 14 MHz. Taking in account the μC clock of 72 MHz, the highest possible value for the ADC clock is 12 MHz, which is set by a divider value of 6.
The last value to be chosen is the mentioned convolution time (T_c is calculated as the selected value plus 12.5 ADC clock cycles), which gives a sample frequency calculate by Equation 3.3.

$$f_s = \frac{ADC\ clock}{T_c + 12.5} * \frac{ADCs}{channels} = \frac{12}{T_c + 12.5} * \frac{2}{6}\ [MHz] \tag{3.3}$$

Table 3 shows the calculated results for each possible register value (using Equation 3.3). Based on it the chosen sampling frequency is 47.619 kHz. By using Equation 3.2 we can also calculate the actual data transfer rate (for all channels, including header), resulting in a total of 572,185 kHz. This last value will be used to test the USB communication.

Table 3 – ADC Sampling Frequencies

Register Value	**Convolution Time [cycles]**	**Sampling Frequency [kHz]**
000	1.5	285.71
001	7.5	200
010	13.5	153.85
011	28.5	97.56
100	41.5	74.07
101	55.5	58.82
110	71.5	47.62
111	239.5	15.87

Source: authors

3.2 Implementation

3.2.1 First Attempt

We first tried to build the firmware from scratch, using the tools given by the manufacturer, essentially a set of driver abstractions (HAL drivers). The problem found is that these abstractions are too slow, and don't work when the firmware uses the hardware close to it's limits (as we do for both transfer and sampling rates).

3.2.2 Second Attempt

In the research it was found a high quality open source project called MiniScope (MINISCOPE, n.d.), in which a few options of low budget DIY digital scope (using different μCs) are presented. One of the μCs used by MiniScope is the one selected, so for the implementation it was taken it's firmware as a base project. In this project the author claims to sample and transfer two channels at 461 kHz (but 8 bits only), which is very close to our needs (it is needed a little more transfer but much less sampling speed).

3.2.3 IDE

As it was taken MiniScope as a base project it will be used the same IDE as it, named CooCox (COOCOX, n.d.). It has a full set of tools, and it's completely free (no limitations).

3.2.4 Modifications

The base project samples 2 channels at a different speed, bit rate and does not add any headers to the data. It also has some code to answer a few commands. It was as simple as setting up the registers for 6 channels, changing the sample size, placing the already chosen speed (subsection 3.1.3) and removing any unused code.
The act of changing the sample size was not done by registers. MiniScope was already sampling with 12 bits, but it was ignoring the least significant ones when filling the USB buffer. What was done is to change the bits alignment and putting all the data received from the ADC to the USB buffer.

3.2.5 Testing

At first an attempt using the OS (Windows at that time, later Ubuntu) default driver as made. That did not work well, as it is too generic and thus slow.
At a second try, a simple libusb (LIBUSB, n.d.) program was built to test the transfer rate (calculated in subsection 3.1.3). The reported result is almost perfectly the calculated one.

3.2.5.1 Repository

Again, all code is available at GitHub (TREVISAN, 2017a).

4 Software

4.1 Tools Selection

4.1.1 Top Level Requirements

The exact implementation of each of the items will be discussed later, but to simply set the requirements a general list of them is:

a) Desktop GUI

b) Efficient signal processing (for pitch detection)

c) Real time graph visualization of the signals (oscilloscope like)

d) Access to *libusb* (LIBUSB, n.d.) API

e) Access to MIDI API

4.1.2 Language Choice

4.1.2.1 Java

The first choice was Java, as it meets all requirements. Desktop GUI can be done using Swing, a good library for pitch detection is also available (called TarsosDSP (SIX; CORNELIS; LEMAN, 2014)). There is also a binding to libusb called usb4java and native MIDI support.
Following that idea a functional prototype was built, but a few problems came to rise. The first is that usb4java high-level API had bugs and was not working correctly. The solution was to fall back to the lower level API, but that made things much more complicated as threading and synchronization problems had to be dealt with. There was no good library for real time visualization either, which made really hard to both debug and tune the frequency detection algorithm. On top of that Swing is at least non-pleasant compared to more modern UI programming, so a second approach came to be.

4.1.2.2 JavaScript

In alignment with both current work experience and world programming tendencies JavaScript was taken as a choice. It was seen that requirements fit much better now, for the following reasons.
For the desktop GUI, JavaScript has a few nice and mature Desktop GUI frameworks, like Electron and NW.js.
JavaScript is an interpreted language, and for that has a low efficiency when compared

to C++ or Java. That is huge problem, but there is an easy overcome. As this project tries to build a desktop application, Node.js will be used ultimately, and it has support for C++ bindings. That means the JavaScript code can call a compiled C++ library to calculate the pitch, thus solving the problem.
Graph visualization should not be a problem either as there are a lot of libraries for that. The most problematic requirement in Java was *libusb* support. It is available in JavaScript using *node-usb* (NODE-USB, n.d.), and a few simple tests returned good results with a much simpler API. MIDI was also tested and worked just fine.

4.1.3 Desktop Framework

Now that JavaScript is set as the final selection we need an environment to run it. There are two already listed really mature and popular choices: Electron and NW.js.
At first Electron was used to build a test application, because it is the most popular of the two (in fact even the editor used to write this words is built with it), but the pitch detection call was running slowly. As a mater of fact it was running much faster using pure JS code rather than the C++ library. A deeper research was needed, and the way Electron worked was getting in the project way, but first it's necessary to know what Node.js is.

4.1.3.1 Interpreter

JavaScript is a interpreted language and thus needs an interpreter. The most common one is Google V8, which happens to be the same one used in most web browsers as well as in Node. The difference between browsers and Node is simply the API that comes with them. Web needs firstly to access the UI (html) and ways to modify it, it also needs secure and limited access to hardware and internet calls. Of course that means web JavaScript code cannot use C++ libraries directly.
On the other hand Node is a more pure version of V8, it also gives the possibility to write and call C++ code (feature needed for this project), which ultimately makes it as capable as any desktop program can be. Node also comes with a hand-full set of native resources (like file system and full communication access), but it does **not** provide any kind of GUI. Knowing that it is possible to have a better understanding on how the two desktop environments work.

4.1.3.2 Electron

Electron by design has at least two processes running (HOW..., 2017), one for the "web" and other for Node access. The answer for how the web process access native resources is also to why the execution of the C++ processing library was slow: it uses inter-process communication (IPC). IPC makes things a lot slower, which ultimately makes impossible to use Electron in this project.

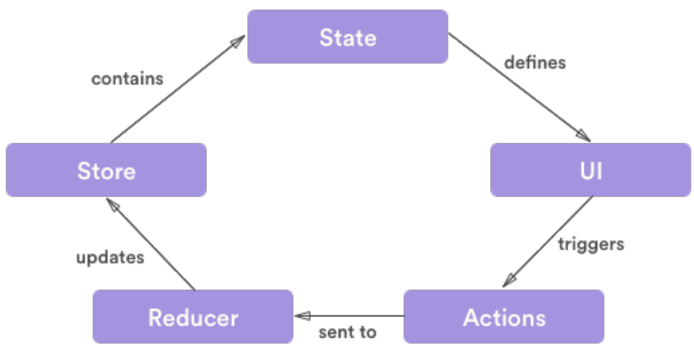
4.1.3.3 Node Webkit

Differently from Electron, NW.js (NODE. . . , n.d.; HOW. . . , 2017) takes the Node environment and combines it with Chromium into a single process, removing the use of IPC. Initial tests reported that the pitch detection library has fast execution as expected.

4.1.4 Architectural Tools

NW.js will go only as far as to give access to both Node and DOM API's. But that is too crude, and not what it was wanted by given up on Java Swing. Again based on current work experience and world tendencies the setup chosen is React + Redux.
React (REACT, n.d.) is a library created by Facebook and world wide used for UI applications. It uses a declarative component-based system that makes it easy to build scalable and reusable code.
React only go as far to help building the UI, but it is also needed to pass the state of the application to the UI components, and that is where Redux (REDUX, n.d.) comes in. It keeps all the application state stored in a single place, described by transition functions. That makes the storage system easy to be tested and used, because all actions (that modify the state) must be well defined and it doesn't rely on the UI (React), making it easy to test.
Figure 15 shows the flow of an application that uses React + Redux. It is obvious to see the simplicity it has, a single path must be followed. This simplicity is what makes it much easier to use against other frameworks like Java Swing.
There is still the choice of the visual library to use, and the chosen one is Semantic UI React (SEMANTIC. . . , n.d.). It has some nice and robust React components to build a well designed application.

Figure 15 – React Redux Flow Diagram



Source: Getting. . . (2016)

4.1.5 Fast Signal Processing

Pitch detection is a heavy problem to solve, and good implementations are time consuming, so it is needed efficiency to run it real-time. The library already said to be used didn't actually existed, the only one available was a pure JavaScript library (PITCHFINDER, n.d.) which is not suitable for this project. The solution was to build an own library based on both the pure JavaScript one and TarsosDSP (SIX; CORNELIS; LEMAN, 2014). Implementation details discussed further on section 4.2.

4.1.6 Real Time Visualization

There are lots of charting libraries available for use with web interfaces (and by extension NW.js), unfortunately none was good fit for real time high density signals such as audio. The solution was again to build one, since all other things are looking to run smoothly in JavaScript, implementation details on section 4.3.

4.2 Pitch Detection

Pitch detection is simply frequency detection with the restriction of note quantization, Table 4 shows the base frequency for each of the 12 existent notes. Multiples of the same frequency are seen as the same note on a different range, known as octave. Even though the quantization makes things simpler it's still a hard task, even more for

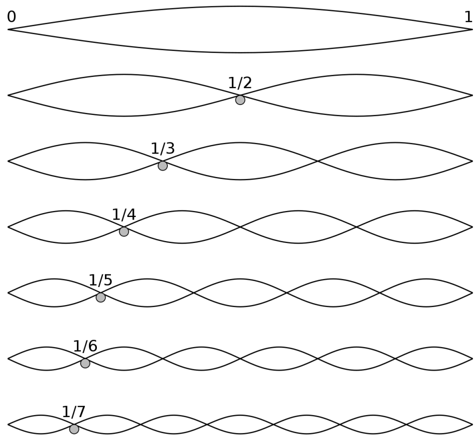
Table 4 – Notes Frequencies

Note Name	Frequency
A	440.00
A#	466.16
B	493.88
C	523.25
C#	554.37
D	587.33
D#	622.25
E	659.25
F	698.46
F#	739.99
G	783.99
G#	830.61

Source: authors

instruments where there is the presence of harmonic series. Harmonic series notes are multiples of the fundamental frequency (most important note) produced by integer sections of the instrument vibration. Figure 16 shows an visual representation of why they exist. The existence of them as well as the presence of both inter-signal and white noise makes necessary the use of non-trivial algorithms for pitch detection, and two of them will be discussed next.

Figure 16 – Harmonic Series



Source: Wikipedia (2017)

4.2.1 YIN algorithm

Autocorrelation is a well know function to calculate a signal's fundamental frequency, but it gives too much error for this project use case. YIN (CHEVEIGNÉ; KAWAHARA, 2002) is a method that uses a few improvements to the autocorrelation method, achieving a much higher precision. It can also be implemented with logarithmic growth as the autocorrelation can be calculated using the fft and ifft algorithms. The algorithm can be divided in 6 steps, as follows:

1. Autocorrelation
2. Difference
3. Cumulative mean normalized difference
4. Absolute threshold
5. Parabolic interpolation
6. Best local estimate

It's important to notice that the absolute threshold is a controlled attempt to regulate the error introduced by the harmonic series (as in Figure 16), it thus gives preference to lower frequencies (below the threshold).

4.2.2 MacLeod algorithm

MacLeod (MCLEOD; WYVILL, 2005) goes for another approach, using the square difference function. More precisely it uses a special normalized version of it. The best result is then calculated by means of using a parabolic interpolation of the highest peak and its two neighbors, this process also gives a threshold constant that limits the detection of the neighbors, thus the possibility of some tuning. As we will see this gave the best results for our project after some tuning.

4.2.3 Implementation

Implementation for both algorithms follow the same pattern, taking the pretty Java code of TarsosDSP (SIX; CORNELIS; LEMAN, 2014) and replacing the syntax and data structures with C++ ones (using standard library for containers). There is also a JavaScript bridge for data type conversion, so we can use the library calls with simple arrays of numbers.
The implementation is not using fft for logarithm growth yet (but quadratic growth instead), following the TarsosDSP library. The faster implementation is kept as a goal for future improvement. All code is available as an open source project at Github and npm (NODE-PITCHFINDER, n.d.).

4.3 Data Visualization

The objective of data visualization is to live debug and tune both hardware gain and algorithms control constants. The ideal case is to have both real time chart as well as a buffered/triggered one, essentially something like an oscilloscope. That has to be performant for real time audio signals, sampled at more than 47 kHz.
There are a great amount of JavaScript DOM libraries for charting, but most of them are too automatic or have too much details, making them too slow for the project need. The solution was to build an own library for that, again available as an open source project at both GitHub and npm (REACT-PLOTTER, n.d.).

4.3.1 Requirements

a) Be a React Component

b) Automatic calculations for array input

c) Option for triggering

d) Option for buffering

e) Minimum Redraw

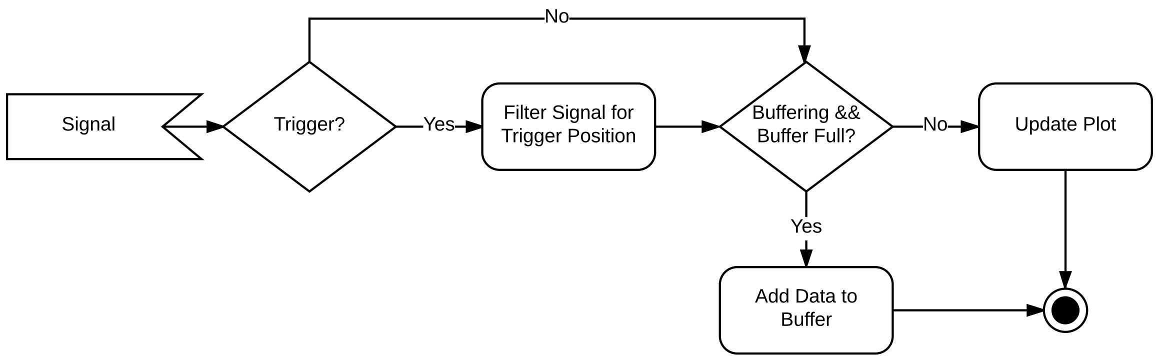
f) Fixed Height/Width

4.3.2 Algorithm

Triggering and buffering are achieved by using a filter that only calls the plotting function when the options are met. This filter is simply the function called to add data, and it is represented by Figure 17.

For the actual drawing a triple buffer technique is used, one for holding the last state

Figure 17 – Add Function Diagram



Source: authors

(called plotBuffer), on for drawing (called drawingBuffer) and finally the one actually rendered (called canvas). That last one is needed so the arrows don't get saved on the drawing scene.

For linear plot time a translation is established, in a way that only the new points will be drawn, the past ones are only translated to the left. The steps of the algorithm are as follow:

1. Clear drawingBuffer
2. Copy plotBuffer to drawingBuffer translating (removing) extra data
3. Draw new data on drawingBuffer
4. Copy drawingBuffer to plotBuffer
5. Copy plotBuffer to canvas
6. Draw arrows on canvas

4.3.3 Implementation

Using the listed requirements (subsection 4.3.1) a minimum API was built as a React component. Being such all it gives is a set of properties, for which the chart is drawn

Table 5 – React Plotter Props

Property	Type	Description
style	Function	Style function (called to print the data)
[trigger]	number	Use trigger
[onlyFull=true]	bool	When using trigger it tells if the view should wait for a complete data set before updating
[width=300]	number	
[height=150]	number	
[initialData=[ ]]	number[ ]	
[appendData=[ ]]	number[ ]	
[dataSize=100]	number	
[pixelSkip=1]	number	Pixels between points
[max=100]	number	Maximum Y Value
[min=-100]	number	Minimum Y Value
[useMean=true]	bool	Use mean calculation, otherwise median

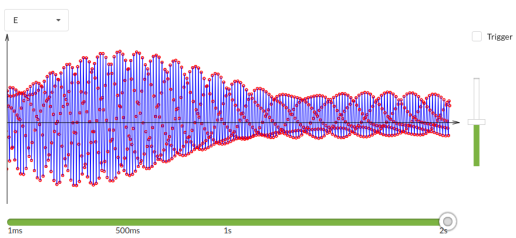
Source: react-plotter (n.d.)

(when needed), they are listed in Table 5. The style property is a function that is called to render each point. Two styles were built, a line plot (points are connected by a straight line) and a digital plot (digital signal standard chart, not used in the final version of this project).

4.3.4 Results

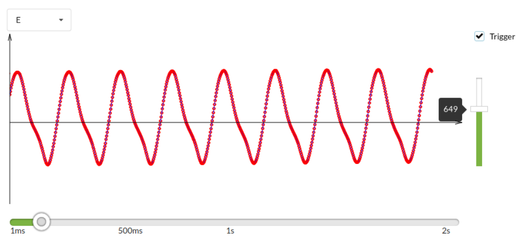
The results are more than satisfactory, tested to be able to run multiple plots of audio speed signals at the same time without much effort. Figure 18 and Figure 19 show how the visualization looks on the project, but full details and working examples are also available at GitHub (REACT-PLOTTER, n.d.).

Figure 18 – Real Time Plot



Source: authors

Figure 19 – Triggered Plot

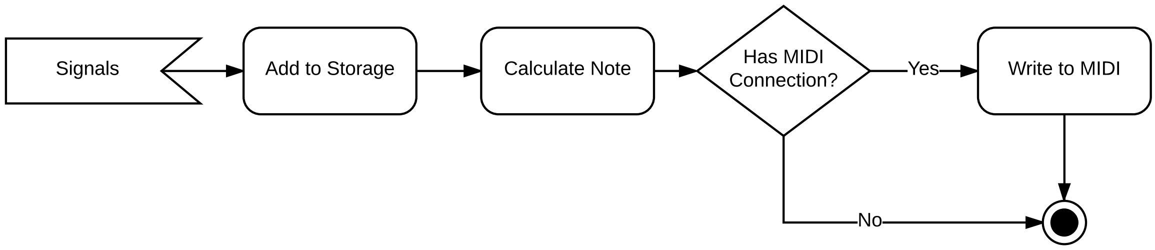


Source: authors

4.4 Main Program

The general idea is to build an UI with three main divisions: Home (with device selection), midi selection and signal-to-MIDI connections, Plot (with the signal visualization) and Options (with the algorithm tuning options, as well as virtual MIDI creation). When a device is selected it's signals will go through a simple process, as in Figure 20.

Figure 20 – Signals Flow Diagram



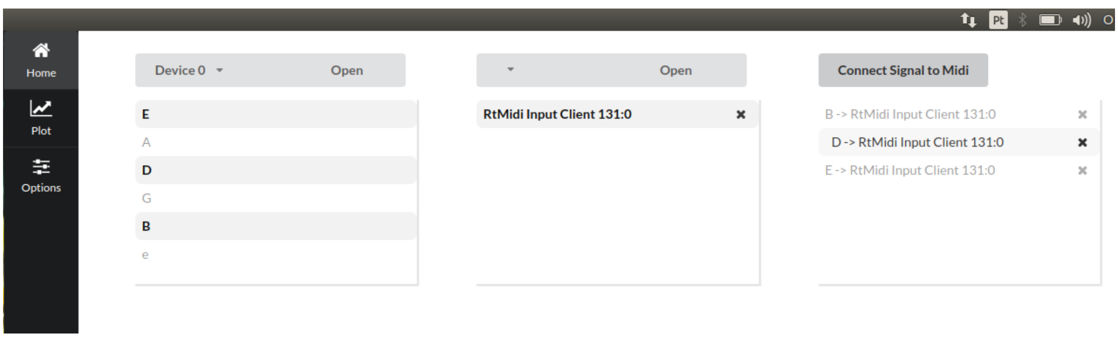
Source: authors

4.4.1 GUI

4.4.1.1 Home

The home page has three horizontally divided sections: Device and signals, MIDI selection and list, and finally connections, as in Figure 21.
The first is used to open the device (there can be only one used at a time). When it is open a list of signals will be displayed (six of them, named as each guitar note). Each signal can be selected so it can be connected to a MIDI device.
The second is to open any given number of existing MIDI devices, which will be listed bellow (can be also closed). The listed devices can also be selected, but only one at a time.
The final section is used to establish the connections, given the selected signals and MIDI, the connections are showed in the box bellow and can be deleted.

Figure 21 – Home Page

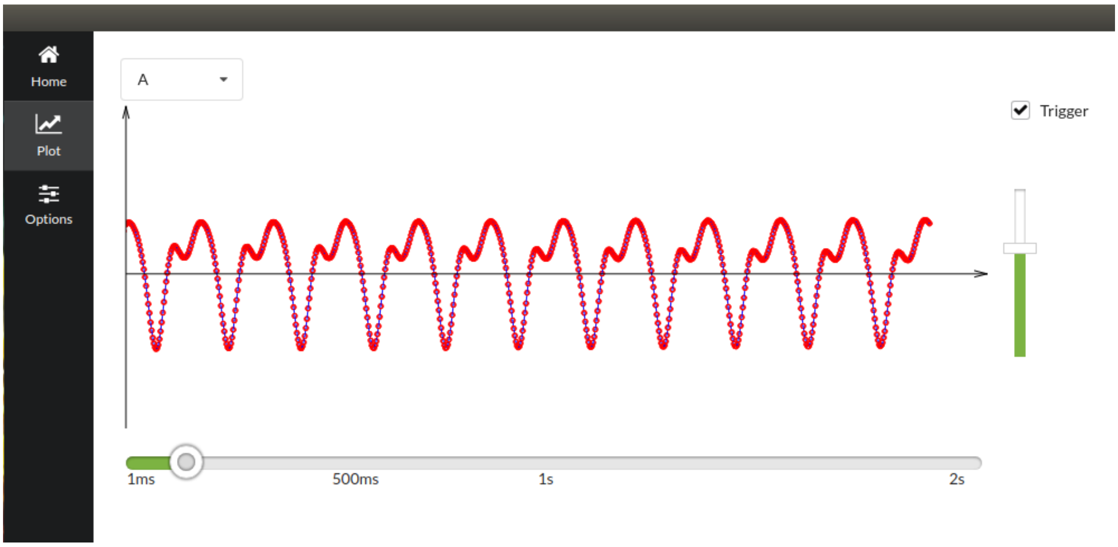


Source: authors

4.4.1.2 Plot

The Plot page is a simply a react-plotter (REACT-PLOTTER, n.d.) component with a few visual controls, being: a dropdown to select which signal is being displayed, a checkbox to enable trigger, a slider to control the time range and another slider to control the trigger value. The full page is as in Figure 22. It was chosen to show only one plot at a time so it's size is bigger, easier to see. This also makes the program a little more performant.

Figure 22 – Plot Page



Source: authors

4.4.1.3 Options Page

The options page is simply an UI to control the used algorithm (from section 4.2) and it's parameters. For both Linux and macOS it's also possible to create virtual MIDI devices, from which our program can write and a synthesizer can read. For Windows this is still possible, but using a hacky solution (since Windows does not give any official API for this) - the easiest option being LoopBe1 (LOOPBE1, n.d.). The page can be seen at Figure 23.

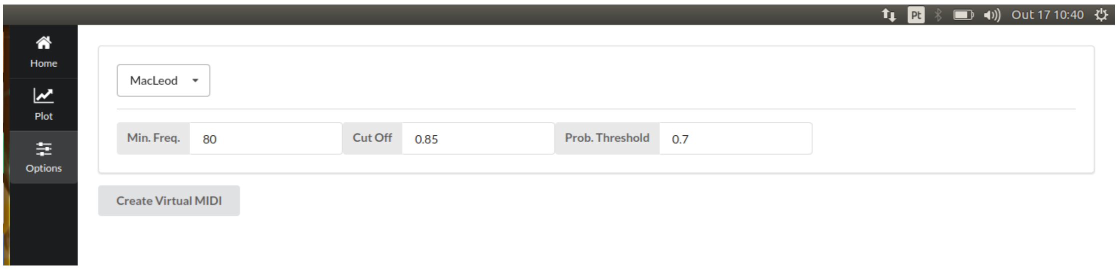
4.4.2 Implementation

4.4.2.1 Resources

At first let's take a look at the resources:

a) USB device: only one can be connected

Figure 23 – Options Page



Source: authors

b) MIDI devices: a list of them can be used at any time

c) Signals: fixed for the only connected device

It's easy to see they are all global, so they can be represented as static classes. But JavaScript has good functional programming capabilities, which are very suitable for global resources. JavaScript imported modules are also scoped, by default, this means it works like a C++ namespace, keeping our static resources separated in a nice way. Taking these mentions in account three modules were built for resource easy access, being MIDI, USB and signal processing.

4.4.2.2 Entry Point

The entry point for this project is both a declarator and connector. It allocates all needed structures (or calls the module that does it), the most significant one being the Redux store (REDUX, n.d.) - store being a short for storage, which is where all of the application visible state is held.
The entry point also connects all callbacks and logic in a declarative way. In this single file all of the program's internal functionalities are declared, so much that if you read it you should also understand the entire program.

4.4.2.3 Reducers

Redux stored data is not defined by a set a properties, like typical OOP applications, instead it uses a more functional programming paradigm. The storage is defined by transfer functions, each of them describing actions that can modify the current state by returning a new one. Each of these functions, called reducer, receive two parameters - the current state and the action to be processed - and should return the new state for the action (or the current one if there are no changes).
Our program has nine reducers, but five of them share the same transfer function, as in Table 6.

Table 6 – Reducers

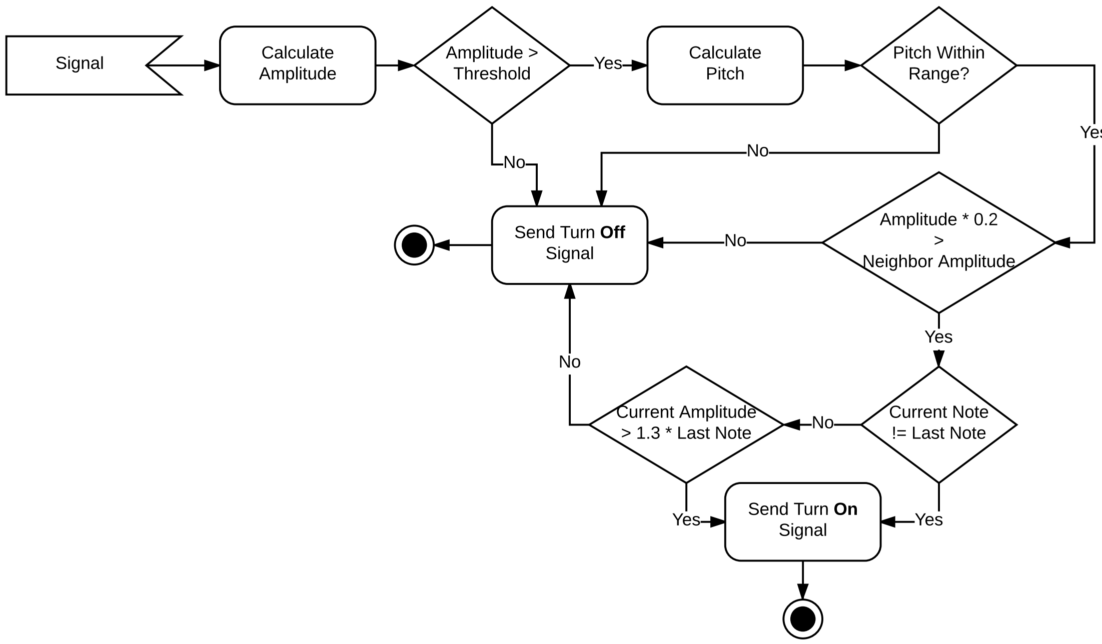
Name	**Data Type**	**Actions**	**Used for**
device	string	set, remove	current selected device
devices	string[ ]	add, remove	list of devices
signals	string[ ]	set, clear	list of signals
signalsData	object[ ]: name: number[ ]	set, clear	list of signals data (name and values)
object	object: any	set, clear, remove	Plot Page options General options MIDI devices Signal to MIDI connections

Source: authors

4.4.2.4 Note Calculation

To calculate the note a few steps are shown in Figure 24. The amplitude calculation is a simple absolute average removing the mean. The pitch within range is a function that limits each string frequency to being close to it's know possible values, as in Table 7.

Figure 24 – Note Detection Diagram



Source: authors

4.4.2.5 Tests

This is for now a prototype of a real world project, so it has a limited amount of tests. The tests fall into one of two categories: unity or timing. Unity tests are made for the signal processing relating modules and also for every reducer, these are all automated and sum up to a total of 31 tests over 9 modules.

Table 7 – Pitch Range

String	**Min. Freq.[Hz]**	**Max. Freq.[Hz]**
E	70	265
A	95	350
D	130	470
G	170	625
B	215	785
e	290	1050

Source: authors

The timing tests were used to check if each separated functionality that may cause processing issues can run in real-time. There are timing tests for: signal average value calculation, signal window buffering, raw data conversion, pitch detection and USB polling.

4.4.2.6 Repository

Again, all code is available at GitHub (TREVISAN, 2017b).

5 Results and Dicussions

5.1 Software Results and Discussions

5.1.1 Results

TODO: measurements and analysis

5.1.2 Future Improvements

5.1.2.1 Note Detection

Real time note detection proved not to be so accurate. Legato (connected) notes may cause a middle note detection and there are miss-detections (mostly at frequency transitions). It works well enough for live play of MIDI instruments, but not for music notation.
For real world music notation a different solution is needed: record the signals and post-process them. This way it is possible to use a more detailed analysis of each signal and thus get very good results, as there is much more computing power available when not being limited by real-time processing.

5.1.2.2 Performance

Though real time analysis work, a few limitations were detected. Buffering can only process each data point one time (can't use a sliding buffer that recycles data). This means that we are close to the processing limits, due to two causes: single core processing and slow algorithms.
One step of the solution is to use multi-core processing for the pitch detection, which can be done using Node.js support for it.
As already stated (subsection 4.2.3) the current implementation uses quadratic growth algorithms, when they can be implemented with logarithmic growth. Fixing this will improve the performance to a point where multi-core processing won't be even needed.

6 Conclusion

Sed consequat tellus et tortor. Ut tempor laoreet quam. Nullam id wisi a libero tristique semper. Nullam nisl massa, rutrum ut, egestas semper, mollis id, leo. Nulla ac massa eu risus blandit mattis. Mauris ut nunc. In hac habitasse platea dictumst. Aliquam eget tortor. Quisque dapibus pede in erat. Nunc enim. In dui nulla, commodo at, consectetuer nec, malesuada nec, elit. Aliquam ornare tellus eu urna. Sed nec metus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

Phasellus id magna. Duis malesuada interdum arcu. Integer metus. Morbi pulvinar pellentesque mi. Suspendisse sed est eu magna molestie egestas. Quisque mi lorem, pulvinar eget, egestas quis, luctus at, ante. Proin auctor vehicula purus. Fusce ac nisl aliquam ante hendrerit pellentesque. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Morbi wisi. Etiam arcu mauris, facilisis sed, eleifend non, nonummy ut, pede. Cras ut lacus tempor metus mollis placerat. Vivamus eu tortor vel metus interdum malesuada.

Sed eleifend, eros sit amet faucibus elementum, urna sapien consectetuer mauris, quis egestas leo justo non risus. Morbi non felis ac libero vulputate fringilla. Mauris libero eros, lacinia non, sodales quis, dapibus porttitor, pede. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Morbi dapibus mauris condimentum nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Etiam sit amet erat. Nulla varius. Etiam tincidunt dui vitae turpis. Donec leo. Morbi vulputate convallis est. Integer aliquet. Pellentesque aliquet sodales urna.

Bibliography

3D printer materials. 2013. Description of the materials used on 3D printers. Available from Internet: <http://www.3dprinterhelp.co.uk/what-materials-do-3d-printers-use>. Quoted in page 16.

AMPLIFIER Circuits. n.d. Simple theory about amplifier circuits. Available from Internet: <http://www.electronics-tutorials.ws/amplifier/amp_1.html>. Quoted in page 17.

CARTER, B. A Single-Supply Op-Amp Circuit Collection. 2000. Available from Internet: <http://electro.uv.es/asignaturas/ea2/archivos/sloa058.pdf>. Accessed: 30 sep. 2017. Quoted in page 19.

CHEVEIGNÉ, A. de; KAWAHARA, H. YIN, a fundamental frequency estimator for speech and music. *The Journal of the Acoustical Society of America*, p. 111, jan. 2002. Available from Internet: <http://audition.ens.fr/adc/pdf/2002_JASA_YIN.pdf>. Accessed: 30 sep. 2017. Quoted in page 32.

COOCOX. n.d. Free/open ARM Cortex-M Development Tool-chain. Available from Internet: <http://www.coocox.org/>. Quoted in page 26.

DIGIKEY. Available from Internet: <https://www.digikey.com/>. Quoted in page 51.

GETTING Started with React, Redux and Immutable: a Test-Driven Tutorial (Part 2). 2016. Theodo. Available from Internet: <https://www.theodo.fr/blog/2016/03/getting-started-with-react-redux-and-immutable-a-test-driven-tutorial-part-2/>. Accessed: 8 oct. 2017. Quoted in page 30.

HEXAPHONIC Pickup. n.d. Available from Internet: <http://www.cycfi.com/projects/six-pack/>. Quoted in page 17.

HOW does Node.js work with NW.js and Electron? 2017. Medium. Available from Internet: <https://medium.com/@paulbjensen/how-does-node-js-work-with-nw-js-and-electron-da9e50e2c3c1>. Accessed: 8 oct. 2017. Quoted 2 times in pages 29 and 30.

HUDAK, Z. *STM32F103C8T6 board, alias Blue Pill*. n.d. Available from Internet: <https://os.mbed.com/users/hudakz/code/STM32F103C8T6_Hello/>. Quoted in page 25.

INA 326, 327 datasheet. 2004. Supplier datasheet for INA 326 and 327. Available from Internet: <http://www.ti.com/lit/ds/symlink/ina326.pdf>. Quoted 2 times in pages 19 and 20.

LIBUSB. n.d. Cross-platform API for generic USB access. Available from Internet: <http://libusb.info/>. Accessed: 8 oct. 2017. Quoted 2 times in pages 27 and 28.

LM741 Operational Amplifier. 2015. Supplier datasheet for LM741. Available from Internet: <http://www.ti.com/lit/ds/symlink/lm741.pdf>. Quoted in page 17.

LOOPBE1. n.d. A Free Virtual MIDI Driver. Available from Internet: <http://www.nerds.de/en/loopbe1.html>. Accessed: 18 oct. 2017. Quoted in page 37.

MAGNETS. n.d. Image of a example of pickup magnets. Available from Internet: <http://tinderwetstudios.com/blog/wp-content/uploads/2015/07/guitar-pickup-magnets.jpg>. Quoted in page 18.

MCLEOD, P.; WYVILL, G. A Smarter Way to Find Pitch. *Proc. International Computer Music Conference*, Barcelona, Spain, p. 138–141, sep. 2005. Available from Internet: <http://miracle.otago.ac.nz/tartini/papers/A_Smarter_Way_to_Find_Pitch.pdf>. Accessed: 30 sep. 2017. Quoted in page 33.

MILMANN, J.; HALKIAS, C. C. *Eletrônica Disositivos e Circuitos*. [S.l.: s.n.], 1981. Volume 2. Quoted in page 19.

MINISCOPE. n.d. A Free Virtual MIDI DriverVery cheap low-speed dual channel PC/USB oscilloscope with STM32 (STM32F103C8T6) microcontroller. Available from Internet: <http://tomeko.net/miniscope_v2c/>. Accessed: 19 oct. 2017. Quoted in page 26.

NAVE, R. *Faraday's Law*. n.d. Theory of the Faraday's Law. Available from Internet: <http://hyperphysics.phy-astr.gsu.edu/hbase/electric/farlaw.html>. Quoted in page 16.

NODE-PITCHFINDER. n.d. A compilation of pitch detection algorithms Node. Available from Internet: <https://github.com/cristovao-trevisan/node-pitchfinder>. Accessed: 8 oct. 2017. Quoted in page 33.

NODE-USB. n.d. Node bindings to libusb. Available from Internet: <https://github.com/tessel/node-usb>. Accessed: 8 oct. 2017. Quoted in page 29.

NODE Webkit. n.d. Available from Internet: <https://nwjs.io/>. Accessed: 8 oct. 2017. Quoted in page 30.

PCB-SHOPPER. Available from Internet: <https://pcbshopper.com/>. Quoted in page 51.

PITCHFINDER. n.d. A compilation of pitch detection algorithms for Javascript. Available from Internet: <https://github.com/peterkhayes/pitchfinder>. Accessed: 8 oct. 2017. Quoted in page 31.

REACT. n.d. A JavaScript library for building user interfaces. Available from Internet: <https://reactjs.org/>. Accessed: 8 oct. 2017. Quoted in page 30.

REACT-PLOTTER. n.d. Real Time (high speed) Plotter Component for React. Available from Internet: <https://github.com/cristovao-trevisan/react-plotter>. Accessed: 16 oct. 2017. Quoted 3 times in pages 33, 35, and 37.

REDUX. n.d. Predictable state container for JavaScript apps. Available from Internet: <http://redux.js.org/>. Accessed: 8 oct. 2017. Quoted 2 times in pages 30 and 38.

SEMANTIC UI React. n.d. The official Semantic-UI-React integration. Available from Internet: <https://react.semantic-ui.com>. Accessed: 9 oct. 2017. Quoted in page 30.

SIX, J.; CORNELIS, O.; LEMAN, M. TarsosDSP, a Real-Time Audio Processing Framework in Java. In: *Proceedings of the 53rd AES Conference (AES 53rd)*. [S.l.: s.n.], 2014. Quoted 3 times in pages 28, 31, and 33.

STM32F103XX Reference Manual. [S.l.], 2015. Available from Internet: <www.st.com/resource/en/reference_manual/cd00171190.pdf>. Accessed: 19 oct. 2017. Quoted in page 25.

TREVISAN, C. *Guitar Digitizer Firmware Codebase*. 2017. Available from Internet: <https://github.com/cristovao-trevisan/guitar-digitizer-firmware>. Quoted in page 27.

TREVISAN, C. *Guitar Digitizer GUI Codebase*. 2017. Available from Internet: <https://github.com/cristovao-trevisan/guitar-digitizer>. Quoted in page 40.

WALLACE, H. *How do guitar pickups work*. 2004. Theory behind the pickup working. Available from Internet: <http://zerocapcable.com/?page_id=219>. Quoted in page 16.

WIKIPEDIA. *Harmonic series (music)*. 2017. Available from Internet: <https://en.wikipedia.org/wiki/Harmonic_series_(music)>. Accessed: 9 oct. 2017. Quoted in page 32.

Appendix

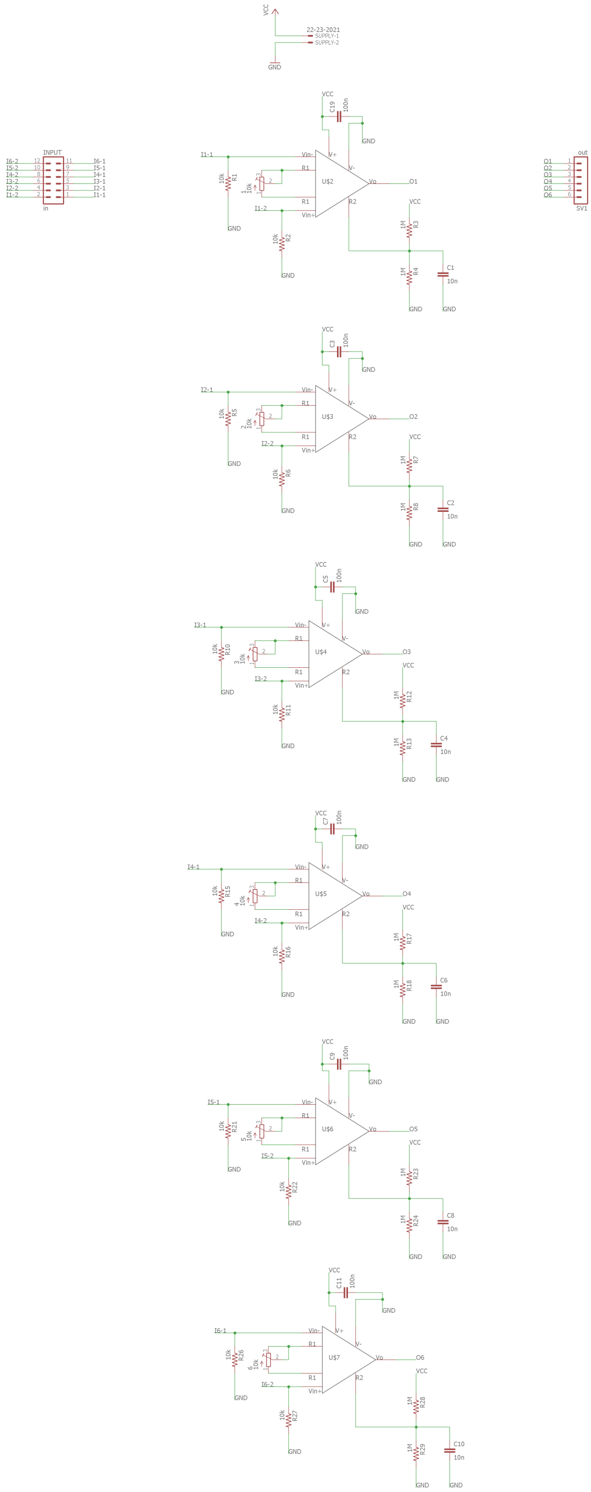
APPENDIX A – Quisque libero justo

Quisque facilisis auctor sapien. Pellentesque gravida hendrerit lectus. Mauris rutrum sodales sapien. Fusce hendrerit sem vel lorem. Integer pellentesque massa vel augue. Integer elit tortor, feugiat quis, sagittis et, ornare non, lacus. Vestibulum posuere pellentesque eros. Quisque venenatis ipsum dictum nulla. Aliquam quis quam non metus eleifend interdum. Nam eget sapien ac mauris malesuada adipiscing. Etiam eleifend neque sed quam. Nulla facilisi. Proin a ligula. Sed id dui eu nibh egestas tincidunt. Suspendisse arcu.

Annex

ANNEX A – INA 326 complete Schematic.

Figure 25 – INA 326 Complete Schematic



Source: authors

B Prototype Cost Table.

Costs are given on Brazilian Real.

Table 8 – Prototype Board Cost Table

Component	Quantity	Cost
INA326	6	120.00
10kΩ Trimmer Potentiometer	6	7.50
10nF Ceramic Capacitor	6	1.00
100nF Electrolytic Capacitor	6	1.50
10kΩ Resistor 5% tolerancy	12	2.00
1MΩ Resistor 5% tolerancy	12	2.00
Pin bar 1x40	1	3.00
PCB	2	50.00

Source: authors

C Production Cost Table.

Costs are given on United States Dollar for production of 1000 samples.

Table 9 – Production Board Prediction Cost Table

Component	Quantity	Cost per Component	Total Cost
INA326	6000	2.53071	15184.26
10kΩ Trimmer Potentiometer	6000	2.16050	12963.00
10nF Ceramic Capacitor	6000	0.00233	13.98
100nF Electrolytic Capacitor	6000	0.04431	265.86
10kΩ Resistor 1% tolerancy	12000	0.00649	77.88
1MΩ Resistor 1% tolerancy	12000	0.00671	80.52
Pin bar 1x40	1000	0.62240	622.40
PCB	1000	0.22	223.94

Source: Digikey () and PCB-shopper ()