Digital Guitar Effects Box

Jordan Spillman, Electrical Engineering

Project Advisor: Dr. Tony Richardson

April 24th, 2018
Evansville, Indiana
Acknowledgements

I would like to thank Dr. Richardson for advice over this project. I would also like to thank Jeff Cron with advice for finding parts and 3D printing parts.
## Table of Contents

I. Introduction  
II. Problem Definition  
III. Design Approach  
   A. Hardware Design  
   B. Software Design  
   C. Constraints  
   D. Costs  
IV. Results and Conclusion  
V. Appendix A – Software Code
List of Figures

1. ADC Pin out
2. USB Audio Soundcard
3. Hardware Block Diagram
4. Hardware Schematic
5. Pure Data Variable Setup
6. Python Variable Setup
7. ADC Setup in Python
8. LCD Setup in Python
9. Distortion Block Diagram
10. Pure Data Distortion Setup
11. Reverb Block Diagram
12. Pure Data Reverb Setup

List of Tables

1. Project Cost
Introduction

Many different digital guitar effect boxes on the market lack the ability to be cost effective, and lack to give the user the ability to communicate wirelessly. These both are needed because quality guitars cost a large amount of money and so the sound has to be generated in hardware. Also, guitarists need to understand how guitar effects are useful and can play a big role in the sound they want to achieve. Many times, novice guitarists are shying away from trying different effects because a single analog effect can cost one hundred dollars. While digital multi effect boards can run as high as three hundred, even to a thousand dollars. These digital guitar effect boards also have many different options that many novice players will not know how to master. Guitarists also need the means to be able to communicate with their equipment wirelessly. This gives guitarists the ability to communicate with their equipment wherever they are on or off stage, or have the audio engineer change it for them, so the guitarist can focus on the notes he is playing.

Problem Definition

The goal of this project is to create a digital guitar effect box that a standard guitar can plug into and play. It must be simple for any novice player to be able to learn and master. The digital effect box will have the distortion, reverb, and clean effects. The microcontroller will need to be able to read in the guitar sample, produce the proper effect to the sample, and output it via digital-to-analog converter. The microcontroller must be able to process the system in a timely manner and to where no latency issues are noticed by the user. The quality of each effect will be judged by the student and advisor to be counted as worthy.
The Guitar Effect Box must:

- be able to apply a digital distortion guitar effect to the inputted signal
- have user input control over the parameters of the effect
- read in sound samples at 44.1kHz (CD Quality)
- use 12-bit data samples
- pass the output out to a quarter inch jack
- be powered by a wall outlet

The digital guitar effect box will not include a guitar, nor an amplifier or speaker; these must be provided by the user.

**Design Approach**

**A. Hardware Design**

The hardware of the guitar effects box are the microcontroller, the ADC converter, the USB audio soundcard, the liquid crystal display (LCD), a quarter inch mono to eighth inch stereo cable, and the power supply. The microcontroller used is the Raspberry Pi 3 [1]. The microprocessor has a 1.2 GHz clock and uses the Arm Cortex 4. The microcontroller has 1GB of Ram, and can run its own operating system. This will help to be able to run the software Pure Data and a separate python script at the same time. The Raspberry Pi 3 has an external storage ravia a microSD. Some guitar effects, such as reverb, require many samples to be noticed. For example, to execute half a second of reverb, one sample will have to be repeated for half a second at 44.1 kHz. This means the sample will be stored at a maximum of 22,050 times. With this microprocessors memory being supplied by the SD card, sampling the signal will be possible
while being able to execute other code. A bonus is the size of the Raspberry Pi, it is only the size of a credit card making it very versatile [1].

Since the Raspberry Pi 3 offers no on-board analog to digital converters(ADC), the MCP3008 ADC was selected to be used from Adafruit [3]. This ADC can input 8 analog signals and communicates to the Raspberry Pi 3 through a serial peripheral interface bus(SPI).

![Pinout of the MCP3008 ADC from Adafruit.](image)

**Figure 1**: Pinout of the MCP3008 ADC from Adafruit.

This application uses channels 1, 2, and 3 of the adc – the rest will not be connected. A block diagram for the connection to the Raspberry Pi can be found in figure 3.

In order to receive the audio signal from the guitar and to output the sound effect to the amp, a USB audio soundcard was used. This soundcard is from “plugable”. This particular soundcard has input and output audio jacks and uses both in stereo. The soundcard works in the Raspberry Pi 3 operating system and on Pure Data software[2].
The box runs power through the micro-USB adapter on the Raspberry Pi 3. The MCP3008 can be powered from the boards 5 or 3 volt sources. The USB soundcard will be powered by the USB port. Any push buttons, or potentiometers will be connected to the boards 3 volts power supply and then connected to a selected port. A block diagram of the hardware is given in figure 3:

![Block diagram of the hardware for the digital guitar box.](image)

Figure 3: Block diagram of the hardware for the digital guitar box.

The schematic for this projects is shown in figure 4:
Figure 4: Schematic of the hardware for the digital guitar box.

B. Software Design

The software must be able to read the inputted guitar signal, apply the desired guitar effect, and output the corresponding signal within 44.1 kHz or 22.68 μsec. Reading in the signal, outputting the signal, and applying the effect to the signal will be done using the software called Pure Data. Pure Data is a visual programming language that can be used for audio or any multimedia. It is able to be manipulated using C or Python, and samples at CD quality which is 44.1 kHz[4].
There will be python code that allows the Raspberry Pi 3 to run “headless”. This means the Raspberry Pi 3 does not need to be connected to a monitor and keyboard. The python code will setup all the General Purpose Input and Output Pins (GPIO Pins). It will also setup the clock for the ADCs, and select which Pure Data file to open based on the inputs of the ADC.

For this to be done, a way to communicate the values to and from the Pure Data software had to be established. This was done by setting up port objects in pure data that can be accessed by sending it over the port in the python script. An example of the pure data objects and the code to send data is shown in figure 5 and 6:

![Pure Data setup for receiving variables](image.png)

**Figure 5:** Pure Data setup for receiving variables
import os

# (Your own Python script that does whatever you need)

def send2Pd(message=' '):
    # Send a message to Pd
    os.system("echo "+ message + " | pdsend 3000")

def audioOn():
    message = '0 1;' # Id=0 (DSP), message=1 (turn it on)
    send2Pd(message)

def setVolume():
    vol = 80 # Set volume value (0-100)
    message = '1 ' + str(vol) + ';'; # make a string for use with pdsend
    send2Pd(message)

---

**Figure 6:** Python setup for sending variables

The net receive function calls out the port number. The route function tells it there is only one route since pure data can handle more than one message on a given port. Then the message sent is stored in volume. It is then called in the effect by using r volume and the number is then stored as a variable. This is also how each effect was chosen to run by switching each effect on or off by multiplying it by 0 or a 1. The effects all run to the output in parallel so there is no overlapping unless the user would like it done.

The Pure Data file is opened by using the popen command. The popen command is called by running the subprocess and using nohup. Nohup allows the command to be released meaning it is not waiting for a return value and can continue through the code.

The raspberry pi was set up to run headless by using the Raspian’s lxde/autostart text file. The file is executed and the commands listed in that file will be done when the screen reaches the desktop loader. This allows the program to run in the desktop environment which lets the Pure Data Gui run without any problems.
The library for the mcp3008 was downloaded and used for the analog to digital conversions. It required that SPI be set up on a Raspberry Pi which there are many manuals online to do so. A sample code of how the ADC was set up is shown in Figure 7:

```python
import time

# Import SPI library (for hardware SPI) and MCP3008 library.
import Adafruit_GPIO.SPI as SPI
import Adafruit_MCP3008

# Software SPI configuration:
CLK = 18
MISO = 23
MOSI = 24
CS = 25

cmp = Adafruit_MCP3008.MCP3008(clk=CLK, cs=CS, miso=MISO, mosi=MOSI)
```

**Figure 7**: ADC Setup in Python

The library for the Adafruit 16x2 LCD was downloaded to control the LCD. A sample code is shown in figure 8:

```python
#LCD Configuration:
olc_rs = 28
lcd_en = 24
lcd_d4 = 23
lcd_d5 = 17
lcd_d6 = 18
lcd_d7 = 22
lcd_backLight = 3
lcd_columns = 16
lcd_rows = 2

lcd = LCD.Adafruit_CharLCD(lcd_rs, lcd_en, lcd_d4, lcd_d5, lcd_d6, lcd_d7, lcd_columns, lcd_rows, lcd_backlight)
```

**Figure 8**: LCD Setup in Python

The effects are distortion, clean, and reverb. Distortion gives the guitar a more “edgy, rock and roll” sound. It is achieved digitally by “hard clipping” the signal. This is done by setting a threshold on the upper and lower values of the sine wave. The user will be able to set these thresholds by the tuning knobs located on the digital guitar effect box. A block diagram is given below in figure 9[2]:

---

[2]: #Refer to figure 9

Distortion was implemented in Pure Data by first sending the signal through a high pass filter with the cut off frequency set at 80 hz. Then the signal was then multiplied and clipped by using the tanh object in Pure Data. This makes the signal clipped “smooth” instead of a “hard clip” which made it sound better. It was then multiplied by the result of an envelope filter to get the original volume level back. It then passes through a multiplier that the user sets so he or she can make the guitar effect louder or softer. Figure 10 shows the Pure Data file for distortion.
The next digital effect is reverb. Reverb is different from the previous two guitar effects because it does not change the current sample of the signal. Reverb is achieved by playing previous samples on top of the current sample. The previous samples are distorted by distorting the sample to a lower level. This guitar effect box will distort the samples exponentially, this will give the it a “fuller” or more broad sound. The user will be able to change the reverb’s execution time, from one tenth of a second to half a second, and the intensity of the distortion done to previous samples. A block diagram is given below in figure 11[2]:

**Figure 11**: Block Diagram of the Reverb Effect

Reverb was implemented in Pure Data by using what is known as the freeverb object. Freeverb uses multiple lowpass comb feedback (LBCF) and allpass filters to delay and make the signal softer. It was implemented using C++ and is manipulated by changing the roomsize, damping, dry, and wet coefficients connected to the object. In this project, dry and wet were both set at .5 which made the signal half of the original signal, and the other half the freeverb effect. The user can change the damping coefficient and the roomsize coefficient. The damping coefficient
changes the feedback gain from 0 to 1, and the roomsize coefficient changes the delay time.

Figure 12 shows the Pure Data file for reverb.

![Figure 12: Reverb effect in Pure Data](image)

The final effect is clean. Clean is done by passing the signal straight through with having no manipulation done to the signal.

For this project, Pure Data was downloaded as well as the PDextended externals for PDsend, PDreceive, tanh, and freeverb. This project was backed up using github.

C. Constraints

One of the digital guitar effects box constraint’s is environmental. This is addressed on the microprocessor by having the ability to turn off functions that will not be in use, thus saving
power to the system. There is no safety concerns with the box shorting out or catching on fire.

Another constraint is manufacturability. This product would be able to be reproduced. The manufacture could use the microprocessor and audio codecs instead of buying the whole microprocessor and USB soundcard. Also, the manufacture would want a 3d printed box for a much better container for the guitar effect box.

The final constraint will be sustainability. For prototype purposes, the wires will be connected through jumper wires. The sustainability in a manufacturing sense would increase if they would use a PCB circuit board. This would allow all push buttons and dials to be soldered instead of held together by jumper wires. It would also be in great interest to use high quality knobs and push buttons, that way they can handle more rough conditions. The weight of the user will also have an impact on the life of the product and packaging material due to the user pushing on the buttons for selection of presets.

The IEEE standard of ISO/ICE/IEEE 24748-5-2017 was considered by using the Raspian operating system on the Raspberry Pi. This standard is met by the team who designed Raspian therefore meets the standard of IEEE[6].

\textit{D. Costs}

A table of costs is given in Table 1. The project box was chosen over 3D printing to ensure that strong material would be use due to the user putting body weight on the product.

\textbf{Table 1:} Needed components and estimated costs. Miscellaneous items were found from the university stalk room.
Results and Conclusions

The project meets all the client’s minimum requirements. Many expansions of this project are possible, especially with the ability for the project being open source. One would be to add presets that lets the user be able to switch to a particular values and effects by hitting a push button. With the raspian system, it would be easy to save a text file with the values and effect selected. Another application would be to control the box wirelessly from a users phone or other mobile device. This could be done with either Bluetooth or a Wi-Fi connection. With the Raspberry Pi’s ability to have Bluetooth connection and Wi-Fi, it would be easy to implement this. A user can also choose to expand on effects already made or create more effects to choose from. Occasionally the software gets stuck in a loop and presents bad feedback, to overcome this the user needs to unplug and plug back in the power. Further expansion on finding why this occurs should be sought out. Figure 13 shows the final picture of the product.

<table>
<thead>
<tr>
<th>Product</th>
<th>Base Cost</th>
<th>Number of Products</th>
<th>Total Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raspberry pi 3</td>
<td>$35.25</td>
<td>1</td>
<td>$35.25</td>
</tr>
<tr>
<td>USB Audio Soundcard</td>
<td>$7.79</td>
<td>1</td>
<td>$7.79</td>
</tr>
<tr>
<td>1/8 inch to 1/4 inch cable (mono)</td>
<td>$5.59</td>
<td>2</td>
<td>$11.08</td>
</tr>
<tr>
<td>Misc. (push buttons, potentiometers, resistors)</td>
<td>N.A.</td>
<td>N.A.</td>
<td>$5.00</td>
</tr>
<tr>
<td>3D printed cost</td>
<td>$20.00 / 800 cm³</td>
<td>244 cm³</td>
<td>$56.10</td>
</tr>
<tr>
<td>Total Cost</td>
<td></td>
<td></td>
<td>$65.22</td>
</tr>
</tbody>
</table>
Figure 13: Final product

Appendix A

Here is the python code for this project:
#!/usr/bin/env python
import os
import time
import Adafruit_GPIO.SPI as SPI
import Adafruit_MCP3008
import Adafruit_CharLCD as LCD
import subprocess
import socket

# LCD Configuration:
lcd_rs = 25
lcd_en = 24
lcd_d4 = 23
lcd_d5 = 17
lcd_d6 = 18
lcd_d7 = 22
lcd_backlight = 2
lcd_columns = 16
lcd_rows = 2
lcd = LCD.Adafruit_CharLCD(lcd_rs, lcd_en, lcd_d4, lcd_d5, lcd_d6, lcd_d7, lcd_columns, lcd_rows, lcd_backlight)

# Setting up ADC pin configuration:
CLR = 5
MISO = 6
MOSI = 19
CS = 18
 sop = Adafruit_MCP3008.MCP3008(clk=CLR, cs=CS, mosi=MOSI, miso=MISO)

# User definitions

def send2f1(message=' '):
    os.system(\"echo \"{}\" | psend 3500\"\"). # Setup port for communicating

def send2f2(message=' '):
    os.system(\"echo \"{}\" | psend 3400\"\"). # Setup port for communicating

def send2f3(message=' '):
    os.system(\"echo \"{}\" | psend 3300\"\"). # Setup port for communicating

def send2f4(message=' '):
    os.system(\"echo \"{}\" | psend 3200\"\"). # Setup port for communicating

def send2f5(message=' '):
    os.system(\"echo \"{}\" | psend 3100\"\"). # Setup port for communicating

def audiooff():
    message = '0 00
send2f2(message)

def audion():
    message = '0 01
send2f2(message)

def setmix1(mix1):
    message = '0 ' + str(mix1) + ';
send2f3(message)

def setmix2(mix2):
    message = '1 ' + str(mix2) + ';
send2f3(message)
```python
def settrim(mix1):
    message = '0 ' + str(mix1) + ';'
    send2Pd6(message)

def setdamp(mix2):
    message = '1 ' + str(mix2) + ';'
    send2Pd3(message)

def setNet():
    message = '0 0.5;'  # Setup message to send to port
    send2Pd4(message)

def setDry():
    message = '1 0.5;'  # Setup message to send to port
    send2Pd4(message)

def Clean(x):
    message = '0 ' + str(x) + ';'
    send2Pd1(message)

def Distortion(x):
    message = '1 ' + str(x) + ';'
    send2Pd1(message)

def Reverb(x):
    message = '2 ' + str(x) + ';'
    send2Pd1(message)

def getmix1_distortion(x):
    value = (x/1023)*100  # Return value between 0 and 100
    return value

def getmix2_distortion(x):
    value = (x/1023)*1
    return value

def getmix_reverb(x):
    value = (x/1023)*1  # Return value between 0 and 1
    return value

Old_Filename = '0'  # Set old file to 0
lcd.clear()  # Clear LCD
subprocess.Popen(['"/home/pi/final_pd.pd"',  # Open puredata file
stdout=open('/dev/null', 'w'),
stderr=open('/dev/null', 'w'),
preexec_fn=os.setpgrp])

while True:
    values = [0]*3  # Setup array for values
    for i in range(3):
        # The read_adc function will get the value of the specified channel (0-7).
        values[1] = ioctl.read_adc(i)
        Select = values[2]  # Get selector knob value
        Mix1_ADC = values[1]  # Get Mix1 value
        Mix2_ADC = values[0]  # Get Mix2 Value
    if(Select < 342):
        Filename = 'Clean'  # For Distorting LCD
        Mixname = '\nNon-edit'
        Reverb(0)  # Turn off reverb effect
        Distortion(0)  # Turn off distortion effect
```

The code provided is a Python script that sets various parameters and reads values from a device using the `ioctl.read_adc` function. It includes definitions for setting trim, damp, network, and dry effects, as well as functions for getting mix 1 and mix 2 distortion values. The script also contains code for clearing the LCD, setting up a puredata file, and a main loop that reads values and applies effects based on user input.
Clean(1)     #Turn on clean effect
e1f(Selector > 341 and Selector < 683):
    Filename = 'Distortion'  #For Distplaying LCD
    Mixname = '\nHarshness, Vol'
    getMix1(getmix1_distortion(float(Mix1_ADC))) #Returns value between 0 and 100
    getMix2(getmix2_distortion(float(Mix2_ADC))) #Returns value between 0 and 1
    getMix3()
    Clean(0)    #Turn off reverb effect
    Distortion(1)
else:
    Filename = 'Reverb'  #For Distplaying LCD
    Mixname = '\nLength, Damping'
    getmix_reverb(float(Mix1_ADC))) #Returns value between 0 and 1
    getdamp(getmix_reverb(float(Mix2_ADC))) #Returns value between 0 and 1
    setWet() #Set wet to .5
    setDry() #Set dry to .5
    Clean(0)    #Turn off clean effect
    Distortion(0)  #Turn off distortion effect
    Reverb(1)
if(Old_Filename != Filename):  #Checks if running same effect
    mix.clear()  #Clear LCD and change to new effect if not same effect
    mix.message(Filename)
    mix.message(Mixname)
AudioOn()  #Turn on the audio
Old_Filename = Filename     #Set old file to current file name