TuneBox

Final Documentation

Beau Bloomfield, Charles Filipiak, Austin Hickman, Jake Reilly

Table of Contents

Introduction

The electric guitar is one of the most popular instruments in the world and has been an iconic element of rock music for generations. This instrument is often characterized by its ability to create numerous distinct sounds through the integration of amplifiers and signal distortion. However, for new guitar players, this key element comes at a price. Often what is required to manipulate the sound of an electric guitar is either an expensive amplifier or a complicated series of pedals that allow the user to switch between different effects using their feet. These pedals are usually designed with a more advanced user in mind: containing a myriad of knobs and switches to finely manipulate the tone. A new guitar player can spend hours toying with a typical pedal, which offers at best a couple 7-segment LED's for display. This is where the TuneBox offers a solution.

Figure 1: Left is a typical guitar pedal interface
Right is the TuneBox interface

TuneBox replaces this foot pedal system with the familiar user interface of a mobile application. In this interactive environment, clearly labeled effects can easily be switched on and off with the touch of a button. This allows a beginning guitar player the freedom to try different sounds without having to read a new instruction manual for each device. Finally, integrating the product with an app allows for flexibility in continually developing new effects modules to be added on after the initial deployment.

The TuneBox is an integrated microprocessor and amplifier device that digitally records the output from an electric instrument. To do this, an input signal from a guitar is run through a variety of different onboard amplifiers to prepare the signal for analog to digital conversion. The analog waveform is then converted to digital bits, via an audio codec, that can be interpreted by the microprocessor. The microprocessor receives the data and performs two functions: digital effects and saving to memory.

The microprocessor takes the digital signal, and via programmed functions, applies audio effects to it. With WiFi functionality through the ESP-8266 and an associated mobile application, the effects played through the TuneBox are selectable and controllable via smartphone. After an effect is selected within the app, the information is transmitted to the PIC32 which applies the digital effects before sending the modified data back to the digital to analog converter. The DAC turns the 10-bit values into a series of step functions that are smoothed using a demodulation circuit. This signal is finally passed through a series of amplifiers, which allows the analog waveform to be output to a speaker system.

The TuneBox also takes the inputted digital data and saves it to memory. Once saved in flash memory, the TuneBox sends the 10-bit values through the ESP-8266 and onto the Android device. Here, the bits are encoded with more information and saved as a .WAV file. Finally, a sharing function can be used to send the .WAV file to other devices or servers.

System Requirements

- Receive an audio signal from 20 Hz to 2000 Hz at a voltage level ranging between 100 mV and 1 V
- Convert this signal into a digital bitstream via the audio codec so that the PIC32 microcontroller can apply audio effects
- Additionally, the microcontroller must be able to store a bitstream ~45 minutes long at most, consisting of ~2 Gbits at most, into attached flash memory

- Once this bitstream has been saved into memory, it must then be able to be broadcasted to a WiFi client socket, in this case: a mobile application
- The mobile application must be able to send command signals via WiFi which the microcontroller will respond to. These signals will determine which effects are applied, whether the bitstream is sent to the phone, and will also communicate information about the state of the microcontroller
- Once all the signal processing is complete, the microcontroller must also be able to pass the digital signal through a digital-to-analog converter for playback through a standard guitar amp

Project Description

Theory of Operation

The TuneBox is designed to provide an easy and intuitive method for the application of guitar effects. The mobile app can be downloaded to a smartphone, and connected to the hardware via a WiFi connection that is produced on board the TuneBox. The TuneBox itself requires the user to connect a quarter inch male-to-male cable from the guitar output to the TuneBox input, and another cable from the TuneBox output to a speaker. Once the cable connections are made and the TuneBox is connected to the mobile app, the user is able to select a desired effects in the app, and the applied effect will be heard from the speaker.

System Block Diagram

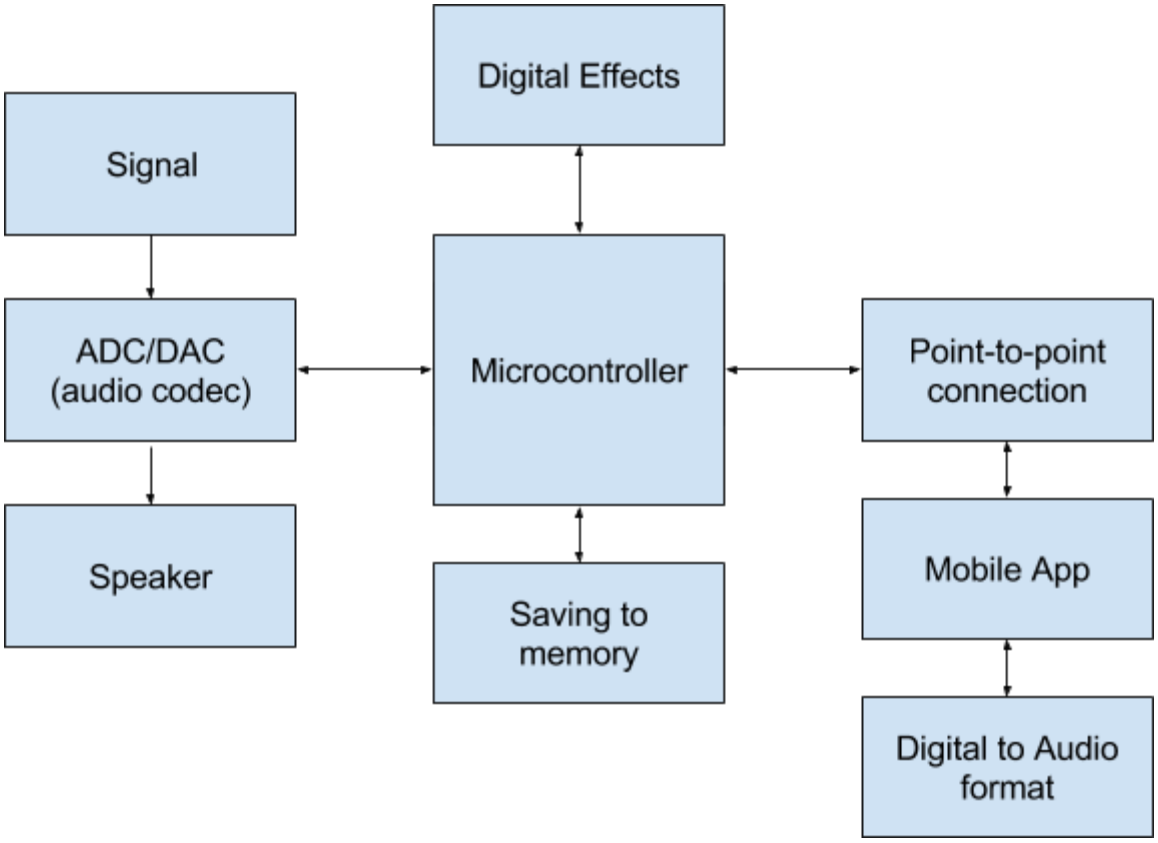


Figure 2: System Block Diagram for the TuneBox

Subsystems

Audio Signal Processing

- Input: digital samples received from the audio codec
- Simultaneously read samples and write to memory chip
    - SPI clock for flash: 10 MHz
- Extract stored waveform from memory
    - Same timing requirements for sampling/recording
- Receive analog audio signal ranging from 100 mV to 1 V from electric instrument

- Convert input analog signal to digital samples
- Receive modified digital samples from the PIC 32
- Convert digital samples to analog signal
- Send analog signal to speaker system at a voltage not exceeding 1.0 V
- Perform signal scaling and DC offset

Memory

Storage is provided by a 1Gbit Spansion S70FL01GS flash memory chip. Each chip includes two memory arrays, and features an SPI interface capable of transferring up to 50Mbps. Memory is available in pages of 512 bytes. Once the page buffer is full, a write cycle is initiated, which typically lasts no longer than 2ms. The memory array is unavailable during a write cycle, so the page buffer for the other array is filled while the first is programming. The microcontroller switches between the two arrays in this fashion, while a counter in the source code keeps track of pages written, which is necessary to access memory during playback. During playback, the opposite process is employed, in which the microcontroller alternates reading memory between arrays. The source code for this operation is given in `verifyMemory.X`. With 16-bit resolution and a sample rate of 44.1kHz, 1Gbit of memory can store over 23 minutes of music. At a sampling rate of 10kHz (which is more than enough for high quality audio in our application, representing an oversampling rate of >10x) more than 1.5 hours of music could be stored. The memory module was implemented and successfully demonstrated, but was not included in the final demonstration because of time constraints.

Signal Description

A system analysis of the audio front end, back end, and demodulation is included below. The signals (as denoted in figures 7-10) are described by the relationships given below, with the following assumptions:

- $x_0(t)$ represents the input signal from the guitar.
- T_s represents the sampling period. During demonstration, an approximate value of T_s = 0.1ms was used.
- $u(t)$ is the unit step function.
- τ is a value between 0 and T_s. The ratio τ/T_s is the duty cycle of the square wave onto which samples are modulated. A value of τ = 0.95 T_s was used for demonstration.
- $h_0[n]$ represents the combination of all digital effects applied by the microcontroller, which are discussed in greater detail in the Digital Effects section.
- The "✻" operator represents convolution in the t or n domain.
- $-V_{DD}$ is provided by a LMC7660 voltage inverter and two 10uF electrolytic capacitors.
- All op-amps are JRC4580D, and operate between +/-3.3V.

The following equations and figures 3-6 describe the complete signal pathway, from the input jack to the output jack. All schematics below were generated using TI WEBENCH and Digikey Scheme-It software.

$$x_1(t) = 4x_0(t)$$

$$x_2(t) = \tfrac{1}{2}V_{DD} + \tfrac{1}{2}x_1(t)$$

$$x_2[n] = x_2(n\,T_s)$$

$$y_0[n] = x_2[n] * h_0[n]$$

$$y_1(t) = \sum_{k=0}^{\infty} y_0[k]\,(u(t - k\,T_s) - u(t - k\,T_s - \tau))$$

$$y_2(t) = y_1(t) * h_1(t)$$

$$y_3(t) = y_2(t)$$

$$H_2(j\omega) = -\left(\frac{j\omega}{R_6}\right)\left(\frac{R_6 R_7 C_1}{1 + j\omega}\right)$$

$$Y_4(j\omega) = Y_3(j\omega)\,H_2(j\omega)$$

$$Y_5(j\omega) = Y_4(j\omega)\,H_3(j\omega)$$

$$y_5(t) = \frac{1}{2\pi}\int_{-\infty}^{+\infty} Y_5(j\omega)\,e^{j\omega t}\,d\omega$$

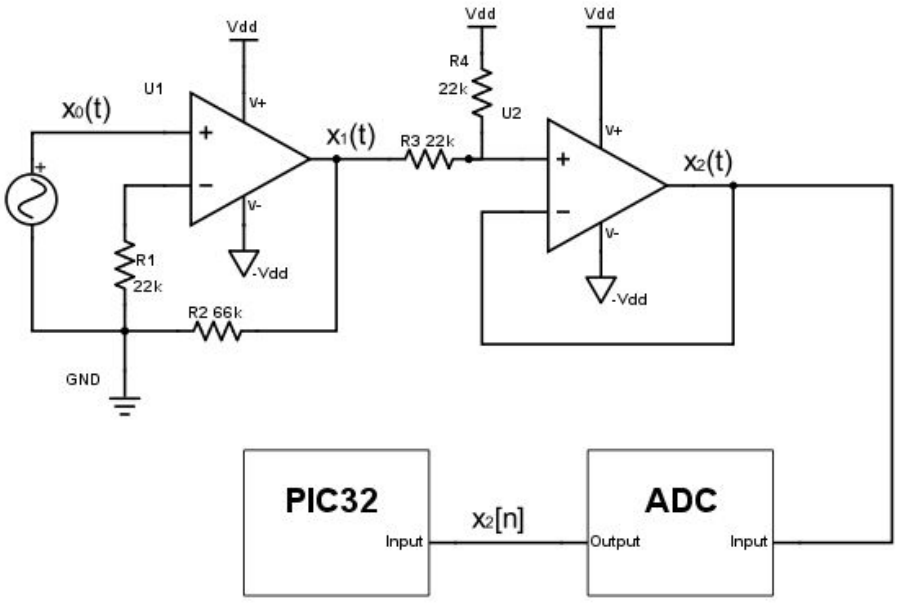


Figure 3: Audio front-end system

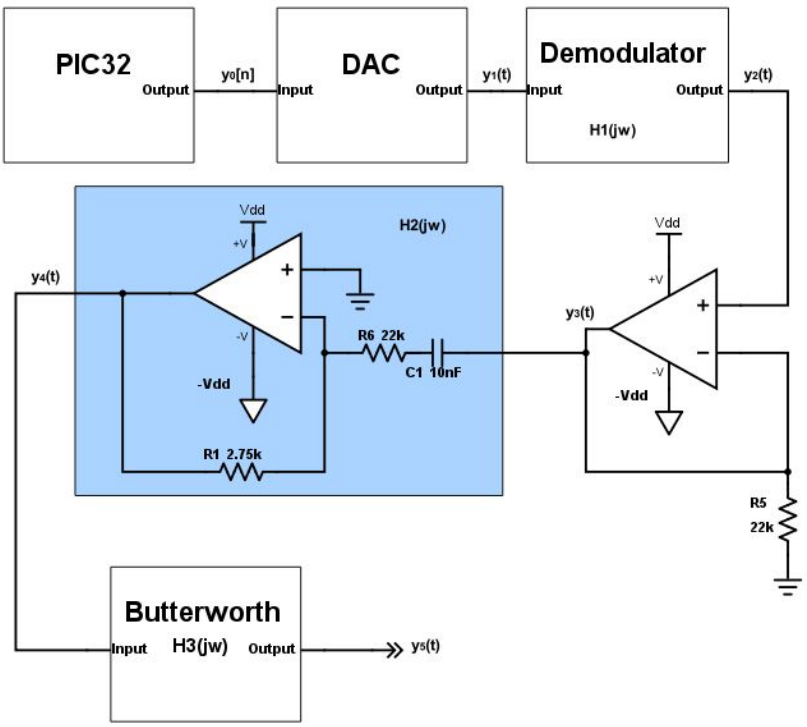


Figure 4: Audio back-end system

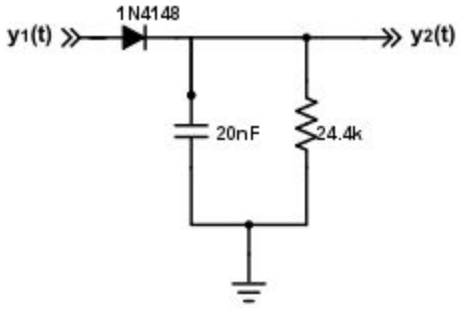


Figure 5: AM demodulator. See the section on $H_1(j\omega)$ for a more detailed description of its behavior.

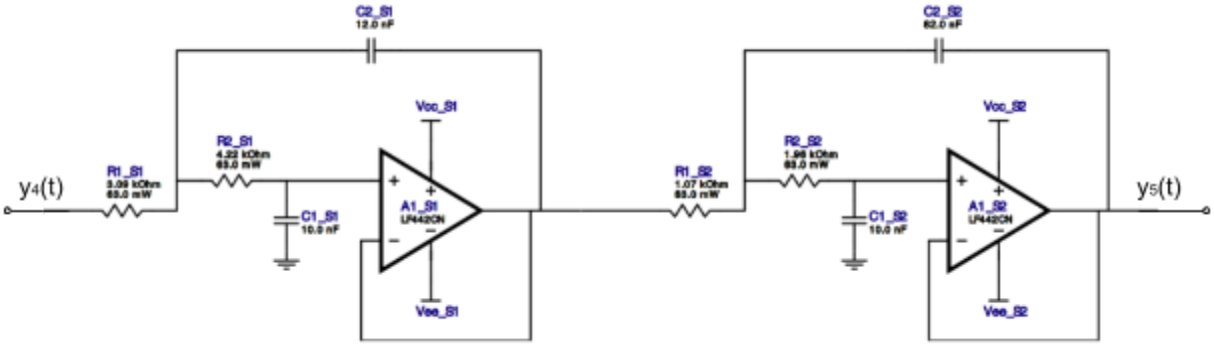


Figure 6: A fourth-order low-pass Butterworth filter which forms $H_3(j\omega)$. See below for a more detailed description of its behavior.

The first back-end hardware filter $H_1(j\omega)$ forms an AM demodulator. This has the effect of stripping the peak values off the square wave generated by the DAC, allowing for reconstruction of the signal from samples in $y_0[n]$. The demodulated output $y_2(t)$ requires additional smoothing, which is provided by $H_3(j\omega)$. While an analytic expression for $H_1(j\omega)$ cannot be easily calculated, it has a low-pass effect, and an example of $y_1(t)$ and $y_2(t)$ is shown in figure 7 below.

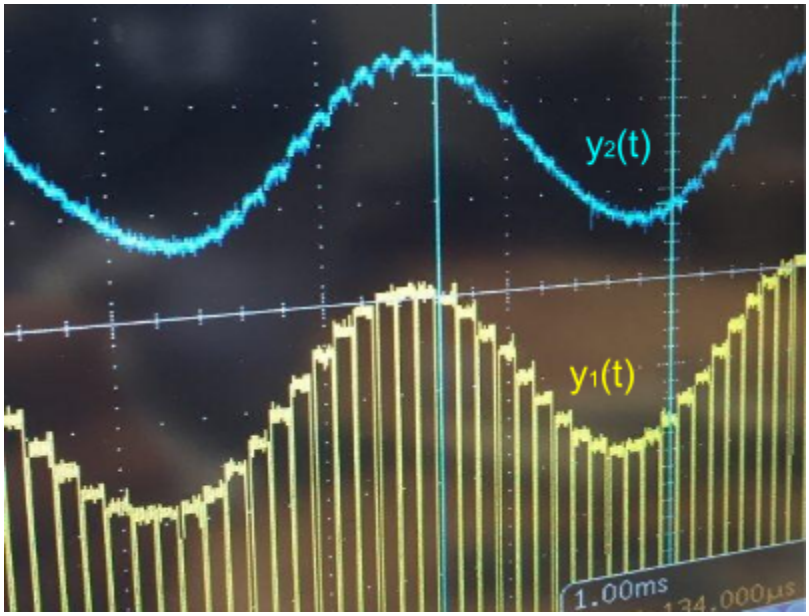


Figure 7: An example of a waveform passing through $H_1(j\omega)$.

The behavior of H_2 was chosen for two reasons. First, $H_2(0) = 0$ which removes any DC offset from $y_4(t)$. Second, $H_2(f) = \frac{1}{8}$ for all $f > 20$ Hz and $\angle H_2(f) = -180^\circ$ for all $f > 20$ Hz. This ensures a constant gain and phase for all audible frequencies. See figures 8 and 9 below for a complete description of filter behavior.

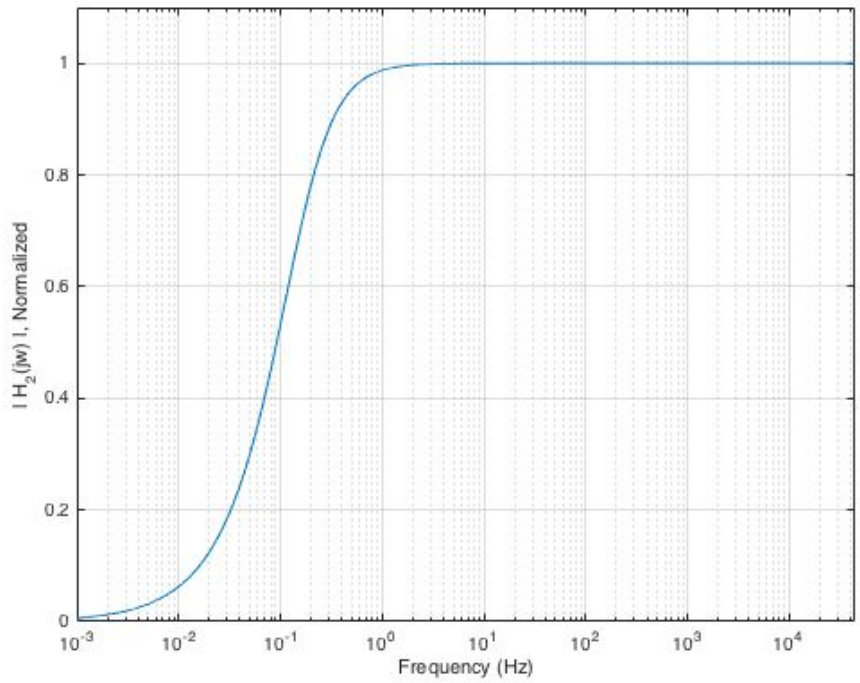


Figure 8: Magnitude of $H_2(j\omega)$

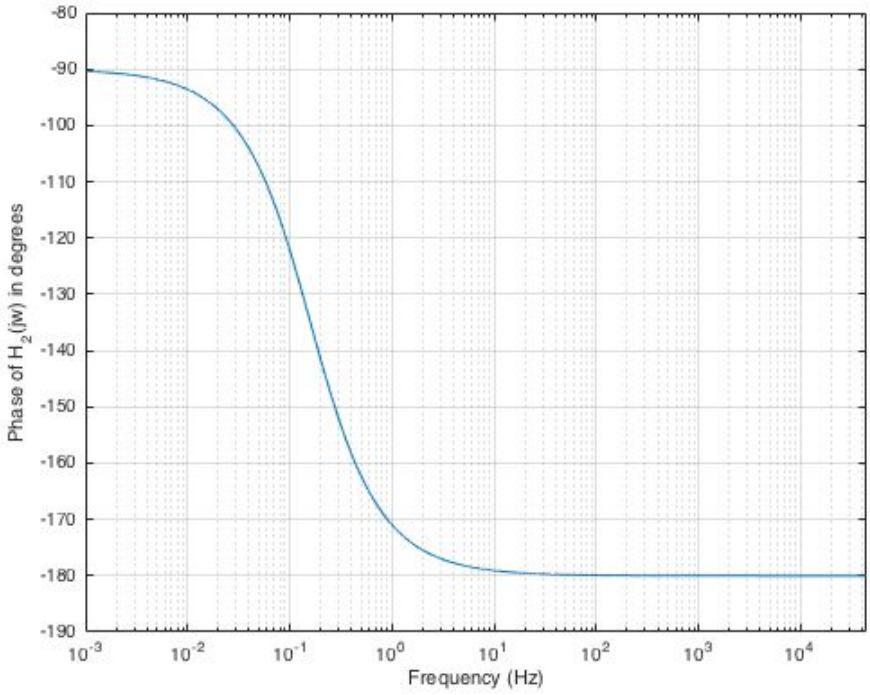


Figure 9: Phase of $H_2(j\omega)$

$H_3(j\omega)$ is a fourth-order, low-pass Butterworth filter, with a cutoff frequency of 4kHz. This filter serves to remove noise and nonlinearities, as well as smoothing the result of $H_1(j\omega)$. The Butterworth response was chosen in order to ensure an even frequency response for all audible components. $H_3(j\omega)$ was designed using Texas Instruments WEBENCH tools. See figure 10 below.

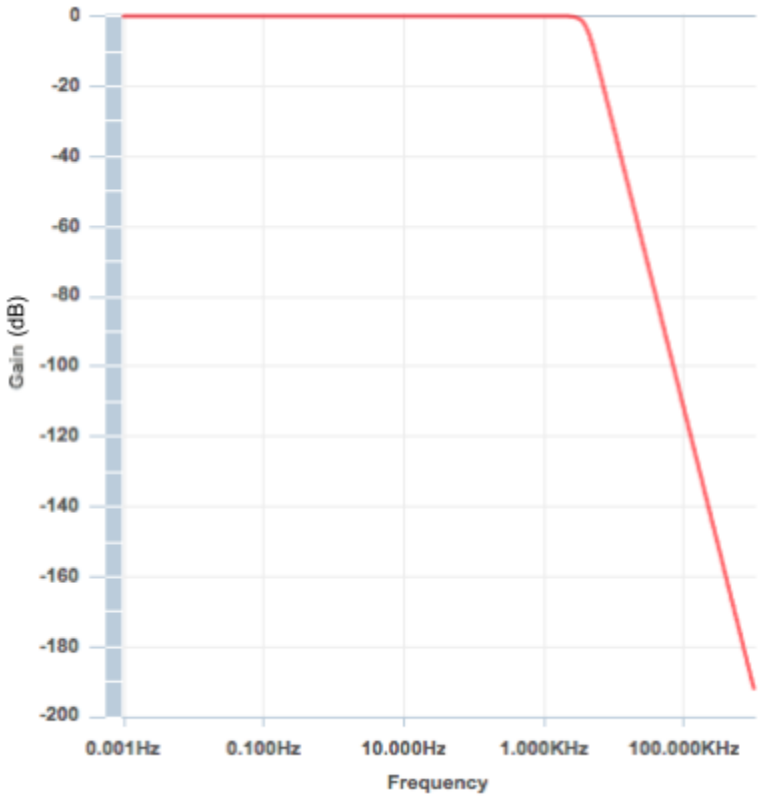


Figure 10: Frequency response of $H_3(j\omega)$.

WiFi Connectivity

Figure 11: IoT Process Flow

The ESP8266 establishes connectivity between the PIC32 controller and the mobile device. In order to do so, an asynchronous serial connection was established to the PIC device. This connection ran at a baud rate of 9600 Hz, sending 8 bits of data per transfer, with one stop bit and no parity bit. The primary UART of the ESP device was used to send information in order to allow for simple flash debugging. This required the devices to be disconnected for initial programming, but worked reliably after the initial setup. UART1 of the PIC32 controller was configured in the same manner to transmit and receive information.

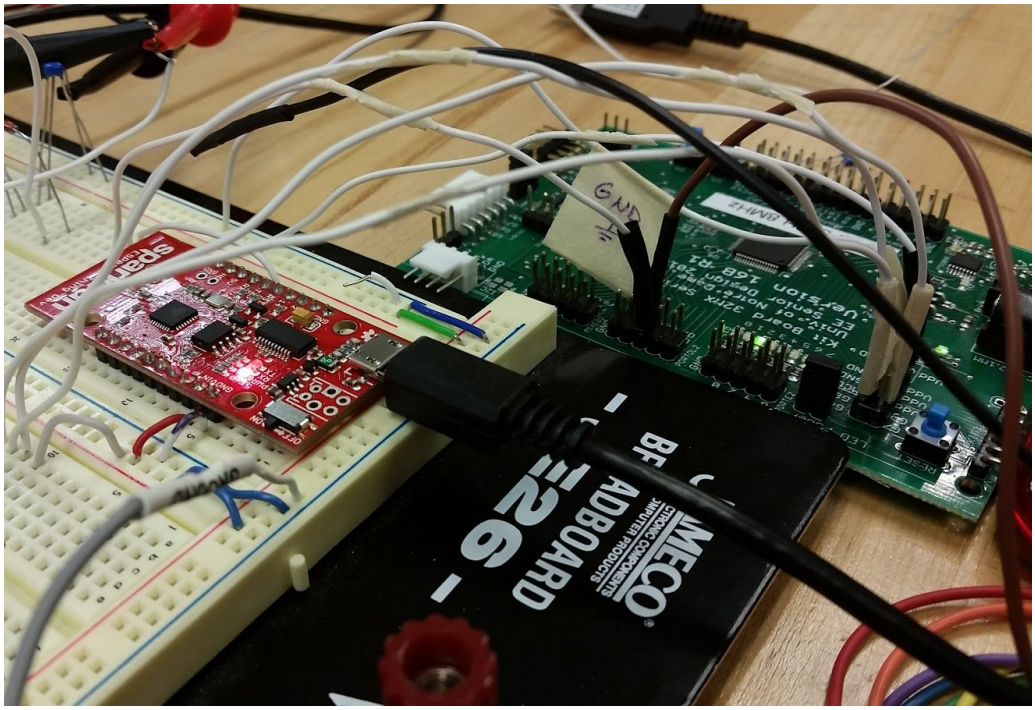


Figure 12: ESP connected to the PIC32 UART

In order to establish a connection with the mobile app that would not rely on the home WiFi network, the ESP8266 hosted it's own software enabled access point that allows a phone to connect directly to it. Once a connection with the phone was established, the ESP would wait to receive HTTP requests from the app. These requests would be in the form *ip.address/info* where *info* was information corresponding to the desired effect. Finally, this data would be converted to a single byte integer that was sent to the PIC via the UART connection. The code for the state machine that accomplished this task can be found in Appendix 1.

In order to test the connections, a variety of different methods were used. First, testing the WiFi access point was done by downloading code to the ESP Dev Thing and

searching for a WiFi signal with the proper name using a mobile phone. Then, the connection between the app buttons and the calling of the HTTP “GET” was tested by downloading versions of the app to an Android smartphone, connecting it to the ESP network, and using the buttons to print to the serial monitor connected to the Dev Thing. The logic analyzer was then connected to the Tx and Rx pins on the ESP board to test if the proper signal was being sent through the UART. Finally, the UART data was used to turn on LED’s on the PIC32. Additionally, the logic analyzer would look at the output of the Tx buffer of the PIC, which was set to output the data it had just received.

Mobile Application

In its final form, the mobile application was used to interface with the PIC32, enabling the user to select his or her desired audio effect. The app consisted primarily of four separate screens composed using .xml code with Java code serving as the “engine” behind the scenes. Upon launch of the app, the user is presented with the following screen:

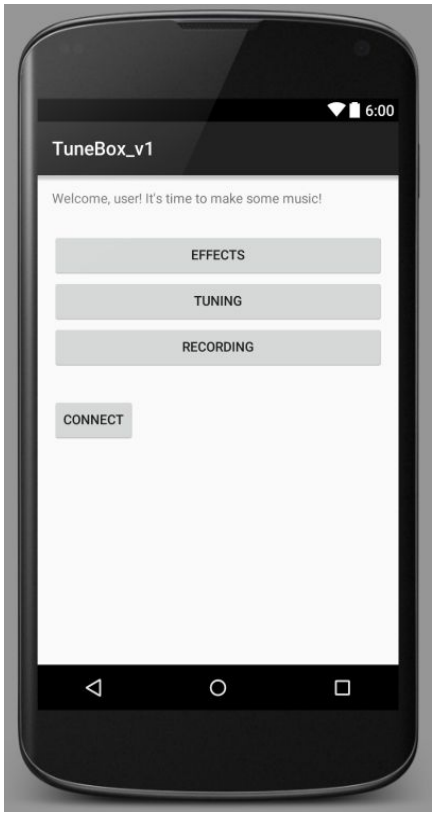


Figure 13: Mobile application home screen

The Java running each of these buttons is patterned after the following code snippet:

```
Button effectsbutton = (Button)findViewById(R.id.effectsbutton);
effectsbutton.setOnClickListener(
       new Button.OnClickListener() {
           public void onClick(View v) {
               String message = "Entering effects mode";
               Toast.makeText(MainActivity.this, message, Toast.LENGTH_SHORT).show();
               Intent effectsIntent = new Intent(MainActivity.this,
EffectsActivity.class);
               startActivity(effectsIntent);
           }
       }
);
```

Each button instance first had to be declared and linked to its corresponding .xml item, as depicted in the first line of the code. Then, a new click-listener had to be instantiated. Within this listener, the button's intended action 9s laid out. In the case above, the buttons do two things. First, they launch a small message at the bottom of the screen (called a "toast") which tells the user what the app is about to do, as in lines five and six. Next, the buttons make use of an intent structure to call up the next app page and launch the next app activity, depending on which button is pressed.

The code shown above is used to send the app from the initial screen into the effects-selection screen, shown here:

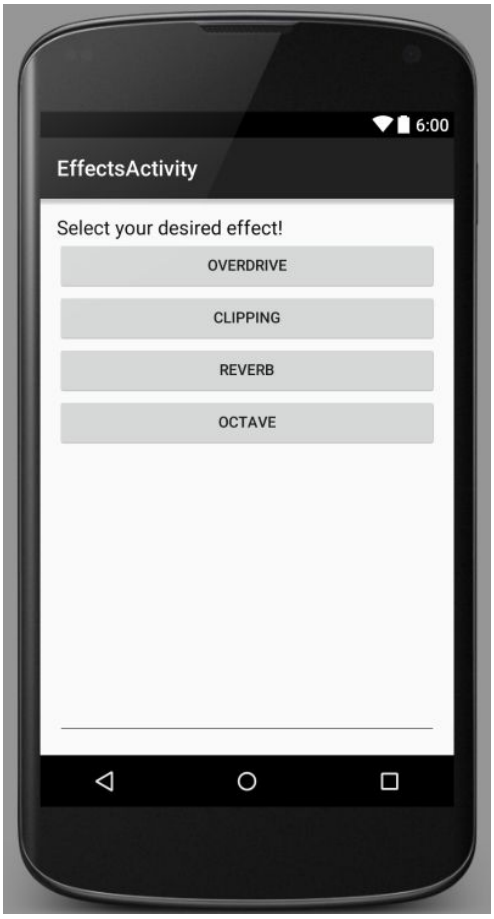


Figure 14: Effects selection screen

Each of the four buttons above represent the effects we implemented in C within the microcontroller. During the Senior Design demonstration day, we used the overdrive effect to demonstrate app functionality. After linking the buttons to their corresponding .xml elements as before, the following code lends functionality to the overdrive button:

```
overdrivebutton.setOnClickListener(
        new Button.OnClickListener() {
            public void onClick(View v) {
                String message = "Overdrive mode";
                Toast.makeText(EffectsActivity.this, message,
Toast.LENGTH_SHORT).show();
                new DownloadTask().execute("http://192.168.4.1/1");
            }
        }
);
```

Again, a click-listener is set to execute once the button is clicked. As before, we used a toast message to confirm to the user that the button was doing as it should. Within the effects-selection section, however, code was written to communicate via WiFi with the ESP device, in this case via an executable function called `DownloadTask()`. That function's code is shown here:

```
private class DownloadTask extends AsyncTask<String, Void, String> {
    @Override
    protected String doInBackground(String... params) {
        //do your request in here so that you don't interrupt the UI thread
        try {
            return downloadContent(params[0]);
        } catch (IOException e) {
            return "Unable to retrieve data. URL may be invalid.";
        }
    }
    @Override
    protected void onPostExecute(String result) {
        //Here you are done with the task
        //Toast.makeText(EffectsActivity.this, result, Toast.LENGTH_LONG).show();
    }
}
```

Within the `DownloadTask()` class is another asynchronous background function called `downloadContent` which is used to actually call the ESP device by IP address. Because this function is asynchronous and reliant on the outside environment, it had to be coded in a robust manner, with error catching capabilities written in. The actual code for `downloadContent` is as follows:

```
private String downloadContent(String myurl) throws IOException {
    InputStream is = null;
    int length = 500;
    try {
        URL url = new URL(myurl);
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setReadTimeout(10000 /* milliseconds */);
        conn.setConnectTimeout(15000 /* milliseconds */);
        conn.setRequestMethod("GET");
        conn.setDoInput(true);
        conn.connect();
        int response = conn.getResponseCode();
        Log.d(TAG, "The response is: " + response);
        is = conn.getInputStream();
        // Convert the InputStream into a string
        String contentAsString = convertInputStreamToString(is, length);
        return contentAsString;
    } finally {
        if (is != null) {
            is.close();
        }
    }
}
```

Essentially, the above code takes in a user-specified IP address in string form and issues an HTTP GET command to that IP address (line 12). Normally, this style of command is used to check that a solid connection has been instantiated. We used GET commands to different ports on the ESP's IP address to call for different responses from the ESP device. Again, robust code was necessary to ensure that the app wouldn't crash if proper communication wasn't established. More information about the interface between the mobile app and the ESP-8266 is given later.

Testing of the mobile application was performed first through the built-in emulator within Android Studio. Once all major bugs were worked out, most of which occurred because of non-robust code (subsequently improved), the app was loaded onto a physical Android phone and tested for hardware stability and proper UI flow. With all necessary code modifications made, the app was then connected to the ESP device via its built-in WiFi transceiver. See the section on interfaces for more information.

Bitstream to Audio Format

We were able to convert digital samples to .WAV file by adding the appropriate digital headers to the pure bitstream. These headers are used to describe the file type and set up the appropriate sample rate and bit speed for playback. In order to do this in for Android devices, we audio libraries typically included with Java could not be used and instead the bytes had to be manually set up using the proper format.

```
byte[]  riff_tag = {82, 73,  70, 70};  //RIFF Tag in ASCII bytes
byte[]  wave_tag = {87, 65, 86, 69};  //WAVE tag in ASCII bytes
byte[]  fmt_tag =  {102, 109, 116, 32};  //fmt tag in ASCII bytes
byte[]  fmt_length = {16, 0,0,0}; //Writing fmt_length in little-endian
byte[]  audio_format = {1, 0} ; //represents a 1 in 2 byte little endian
byte[]  num_channels = {1, 0};   //represents a 1 in 2 byte little endian
byte[]  sample_rate = {0,32,0,0}; //represents 8192 in little endian
byte[]      byte_rate = {0,64,0,0}; //represents 44100*2 in little endian
byte[]  block_align = {02,00};      // represents 2 in little endian
byte[]  bits_per_sample = {16,0};   //represents 16 in little endian
byte[]  data_tag = {100, 97, 116, 97};  // data tag in ASCII bytes
byte[] data = bFile;
byte[]  data_length = {0, 0, 0, 0 };
byte[]  riff_length = {0,0,0,0};
```

Figure 15: Assigning the .WAV header

Digital Effects

- The digital effects live on-board the PIC32, and are applied depending on the user's selection in the mobile app
- The effects are written in C and are called as functions within the main program
- Each effect receives an input of digital samples of the analog waveform
- Apply four effects (chosen by user): Clipping, Octave Shift, Reverb, and Overdrive
- Outputs a modified waveform to be sent from the PIC32 to the Audio Codec or the mobile app.
- Digital effects are represented by $h_0[n]$ in the audio processing section. $h_0[n]$ is time-invariant but not necessarily linear.

Clipping:

The clipping function is designed to provide a "grunge" sound to the guitar. To do this, the function begins by taking each digital bit received by the PIC32 as an input, and storing them in a static array within the function. The array is then checked for a maximum. The static array is periodically overwritten to account for fluctuation in input amplitude (i.e. the change in how hard the guitar is being strummed). After a maximum is calculated, the output values can be restricted to a fraction of the maximum. For example, the code in Figure X clips the output at 0.8 of the determined maximum. This

means that any input value that is 80% of the maximum or greater will be replaced with the value 0.8 of the max value. Figure 16 shows the final version of the Clipping C code.

```
26  int Clipping(short input) {
27
28      static int array[SIZE];
29      static int i=0;
30      int j;
31      static short maximum = 0;
32      short output;
33      array[i] = input;
34      i++;
35
36      if(i>=SIZE) {
37          i = 0; }
38
39      for (j = 0; j<=i; j++) {
40
41          if(array[j] > maximum) {
42              maximum = array[j];
43          }
44      }
45
46      if(input >= .8*maximum) {
47          output = .8*maximum;
48      }
49      else if(input <= -.8*maximum) {
50          output = -.8*maximum;
51      }
52      else {
53          output = input;
54      }
55
56      return output;
57  }
```

Figure 16: Clipping C code

The clipping function is illustrated in Figure 17, in which a sine wave is clipped to 55% of its peak value.

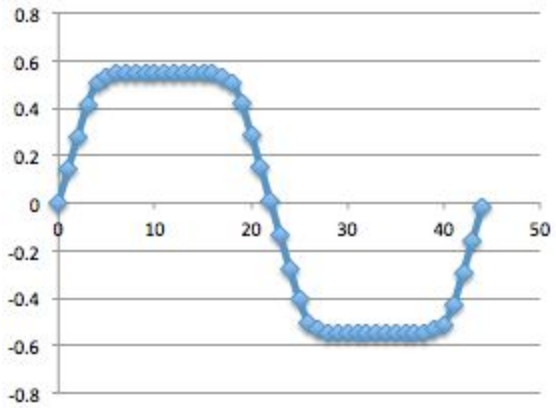


Figure 17: Clipped Sine Wave

Octave:

The octave effect adds harmonics to the waveform, giving the audio a greater range of frequencies. The function is given each digital bit as an input short. The inputs are stored in a static array. The function then attempts to find the frequency of the audio from the stored bits. Once a frequency is determined, sine waves of double and half of the frequency are added to the output. The octave effect is illustrated, given an input sine wave, in Figure 18.

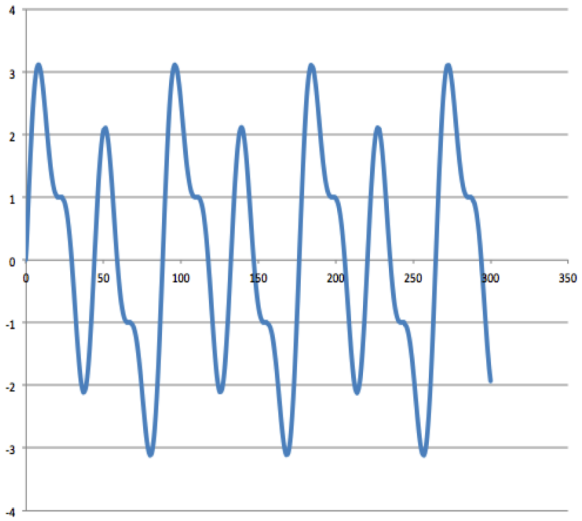


Figure 18: Octave Effect for Sine Wave Input

Reverb:

The reverberation effect can be described as an echo. The function takes each digital sample as an input short. The bits are all stored in a static array which is periodically overwritten. The values stored from two seconds prior dampened and added to the current input value. This value is returned by the function. An example of this effect can be seen in Figure 19. The effect is also capable of adding additional “echoes” to the output waveform. Two echoes are illustrated in Figure 19.

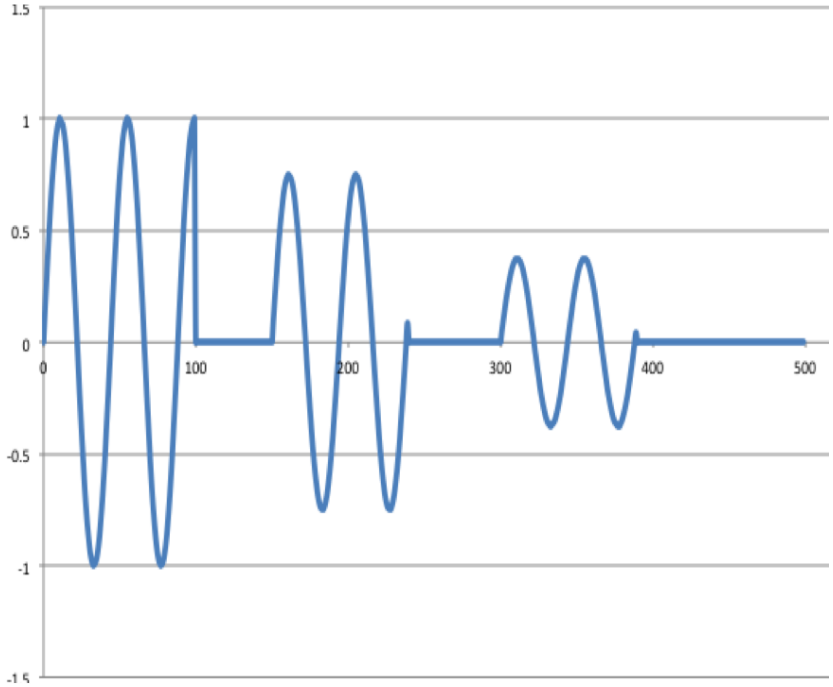


Figure 19: Reverb Effect for Input Sine Wave

Overdrive:

The output waveform of overdrive function is similar to the clipping function, but differs in how the waveform is modified. Rather than clipping the waveform at a percentage of the maximum, the overdrive function establishes the cutoff point at the maximum, and

then amplifies the signal so that signal is “clipped” at the maximum. An illustration of this effect can be seen in Figure 20.

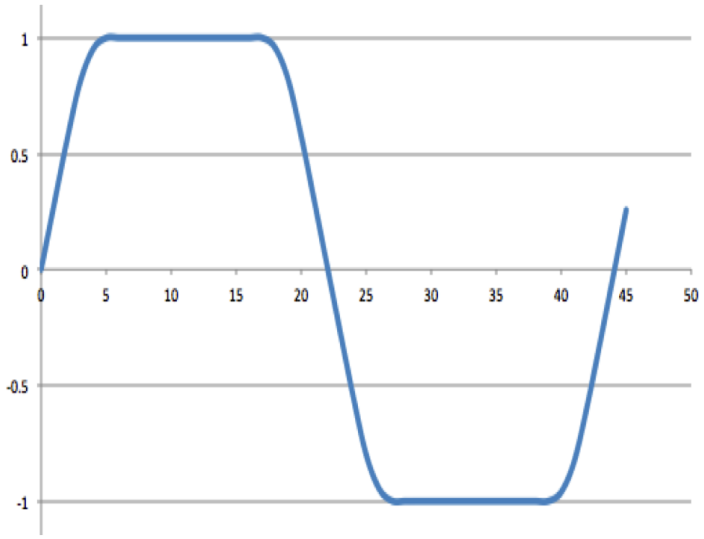


Figure 20: Overdrive Effect for Input Sine Wave

Interfaces

Analog-to-Digital Conversion (ADC) and Digital-to-Analog Conversion (DAC):

The ADC interfaces between the conditioned guitar signal and the software on the PIC32. The guitar signal, once modified by the front end analog system, is fed into pin B4 on the PIC32 and processed by the on-board ADC. The digital bits are then fed to the main program, which applies effects chosen by the user.

Unlike the ADC module, the DAC exists as an IC that is independent of the PIC32. The IC chosen to perform DAC is the MCP4921. The digital samples, after being processed by the PIC32, are sent as serial data to pin 4 (SDI) of the MCP4921. The data is then

converted to voltage values and output on pin 8 (VoutA). The output is a series of square waves which are then demodulated by the analog circuitry.

UART:

The UART connection is established through the connection of the Tx Pin of the ESP and the U1RX pin of the PIC32 as well as the Rx pin U1TX pin of the two boards. The connection runs at 9600 Hz and sends an 8 bit signal with no parity. On the PIC, the UART connection is established by dividing the peripheral bus clock, initializing the UART, waiting for the receive buffer flag to go off, reading the receive buffer, and writing to the transmit buffer. The ESP code simply calls the function serial.init(9600) and then uses serial.print() to write integer values to the Rx buffer.

WiFi:

The WiFi connection is established on the Arduino using the ESP8266Wifi.h library. It allows the calling of Wifi.mode(WIFI_AP) that directs the ESP to host it's own WiFi server. After initializing the connection and giving a host name and password, the program waits until a client connects to its IP address. Then, it reads the corresponding GET string and sends that information to the UART. Before terminating, the server flushes the client information and waits for a new GET function to be called.

User/Installation Manual

Currently, installing the TuneBox application requires connecting the phone via USB to the computer containing the original software and selecting Debug>*Phone_Name* in Android Studio. The benefit of using Android for this project is that becoming a developer and making the app available for download on the Play Store is much simpler than for iOS.

To set up the TuneBox, first attach the INPUT end of the device to the output of an electric guitar via a ¼'' Male to Male Audio Cable. Next, attach the OUTPUT of the device to the input of a standard amplifier-speaker system for electric guitars using the same type of chord connection. Powering on the device as well as the guitar amp will allow for noticeable audio to begin to be heard through the amplifier.

To set up the application on an Android device with the app TuneBox_v1 installed, go to: Settings>Wi-Fi and make sure WiFi is turned on. Next, if the TuneBox is already powered up, the Android phone is able to discover a network with the name "TuneBox" and the password "password". After entering this information, open the app itself and select "EFFECTS". The different available sounds will be displayed and you will be able to select between them. The user can visually acknowledge that the app is sending information through the ESP to the PIC device because an LED corresponding to the effect will be illuminated.

To-Market Design Changes

In order to be able to bring our device to market, there are a number of improvements that would be required. First, the PCB board would need to be corrected to account for our change in parts. Using a designated Audio Codec to accomplish ADC and DAC would still be beneficial as well. We would also need to edit the code for other effects in order to allow for switching between different sounds. This would also most likely require a faster processor to be used in the design. Finally, because SoundCloud doesn't allow most mobile apps to upload sound files, we had ignored the saving and uploading of audio data. To make this a more commercially viable product, determining a host site where we can save audio files would be necessary.

Conclusions

From the start, our project was very ambitious. Even after removing the various extraneous features which we initially envisioned, the project remained complicated. We are proud of how it turned out. Despite the setbacks along the way which ultimately led to the use of development boards for the final demonstration, the project's core functionality was not compromised. During the design process, we gained valuable insight into how different forms of electrical and computer engineering can be brought together to create a unique product. If we were to repeat the process, we would think about how to more efficiently achieve our end goals while ensuring that we were using the best parts for the job. Our overall conclusion is that a good design is one which

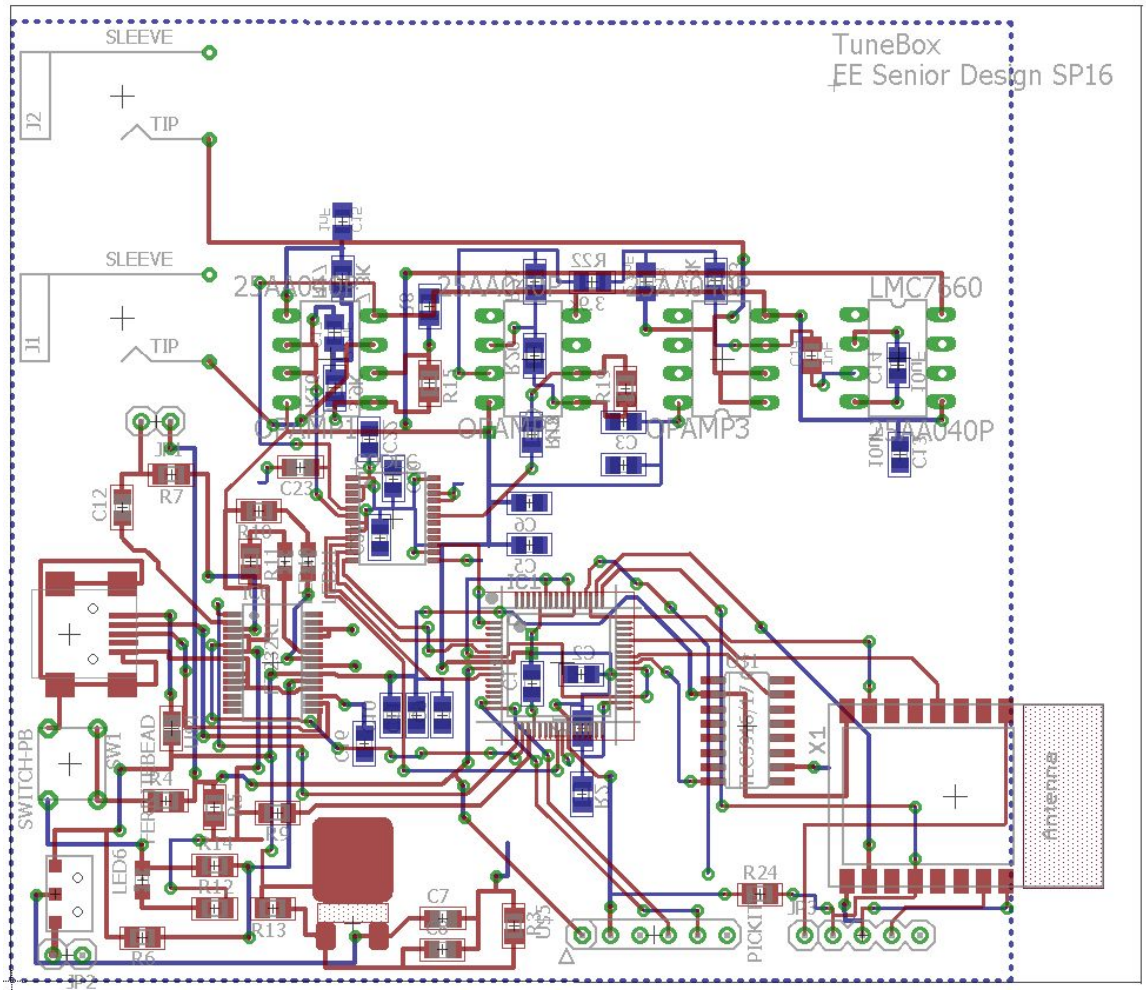
properly manipulates the small factors and details while maintaining respect for the overall goal throughout the process.

Appendices

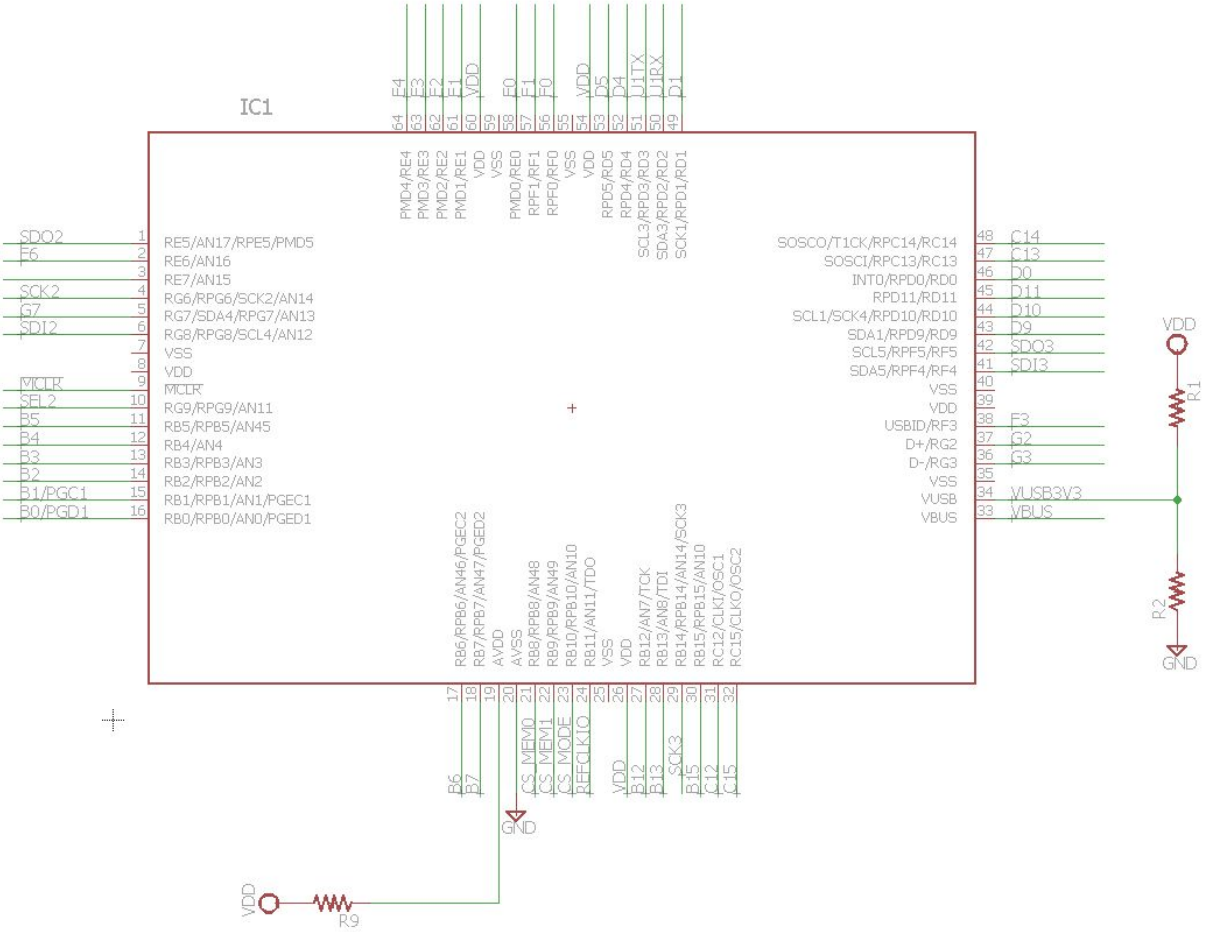
Appendix 1: ESP8266 State Machine Code

```
#include <ESP8266WiFi.h>

//////////////////////
// WiFi Definitions //
//////////////////////

const char WiFiAPPSK[] = "password";

//////////////////////
// Pin Definitions //
//////////////////////

const int LED_PIN = 5; // Thing's onboard, green LED

WiFiServer server(80);

void setupWiFi(){
  WiFi.mode(WIFI_AP);

  const char AP_NameChar[] = "tuneBox";
  WiFi.softAP(AP_NameChar, WiFiAPPSK);
}
void initHardware(){
  Serial.begin(9600);
  pinMode(LED_PIN, OUTPUT);
  digitalWrite(LED_PIN, HIGH);
  // Don't need to set ANALOG_PIN as input,
  // that's all it can be.
}
void setup() {
  initHardware();
  setupWiFi();
  server.begin();
}
void loop() {
  // Check if a client has connected
  WiFiClient client = server.available();
  if (!client) {
    return;
  }
  // Read the first line of the request
  String req = client.readString();
  byte reqInt = (byte) req[5];

  //
  if (req != 0){
    digitalWrite(LED_PIN, HIGH);
    Serial.print(reqInt - 48); //converts ascii representation to actual int value
    //Serial.println("We got the message");
  }
  else{
    //Serial.println("We didn't get the message");
  }
  client.flush();
}
```

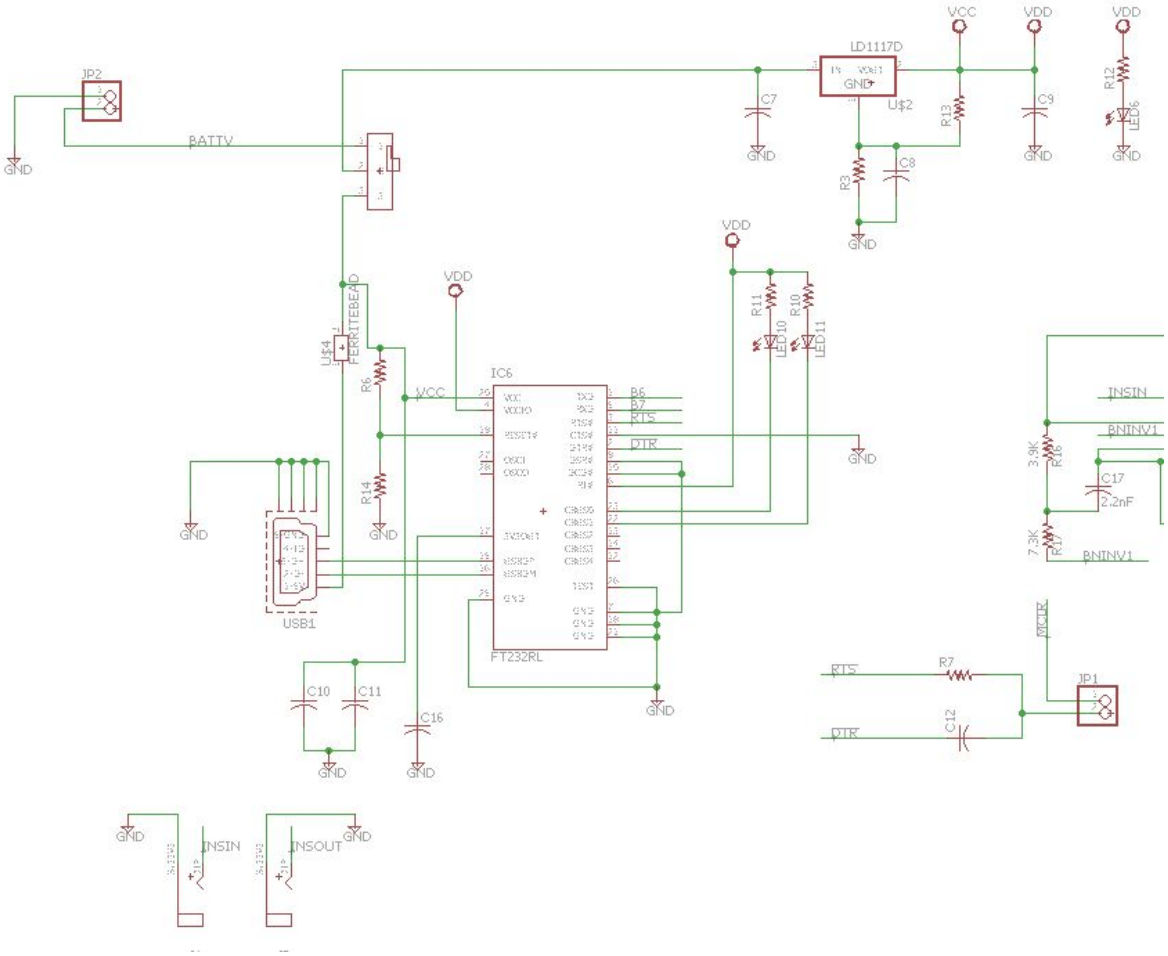
Appendix 2, Part A: Board Design



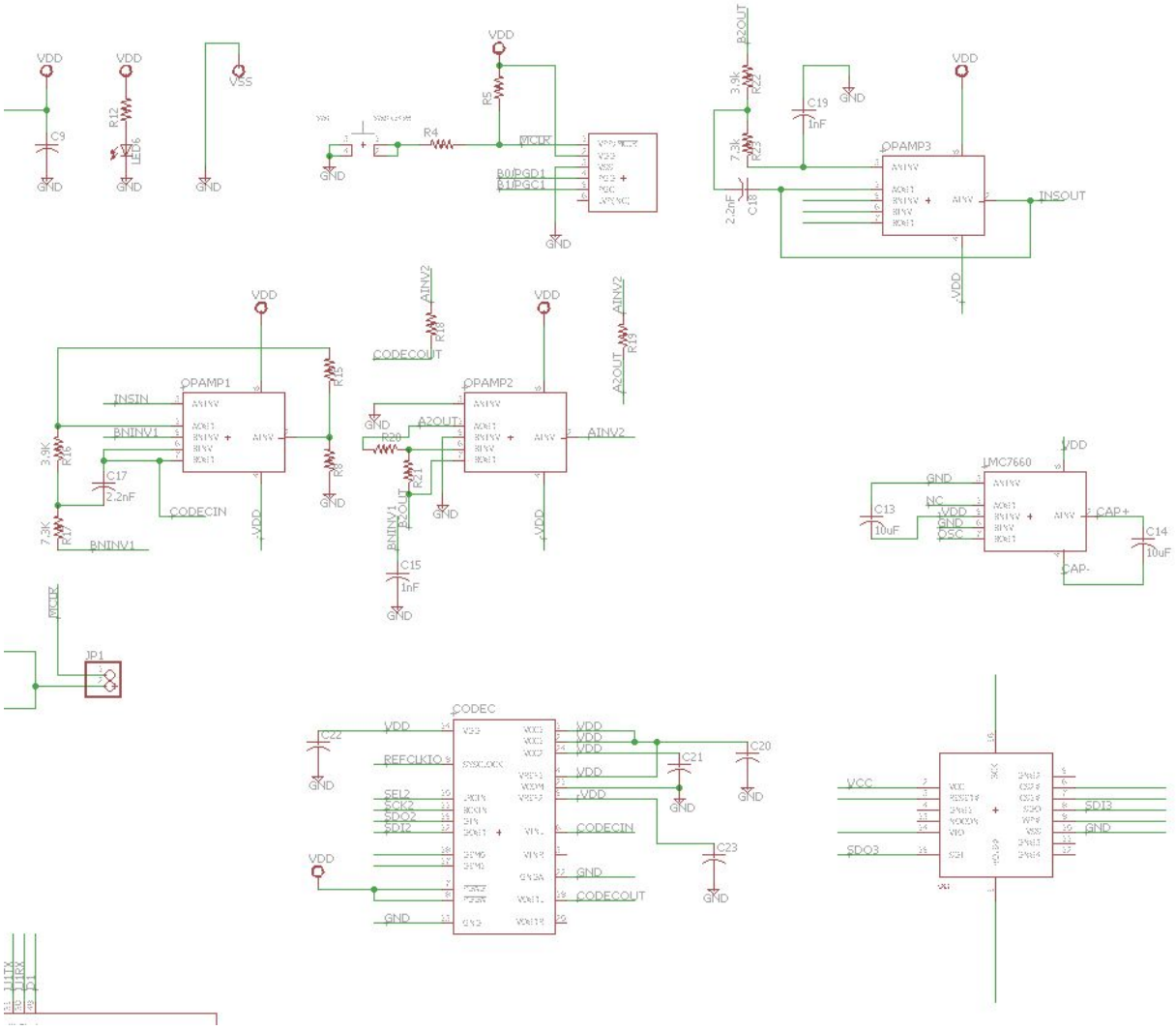
Appendix 2, Part B: Schematic Design I



Appendix 2, Part C: Schematic Design II



Appendix 2, Part D: Schematic Design III



Appendix 3: Complete Software Listings

- Please see the relevant section of our website for complete code listings

Appendix 4: Datasheets

- Please see the relevant section of our website for datasheets used in the project