Junior Design Final Project: System Verification Documentation

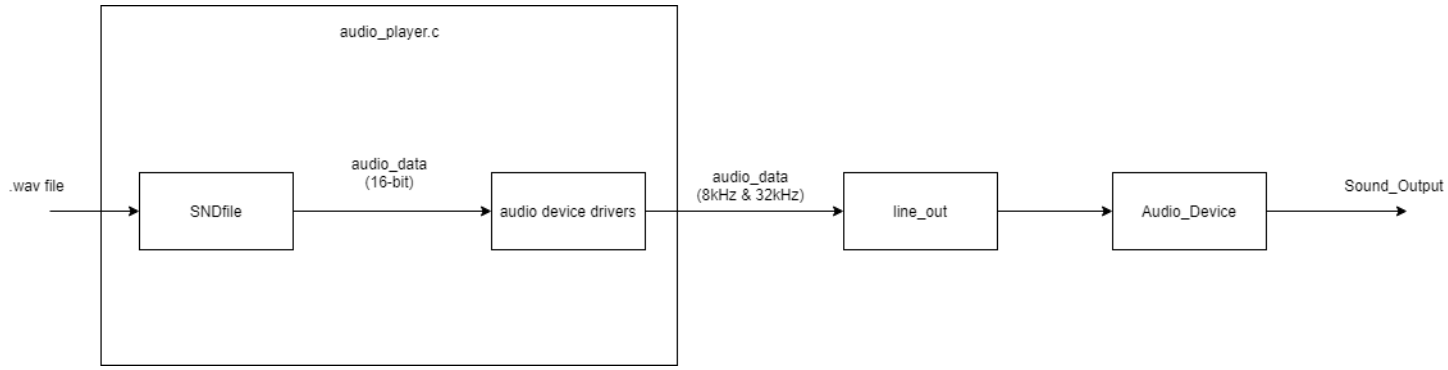
Interface Definitions

Project: Music Box
Block 1: Audio Player
Block 2: LED Lighting System
Block 3: PCB Design
Block 4: Audio Recorder
Block 5: Fourier Transform
Block 6: Enclosure
Block 7: Human Interface Device

Group Members:
Max Altenhofen
Joshua Walund
Joshua Wentzel

1 Audio Player Assigned Member: Max Altenhofen

2 Audio Player Diagram



3 Audio Player: Interface Definitions

Inputs

1. .WAV File: Either a pre-edited tune or a recorded sound that has been sent through an FFT program will be taken in by the audio player for processing. Which one is not determined by the audio player itself as the song manager determines that.

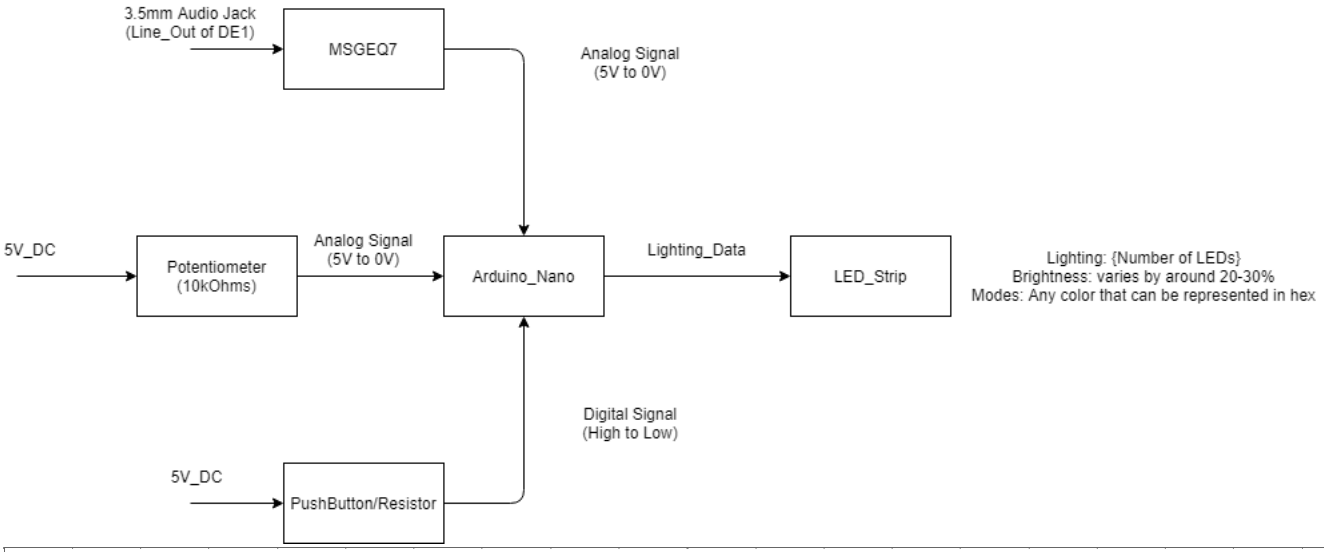
2. Maximum .WAV file parameters:
   Sample Rate: 32,000 Samples per second
   Sample Depth: 16 Bits
   Bit Rate: 32,000 X 16 = 512,000 Bits per Second

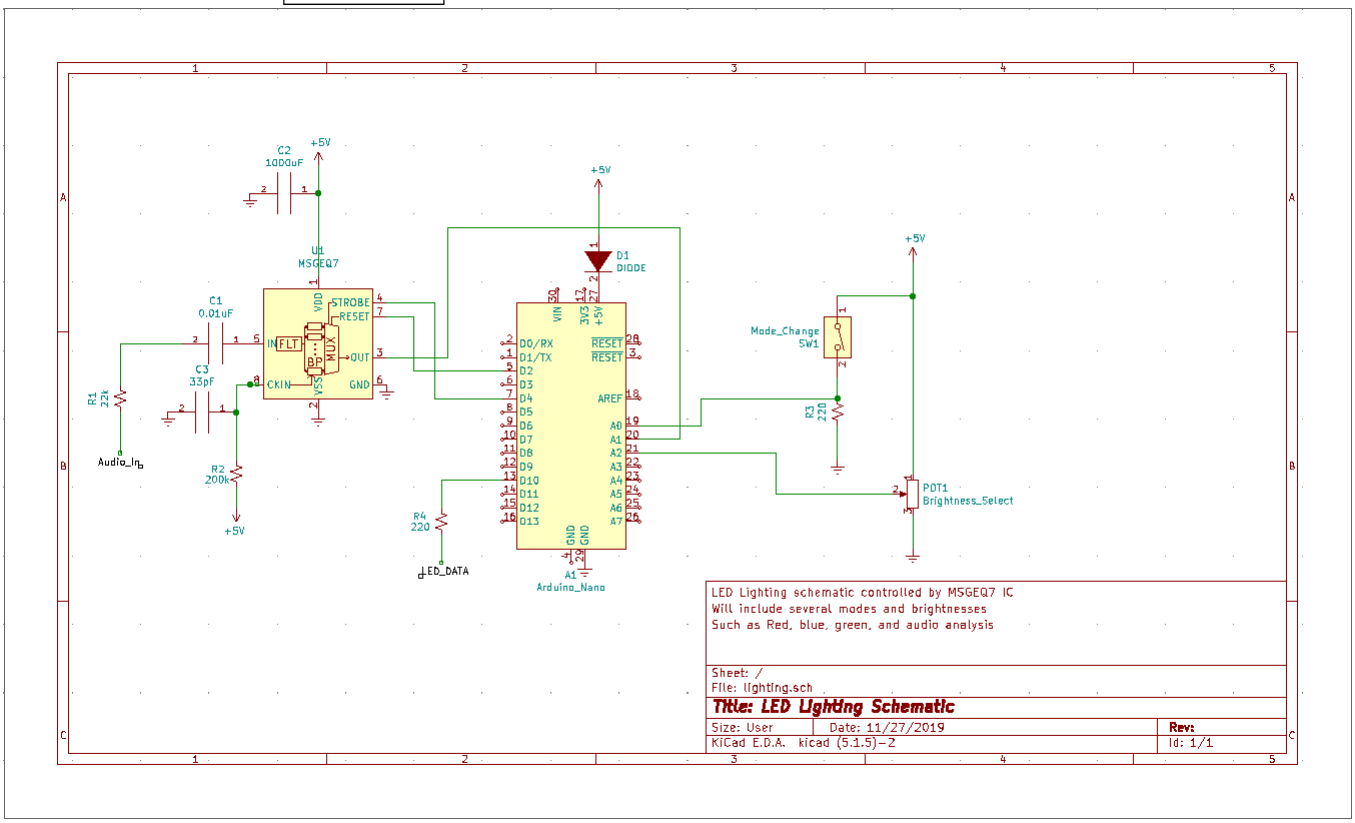
3. Minimum .WAV file parameters:
   Sample Rate: 8,000 Samples per Second
   Sample Depth: 16 Bits
   Bit Rate: 8,000 X 16 = 128,000 Bits per Second

Footnote on Audio player: Any .wav file may be played through the audio device so long as it meets the above requirements. Sound editing software such as Audacity may be used to do so. Due to the finicky nature of the DE1's audio device, it is possible to play simple sounds at higher sample rates as is the case with the audio_recorder output file

4 LED Lighting System Assigned Member: Max Altenhofen

5 LED Lighting System Diagram and Schematic





6 LED Lighting System: Interface Definitions

1. User Interface: Mode Change Button
   Six Functioning modes:

(a) RainbowFade2White: Static colors. Along the length of the LEDs, fades through the color spectrum to end in white.

(b) Red: Changes color to Red

(c) Green: Changes color to Green

(d) Blue: Changes color to Blue

(e) White: Changes color to white

(f) Color Change: Uses the brightness potentiometer to change colors

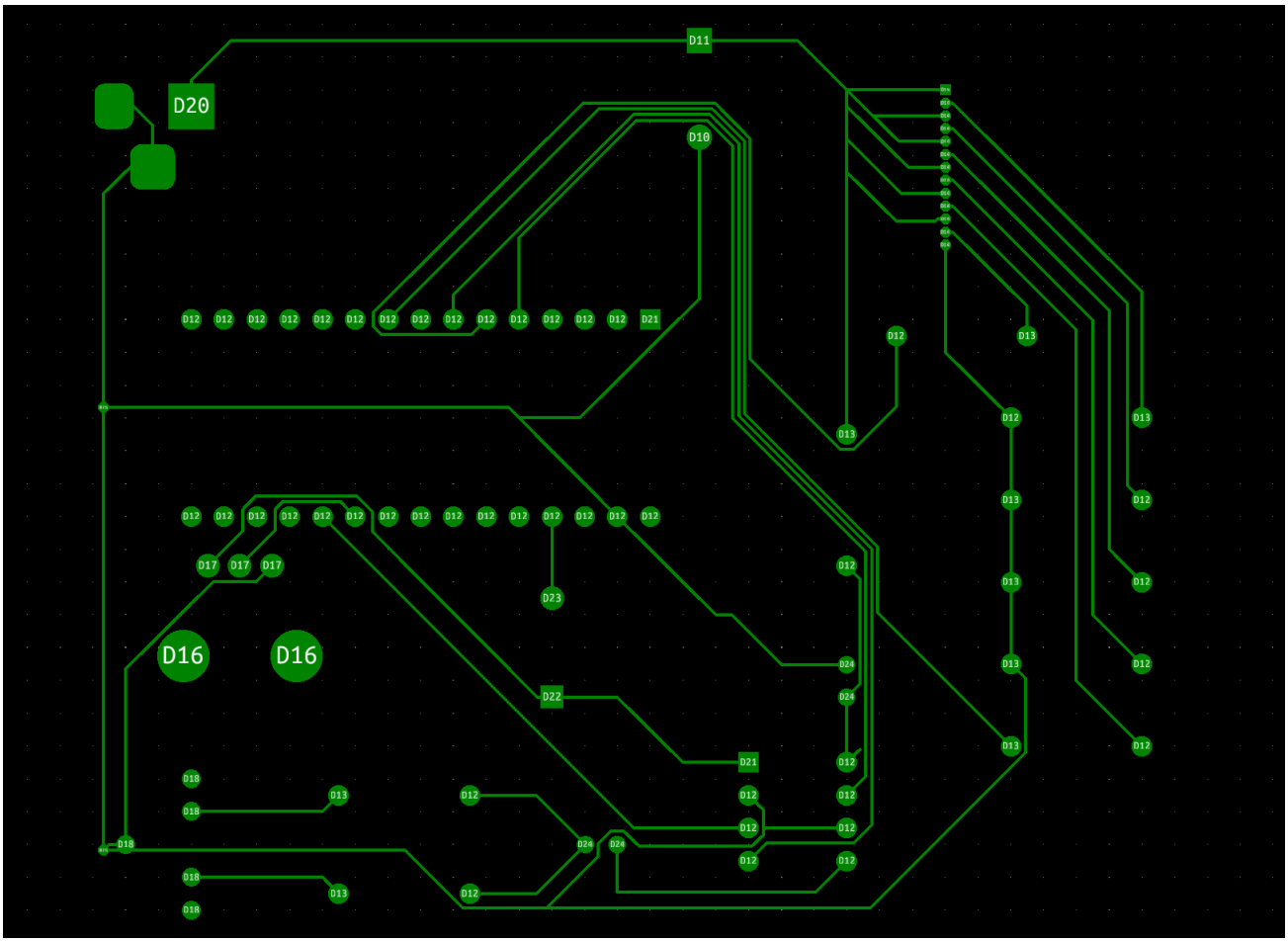
One Mostly Functioning Mode: Audio Analysis Mode.
Reacts to sound although a bit haphazardly.
How to fix issue in a later version of the circuit: Add a pre-amp to the audio jack and use the recommended capacitor for CKIN on the MSGEQ7.

2. MSGEQ7 Input Voltage , Supply Current:
   Maximum: 5.5 Volts , 1 Milliampere
   Typical: 5.0 Volts , 0.8 Milliampere
   Minimum: 2.7 Volts , 0.5 Milliampere

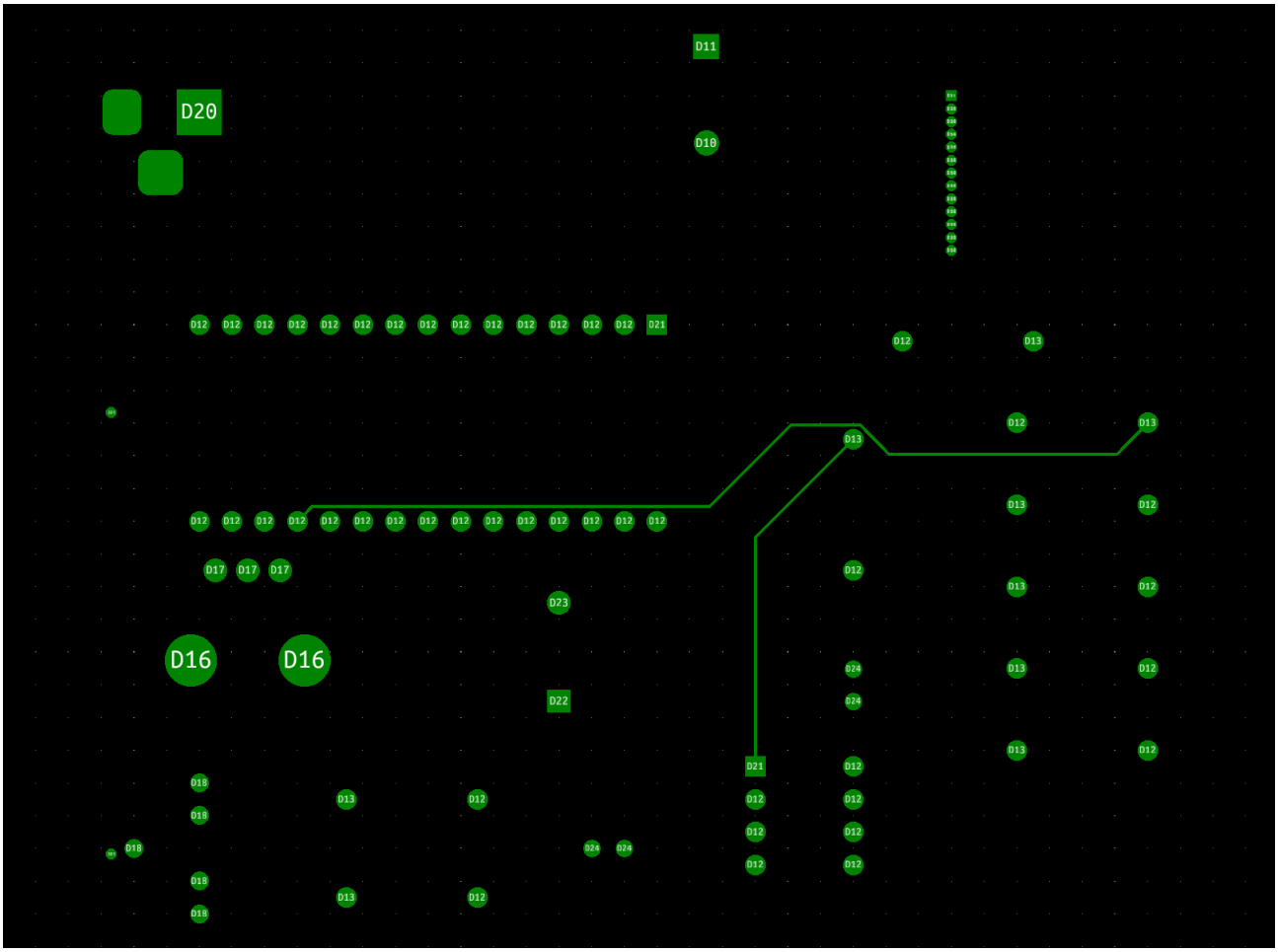
3. MSGEQ7 Output Current and Voltage Offset:
   Current: 1 Milliampere
   Voltage Offset: 600 Millivolts

7 PCB Design Assigned Member: Max Altenhofen

8 PCB Design Front Copper Layer



9 PCB Design Back Copper Layer



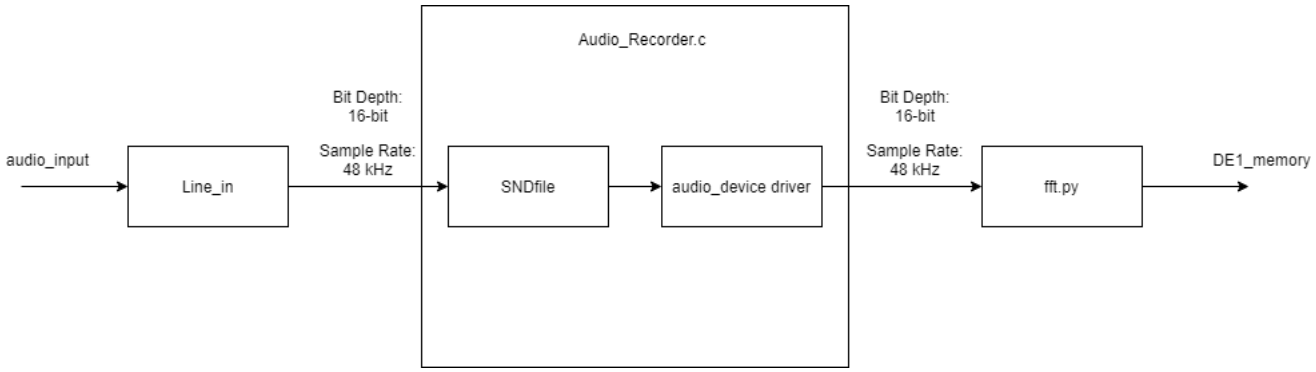
10 PCB Design: Interface Definitions

The design for the PCB includes the whole circuit design for the Lighting system as well as unused resistors meant to be used by the GPIO port on the DE1-SoC. The block interface definitions are, therefore, the same as the above block.

Footnote on PCB: Unfortunately, due to this being Max's first PCB design, some mistakes were made that rendered the PCB mostly unusable. The first of these mistakes is the jumper used to connect the buttons and LEDs to the PCB. The jumper footprint doesn't appear to match with any existing jumper. The second and last mistake has to do with the 3.5mm audio jack footprint. Due to that footprint not existing, Max had to make it on his own. The mistake is almost completely unforseeable as the pads for the audio jack are just barely too small which appears to be due to a manufacturing condition. These are both very easily avoidable in the future by ensuring that the footprints are gathered from better sources.

11 Audio Recorder Assigned Member: Joshua Wentzel

12 Audio Recorder Diagram



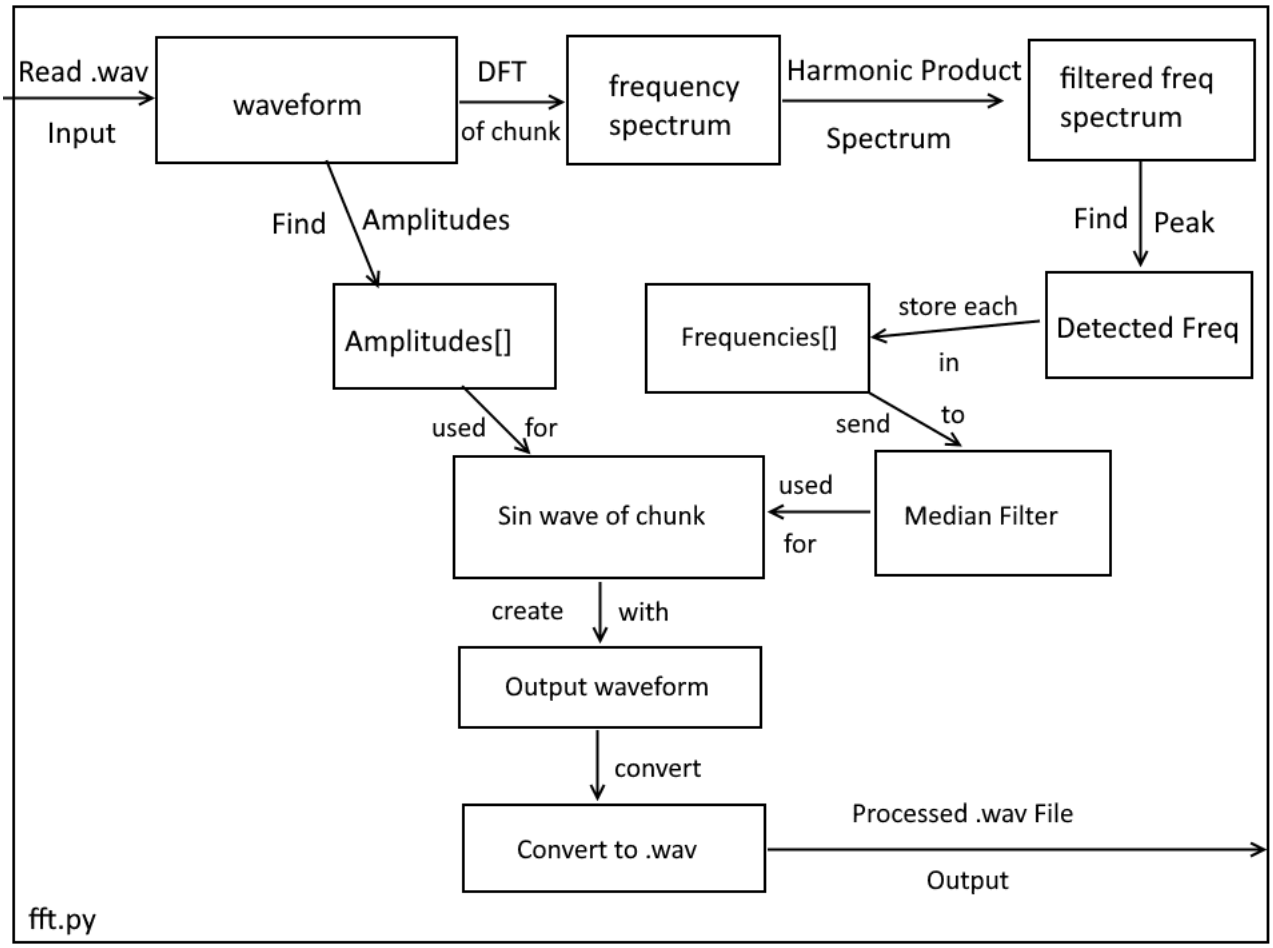
13 Audio Recorder: Interface Definitions

1. Input: Line in
   The audio device drivers will sample whatever it sees from line in.
   Sample Rate: 48kHz
   Bit Depth: 16-bit
   Bit rate: 16 X 48,000 = 768,000 bits per second

2. Output: .wav file
   Sample rate: 48 kHz
   Bit depth: 16-bit
   Bit rate: 16 X 48,000 = 768,000 bits per second

14 Fourier Transform Assigned Member: Joshua Wentzel

15 Fourier Transform Diagram



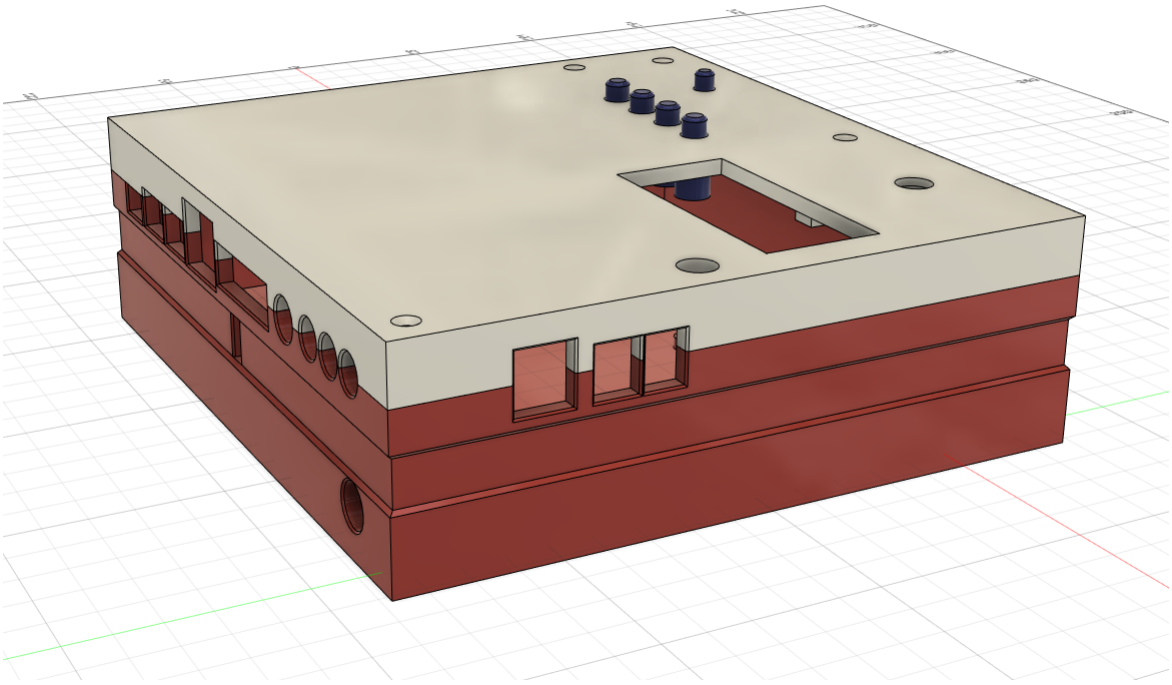
16 Fourier Transform: Interface Definitions

1. Input: Recorder_To_FFT
   Receives .wav file from audio_recorder block.
   Bit Depth: 16-bit
   Sample Rate: 48kHz

2. Output: FFT_To_Player
   Saves .wav file so that it may be played on command at a later time.
   Bit Depth: 16-bit
   Sample Rate: 48kHz

17 Enclosure Assigned Member: Joshua Wentzel

18 Enclosure Diagram



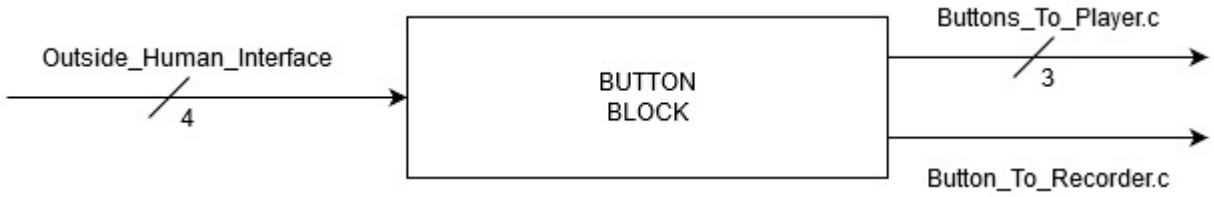
19 Enclosure: Interface Definitions

1. Inputs
   Four Song Buttons: Detected on Release
   Light Button: Detected on Press
   Brightness Potentiometer: Controls brightness/color in color mode
   Audio_in: Line in for audio recording
   Daughter Board Power Supply: Powers daughterboard/s. 5V12A
   DE1-SoC Power Supply: Powers the DE1-SoC 12V2A

2. Outputs
   7 segment display: Displays load time percentage
   Lights: StYlE pOiNtS

20 Human Interface Device Assigned Member: Joshua Walund

21 Human Interface Device Diagram



22 Human Interface Device: Interface Definitions

1. User Interface: 4x Play/Record Buttons
   These buttons tell the DE1-SoC to execute one of 4 different operations: Play songs 1 or 2, Record song, and play recorded song.
2. Output: It does the actions above when the buttons are pressed.

23 Bill of Materials

1. DE1-SoC: $249
2. Arduino Nano: $8.99
3. 1xdiode: $0.39
4. 2x10k Resistors, 2x220 Resistors: Large pack of various resistors $10
5. 1xPushbutton: $0.26
6. 1x 30 RGB LED/meter strip: $21.26
7. 1x33pF capacitor, 2x10nF capacitor, 1x1000uF Electrolytic capacitor: Large packs of various capacitors $10
8. 1xMSGEQ7 Equalizer IC: $5.99
9. 3.5mm Audio Jack: $0.75
10. DC Power Jack: $1.20
11. 5V10A Power Supply: $17.99

24 Customer Requirements

1. Customer Requirement: The system must control lights. Engineering Requirement: An RGB LED must have at least 10 different colors and 10 different brightness levels for the system lighting.

2. Customer Requirement: The system should play pre-programmed songs. Engineering Requirement: The user should be able to select from at least two pre-programmed songs to be played with the project. The songs should be at least 20 seconds long.

3. Customer Requirement: The system should be able to playback recorded songs. Engineering Requirement: The system will record audio data of 100 Hz to 5 kHz. It will playback a tone that locks to the correct note of a piano, with less than 4% error on frequency. The duration of the played back note will be accurate within .02 seconds. The total song length will be at least 20 seconds.

4. Customer Requirement: The system must be aesthetically pleasing. Engineering Requirement: There will be no visible tape, cardboard, or other non-drafted materials on the final prototype.

5. Customer Requirement: The system must be easy to use. Engineering Requirement: 9 of 10 people should be able to easily read the documentation on the project to understand how to change the lights, select a song to be played, and record their own song to be played back.

   ADDITIONAL REQUIREMENTS

6. Customer Requirement: Make the project using cutting edge technology. Engineering Requirement: Make this project with an FPGA to control lights, play songs, record songs, and compute the FFTs.

7. Customer Requirement: Systems lights will have a mode that responds to the sound produced by the DE1-SoC through Line_Out.

25 Source Code

Fourier Transform Code:

```
import scipy.signal as sp
import matplotlib.pyplot as plt
import numpy as np
import wavio
import sys

from scipy.io.wavfile import write

import matplotlib
matplotlib.use('WebAgg')
```

```
# matplotlib inline


def freq_to_key(freq):
    return round(12 * np.log2(freq / 440) + 49)


def key_to_freq(key):
    return 2**((key - 49) / 12) * 440


def resize(array, samples):
    array = array[:-(len(array) % samples)]
    return sum([array[n::samples] for n in range(0, samples)]) / samples


def squish(squish_this, num_squishes):
    output = np.copy(squish_this)
    for squish in range(2, num_squishes):
        output[:len(output) // squish] += resize(squish_this, squish)
    return output


myrecording = wavio.read(sys.argv[1])

fs = myrecording.rate  # 44100 samples per second

if len(sys.argv) >= 3:
    start_chunk = int(sys.argv[2])
else:
    start_chunk = 50

CHUNK_SIZE = 256
chunk_length_secs = CHUNK_SIZE / fs
WINDOW_SIZE = 20024  # 4000

LOW_KEY = 20
HIGH_KEY = 100

scan_key = 1

myrecording = [item[0] for item in myrecording.data]
myrecording = np.array(myrecording)


idx_upper_bound = int(WINDOW_SIZE * key_to_freq(HIGH_KEY) / fs)
```

```
idx_lower_bound = int(WINDOW_SIZE * key_to_freq(LOW_KEY) / fs)


def analyze_chunk(chunk_num, data, acc):
    wavdata = data[chunk_num * CHUNK_SIZE:chunk_num * CHUNK_SIZE + WINDOW_SIZE]
    windowed_data = sp.windows.general_gaussian(
        len(wavdata), p=1, sig=WINDOW_SIZE) * wavdata

    power = np.abs(np.fft.rfft(windowed_data))
    freq_peaks = []
    power_harmonics = squish(power, 5)
    freqs = np.fft.rfftfreq(wavdata.size, d=1 / fs)

    power_harmonics[:idx_lower_bound] *= 0
    power_harmonics[idx_upper_bound:] *= 0

    freq_peaks = sp.find_peaks(
        power_harmonics[idx_lower_bound:idx_upper_bound], distance=3, height=500000)
    bin_width = abs(freqs[1] - freqs[0])

    new_peak_frequencies = []
    new_peak_magnitudes = []
    new_bins = []
    for peak in freq_peaks[0]:
        a = power_harmonics[peak + idx_lower_bound - 1]
        b = power_harmonics[peak + idx_lower_bound]
        c = power_harmonics[peak + idx_lower_bound + 1]
        bin_offset = 0.5 * (a - b) / (a - 2 * b + c)
        new_peak = b - 0.25 * (a - c) * bin_offset
        new_freq = freqs[peak + idx_lower_bound] + bin_width * bin_offset
        new_peak_frequencies.append(new_freq)
        new_peak_magnitudes.append(new_peak)
        new_bins.append(bin_offset + peak)

    best_freq = new_peak_frequencies[np.argmax(new_peak_magnitudes)]
    old_best_freq_idx = np.argmax(power_harmonics[:idx_upper_bound])
    old_best_key = freq_to_key(freqs[old_best_freq_idx])

    half = CHUNK_SIZE // 2
    middle = len(wavdata) // 2
    max_power2 = np.max(wavdata[middle - half:middle + half])
    best_freq_indx = np.argmax(power_harmonics[:idx_upper_bound])
    best_key = freq_to_key(best_freq)

    if not (best_key == old_best_key):
        if (old_best_key > best_key):
```

```
            print(old_best_key, best_key, chunk_num, "Moved Down!")
        else:
            print(old_best_key, best_key, chunk_num, "Moved Up!")
            # don't trust the new results when the peak goes up
            best_key = old_best_key

    length = 2 * np.pi * key_to_freq(best_key) * chunk_length_secs
    x = np.linspace(acc, acc + length, CHUNK_SIZE)
    acc += length
    chunk_volume = max_power2
    chunk_data = chunk_volume * np.sin(x)

    return key_to_freq(
        best_key), best_key, best_freq, chunk_volume, chunk_data, acc, freqs, power, power_


def compute_waves(chunk_num, f, p_l, p_c, p_n, acc):
    shared_last = (p_c / 2 + p_l / 2)
    shared_next = (p_n / 2 + p_c / 2)

    p = np.linspace(shared_last, shared_next, CHUNK_SIZE)
    length = 2 * np.pi * f * chunk_length_secs
    x = np.linspace(acc, acc + length, CHUNK_SIZE)
    acc += length
    return np.sin(x) * p, acc


num_chunks = len(myrecording) // CHUNK_SIZE

output_data = np.zeros(num_chunks * CHUNK_SIZE)

accum = 0
chunk_powers = []
chunk_freqs = []
pwr_harmonics = []
limit = num_chunks - (WINDOW_SIZE // CHUNK_SIZE) - 1
for chunk in range(0, limit):
    output_freq, output_key, measured_freq, measured_volume, output_sin, accum, freqs, powe
        chunk, myrecording, accum)
    pwr_harmonics.append(power_harmonics)
    chunk_freqs.append(output_freq)
    chunk_powers.append(abs(measured_volume))

old_chunk_freqs = np.copy(chunk_freqs)
chunk_freqs = sp.medfilt(chunk_freqs, (fs // WINDOW_SIZE) * 30 + 1)
```

```
accum = 0
for chunk in range(0, limit):
    frequency = chunk_freqs[chunk]
    next_power = 0
    last_power = 0

    current_power = chunk_powers[chunk]
    if (chunk + 1 < limit):
        next_power = chunk_powers[chunk + 1]
    if (chunk - 1 >= 0):
        last_power = chunk_powers[chunk - 1]
    sin_data, accum = compute_waves(
        chunk, frequency, last_power, current_power, next_power, accum)

    output_data[chunk * CHUNK_SIZE:(chunk + 1) * CHUNK_SIZE] = sin_data


print("Sample Rate: " + str(fs) + " Hz")
wavio.write("output.wav", output_data, fs, sampwidth=2)

chunks_shown = 0
chunk = start_chunk

output_freq, output_key, measured_freq, measured_volume, output_sin, accum, freqs, powers,
    chunk, myrecording, accum)

print("Detected freq = " + str(measured_freq))
print(chunk, output_key, measured_volume, output_freq)


fig = plt.figure(1, figsize=[8, 8.9])
plt.subplots_adjust(left=0.1, right=0.9, top=1, bottom=0.05)
plt.subplot(717)
plt.plot(output_data[:])
plt.axvline(x=(start_chunk + 0.5) * CHUNK_SIZE, color='b')

print("shape = " + str(myrecording.shape))
print("Space between frequency bins: " + str(freqs[1] - freqs[0]) + " Hz")

plt.subplot(716)
plt.yscale("log")
for note in range(16, 108, 12):
    plt.hlines(
        y=key_to_freq(note),
        color='black',
        linewidth=0.5,
```

```
        xmin=0,
        xmax=num_chunks)
plt.hlines(
    y=[key_to_freq(LOW_KEY),key_to_freq(HIGH_KEY)],
    color='r',
    linewidth=0.5,
    xmin=0,
    xmax=num_chunks)
plt.axvline(x=(start_chunk + 0.5), color='b', linewidth=0.4)
plt.plot(chunk_freqs)

plt.subplot(715)
plt.yscale("log")
for note in range(16, 108, 12):
    plt.hlines(
        y=key_to_freq(note),
        color='black',
        linewidth=0.5,
        xmin=0,
        xmax=num_chunks)
plt.hlines(
    y=[key_to_freq(LOW_KEY),key_to_freq(HIGH_KEY)],
    color='r',
    linewidth=0.5,
    xmin=0,
    xmax=num_chunks)
plt.axvline(x=(start_chunk + 0.5), color='b', linewidth=0.4)
plt.plot(old_chunk_freqs)

plt.subplot(713)
plt.xscale("log")
plt.plot(freqs[:], powers[:fs // 2])
plt.axvline(x=freqs[best_freq_indx], color='r')

for note in range(16, 108, 12):
    plt.axvline(x=key_to_freq(note), color='black', linewidth=0.5)

plt.axvline(x=key_to_freq(LOW_KEY), color='r', linewidth=0.5)
plt.axvline(x=key_to_freq(HIGH_KEY), color='r', linewidth=0.5)


plt.subplot(714)
plt.xscale("log")
plt.xlim([key_to_freq(LOW_KEY), key_to_freq(HIGH_KEY)])
plt.plot(freqs[:], power_harmonics)
for note in range(16, 108, 12):
```

```
    plt.axvline(x=key_to_freq(note), color='black', linewidth=0.5)
for peak in peaks[0]:
    plt.axvline(x=freqs[peak + idx_lower_bound],
                color='green', linewidth=1, linestyle='--')
print(peaks[0])

plt.axvline(x=key_to_freq(LOW_KEY), color='r', linewidth=0.5)
plt.axvline(x=key_to_freq(HIGH_KEY), color='r', linewidth=0.5)
plt.axvline(x=freqs[best_freq_indx], color='r')

plt.subplot(712)
plt.plot(myrecording[start_chunk *
                     CHUNK_SIZE:(start_chunk) *
                     CHUNK_SIZE +
                     WINDOW_SIZE])
plt.subplot(711)
plt.plot(myrecording)
plt.axvline(x=(start_chunk + 0.5) * CHUNK_SIZE, color='Blue')
plt.show()
```

Lighting Code used in Arduino:

```
#include <Adafruit_NeoPixel.h>
#ifdef __AVR__
#include <avr/power.h>
#endif

#define LED_DATA 12
#define POT_BRIGHTNESS A3

#define NUM_LEDS 6
#define BTN_PIN 7
#define BRIGHTNESS 10
#define MAX_MODE 7
#define DELAY 13

#define MSGEQ_OUT A6
#define STROBE 6
#define RESET 9

//MINIMUM DELAY FOR BUTTON TO TRIGGER NEXT ISR
#define ISR_DELAY 400
Adafruit_NeoPixel strip = Adafruit_NeoPixel(NUM_LEDS, LED_DATA, NEO_GRB + NEO_KHZ800);

byte neopix_gamma[] = {
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,
```

```
  1,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  2,  2,
  2,  3,  3,  3,  3,  3,  3,  3,  4,  4,  4,  4,  4,  5,  5,  5,
  5,  6,  6,  6,  6,  7,  7,  7,  7,  8,  8,  8,  9,  9,  9, 10,
  10, 10, 11, 11, 11, 12, 12, 13, 13, 13, 14, 14, 15, 15, 16, 16,
  17, 17, 18, 18, 19, 19, 20, 20, 21, 21, 22, 22, 23, 24, 24, 25,
  25, 26, 27, 27, 28, 29, 29, 30, 31, 32, 32, 33, 34, 35, 35, 36,
  37, 38, 39, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 50,
  51, 52, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 66, 67, 68,
  69, 70, 72, 73, 74, 75, 77, 78, 79, 81, 82, 83, 85, 86, 87, 89,
  90, 92, 93, 95, 96, 98, 99, 101, 102, 104, 105, 107, 109, 110, 112, 114,
  115, 117, 119, 120, 122, 124, 126, 127, 129, 131, 133, 135, 137, 138, 140, 142,
  144, 146, 148, 150, 152, 154, 156, 158, 160, 162, 164, 167, 169, 171, 173, 175,
  177, 180, 182, 184, 186, 189, 191, 193, 196, 198, 200, 203, 205, 208, 210, 213,
  215, 218, 220, 223, 225, 228, 231, 233, 236, 239, 241, 244, 247, 249, 252, 255
};

uint16_t time_since_isr=0;
boolean breakMode=false;
uint8_t mode=0;
int spectrumValue[8];
uint8_t mapValue[8];
uint32_t audioBuffer[NUM_LEDS];
int filter = 0;
uint16_t brightness;

bool countUp=true;
uint32_t lastColor=0;

void setup() {

#if defined (__AVR_ATtiny85__)
  if (F_CPU == 16000000) clock_prescale_set(clock_div_1);
#endif
  brightness = BRIGHTNESS;
  strip.setBrightness(BRIGHTNESS);
  strip.begin();
  strip.show();

  Serial.begin(9600);
  Serial.println("START");
  pinMode(MSGEQ_OUT, INPUT);
  pinMode(STROBE, OUTPUT);
  pinMode(RESET, OUTPUT);
  pinMode(BTN_PIN, INPUT);

  digitalWrite(RESET, LOW);
```

```
  digitalWrite(STROBE, HIGH);

  pinMode(POT_BRIGHTNESS, INPUT);
  // Attach interrupt to Pin 3 for Push Button to switch modes
  attachInterrupt(digitalPinToInterrupt(BTN_PIN), changeMode, RISING);
}

void loop() {
  breakMode=false;
switch(mode){
    case 0:
      Serial.println("Raindow");
      rainbowFade2White(400,255,10);
      break;
   case 1:
      Serial.println("Music");
      musicAnalyzer();
      break;
   case 2:
      Serial.println("Red");
      plainColor(255,0,0);
      break;
    case 3:
      Serial.println("Green");
      plainColor(0,255,0);
      break;
    case 4:
      Serial.println("Blue");
      plainColor(0,0,255);
      break;
   case 5:
      Serial.println("White");
      plainColor(255,255,255);
      break;
   case 6:
      Serial.println("ColorRoom");
      colorRoom();
      break;
   default:
      Serial.println("DEFAULT");
      pulseWhite(300);
      break;
  }
}

/* ISR for Button to switch modes */
```

```
void changeMode() {
  if(millis()-time_since_isr>ISR_DELAY){
    Serial.println("PRESSED");
    time_since_isr=millis();
    mode=(mode+1)%MAX_MODE;
    breakMode=true;
  }
}
/* Checks if Potentiometer value has changed, sets new Brightness and return true */
boolean checkBrightness(){
  //WS2812 takes value between 0-255
  uint16_t bright = map(constrain(analogRead(POT_BRIGHTNESS),0,1024), 0, 1024, 10, 255);
  if (abs(bright-brightness)>10){
    brightness = bright;
    strip.setBrightness(brightness);
    return true;
  }
  return false;
}

//Animation showing fading rainBow color along the strip
void rainbowFade2White(uint8_t wait, int rainbowLoops, int whiteLoops) {
  float fadeMax = 100.0;
  int fadeVal = fadeMax;
  uint32_t wheelVal;
  int redVal, greenVal, blueVal;

  for (int k = 0 ; k < rainbowLoops ; k ++) {
    for (int j = 0; j < 256; j++) {
      for (int i = 0; i < strip.numPixels(); i++) {
        wheelVal = Wheel(((i * 256 / strip.numPixels()) + j) & 255);
        redVal = red(wheelVal) * float(fadeVal / fadeMax);
        greenVal = green(wheelVal) * float(fadeVal / fadeMax);
        blueVal = blue(wheelVal) * float(fadeVal / fadeMax);
        strip.setPixelColor( i, strip.Color( redVal, greenVal, blueVal ) );
      }
      if(breakMode)
        return;
        checkBrightness();
        strip.show();
        delay(wait);
    }
  }
  delay(500);
}
```

```
// Use Brightness Potentiometer to change Color
void colorRoom(){
    strip.setBrightness(255);
    uint32_t colorNew = map(constrain(analogRead(POT_BRIGHTNESS),0,1024), 0, 1024, 0, 255);
    if (abs(colorNew-lastColor)>10){
      lastColor=colorNew;
      for (int i = 0; i < strip.numPixels(); i++)
             strip.setPixelColor(i, Wheel((colorNew) & 255));
      strip.show();
    }
    delay(10);
}

void plainWhite(){
  for (int i = 0; i < strip.numPixels(); i++) {
        strip.setPixelColor( i, strip.Color( 255, 255, 255 ) );
   }
  strip.show();
  while(!breakMode){
    if(checkBrightness())
      strip.show();
  }
}

//Shows plain color along the strip
void plainColor(uint8_t red,uint8_t green,uint8_t blue){
  for (int i = 0; i < strip.numPixels(); i++)
        strip.setPixelColor( i, strip.Color( red, green, blue ) );
  strip.show();
  while(!breakMode){
    if(checkBrightness())
      strip.show();
  }
}

void musicAnalyzer(){
  while(true){
    checkBrightness();
    if(breakMode)
      return;
    //Reset MSGEQ
    digitalWrite(RESET, HIGH);
    digitalWrite(RESET, LOW);
    delayMicroseconds(76);
    //Read all 8 Audio Bands
    for (int i = 0; i < 8; i++){
```

```
      digitalWrite(STROBE, LOW);
      delayMicroseconds(50);
      spectrumValue[i] = analogRead(MSGEQ_OUT);
      spectrumValue[i] = constrain(spectrumValue[i], filter, 1023);
      digitalWrite(STROBE, HIGH);
      mapValue[i] = map(spectrumValue[i], filter, 1023, 0, 255);
      if (mapValue[i] < 50)
        mapValue[i] = 0;
    }
      Serial.println(mapValue[0]);
      Serial.println(mapValue[2]);
      Serial.println(mapValue[4]);
    //Shift LED values forward
    for (int k = NUM_LEDS - 1; k > 0; k--) {
      audioBuffer[k] = audioBuffer[k - 1];
    }

    //Load new Audio value to first LED
    //Uses band 0,2,4 (Bass(red), Middle(green), High(blue) Frequency Band)
    //Lowest 8Bit: Blue , Middle Green , Highest Red
    //Use audioBuffer[NUM_LEDS-1] when using LED shift backwards!
    audioBuffer[0] = mapValue[0]<<16; //RED
    audioBuffer[0] |= (mapValue[2]/2)<<8;  //GREEN
    audioBuffer[0] |= mapValue[4]/4;         //BLUE

    //Send new LED values to WS2812
    for ( int i = 0; i < NUM_LEDS; i++)
      strip.setPixelColor(i, strip.Color((audioBuffer[i]>>16), (audioBuffer[i]>>8), audioBu
    strip.show();
    delay(DELAY);
  }
}

void colorWipe(uint32_t c, uint8_t wait) {
  for (uint16_t i = 0; i < strip.numPixels(); i++) {
    strip.setPixelColor(i, c);
    strip.show();
    delay(wait);
  }
}

void pulseWhite(uint8_t wait) {
  for (int j = 0; j < 256 ; j++) {
    for (uint16_t i = 0; i < strip.numPixels(); i++) {
      strip.setPixelColor(i, strip.Color(0, 0, 0, neopix_gamma[j] ) );
    }
```

```
    checkBrightness();
    if(breakMode)
      return;
    delay(wait);
    strip.show();
  }
  for (int j = 255; j >= 0 ; j--) {
    for (uint16_t i = 0; i < strip.numPixels(); i++) {
      strip.setPixelColor(i, strip.Color(0, 0, 0, neopix_gamma[j] ) );
    }
    delay(wait);
    strip.show();
  }
}




void whiteOverRainbow(uint8_t wait, uint8_t whiteSpeed, uint8_t whiteLength ) {

  if (whiteLength >= strip.numPixels()) whiteLength = strip.numPixels() - 1;

  int head = whiteLength - 1;
  int tail = 0;

  int loops = 3;
  int loopNum = 0;

  static unsigned long lastTime = 0;


  while (true) {
    for (int j = 0; j < 256; j++) {
      for (uint16_t i = 0; i < strip.numPixels(); i++) {
        if ((i >= tail && i <= head) || (tail > head && i >= tail) || (tail > head && i <=
          strip.setPixelColor(i, strip.Color(0, 0, 0, 255 ) );
        }
        else {
          strip.setPixelColor(i, Wheel(((i * 256 / strip.numPixels()) + j) & 255));
        }

      }

      if (millis() - lastTime > whiteSpeed) {
        head++;
        tail++;
```

```
        if (head == strip.numPixels()) {
          loopNum++;
        }
        lastTime = millis();
      }

      if (loopNum == loops) return;

      head %= strip.numPixels();
      tail %= strip.numPixels();
      strip.show();
      delay(wait);
    }
  }

}
void fullWhite() {

  for (uint16_t i = 0; i < strip.numPixels(); i++) {
    strip.setPixelColor(i, strip.Color(0, 0, 0, 255 ) );
  }
  strip.show();
}


// Slightly different, this makes the rainbow equally distributed throughout
void rainbowCycle(uint8_t wait) {
  uint16_t i, j;

  for (j = 0; j < 256 * 5; j++) { // 5 cycles of all colors on wheel
    for (i = 0; i < strip.numPixels(); i++) {
      strip.setPixelColor(i, Wheel(((i * 256 / strip.numPixels()) + j) & 255));
    }
    strip.show();
    delay(wait);
  }
}

void rainbow(uint8_t wait) {
  uint16_t i, j;

  for (j = 0; j < 256; j++) {
    for (i = 0; i < strip.numPixels(); i++) {
      strip.setPixelColor(i, Wheel((i + j) & 255));
    }
    strip.show();
```

```
    delay(wait);
  }
}

// Input a value 0 to 255 to get a color value.
// The colours are a transition r - g - b - back to r.
uint32_t Wheel(byte WheelPos) {
  WheelPos = 255 - WheelPos;
  if (WheelPos < 85) {
    return strip.Color(255 - WheelPos * 3, 0, WheelPos * 3, 0);
  }
  if (WheelPos < 170) {
    WheelPos -= 85;
    return strip.Color(0, WheelPos * 3, 255 - WheelPos * 3, 0);
  }
  WheelPos -= 170;
  return strip.Color(WheelPos * 3, 255 - WheelPos * 3, 0, 0);
}

uint8_t red(uint32_t c) {
  return (c >> 16);
}
uint8_t green(uint32_t c) {
  return (c >> 8);
}
uint8_t blue(uint32_t c) {
  return (c);
}
```

Audio Player Code used on DE1-SoC:

```
    import wave
    import audio
    import sys
    import math
    import time
    import struct
def play_audio(string file)
    if not audio.open_dev()
        sys.exit()
    audio.init()
    song = wave.open(file,'rb')
    audio.sampling_rate(song.getframerate())

    song = wave.open('out.wav','rb')
    volume = audio.MAX_VOLUME / 10000
    frame_num = song.getnframes()
```

```
song.setpos(0)
tmp = '<' + 'h' * framenum
aud_data = struct.unpack(tmp,song.readframes(frame_num))

for index in range(0, frame_num - 1)
    audio.wait_write()
    data = aud_data[index] * volume
    audio.write_left[data]
    audio.write_right[data]
song.close()
audio.close()
```