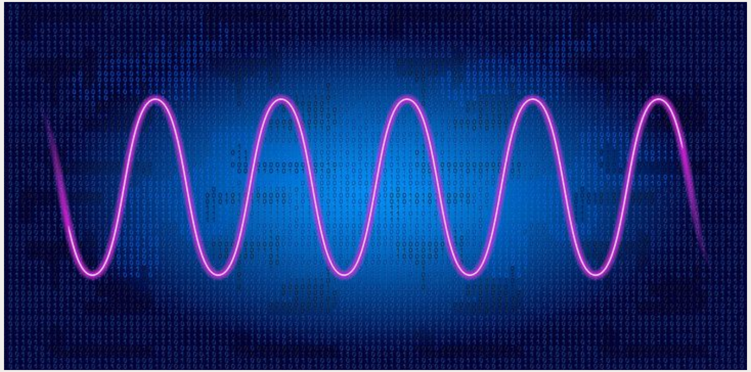
Jeremy Lee
Sean Liang
Tyler Shu

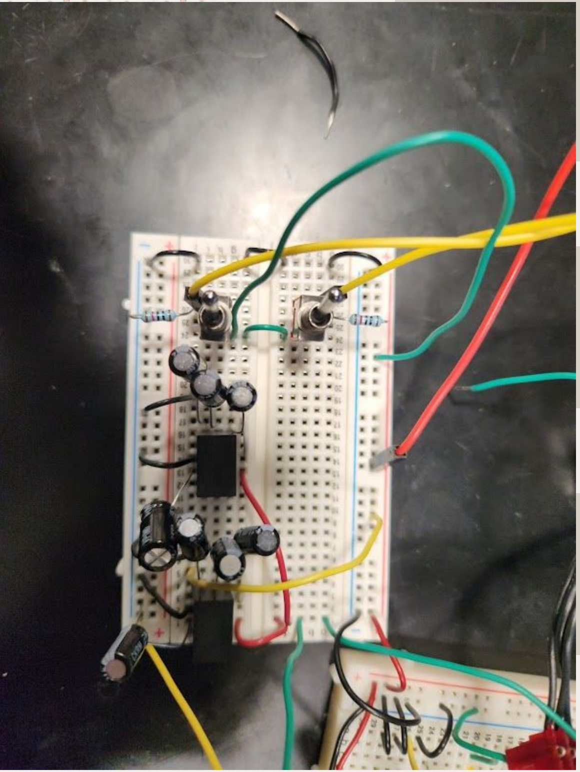


MIDI Music Box

Background and Problem

- Music can be expensive / inaccessible
- The use of MIDI controllers are very common in music production
- Cheap MIDI keyboards have no sound output
- Setting up can get complicated
- Want to make a simple plug-and-play product

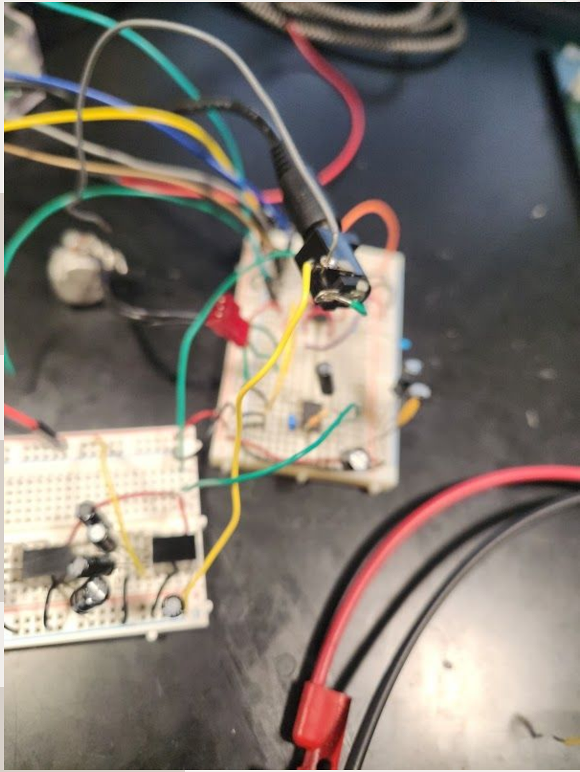
Components

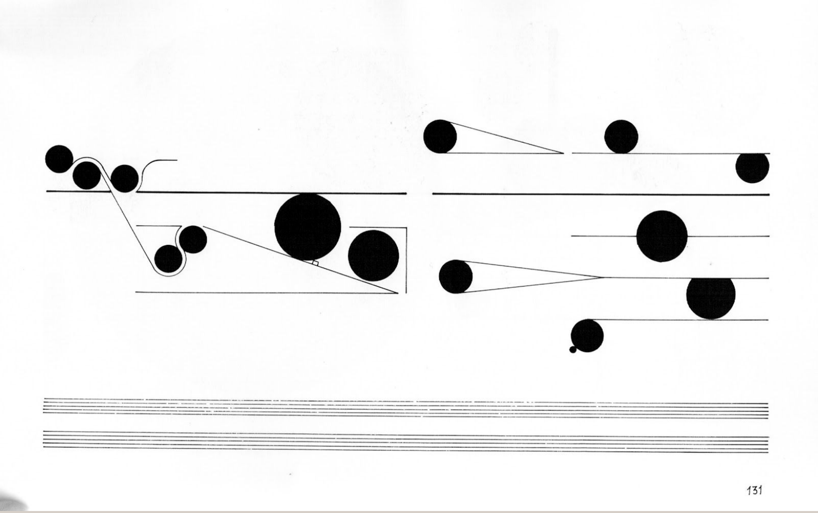


High-Level Requirements

Subsystems and Design

Functional Test Results





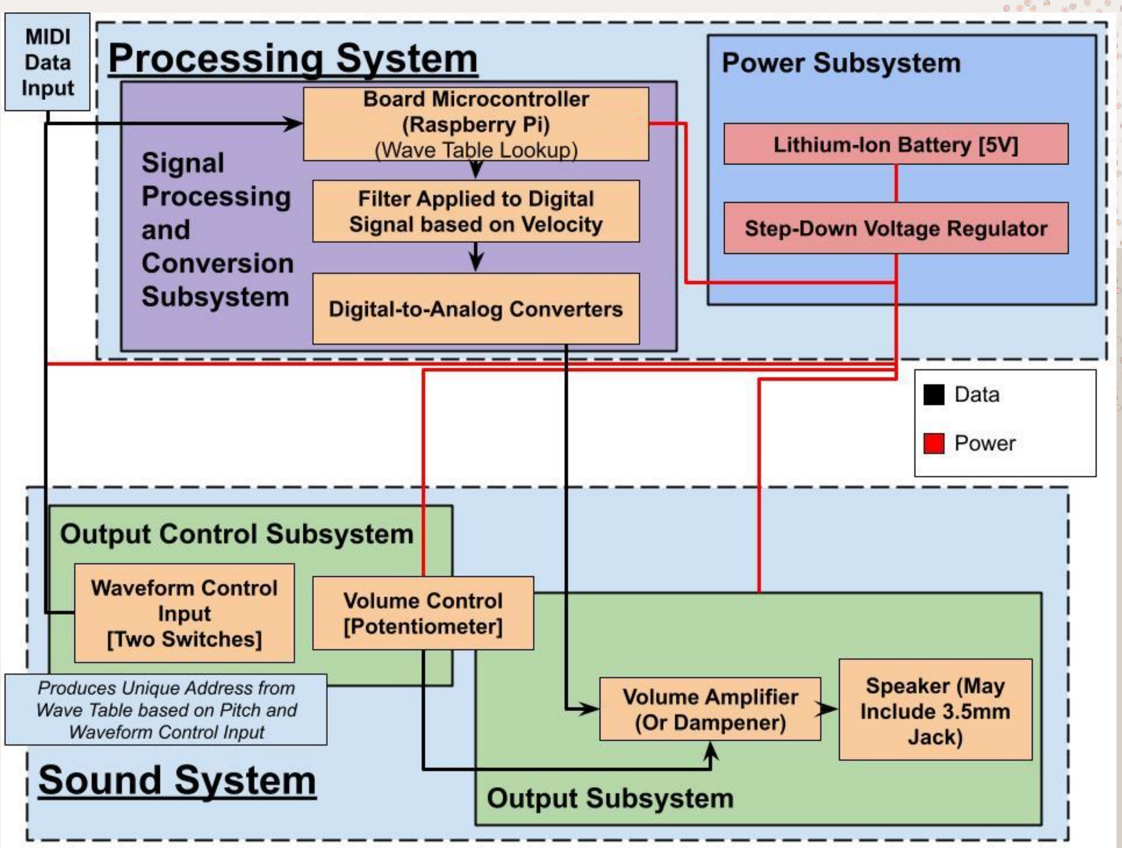
High Level Requirements

Project Goals and Requirements:

1. Synthesize **Four** different **waveforms**: **Sine**, **Square**, **Triangle**, **Sawtooth**
2. Produce **Eight Note Polyphony** - that is, **simultaneously play** 8 notes
3. Produce **Pitch** in the **Frequency Range C2(65.4Hz) - C5(525.5Hz)**, with additional capability of capability of **15kHz** with Harmonics
4. Drive a speaker utilizing up to **20W** of power

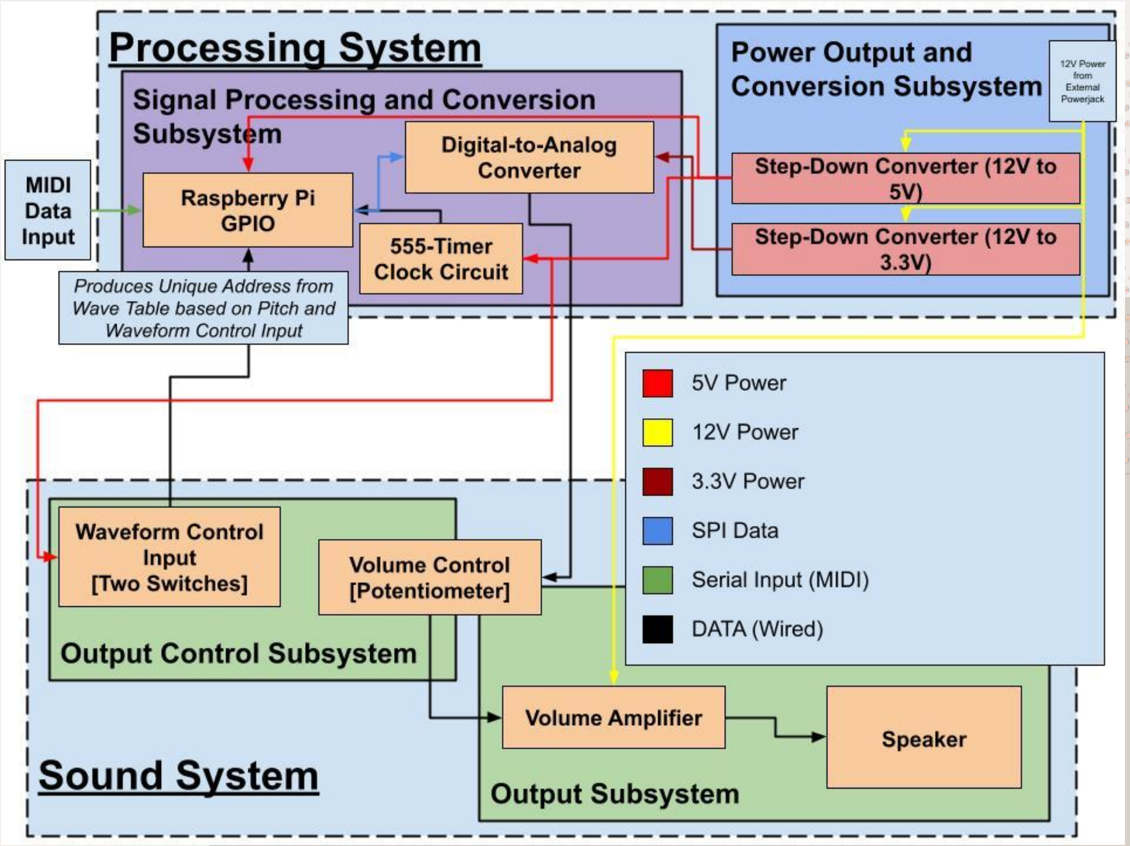
Subsystems - Original Block Diagram

- **Processing System**
  - Processes MIDI Input and convert to Analog Signal
  - Included potential filter
- **Power Subsystem**
  - Generate and Convert Voltage to appropriate levels
  - Utilize a Li-Ion battery and Step-Down regulator for 3.3V
- **Output Control Subsystem**
  - Enable User-Controlled Input for Sound Variability
- **Output Subsystem**
  - Amplify and Output modified Analog Signal



Subsystems - Final Block Diagram

- **Signal Processing and Conversion Subsystem**
  - Incorporated Timer Circuit to create consistent output
  - Power incorporated to power Raspberry Pi as opposed to Micro-USB Power Source
- **Power Output and Conversion Subsystem**
  - Incorporated DC Power Jack and Step-Down Converters
- **Output Control Subsystem**
  - No changes made
- **Output Subsystem**
  - Final output changed from 3.5mm Jack to a Speaker



Final Schematic

Components:

Raspberry Pi: RPI Model 3B+

- Reads input data from MIDI Controller
- Outputs data to DAC representing Digital Signal
- Also reads data from Switches connected via GPIO Pins

Timer IC: LM555

- Used to standardize the Sample Rate by clocking RPI code

Digital-to-Analog Converter (DAC): MCP4911

- Converts Digital Signal from Raspberry Pi to Analog Signal

Amplifier: LM386

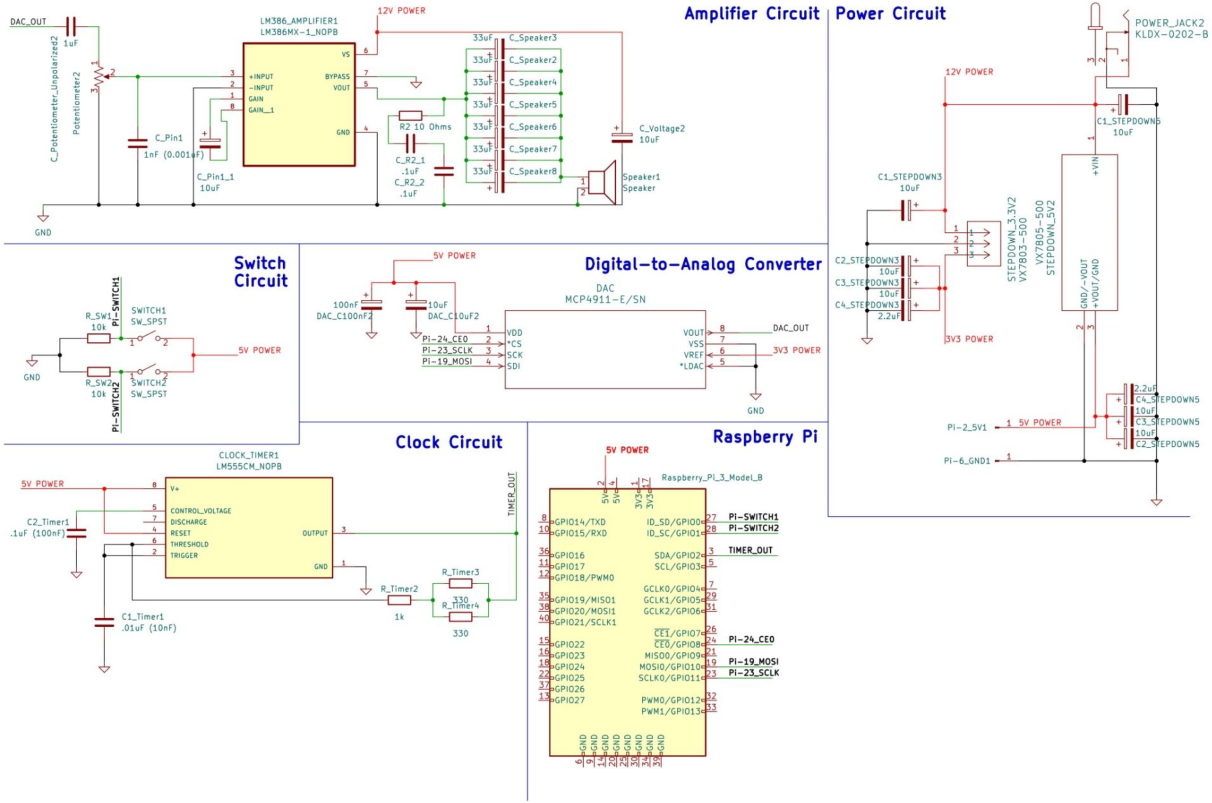
- Amplifies Analog Signals, amplifies Analog Signal from DAC

12V to 5V Step Down Converter: VX7805-500

- Provides power for majority of circuit

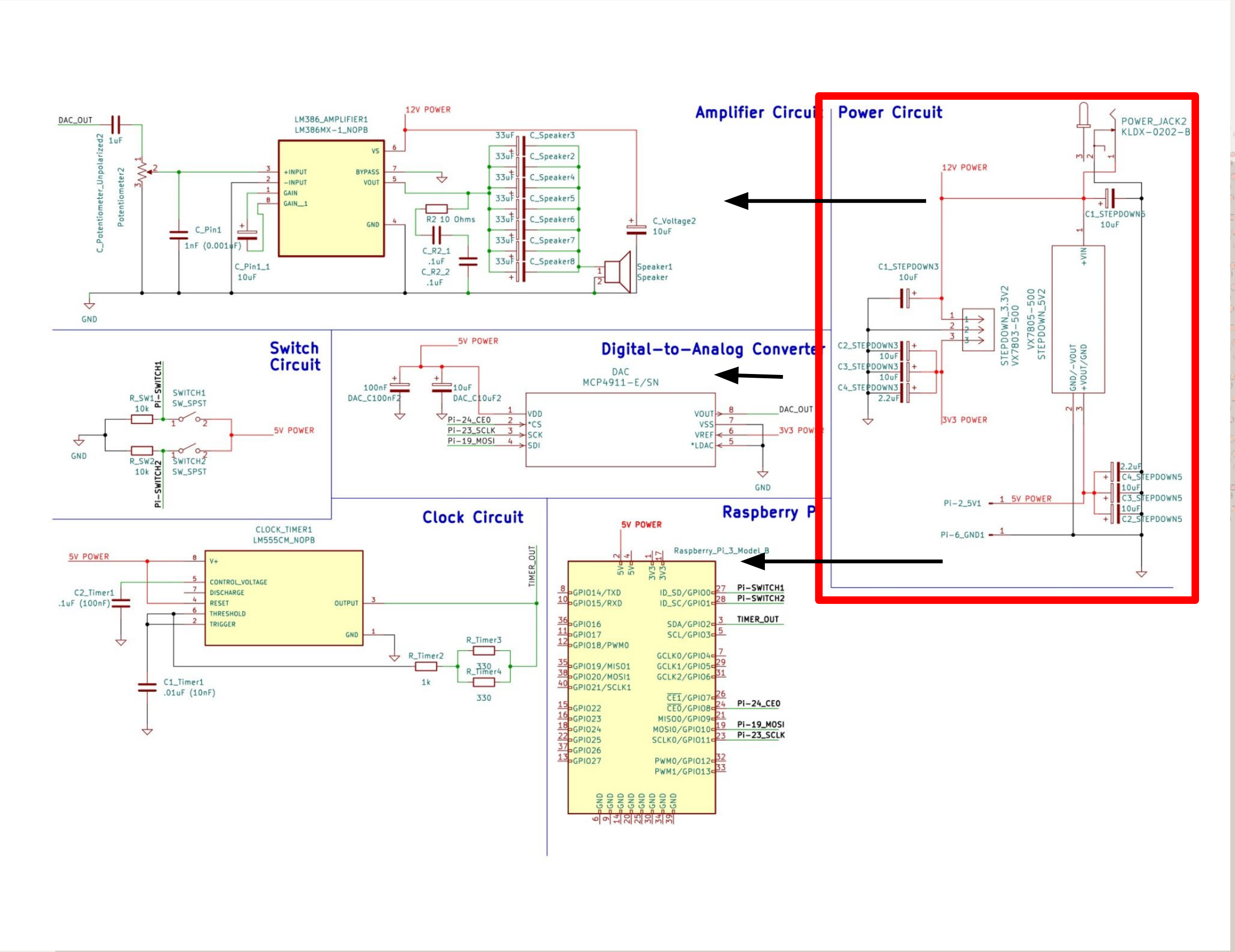
12V to 3.3V Step Down Converter: VS7803-500

- Provides reference voltage for DAC



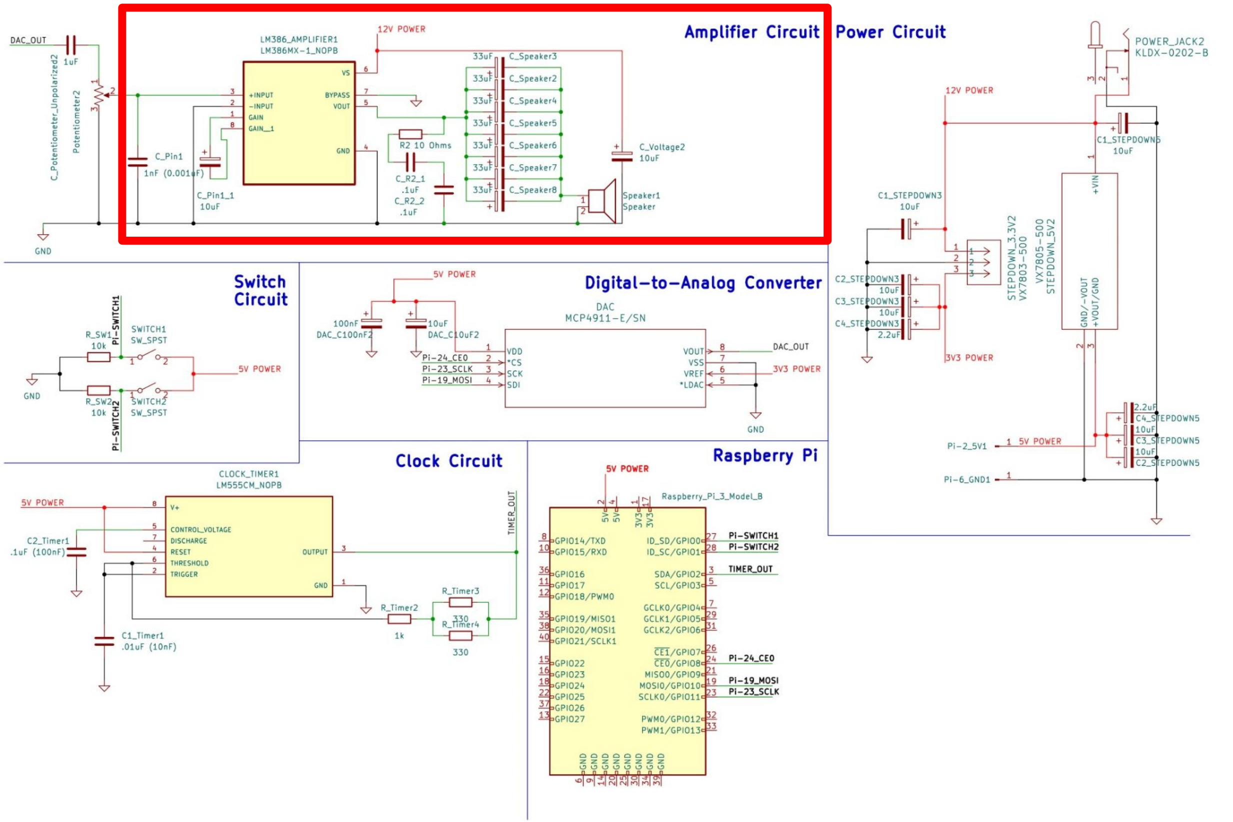
Power Subsystem

- Utilizes the Power Circuit, producing 5V lines and 3.3V lines
- 3.3V line feeds into DAC
- 5V line feeds into all other components



Output Subsystem

- Utilizes the Amplifier Circuit[1]
  - Connected to the Output Control Subsystem
  - Takes in dampened DAC output
- Amplifier Circuit contains an LM386 Amplifier
  - Amplifier responsible for output gain
- Amplifier Circuit also contains the Speaker

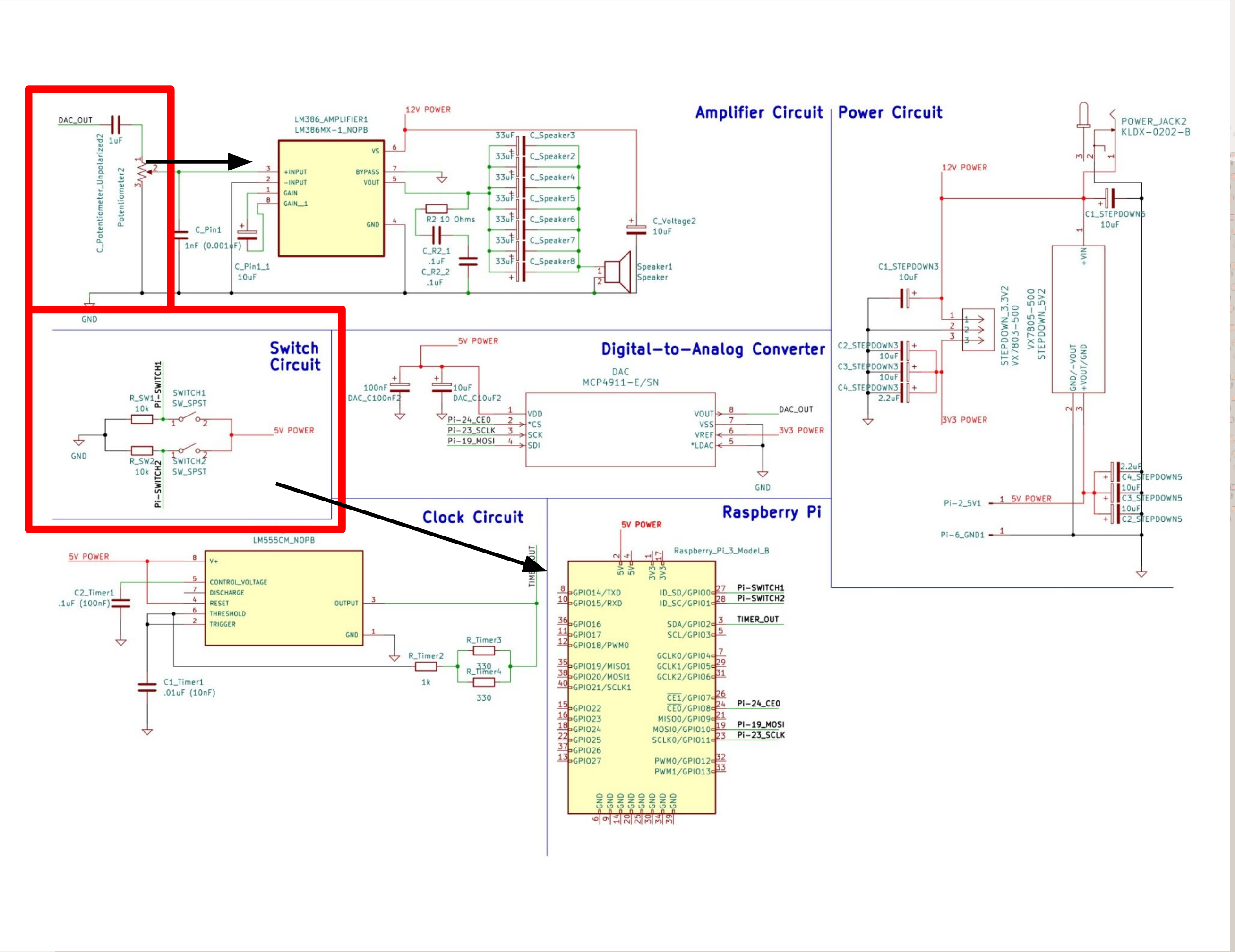


[1]Wenzel, C. (n.d.). *Audio Amplifiers*. techlib.com. http://techlib.com/electronics/audioamps.html

Output Control Subsystem

- Utilizes the Switch Circuit to control type of Waveform
  - Connected to GPIO Pins on Raspberry Pi

- Utilizes part of the Amplifier Circuit to control gain of waveform
  - Integrated into Output Subsystem
  - Originally intended to be part of Output Control Subsystem, but was built into Amplifier Circuit
  - Takes input from the DAC, outputs to Output Subsystem



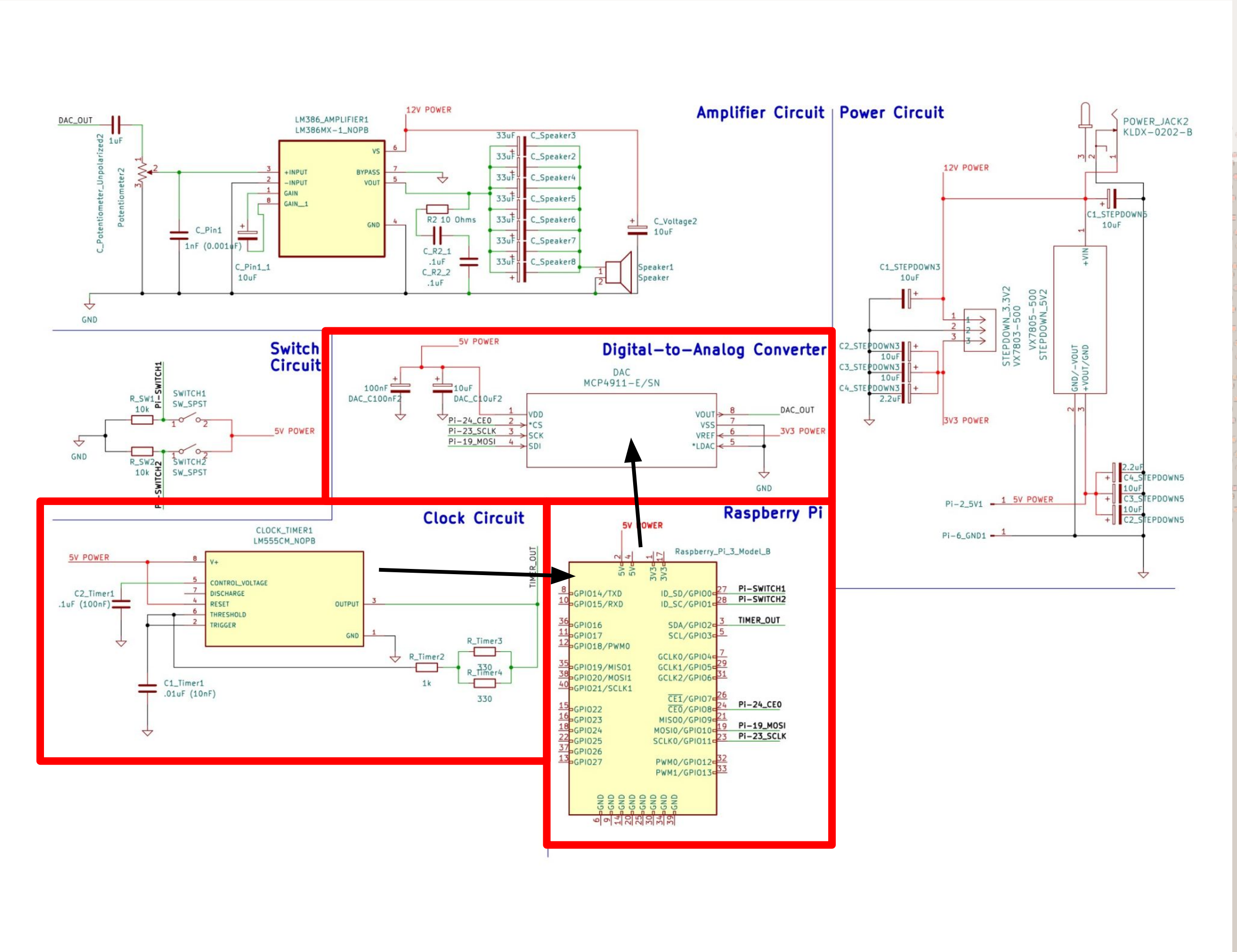
Signal Processing and Conversion Subsystem

- Utilizes the Raspberry Pi, DAC[1], and Clock Circuit

- Clock Circuit feeds into Raspberry Pi to provide constant external clocking

- Raspberry Pi connected to DAC to provide Digital Data for conversion

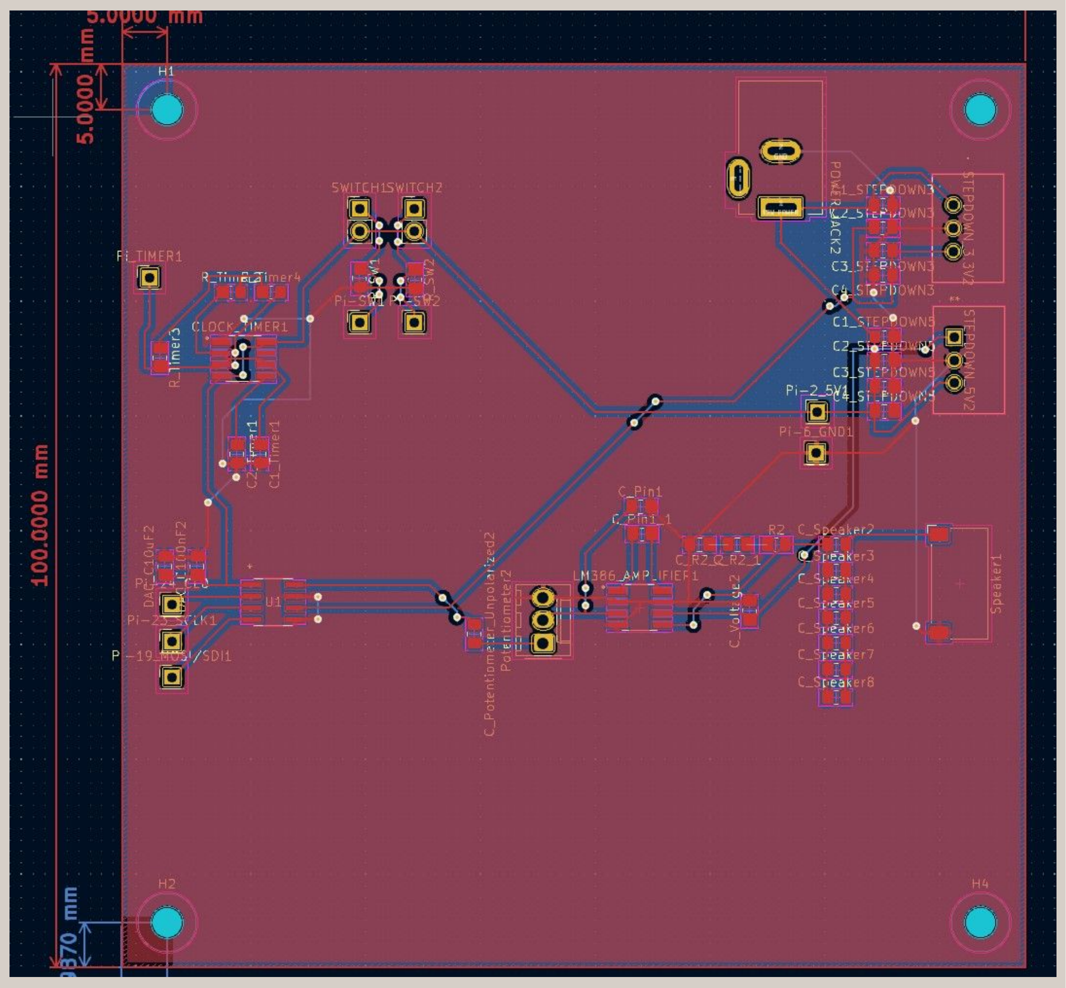
- DAC connected to Amplifier Circuit



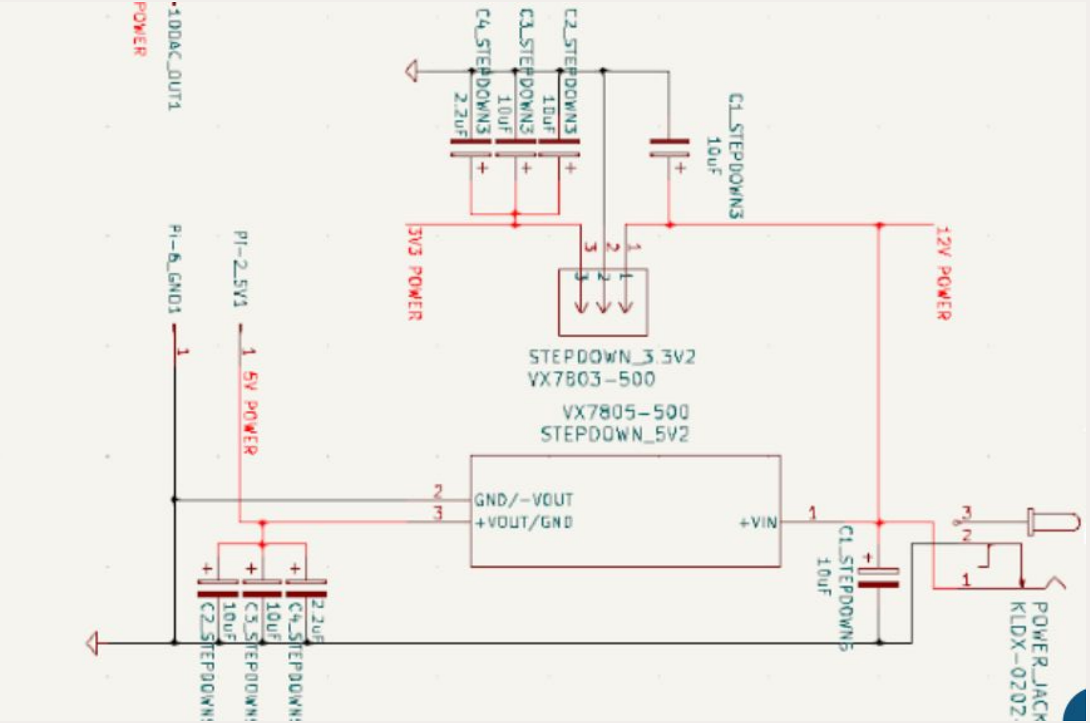
[1]paulv. (2015, October 25). *Add an analog output to the Pi (DAC)*. Raspberry Pi Forums. https://forums.raspberrypi.com//viewtopic.php?f=37&t=124184

PCB Design

Size Standardized for Container Dimensions



Design - Power Output and Conversion Subsystem



Power Output and Conversion Subsystem Timeline

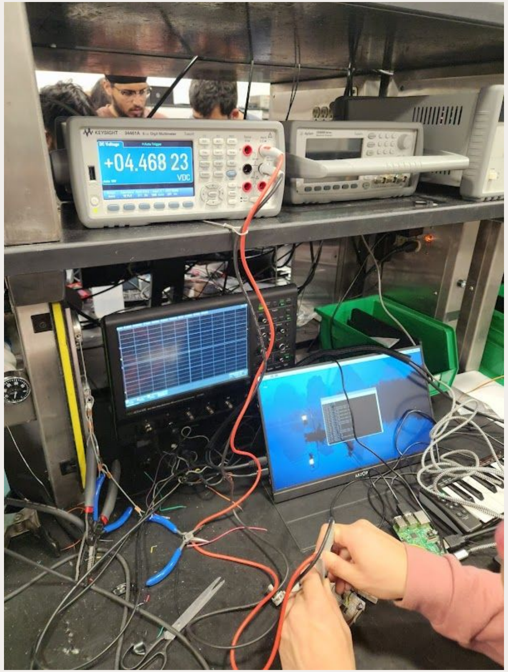
Initial Power Subsystem Stage

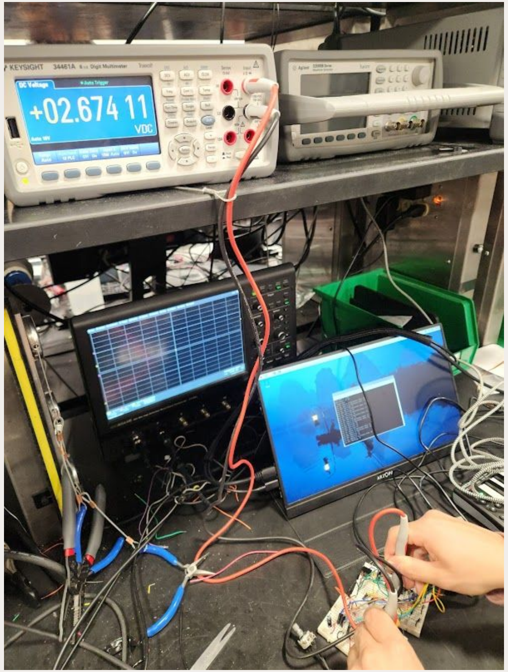
- Amplifier powered by lab power supply
- DAC was powered by Raspberry PI GPIO pins

Final Power Subsystem Stage

- Amplifier powered by 12V power jack
- 12V to 5V DC-DC converter output powers Raspberry PI and DAC
- 12V to 3.3V DC-DC converter output used for reference voltage for the DAC

Results - Power Output and Conversion Subsystem





Verification of 5V Power Supply Using Multimeter

Target Range: 4.85V - 5.15V

- Failed to consistently hit target voltage, but came close
- Power issues caused by inconsistent and insufficient voltage lead to issue powering the Raspberry Pi
- New issue, issue arose after break

Verification of 3.3V Power Supply using Multimeter

Target Range: 3.15V - 3.45V

- Failed to consistently hit target voltage, issue may be with booster or other load factors

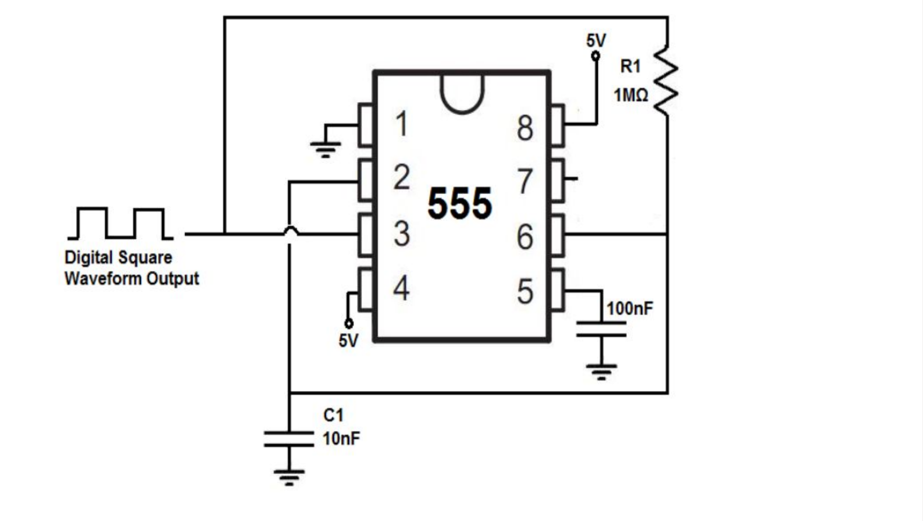
Clock Design: Sample Rate

Problem

- Naive approach to read MIDI data and output digital data to the DAC resulted in a change in the sample rate depending on the amount of notes pressed
- No set sampling frequency

Solution

- Create a simple square wave circuit of a desired frequency
- Used LM555, capacitors and a resistor
- Adjusted resistance to get desired clock frequency



To create a 6Hz signal, R_1= 10MΩ and C= 10nF.

To create a 600Hz signal, R_1= 100KΩ and C= 10nF.

To create a 134Hz signal, R_1= 470KΩ and C= 10nF.

To create a 1.7KHz signal, R_1= 33KΩ and C= 10nF.

To create a 43KHz signal, R_1= 1KΩ and C= 10nF.

To create a 180KHz signal, R_1= 150Ω and C= 10nF.

To create a 252KHz signal, R_1= 100Ω and C= 10nF.

How to build a clock circuit with a 555 timer. (n.d.). https://www.learningaboutelectronics.com/Articles/555-timer-clock-circuit.php

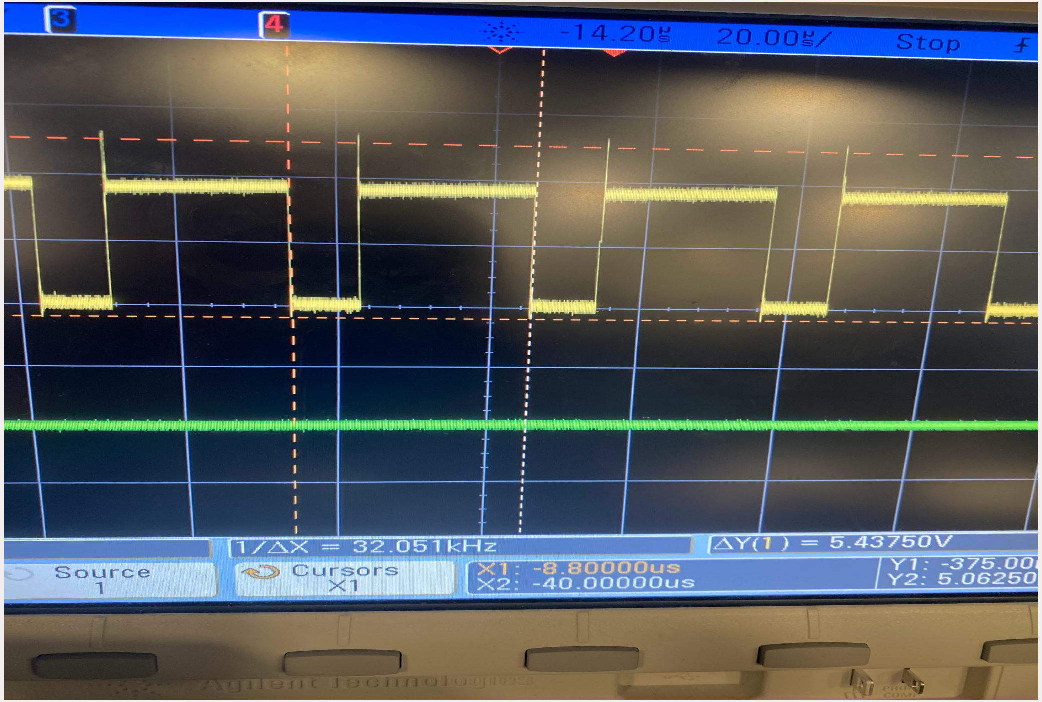
Clock Design - Results

Results

- Able to create a square wave of our desired sampling frequency of 32 khz with a resistor value of 1220 ohms

Future Problems

- Duty Ratio might be a problem in calculating the sampling rate for our program



MIDI Data: An Overview

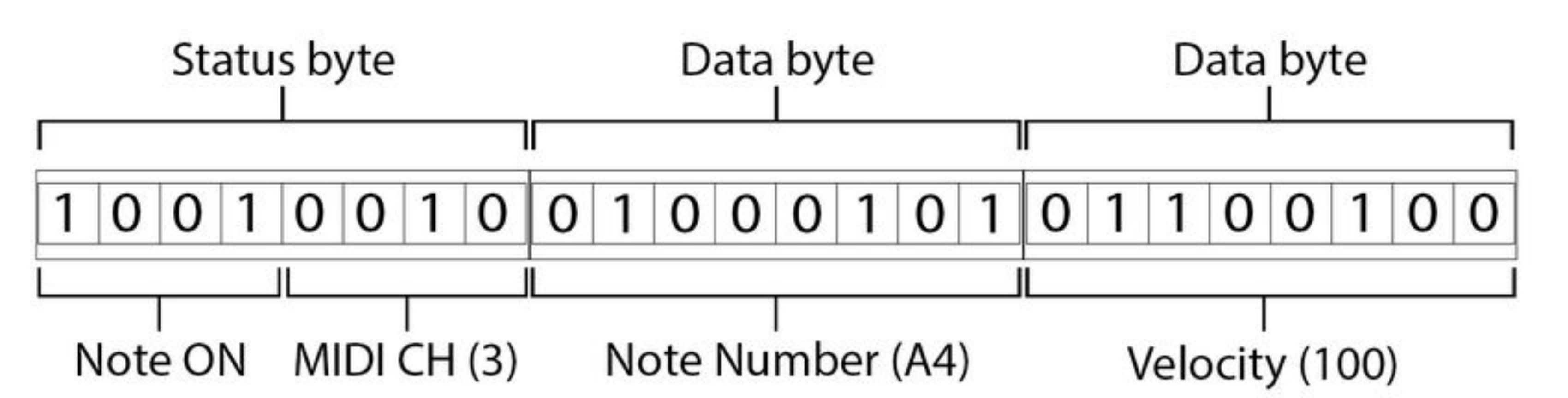
- Musical Instrument Digital Interface (MIDI) is a standard for both transmitting and storing music.
- Data is sent serially
- MIDI data itself is *not* audio
- Data consists of MIDI messages: one status byte followed by up to two data bytes

Status	Explanation	Msg Size	Byte 1	Byte 2
0x8c	Note Off	2	pitch	velocity
0x9c	Note On	2	pitch	velocity
0xAc	Key Pressure	2	key	pressure
0xBc	Controller Change	2	controller	value
0xCc	Program Change	1	preset	
0xDc	Channel Pressure	1	pressure	
0xEc	Pitch Bend	2	bend LSB	bend MSB
0xF0	System Exclusive	*n*	vendor ID	anything
0xF2	Song Position	2	position LSB	position MSB
0xF3	Song Select	1	song number	
0xF5	*Unofficial Bus Select*	*1*	*bus number*	
0xF6	Tune Request	0		
0xF7	End of SysEx	0		
0xF8	Timing Tick	0		
0xFA	Start Song	0		
0xFB	Continue Song	0		
0xFC	Stop Song	0		
0xFE	Active Sensing	0		
0xFF	System Reset	0		

- A status byte always starts with a 1, while a data byte always starts with a 0
- For our project, we focused on two messages: note on & note off
- Note on is sent at the start of a note press and note off is sent at the release

MIDI Data: An Overview (con’t)

- For note on & note off messages, the two data bytes contain information about the pitch and the velocity (how “hard” a note is played)
- We don’t use any MIDI channels for our project
- A note on with a velocity of 0 is equivalent to a note off



Python Design: An Overview

- General idea: read MIDI messages sent from the controller, specifically NOTE ON / NOTE OFF
- Based on the notes being played, calculate the output value and send the data to the DAC

Python Design: RtMidi

- For reading MIDI data, we used the PyRtMidi library (based on RtMidi for C++)
- This allowed the Pi to detect the MIDI controller and parse MIDI messages
- We only read input MIDI data, the program doesn't need to output any MIDI data

```
ports = range(midiin.getPortCount())
if ports:
    for i in ports:
        print(i)
        print(midiin.getPortName(i))
    print("Opening port 1!")
    midiin.openPort(1)
    while True:
        midi = midiin.getMessage(0)
        # code that should run once every note on / note off. in other words any time a MIDI message is received
        if midi:
            midi_init(midi)

        #code that should run every sampling period. NOT FULLY IMPLEMENTED YET: runs in the while true loop, freq unknown
        if len(note_samples) == 0: # if there are no notes, the output is 0
            output = 0
        else: # calculate output from each note
            for note in phase_incr: # calculate phases for the current sample.
                phase[note] += phase_incr[note] #increase the phase of each note by the correct phase increment
                phase[note] = phase[note] % wt_resolution #cycles through wavetable if needed
            for note in note_samples:
                note_samples[note] = sine[round(phase[note])] #add values for all notes to calculate output
            output = sum(note_samples.values()) #value to output to DAC
        print(output)
```

Python Design: RtMidi

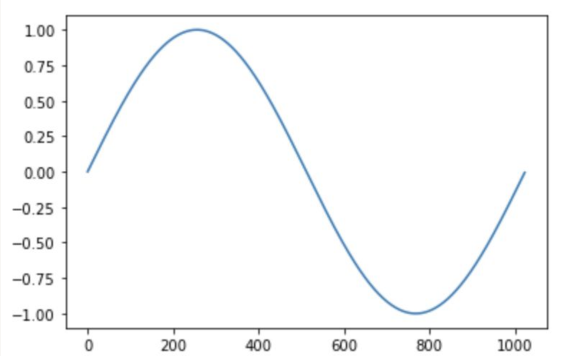
- Once the program detects a MIDI message, it checks for note on or note off messages
- For a note on message, the program does some calculations for the output value of the note
- For a note off, the program clears the output value of the note

```
note_samples = {} # sample values for each separate note
phase = {} # current phase (sample number) of notes
phase_incr = {} # phase increment for each note

def midi_init(midi): #initialize values for a note. phase & value initialized to 0. phase increment is calculated here
    if midi.isNoteOn():
        global polyphony
        polyphony += 1
        frequency = 440 * (2 ** ((midi.getNoteNumber() - 69) / 12)) # calculate frequency (Hz) from MIDI note number
        phase[midi.getMidiNoteName(midi.getNoteNumber())] = 0
        phase_incr[midi.getMidiNoteName(midi.getNoteNumber())] = wt_resolution * frequency / sample_rate # how many samples needed to jump in wavetbale
        note_samples[midi.getMidiNoteName(midi.getNoteNumber())] = 0
        #print('ON: ', midi.getMidiNoteName(midi.getNoteNumber()), midi.getVelocity(), phase[midi.getMidiNoteName(midi.getNoteNumber())])
        #print('Note ON:', polyphony, 'note(s)', midi.getMidiNoteName(midi.getNoteNumber()), frequency, 'Hz, phase increment:', phase_incr)
        #print(note_samples.keys(), 'Output: ', output)
    elif midi.isNoteOff(): #deletes dictionary key:value for the note turned off.
        polyphony -= 1
        #print('Note OFF:', polyphony, 'note(s)', midi.getMidiNoteName(midi.getNoteNumber()))
        del note_samples[midi.getMidiNoteName(midi.getNoteNumber())] #clear sample value when note is turned off
        del phase[midi.getMidiNoteName(midi.getNoteNumber())] # clear phase calculation
        del phase_incr[midi.getMidiNoteName(midi.getNoteNumber())] # clear phase increment
    elif midi.isController():
        print('CONTROLLER ', midi.getControlNumber(), midi.getControlValue())
```

Python Design: Wave Table

- For calculating the output samples, we use the same technique used in **wavetable synthesis**
- One period of a wave is stored in memory (we use 1024 samples, generally the number of samples is a power of 2)
- At a given sample, the output value is one sample of the wavetable.
- The current position of the wavetable can be thought of as the **phase**.
- Every sample, the phase of the wavetable increments by a certain amount.
- The increment depends on the table size, frequency of the note, and playback rate.
- Our wavetables were initialized in python as arrays.



Python Design: Wave Table

- To implement this in python, we used dictionaries for each note's output value, current phase, and phase increment
- The key for the dictionaries are the note numbers, so each note has one key:pair value
- The phase increment is constant for each note and only needs to be calculated once
- The frequency of a note is $440 \bullet 2^{(nn - 69 / 12)}$ where nn is the note number (from the MIDI message)
- The phase increment is the frequency of the note multiplied by the number of samples in the wavetable, divided by the playback rate
- Each sample, the phase increment is added to the phase, and the wavetable value at the phase is set.
- The final output is the sum of each individual note's output

```
for note in phase_incr: # calculate phases for the current sample.
    phase[note] += phase_incr[note] #increase the phase of each note by the correct phase increment
    phase[note] = phase[note] % wt_resolution #cycles through wavetable if needed
for note in note_samples:
    note_samples[note] = sine[round(phase[note])] #add values for all notes to calculate output
output = sum(note_samples.values()) #value to output to DAC
```

C++ Design: An Faster Solution

- Python Implementation too slow, extremely limited sample rate caused frequency range to be too small
- Ported program over to C++
  - RtMidi library for Python based on same library for C++, program needed to be modified accordingly
  - SPI Library for Python not available in C++, researched and used the WiringPi library which included an SPI sub-library
- General idea for program stayed the same
- Fun Fact: C++ is anywhere from 10x to 100x faster than Python, depending on context![1]

[1]Bales, R. (2023, May 20). *C++ vs. python: Full comparison*. History-Computer. https://history-computer.com/c-vs-python-2/

C++ Design: Initial Computations

Table Computation based on Sample Rate

- Similar to Python Approach, tables are pre-computed
- One table of 1024 samples for each wave computed
- Other computations to relieve computational load later in the program:
  - Frequency for each note number computed
  - Phase Increment for each note number computed

Sample Rate Calculation

- Sample rate calculated at the beginning of operations to evaluate timer/clock-circuit performance
- Evaluated using nanosecond precision timing, counts predetermined number of clock cycles

```
// Objects to hold values relevant to the audio
double frequencies[NUM_POSSIBLE_NOTES];
double phase_increments[NUM_POSSIBLE_NOTES];
```

```
void initTables( void )
{
  calculateFrequencies( );
  calculatePhaseIncrements( );
  calculateWaveforms( );
}
```

```
double sineWave[ 1024 ];
double squareWave[ 1024 ];
double triangleWave[ 1024 ];
double sawtoothWave[ 1024 ];
```

```
// Adjusts sample rate based on clock rate. Reads a single clock cycle
// and sets based on calculation.
void setSampleRate( struct timespec& startTime, struct timespec& endTime )
{
  int count = 0;

  clock_gettime(CLOCK_MONOTONIC_RAW, &startTime);

  while(count < 100 )
  {
    while( digitalRead( CLOCK_PIN ) == LOW ) {}
    while (digitalRead( CLOCK_PIN ) == HIGH ){}
    count++;
  }

  clock_gettime(CLOCK_MONOTONIC_RAW, &endTime);

  // Calculate elapsed time in nanoseconds
  double elapsedTime = ( endTime.tv_sec - startTime.tv_sec ) * 1e9 + (endTime.tv_nsec - startTime.tv_nsec);
  elapsedTime = elapsedTime / 100;

  double frequency = 1.0 / (elapsedTime * 1e-9);

  CLOCK_RATE = (int)frequency;
  SAMPLE_RATE = CLOCK_RATE;
  std::cout << "Clock Rate is:" << CLOCK_RATE << std::endl;

}
```

C++ Design: GPIO and SPI Setup

GPIO Initialization

- GPIO Pins Setup using WiringPi library
- Pin Numbers defined by WiringPi Convention assigned based on constants set at the top of the code
- Initial Setup of specific pins required

SPI Initialization

- SPI Protocol Initialization using WiringPi sub-library, WiringPiSPI
- Channel and Speed specified for initialization

```
// Setup Wiring Pi and SPI
void setupGPIOAndSPI( void )
{
  int setup;

  // Setup SPI Port
  std::cout << "Setting up SPI..." << std::endl;
  setup = wiringPiSPISetup( SPI_CHANNEL, SPI_SPEED );
  std::cout << "SPI set up with Channel " << SPI_CHANNEL << " and Speed " << SPI_SPEED << std::endl;

  // Set up GPIO Pin for Clocking
  std::cout << "Setting up Clock Pin and Switch Pins..." << std::endl;
  setup = wiringPiSetup();
  pinMode( CLOCK_PIN, INPUT );
  pinMode( SW_PIN_1, INPUT);
  pinMode( SW_PIN_2, INPUT);
  std::cout << "Clock Pin set up using WiringPi Pin " << CLOCK_PIN << std::endl;
  std::cout << "Please note that WiringPi Numbering Convention differs from RPI's.\n" << std::endl;
  std::cout << "Switch Pins set up using WiringPi Pins " << SW_PIN_1 << " and " << SW_PIN_2 << std::endl;
  std::cout << "GPIO and SPI Setup Complete!" << std::endl;

  return;
} // EoF
```

C++ Design: Receiving MIDI Data

Utilizing the RtMidi Library

- C++ code utilizes the RtMidi library to handle receiving MIDI data from the USB Ports
- Initializes a RtMidiIn object, Opens a Port for reading
- Callback Function for Interrupt-based approach ignored
  - Initial implementations used Interrupt method but encountered major errors with segmentation faults created by interrupt approach

```
RtMidiIn* setupMIDI_Polling( void )
{
  // Set up a signal handler to ensure that 'done' is set to true when the user presses Ctrl+C,
  // So that the program can terminate the while loop and end gracefully
  (void)signal(SIGINT, finish);

  RtMidiIn *midiin = nullptr;

  // RtmidiIn Constructor
  try
  {
    midiin = new RtMidiIn();
  }
  catch(RtMidiError &error) ···

  // Check available ports
  unsigned int nPorts = midiin->getPortCount();
  if(nPorts == 0) ···

  // Open the first available port
  midiin->openPort(1);

  // Don't ignore sysex, timing, or active sensing messages
  midiin->ignoreTypes(false, false, false);

  // Install the function defined above as the callback. This function will be called
  // whenever there is MIDI data to be read on the port.
  // Note that we don't actually want to use a callback (explained in midiMain). We want
  // to use a polling method instead, so we forgo passing in a callback function.
  //midiin->setCallback(&MIDICallback);

  std::cout << "MIDI Protcol Set Up!" << std::endl;

  return midiin;

} // Eof setupMIDI_Polling
```

C++ Design: Reading the Buffer

Reading the Available Message

- Program checks the associated port for data in its buffer
  - Avoids the Interrupt-based approach. Previous implementation using such would cause a Segmentation Fault due to memory access of removed entries when interrupt occurs during mathematical operations
- Processes any new data/messages from the MIDI Controller
- Messages consist of NOTE ON or NOTE OFF messages
  - First Byte contains Note Number
  - Second Byte contains Note Velocity (Keypress Intensity)
- NOTE ON Message: Note added to map/dictionary of “ON NOTES”
- NOTE OFF Message: Note removed from map/dictionary of “ON NOTES”

```
// Get midi message
std::vector<unsigned char> message;
midiin->getMessage(&message);
// getMessage will return an empty vector if no message
if(!message.empty())
{
  // Message received, process it.
  unsigned int nBytes = message.size();
  int messageByte;

  for(unsigned int i = 0; i < nBytes; i++)
  {
    messageByte = (int)message.at(i);
  }
  // Check for Note On or Note Off
  // message->at(1) is the note number
  // message->at(2) is the velocity
  // A velocity of 0 is actually a Note Off message, so we check for that
  if(message.at(2) > 0)
  {
    // Note On -> Check if exceeded Polyphony. If not, add note to
    // dictionary. If so, do nothing.
    if( phase.size() < 8 )
    {
      // Add note to dictionary
      phase[message.at(1)] = 0;
    }
  }
  // Note Off -> Remove from dictionary. We expect the note to exist since it
  // must have been added previously, so we don't have to check for the key's
  // existence in the dictionary.
  else
  {
    // Remove key-value pair from dictionary
    auto itToRemove = phase.find(message.at(1));
    if(itToRemove != phase.end())
    {
      phase.erase(message.at(1));
    }
    else
    {
```

C++ Design: Determining the Waveform

Checking the GPIO Pins and Switches

- Switch status checked upon each iteration using the WiringPi Library
- Pointer re-assigned based on target waveform

```cpp
int checkSwitchPins()
{
  int switch1 = digitalRead( SW_PIN_1 );
  int switch2 = digitalRead( SW_PIN_2 );

  if( switch1 == LOW && switch2 == LOW )
  {
    return 0;
  }
  else if( switch1 == LOW && switch2 == HIGH )
  {
    return 1;
  }
  else if( switch1 == HIGH && switch2 == LOW )
  {
    return 2;
  }
  else if( switch1 == HIGH && switch2 == HIGH )
  {
    return 3;
  }

  return -1;

}
```

Switch 1	Switch 2	Wave Produced
0	0	Square
0	1	Sine
1	0	Triangle
1	1	Sawtooth

```cpp
wave = checkSwitchPins( );
if( wave == 0 )
{
  //std::cout << "Sine Wave!" << std::endl;
  arrayPtr = sineWave;
}
else if( wave == 1 )
{
  //std::cout << "Square Wave!" << std::endl;
  arrayPtr = squareWave;
}
else if( wave == 2 )
{
  //std::cout << "Triangle Wave!" << std::endl;
  arrayPtr = triangleWave;
}
else if( wave == 3 )
{
  //std::cout << "Sawtooth Wave!" << std::endl;
  arrayPtr = sawtoothWave;
}
else
{
  //std::cout << "Unknown Wave. Check for errors!" << std::endl;
  arrayPtr = sineWave;
}
```

C++ Design: DAC Output

```cpp
    double sum = 0;

    // Increment phase of values in phase dict
    for(const auto& pair: phase)
    {
      int key = pair.first;
      phase[ key ] += phase_increments[ key - NOTE_NUM_OFFSET ];
      phase[ key ] = fmod( phase[ key ], WT_RESOLUTION - 1 );

      sum +=  ( arrayPtr[ (int)phase[ key ] ] + 1 ) * 64;

    }

    // In some cases, sum can exceed maximum (for 10-bit, 1024).
    // Check for this specific cases and reduce if needed.
    if( sum > 1024 )
    {
      sum = 1023;
    }


    // Output to DAC. Convert to Integer to do so.
    int output = (int)sum;
    outputToDAC( output, 10 );
```

Incrementing the Phase

With pre-calculated frequencies and phase increments, the phase of each wave in the dictionary is added, summed up into one output wave/value

```cpp
void outputToDAC( int value, int bits )
{
  // Divide value into bytes. Value passed in is integer we want
  // to output to the DAC, preprocessed.

  // Configured for 10 bit DAC
  if( bits == 10 )
  {
    char lowByte;
    char highByte;
    char output[3];
    int result;

    // Value is up to 1024, we can assume is within first 10 bits.
    // Low byte has 6 data bits, with the last two bits unused
    // B7=0:write to DAC, B6=0:unbuffered
    // B5=1:Gain=1X, B4=1:Output is Active
    lowByte = 0b00000000;
    lowByte |= (value << 2);
    lowByte &= 0b11111100;

    // High byte has 4 data bits, with first four bits being control
    highByte = 0b00110000;
    highByte |= (value >> 6);

    // Put into output, we need to pass in as reference
    output[0] = highByte;
    output[1] = lowByte;

    // Now call the SPI Function to output
    result = wiringPiSPIDataRW( SPI_CHANNEL, reinterpret_cast<unsigned char*>(output), 2 );

  }

} // EoF
```

Following the SPI Protocol

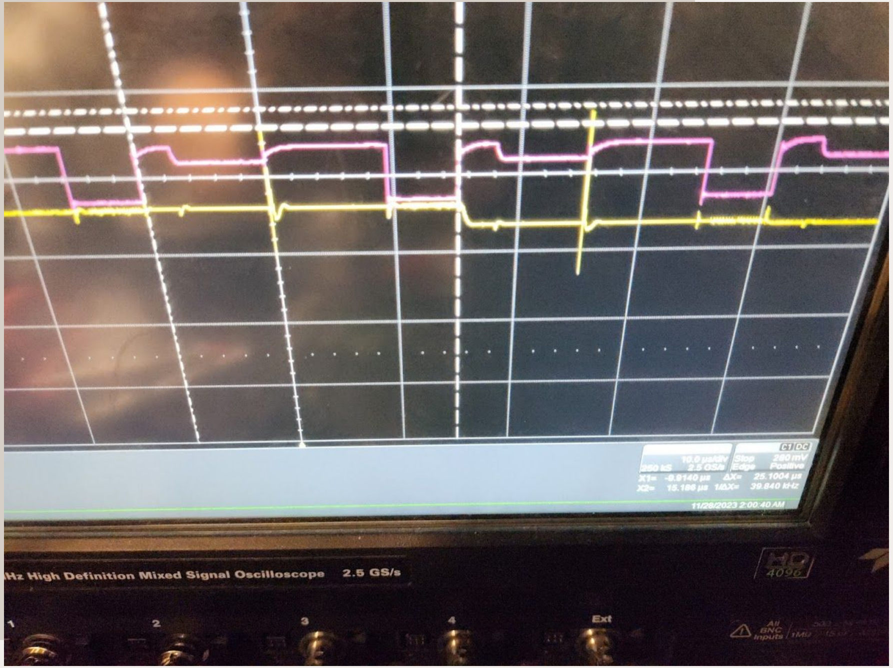
The output value is then masked and formatted appropriately following the SPI Protocol, and output to the DAC using the WiringPi library. The function is currently configured to output 10-bit output to the DAC, which includes 4 Control Bits at the beginning

Requirements and Verification - Signal Processing and Conversion Subsystem

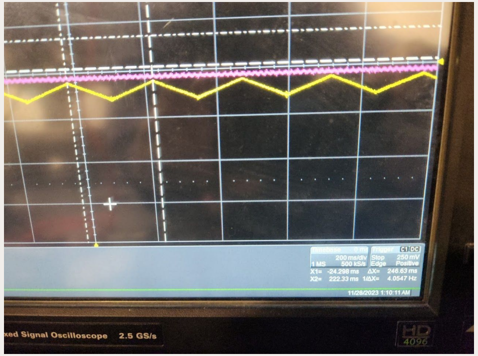
The following Requirements and Verification table was generated to evaluate the subsystem's performance and provide a goal for functionality. Functional test results are provided in the following slides.

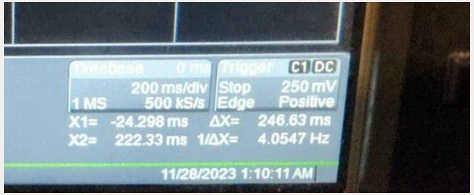
Requirements	Verification
- The Raspberry Pi must be able to read Serial input from its serial ports utilizing the MIDI protocol, at the rate determined by the protocol (31250 bits per second) - The DAC must contain a resolution of a minimum of 10-bits - The DAC must be able to output waveforms with frequencies within the target range, up to 15KHz - The DAC must be able to produce 4 different waveforms (Sine, Square, Triangle, Sawtooth)	- Verify Serial reading by passing in test input with predefined waveform, and verifying based on output audio - Utilize all bits of DAC Components capable of 10-bits. Evaluate based on waveform clarity with Oscilloscope - Verify DAC Frequency Range and waveform shape using Oscilloscope and test input

Test Results - Signal Processing and Conversion Subsystem



Maximum frequency evaluated by measuring the Sample Rate of the DAC



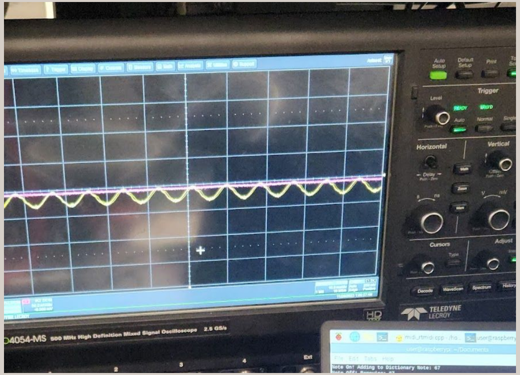


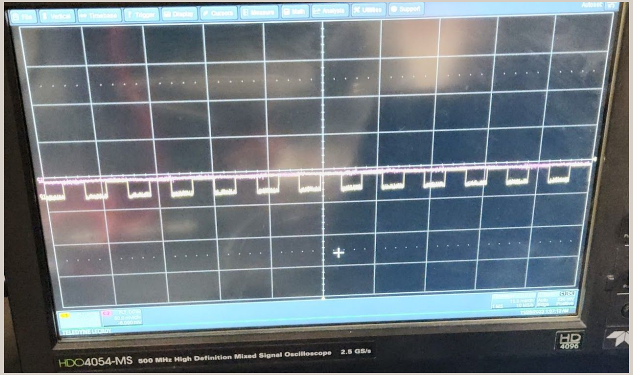
DAC Frequency Range

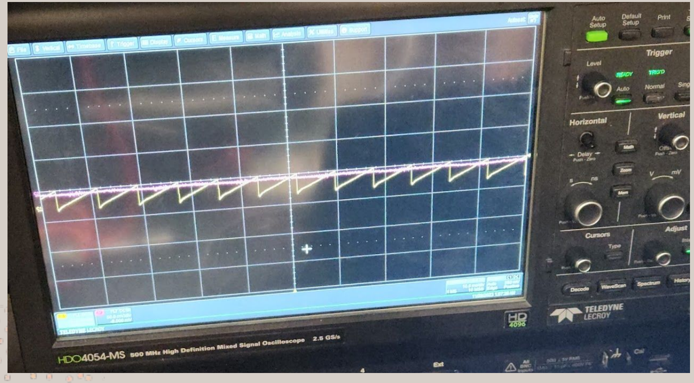
The DAC should be able to output signals up with frequencies up to 15kHz. By measuring the rate of the DAC's clock rate and output to be 39.840kHz, we calculate the Nyquist Frequency (Maximum Frequency) to be 39.840kHz / 2 = 19.92kHz, exceeding our requirement.

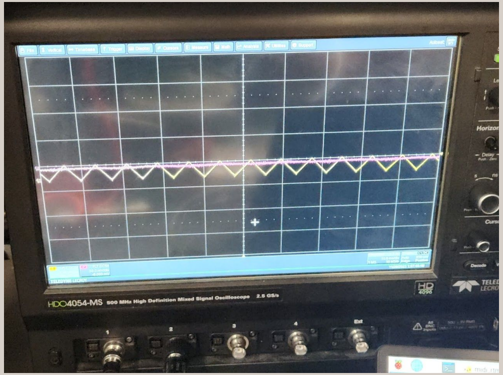
Frequencies of as low as 4 Hz were also achieved!

Test Results - Signal Processing and Conversion Subsystem







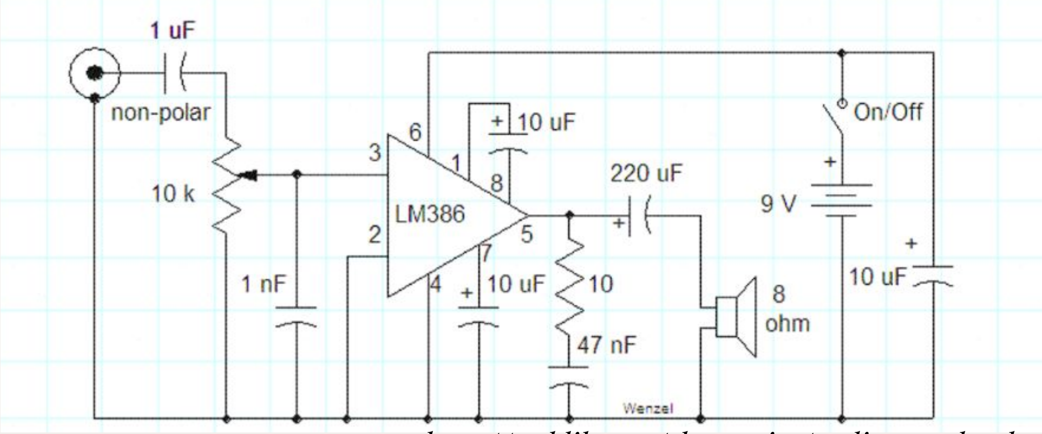


DAC Waveform Outputs

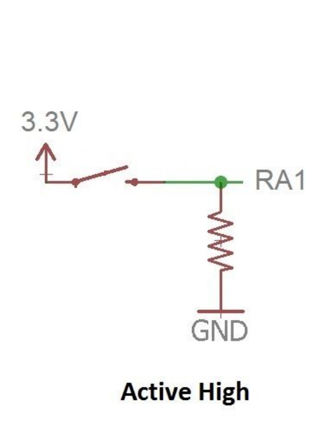
The DAC is able to output four different waveforms. Shown above are the Sine, Square, Triangle, and Sawtooth Waveforms that are the output of the DAC.

Design - Output and Control Subsystem

- Simple switch design to send high or low signal to GPIO pins
- 10K potentiometer for a controllable gain from 20 to 200



http://techlib.com/electronics/audioamps.html



Switch 1	Switch 2	Wave Produced
0	0	Square
0	1	Sine
1	0	Triangle
1	1	Sawtooth

Verification - Output Control Subsystem and Output Subsystem

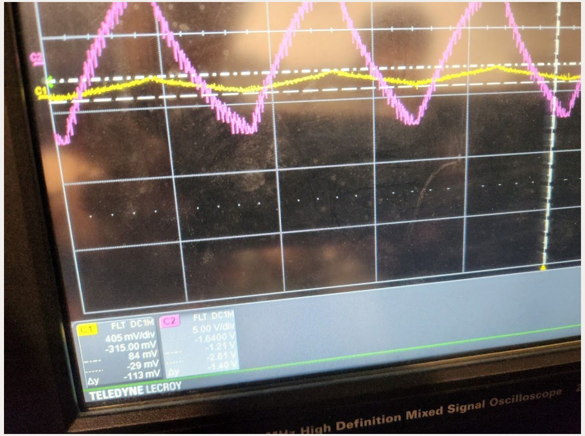
Waveform Variability

Able to hear a difference in the sound when flipping switches, but ran into issues where one GPIO pin wasn't getting any voltage high signal

Verification - Output Control Subsystem and Output Subsystem

Gain Variability

Gain was achieved using a potentiometer linked to an amplifier circuit. Maximum gain of 100 was achieved, with values over such causing clipping, likely driven by the limitation of the amplifier used. Gain is able to be controlled using the potentiometer.

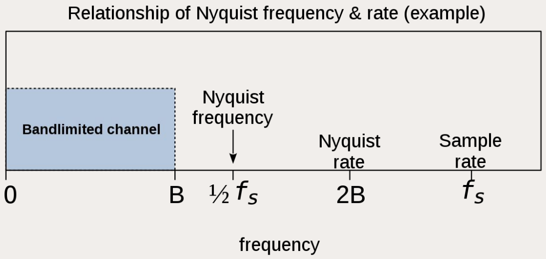


Maximum gain value calculated: 11.3 / 0.113 = 100

Verification - Output Subsystem

Speaker Frequency Range

As mentioned previously, Nyquist Frequency was calculated to be above 15kHz, indicating that the speaker should be able to output such



Speaker Power Rating

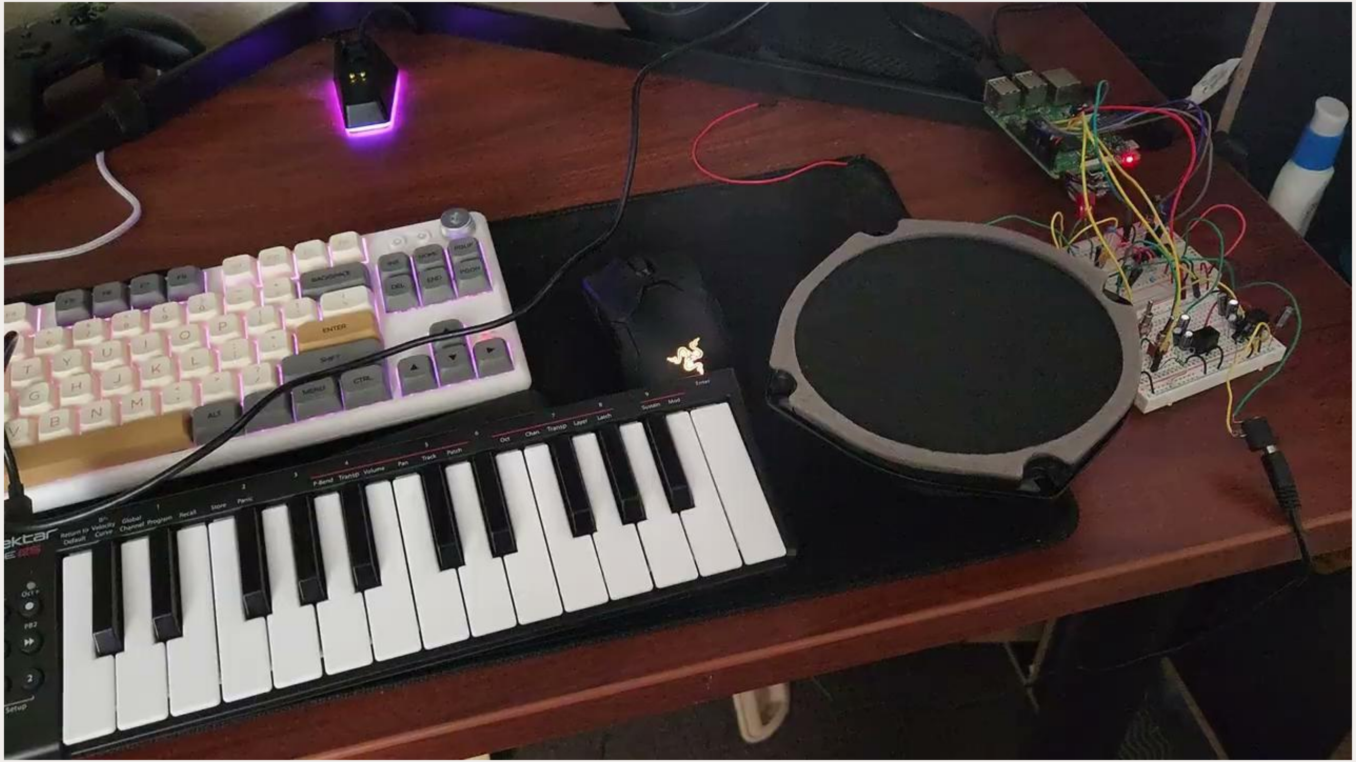
The Speaker is rated for 35W, indicating maximum load. However, the circuit was providing about 1 watt of power as we used a LM386 which had an power rating of 1 watt

Rated Input Power (AES Continuous)	35 Watts RMS (AES Continuous)
Maximum Input Power (IES short term)	70 Watts Peak (IEC Short Term)
Recommended Amplifier Power	35 Watts FTC

Verification - Output Subsystem

Project Final Output

- Full range of MIDI Notes able to be output
- Different types of waveforms available
- 8 Note Polyphony achieved
- Speaker can be driven up to specified wattage, but is not due to circuit limitations



Conclusions - What we Learned

MIDI Data

- How MIDI Data is Formatted
- How MIDI Data is Delivered

Interfacing with DACs

- The various types of protocols DACs utilize
- How DAC Resolution can affect precision
- How to synchronize your application with DACs

Signal Processing

- How to generate and sample signals based on a predetermined sample rate
- How specific frequencies and waveforms can be implemented and produced using physical devices

Engineering Design

- How Subsystems feed into the overall purpose
- How to research and and generate circuit designs to fulfill your goals
- How diagnose issues with the project and incrementally improve aspects about it

Conclusions - What We Would Do Differently

Experiment with Different DACs

- The project was largely limited by the capabilities of the DAC in use, with issues such as **Quantization**
- Explore how the various different data protocols could affect the quality of sound we produce
- Explore how different bit resolutions could affect the quality of sound we produce

Experiment with Different Audio Amplifier Circuits

- Alternative amplifier circuits could have provided lower-noise amplification, leading to higher sound quality
- Alternative circuits could have had a larger amplification effect, enabling the project to drive larger, higher power hardware

Engineering Design

- Researching more in-depth designs to advance our design even further
- Create a more appropriate schedule that better reflected the turnaround parts for parts and components

Recommendations for further work.

Switch to a 16-bit DAC or other Higher Resolution

- A higher bit depth would give more dynamic range, precision and potentially less noise
- Reduces quantization issues

Improve the Capabilities of the Amplifier Circuit

- Improve and employ a lower-noise and higher-power Amplifier Circuit, to further improve the sound quality and increase the volume range at which sound can be produced

Utilize Other Microcontrollers

- With the goal of keeping overall cost of the project design, the potential usage of cheaper microcontrollers would be beneficial
- Even using the Raspberry Pi Pico may be sufficient

References

Bales, R. (2023, May 20). *C++ vs. python: Full comparison*. History-Computer. https://history-computer.com/c-vs-python-2/

How to build a clock circuit with a 555 timer. (n.d.). https://www.learningaboutelectronics.com/Articles/555-timer-clock-circuit.php

paulv. (2015, October 25). *Add an analog output to the Pi (DAC)*. Raspberry Pi Forums. https://forums.raspberrypi.com//viewtopic.php?f=37&t=124184

Wenzel, C. (n.d.). *Audio Amplifiers*. techlib.com. http://techlib.com/electronics/audioamps.html

Thank You!