Matthew Baum

ECE 4999 Independent Project

5/19/2025

Advisor: Professor Bruce Land

Introduction

A MIDI guitar can represent a guitar's sound output as corresponding MIDI messages for use with digital audio software. The applications of MIDI devices are almost limitless, being applicable for music production, virtual instrument performance, sampling, and even lighting and effects. Producing an accurate real-time MIDI guitar can be challenging for numerous reasons.

First, accurate per-string pitch detection requires six concurrent pitch identifications, which may create a computational bottleneck. Guitar strings also have prominent overtones, where the second, third and fourth harmonics can influence incorrect pitch readings from standard frequency detection algorithms such as the fast fourier transform.

To reduce computational latency, efficient pre-processing and pitch identification algorithms are important. Off-loading amplification, filtering, and note-detection to analog circuitry may yield better results for optimizing a real-time system.

Although traditional six-string magnetic electric guitar pickups are the standard medium for guitar signal capture, they do not output individual string audio. Therefore, piezo-electric and per-string magnetic pickups should be considered for proper individual note detection.

Another major hurdle is a guitar's unique range of expressivity. Unlike pianos, guitars vary dramatically in pitch per fret (each note position on the neck), as each string can be bent, causing pitches between standard 440 Hz tuned musical notes. This is an issue because the notes in the MIDI protocol also correspond to these discrete musical notes. Also, string plucking and muting can significantly affect the sound. While these different forms of expression on the guitar make the instrument versatile and capable, they lead to major challenges in discerning a corresponding MIDI output. For a high-level and intricate guitar player, a viable MIDI guitar would detect and output discrete MIDI messages addressing each of these nuances. My long-time experience as a guitarist and interest in furthering my understanding of embedded audio systems has brought me to pursue this project with Professor Bruce Land.

Design

High Level Overview

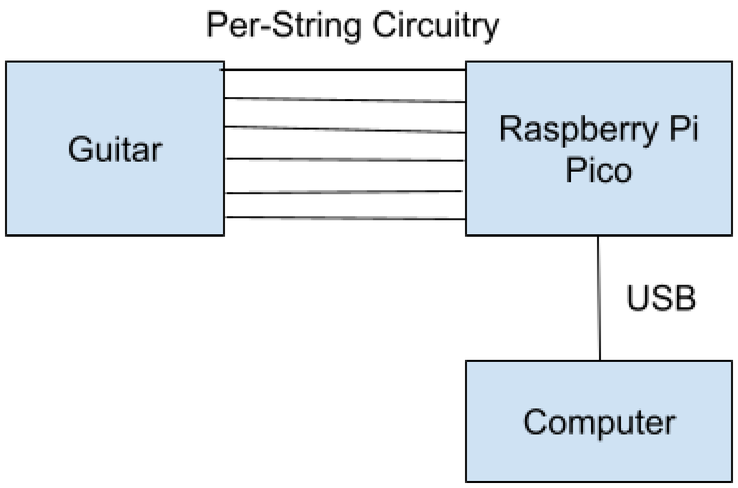


Figure 1: High Level Design

The high level goal of this project is for the guitar to play notes, the circuitry to preprocess the audio for the Raspberry Pi Pico to do pitch detection and MIDI conversion, and then send the MIDI messages to the computer for use with a digital audio workstation.

Hardware

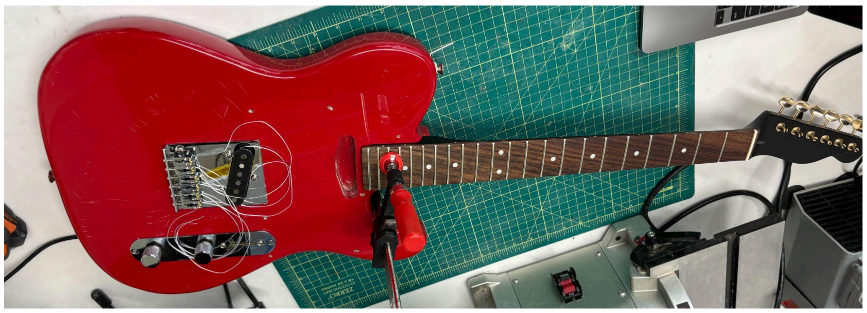


FIgure 2: Constructed guitar

I. Guitar Analog Circuitry

Each string's signal is captured via a piezo-electric bridge microphone, which captures the string's vibration and converts it into an electrical signal. The piezo-electric signal's output produces a small charge with exponentially decaying dynamics, unlike magnetic pickups which exhibit more linear dynamics. This exponential response leads to large variations in voltage output. Lower amplitude notes will then register with less precise ADC readings, thereby increasing inaccuracy in pitch identification.

Piezo-electric microphones also exhibit high output impedance which makes their signal vulnerable to degradation when directly interfacing with audio inputs. To address this, preamplification and compression are commonly employed to stabilize and drive the piezo-electric's dynamic range for more consistent processing. While compression is not used in this prototype, it remains a promising option in future work.

The signal chain includes a level shifter, charge amplification stage, and a 2-pole low-pass Sallen-Key Butterworth filter with a 1kHz cutoff frequency (3 dB point). The level shifter stage centers the voltage between 0 and 3.3V for the Raspberry Pi Pico's ADC. The charge amplifier converts the piezo's small charge output into a usable low-impedance voltage signal. The low-pass filter removes frequencies above the highest expected note per string, reducing false pitch detections. An additional high-pass filter could also improve accuracy by attenuating sub-audio frequencies, but was not included in this prototype. This prototype focused on using the low E-string, where the highest possible note is an E4 on the 22nd fret (329.63 Hz). This circuit was built with the general purpose of being applied to higher frequency strings, so the filter cutoff value could have been smaller (closer to 329.63 Hz). Software can also be used to limit note outputs (described in the software section), but it is preferable to first increase the accuracy in the preprocessing stage. As shown in the figures, the circuit reduced the overall dynamic range that the raw piezo-electric signal produced and conditioned it for accurate ADC sampling.

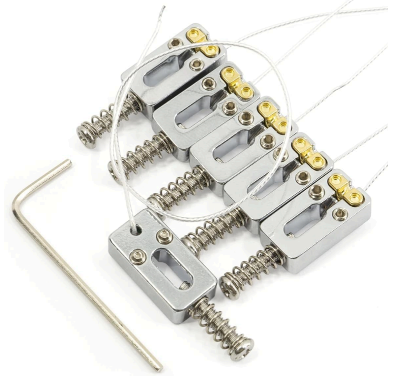
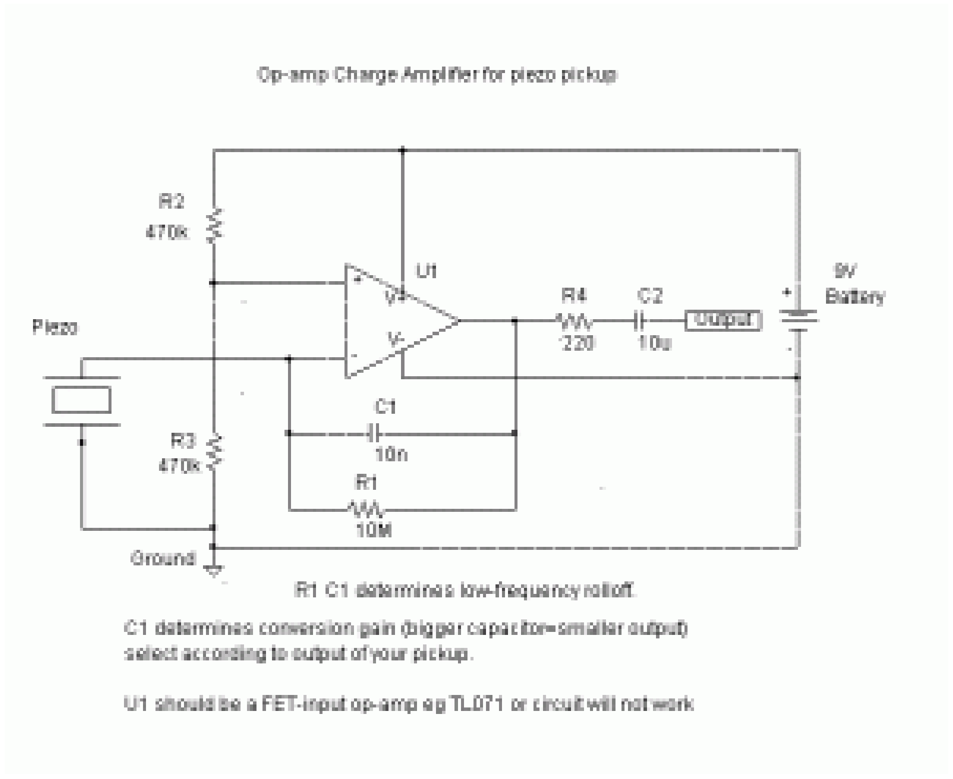


Figure 3: Domofa Musiclily Piezo-electric bridge pickup (Source: Amazon)

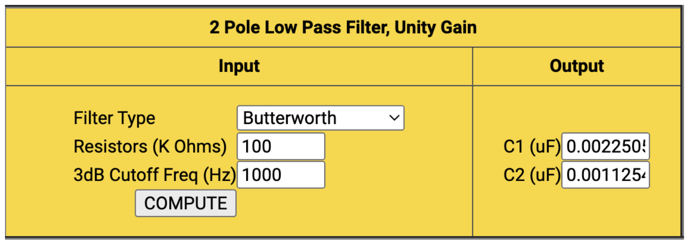

Op-amp Charge Amplifier for piezo pickup

R1 C1 determines low-frequency rolloff.

C1 determines conversion gain (bigger capacitor=smaller output) select according to output of your pickup.

U1 should be a FET-input op-amp eg TL071 or circuit will not work

Figure 4: Piezo Amplification Circuitry
(Source: Seymour Duncan)


2 Pole Low Pass Filter, Unity Gain

Input		Output	
Filter Type	Butterworth	C1 (uF)	0.002250!
Resistors (K Ohms)	100	C2 (uF)	0.001125‹
3dB Cutoff Freq (Hz)	1000		

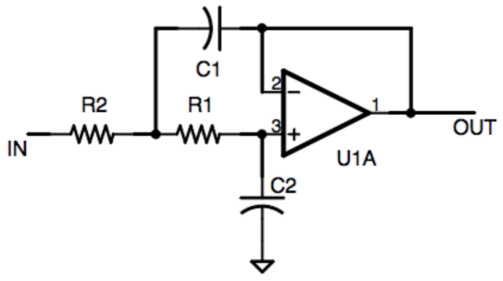


Figure 5: Unity-gain Sallen-Key Butterworth 2-Pole Low-Pass filter calculation
Source: space-inst

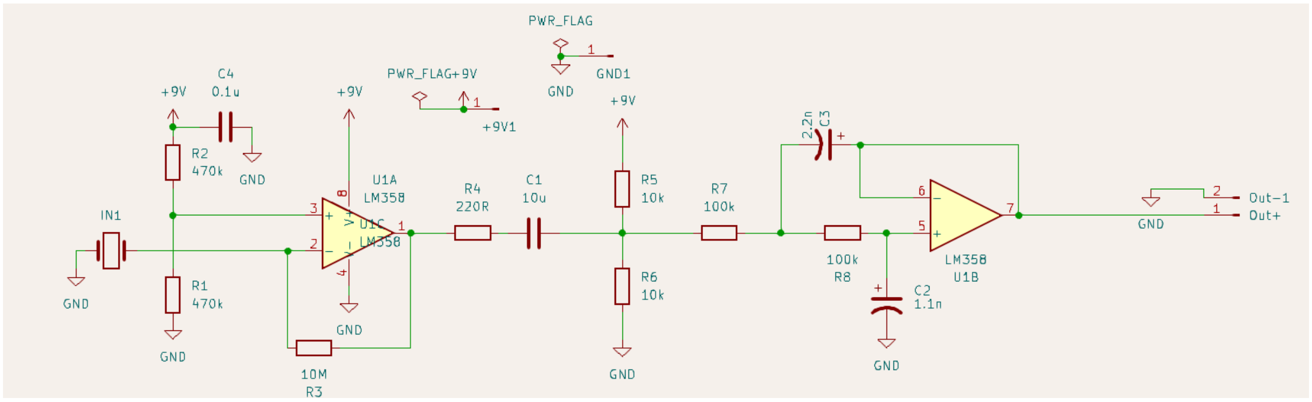
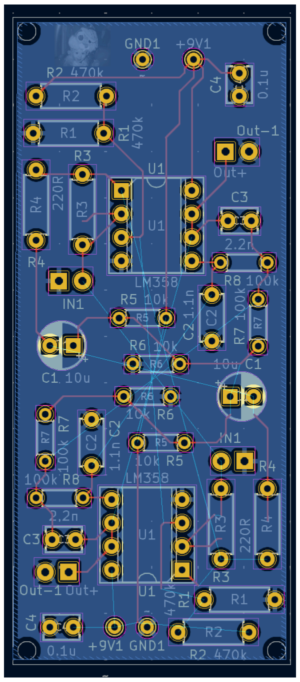


Figure 6: Combined Circuit in KiCad. (Note: The MCP6242 op-amp was used so the 3.3V voltage supply from the Raspberry Pi Pico could be used instead of 9V)

I then created the PCB layout so that the circuit would fit into the guitar's electronics cavity.



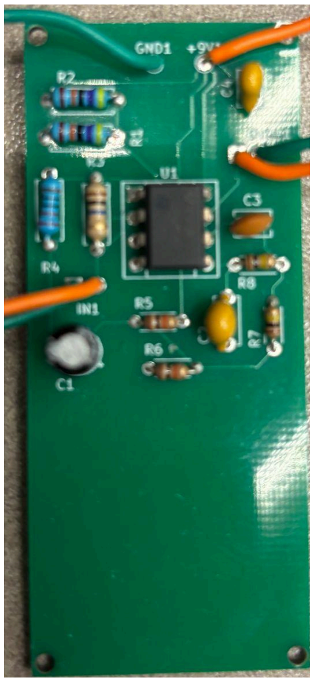


Figure 7: PCB layout

Figure 8: Completed PCB for guitar chamber

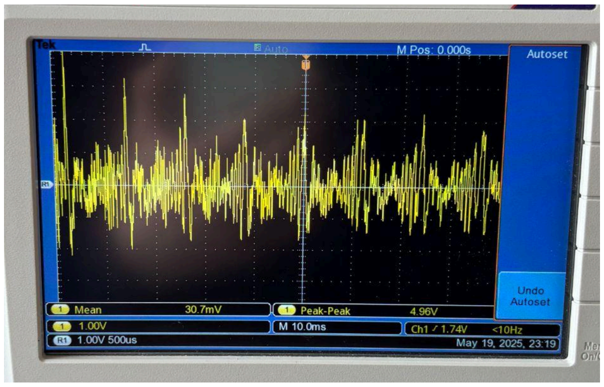
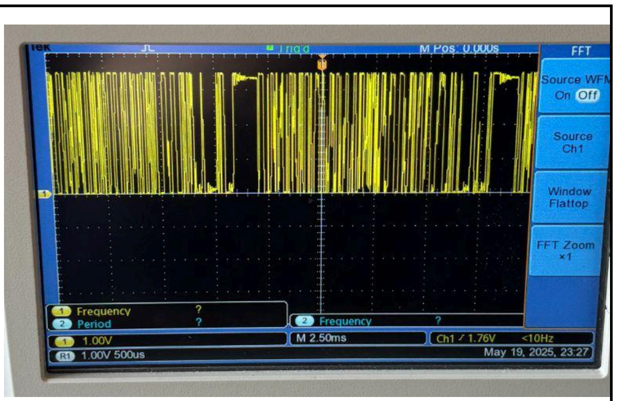
	

Figure 9: Piezo-Electric signal without circuit	**Figure 10: Piezo-Electric signal with circuit**

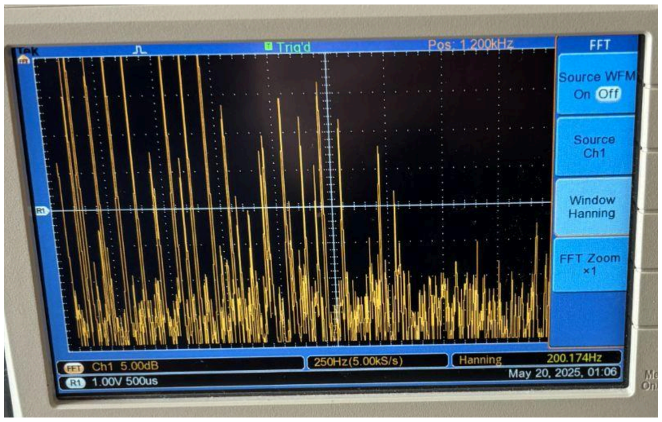


Figure 11: Guitar signal FFT with preprocessing

Software Design

High level Overview:

The code runs a while loop with DMA storing the ADC readings at a 5kHz sample rate and a 512 buffer size. Once the buffer is full, the buffer is copied to the processing buffer for the fast fourier transform (using the C function memcpy is faster than a ping pong buffer in this scenario). A frequency array holds the 9 most recent calculated frequencies. Upon the frequency array being filled, a frequency to MIDI conversion formula is applied to the median frequency and then the MIDI message is sent via USB.

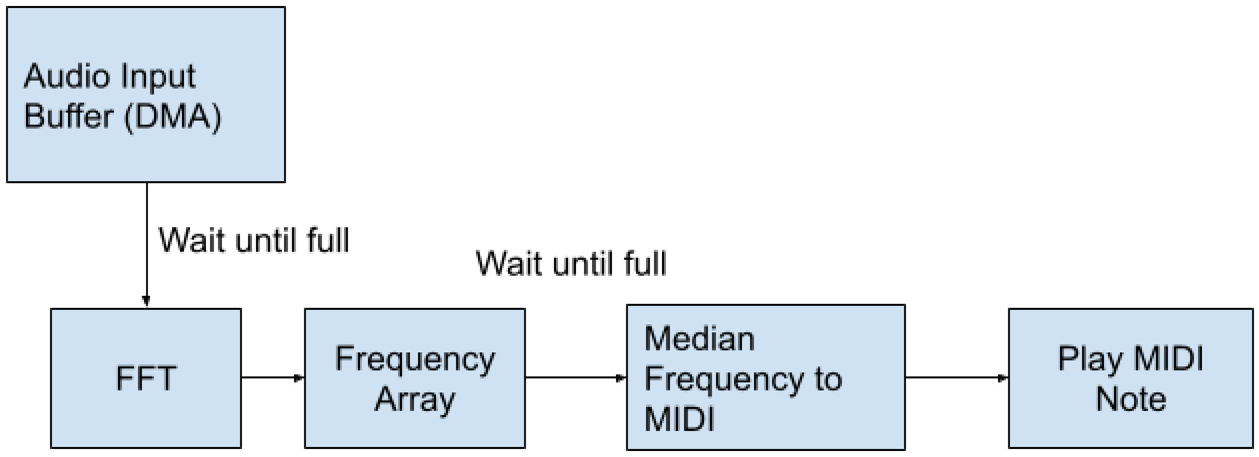


Figure 12: High level software design

I. Audio Input.

The audio input is acquired by the RPP's built-in ADC (ADC0, GPIO 26). Functionality testing involved 44.1 kHz polling, interrupt-based sampling, and direct memory access (DMA). Once working, the next objective was creating an audio loopback, which takes an audio input and immediately outputs it. This was done in the same iterative steps. Tests were done with sine waves, where readings were checked with an oscilloscope. Connecting the audio output to a speaker for audio quality checking was also done. This built-in 12-bit ADC produces a lot of noise due to electrical interference with other components on the board, so much so that only 10 bits are viable for processing. For the final prototype, DMA was used for its fast data transfer and its reduced CPU use. The ADC_DREQ set at a 5kHz sampling rate (9600 clock division) was ultimately used. For the sixth string, this sampling rate could have been lower to increase bin resolution for the lower frequencies.

II. Pitch detection

Pitch identification for each individual string is used by an FFT algorithm from Professor Bruce Land's FFT spectrogram. Initial functionality testing was done with a sine wave generator outputting from a computer (at line level), basic preprocessing (filtering and amplification), into the DMA audio input code. It was soon apparent that sampling at 44.1kHz and 512 samples produced a bin size of 86.13 Hz (44.1kHz/512), much too large for low frequency resolution, especially on the lowest string where the low E string is roughly 82 Hz. Therefore, I lowered the sampling rate to 5 kHz to get a bin size of 9.77.

A frequency array is implemented as a buffer to hold the 9 most recent frequencies. It is sorted and returns the mean frequency when the index reaches the end of the array. This allows for more stable, less arbitrary pitch outputs (due to noise) since the output frequency depends on numerous data points instead of one. As a prototype, this step is a major bottleneck, as (512/5000 * 9) leads to a MIDI note being outputted every 0.9216 seconds. Reducing the frequency array size and doing further preprocessing (better filter design) could improve accuracy and efficiency during this step.

III. MIDI Conversion

Upon receiving the median frequency, I experimented with two frequency to MIDI conversion formulas.

1. MIDI Note = $12 * \log_2(\text{frequency} / 440) + 69$
2. n = (int) ((12 * log(f / 220.0) / log(2.0)) + 57.01)

Formula 2 led to lower precision/accuracy based on testing, so formula 1 was used.

IV. USB MIDI Output

The USB port used to flash code to the RPP also served as the USB MIDI connection port. Once the code is run, the RPP is then recognized by the connected computer as a USB MIDI device. To configure the RPP in the code as such, a modified TinyUSB library repository by Github user infovore provided a functioning out-of-the box example for the RPP. This repository modifies the CMakeLists.txt file to include TinyUSB and MIDI files and declares the associated functions in the main file. The prototyped audio pitch detection code was reformatted to work within the TinyUSB code. To be detected as a USB MIDI device, the code calls the function tud_task() every iteration of the loop. MIDI messages are sent with the tud_midi_n_stream_write(0,0,msg,3) function, where the message packet is an array of three 8 bit unsigned integers (MIDI documentation is provided in the reference section). For the low E string, notes were constrained between 28 (E4) and 127 because the FFT would commonly return the pitch an octave higher than what was originally played on the guitar, most likely due to the second harmonic (an octave higher). Basic "new note" logic was implemented so only current notes different from the previous note would be sent. There is much room for improvement in this step which is discussed in the "Further Work" section.

```
// Turn off note.
   msg[0] = 0x80;      // Note Off - Channel 1
   msg[1] = note;      // Note Number
   msg[2] = 0;         // Velocity
   tud_midi_n_stream_write(0, 0, msg, 3);
   // Turn on note
   msg[0] = 0x90;      // Note On - Channel 1
   msg[1] = curr_note; // Note Number
   msg[2] = 127;       // Velocity
   tud_midi_n_stream_write(0, 0, msg, 3);
```

Figure 13: Turning off and on MIDI note example with TinyUSB.

Results and Conclusion

This system is successfully able to detect guitar audio and output the corresponding MIDI notes to the computer via USB connection. The latency is very apparent and octaves and third harmonic (the perfect fifth interval) of the original note are often sent. Sometimes, the note A#0 is sent, and I could not debug this in time for the final demo. Overall, this project expanded my skills in woodworking, circuit design, PCB layout, soldering, embedded C programming, prototyping, documentation, and researching. Time did not permit for me to scale to multiple strings, and numerous unexpected software and hardware obstacles slowed down my progress at times. Regardless, I achieved a basic functioning prototype and have many ideas for improving the project in the future.

Further Work

The primary objective for future work centers around scaling the system to six-strings so it can function as a normal guitar. This requires running six separate pitch identification threads and most likely multiplexing. However, I first want to further address latency, note accuracy, and new note detection on a single string. For lower latency, I plan to lower the FFT buffer size to either 256 or 128. It was unfortunate that my FFT code could not run lower than 512, so I will have to debug this. I also look to thread the FFT algorithm for one string, since this will probably improve computational efficiency and prepare my program for scaling to multiple strings. Pitch bending is another goal I would like to work on. Machine learning algorithms may be important for detecting and processing the different styles of guitar playing and techniques that were discussed. Professor Land suggested “new note” logic with an Schottky diode RC circuit to provide interrupt callbacks on plucked notes. I am also interested in switching to an external ADC for higher quality readings with less noise, as it may provide better pitch detection results.

Acknowledgements

I would like to thank Professor Bruce Land for his mentorship, weekly meetings, and his continuous support throughout all aspects of the project. I also thank Professor Hunter Adams for his help and encouragement and Professor Travis Johns in the computer music department for his guidance.

References

Domofa Piezo-electric bridge:
https://www.amazon.com/Domofa-Upgraded-Version-Tremolo-Electric/dp/B0CSD3R3YF?source=ps-sl-shoppingads-lpcontext&ref_=fplfs&psc=1&smid=AB5TS5944X1DE&gQT=1

DMA with DAC: https://vanhunteradams.com/Pico/DAC/DMA_DAC.html

FFT Spectrogram:
https://people.ece.cornell.edu/land/courses/ece4760/RP2040/C_SDK_DSP/index_spectrogram.html

Frequency to MIDI Conversion Formula:
https://newt.phys.unsw.edu.au/jw/notes.html#:~:text=m%20for%20the%20note%20A4,)%2F12(440%20Hz).

MCP6242 Datasheet:
https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ProductDocuments/DataSheets/21882d.pdf

MCP 4822 Datasheet:
https://vanhunteradams.com/Pico/Birds/DAC.pdf

MIDI Message Documentation: https://midi.org/expanded-midi-1-0-messages-list

Piezo Amplification Circuitry:
https://www.seymourduncan.com/blog/latest-updates/piezo-vs-magnetic-pickups

Raspberry Pi Pico Datasheet: https://datasheets.raspberrypi.com/pico/pico-datasheet.pdf

Raspberry Pi Pico SDK: https://datasheets.raspberrypi.com/pico/raspberry-pi-pico-c-sdk.pdf

RP2040 Datasheet: https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf

Sallen-Key Low-Pass Butterworth Filter Calculator:
https://space-inst.blogspot.com/2020/05/sallen-key-lowpass-butterworth-filter.html

TinyUSB USB MIDI Device for Raspberry Pi Pico:
https://github.com/infovore/pico-example-midi