ChordSync

Zawad Munshi, Owen Park, Neel Vora, Akshaya Naapa Ramesh, Hudson Hall
https://github.com/owenpark8/eecs473-ChordSync

EECS 473 Fall 2024

Table of Contents

I. Introduction and Overview

The electric guitar, invented in 1940, has seen little innovation to enhance learning and interactivity. Our project addresses this by developing a smart electric guitar with integrated neck display screens to visually guide users on finger placement, making the instrument more intuitive and engaging for beginners.

Our prototype for the Design Expo demonstrated this vision with a functioning display system highlighting finger placement and sound detection for real-time feedback. We ensured external components, including display screens and a Raspberry Pi with a custom PCB, did not compromise playability or structural stability.

While full song feedback (note detection) was not fully implemented, we showcased a "learn mode" at the re-demo. This mode displayed chords on the neck screens and provided real-time feedback via the app, demonstrating the system’s potential to transform guitar learning. Despite falling short of providing feedback for entire songs, this feature aligned with our goal of creating a more intuitive and engaging learning experience. With further development, full song detection remains achievable within the existing design.

II. Description of project

Audio System

Hardware

To analyze the guitar's audio, we connected a Guitar Headphone Amp to the guitar and routed its audio output to the Raspberry Pi via a USB ADC/DAC unit. This setup treated the guitar's output (from the string pickups) as a clean "microphone" input for analysis. The system worked well, allowing audio to pass from the guitar, through the Pi, and into the speaker with minimal latency or sound distortion.

An attempt was made to modify the amp by directly soldering input wires to the guitar pickups and testing electrical control via the MCU’s GPIO using transistors to short buttons on the amp’s PCB. However, this setup introduced significant noise in the analog audio transmission, rendering it unsuitable. Ultimately, the unmodified Guitar Headphone Amp was used. A custom amplifier circuit PCB was considered but abandoned due to the team’s limited expertise in analog circuit design, which made the risk too high.

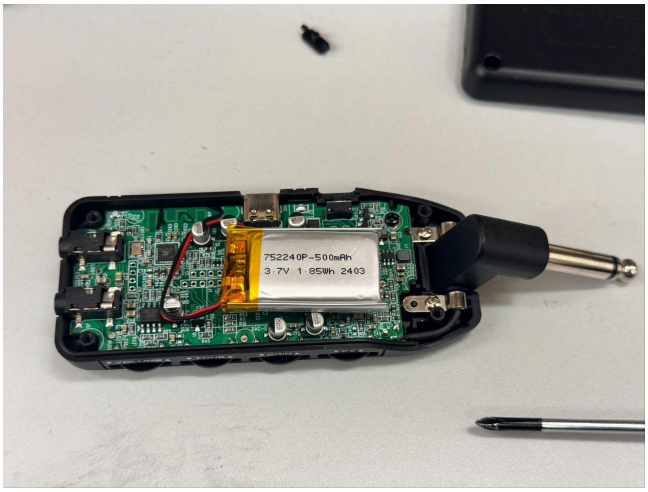


Figure 1. Internals of headphone-amp

A USB speaker to play the guitar audio from the Pi. This was a regular computer speaker off Amazon, but worked well for the quality of audio needed from the instrument.

Software

Audio Passthrough

To enable audio passthrough, proper device assignment in Raspberry Pi OS was required. The USB speaker appeared as both an input and output device, while the ADC/DAC connecting the Pi to the guitar had an output device we didn't use. The following shell command was used for passthrough:

```
arecord --rate=88200 --format=S16_LE --buffer-time=1 - | aplay - --buffer-time=1
```

This set the sample rate to 88,200 Hz (twice the ADC signal of 44,100 Hz) to prevent aliasing and minimized latency. Devices were not explicitly specified in the command to avoid locking them, which would prevent running multiple arecord commands on the same input. Instead, input and output devices were configured globally using PulseAudio. This allowed ALSA tools (arecord/aplay) to share default devices across multiple processes, ensuring smooth audio passthrough and processing.

Audio Processing for Detection

The project's initial goal was to provide note-by-note feedback to users based on the song they select to learn. A stretch goal was to enable users to upload guitar tab files, which typically include fret and string placement, notes, and MIDI-formatted durations. To support this, the

SongInfo interface was designed to integrate the reference song database with audio processing and fretboard display interfaces.

To ensure accurate feedback, audio from the user's performance is converted to MIDI for comparison rather than relying on traditional audio detection methods like spectrogram analysis. This approach was chosen due to the team's limited Digital Signal Processing experience and the reliability of MIDI comparisons, despite potential latency.

Spotify's open-source software, **Basic Pitch**, was used to convert user audio (wav files) to MIDI. This tool processes note length and timing using parameters such as onset threshold, frame threshold, minimum note length, and BPM.

Before integrating the Basic Pitch Python library into the project workflow, its performance was thoroughly evaluated. Approximately 20 songs from Online Sequencer—a platform that provides MIDI files and playable audio—were used for testing. By fine-tuning parameters such as onset_threshold=0.70, frame_threshold=0.50, and minimum_note_length=11, Basic Pitch consistently generated MIDI outputs closely matching the reference files, with errors within a few milliseconds (see Figure #).

After applying data cleaning techniques, such as removing repetitive notes with identical timestamps, the output MIDI files were nearly identical to the reference. Testing with human guitar input yielded similar results, confirming Basic Pitch as a reliable tool for this project's audio-to-MIDI conversion needs.

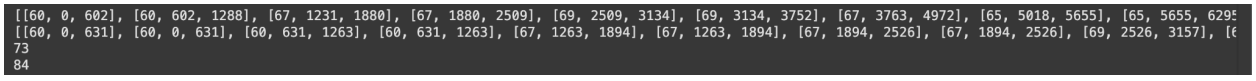


Figure 2. MIDI note number, start, and end times of the notes of Twinkle Twinkle (top row is reference MIDI notes & bottom row is uncleaned Basic Pitch MIDI output)

Since Basic Pitch requires at least one second of audio to generate a MIDI file, real-time feedback was not feasible. To address this, the interface was designed with two learning modes: Chord Mode and Song Mode. In Song Mode, users play a song at least a minute long and receive performance feedback at the end. Chord Mode focuses on shorter segments, providing quicker feedback after users play a single chord. Both modes share the same interface, with analysis based on Basic Pitch output.

For note-by-note analysis, the MIDI output from the user’s performance was parsed into a nested-list Python structure containing the MIDI note number, absolute start time, and end time. This conversion was necessary as MIDI note timings are typically relative to prior notes.

This parsing method also supported processing Guitar Pro Tab files for the reference song database.

```
Message('note_on', channel=0, note=60, velocity=0, time=62),
Message('note_on', channel=0, note=67, velocity=0, time=5),
Message('note_on', channel=0, note=67, velocity=77, time=148),
Message('note_on', channel=0, note=67, velocity=0, time=199),
Message('note_on', channel=0, note=48, velocity=0, time=5),
Message('pitchwheel', channel=0, pitch=1365, time=11),
Message('note_on', channel=0, note=60, velocity=72, time=0),
Message('pitchwheel', channel=0, pitch=1365, time=5),
Message('note_on', channel=0, note=60, velocity=0, time=0),
Message('note_on', channel=0, note=69, velocity=85, time=0),
Message('note_on', channel=0, note=41, velocity=88, time=5),
Message('note_on', channel=0, note=60, velocity=77, time=0),
```

Figure 3. Default MIDI file output

Once MIDI data was parsed, the pybind11 library transferred formatted note data from a Python nested list to a nested integer vector in C++ to align with the SongInfo struct. This struct, central to the user interface, fretboard implementation, and database formatting, was also used for audio processing, which was conducted in C++ for consistency and organization.

In the audio processing interface (initially called playerMode), both Chord and Song Mode analyses were performed. In Chord Mode, recorded notes in SongInfo format were compared to a reference Chord SongInfo struct from the database. Each reference note was checked against the player's notes to confirm all required notes were played correctly. Timing was not considered, as chords require simultaneous strumming, guided by the LCD fretboard. Mistakes were flagged if incorrect notes were detected, ensuring accuracy in chord evaluation.

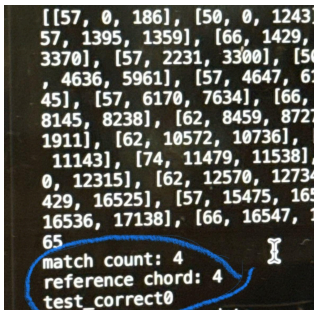


Figure 4. Testing of Chord Mode (G major Chord)

In Song Mode, timestamps and note durations are critical for detecting whether users played the correct notes on beat and held them for the appropriate duration. A map with an integer key and a vector of a notesPlayed struct was used for comparison. Each notesPlayed struct contained the reference note's start time, duration, and a tag indicating if it was played correctly. The map stored MIDI note numbers as keys and vectors of notesPlayed for reference notes. For each played note, the map identified the closest reference note by comparing start timestamps. A played note was considered correct if it was within 50% of the reference duration and within one second of the reference start time. Matched reference notes were marked as

"seen." Finally, the map was iterated to determine if all reference notes were played correctly. Figure 3. illustrates this detection process using a recorded wav file of "Twinkle Twinkle."

Figure 3. Song detection analysis based on computer recording of Twinkle Twinkle (Right most picture shows output of song detection).

The note detection algorithm was validated using songs from Online Sequencer, but not with human playing due to the complexity of timing in Song Mode. Unlike Chord Mode, Song Mode depends heavily on precise synchronization with the user interface, fretboard, and human response time. Full validation required an almost fully integrated system, which was not completed due to the complexity and time constraints of integrating the fretboard and interface.

To address MIDI timing issues, a start-tone system was tested. The Pi would play a start tone, and all recorded note timings were adjusted relative to it. For instance, if the first note was expected 5 seconds after the start tone, the system aligned timings accordingly. While effective in initial tests, this approach was abandoned to focus on a more reliable Chord Mode.

Displays

Hardware

We had used 6 Hosyond Raspberry Pi 3.5” Displays, with a resolution of 480x320 and the ILI4986 driver. Although these displays do have touch screen capabilities, we had decided the unreliability of the resistive screens were too limiting. We went with these displays for a few reasons: One, the size and resolution worked well for our purposes. We were able to align these on the guitar and still make for a comfortable fit. Two, they used SPI. Despite the inherent issues and limitations of using SPI for graphics (as discussed later), they allowed us to communicate with a large number of displays without complex wiring and with relatively weak display controllers (being our STM32). Three, they were cheap to acquire and test with, and a few of us already had experience working with similar displays in the past. Four, the position of the connectors on the side meant that we could easily hang the pins off the side of the guitar neck, letting us connect wires/PCBs to them easily with minimal cutting into the guitar.

We did have a number of issues getting these displays to work properly. Getting the SPI on all of the screens to work reliably under initial testing (specifically, with breadboards and protoboards) was a challenge. The wiring of these screens are extremely sensitive, and a slight tug on any of the wires could cause the entire system to go down. Even something as simple as a reset line unplugged from one screen would cause the entirety of SPI to fail for all of them. This was especially problematic for initial testing on breadboards, since we needed every single connection to be solid and breadboards were rather finicky to work with. We were eventually able to do some testing by lowering our SPI speeds to a very low number, but we could not test properly until our actual display connector PCBs came in.

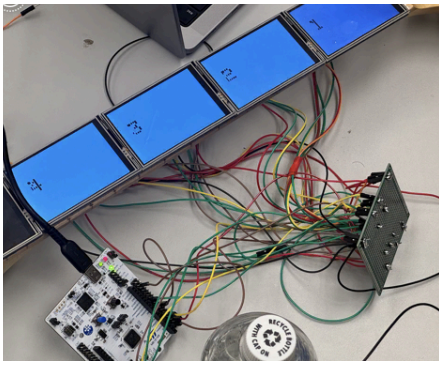


Figure 6. Prototyping LCD Fretboard with protoboard

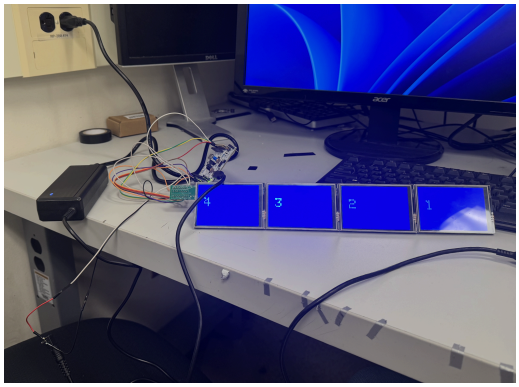


Figure 7. LCD Fretboard with Fretboard PCB

Software

Controlling the Screens

We controlled six LCD screens using three SPI lines (two screens per line), with shared reset and data/command select pins. The screens were initialized with optimized voltage, brightness, and frame rate settings, and configured for 16-bit color (the lowest bit depth supported). The SPI clock was tuned for maximum stable speed. A draw_rectangle() function (Appendix A) was implemented to input pixel values, which was later used to display notes on the screens.

Communicating with the Raspberry Pi

Communication between the MCU and Raspberry Pi (RPi) was handled via UART using a defined messaging protocol (Appendix B). Messages included a header byte, triggering interrupts on the MCU to process incoming data, such as song and note information. Songs were stored on the RPi (due to its larger storage) and parsed into a digestible format for the MCU (Appendix D). The MCU stored one song at a time.

Playing Songs

The fretboard was represented as a coordinate grid, with frets as the x-axis and strings as the y-axis, consistent with MIDI formatting (fret 0 as open string). The draw_rectangle() function was extended into a draw_note() function (Appendix C) to convert notes into pixel coordinates for display. Fret and string measurements were used to map notes to pixels.

A 1ms timer interrupt tracked rhythm and note timing. Notes were displayed in a sequence of colors (red, yellow, green, and white/black for dark mode) at fixed intervals before the note's original timestamp, signaling upcoming notes to the user. The process_loaded_song() function expanded notes into this sequence and sorted them by timestamp. During each timer interrupt, the sorted array was checked, and draw_note() was called to display notes as their timestamps were reached.

Web Application

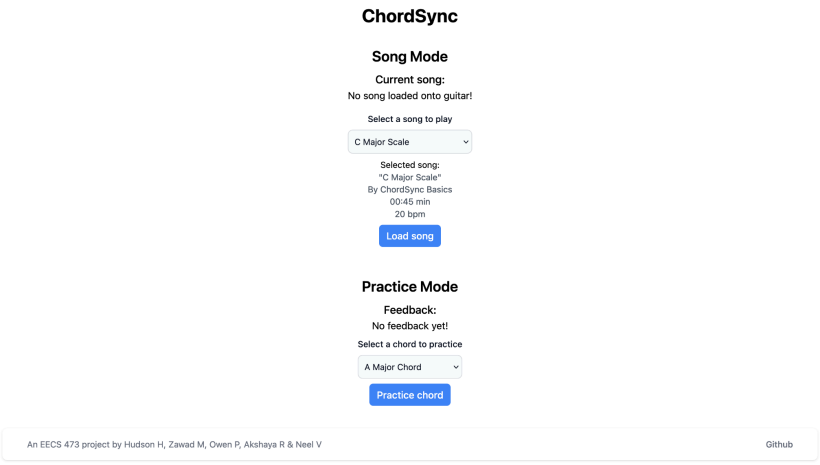


Figure 8: Web application frontend

The web application serves as the central interface for the smart guitar system, hosted on the Raspberry Pi using the cpp-httplib library as a lightweight HTTP server. It allows users to select songs or chords, control playback, and receive real-time feedback via the guitar's display

system. Built with pre-rendered HTML templates populated dynamically with data from a SQLite database, the app ensures a seamless user experience. The htmx library enables dynamic updates, such as replacing buttons or forms, without full-page reloads, while TailwindCSS handles styling.

The web server supports multiple RESTful API endpoints for functionalities like displaying the main interface (/), updating song info with Server-Sent Events (/song-info), and controlling playback (/play-song and /stop-song). These endpoints interact with the STM32 microcontroller via UART to manage the guitar's displays. Dynamic features, such as chord selection forms, are enabled by injecting data into HTML templates at runtime using the {fmt} library. A custom build-time script compiles static assets (HTML, CSS, JS) into constexpr char[] for fast delivery, reducing runtime dependencies and file I/O. Compile-time regular expressions further optimize API request parsing, avoiding bulky formats such as JSON.

Each API request is handled as a separate thread, allowing the server to process multiple requests concurrently. This multithreaded design improves responsiveness but required careful synchronization mechanisms to manage shared resources, such as the current song state and MCU communication. Critical sections, such as those involving UART communication or database access, are protected using mutexes and condition variables. For instance, when a user requests to play a song through the /play-song endpoint, a mutex ensures exclusive access to the MCU communication interface while also coordinating updates to shared state variables like the current song ID and playback status.

Communication with the STM32 is handled via WiringPi over UART, using a defined messaging protocol with error checks and acknowledgments for reliable playback and note/chord display. The SQLite database stores song data, including titles, artists, and MIDI note sequences, enabling efficient management of large song libraries and support for future features like user-uploaded songs. This design ensures a robust, user-friendly, and efficient system.

PCBs

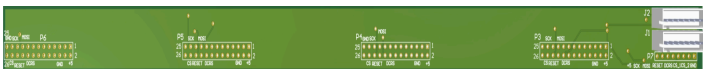


Figure 9: Long Neck PCB, 4 screen interface

Figure 10: Short Neck PCB, 2 screen interface

We used three custom PCBs for this project, two of which simplified wiring and maintained signal integrity for data lines connecting six LCD screens. The neck PCBs were designed to support either six or four screens in case of integration issues. The longer PCB connected to the STM32 controller and bridged connections to the second PCB.

Testing showed that three LCD screens could share one SPI line, but for better performance, we used three SPI lines with two screens each to optimize speed for playing songs. The longer neck PCB required 16 connections from the main PCB, while the smaller one needed eight. Each SPI line included MOSI, CLK, RESET, and RS (shared across screens), as well as unique CS lines for each screen. Power and ground (5V) were supplied by the main PCB, ensuring proper operation of all screens.

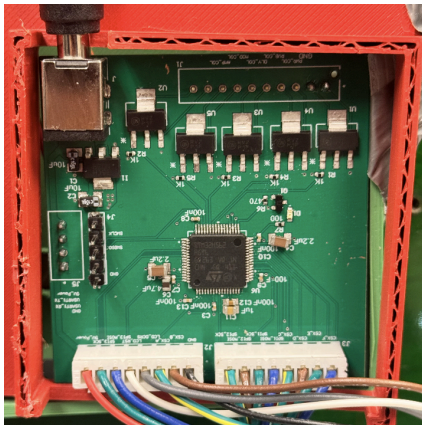


Figure 11: Display Control PCB

We chose the STM32H562 MCU for its superior speed over the STM32H503 and its built-in support for three SPI lines. The MCU featured five GPIO pins and a UART line for interfacing with the guitar amp board and Raspberry Pi. A 5-pin header breakout was used for the SW debug pins (SWDIO, SWCLK, and NRST), allowing us to flash code with an ST-Link V3 and restart the board via a jumper for easy testing and resetting.

To power the MCU, we used an LM1117 LDO with a fixed 3.3V output, supplied by a stable 5V input from a 50W AC/DC converter with a barrel plug, matched with a barrel jack on the PCB. A 40-mil 5V trace was used to power the Raspberry Pi and all LCD screens. For debugging, we included a visual indicator LED.

Five simple MOSFET circuits were initially added to interface with the guitar amp but were later discarded due to noise issues with the guitar's internal wiring. JST connectors were used for screens and the Raspberry Pi, ensuring robust, reliable connections with properly sized crimps that resisted accidental disconnection.

Mechanicals

Many final audio issues stemmed from the unreliability of the base guitar, which was a cheap, used model purchased to stay under budget. While modifications added complexity, several issues, such as unreliable tuning, existed in its stock state. Replacement parts, like new strings, were assumed to resolve these problems, but challenges persisted. For instance, a tuning peg broke the morning of the demo. Though repaired by the re-demo, it was too late to adopt a new guitar, resulting in a playable but less reliable instrument.

Despite these setbacks, significant mechanical work was done to make the guitar functional. Initial testing with raised strings was successful, allowing proper retuning and the ability to play basic songs.

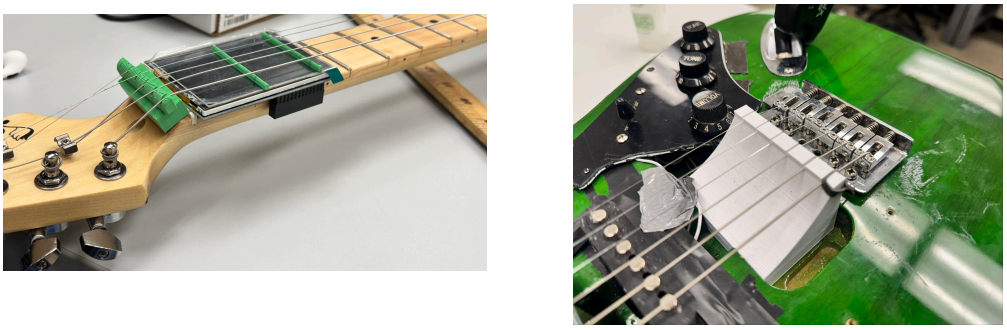


Figure 12. Prototyping of 3D printed Guitar nut (left) & string riser (right)

By the end of our project, the project had reliable and well-fitted string risers that were not prone to slipping and accounted for the shape of the guitar:

Figure 13. Milled cuts for connectors + PCB

Fitting all six displays onto the guitar neck presented challenges. After raising the strings, holes were drilled into the neck for the display connector pins, particularly for the bottom three screens. A mill was used to cut precise slots in the wood, ensuring proper alignment.

Additionally, the smaller PCB was fit within the guitar body using these clean cuts, accommodating all six screens effectively.

Another challenge was creating proper frets. Traditional metal frets were unsuitable because they require a notch to attach to the wood, which wasn't possible with the flat displays. Instead, frets were 3D-printed, requiring multiple iterations to achieve the right height—tall enough to avoid string interference but not too short to hinder playability. Frets were attached using double-sided tape instead of glue, allowing for adjustments as needed.

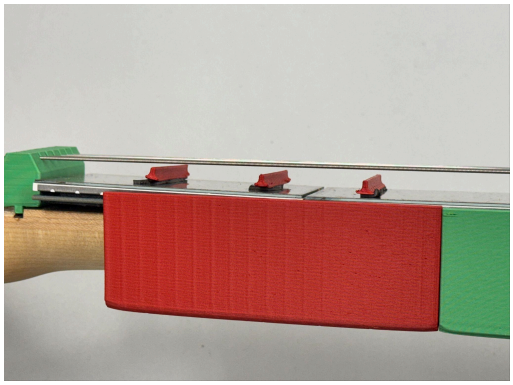


Figure 14. 3-D printed frets with raised strings, Brackets for PCB protection

Remaining 3D-prints necessary for our project were protectors for our electronics. L-shaped brackets were printed to protect the neck circuitry for the displays, and a box was printed for the back of the guitar body to house all of the other electronics (Raspberry Pi, Controller PCB).

Overall System Overview

There are a number of primary systems that make up our product: stacked fretboard display, UI software interface, sound detection system, and central processing. We used 2 main computation sources in this system, one being our central computer with the Raspberry Pi and the other being an STM32H562 microcontroller. The overall system will be powered via a power supply that will supply power to all of our electrical components: LCD screens, STM32H562, Raspberry Pi, DAC and ADC. The PCB will have buck converters and low dropout regulators to break down our voltage signals.

Our central processing unit is the Raspberry Pi. As shown in the figure below, the Pi will be in charge of receiving digital audio signals through the ADC and amplifier circuit where it will be able to determine the accuracy of the note being played while also passing the note to the speaker where it can be played via speaker. The Pi also interfaces with our software UI, the

website, where it receives instructions on songs to load and compare with in the appropriate file formats. The Pi drives the control signals to the STM32H562 which in turn controls the display screens on the fretboard display.

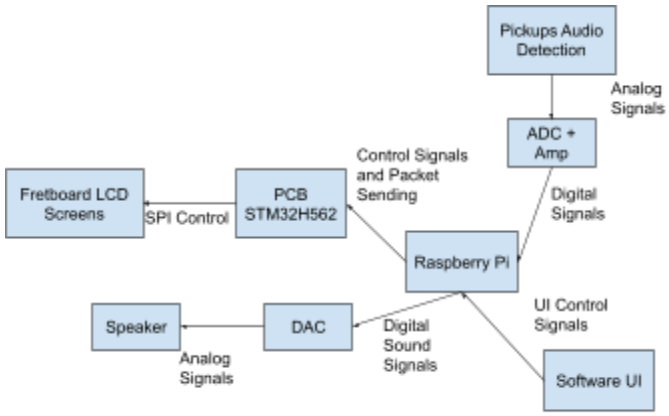


Figure 14. Data Flow

III. Milestones, Schedule, and Budget

Original Milestones:

Milestone 1 (10/10)
Fretboard LCD able to light up specific notes on the guitar
User Interface app/website able to send data to RPI
Preliminary Schematics for PCBs
Milestone 2 (11/5)
PCB assembled
Detect/identify notes on RPI
Ability to parse MIDI files/Songs by note, but not integrated with fretboard
Most components begun to integrate, but incomplete

We were able to stick with our milestones for the most part, but there were a few deliverables that we had spent a little longer on. We were late on getting our LCD's showing notes properly, but were able to get audio detection working early. By the end of milestone 2, we had the LCD's working, basic communication between the Pi and STM, and progress on audio detection. We ordered our PCB's late, but was fortunate in that the actual boards came in on time despite this.

However, we had issues flashing the board. Our PCB gave us initial issues with flashing, and then was reliable for weeks. However, by the days nearing demo day, these initial flashing issues started appearing again. This severely limited our ability to integrate our final product, and we had ordered a revision of the PCB expecting that the unreliable flashing would be fixed. We had discovered the true issue very late (the cables for the STLink Programmer were experiencing signal degradation, and replacing them solved our flashing issues). However, since this took us days to debug, we had limited time to integrate our systems at the end of the project.

IV. Lessons learned

We have all learned new technical skills through this project, and were exposed to challenges that none of us have encountered before. Our project involved extensive audio processing, mechanical engineering, advanced display control and both backend and front end web development. We were able to get a significant amount done considering our relative skill sets. (our displays worked well and reliably after months of trial and error, our audio processing on its own worked very well, and most of our 3D prints fit very well). This may have been a project better suited for a longer time period and with a greater amount of prototyping. Many of our issues could have been solved by more expensive components, and a company attempting a similar project would have access to a greater variety of custom components and tooling that would have made this a much more polished product.

Our main issue was integration, and many of our components relied on others in order to test and work properly. For example, in order to get song detection fully working, it was necessary for the display logic to be timed properly. Since we had limited time to get the displays working on the guitar itself, we shifted the use case to just chords in order to still take full advantage of the audio processing backend we had created.

The interfacing with the website was also significantly harder than expected - even though we could do audio processing properly, something as small as a simple string for feedback took hours of debugging and development to get working properly. Near the end of the project, we had to cut out software features (like song detection) specifically because integrating it into the website would have taken too much effort for the time we had left. Next time, we would have definitely agreed on solid interfaces between all of our software components, and stick with it throughout the entire project. We would have also developed these separate components more in tandem from the very beginning, as we started integrating all of our systems rather late.

V. Contributions of each member of team

Team member	Contribution	Effort
Hudson Hall	Fretboard and LCD interface, wrote the majority of mcu firmware. Found and converted guitar tab files online for use by our system. Wrote code for interpreting guitar tab files. Used prior music knowledge to play guitar and ensure correctness of sound. Wrote note information for scales and chords.	20%
Zawad Munshi	Did all of the mechanicals and hardware testing for all of the components. Designed the PCBs for the neck. Assisted with other various parts of the project, like LCD code, audio code, and overall integration of the system.	20%
Owen Park	Developed the web application with API endpoints and database integration as the user interface. Integrated the application with the STM32 over UART. Ensured reliable software-hardware integration.	20%
Akshaya Naapa Ramesh	Sound detection code (basic pitch configuration, parsing MIDI data & pybind configuration, Chord & Song mode interface, starter code for uploading MIDI/tabs to database), helped debug LCD code	20%
Neel Vora	Completed main PCB design, including part selection, schematic design, layout, and soldering. Worked on overall physical system architecture and wiring. Also worked on electrical prototyping and some aspects of mechanical design.	20%

VI. Cost of Manufacture

Our cost for peripheral components on the PCB was $48.03. The cost of the individual STM32H562 was $6.41. For the Raspberry Pi 5 and its corresponding SD card adds up to around $82. For the 6 displays the total cost was $107.94. The speaker and power supply totalled up to $38 and the actual PCB manufacture cost $33 with shipping so factoring that out, the cost for production of the PCB was listed at $3 without any parts included. And finally, the cost of the guitar which we had at $50, although the quality we realized was not the best so ideally we would upgrade there. So overall, for production without shipping the cost included is around $335. With a more refined product we believe this is a reasonable price, which should likely

drop by about another $40-$50 dollars from bulk parts should we manufacture it at a larger scale. We feel that since decent quality electric guitars can be bought for around $200 dollars, the additions we've made for this is a suitable price once mass produced. Although slightly pricier it waves the cost of a learning subscription cost having it all integrated into the guitar.

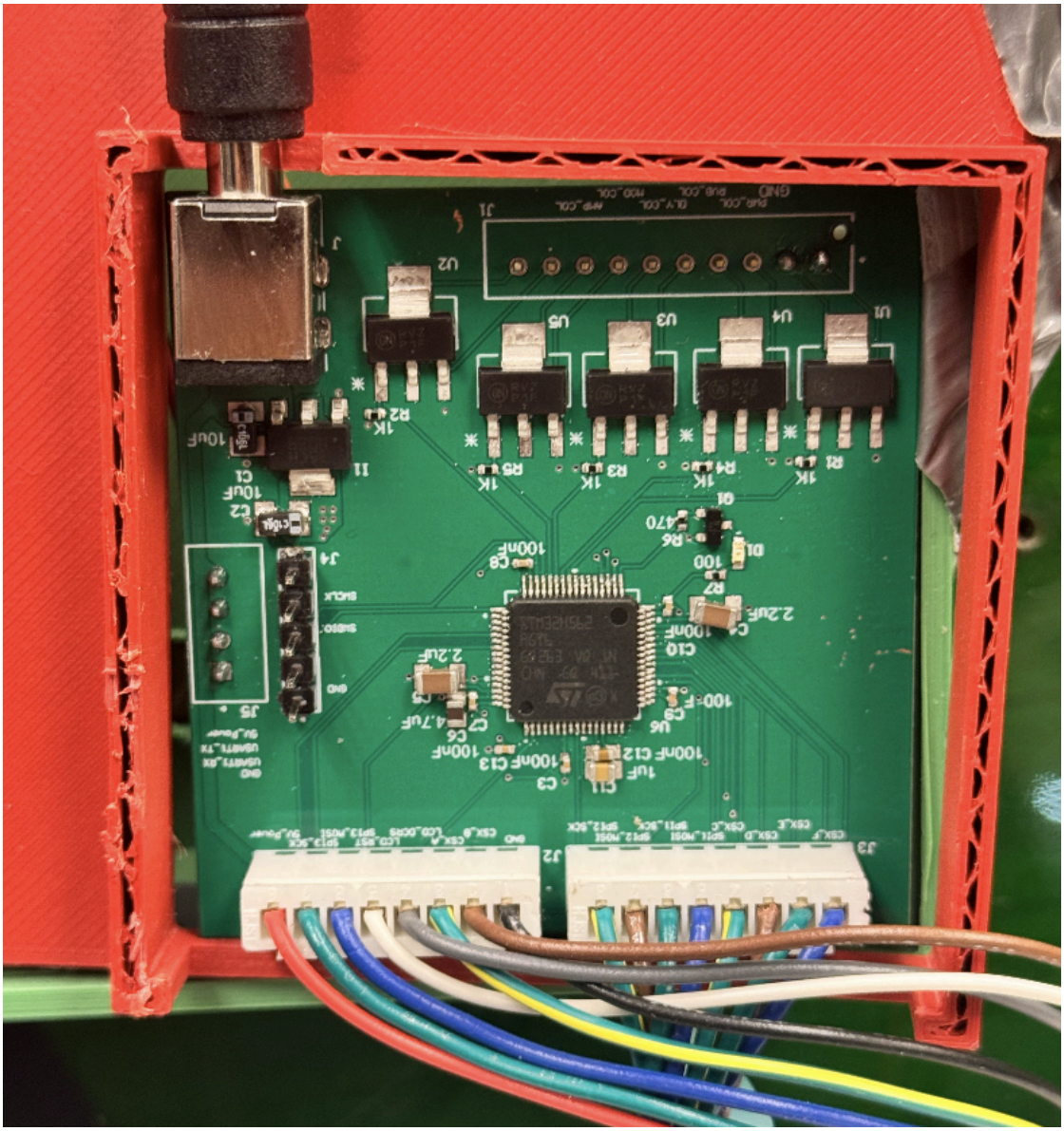
VII. Parts and Budget

The total cost here represents the total including our PCB cost, but since we used JLC balance instead of a card we couldn't add it to this sheet but the total cost of $572.90 does reflect this cost included. Overall, we were under our total budget of $1000 ($200 x 5) by a considerable amount.

Number of Group Members:	5		**Team Name:**	Guitar				**Running Total**	$572.90
Purchaser	**Purchase Date**	**Source**	**Item**	**Item Description**	**Quantity**	**Price**	**Tax**	**Shipping**	**Total Cost**
Owen Park	9/27/2024	Mouser	NUCLEO-H503RB	STM32H5 Nucleo	2	$32.90	$1.97	$7.99	$42.86
Zawad Munshi	9/15/2024	Micro Center	Raspberry Pi 5	Raspberry Pi 5 w/ 8	1	$79.99	$4.78	$0.00	$84.77
Zawad Munshi	9/15/2024	Micro Center	128gb SD Card	Inland 128GB Micro	1	$11.99	$0.70	$0.00	$12.69
Zawad Munshi	9/15/2024	Music Go Round	Glarry Strat Copy	Electric Guitar	1	$49.99	$3.00	$0.00	$52.99
Zawad Munshi	10/6/2024	Amazon	3.5" Display	Touch Screen TFT L	6	$107.94	$6.48	$0.00	$114.42
Zawad Munshi	11/3/2024	Amazon	3.5" Display	Touch Screen TFT L	1	$17.99	$1.08	0	$19.07
Neel Vora	11/11/2024	Digikey	PCB Components	All components for I	Various quantities	$48.03	$4.74	$16.99	$69.76
Neel Vora	11/11/2024	Mouser	STM32H5s	MCU	3	$19.23	$14.99	$1.15	$35.57
Neel Vora	12/1/2024	Digikey	PCB Connectors	Reorder of PCB cor	Various quantities	$6.33	$0.38	$13.99	$20.70
Zawad Munshi	11/6/2024	Amazon	SONICAKE Guitar H	SONICAKE Guitar H	1	$36.99	$2.22	$0.00	$39.21
Zawad Munshi	11/24/2024	Amazon	SABRENT USB Exte	SABRENT USB External Stereo Sound		$6.99	$0.62	$0.00	$7.61
Neel Vora	11/11/2024	Amazon	Power Supply	5V, 50W Power Sup	1	$21.99	$1.14	$0.00	$23.13
Neel Vora	11/11/2024	Amazon	Speaker	Speaker	1	15.98	$1.14	$0.00	$17.12

VIII. References and Citations

Main PCB



Software Libraries

Library	License	Link
Basic Pitch	Apache 2.0	https://github.com/spotify/basic-pitch
MIDO	MIT	https://github.com/mido/mido
pybind11	3-Clause BSD	https://github.com/pybind/pybind11
Advanced Linux Sound Architecture (ALSA)	LGPL 2.1	https://github.com/alsa-project
Tailwind CSS	MIT	https://github.com/tailwindlabs/tailwindcss
htmx	0-Clause BSD	https://github.com/bigskysoftware/htmx
PyGuitarPro	LGPL 3.0	https://github.com/Perlence/PyGuitarPro
SQLite	Public Domain	https://github.com/sqlite/sqlite
SQLiteCPP	MIT	https://github.com/SRombauts/SQLiteCpp
Compile Time Regular Expressions (ctre)	Apache 2.0	https://github.com/hanickadot/compile-time-regular-expressions
{fmt}	MIT (with extra clause)	https://github.com/fmtlib/fmt
cpp-httplib	MIT	https://github.com/yhirose/cpp-httplib
WiringPi	LGPL 3.0	https://github.com/WiringPi/WiringPi
TuxGuitar	LGPL 2.1	https://github.com/helge17/tuxguitar

Elisabeth Gadzic (a friend and University of Michigan Biomedical Engineering Student) assisted with mechanicals, CAD, and some manual labor like soldering headers)

University of Michigan Wilson Student Team Project Center

Interfaces

hardware.hpp

```
#pragma once

#include <cstdint>

#include "main.h"

class Pin {
public:
		Pin() = default;

		Pin(GPIO_TypeDef* port, std::uint16_t pin) : m_port{port}, m_pin{pin} {}

		[[nodiscard]] auto read() const -> bool;

		auto write(bool val) const -> void;

		auto set() const -> void;
		auto reset() const -> void;

private:
		GPIO_TypeDef* m_port{};
		std::uint16_t m_pin{};
};

class SPI {
		static constexpr uint32_t m_timeout = HAL_MAX_DELAY;

public:
		SPI() = default;

		SPI(SPI_HandleTypeDef* hspi, Pin csx) : m_hspi(hspi), m_csx(csx) {
		m_csx.set(); // CS is active low
		}

		auto spi_write(uint8_t const* data, uint16_t const size) const -> void;

		auto spi_write_stay_selected(uint8_t const* data, uint16_t const size) const -> void;

		auto spi_write_long(uint8_t const* data, std::size_t size) const -> void;

		auto spi_read(uint8_t* data, uint16_t const size) const -> void;

private:
		SPI_HandleTypeDef* m_hspi{};
		Pin m_csx{};
}
```

amp.hpp

#pragma once

```
#include "hardware.hpp"

class Amplifier {
public:
   Amplifier() = default;

   Amplifier(Pin amp, Pin mod, Pin dly, Pin rvb, Pin pwr)
      : m_amp(amp), m_mod(mod), m_dly(dly), m_rvb(rvb), m_pwr(pwr) {}

   void power_on() {
      m_mod.reset();
      m_dly.reset();
      m_rvb.reset();

      m_pwr.set();
      HAL_Delay(2000);
      m_pwr.reset();

      m_amp.set();
      HAL_Delay(2000);
      m_amp.reset();
   }

private:
   Pin m_amp;
   Pin m_mod;
   Pin m_dly;
   Pin m_rvb;
   Pin m_pwr;
};
```

lcd.hpp

```
#pragma once

#include <cstdint>
#include <vector>

#include "hardware.hpp"

/*---------------Various other control signals---------------------*/
#define ILI9486_SLPOUT 0x11 // Sleep mode off
#define ILI9486_DISPON 0x29 // Turn display on
#define ILI9486_CASET 0x2A  // Coloumn Address Set
#define ILI9486_PASET 0x2B  //
#define ILI9486_RAMWR 0x2C


/*-----------------LCD Screen pixel dimensions--------------------*/
#define ILI9486_TFTWIDTH  320
#define ILI9486_TFTHEIGHT 480

/*----------------------Color Definitions-------------------------*/
// 16 Bits for Color
constexpr uint16_t WHITE = 0xFFFF;
constexpr uint16_t BLACK = 0x0000;
constexpr uint16_t GRAY = 0x18c3;
constexpr uint16_t RED = 0xF800;
constexpr uint16_t GREEN = 0x07E0;
constexpr uint16_t BLUE = 0x001F;
constexpr uint16_t YELLOW = 0xFFE0;

/**
 * Coordinates of pixels on LCD array
 */
struct pixel_location_t {
        uint16_t x;
        uint16_t y;
};

/**
 * Interface for the ILI9486 LCD controller
 */
class LCD {
public:
        LCD() = default;

        LCD(SPI const &spi, Pin const &reg_sel, Pin const &reset) : m_spi(spi), m_reg_sel(reg_sel), m_reset(reset) {}

        auto init() const -> void;

        /**
        * @brief Fills the entire screen with a color
        * @param color fill color of screen
        */
        auto fill_screen(uint16_t color) const -> void;

        /**
        * @brief Clears the screen to be all white
        */
        auto clear_screen(bool dark_mode) const -> void;

        /**
        * @brief Draws a horizontal line that spans across the full width of the screen
        * @param pos starting position of line (left)
        * @param h height of the line in pixels
        * @param color fill color of rectangle
        */
        auto draw_horizontal_line(pixel_location_t pos, uint16_t h, uint16_t color) const -> void;

        /**
        * @brief Draws a filled color rectangle
        * @param pos top left position of rectangle
        * @param w horizontal width of rectangle in pixels
        * @param h vertical height of rectangle in pixels
        * @param color fill color of rectangle
        */
        auto draw_rectangle(pixel_location_t pos, uint16_t w, uint16_t h, uint16_t color) const -> void;

        /**
        * @brief Draws a filled color rectangle
        * @param pos top left position of bitmap
        * @param w horizontal width of bitmap in pixels
        * @param h vertical height of bitmap in pixels
        */
        auto draw_bitmap(pixel_location_t pos, uint16_t w, uint16_t h, const std::vector<uint8_t> &bitmap) const -> void;


        /*------------BACKDOOR FUNCTIONS---------------*/
```

```
        auto send_command(uint8_t command) const -> void;
        auto send_data(uint8_t data) const -> void;
        auto send_data_long(uint8_t const *data, std::size_t size) const -> void;

        // set_addr_window requires that no other non image SPI data is sent after this, until transaction is over
        // bounds of column and row are inclusive
        auto set_addr_window(uint16_t x0, uint16_t y0, uint16_t x1, uint16_t y1) const -> void;

        auto start_reset() const -> void;
        auto end_reset() const -> void;
        auto set_command() const -> void;
        auto set_data() const -> void;
        auto noop() const -> void;


private:
        SPI m_spi{};
        Pin m_reg_sel{}; // LCD register select (command/data)
        Pin m_reset{};   // LCD reset
        static constexpr uint16_t width = ILI9486_TFTHEIGHT;  // width in pixels of LCD screen
        static constexpr uint16_t height = ILI9486_TFTWIDTH; // height in pixels of LCD screen

};
```

fretboard.hpp

```
#pragma once

#include <array>
#include <cstdint>

#include "chords.hpp"
#include "guitar.hpp"
#include "lcd.hpp"
#include "messaging.hpp"
#include "timing.hpp"

#include "main.h"

/**
 * Class for controlling the entire fretboard of the guitar, which spans multiple LCD screens.
 * Interfaced with coordinates on both "fretboard grid" or "pixel grid" as continous grids across LCD screens
 */
class Fretboard {
		static constexpr std::size_t NUM_LCDS = 6;
		static constexpr std::size_t NUM_FRETS = 23;
		static constexpr std::size_t NUM_STRINGS = 6;
		static constexpr uint16_t TOTAL_PIXEL_WIDTH = NUM_LCDS * ILI9486_TFTHEIGHT;
		static constexpr uint32_t WARNING_DELAY = timing::NOTE_WARNING_DELAY.count(); // Delay between green, yellow, red in ms

		struct NoteLocationRectangle {
				pixel_location_t pixel_loc;
				uint16_t w;
				uint16_t h;
		};

		struct NoteTimestamp { // note to be played at timestamp x
				NoteLocation note_loc;
				uint32_t timestamp;
				uint16_t color;
		};

		enum class uart_state {
				NEW_MSG, // Process a new header and message (2 bytes)
				SONG_ID,
				NOTE
		};


		std::array<LCD, NUM_LCDS> m_lcds{};
		std::size_t m_song_size{};
		std::array<NoteDataMessage, MAX_NOTES_IN_SONG> m_song{}; // elements in this array should be sorted
		std::size_t m_timestamps_size{}; // total size of m_timestamps
		std::size_t m_timestamp_idx{};   // index of next note to be displayed on the fretboard
		std::array<NoteTimestamp, MAX_NOTES_IN_SONG * 4> m_timestamps{}; // first display green, yellow, then clear with white (4 colors)
		UART_HandleTypeDef* m_huart;
		uint32_t m_song_count_ms; // keeps track of the amount of ms elapsed, incremented every 1 ms
		uint8_t m_uart_buf[sizeof(NoteDataMessage) + 1] = {0};
		uart_state m_uart_state;
		uint8_t m_song_id;
		bool m_playing_song; // true iff we are currently playing a song on the fretboard
		bool m_dark_mode = true;


public:
		Fretboard() = default;

		/**
		 * @brief Constructor for Fretboard with 6 LCDs
		 *
		 */
		Fretboard(LCD const& lcd_1, LCD const& lcd_2, LCD const& lcd_3, LCD const& lcd_4, LCD const& lcd_5, LCD const& lcd_6, UART_HandleTypeDef* huart)
		: m_lcds{lcd_1, lcd_2, lcd_3, lcd_4, lcd_5, lcd_6}, m_huart(huart) {}


		/**
		 * @brief Initializes LCD screens on the fretboard
		 *
		 */
		auto init() -> void;

		/**
		 * @brief Writes a circle to a specific location on the fretboard
		 * @param fretboard_location coordinates of note on fretboard grid
		 * @param color color of circle to be written
		 */
		auto draw_note(NoteLocation note_location, uint16_t color) -> void;


		/**
		* @brief draws indicator to play open string  TODO: determine what kind of indicator
		* @param string string to play
```

```
        * @param color color of indicator
        */
        auto draw_string(string_e string, uint16_t color) -> void {
        uint16_t pixel_y;
        uint16_t height;
        convert_string_to_y(string, pixel_y, height); // Gives values to pixel_y and height
        m_lcds[0].draw_horizontal_line({0, pixel_y + height / 8}, height / 2, color);
        }

        auto piano_tiles() -> void;


        /**
        * @brief Handles a uart message, called on interrupt
        * @note Make sure state transition happens before call to interrupt
        */
        auto handle_uart_message() -> void;

        /**
        * @brief increments counter and checks to see if we have to display a note
        */
        auto handle_song_time() -> void;

        auto handle_uart_error() -> void;

private:
        /**
        * @brief converts note location to note pixel location
        * @param note_location note location to be converted
        * @return pixel coordinate of note
        */
        static auto convert_note_to_pixels(NoteLocation note_location) -> NoteLocationRectangle;

        /**
        * @brief Clears the fretboard LCDs
        */
        auto clear() -> void;

        /*
        * @brief converts string to height and pixel_location.y
        * @param y string to be converted
        * @param pixel_y changes in this function, starting pixel location y
        * @param height changes in this function, height to next pixel
        */
        static auto convert_string_to_y(string_e y, uint16_t& pixel_y, uint16_t& height) -> void;

        /**
        * @brief increments counter and checks to see if we have to display a note
        */
        auto process_loaded_song() -> void;

        /**
        * @brief Corrects m_timestamps when switching light mode/dark mode
        */
        auto process_color_mode() -> void;

        auto send_ack() -> void;
        auto rec_new_msg() -> void;
};
```

guitar.hpp

```
#pragma once

#include <cstdint>

/**
 * @enum string_e
 * @brief Represents guitar string identifiers.
 */
enum class string_e : std::uint8_t {
    LOW_E = 0x00,
    A = 0x01,
    D = 0x02,
    G = 0x03,
    B = 0x04,
    HIGH_E = 0x05,
};

/**
 * @typedef fret_t
 * @brief Represents the fret number on a guitar.
 *
 * Holds values in the range [0-23].
 * - Fret `0`: Open string (no fretting).
 * - Fret `1`: Closest to the tuning pegs.
 * - ...
 * - Fret `23`: Closest to the guitar body.
 */
using fret_t = std::uint8_t;

/**
 * @struct NoteLocation
 * @brief Represents a location on the guitar fretboard.
 */
struct NoteLocation {
    fret_t fret;
    string_e string;
};
```

messaging.hpp

```
#pragma once

#include <array>
#include <cstdint>

// Maximum data size for the message payload.
constexpr std::size_t MAX_DATA_SIZE = 6;

// Maximum number of notes in a song
constexpr std::size_t MAX_NOTES_IN_SONG = 50;

/**
 * @enum MessageType
 * @brief Defines various message types for communication.
 */
enum class MessageType : std::uint8_t {
   // clang-format off
   None           = 0x00, ///< Default
   Reset          = 0x41, ///< Reset the device
   Clear          = 0x42, ///< Clear the screens

   StartSongLoading = 0x52, ///< Start loading a song with a Song ID
   EndSongLoading   = 0x53, ///< End loading of the song
   Note           = 0x54, ///< Send a note with timestamp, fret, and string
   StartSong      = 0x55, ///< Start playing the loaded song
   EndSong        = 0x56, ///< End playing the loaded song
   RequestSongID    = 0x57, ///< Request the song ID
   LoadedSongID     = 0x58, ///< Send the loaded song ID

   DarkMode       = 0x60, ///< Set Dark Mode
   LightMode      = 0x61, ///< Set Light Mode (default)



   HoldAMajorChord  = 0x80, ///< Display A Major Chord indefinitely
   HoldCMajorChord  = 0x81, ///< Display C Major Chord indefinitely
   HoldDMajorChord  = 0x82, ///< Display D Major Chord indefinitely
   HoldEMajorChord  = 0x83, ///< Display E Major Chord indefinitely
   HoldFMajorChord  = 0x84, ///< Display F Major Chord indefinitely
   HoldGMajorChord  = 0x85, ///< Display G Major Chord indefinitely

   ACK            = 0x06  ///< Acknowledgment message
   // clang-format on
};

using ControlMessage = std::array<std::uint8_t, 2>;

constexpr std::uint8_t MESSAGE_HEADER = 0x01;

constexpr ControlMessage RESET_MESSAGE            = {MESSAGE_HEADER, static_cast<uint8_t>(MessageType::Reset)};
constexpr ControlMessage CLEAR_MESSAGE            = {MESSAGE_HEADER, static_cast<uint8_t>(MessageType::Clear)};
constexpr ControlMessage START_SONG_LOADING_MESSAGE = {MESSAGE_HEADER, static_cast<uint8_t>(MessageType::StartSongLoading)};
constexpr ControlMessage END_SONG_LOADING_MESSAGE   = {MESSAGE_HEADER, static_cast<uint8_t>(MessageType::EndSongLoading)};
constexpr ControlMessage NOTE_MESSAGE             = {MESSAGE_HEADER, static_cast<uint8_t>(MessageType::Note)};
constexpr ControlMessage START_SONG_MESSAGE       = {MESSAGE_HEADER, static_cast<uint8_t>(MessageType::StartSong)};
constexpr ControlMessage END_SONG_MESSAGE         = {MESSAGE_HEADER, static_cast<uint8_t>(MessageType::EndSong)};
constexpr ControlMessage REQUEST_SONG_ID_MESSAGE    = {MESSAGE_HEADER, static_cast<uint8_t>(MessageType::RequestSongID)};
constexpr ControlMessage LOADED_SONG_ID_MESSAGE     = {MESSAGE_HEADER, static_cast<uint8_t>(MessageType::LoadedSongID)};
constexpr ControlMessage DARK_MODE_MESSAGE        = {MESSAGE_HEADER, static_cast<uint8_t>(MessageType::DarkMode)};
constexpr ControlMessage LIGHT_MODE_MESSAGE       = {MESSAGE_HEADER, static_cast<uint8_t>(MessageType::LightMode)};
constexpr ControlMessage HOLD_A_MAJOR_CHORD_MESSAGE = {MESSAGE_HEADER, static_cast<uint8_t>(MessageType::HoldAMajorChord)};
constexpr ControlMessage HOLD_C_MAJOR_CHORD_MESSAGE = {MESSAGE_HEADER, static_cast<uint8_t>(MessageType::HoldCMajorChord)};
constexpr ControlMessage HOLD_D_MAJOR_CHORD_MESSAGE = {MESSAGE_HEADER, static_cast<uint8_t>(MessageType::HoldDMajorChord)};
constexpr ControlMessage HOLD_E_MAJOR_CHORD_MESSAGE = {MESSAGE_HEADER, static_cast<uint8_t>(MessageType::HoldEMajorChord)};
constexpr ControlMessage HOLD_F_MAJOR_CHORD_MESSAGE = {MESSAGE_HEADER, static_cast<uint8_t>(MessageType::HoldFMajorChord)};
constexpr ControlMessage HOLD_G_MAJOR_CHORD_MESSAGE = {MESSAGE_HEADER, static_cast<uint8_t>(MessageType::HoldGMajorChord)};
constexpr ControlMessage ACK_MESSAGE              = {MESSAGE_HEADER, static_cast<uint8_t>(MessageType::ACK)};

struct StartSongLoadingDataMessage {
   std::uint8_t song_id;
};

struct __attribute__((packed)) NoteDataMessage {
   std::uint32_t timestamp_ms;
   std::uint16_t length_ms;
   std::uint8_t fret : 4;
   std::uint8_t string : 4;
};

struct LoadedSongDataMessage {
   std::uint8_t song_id;
};
```

timing.hpp

```
#pragma once

#include <chrono>

namespace timing {
   using namespace std::chrono_literals;

   constexpr std::chrono::milliseconds LCD_BOOTUP_TIME = 2000ms;
   constexpr std::chrono::seconds SONG_START_DELAY = 3s;
   constexpr std::chrono::milliseconds NOTE_WARNING_DELAY = 300ms;
   constexpr unsigned int UART_BAUDRATE = 9600;

} // namespace timing
```

mcu.hpp

```
#pragma once

#include <cstdint>
#include <mutex>
#include <string>

#include <fmt/format.h>

#include "data.hpp"
#include "messaging.hpp"
#include "serial.hpp"

namespace mcu {
  using song_id_t = std::uint8_t;

  constexpr std::chrono::seconds ACK_TIMEOUT = std::chrono::seconds(5);

  class NoACKException : public std::exception {
  private:
    std::string m_message;

  public:
    explicit NoACKException(std::string const& details) : m_message(fmt::format("Did not receive ACK! {}", details)) {}

    [[nodiscard]] auto what() const noexcept -> char const* override { return m_message.c_str(); }
  };

  class NoMsgException : public std::exception {
  private:
    std::string m_message;

  public:
    explicit NoMsgException(std::string const& details) : m_message(fmt::format("Did not receive message! {}", details)) {}

    [[nodiscard]] auto what() const noexcept -> char const* override { return m_message.c_str(); }
  };

  class UnexpectedMsgException : public std::exception {
  private:
    std::string m_message;

  public:
    explicit UnexpectedMsgException(std::string const& details) : m_message(fmt::format("Received Unexpected message! {}", details)) {}

    [[nodiscard]] auto what() const noexcept -> char const* override { return m_message.c_str(); }
  };

  extern Serial serial;

  auto send_reset() -> void;
  auto send_clear() -> void;
  auto send_song(data::songs::SongInfo const& song) -> void;
  [[nodiscard]] auto get_loaded_song_id() -> std::uint8_t;
  auto play_loaded_song() -> void;
  auto end_loaded_song() -> void;

  auto hold_major_chord(MessageType chord_message_type) -> void;

  // Backdoor
  auto send_control_message(ControlMessage const& message) -> void;

  extern std::mutex mut;
  extern song_id_t current_song_id;
  extern bool playing;
} // namespace mcu
```

serial.hpp

```
#pragma once

#ifdef DEBUG
#include <iostream>
#endif
#include <chrono>
#include <cstddef>
#include <cstdint>

#include <wiringSerial.hpp>

#include "timing.hpp"

class Serial {
public:
   Serial() = default;
   ~Serial();

   auto init() -> bool;

   auto send(std::uint8_t const data) const -> void;
   auto send(std::uint8_t const* buffer, std::size_t size) const -> void;

   auto receive(std::uint8_t* buffer, std::size_t size, std::chrono::seconds timeout = std::chrono::seconds(3)) const -> bool;

   auto flush() const -> void;


private:
   static std::string constexpr m_dev_name = "/dev/ttyAMA0";

   int m_fd = -1;
};
```

data.hpp

```
#pragma once

#include <chrono>
#include <string>
#include <vector>

#include <SQLiteCpp/SQLiteCpp.h>

#include "guitar.hpp"

namespace data {
   extern std::string const data_directory;
   extern std::string const db_filename;

   auto init() -> bool;

   /**
    * @brief Checks if a directory exists and creates it if it does not exist
    *
    * @param directory the full pathname of the directory to check
    */
   auto create_directory_if_not_exists(std::string const& directory) -> bool;

   namespace songs {

      struct __attribute__((packed)) Note {
         std::uint32_t start_timestamp_ms;
         std::uint16_t length_ms;
         std::uint8_t midi_note;
         std::uint8_t fret : 4;
         std::uint8_t string : 4;
      };

      struct SongInfo {
         std::uint8_t id;
         std::string title;
         std::string artist;
         std::chrono::seconds length = std::chrono::seconds(0);
         std::uint16_t bpm;
         std::vector<Note> notes;
      };

      extern std::string const song_table_name;
      extern std::string const song_table_schema;

      auto create_table_if_not_exists(SQLite::Database& db) -> void;
      auto insert_new_song(SQLite::Database& db, SongInfo const& song) -> void;
      auto get_all_songs(SQLite::Database& db) -> std::vector<SongInfo>;
      auto song_id_exists(SQLite::Database& db, std::uint8_t song_id) -> bool;
      auto get_song_by_id(SQLite::Database& db, std::uint8_t song_id) -> SongInfo;

      static SongInfo const ode_to_joy =  {
                  // Song content omitted in report
      };

      namespace chords {
                  // Chord content omitted in report
         SongInfo const a_major_chord = {};
         SongInfo const c_major_chord = {};
         SongInfo const d_major_chord = {};
         SongInfo const e_major_chord = {};
         SongInfo const f_major_chord = {};
         SongInfo const g_major_chord = {};

      } // namespace chords

      namespace scales {
                  // Scale content omitted in report
         SongInfo const c_major_scale = {};
         SongInfo const d_major_scale = {};
         SongInfo const g_major_scale = {};
         SongInfo const f_major_scale = {};
      } // namespace scales


   } // namespace songs

} // namespace data
```

Appendix

Appendix A - draw_rectangle()

```
/**
 * @brief Draws a filled color rectangle
 * @param pos top left position of rectangle
 * @param w horizontal width of rectangle in pixels
 * @param h vertical height of rectangle in pixels
 * @param color fill color of rectangle
 */
auto draw_rectangle(pixel_location_t pos, uint16_t w, uint16_t h, uint16_t color) const -> void {
   if ((pos.x >= width) || (pos.y >= height)) return;
   if ((pos.x + w - 1) >= width) w = width - pos.x; // if our rectangle extends past the horizontal dimensions of the screen
   if ((pos.y + h - 1) >= height) h = height - pos.y;
   set_addr_window(pos.x, pos.y, pos.x + w - 1, pos.y + h - 1);

   uint8_t r = (color & 0xF800) >> 11;
   uint8_t g = (color & 0x07E0) >> 5;
   uint8_t b = color & 0x001F;

   r = (r * 255) / 31;
   g = (g * 255) / 63;
   b = (b * 255) / 31;
   set_data();
   for (int i = 0; i < w * h; ++i) {
      m_spi.spi_write_stay_selected(&r, 1);
      m_spi.spi_write_stay_selected(&g, 1);
      m_spi.spi_write_stay_selected(&b, 1);
      }
      noop(); // NOOP, reset csx, and end data stream
   }
```

Appendix B - Message Types

```
enum class MessageType : std::uint8_t {
   None           = 0x00, ///< Default
   Reset          = 0x41, ///< Reset the device
   Clear          = 0x42, ///< Clear the screens

   StartSongLoading = 0x52, ///< Start loading a song with a Song ID
   EndSongLoading   = 0x53, ///< End loading of the song
   Note           = 0x54, ///< Send a note with timestamp, fret, and string
   StartSong      = 0x55, ///< Start playing the loaded song
   EndSong        = 0x56, ///< End playing the loaded song
   RequestSongID    = 0x57, ///< Request the song ID
   LoadedSongID     = 0x58, ///< Send the loaded song ID

   DarkMode       = 0x60, ///< Set Dark Mode
   LightMode      = 0x61, ///< Set Light Mode (default)



   HoldAMajorChord  = 0x80, ///< Display A Major Chord indefinitely
   HoldCMajorChord  = 0x81, ///< Display C Major Chord indefinitely
   HoldDMajorChord  = 0x82, ///< Display D Major Chord indefinitely
   HoldEMajorChord  = 0x83, ///< Display E Major Chord indefinitely
   HoldFMajorChord  = 0x84, ///< Display F Major Chord indefinitely
   HoldGMajorChord  = 0x85, ///< Display G Major Chord indefinitely

   ACK            = 0x06  ///< Acknowledgment message
};
```

Appendix C - draw_note()

```
/**
 * @brief Writes a circle to a specific location on the fretboard
 * @param fretboard_location coordinates of note on fretboard grid
 * @param color color of circle to be written
 */
auto draw_note(NoteLocation note_location, uint16_t color) -> void {
    NoteLocationRectangle note_location_rectangle = convert_note_to_pixels(note_location);
    uint16_t lcd_index_2 = (note_location_rectangle.pixel_loc.x + note_location_rectangle.w) / 480;
                // In both cases, we want to draw a rectangle on the first LCD, our
    // if condition will confine the set_addr_window to not go beyond the bounds of the screen
    if (lcd_index_1 < NUM_LCDS) {
      m_lcds[lcd_index_1].draw_rectangle(
          {static_cast<uint16_t>(note_location_rectangle.pixel_loc.x % 480), note_location_rectangle.pixel_loc.y},
          note_location_rectangle.w, note_location_rectangle.h, color);
    }
    if (lcd_index_1 != lcd_index_2) {
      // Note goes across screens
      if (lcd_index_2 < NUM_LCDS) {
        m_lcds[lcd_index_2].draw_rectangle({0, note_location_rectangle.pixel_loc.y},
                                           (note_location_rectangle.pixel_loc.x + note_location_rectangle.w) % 480, note_location_rectangle.h,
                                           color);
      }
    }
}
```

Appendix D - process_loaded_song()

```
/**
 * @brief increments counter and checks to see if we have to display a note
 */
auto process_loaded_song() -> void {
    for (size_t i = 0; i < m_song_size; ++i) { // parse through notedata and populated timestamps
      auto const& note_data = m_song[i];
      m_timestamps[m_timestamps_size++] = {{static_cast<fret_t>(note_data.fret), static_cast<string_e>(note_data.string)},
                                           note_data.timestamp_ms,
                                           RED};
      m_timestamps[m_timestamps_size++] = {{static_cast<fret_t>(note_data.fret), static_cast<string_e>(note_data.string)},
                                           note_data.timestamp_ms + WARNING_DELAY,
                                           YELLOW};
      m_timestamps[m_timestamps_size++] = {{static_cast<fret_t>(note_data.fret), static_cast<string_e>(note_data.string)},
                                           note_data.timestamp_ms + WARNING_DELAY * 2,
                                           GREEN};
      m_timestamps[m_timestamps_size++] = {{static_cast<fret_t>(note_data.fret), static_cast<string_e>(note_data.string)},
                                           note_data.timestamp_ms + WARNING_DELAY * 2 + note_data.length_ms,
                                           m_dark_mode ? BLACK : WHITE};
    }
    // Insertion sort since the array is already mostly sorted
    for (int i = 0; i < m_timestamps_size; ++i) {
      int swapped = i;
      for (int j = i + 1; j < m_timestamps_size; ++j) {
        if (m_timestamps[j].timestamp < m_timestamps[swapped].timestamp) swapped = j;
      }
      std::swap(m_timestamps[i], m_timestamps[swapped]);
    }
}
```

Appendix E - parse_gp_file()

```
# parses gpx file and then outputs vector of ints [[start_timestamp_ms, length_ms, note value, int fret, int string]]
# fret == 0 means open string
def parse_gp_file(gp_file_path):
    song = guitarpro.parse(gp_file_path)

    debug_print(f"Available tracks:")
    debug_print(f"Song tempo: {song.tempo}")
    for track in song.tracks:
        debug_print(f"Track {track.number}: {track.name}")
    debug_print("------------------------------------------------") # empty line
    song_info_out = []
    track = song.tracks[0] # the first track is usually the correct one / the main part
    timestamp = 0.0
    for measure in track.measures:
        for voice in measure.voices:
            for beat in voice.beats:
                # Iterate over notes in the beat
                for note in beat.notes:
                    midi_note_val = string_dict[note.string-1] + note.value # string is 1-indexed
                    fret_number = note.value
                    string_number = note.string - 1 # string is 1-indexed
                    length = beat.duration.time
```

```
            _, _, note_type = str(note.type).partition('.')

            debug_print(f"Start timestamp: {timestamp:.2f} ms, Length: {length}, MidiNote val: {midi_note_val}, Fret: {fret_number}, String: {string_number}, NoteType: {note_type}")
            song_info_out.append([int(timestamp), int(length), int(midi_note_val), int(fret_number), int(string_number)])

        # Increment timestamp
        timestamp += beat.duration.time
  return song_info_out
```

Appendix F - ChordAnalyzer

```
class ChordAnalyzer {
public:
  ChordAnalyzer() : m_running(true), m_worker_thread(&ChordAnalyzer::worker_thread_function, this) {}

  ~ChordAnalyzer() {
    return future;
  }

private:
  void worker_thread_function() {
    py::scoped_interpreter guard{};
    auto sys = py::module::import("sys");
    sys.attr("path").attr("insert")(0, PY_VENV_PATH);
    sys.attr("path").attr("insert")(0, PY_MODULE_PATH);
    auto get_record_convert_module = py::module::import("record_convert");


        }
            std::string command = "arecord --duration=" + std::to_string(reference_chord.length.count()) + " --rate=88200 --format=S16_LE " +

            py::object get_prediction = get_record_convert_module.attr("prediction");
            get_prediction(reference_chord.id, 120);
            py::object get_record_convert = get_record_convert_module.attr("record_convert_offset");
            auto numbers = get_record_convert(reference_chord.id).cast<std::vector<std::vector<int>>>();

            py::object get_remove_files = get_record_convert_module.attr("remove_files");
   std::atomic<bool> m_running;
  std::thread m_worker_thread;
  std::queue<std::pair<data::songs::SongInfo, std::promise<bool>>> m_request_queue;
  std::mutex m_queue_mutex;
  std::condition_variable m_condition;
};
```

Appendix G - Static Source File Generation

Script

```
#!/bin/bash

# Argument parsing
if [ "$#" -ne 2 ]; then
	echo "Usage: $0 <source-directory> <output-file>"
	exit 1
fi

SOURCES_DIR="$1"
OUTPUT_FILE="$2"

generate_content_constexpr() {
	local path=$1
	local lc_name=$2

	local content=$(tr -d '\n' < "$path")
	local content_name=${lc_name}_content

	echo "constexpr char const $content_name[] = R\"RAW($content)RAW\";"
}

generate_iter_macro() {
	local lc_name=$1
	local file_name=$2
	local uc_name=$(echo "$lc_name" | tr 'a-z' 'A-Z')

	local content_name=${lc_name}_content

	echo "_F($uc_name, $content_name, \"$file_name\", __VA_ARGS__)\\"
}

# Clear the output file
> "$OUTPUT_FILE"

echo "Getting source files..."
file_paths=$(find "$SOURCES_DIR" -type f)

for file_path in $(find "$SOURCES_DIR" -type f); do
	file_name=$(basename "$file_path")

	echo "$file_path"
done

{
	echo "#pragma once"
	echo ""
	echo "namespace web {"

	for file_path in $(find "$SOURCES_DIR" -type f); do
	file_name=$(basename "$file_path")
	# Replace non-alphanumeric characters with underscores to form a valid variable name
	formatted_name=$(echo "$file_name" | sed -e 's/[^a-zA-Z0-9]/_/g')

	generate_content_constexpr "$file_path" "$formatted_name"
	done

	echo "#define SOURCE_FILES_ITER(_F, ...)\\"

	for file_path in $(find "$SOURCES_DIR" -type f); do
	file_name=$(basename "$file_path")
	formatted_name=$(echo "$file_name" | sed -e 's/[^a-zA-Z0-9]/_/g')

	generate_iter_macro "$formatted_name" "$file_name"
	done

	echo ""
	echo "} // namespace web"
} >> "$OUTPUT_FILE"

echo "Web sources generated into $OUTPUT_FILE"
```

Example Output

```
#pragma once

namespace web {
	constexpr char const htmx_js_content[] = R"RAW({CONTENT HERE!})RAW";
	constexpr char const style_css_content[] = R"RAW({CONTENT HERE!})RAW";
	constexpr char const htmx_min_js_content[] = R"RAW({CONTENT HERE!})RAW";
	constexpr char const compiled_css_content[] = R"RAW({CONTENT HERE!})RAW";
	constexpr char const songinfo_html_content[] = R"RAW({CONTENT HERE!})RAW";
	constexpr char const chordselectform_html_content[] = R"RAW({CONTENT HERE!})RAW";
	constexpr char const button_html_content[] = R"RAW({CONTENT HERE!})RAW";
	constexpr char const error_html_content[] = R"RAW({CONTENT HERE!})RAW";
	constexpr char const songselectform_html_content[] = R"RAW({CONTENT HERE!})RAW";
	constexpr char const htmx_rt_js_content[] = R"RAW({CONTENT HERE!})RAW";
	constexpr char const index_html_content[] = R"RAW({CONTENT HERE!})RAW";
	constexpr char const htmx_sse_js_content[] = R"RAW({CONTENT HERE!})RAW";
#define SOURCE_FILES_ITER(_F, ...)                                                                  \
	_F(HTMX_JS, htmx_js_content, "htmx.js", __VA_ARGS__)                                            \
	_F(STYLE_CSS, style_css_content, "style.css", __VA_ARGS__)                                      \
	_F(HTMX_MIN_JS, htmx_min_js_content, "htmx.min.js", __VA_ARGS__)                                \
	_F(COMPILED_CSS, compiled_css_content, "compiled.css", __VA_ARGS__)                             \
	_F(SONGINFO_HTML, songinfo_html_content, "songinfo.html", __VA_ARGS__)                                  \
	_F(CHORDSELECTFORM_HTML, chordselectform_html_content, "chordselectform.html", __VA_ARGS__)     \
	_F(BUTTON_HTML, button_html_content, "button.html", __VA_ARGS__)                                \
	_F(ERROR_HTML, error_html_content, "error.html", __VA_ARGS__)                                   \
	_F(SONGSELECTFORM_HTML, songselectform_html_content, "songselectform.html", __VA_ARGS__)        \
	_F(HTMX_RT_JS, htmx_rt_js_content, "htmx.rt.js", __VA_ARGS__)                                   \
	_F(INDEX_HTML, index_html_content, "index.html", __VA_ARGS__)                                   \
	_F(HTMX_SSE_JS, htmx_sse_js_content, "htmx.sse.js", __VA_ARGS__)

} // namespace web
```