MIE438

Project Report - The SynthBoard

Click here to view project video

STM32CubeIDE project .zip folder can be found in Quercus submission

Gal Cohen 1007757768
Nisha Ghai 1008131718
Kevin Qu 1007897180
Allen Tao 1008012507
Alex Wang 1008076172

April 12, 2024

1 The Problem

1.1 Original Goals and Plans for Project

Traditional synthesizers, like the Roland SH-101, play a pivotal role in the sound design and music industry. However, they suffer prohibitive weaknesses in price and real-time audio manipulation for live performances. Thus, by introducing a new electronic instrument that is both accessible for aspiring musicians and primed for live performance, the Launchpad has seen an explosion in popularity, characterized by its illuminated button matrix and extensive customization options. As a drawback, they don't offer the same propensity for raw audio generation and customization like a traditional synthesizer, hampering the artist's creative flexibility.

Hence our original intention is to create a small-scale, open-source audio "SynthBoard" that offers the best of both worlds. In particular, we aim to implement the following features: grid layout button matrix to trigger audio, configurable LED lighting patterns, ability to generate waveforms, ability to manipulate waveforms, volume control, and speaker output.

2 Changelog

The change(s) made between the initial proposal and the final report are tabulated below:

Before	After	Rationale
Translucent button matrix, allowing for full LED light passthrough	Backlit LED button matrix	Due to constraints on 3D printing services, translucent, flexible material was not available to print for our buttons. Instead, we simply don't provide a cover over the buttons, leaving them exposed and hence allowing total LED light visibility.
External built-in speaker	Exposed AUX port to connect audio peripherals	Due to power constraints, to power all the LEDs and microcontroller, there was not sufficient power to drive the speaker as well.

Otherwise, we were able to implement all the core features outlined in the proposal (Lighting, Notes, Basic Sound Customization, Volume Control, Power Supply Intake), and one bonus feature (Polyphony), with some exceptions detailed in section 6.2

3 Final Design and Operation

3.1 Hardware

All hardware design decisions in the following section were implemented in the Fusion360 software by AutoDesk.

3.1.1 Case

When it came to designing the case, we used a reference design from *GreatScott* [3], who built a 6x6 button launchpad, which we resized to 4x4. The relative dimensions of the case were preserved.

We then expanded on this design to account for the potentiometers. We are using 8 knobs, however we needed to make sure not to extend the case beyond our 3D printer's maximum printing dimensions. The printer we used was the Creality Ender 3 S1 Pro which had printing limits of 220 x 220 x 250 mm. Thus, we extended the design by 85 mm to make it 210 mm lengthwise. The width and height of the case are 121 mm and 46.5 mm respectively, both within printing boundaries.

Alongside the extension of the board, we measured the diameter of the potentiometers to be 6.8 mm. To account for uncertainty in the measurement and the printing process, 8 holes of diameter 7.8 mm were added to the top case (1mm diameter extension).

3.1.2 Clamps

To suspend the microcontroller underneath the potentiometers, we created clamps that would allow the pins to be exposed and easily connect to wires. We measured the thickness of the PCB to create the notch and adjusted the clamp height based on the top and bottom pins of the PCB.

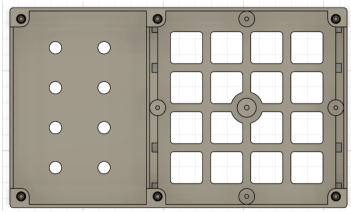

Figure 1: Extending design and adding potentiometer holes.

Initially, we found with the addition of the clamps that we did not have enough space to fit both the potentiometers and the microcontroller inside the case. As a result, the height of our design had to increase by 17.4 mm to its final height of 46.5 mm. This resulted in changing our originally planned 25 mm M5 screws to 40 mm.

3.1.3 LED Prisms

To allow for LEDs to shine when we press a button, we designed angled prisms that our LEDs can rest on while facing their respective buttons. Using 4 prisms for 4 rows of buttons, we provided slots for the prisms to be inserted into in the case in between the buttons.

3.1.4 Screws

We used an M4 screw at the center and M3 screws at each edge of the perfboard for securement. The M4's thicker diameter provided strong grip, while the M3's smaller size avoided excessive force. M5 screws were chosen for the top and bottom casing due to their ideal size and strength, with M6 screws deemed too large. Figure 1 shows the placement of these screws: M4 in the center, M3 at edges, and M5 at six corners.

3.1.5 Design Validation

Our design's integrity was validated through Autodesk Fusion360 by assembling all components in a single file. This process allowed us to check for interference, ensuring all parts—potentiometers, microcontroller, clamps, and LED prisms—fit correctly within the case. By simulating the assembly, we confirmed alignment, clearances, and the absence of conflicts, verifying the feasibility and functionality of our design efficiently.

3.1.6 3D Printing

We employed Cura for slicing and the Creality Ender 3 S1 Pro for printing, adjusting settings per component for optimal outcomes.

- **Case**: Layer height was set to 0.2 mm, infill to 30%, and wall thickness to 1.2 mm, balancing strength and printing efficiency.
- **Clamps and LED Prisms**: These smaller, precision parts were printed with a 0.1 mm layer height and 100% infill for accuracy and durability.
- **Temperature and Speed**: Printing temperature was 210°C, with speeds of 50 mm/s for larger parts and 40 mm/s for detailed components, optimizing for quality and detail.

3.2 Electrical

3.2.1 Button Grid

To wire a 4x4 grid of buttons in a pin efficient manner, a total of 8 pins were used, with 4 corresponding to each of the 4 rows, and the other 4 to the 4 columns. The microcontroller would then poll through each row while holding a particular column high, to determine if a button is pressed.

However, an issue that arises from this pin efficient wiring is "keyboard ghosting", wherein the pressing of several buttons will result in extra buttons closing the circuit that are not included in the original buttons pressed (figure 3).

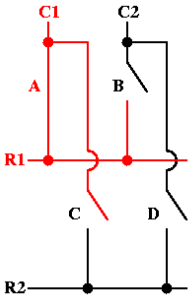

Figure 2: Single key press example. Switch A is pressed, completing the circuit between column 1 and row 1 [2].

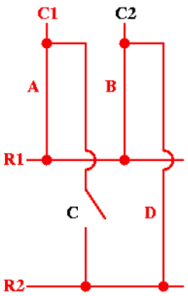

Figure 3: Keyboard ghosting: switches A, B, and D are closed, however C is incorrectly activated as well [2].

To resolve this, we wired diodes to the end of each switch. The whole circuit was soldered onto a perforated board for minimum form factor, durability, and low cost. Solid core cables were stripped and soldered to make electrical connections between the switches and diodes, and female end cables branch out of each row and each column to eventually connect to the male pin headers on the microcontroller.

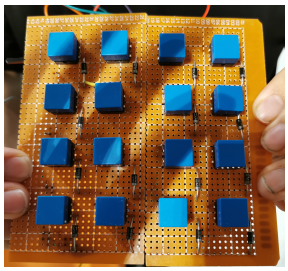

Figure 4: Soldered Perfboard Front Side

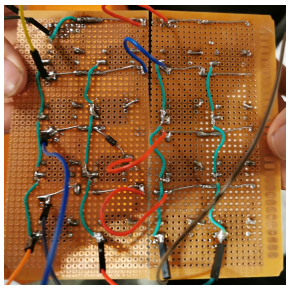

Figure 5: Soldered Perfboard Back Side

3.2.2 LEDs

To enable lighting effects around our buttons, we needed LEDs which support RGB and are of small form factor. We eventually settled on the WS2812B LEDs. They differ from traditional LED's by being surface mountable, providing the slim form factor required to clear each LED's behind the push buttons. Furthermore, an arbitrary number of LED's can be daisy chained together, whilst retaining the ability to address each one individually using its SPI protocol. This can be done whilst only requiring 3 cables, compared to traditional LEDs where each LED must have a dedicated pin for individual control, which is very wasteful.

To mount the LED's, we cut the LED's and soldered wires to space the LEDs' out to match the layout of the push buttons inside our case, as shown in figure 6 below. We take care to preserve the daisy chain configuration to eventually allow control over each of the 16 individual LEDs, using just 3 pins.

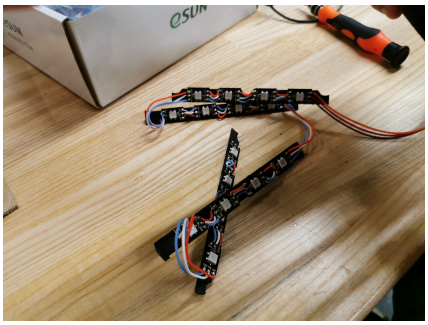

Figure 6: WS2812B LED Array

3.2.3 Potentiometers

We soldered jumper cables to the 3 terminals of the potentiometers. Each potentiometer is then wired to an ADC input on the microcontroller via its middle pin. The left pin was connected to the 3V power source on the microcontroller, and the right pin was connected to GND (when knob is facing upwards) [7].

3.3 Microcontroller

To choose a suitable microcontroller for our SynthBoard, we needed a platform which supports:

- At least 8 GPIO digital pins for our 4x4 button matrix
- Ideally at least 8 analog pins for our 8 potentiometers
- Audio DAC or speaker driver to output sound

The STM32F407G-DISC1 was sufficient in meeting all of the above specs [8], and was hence chosen for this project. Although several other microcontrollers could also satisfy our I/O requirements, what sets the STM32F407G-DISC1 apart from its competitors is its built in audio DAC with an integrated class D speaker driver, allowing us to directly connect external speakers or headphones to the board, without extra hardware.

4 Program Flow and Features

4.1 User Interaction / Program Flow

A diagram of how our SynthBoard interfaces with the user is presented below:

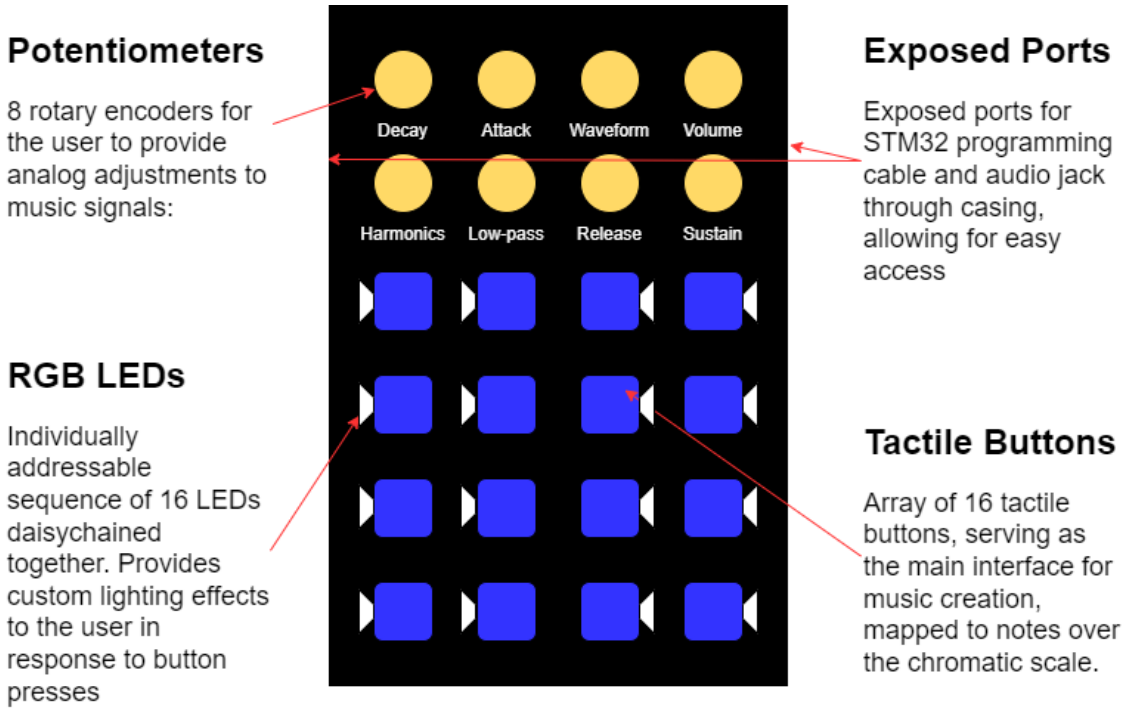

Figure 7: User Interaction with the SynthBoard

A system architecture diagram is attached in the appendix in figure 9.

4.2 Microcontroller Features Used

The following features of our microcontroller were used [7]:

1. Audio DAC with integrated class D speaker driver: used to drive the speaker through the available stereo headphone output jack [6].
2. 8 GPIO digital pins (PB0-2, PE7-11 in figure 11): used to interface with the grid of 16 push buttons. Each of the 4 rows and each of the 4 columns were hooked up to a port for port use efficiency.

3. 8 GPIO pins read using analog with an ADC converter (DMA) (PA0-7 in figure 11): used to read values from our 8 potentiometer knobs
4. SPI interface (with DMA) (PB15 in figure 11): used to interface with WS2812B LEDs.

4.3 Next Steps

There were several features of the microcontroller that can extend the functionality of the SynthBoard in the future. Some ideas include:

1. Built in digital microphone, allows the user to record custom audio clips and apply audio transformations.
2. Leftover pins and compute. This allows possibility to support more buttons and knobs. Also, implementing a sequencer, custom key remapping, and custom lighting profiles will improve customizability.

5 Program Structure

We used the STM32CubeIDE to program our STM32F407G-DISC1 microcontroller. This IDE is attractive since it is tailored towards STM32 series microcontrollers, and also includes a wide selection of developer tools including a compiler, debugger and code editor [5].

We chose to develop in C++. It supports more features and flexibility than C, while maintaining its ability for low level hardware control. In addition, we also leverage C++'s object oriented features to organize our large codebase, since it consists of several separate features corresponding to different audio and lighting effects we implement on our SynthBoard. Finally, C++ is very efficient in runtime, making it desirable for real time audio generation especially when implementing complex audio manipulation subroutines. This did introduce some complexity as the STM32CubeIDE does not support C++ only projects, meaning we had to call our main C++ loop from the 'main.c' file [7]. This also required special consideration when including our C++ files in the auto generated C file, and vice versa.

5.1 Overall Code Structure

We adhered to object oriented programming principles to organize our codebase. This includes defining objects for all major audio synthesis components: `ADSREnvelope`, `Oscillator`, `Synthesizer`, and `LowPassFilter`. In addition, external libraries were also included, encompassing audio generation drivers [9], and WS2812B LED wrapper functions for RGB lighting control [10].

5.2 Button Matrix

The 16-button matrix is individually controlled via 8 GPIO digital pins: 4 for each row, and 4 for each column. To then detect which button(s) are pressed, we employ a nested for loop. The outer loop iterates through the 4 column wires, and sets each pin to output mode, and its pull to `GPIO_NOPULL` (to avoid leaving it in an indeterminate state). The inner loop iterates through the 4 row wires, and sets each pin to input mode, and the pull to `GPIO_PULLUP`. At this point, if the button corresponding to this particular column and row is pressed, the input row pin would read 0. We store the state of each button in a 4x4 2D array, then reset the row pin to `GPIO_PULLDOWN`, before resuming the loop to check other buttons. By scanning through the whole button array like this, we are able to detect key presses for an arbitrary number of notes.

It's worth noting that switch debouncing is commonly needed for our switches. To resolve this issue, we limited the input sampling rate to 10 milliseconds. This prevents the program from reading oscillatory button presses caused by debouncing.

5.3 WS2812B LEDs

We interface with the WS2812B LED's using DMA. To do this, we enabled DMA with the SPI peripheral in the STM32CubeIDE. To control the LEDs, we send bits in accordance with the protocol baked into these LEDs. In particular, a 0 is encoded using a high pulse of $0.35\mu s$ following by a low pulse of $0.90\mu s$. A 1 is implemented with a high pulse of $0.90\mu s$, followed by a low pulse of $0.35\mu s$ [10]. Using an 8 bit SPI protocol, sending a 0 or 1 would then correspond to sending `0x80` or `0xFC` respectively. This was then wrapped in `ws2812_pixel` to allow us to control individual LED's and its RGB value in our code [10].

5.4 Potentiometers

Each of the 8 potentiometers is connected to one of the ADC (Analog-to-Digital Converter) channels on the STM32 microcontroller [7], without using a multiplexer setup. This direct connection is possible because the number of available GPIO pins is sufficient for our needs, allowing us to avoid the complexity and potential performance penalty of a multiplexer. The ADC channels are set up to continuously convert the analog voltage levels into digital values, which are then stored in an array, `adc1Buffer`. These values are periodically read and processed in the `getInputs` function in `InputHandler.cpp` to adjust the synthesizer's settings.

5.4.1 Initialization and Configuration

The initialization of the ADC1 peripheral, detailed in `MX_ADC1_Init()` within `main.c`, configures the ADC for continuous conversion mode with a resolution of 8 bits. This configuration allows for a range of 0-255 in digital output, which captures the variations in potentiometer settings from minimum to maximum. The initialization ensures that the ADC reads multiple channels and stores the results in `adc1Buffer` using DMA to efficiently handle data transfers without CPU intervention. This configuration is selected to balance accuracy with processing efficiency, as the potentiometers do not require high precision due to their manual operation.

- **Clock Configuration:** The ADC clock is set to run at one-eighth of the PCLK2, which is configured to 84 MHz in the Clock Configuration GUI, effectively running the ADC at 10.5 MHz.
- **Continuous Conversion Mode:** We employ a polling mechanism rather than interrupt requests to minimize CPU cycle usage. This mode uses a timer to trigger ADC conversions approximately every 10 ms, which is sufficient for user interaction while optimizing performance and energy consumption.
- **DMA Configuration:** DMA is set to circular mode, which automates the process of reading ADC values into a buffer without CPU intervention. This setup enhances system efficiency by continuously updating the buffer with new readings in a cycle.

5.4.2 Reading and Using Analog Inputs

In the `EventLoopCpp()` function within `event_loop.cpp`, the ADC is started in DMA mode, which automatically populates `adc1Buffer` with the latest readings from the potentiometers. The `getInputs()` function is periodically called to update the `analogInputs` array from this buffer. During this update, interrupts are temporarily disabled to ensure data consistency, avoiding mid-read changes. The separation of hardware interaction and application logic through different levels of abstraction ensures maintainability and performance.

5.4.3 Application to Synthesizer Parameters

Each potentiometer is assigned to control a specific parameter of the synthesizer:

- `analogInputs[0]` is used to adjust the overall volume of the output. It scales the output volume linearly with the potentiometer setting.
- `analogInputs[1]` and `analogInputs[7]` control the waveform and harmonic content of the sound, respectively, affecting the timbre and color of the audio output.
- `analogInputs[2-5]` are used by the ADSR envelope (`ADSREnvelope`), influencing the attack, decay, sustain, and release phases of the sound, which shape the sound's amplitude envelope over time.
- `analogInputs[6]` controls the cutoff frequency of the low-pass filter (`LowPassFilter`), shaping the frequency of the synthesized sound by attenuating high-frequency components based on the knob's position.

5.5 Audio Manipulation

To support waveform generation and manipulation, classes supporting the desired functionality as methods were implemented. This section will cover each of these classes as well as their functionality.

5.5.1 Attack Decay Sustain Release (ADSR)

The `ADSREnvelope` class plays a critical role in the dynamic sound manipulation within our audio synthesizer, managing the Attack, Decay, Sustain, and Release (ADSR) phases of sound amplitude for each note. This section outlines the algorithms and functionalities implemented in the `ADSREnvelope.hpp`.

- **Attack Phase:** The attack phase is the first stage of the sound's lifecycle, where the amplitude increases from zero to its peak. If the attack time is set to one or less, the amplitude reaches the maximum

instantaneously (65280, derived from the input range maximum of 255 being scaled by 256). For longer attack times, the amplitude ascends linearly, calculated by 65280/(`attack` × `SAMPLE_RATE`/`factor`).

- **Decay Phase:** Following the attack, the decay phase begins, where the sound's amplitude reduces from its peak to the specified sustain level. If the decay parameter is one or less, the amplitude directly adjusts to the sustain level (`sustain` × 256). For more prolonged decays, the amplitude decreases linearly using the formula (65280 − `sustain` × 256)/(`decay` × `SAMPLE_RATE`/`factor`) until it stabilizes.
- **Sustain Phase:** The sustain phase maintains the sound at a constant amplitude level, which is held as long as the note is active. There are no dynamic changes during this phase; the sound continues at the decay's concluding amplitude level until the note release is initiated.
- **Release Phase:** The final phase of the ADSR envelope is the release, where the sound's amplitude decays back to zero after the note is released. If the release duration is one or less, the amplitude is set to zero immediately. For longer releases, the amplitude reduces progressively at a rate determined by `sustain` × 256/(`release` × `SAMPLE_RATE`/`factor`). Once the amplitude reaches zero, the state is set to `Off`, marking the end of the sound's lifecycle.

5.5.2 Oscillator

Code related to generating raw audio is implemented in `Oscillator.hpp`. In particular, this contains the frequencies for notes starting from C4, spanning a little over full chromatic scale up to E flat, defined in `NoteFrequencies`. This is also responsible for implementing the generation of harmonics over the current note.

The `Waveform` enum simplifies the selection of waveform types, while waveform and harmonic settings are determined through shifts on control byte inputs.

The phase increment is designed to increment the phase while considering the sample rate, thus ensuring that the phase wraps correctly at its maximum value. This approach leverages the integer overflow to cycle the phase without additional logic checks.

5.5.3 Synthesizer

The `Synthesizer` class embodies the encapsulation principle, centralizing the waveform generation algorithms of 4 waveforms: sine wave, square wave, sawtooth wave, and triangle wave as seen in figure 10. This approach enhances the maintainability and scalability of the code.

The sine wave generation leverages a precomputed lookup table, `sineTable`, which is populated during initialization. This table drastically reduces the computational cost by avoiding repetitive calculations of the sine function, which would be computationally expensive to compute in real-time for each audio sample.

The waveform generation functions use the maximum and minimum values for a 16-bit signed integer. These values are crucial for scaling the waveform amplitudes to fit within the dynamic range of 16-bit audio. Additionally, the phase of the waveforms is used for determining their position within the cycle, and is represented as a 16-bit integer. This choice utilizes integer overflow, when the phase calculation exceeds 65535, it wraps around to zero, thereby mimicking the cyclic nature of waveforms without needing explicit modulo operations, and allowing us to only define one cycle for each waveform that maps to a 16 bit integer range.

Each waveform generation method calculates the signal by summing up the fundamental tone and its harmonics. This is based on additive synthesis principles where multiple sine waves at different frequencies are combined to produce complex sounds. The class maintains state internally with member variables like `sample`, and `scaledPhase`, which store intermediate results and loop counters. This localized state management helps in keeping the waveform generation methods stateless from an external perspective, allowing reusability.

5.5.4 Low Pass Filter

This class implements the low pass filter, whose corner frequency is determined based on the potentiometer knob position as determined by the user. This class enforces a minimum threshold on the cutoff frequency, which is a practical safeguard to prevent the filter from becoming non-functional. A low-pass filter is applied to the incoming audio signals, effectively smoothing out rapid changes in the audio stream by blending the current input with the previous output, weighted by a cutoff frequency. The cutoff frequency acts as a configurable parameter that determines the degree of filtering applied, making the filter adaptable to different operating conditions.

5.6 Audio Output

5.6.1 Audio Buffer

The audio output is read from a continuously-updated buffer that is sent to the DAC over the I2S communication protocol. To enable continuous sound playback, the data is sent in chunks, specifically at half the buffer at a time. Every time half of the buffer is played, the program triggers a callback function, `AUDIO_OUT_HalfTransfer_CallBack` and `AUDIO_OUT_TransferComplete_Callback`, for when the entire buffer has been played. This fills the buffer with new data at each callback. By using a chunked buffer, there is no artifact or delay associated with the program computing parts of the buffer upon completion of a full buffer being sent and having to wait for new, correct data. The buffer is also filled with two entries at a time because the built-in audio system on the STM32 board supports stereo outputs while our speaker setup only outputs mono [1]. Thus, we fill the buffer with the same data for two consecutive entries.

In addition, each callback checks if the time has surpassed a certain threshold without any user input. If this occurs, interrupts are disabled and pause any more interrupts from happening until the next user input. This feature ensures that the program remains responsive without being negatively affected by nested interrupts.

5.6.2 DAC conversion

Once data is sent to the DAC, it will then convert the data to analog and then play the audio through the built-in class D audio amplifier. Currently, the DAC is configured to operate at 32KHz, well above what is required by the Nyquist-Shannon sampling criterion [11]. 32KHz was selected because at higher frequencies, we begin to run into performance issues. The highest available note is E5, which is roughly 660Hz. Even with the harmonics feature turned to its maximum, the highest frequency wave that is being played is at $660 * 8 \approx 5.3KHz$, well below the sampling rate that we are operating at.

5.7 Time Handling

The time variable keeps track of the relative time since the system began processing audio, which is essential for handling audio buffer updates and managing user inputs effectively. In each call to the `fillAudioBuffer()`, the variable is updated by calculating the amount of time elapsed based on the buffer size and the sample rate, adjusting by a scaling factor related to note playback, used to tune the note frequencies. This ensures that time progresses in sync with the audio output.

5.8 Code Optimization

5.8.1 Inline Functions and Macros

Highly iterated functions related to waveform generation, including the `nextSample`, `getActiveNotesCount`, `generateSineWave`, `generateSquareWave`, etc. methods, which are called in each loop iteration over the entire audio buffer, are declared as inline functions to reduce function overhead, while minimizing code size. On the STM32 we are much more constrained with respect to RAM rather than storage, hence these optimizations bode well for our target platform in terms of runtime performance [7].

For the WS2812B library, we defined `WS2812_FILL_BUFFER(COLOR)` as a macro, since it is executed several times in the `ws2812_pixel` and `ws2812_pixel_all` wrappers which govern LED control. Here, we are increasing code memory usage to minimize overhead, which again is desirable for our particular STM32 platform [7].

5.8.2 Integer Arithmetic

Although as per [8], our STM32F407G-DISC1 does support floating point calculation through its FPU, we know from [4] that floating point calculations are regardless slower than integer arithmetic, and a trick can be employed using pre-multiplication to preserve significant figures while eliminating the need to define floats. Hence, an optimization we made in this spirit was to delay division to the very last step, instead performing all other integer arithmetic operations prior to this first. This allows us to manually hold the decimal places without having to explicitly define floating point numbers.

Examples of where this was applied includes the `generateTriangleWave` method inside `Sythnthesizer.hpp`.

5.8.3 Bitwise Operations

Bitwise operations are faster than typical multiplication and division operations. Hence wherever possible, bit shifts were performed in lieu of multiplication and division. For example, right shifts were used extensively in `Oscillator.hpp` instead of multiplication operations to optimize runtime. Example: `harmonics = 1 + (numHarmonics >> 5);` instead of `harmonics = 1 + (numHarmonics / 32);`.

5.8.4 Code Motion

Wherever possible, we factor out unchanging code inside any loop to the outside. An example is the for loop in `nextSample` inside `Oscillator.hpp`, where `SAMPLE_RATE` and `NOTE_SCALING` are defined as macros outside the loop.

5.8.5 Compiler Settings

To further optimize our code, we used a reduced standard C/C++ library to save space, as well as no debugging flags with `-Ofast` for improved performance. Our specific settings are shown in figure 12.

6 Testing and Validation

6.1 Testing

Testing was segmented into quantitative and qualitative measures to comprehensively evaluate the performance and usability of the synthesizer.

6.1.1 Quantitative Tests:

Quantitative testing focused on measurable parameters to ensure the synthesizer operates within the expected specifications:

- **Volume Range:** We measured the decibel levels at the minimum and maximum volume settings to confirm dynamic range adequacy [dB]. Due to lack of access to a powerful external speaker on testing day, we were relegated to outputting sound through earbuds. If hooked to an external speaker, this output level can be adjusted to an ample volume at the user's discretion.
- **Pitch Range:** Using a tuning app, we tested the lowest and highest pitches the device could generate, ensuring a broad tonal range. [Hz]. The results were in line with the frequencies hard coded for each note in `NoteFrequencies` inside `Oscillator.hpp`.
- **Harmonics / Polyphony:** We assessed the maximum number of concurrent notes or chords that could be played, verifying the synthesizer's capability for complex sound production [#]. In testing, we were able to play all 16 notes simultaneously, although this is not reliable due to compute issues, discussed in section 6.2.

6.1.2 Qualitative Tests:

Qualitative assessments were performed to evaluate the synthesizer's sensory and experiential aspects:

- **LED Effect Visibility:** We tested the visibility and impact of the LED lighting effects under various ambient conditions to ensure performance appeal.
- **Waveform Generation:** We compared the sound of our generated waveforms (sawtooth, triangle, square, sine) with those produced by industry-standard synthesizers via audio samples online.
- **Envelope Parameters:** Attack, sustain, and decay was tested against a professional synthesizer audio sample to confirm that they resemble the responsiveness and quality expected by users. We were not able to test release; reasons are documented in section 6.2.

6.2 Bugs and Commentary

6.2.1 Bugs

The main bug that we have is the fact that we are hitting the upper bound of our computing power. For example, polyphony of more than a few notes/higher harmonics causes the double buffer filling interrupt routines to not finish before half the buffer has already played, and the interrupts get nested, meaning user inputs aren't being read by the program and the audio has artifacts and clicking sounds as the data in the buffer is not in order.

We also ran into problems with programming two of the potentiometer effects: Release and Low-Pass Filter. This is why two of the eight features are not demonstrated in the demo video.

During the creation of the electrical circuit for the key matrix and potentiometers, we frequently ran into issues with poor solder connections, incorrect wiring or broken components. To solve such electrical problems, we approached it in two different ways. First, we extensively used multimeters to test for proper connections between various parts of the circuit. Secondly, we also used a breadboard and Arduino UNO loaded with a simple testing program to test components and our soldering.

The outputs from the various audio devices we tested were weak and nearly inaudible at lower volumes. We primarily worked with three devices: a pair of earbuds, a small JBL speaker, and an 8 Ohm DIY-grade speaker. The audio driver in the board was made for headphones and does a poor job of amplifying the sound, requiring all use to be in a quiet environment to be audible. The JBL speaker was powered externally to provide sufficient amplification and was audible even in a noisy environment.

6.2.2 Comments

Overall, our project was fun to create and try out since we were able to implement many relevant, real-world audio features. It was our first time working with such an involved system, requiring substantial development in CAD/3D printing, embedded programming and electrical design. It was also very rewarding to work with a modern platform like the STM32 and apply various embedded development techniques taught in class, creating a coherent and working final product.

If we were to create this again, we would stick with the same microcontroller and peripheral options. The features not implemented are possible to develop on our current platform, as the STM32 still harbours unused program space, RAM, and I/O pins which could be allocated to these features. However, higher clock speeds on our microcontroller would be desired to support compute intensive operations, like polyphony with more than 6 notes, which our current implementation struggles with.

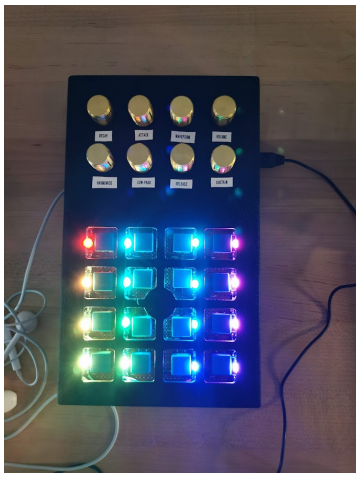

Figure 8: The completed SynthBoard with all the LEDs lit up

7 Future Improvements

- **ADC Injected Conversion Mode:** Considering this mode for future implementations could allow adding an offset to conversion results, eliminating divide-by-zero checks and speeding up the system.
- **Further Dividing the Clock Prescaler:** By further reducing the ADC conversion clock, we could decrease energy consumption without impacting performance due to the nature of user controls.
- **Waveform Customization:** By allowing the user to define a custom waveform, a much larger variety of sounds can be outputted.
- **Additional Tunable Parameters:** Further changes can be made in allowing fine-tuned sound generation, such as a customizable ADSR envelope shape and effects such as reverb and distortion.
- **MIDI Integration:** Allowing a user to both play and record their music in an exportable format into another music program would allow the SynthBoard to be used in music creation settings as well.

8 References

[1] Cirrus Logic. *CS43L22 Datasheet*. https://datasheet.octopart.com/CS43L22-CNZ-Cirrus-Logic-datasheet-5397077.pdf. Accessed: Jan. 22, 2024. 2008.

[2] D. Dribin. *Keyboard Matrix Help*. Accessed: Jan. 22, 2024. June 2000. URL: https://www.dribin.org/dave/keyboard/one_html/.

[3] GreatScottLab. *Make Your Own Launchpad*. Accessed: Jan. 22, 2024. 2024. URL: https://www.instructables.com/Make-Your-Own-Launchpad/.

[4] *MIE438 – Microprocessors and Embedded Microcontrollers: Laboratory Experiment 9*. Lab Guide. Department of Mechanical and Industrial Engineering. University of Toronto, 2024.

[5] *STM32CubeIDE*. https://www.st.com/en/development-tools/stm32cubeide.html. Accessed: Jan. 22, 2024.

[6] STMicroelectronics. *MB997-F407VGT6-B02 Schematic*. https://www.st.com/content/ccc/resource/technical/layouts_and_diagrams/schematic_pack/group1/0f/91/8b/39/b3/78/4d/c4/MB997-F407VGT6-B02_Schematic/files/MB997-F407VGT6-B02_Schematic.pdf/jcr:content/translations/en.MB997-F407VGT6-B02_Schematic.pdf. Accessed: Jan. 22, 2024. 2012.

[7] STMicroelectronics. *STM32F405RG Datasheet*. https://www.st.com/resource/en/datasheet/stm32f405rg.pdf. Accessed: Jan. 22, 2024. 2020.

[8] STMicroelectronics. *STM32F4DISCOVERY - STMicroelectronics*. https://www.st.com/en/evaluation-tools/stm32f4discovery.html. Accessed: Jan. 21, 2024.

[9] *WavePlayer using STM32 Discovery*. https://controllerstech.com/waveplayer-using-stm32-discovery/. Accessed: 2024-04-02. Jan. 2021.

[10] M. van der Werff. *Controlling WS2812(B) leds using STM32 HAL SPI*. https://www.newinnovations.nl/post/controlling-ws2812-and-ws2812b-using-only-stm32-spi/. Accessed: 2024-04-09. May 2021.

[11] Wikipedia. *Nyquist–Shannon sampling theorem*. Accessed: Jan. 22, 2024. 2024. URL: https://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem.

9 Appendix

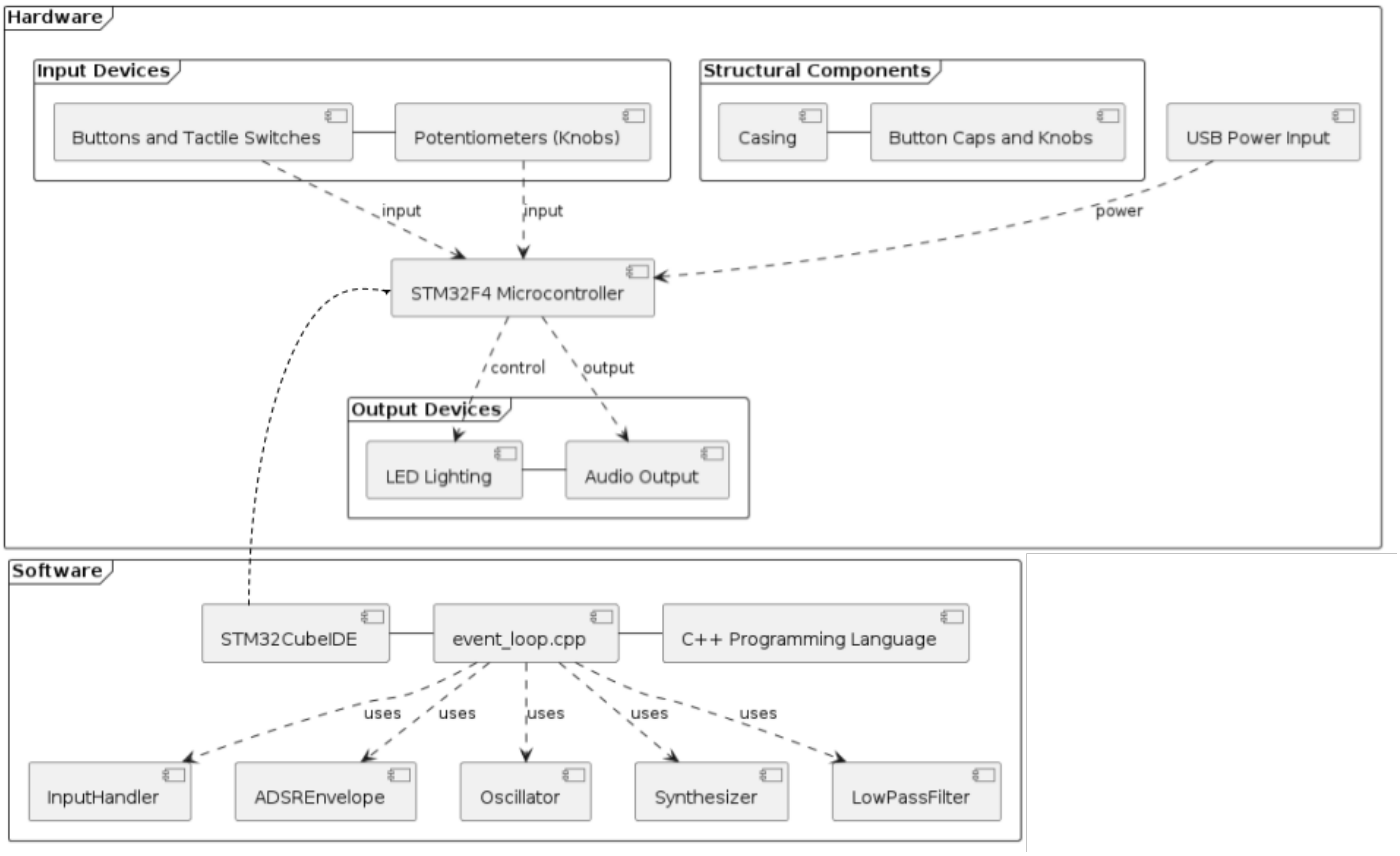

Figure 9: SynthBoard System Architecture.

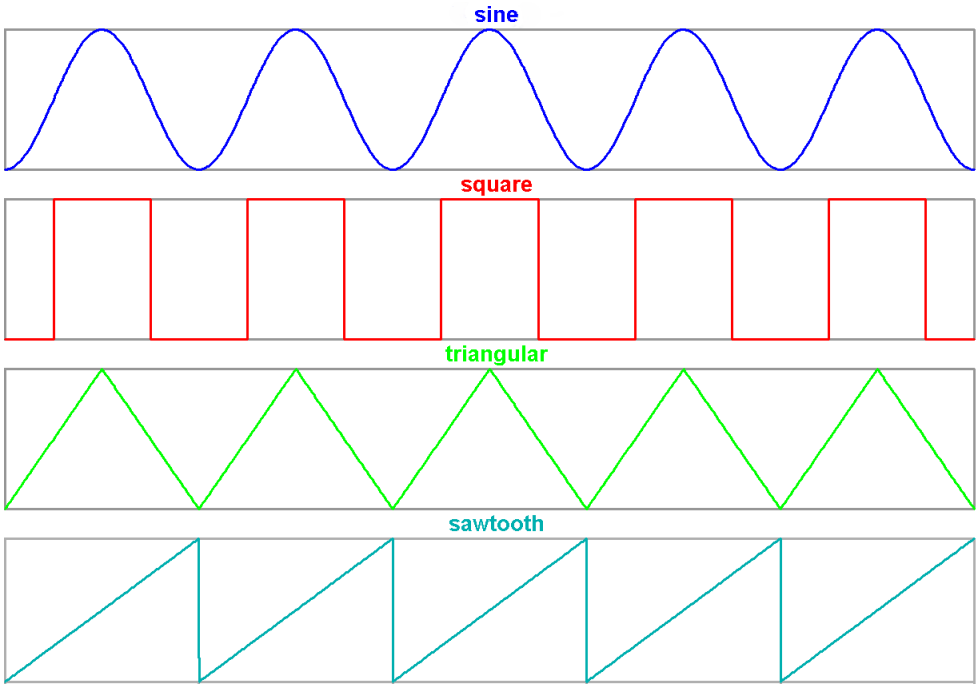

Figure 10: The waveform shapes the user can choose from.

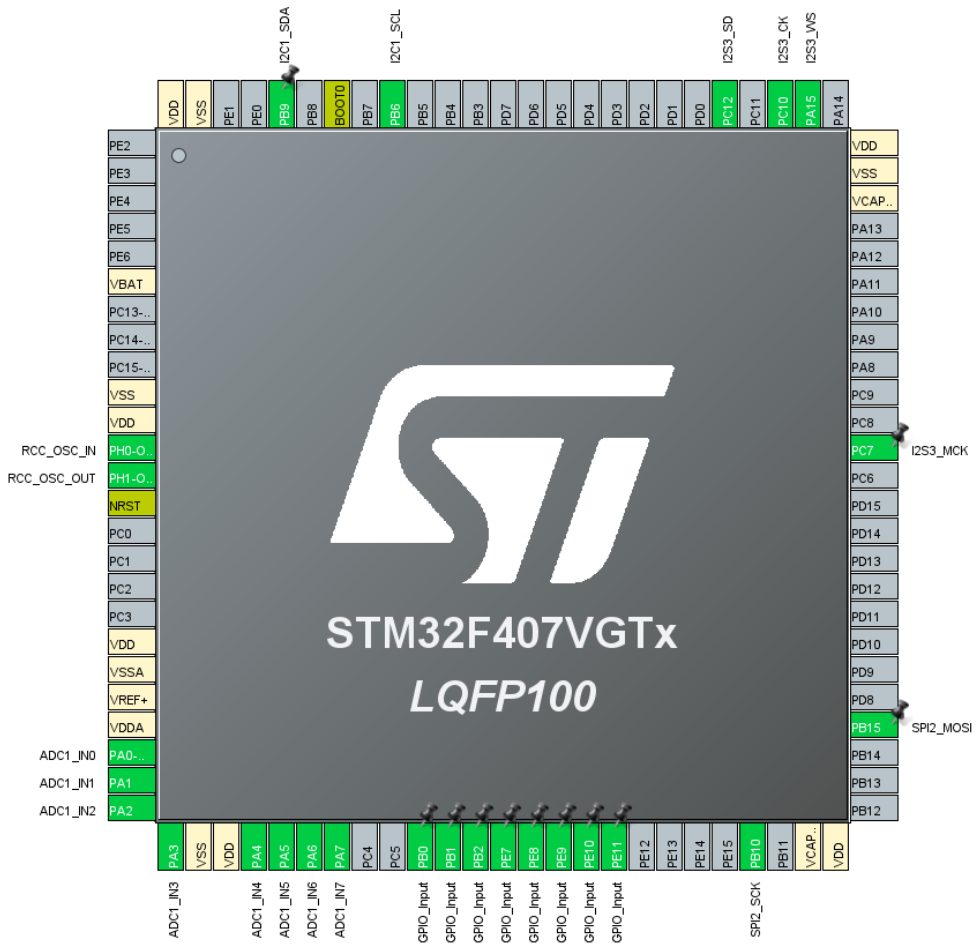

Figure 11: Pin mappings on STM32F407G-DISC1 microcontroller.

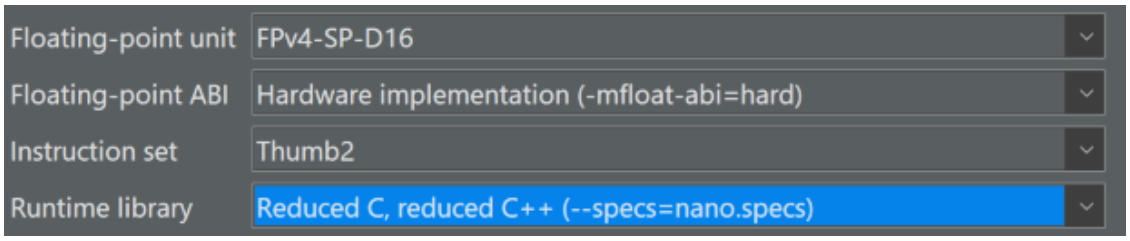

Figure 12: Compiler settings in STM32CubeIDE