Developing a more affordable open source digital synthesizer with the ESP32

Elizabeth Levin
January 2024

Table of Contents

Abstract

Physical musical synthesizers are often expensive, with entry level equipment costing a minimum of 100 dollars, which can be a steep entry point for beginners. In an effort to both make a cheaper hardware alternative, and to learn more about digital synthesis technology, I have developed an open source digital synthesizer for under 50 dollars. In this paper, I document the development process of this synthesizer, as well as leave resources for the reader to create their own version at home.

Background

Electronic synthesizers were first invented in 1964 simultaneously by Bob Moog and Don Buchla. These first synthesizers were analog synthesizers, using physical hardware and circuitry to create sounds and filters. (Deckert, 2024). Naturally, these synthesizers were often large and expensive. Today however, the invention of digital computers and microprocessors has allowed synthesizers to enter the digital world, with the first commercially available digital synthesizer being the Synclavier in 1975 (Deckert, 2024). Digital hardware allows for lighter weight and more versatile devices, since the chip can be programmed for many different types of operations. Additionally, digital hardware allows for the ability of saving preset parameters. However, even with the rapidly advancing technology, digital synthesizers remain relatively expensive. Entry level digital synthesizers cost a minimum of 100 dollars, which is not always accessible to beginners. I was inspired to create this device when I was deterred from buying a digital musical instrument due to the high price tag.

Criteria

I have set 5 criteria for this digital synthesizer:

1. Generate the 4 classic Synthesis waves: Sine, Square, Triangle, and Sawtooth
2. Have Polyphonic abilities (play multiple notes at once)
3. Generate Sound Envelope
4. Ability to connect to an external MIDI controller
5. Maintain a total cost of under 50 dollars

These criteria will provide necessary parameters in order to be playable and provide the user with enough flexibility to create different sounds.

Design

Physical Design

The design is meant to be light weight and user friendly. It is inspired by the design of the Nintendo Gameboy, as it captures the nostalgic 8-bit aesthetic reflected by the sounds produced by the synthesizer. There are 4 rotary encoders, used for adjusting sound parameters on the screen. The display is a 20x4 Character LCD with I2C interface. 5 buttons are included to the side of the display for navigation of the UI, however currently only the left and right buttons are in use for switching pages on the display. A 150mm soft membrane potentiometer is included at the bottom of the device. It can be used to slide notes up and down, or to play notes when there is no keyboard connected. A volume potentiometer is included to control the output level of the speaker. On the right side of the device is a USB port for connecting a MIDI controller, and underneath is a USB cable for connecting the device to power or to a computer for programming.

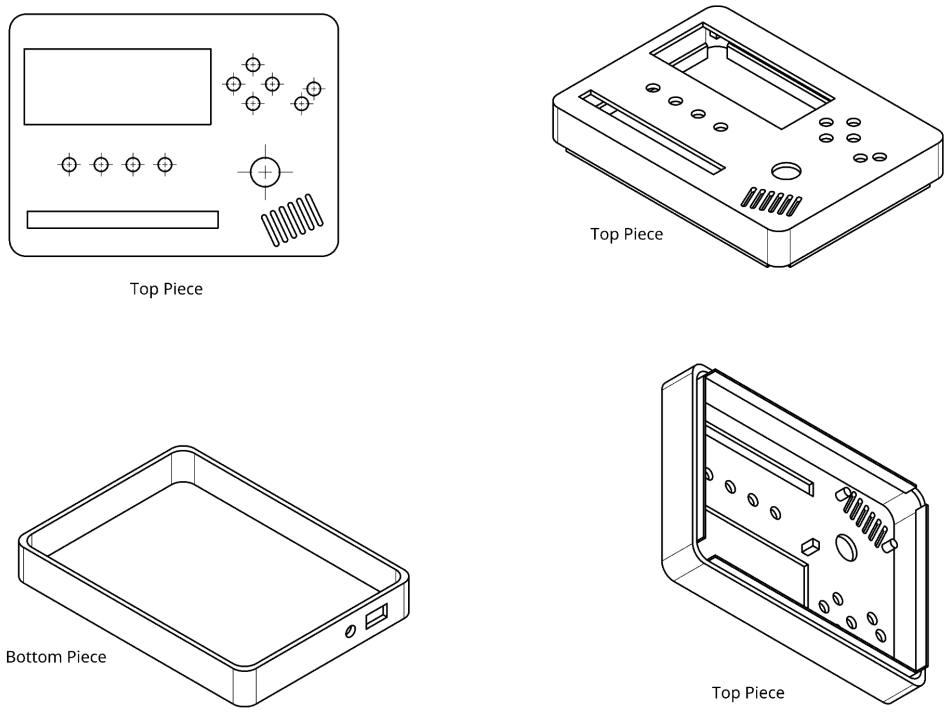


Figure 1: Synthesizer Housing Drawings

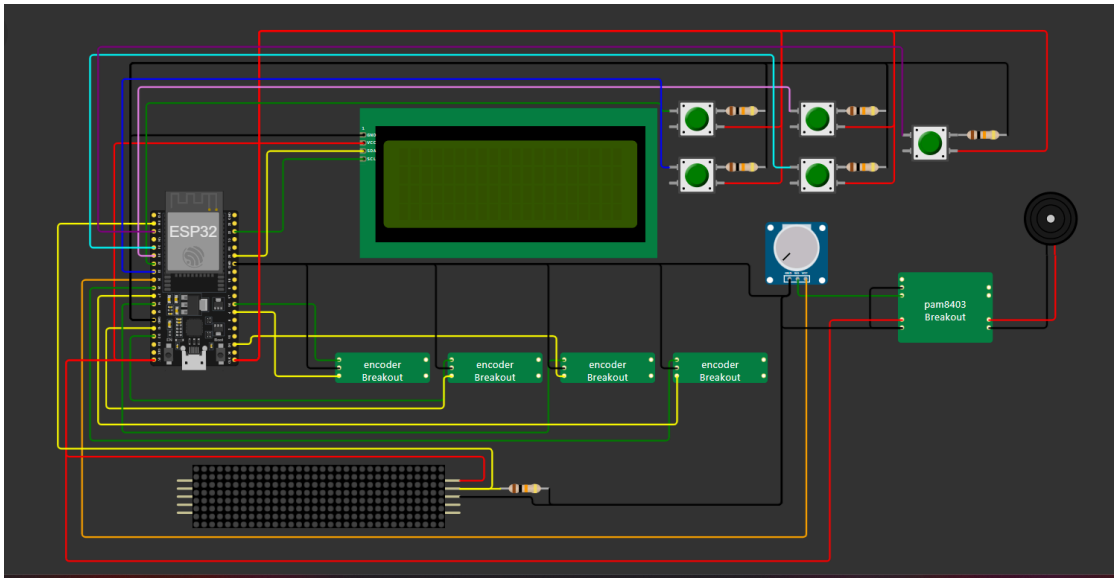


Figure 2: Wiring Schematic

Additional Wiring Diagrams with further details are provided in appendix A.

Program Architecture

In this Synthesizer, I run all audio synthesis and UI control on core 1 of the ESP32. On core 0, I handle USB communications with the MIDI Keyboard. Upon receiving a packet from the keyboard, the data is stored in a shared buffer, that is then accessed and interpreted by the process on core 1.

An interrupt timer runs continuously at a frequency of roughly 22khz (45 microsecond sampling time). In the ISR, the DAC output is updated and a flag is raised for the next calculation. Within the main loop, the next DAC output is calculated when the flag is raised, and upon finishing calculations, the flag is lowered until the ISR runs again. During Calculations, the current time is incremented by 45 (the sampling period) for all active oscillators. The MIDI queue is checked for new information, and interpreted. Upon the keys being pressed, the keys are assigned to currently inactive oscillators. The global pitch bend and modulation depth is also updated. The amplitude of the active oscillators is adjusted by the envelope, and the phase shift counter is adjusted by the Low frequency Oscillator to modulate the frequency of the output wave. After all of these parameters are calculated, they are used to calculate the output of the wave. After the outputs of all of the active Oscillators are calculated, they are fed into a volume mixer, where they are put together into a single output. This value is fed through the on board 8-bit DAC, and output to the speaker.

Development

Oscillators and Waveforms

The first step in subtractive synthesis is to generate a base waveform by using an Oscillator. In the analog world, synthesizers consisted of physical voltage controlled Oscillators to generate a waveform (Deckert, 2024). In my digital representation, I have a data structure that holds the amplitude, frequency, duty cycle, current time from key press, and waveform of the wave I want to generate. I also include other information like a shift counter for modulation (which I will cover in the modulation section), the calculated output of the waveform, and the key number being pressed.

```
typedef struct{
  int freq;
  u_int8_t amp;
  float duty;
  int currentTime;
  wave waveform;
  float shiftCounter;
  noteState state;
  int output;
  int key;

} Oscillator;
```

Figure 3: Oscillator Structure Definition

Although there are many waveforms that could be used for this project, I am using the 4 basic wave forms of Sine, Square, Triangle, and Sawtooth. These 4 waveforms were commonly used in early subtractive synthesizers, as they were simple and gave a good amount of variety in sound (Hass, 2023). I will now cover the calculations for each wave.

Square Wave

The square wave is the most simple of the 4 waves used in this project. The wave is pulled high (to the amplitude) for a certain percentage of the cycle, and pulled low (to 0) for the rest of the cycle. This repeated immediate switching of the output from low to high creates a square like shape in the waveform.

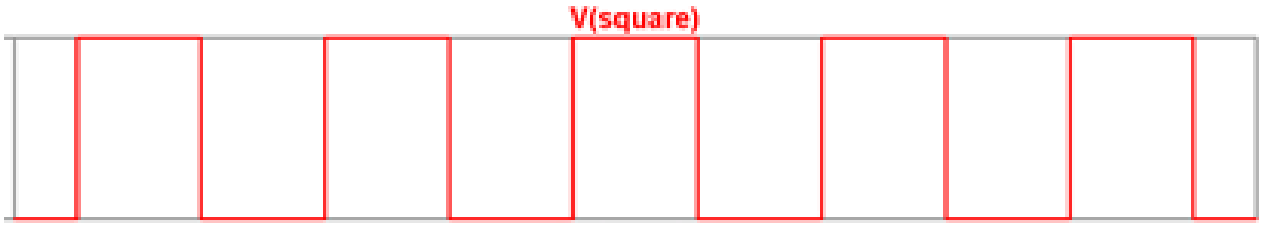


Figure 4: Square Wave (Alonso, 2016)

The amount of time that the output stays at high is represented by the duty cycle. At a duty cycle of 0.5, the wave spends half of the time at high, and the other half at low. Setting the cycle to 0.2 will cause the wave to spend a 5th of the cycle at high, and the rest of the cycle at low.

Triangle Wave

The triangle wave is a simplified representation of the sine wave, where the function linearly approaches and moves away from the amplitude.

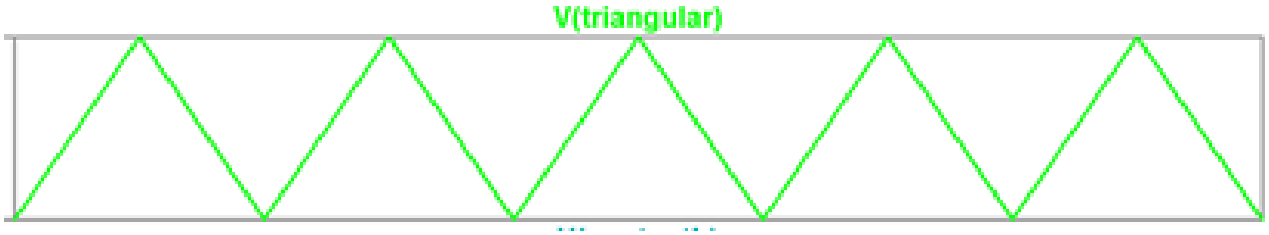


Figure 5: Triangle Wave (Alonso, 2016)

It can be calculated using 2 lines. The first line starts at 0 and ends at the amplitude. The second line starts at the amplitude and ends at 0.

In a cycle, the function should reach the amplitude at half the period length. This creates the function:

$$y = \frac{A}{0.5 \bullet P} x$$

Where y is the output, A is the amplitude, P is the period, and x is the current time within the period.

For the second half of the cycle, the line should head in the opposite direction from the amplitude to 0. This means that the slope will be:

$$- \frac{A}{0.5 \bullet P}$$

To find the Y intercept, we plug in the point (0.5P,A) into the equation

$$y = -\frac{A}{0.5 \bullet P}x + b$$

Where b is the y intercept.

$$A = -\frac{A}{0.5 \bullet P}(0.5 \bullet P) + b$$
$$A = -A + b$$
$$b = 2A$$

Thus the equation is:

$$y = -\frac{A}{0.5 \bullet P}x + 2A$$

Putting the two together creates the mathematical expression

$$y = \begin{cases} y = \frac{A}{0.5 \bullet P}x & x \leq 0.5P \\ y = \frac{-A}{0.5 \bullet P}x + 2A & x > 0.5P \end{cases}$$

Sawtooth Wave

The sawtooth wave consists of a linear function that starts at 0 and ends at the amplitude for each cycle. This creates a rather buzzy effect in the sound.

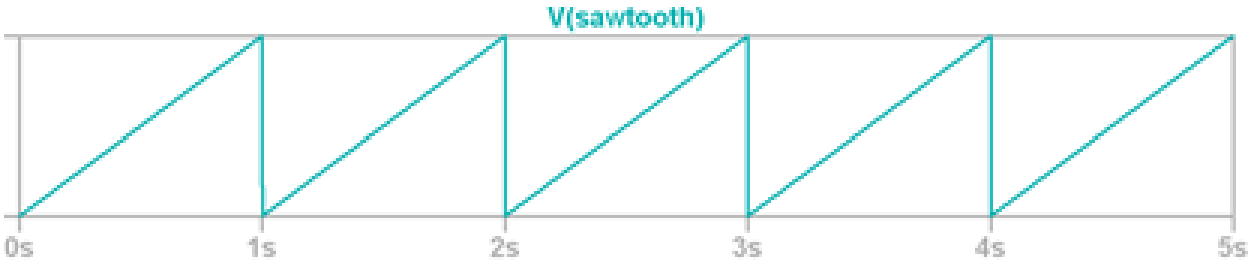


Figure 6: Sawtooth Wave (Alonso, 2016)

To recreate this wave, I simply use 1 linear function to go from 0 to the amplitude.

$$y = \frac{A}{P}x$$

Where y is the output, A is the amplitude, P is the period, and x is the point in the cycle. Since the output resets back to the amplitude every cycle, the waveform creates the vertical line between cycles.

Sine Wave

The sinewave is considered the purest waveform. It only carries one harmonic, which is that of the frequency. In order to generate the sine wave, I chose to use Arduino's sine wave function, which uses the standard GNU Math library. Another option would be to pre-generate a sine table, and use that to get the sine values for the function. However, upon performing a performance test by timing both functions in microseconds, both functions inconsistently returned either 0 or 1 microseconds. This signifies that both operations take under a microsecond, and therefore the difference is insignificant. Thus, I chose to use Arduino's sine function, as it yields more accurate results.

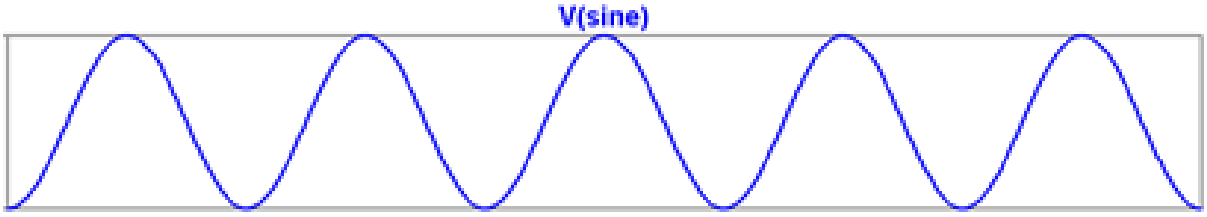


Figure 7: Sine Wave (Alonso, 2016)

The typical equation for a sine wave would be Asin(fx). However, the sin function uses radians, so to convert from degrees, I multiply the frequency by 6.28 (2pi). Additionally, A typical sine function will go both above and below 0, but the DAC does not accept negative outputs. Therefore, I perform a vertical shift of half the amplitude, and also multiply the sin by half the amplitude. This makes it so that the peak of the sine wave is at the amplitude, and the lowest point of the sine wave is 0.

Equation:

$$y = \frac{A}{2} sin(2\pi f x) + \frac{A}{2}$$

Pitch Bend and Modulation

When a single note is fluidly changed in frequency, it is called a pitch bend. The pitch bend on a keyboard can either be positive or negative, with a positive pitch bend increasing the notes frequency, and a negative pitch bend decreasing its frequency. If the pitch bend is repeatedly moved up and down, it would result in frequency modulation of the note. Frequency modulation has a carrier wave (the original sound wave of the note), and the modulating wave (the waveform determining the change in frequency at a given time). Synthesizers produce a Low frequency oscillator (LFO) to control modulation of different parameters of the sound wave. This includes pulse width, sound cutoff, and frequency. In this project, I implemented an LFO to allow for frequency modulation of the note played.

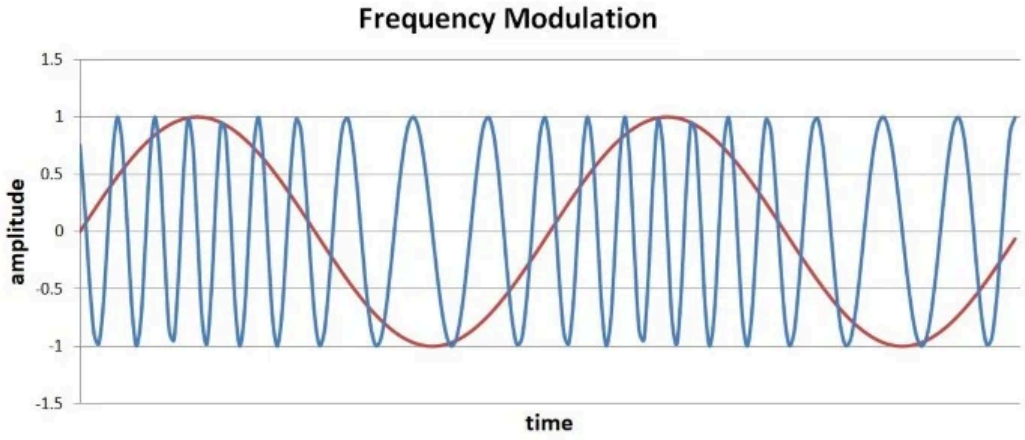


Figure 8: Frequency Modulation Graph (Frequency Modulation)

In order to affect the frequency of a sound wave, one might assume that you simply take the calculated output of the LFO, and add it to the frequency parameter of the carrier wave like this:

$$Output = A_c sin((F_c + m(x))x)$$

Where A_c is the carrier wave, m(x) is the modulator wave, and F_c is the original frequency of the carrier wave.

This was my initial approach to this problem. However, it proved to be a naive approach. The issue becomes the phase shifting of the 2 waves. When graphing 2 waves with a frequency difference of 1, their relative phases start close to each other when closer to 0, but the phase begins to diverge between the two over time.

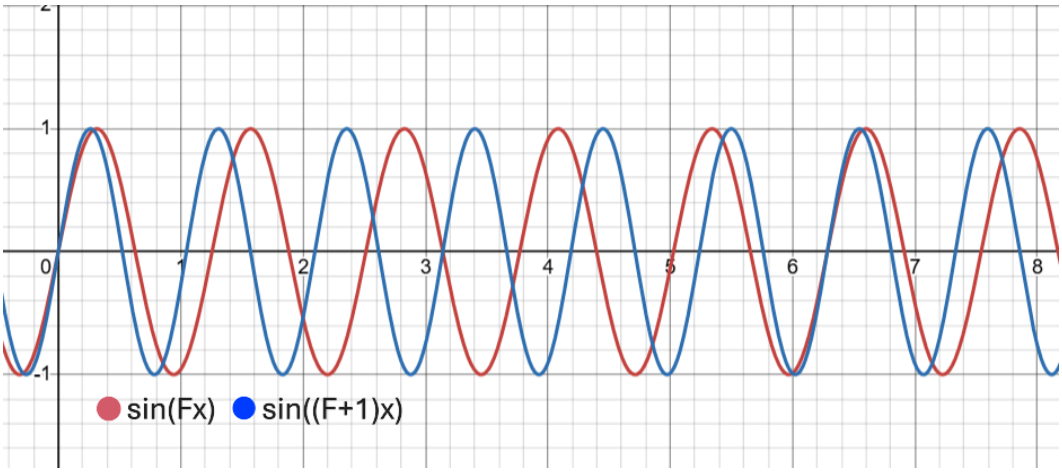


Figure 9: Amplitude vs time of 2 waves with frequency difference of 1

When working on an order of microseconds, this phase shift becomes immediately apparent.

When discreetly changing from one sound wave (F(x)), and the other wave (F(x+1)), The dramatic change in phase creates an audible “popping” sound in the output. Therefore the phase needs to be adjusted before switching to the other frequency.

An increase in frequency can be thought of as an increase in the speed of playback of a waveform. Therefore, to modify the frequency of a waveform, one can instead change the phase.

Every time I call the wave calculation function, I increase x (current time) by my sampling period, which is currently 45 microseconds. If I were to add another 45 to x every time I ran the loop, without actually waiting those extra 45 microseconds, I would simulate double the speed of the waveform, and would be able to increase the frequency instantaneously while maintaining the phase of the wave .

The difference in speed from the carrier wave to the new wave is

$$\frac{(F_c + m(x))}{F_c}$$

We would multiply this by the sampling period to get the new step size. Since the timer is already being incremented by this value each time, we only need to add the difference, which would be

$$S\frac{(F_c + m(x))}{F_c} - S = \frac{S * m(x)}{F_c}$$

Where S is the sampling period. Modifying the channel's timer could affect other parameters such as the envelope. Therefore, I implemented a new variable tracking the phase shift of the carrier wave. Every cycle we increment this phaseCounter by this equation, and calculate the sine wave as

$$A_c \, sin(F_c \, (x + \phi))$$

Where ϕ is the phase shift counter.
This logic also applies to the pitch bend, and we can add it on at the end for an extra increase or decrease in the frequency.

This is the formula I applied within my program, writing it out like so:

```
void calculateWave(Oscillator &channel){

  //calculate modulo
  int modulo = int((waves[lfo.waveform])(lfo.freq,lfo.amp,lfo.duty,channel.currentTime));
  modulo -= (lfo.amp/2); // decrease modulo to be above and below 0
  modulo+=globalVals.pitchBend;

  //get phase shift for carrier wave
  float m = float((channel.freq + modulo))/channel.freq;
  channel.shiftCounter += (m*SAMPLING_PERIOD - SAMPLING_PERIOD);
  int newX = channel.currentTime + channel.shiftCounter;

  //calculate output
  channel.output = (waves[globalVals.waveform])(channel.freq,channel.amp,globalVals.duty,newX);
}
```

Figure 10: Code for Modulo Implementation

However, this can be taken further mathematically.
Since we increment the shift counter each loop with itself (+=), it can be represented mathematically by integrating the equation.

$$\phi = \int_0^x \frac{SA_m sin(F_m t)}{F_c} dt$$

$$\phi = [\frac{-SA_m cos(F_m t)}{F_c * F_m}]_0^x dt$$

$$\phi = \frac{-SA_m cos(F_m t)}{F_c * F_m}$$

Plugging this back into the equation, we get

$$A_c sin(F_c(x + \frac{-SA_m cos(F_m x)}{F_c * F_m}))$$

$$A_c sin(F_c x - \frac{SA_m}{F_m} cos(F_m x))$$

This equation is very similar to the fundamental frequency modulation equation of sinusoidal waves, which is

$$A_c cos(F_c t + m_f sin(F_m t))$$

Which can also be written as

$$A_c sin(F_c t - m_f cos(F_m t))$$

Where mf is the modulation index defined as Δf/fm (Frequency Modulation Fundamentals, 2024).

Δf is the instantaneous change in frequency multiplied by A_m. The formula I derived reflects that, with the sampling rate S being the instantaneous change in frequency.

USB MIDI

An important aspect of creating digital music is the ability to play the sounds you create. Modern synthesizers expand the users ability to control the instruments by communicating with Musical Instrument Digital Interface [MIDI] controllers. The development of this protocol was proposed in 1981, when Dave Smith and Chet Woods published a paper published to the Audio Engineering Society about their Universal Synthesizer Interface in 1981. After some changes were made to the proposed idea in collaboration with some Japanese companies like Yamaha and Roland, the MIDI standard was finalized, and the first commercially available instrument to include MIDI, the The Prophet 600 by SCI, was shipped in 1983 (MIDI History Chapter 6-MIDI Begins, 2024).

The MIDI Protocol is a standardized communication protocol between audio devices. It works in the form of events, where a midi controller will send an event, notifying a device like computer or synthesizer, that an event has occurred. Events include a note being turned on or off, a dial being set to a specific position, and other control changes (Summary of MIDI 1.0 Messages).

Each MIDI event typically consists of 3 bytes: 1 status byte and 2 data bytes. Any unused data bytes are padded with zeros. The status byte consists of the event identifier (4 MSB) and the channel number (4 LSB). The two data bytes consist of data pertaining to the specific event (Summary of MIDI 1.0 Messages). For example: In a Note off event on channel 0, the status byte would be 10000000 (event ID 8, channel 0). For the data, data byte 1 represents the note number, and data byte 2 represents the velocity. See Appendix C for a full table of Midi Event specifications.

Although there are many different Serial Communication methods that can be used to send MIDI, for this project I have chosen to use USB, as it is fast and widely adopted by most MIDI controllers.

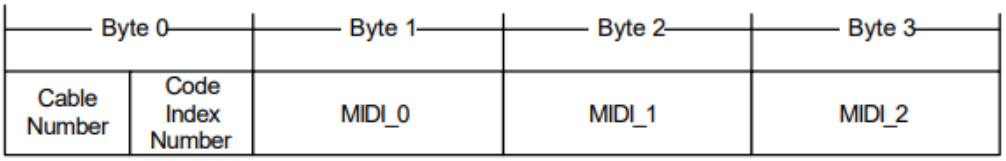


Figure 11: 32-bit USB-MIDI Event Packet Description (USB, 1999).

MIDI Events over USB are sent via 32-bit packets. The first byte holds the Cable Number (4 MSB) and the Code Index Number (4 LSB). The Code Index Number re-iterates the event type that is then specified in the first MIDI byte (USB,1999).

USB is a host driven protocol, so the ESP32 must be able to act as a USB host when communicating with the controller. The ESP32 does not have native USB capabilities, so an external USB Host processor must be added to the setup. In this project, I am using the Mini USB Host Shield 2.0. It is a board based on the MAX3421E chip by Analog Devices. It is a USB Peripheral/Host Controller with an SPI Interface (Analog Devices, 2013). The breakout board itself is poorly documented, but I was able to find the pins I needed. See Appendix A.3 for the boards pinout.

A jumper is soldered between the Vbus pin and the chip's power. The chip runs off 3.3Volts while most USB peripherals need 5V, so the jumper must be de-soldered or the trace from the bus pin to the chip must be cut.

For this project, I used the USB Host Shield Library 2.0 for taking care of communication with the MAX3421E and sending/handling USB packets.

In order to test the connection, I utilized the example program written by Yuuichi Akagawa, who was the contributor of the USB MIDI functionalities in the library. The program continuously polls the midi device for a new message, and returns the received data if there is a new midi message, printing it over serial. This is where I discovered an interesting issue. Because of the delay caused by printing to serial, the program does not poll the MIDI device as frequently, and MIDI packets would get dropped if there were too many events in a short amount of time. This means that the controller is sensitive to the polling rate, and the polling rate must remain consistently high to ensure the reception of all midi messages. I would expect a slower polling rate to cause a delayed reception of event data, but not dropping data completely.

To ensure the highest possible polling rates, I consistently poll the MIDI controller on a separate process, adding received messages to a queue shared with the main process. If the two processes were to try to access the queue at the same time, it could cause a collision. Thus, I use a mutex to manage access to the critical section, that being the queue.

Polyphony

Classical Synthesizers are either Monophonic or Polyphonic. Monophonic synthesizers can only play one note at a time. When pressing two notes at the same time on a Monophonic synthesizer, only the last note pressed will be played. Polyphonic synthesizers, on the other hand, have multiple voices that can be played at once. In analog synthesizers, the number of voices was limited to the number of hardware oscillators in the hardware (Deckert, 2024). Digital Synthesizers on the other hand, are only limited by their computational ability.

In order to manage multiple voices, I create an array of oscillators. Each oscillator begins in the INACTIVE state. When a NOTE ON event is read from the MIDI Controller, the program iterates through the array of oscillators to find the first inactive oscillator. That oscillator is then assigned to that key, by setting the oscillator key number to the key number of the turned on note. The state of the oscillator is then changed to ATTACKING, signifying that the note is in the attack stage of the envelope (will touch on this further in the Envelope section). If another note is pressed while the first note is still down, then another oscillator will be assigned to that key. When the note is released, the state of the note is changed to RELEASING, signifying that the note is in the release stage of the envelope (will touch on this further in the Envelope section). Once the note is finished releasing, the parameters of the oscillator are reset, and the state of the oscillator is set to INACTIVE, unbinding the oscillator from the key. In my

current design, if the number of notes played at once goes over the number of oscillators, the last notes get ignored.

When playing multiple notes together, an important thing to note is the combined volume of the notes. If one note is being played at an amplitude of 255, playing multiple notes at the amplitude would put the output to the dac above the 8 bit limit. Thus, the amplitude of the notes needs to be balanced. To do this, I implement a volume mixer.

In this function, I check how many notes are playing at the same time. I leave a default amplitude of ¼ the total maximum output of the dac for each note (255/4) so that up to 4 notes can be played at the same time without modification to the overall volume.

If more than 4 notes are played at the same time, the volume of the notes is scaled, so that the amplitude of all the notes combined add to 255. For example if a fifth note is turned on, the amplitude of the 4 notes already being played will be scaled down from 255/4 to 255/5.

Envelope

The Audio envelope allows for the shaping of the amplitude of the wave when the note is pressed and released. The volume can quickly increase and decrease, making a short and sharp sound. Or, the volume could slowly increase and decrease, creating a much smoother and mellow note. The Attack Decay Sustain Release (ADSR) Envelope generator was first introduced in the Moog Synthesizer, one of the first Voltage controlled modular synthesizers, released in 1964 (Pinch, 2004). This envelope architecture is still widely used to this day for musical synthesis. The ADSR envelope controls 4 parameters to shape the amplitude of the output:

Attack: Length of time it takes for the wave to reach its maximum amplitude after being pressed
Decay: Amount of time it takes for the wave to come back down from the maximum amplitude to the sustain amplitude
Sustain: Amplitude at which to hold the note
Release: Length of time it takes for the amplitude of the wave to decrease to 0 after the note is released.

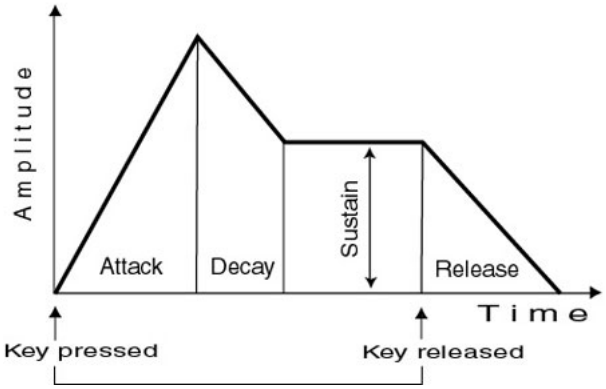


Figure 12: Audio Envelope Graph (“Quick Guide to Envelopes”, 2022)

In order to incorporate these parameters into my synthesizer, I defined states within the Oscillator structure to keep track of the current state of the note.

ATTACKING: State of note when key is pressed
DECAYING: State of note when attacking is finished
RELEASING: State of note when key is released
INACTIVE: State of note when releasing is finished

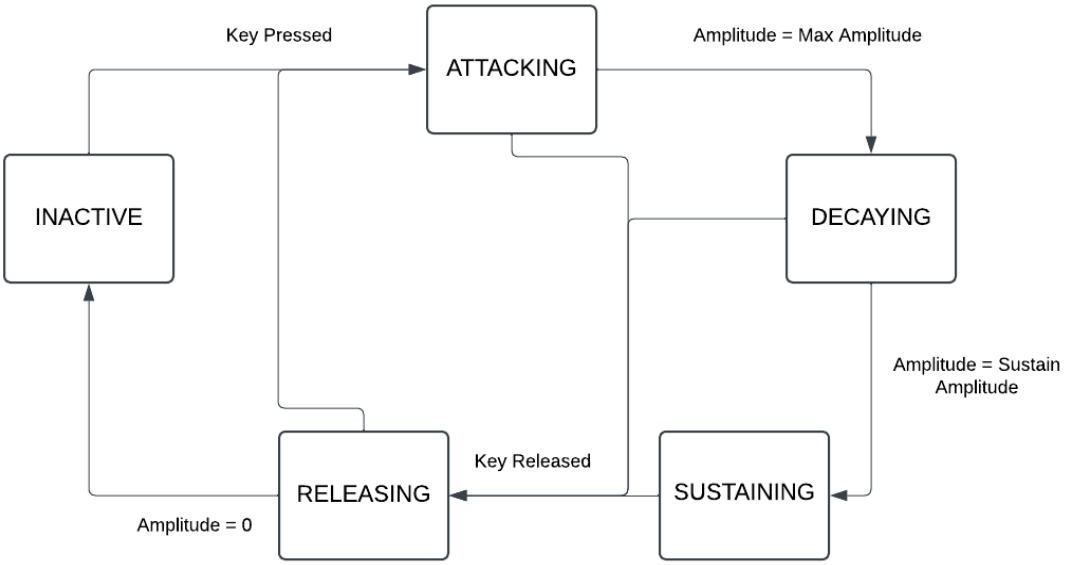


Figure 13: Envelope State Machine

In this project, I use a linear function for the change in amplitude over time, but other functions, such as quadratic,are also used in music production.

The increase to the amplitude during attacking is

$$\frac{A_{max} * S}{\alpha * 1000}$$

Where A_{max} is the maximum amplitude of the wave
S is the Sampling Rate in microseconds,
α is the Attack time in milliseconds.

Similarly, the decrease to the amplitude while decaying is

$$\frac{A_{max}{}^{*}S}{\beta*1000}$$

Where β is the Decay time in milliseconds.

For the Release however, the key may be released before the note has finished decaying. In order to keep the length of time spent releasing consistent to the release value, I use an extra variable `relTime`, that records the time the note was released. The equation is then:

$$\frac{A_{rel}{}^{*}S}{\rho*1000}$$

Where ρ is the Release length in milliseconds, and A_{rel} is the recorded time at which the note was released.

One of these 3 functions will increment or decrement the Amplitude of the output wave every loop. To avoid redundancy, the oscillator will stay in the decaying state while sustaining, and the function will only decrement the amplitude if it is above the sustain value.

Results

The result is a fully functional 8 bit digital synthesizer. It generates the 4 basic waveforms, has polyphonic functionality, uses an envelope generator, and can communicate over MIDI. Additionally, the total price comes out to be $49.80, which is just under the price limit set out for this project.

Figure 14: Image of finished project

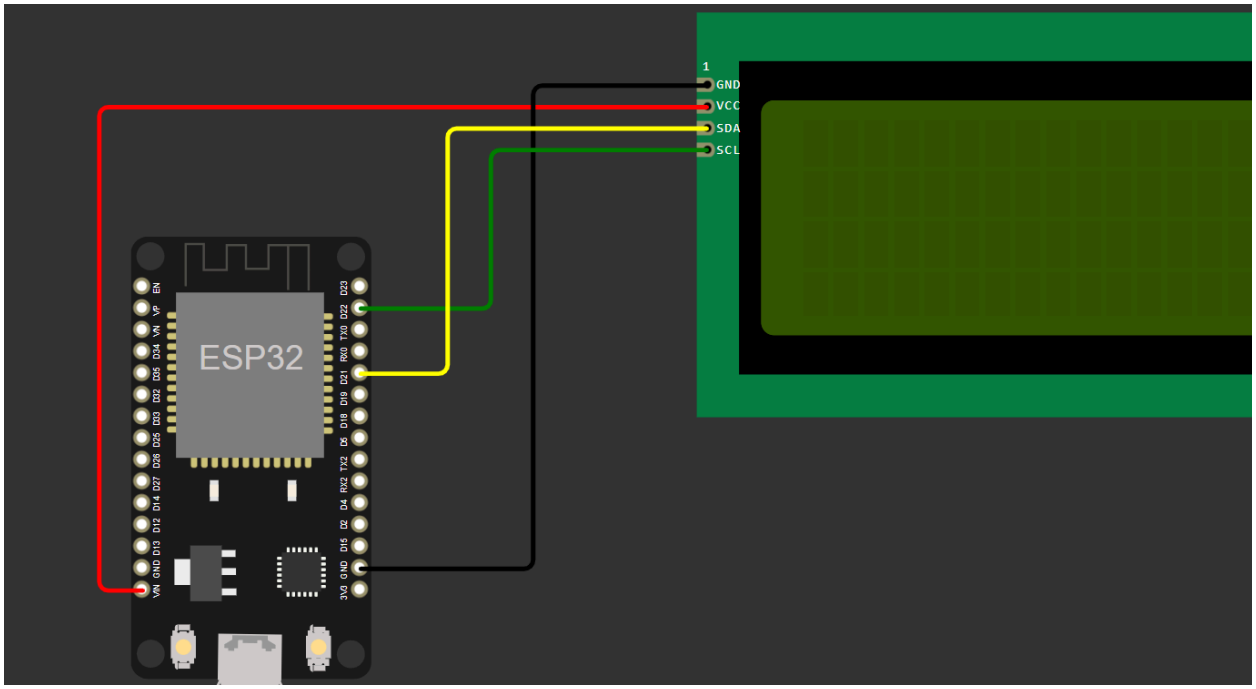
Future Work

If given more time, there are multiple features I would like to include or further develop, such as effects like filters and echo, the addition of an external DAC for higher bit depth, and an SD card module for storing preset parameters. An audio out port would also be a great addition, as it would allow to record the sounds and play it through a higher quality speaker. Additionally, there are minor bugs that could be ironed out with additional time. For example, the USB handler will occasionally drop a NOTE_OFF message, making the note stay on after release. It is unclear why this happens, and needs more experimentation. An Oscilloscope with USB interpretation capabilities may be helpful for diagnosing the problem.

Appendices

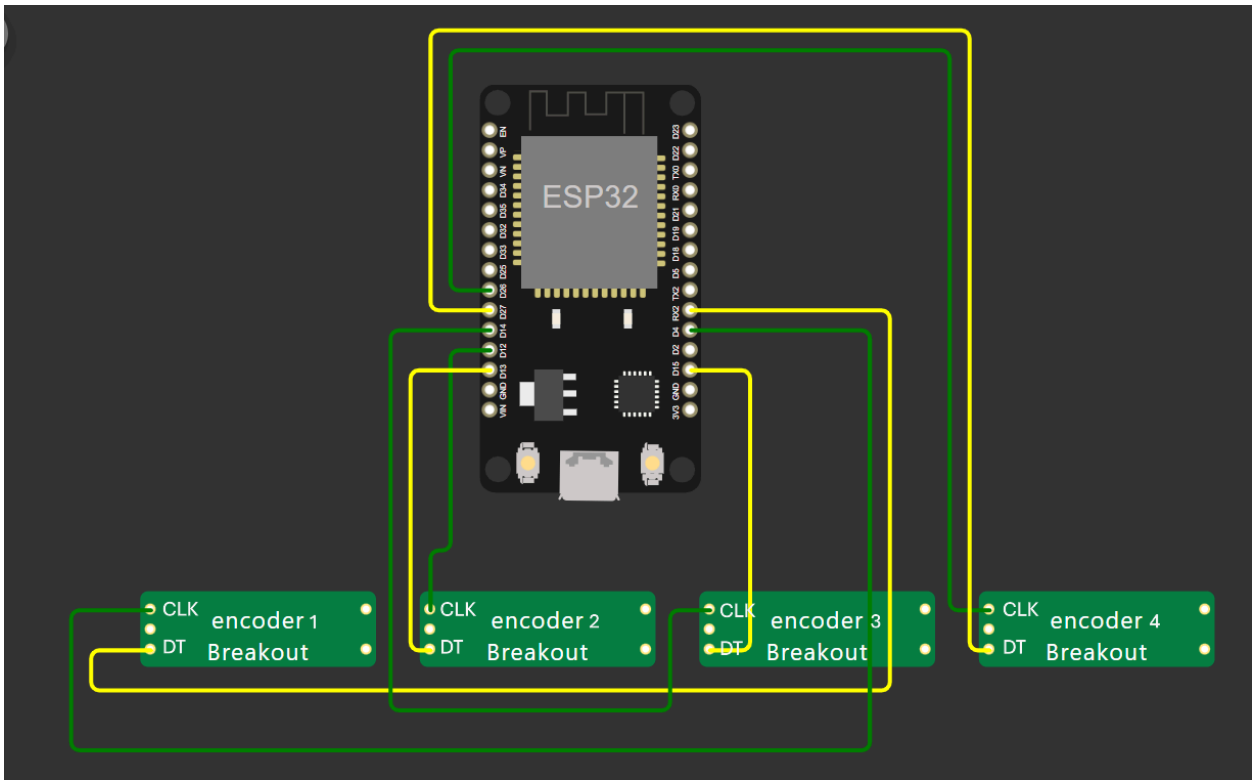
Appendix A: Detailed Wiring Diagrams

A.1 LCD Wiring



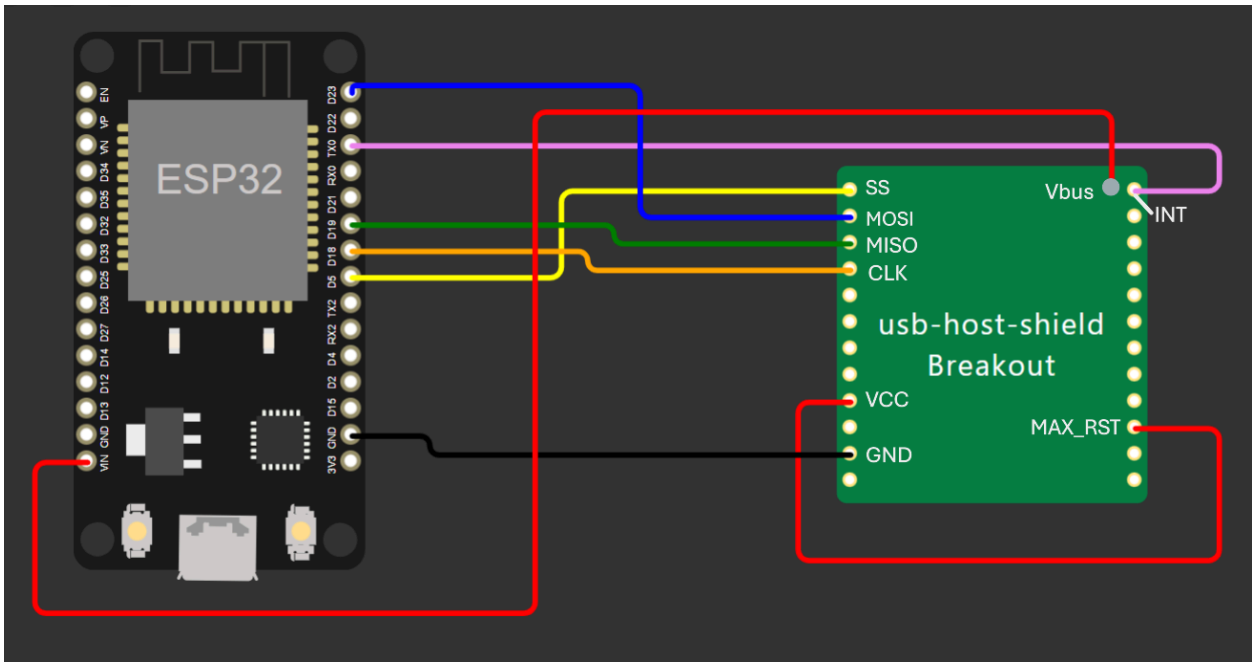
	LCD VCC	LCD GND	LCD SCL	LCD SDA
ESP Pin	Vin/5V	GND	GPIO22	GPIO21

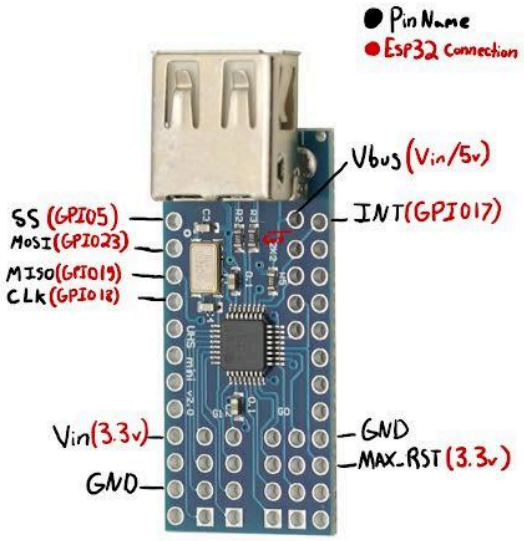
A.2 Encoder Wiring



	Encoder 1	Encoder 2	Encoder 3	Encoder 4
DT pin	ESP GPIO4	ESP GPIO13	ESP GPIO15	ESP GPIO27
CLK pin	ESP GPIO2	ESP GPIO12	ESP GPIO14	ESP GPIO26

A.3 USB Host Shield 2.0 Wiring

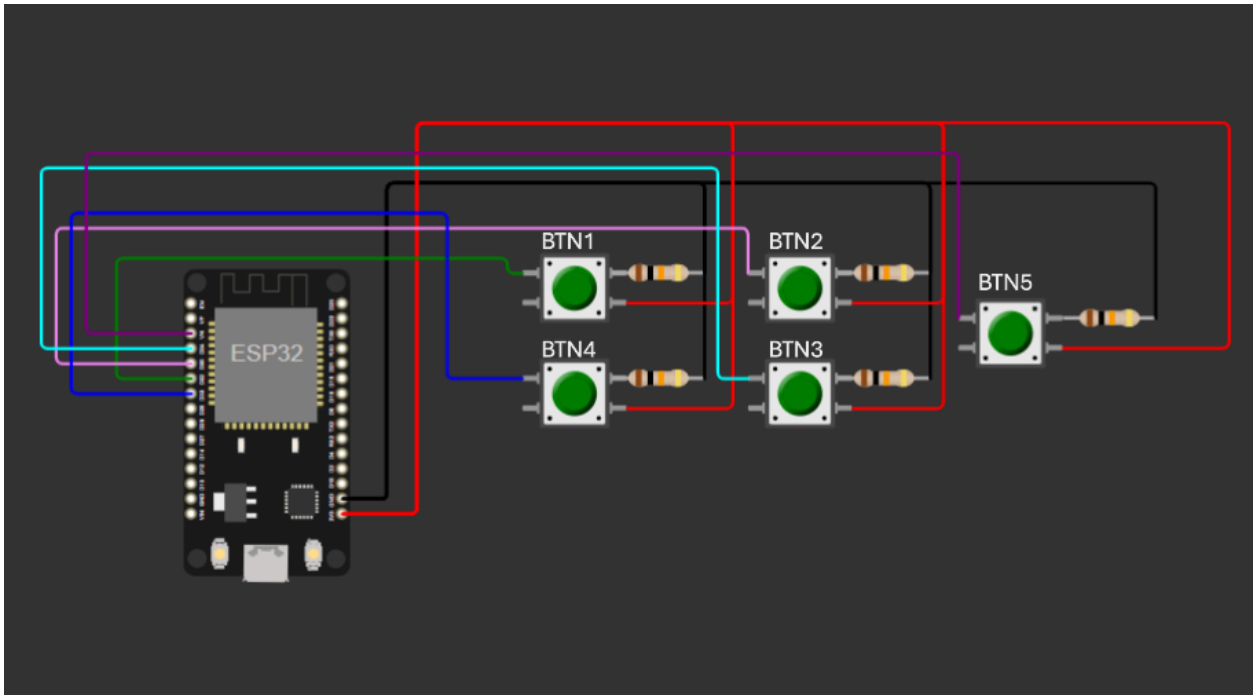




USB Host Pins	ESP Pins
VCC	3.3V
INT	GPIO17
SS	GPIO5

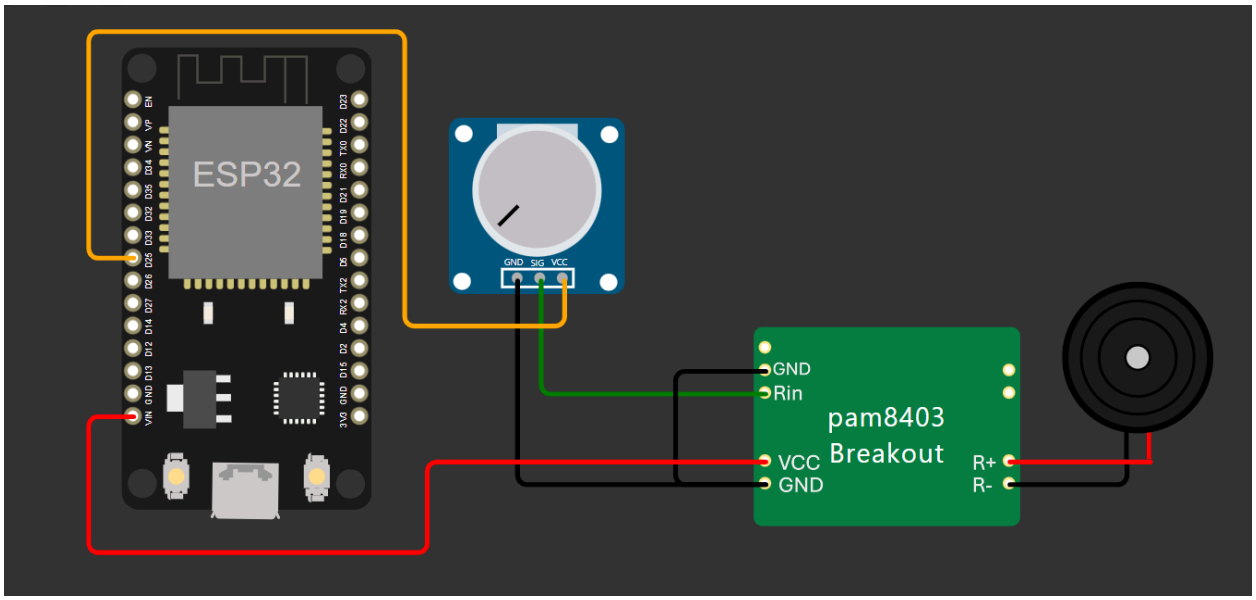
MOSI	GPIO23
MISO	GPIO19
CLK	GPIO18
MAX_RST	3.3V
GND	GND
VBus	Vin/5v

A.4 Button Wiring



	Button1	Button2	Button3	Button4	Button5	VCC	GND
ESP Pin	GPIO32	GPIO35	GPIO34	GPIO33	GPIO39	3.3v	10KΩ res to GND

A.5 Speaker Wiring



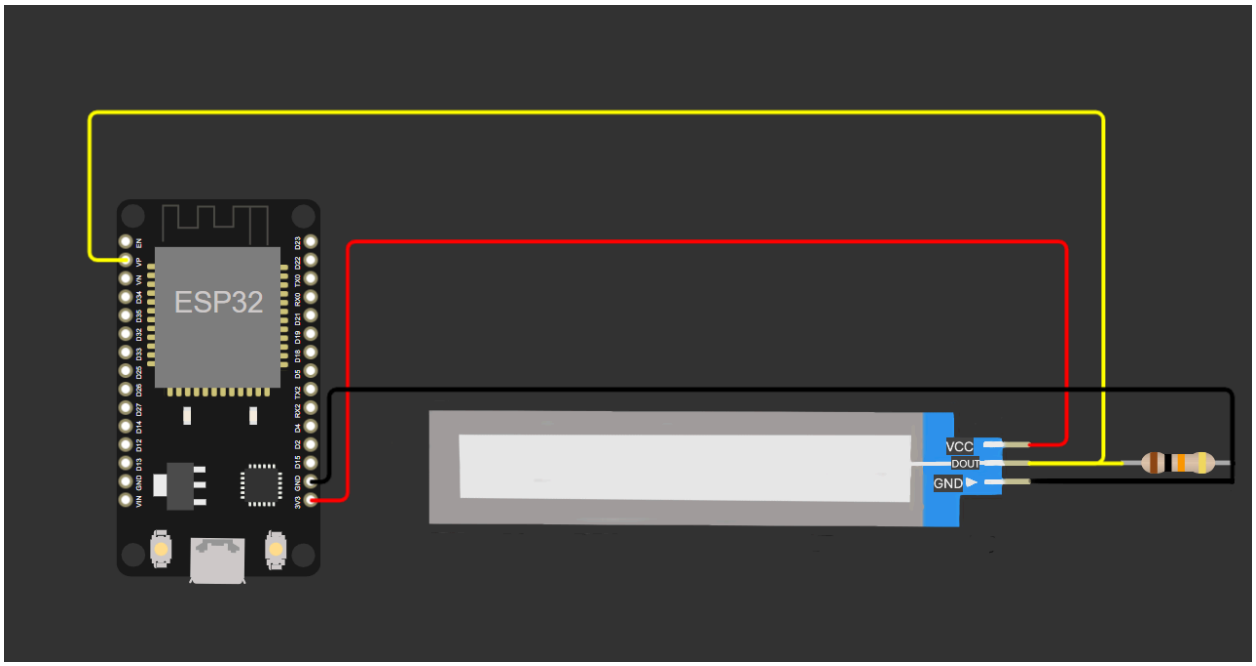
Volume Potentiometer

Volume Pot VCC	Volume Pot OUT	Volume Pot GND
ESP GPIO 25	PAM8403 Rin	ESP GND

PAM8403 Amplifier

Rin	VCC	GND	R+	R-	⊥
Volume pot OUT	ESP Vin/5v	ESP GND	Speaker+	Speaker-	GND

A.6 Softpot Membrane Potentiometer Wiring



VCC	GND	DOUT	
ESP 3.3V	ESP GND	ESP GPIO36	10KΩ res to GND

Appendix B: Bill Of Materials

Part	Manufacturer	Specifications	Part#	Qty.	Vendor	Cost(ind.)	Cost(total)
ESP32 devkit-V1	AiTrip		15363	1	Amazon	$5.00	$5.00
I2C LCD Screen	huyouming	20x4 LCD Working voltage: 5V ST7066U Controller Communication:I2C	B0D2LDJ3JY(ASIN)	1	Amazon	$9.99	$9.99
Potentiometer	Taiss	10kohm	B0B3126K2M	1	Amazon	10 for $8.66, 1 for $0.86	$0.86
Speaker	Dweii	Impedance: 8 ohm Output: 3w	701715519671	1	Amazon	4 for $9.99, 1 for $2.49	$2.49
Rotary Encoder	WWZMDiB		EC11	4	Amazon	6 for $8.88, 1 for $1.48	$5.92
Softpot Potentiometer	Spectra Symbol	150mm Linearity: +-3%		1	spectrasymbol.com	$14.58	$14.58
PAM8403 Amplifier	EPLZON	Max Voltage: 5.5V Min Voltage: 2.5V 2 channel 3w output		1	Amazon	10 for $9.99, 1 for $0.99	$0.99
Mini USB Host Shield 2.0		Working Voltage: 3.3V		1	AliExpress	$4.79	$4.79
10k ohm Resistors	TE Connectivity Passive Product		CFR100J10K	6	DigiKey	$0.28	$1.68
Push Button 7mm	uxcell		a11111400ux0132	5	Amazon	10 for $7.01, 1 for $0.70	$3.50

Appendix C: Repository

Repository: https://github.com/TheSeaUrchin/Pixis-Slide

References

Alonso, Gabino. “LTSPICE: Generating Triangular & Sawtooth Waveforms.” *LTspice: Generating Triangular & Sawtooth Waveforms | Analog Devices*, Analog Devices, Mar. 2016.

Analog Devices, ‘MAX3421E: USB Peripheral/Host Controller with SPI Interface Data Sheet (Rev.4)’, Jul. 2013.

Deckert, C. (2024). Good Vibrations from Electronic Circuits: The Technological Development of the Synthesizer.

“Frequency Modulation Fundamentals - Mini-Circuits Blog.” *Mini*, 8 Mar. 2024.

“Frequency Modulation: Theory, Time Domain, Frequency Domain: Radio Frequency Modulation: Electronics Textbook.” *All About Circuits*.

Hass, Jeffery. “*Synthesis Chapter Four: Waveforms*” *Introduction to Computer Music,* Indiana University

“MIDI History Chapter 6-MIDI Begins 1981-1983.” *MIDI.Org*, MIDI Association, Nov. 2024.

Pinch, T., & Trocco, F. (2004). Analog days: The invention and impact of the Moog synthesizer. Harvard University Press.

“Quick Guide to Envelopes.” *Making Music*, 3 Apr. 2022.

“Summary of MIDI 1.0 Messages.” *MIDI.Org*, MIDI Association, Feb. 2024.

USB, ‘Universal Serial Bus Device Class Definition for MIDI Devices (Release 1.0)’, Nov. 1999.