# Table of Contents

## Overview
- Parts

## Synthesizer Fundamentals
- Audio Oscillator, a.k.a VCO
- Wave Shapes/Wavetables
- Filter, a.k.a. LPF, HPF, BP, notch
- Amplifier/Attenuator, a.k.a. VCA
- Envelope Generator, a.k.a. EG
- Subtractive Synth Signal Flow
- Low Frequency Oscillator, a.k.a., LFO
- Ring Modulation
- Multiple Voices / Polyphony

## CircuitPython
- CircuitPython Quickstart
- Safe Mode
- Flash Resetting UF2

## synthio Getting Started
- synthio Specific Nomenclature

## Simple synthio Examples
- Simple Note
- Simple Chord
- Simple Chord with Mixer
- Amplitude Envelope
- Filters
- MIDI Note Input
- MIDI CC Input
- LFO Pitch Bend
- LFO Tremolo
- LFO Tremolo

## Advanced synthio Examples
- Waveshapes
- Waveshape Morphing
- Detuning Oscillators for Fatter Sound
- Wavetables
- Larger Examples
- Tyrell Desktop Synth
- Monosynth
- Wavetable Polysynth
- Mini Arpeggiator
Overview

You can create your own polyphonic audio synthesizer on a microcontroller board using CircuitPython. The `synthio` module makes it sound great and it's easy to use.

This guide covers the fundamentals of audio synthesis using `synthio`, including practical examples. You'll be building your own small-but-powerful synths in no time.
synthio History

The `synthio` library was originally created as single-voice, monophonic square wave player for MIDI files.

As of CircuitPython 8.1, `synthio` received significant development effort from Jeff Epler () to turn it into a 12-voice, polyphonic synthesizer that uses a wide variety of waveforms, while adding filters, envelopes, modulation control, and a ring-modulation effect.

Check out the full `synthio` documentation here ()

Up to 12 simultaneous voices are supported, though the exact limits depend on the sample rate, and the complexity of the effects applied.

Hardware Requirements

While `synthio` is built into CircuitPython, this doesn't mean that every CircuitPython-capable board can run synthio effectively. Some chips to use for the best results are:

- RP2040 -- e.g., Raspberry Pi Pico, QT Py RP2040
- EPS32-S2
- EPS32-S3
- iMX RT1011 -- e.g., Metro M7
- SAMD51 "M4" (N.B., a bit hard to find these days)
- nRF52 -- e.g., Feather nRF52840, CLUE

Your chosen board will also need to support one of the three audio output methods:

- `audioio.AudioOut` -- output to a built-in Digital to Analog Converter (DAC)
- `audiobusio.I2SOut` -- output to I2S DAC amplifier board
- `audiopwmio.PWMAudioOut` -- output of pulse-width modulated square waves that will go through an external resistor-capacitor (RC) circuit to convert to audio

We'll cover these in more depth later in the guide.
Board Check
You can check to see which modules are supported on any given board by checking the CircuitPython.org download page for that board.

In this example, the Feather RP2040 has synthio listed, as well as two different audio output modules that will work: audiobusio and audiopwmio.

Parts
You can run synthio on a wide variety of boards, as noted above.

The simplest hardware to get up and running quickly is the PropMaker Feather RP2040, thanks to its on-board I2S amplifier and terminal blocks to screw in your speaker wires.

Adafruit RP2040 Prop-Maker Feather with I2S Audio Amplifier
The Adafruit Feather series gives you lots of options for a small, portable, rechargeable microcontroller board. By picking a feather and stacking on a FeatherWing you can create...
https://www.adafruit.com/product/5768

Mono Enclosed Speaker with Plain Wires - 3W 4 Ohm
Listen up! This single 2.8" x 1.2" speaker is the perfect addition to any audio project where you need 4 ohm impedance and 3W or less of power. We...
https://www.adafruit.com/product/4445
For the example videos in this guide we used the parts listed below. Note: any QT Py ESP32-S2, QT Py ESP32-S3, or QT Py RP2040 will work the same for the examples in this guide.

**Adafruit QT Py ESP32-S2 WiFi Dev Board with STEMMA QT**
What has your favorite Espressif WiFi microcontroller, comes with our favorite connector - the STEMMA QT, a chainable I2C port, and has...
https://www.adafruit.com/product/5325

**Adafruit Audio BFF Add-on for QT Py and Xiao**
Our QT Py boards are a great way to make very small microcontroller projects that pack a ton of power - and now we have a way for you to turn many QT Py boards into powerful audio play...
https://www.adafruit.com/product/5769

or

**Adafruit I2S Amplifier BFF Add-On for QT Py and Xiao**
Our QT Py boards are a great way to make very small microcontroller projects that pack a ton of power - and now we have a way for you to add an I2S 3 Watt amplifier, for high quality...
https://www.adafruit.com/product/5770
Molex PicoBlade 2-pin Cable - 200mm
When 0.1" is too big, and JST PH's too chunky, the ultra-slim "PicoBlade" is a reliable alternative. These are only 1.25mm pitch, but have a nice clicky...
https://www.adafruit.com/product/3922

Also:

Mini Oval Speaker - 8 Ohm 1 Watt
Hear the good news! This wee speaker is a great addition to any audio project where you need 8 ohm impedance and 1W or less of power. We particularly like...
https://www.adafruit.com/product/3923

Short Male Header Kit for ItsyBitsy
These three Short Male Headers alone are, well, lonely. But pair them with our
https://www.adafruit.com/product/4173
Short Female Header Kit for ItsyBitsy
These three Short Female Headers alone are, well, lonely. But pair them with any of our
https://www.adafruit.com/product/4174

Synthesizer Fundamentals

Time to review some fundamental synthesis concepts and how they relate to synthio.

There are many different ways to approach audio synthesis -- additive synthesis, subtractive synthesis, frequency modulation, phase distortion, waveshaping, physical modeling, wavetables, and more.

We'll look at the building blocks of how subtractive synthesis works, since it is the most similar to the synthio approach, as well as wavetable synthesis, as this is another feature of synthio.

Note: synthio is not a fully modular synthesizer. It's signal flow is fixed, with each note containing the main subtractive synth building block wired in a specific way we'll discuss later. There are, however, some objects that can be wired creative into a synthio Note, so it can be considered a semi-modular synth.
Audio Oscillator, a.k.a VCO
Arguably the most important piece of the subtractive synthesis puzzle -- the audio rate oscillator. This is a wave that oscillates at an audible rate -- anywhere from ~20Hz to 20,000Hz. Ultimately, this electrical signal is used to vibrate a speaker cone which disturbs the air, creating sound waves that can reach your ear.

The rate of the oscillation determines the pitch we hear. In many synthesizers the rate/pitch is controlled via voltage signals, hence the name Voltage Controlled Oscillator, or VCO. These pitch CV signals typically come from a keyboard or step sequencer.

There is a secondary pitch modification control on most oscillators called pitch bend which can nudge the key that’s currently being played up or down. Many keyboards have a dedicated pitch bend wheel used for this purpose.
Wave Shapes/Wavetables
The shape of the oscillator's wave determines the timbre or "character" of the sound, based on the different harmonic content of the sound above and beyond the fundamental frequency.

Common wave shapes include sawtooth, triangle, square, and sine.

Some oscillators (including synthio) have an extra trick up their sleeve -- instead of being confined to a single wave shape, they can use buffered single-cycle waveforms of any shape imaginable, and can morph between two or more of these shapes! This is called wavetable synthesis, and it gives huge flexibility in sound design.

Patch Symbols from PATCH & TWEAK by Kim Bjørn and Chris Meyer, published by Bjooks, are licensed under CC BY-ND 4.0. The included symbols are available free of use under the following terms and license.

Control Voltage, or CV, is the name of the type of signal sent from one synth module to the next that can be used to adjust attributes such as pitch, attenuation, and modulation. Typical CV levels on a Eurorack synthesizer system accept smoothly variable -5V to +5V, where 1V is equal to one octave of pitch.
Filter, a.k.a. LPF, HPF, BP, notch

Most oscillator waveforms include a lot of rich harmonics that give them their distinct characters, such as the "buzziness" of a sawtooth wave. A filter can be used to cut off some of those harmonics above or below a certain frequency.

Low-pass filters can be set to reduce or remove sound above a certain cutoff frequency you chose, which can give your audio a sort of muffled sound.

High-pass filters can be set to reduce or remove sound below a certain cutoff frequency, which can give your sound a sort of bright or "tinny" character.

Band-pass filters reduce or remove sound both above and below their cutoff frequency.

Notch filters work like an inverted band-pass filter, by reducing or removing sound within a min-max cutoff frequency.

Many filters also include a control called resonance (or 'Q'), which allows you to emphasize the harmonics right at the cutoff frequency. This is the little hump seen in the first two diagrams shown here. Strong resonance loops can start to feedback on themselves and even self-oscillate, which can be an interesting sounding effect.

The particular cutoff frequency of a filter is often modified with incoming control voltage, sent from another modulation source such as an LFO (more on this below).
Amplifier/Attenuator, a.k.a. VCA

An oscillator that drones on forever could get pretty tiresome. In order to play sounds for fixed durations we need a way to turn the volume up and down!

An amplifier (a misnomer we live with in the synth world, because it is technically a loudness attenuator) is used to adjust the volume of the audio oscillator. It will generally be "off" until a CV signal tells it to open up to partial or full volume, allowing the sound to pass through.

Again, this signal that opens and closes the amplifier varying amounts is a control voltage (often coming from an envelope generator as discussed below), hence the name Voltage Controlled Amplifier, or VCA.

synthio Filter

Note has a biquad filter (Note.filter) that can be set as LPF, HPF, and BPF.

synthio VCA

Note has an implicit loudness attenuator built into it tied to the envelope generator (see below).
Envelope Generator, a.k.a. EG

Envelopes are used to "shape" the loudness of a sound over time, so that we don't live with simple ON-OFF attenuation of the volume.

The envelope generator, or EG, is what tells the VCA how loud or quiet to get during the different stages of a note's lifetime.

A typical set of stages on an EG are:

- **Attack**: how long it takes from the onset of a note until it reaches its full volume
- **Decay**: how long it takes for the volume to drop from full attack level to the sustain level
- **Sustain**: the level to stay at while a note is held
- **Release**: how long to take to drop from the sustain level to fully "off" after the note is told to stop playing

For brevity, this type of parameter set is usually referred to as an ADSR envelope generator.

The EG sends control voltage (CV) to the VCA that varies as the EG moves through its stages, thus attenuating the loudness of the VCO signal traveling through the VCA.

But what tells the EG when to start and stop? This is a simple ON-OFF signal called a Gate. A typical scenario is a key being pressed on a keyboard sends a high Gate signal to the EG, and when the key is released on the keyboard a low Gate signal is sent to the EG.
Subtractive Synth Signal Flow

Using the above building blocks here is the signal flow for a typical subtractive synth:

- You press a key on a keyboard, which sends 1V/Oct pitch CV to the sawtooth oscillator, and a Gate signal to the ADSR envelope generator.
- Oscillator sends audio to the filter, where certain harmonic above a frequency are reduced.
- Filter sends the audio to the amplifier. The envelope generator tells the amplifier to attenuate the audio loudness according to its ADSR stages/settings.
- The audio then heads off to an audio output device, e.g., speakers/headphones/amp, etc. (optionally, it first passes through a mixer where it can be combined with other audio.)

Gate signals are very similar to CV, but unlike the wide variation of CV levels possible, Gate is binary -- high/low a.k.a. on/off are all the Gate can do. Eurorack modules typically send 0V as low and anywhere from +5v to +8V as high.

Here’s how it works:

Would you like to add or modify any text in this document? Please let me know if you need any clarification or assistance.
Low Frequency Oscillator, a.k.a., LFO
At any step of the synth signal chain where a knob can be twiddled by hand to modulate the signal, a Low Frequency Oscillator, or LFO, can automate the twiddling.

LFOs are very similar to audio oscillators, they just have much slower rates, running anywhere from just below audible rates to many seconds or even minutes to complete a single cycle.

A typical use for an LFO is to send CV signal to the cutoff frequency input of a filter so that it varies over time.

Ring Modulation
Ring modulation is a method of multiplying the oscillator frequency (called the "carrier") by a second frequency (called the "modulator"), and then passing along only the sideband frequencies.

Ring mods can produce special effects (Dalek voice from Dr. Who is a classic example) and metallic sounds, since the harmonics produced tend to not be very musical sounding ones.
Multiple Voices / Polyphony

The examples above represent a single synthesizer voice, sometimes referred to as a monophonic synth. These are great for basslines and sometimes leads, but what about when you want to play a chord?

synthio Ring Mod

synthio.Note contains ring frequency, ring bend, and ring waveform parameters for creating ring mod effects.
The good news is, you simply add more voices! synthio is a truly polyphonic synthesizer, meaning it can play multiple full voices/notes at once. Each note has a full signal chain, so you can adjust all of the parameters (wave shape, pitch, filter, envelope) individually per voice.

These voices/notes can be sent through the audio mixer to combine them on their way to the audio output.

**Single Voice**
Here is a graphical example of a single voice/note.

**Two Voices**
This is an example of two voices/notes, running through an audio mixer on their way to the audio output.

**LFOs Added**
We've added a couple of LFOs for modulation of filter cutoff on one voice and pitch bend on another.
Big Polyphony!
Here's an example with eight voices. `synthio` is capable of 12 (or possibly more) note polyphony!

CircuitPython

[CircuitPython](https://circuitpython.org) is a derivative of [MicroPython](https://micropython.org) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files on the CIRCUITPY drive to iterate.

**CircuitPython Quickstart**

Follow this step-by-step to quickly get CircuitPython running on your board.

**Download the latest version of CircuitPython for this board via circuitpython.org**

Click the link above to download the latest CircuitPython UF2 file.

Save it wherever is convenient for you.
To enter the bootloader, hold down the BOOT/BOOTSEL button (highlighted in red above), and while continuing to hold it (don't let go!), press and release the reset button (highlighted in blue above). Continue to hold the BOOT/BOOTSEL button until the RPI-RP2 drive appears!

If the drive does not appear, release all the buttons, and then repeat the process above.

You can also start with your board unplugged from USB, press and hold the BOOTSEL button (highlighted in red above), continue to hold it while plugging it into USB, and wait for the drive to appear before releasing the button.

A lot of people end up using charge-only USB cables and it is very frustrating! Make sure you have a USB cable you know is good for data sync.
You will see a new disk drive appear called RPI-RP2.

Drag the adafruit_circuitpython_etc.uf2 file to RPI-RP2.

The RPI-RP2 drive will disappear and a new disk drive called CIRCUITPY will appear.

That's it, you're done! :)

Safe Mode

You want to edit your code.py or modify the files on your CIRCUITPY drive, but find that you can't. Perhaps your board has gotten into a state where CIRCUITPY is read-only. You may have turned off the CIRCUITPY drive altogether. Whatever the reason, safe mode can help.
Safe mode in CircuitPython does not run any user code on startup, and disables auto-reload. This means a few things. First, safe mode bypasses any code in boot.py (where you can set CIRCUITPY read-only or turn it off completely). Second, it does not run the code in code.py. And finally, it does not automatically soft-reload when data is written to the CIRCUITPY drive.

Therefore, whatever you may have done to put your board in a non-interactive state, safe mode gives you the opportunity to correct it without losing all of the data on the CIRCUITPY drive.

Entering Safe Mode

To enter safe mode when using CircuitPython, plug in your board or hit reset (highlighted in red above). Immediately after the board starts up or resets, it waits 1000ms. On some boards, the onboard status LED (highlighted in green above) will blink yellow during that time. If you press reset during that 1000ms, the board will start up in safe mode. It can be difficult to react to the yellow LED, so you may want to think of it simply as a slow double click of the reset button. (Remember, a fast double click of reset enters the bootloader.)

In Safe Mode

If you successfully enter safe mode on CircuitPython, the LED will intermittently blink yellow three times.

If you connect to the serial console, you'll find the following message.

```
Auto-reload is off.
Running in safe mode! Not running saved code.
CircuitPython is in safe mode because you pressed the reset button during boot. Press again to exit safe mode.
Press any key to enter the REPL. Use CTRL-D to reload.
```

You can now edit the contents of the CIRCUITPY drive. Remember, your code will not run until you press the reset button, or unplug and plug in your board, to get out of safe mode.

Flash Resetting UF2

If your board ever gets into a really weird state and doesn't even show up as a disk drive when installing CircuitPython, try loading this 'nuke' UF2 which will do a 'deep
clean' on your Flash Memory. You will lose all the files on the board, but at least you'll be able to revive it! After loading this UF2, follow the steps above to re-install CircuitPython.

Download flash erasing "nuke" UF2

synthio Getting Started

You can choose from a number of different boards when using synthio. The simplest setup is to use the PropMaker Feather RP2040 with a speaker's negative and positive wires screwed into the speaker terminal blocks as shown here.

Alternately, you can use a QT Py ESP32-S2 with either the I2S Amplifier BFF or the Audio BFF (which has the same amplifier but also adds an SD card reader).

Solder header pins and sockets onto the QT Py and BFF to connect them belly-to-belly as shown, then plug in your speaker (no need for an SD card).
synthio Specific Nomenclature

Envelope

The Envelope controls how a note's amplitude changes over time. This is divided into 4 sections and controlled by 5 numbers:

- In the first phase, the attack, the note's amplitude rises to the \textit{attack\_amplitude} over \textit{attack\_time} seconds.
- In the second phase, the decay, the note's amplitude falls by a factor of \textit{sustain\_amplitude} over \textit{decay\_time} seconds.
- This amplitude is maintained until the note is released, unless the \textit{sustain\_amplitude} is zero, in which case the note is released automatically when the amplitude reaches 0.
- When the note is released, it falls to zero amplitude at a rate given by \textit{release\_time}.

Different values evoke different kinds of instruments. For instance, a plucked instrument has a very small \textit{attack\_time} and fades away in a defined length of time, meaning it has a \textit{sustain\_amplitude} of zero.

```
envelope = synthio.Envelope(attack_time=0.1, decay_time = 0.1, release_time=0.1, attack_level=1, sustain_level=0.05)
```
**Waveform**

A "waveform" is an abstract kind of object. It can be an `array.array('h', ...)` or a `np.array(..., dtype=np.int16)`, a `memoryview` or any other in memory buffer that contains signed 16-bit values.

This waveform is usually a small file, with just a few hundred samples, because it represents just a single "cycle" of the sound.

**Waveforms are used by** [Notes](#) **and by LFOs.**

```python
import synthio
import ulab.numpy as np

wave_sine = np.array(np.sin(np.linspace(0, 2*np.pi, SAMPLE_SIZE, endpoint=False)) * SAMPLE_VOLUME, dtype=np.int16)
wave_saw = np.linspace(SAMPLE_VOLUME, -SAMPLE_VOLUME, num=SAMPLE_SIZE, dtype=np.int16)
wave_tri = np.concatenate((np.linspace(-SAMPLE_VOLUME, SAMPLE_VOLUME, num=half_period, dtype=np.int16),
                           np.linspace(SAMPLE_VOLUME, -SAMPLE_VOLUME, num=half_period, dtype=np.int16)))
wave_square = np.concatenate((np.full(half_period, SAMPLE_VOLUME, dtype=np.int16),
                              np.full(half_period, -SAMPLE_VOLUME, dtype=np.int16)))
```

**Note**

Each individual sound is a [Note](#). A note has a bunch of properties:

- **waveform**: a small memory buffer that sets the overall tone of an instrument.
- **envelope**: an object to define how the overall note amplitude changes over time
- **frequency**: the basic frequency of the note in Hz (which can be modified by **bend**)
- **bend**: a modification to the basic frequency of the note, in units of 1/octave. This can be a number, or a [Block](#) in case the bend value should vary over time (e.g., a vibrato effect)
- **amplitude**: a modification to the note amplitude. This can be a number, or a [Block](#) in case the amplitude value should vary over time (e.g., a tremolo effect)
- **ring_waveform**, **ring_frequency**, **ring_bend**: A second, optional waveform that modulates the main waveform to create richer sounds

```python
note1 = synthio.Note(frequency, panning, waveform, envelope, amplitude, bend, filter, ring_frequency, ring_bend, ring_waveform)
```
Blocks: LFOs and Math

Another way to create rich sounds is to vary parameters over time. The Note parameters that can be varied in this way are bend, amplitude, and ring amplitude.

Furthermore, multiple Math and LFO blocks can be connected together.

An LFO has these settable properties:

- waveform
- rate
- gain
- offset
- once

It also has a phase accumulator and an output value. Here’s how an LFO works:

- the phase advances over time, at the given rate. If rate is negative, then it "advances" towards smaller numbers.
- when the phase passes 1, it wraps around to 0. When going in reverse, values wrap from 0 back to 1.
- When once is True, the phase will stick at 0 or 1 rather than wrapping around.
- The waveform is indexed based on the phase. This value is multiplied by gain and then offset is added to create the final output value.

The waveform lets you create different ramps, the same as for notes.

```
lfo = synthio.LFO(rate=0.6, scale=0.05, phase_offset=0.0, once=False)
```

For more details, check out the full synthio documentation here.

Next, we’ll set up CircuitPython and the libraries needed, then get into the code examples.

---

Simple synthio Examples

Let’s get to some real-world examples! These are adapted from Tod Kurt’s excellent CircuitPython Synthio Tricks page.
import time
import board
import digitalio
import synthio

# for PWM audio with an RC filter
# import audiopwmio
# audio = audiopwmio.PWMAudioOut(board.GP10)

# for I2S audio with external I2S DAC board
import audiobusio

# I2S on Audio BFF or Amp BFF on QT Py:
# audio = audiobusio.I2SOut(bit_clock=board.A3, word_select=board.A2, data=board.A1)

# I2S audio on PropMaker Feather RP2040
power = digitalio.DigitalInOut(board.EXTERNAL_POWER)
power.switch_to_output(value=True)
audio = audiobusio.I2SOut(board.I2S_BIT_CLOCK, board.I2S_WORD_SELECT, board.I2S_DATA)

synth = synthio.Synthesizer(sample_rate=44100)
audio.play(synth)

while True:
    synth.press(65)  # midi note 65 = F4
time.sleep(0.5)
    synth.release(65)  # release the note we pressed
time.sleep(2)
import time
import board
import digitalio
import synthio

# for PWM audio with an RC filter
# import audiopwmio
# audio = audiopwmio.PWMAudioOut(board.GP10)

# for I2S audio with external I2S DAC board
import audiobusio
# I2S on Audio BFF or Amp BFF on QT Py:
# audio = audiobusio.I2SOut(bit_clock=board.A3, word_select=board.A2, data=board.A1)

# I2S audio on PropMaker Feather RP2040
power = digitalio.DigitalInOut(board.EXTERNAL_POWER)
power.switch_to_output(value=True)
audio = audiobusio.I2SOut(board.I2S_BIT_CLOCK, board.I2S_WORD_SELECT, board.I2S_DATA)
synth = synthio.Synthesizer(sample_rate=44100)
audio.play(synth)

while True:
    synth.press((65, 69, 72))  # midi note 65 = F4
    time.sleep(1)
    synth.release((65, 69, 72))  # release the note we pressed
    time.sleep(2)

# Simple Chord with Mixer

©Adafruit Industries
```python
mixer.voice[0].play(synth)
mixer.voice[0].level = 0.4

while True:
    synth.press((65, 69, 72))  # midi note 65 = F4
    time.sleep(0.5)
    synth.release((65, 69, 72))  # release the note we pressed
    time.sleep(0.5)
    mixer.voice[0].level = (mixer.voice[0].level - 0.1) % 0.4  # reduce volume each pass
```

```
# SPDX-FileCopyrightText: 2023 John Park and @todbot / Tod Kurt
#
# SPDX-License-Identifier: MIT

import time
import board
import digitalio
import audiomixer
import synthio

# for PWM audio with an RC filter
# import audiopwmio
# audio = audiopwmio.PWMAudioOut(board.GP10)

# for I2S audio with external I2S DAC board
import audiobusio

# I2S on Audio BFF or Amp BFF on QT Py:
# audio = audiobusio.I2SOut(bit_clock=board.A3, word_select=board.A2, data=board.A1)

# I2S audio on PropMaker Feather RP2040
power = digitalio.DigitalInOut(board.EXTERNAL_POWER)
power.switch_to_output(value=True)

# audio = audiobusio.I2SOut(board.I2S_BIT_CLOCK, board.I2S_WORD_SELECT, board.I2S_DATA)

mixer = audiomixer.Mixer(channel_count=1, sample_rate=22050, buffer_size=2048)

amp_env_slow = synthio.Envelope(
    attack_time=0.2,
    sustain_level=1.0,
    release_time=0.8
)

amp_env_fast = synthio.Envelope(
    attack_time=0.1,
    sustain_level=0.5,
    release_time=0.2
)```
```python
synth = synthio.Synthesizer(channel_count=1, sample_rate=22050,
envelope=amp_env_slow)

audio.play(mixer)
mixer.voice[0].play(synth)
mixer.voice[0].level = 0.2

while True:
    synth.envelope = amp_env_slow
    synth.press(46)
    time.sleep(1.25)
    synth.release(46)
    time.sleep(1.25)

    synth.envelope = amp_env_fast
    synth.press(51)
    time.sleep(1.25)
    synth.release(51)
    time.sleep(1.25)
```

# Filters

© Adafruit Industries
mixer = audiomixer.Mixer(channel_count=1, sample_rate=22050, buffer_size=2048)

amp_env_slow = synthio.Envelope(
    attack_time=0.2,
    sustain_level=1.0,
    release_time=0.8
)

synth = synthio.Synthesizer(channel_count=1, sample_rate=22050,
envelope=amp_env_slow)

# set up filters
frequency = 2000
resonance = 1.5
lpf = synth.low_pass_filter(frequency, resonance)
hpf = synth.high_pass_filter(frequency, resonance)
bpf = synth.band_pass_filter(frequency, resonance)

note1 = synthio.Note(frequency=330, filter=None)

audio.play(mixer)
mixer.voice[0].play(synth)
mixer.voice[0].level = 0.2

while True:
    # no filter
    note1.filter = None
    synth.press(note1)
    synth.release(note1)
    time.sleep(1.25)

    # lpf
    note1.filter = lpf
    synth.press(note1)
    synth.release(note1)
    time.sleep(1.25)

    # hpf
    note1.filter = hpf
    synth.press(note1)
    synth.release(note1)
    time.sleep(1.25)

    # bpf
    note1.filter = bpf
    synth.press(note1)
    synth.release(note1)
    time.sleep(1.25)
import board
import digitalio
import audiomixer
import synthio
import usb_midi
import adafruit_midi
from adafruit_midi.note_on import NoteOn
from adafruit_midi.note_off import NoteOff

# note for ESP32-S2 boards, due to not enough available endpoints,
# to enable USB MIDI, create a "boot.py" with following in it, then power cycle
board:
#  import usb_hid
#  import usb_midi
#  usb_hid.disable()
#  usb_midi.enable()
#  print("enabled USB MIDI, disabled USB HID")

# for PWM audio with an RC filter
# import audiopwmio
# audio = audiopwmio.PWMAudioOut(board.GP10)

# for I2S audio with external I2S DAC board
import audiobusio

# I2S on Audio BFF or Amp BFF on QT Py:
# audio = audiobusio.I2SOut(bit_clock=board.A3, word_select=board.A2, data=board.A1)

# I2S audio on PropMaker Feather RP2040
power = digitalio.DigitalInOut(board.EXTERNAL_POWER)
power.switch_to_output(value=True)
audio = audiobusio.I2SOut(board.I2S_BIT_CLOCK, board.I2S_WORD_SELECT, board.I2S_DATA)

mixer = audiomixer.Mixer(channel_count=1, sample_rate=22050, buffer_size=2048)
midi = adafruit_midi.MIDI(midi_in=usb_midi.ports[0], in_channel=0)
amp_env_med = synthio.Envelope(
    attack_time=0.05,
    sustain_level=0.8,
    release_time=0.2
)
synth = synthio.Synthesizer(channel_count=1, sample_rate=22050,
envelope=amp_env_med)

note1 = synthio.Note(frequency=330, filter=None)

audio.play(mixer)
mixer.voice[0].play(synth)
mixer.voice[0].level = 0.2

while True:
    msg = midi.receive()
    if isinstance(msg, NoteOn) and msg.velocity != 0:
        print("noteOn: ", msg.note, "vel:", msg.velocity)
        note1.frequency = synthio.midi_to_hz(msg.note)
        synth.press(note1)
    elif isinstance(msg, NoteOff) or isinstance(msg, NoteOn) and msg.velocity == 0:
        print("noteOff: ", msg.note, "vel:", msg.velocity)
        note1.frequency = synthio.midi_to_hz(msg.note)
        synth.release(note1)

# SPDX-FileCopyrightText: 2023 John Park and @todbot / Tod Kurt
#
# SPDX-License-Identifier: MIT

import board
import busio
import digitalio
import audiomixer
import synthio
import usb_midi
from adafruit_midi.note_on import NoteOn
from adafruit_midi.note_off import NoteOff
from adafruit_midi.control_change import ControlChange

# note for ESP32-S2 boards, due to not enough available endpoints,
# to enable USB MIDI, create a "boot.py" with following in it, then power cycle
# board:
#  import usb_hid
#  import usb_midi
#  usb_hid.disable()
#  usb_midi.enable()
#  print("enabled USB MIDI, disabled USB HID")

# for PWM audio with an RC filter
# import audiopwmio
# audio = audiopwmio.PWMAudioOut(board.GP10)

# for I2S audio with external I2S DAC board

©Adafruit Industries
import audiobusio

# I2S on Audio BFF or Amp BFF on QT Py:
# audio = audiobusio.I2SOut(bit_clock=board.A3, word_select=board.A2, data=board.A1)

# I2S audio on PropMaker Feather RP2040
power = digitalio.DigitalInOut(board.EXTERNAL_POWER)
power.switch_to_output(value=True)
audio = audiobusio.I2SOut(board.I2S_BIT_CLOCK, board.I2S_WORD_SELECT, board.I2S_DATA)
mixer = audiomixer.Mixer(channel_count=1, sample_rate=22050, buffer_size=2048)

midi_channel = 1
uart = busio.UART(tx=board.TX, rx=board.RX, baudrate=31250, timeout=0.001)
midi_usb = adafruit_midi.MIDI(midi_in=usb_midi.ports[0], in_channel=0)
midi_uart = adafruit_midi.MIDI(midi_in=uart, in_channel=midi_channel-1)

amp_env_med = synthio.Envelope(
    attack_time=0.05,
    sustain_level=0.8,
    release_time=0.2
)

synth = synthio.Synthesizer(channel_count=1, sample_rate=22050,
    envelope=amp_env_med)

# set up filters
filter_freq = 4000
filter_res = 0.5
filter_freq_lo = 100  # filter lowest freq
filter_freq_hi = 4500  # filter highest freq
filter_res_lo = 0.5  # filter q lowest value
filter_res_hi = 2.0  # filter q highest value
lpf = synth.low_pass_filter(filter_freq, filter_res)
note1 = synthio.Note(frequency=330, filter=lpf)

audio.play(mixer)
mixer.voice[0].play(synth)
mixer.voice[0].level = 0.2

def map_range(s, a1, a2, b1, b2):
    return b1 + ((s - a1) * (b2 - b1) / (a2 - a1))

while True:
    note1.filter = synth.low_pass_filter(filter_freq, filter_res)
    msg = midi_uart.receive() or midi_usb.receive()
    if isinstance(msg, NoteOn) and msg.velocity != 0:
        print("noteOn: ", msg.note, "vel:", msg.velocity)
        note1.frequency = synthio.midi_to_hz(msg.note)
        synth.press(note1)
    elif isinstance(msg, NoteOff) or isinstance(msg, NoteOn) and msg.velocity == 0:
        print("noteOff: ", msg.note, "vel:", msg.velocity)
        note1.frequency = synthio.midi_to_hz(msg.note)
        synth.release(note1)
    elif isinstance(msg, ControlChange):
        print("CC", msg.control, ", ", msg.value)
        if msg.control == 21:  # filter cutoff
            filter_freq = map_range(msg.value, 0, 127, filter_freq_lo,
                filter_freq_hi)
        elif msg.control == 22:  # filter Q
            filter_res = map_range(msg.value, 0, 127, filter_res_lo,
                filter_res_hi)
        elif msg.control == 7:  # volume
            mixer.voice[0].level = map_range(msg.value, 0, 127, 0.0, 1.0)
import board
import digitalio
import audiomixer
import synthio

# for PWM audio with an RC filter
# import audiopwmio
# audio = audiopwmio.PWMAudioOut(board.GP10)

# for I2S audio with external I2S DAC board
import audiobusio

# I2S on Audio BFF or Amp BFF on QT Py:
# audio = audiobusio.I2SOut(bit_clock=board.A3, word_select=board.A2, data=board.A1)

# I2S audio on PropMaker Feather RP2040
power = digitalio.DigitalInOut(board.EXTERNAL_POWER)
power.switch_to_output(value=True)
audio = audiobusio.I2SOut(board.I2S_BIT_CLOCK, board.I2S_WORD_SELECT, board.I2S_DATA)
mixer = audiomixer.Mixer(channel_count=1, sample_rate=44100, buffer_size=4096)
synth = synthio.Synthesizer(channel_count=1, sample_rate=44100)

audio.play(mixer)
mixer.voice[0].play(synth)
mixer.voice[0].level = 0.1

lfo = synthio.LFO(rate=0.6, scale=0.05)  # 1 Hz lfo at 0.25%
midi_note = 52
note = synthio.Note(synthio.midi_to_hz(midi_note), bend=lfo)
synth.press(note)

while True:
    pass
LFO Tremolo

```python
import time
import board
import digitalio
import audiomixer
import synthio
# for PWM audio with an RC filter
# import audiopwmio
# audio = audiopwmio.PWMAudioOut(board.GP10)

# for I2S audio with external I2S DAC board
import audiobusio

# I2S on Audio BFF or Amp BFF on QT Py:
# audio = audiobusio.I2SOut(bit_clock=board.A3, word_select=board.A2, data=board.A1)

# I2S audio on PropMaker Feather RP2040
power = digitalio.DigitalInOut(board.EXTERNAL_POWER)
power.switch_to_output(value=True)
audio = audiobusio.I2SOut(board.I2S_BIT_CLOCK, board.I2S.Word_SELECT, board.I2S_DATA)

mixer = audiomixer.Mixer(channel_count=1, sample_rate=44100, buffer_size=4096)
synth = synthio.Synthesizer(channel_count=1, sample_rate=44100)

audio.play(mixer)
mixer.voice[0].play(synth)
mixer.voice[0].level = 0.2

lfo_tremolo = synthio.LFO(rate=0.8, scale=0.3, offset=0.7)

midi_note = 50
note1 = synthio.Note(synthio.midi_to_hz(midi_note), amplitude=lfo_tremolo)
synth.press(note1)

while True:
    print(lfo_tremolo.value)
    time.sleep(0.05)
```

© Adafruit Industries
Advanced synthio Examples

Waveshapes
The default waveshape of a Note’s oscillator in synthio is a square wave. You can use any single-cycle waveshape imaginable, however, by creating a wave shape buffer with sample points in the shape of your wave.

These are then assigned to the Note’s waveform parameter.
import time
import board
import digitalio
import audiomixer
import synthio
import ulab.numpy as np

# for PWM audio with an RC filter
# import audiopwmio
# audio = audiopwmio.PWMAudioOut(board.GP10)

# for I2S audio with external I2S DAC board
import audiobusio

# I2S on Audio BFF or Amp BFF on QT Py:
# audio = audiobusio.I2SOut(bit_clock=board.A3, word_select=board.A2, data=board.A1)

# I2S audio on PropMaker Feather RP2040
power = digitalio.DigitalInOut(board.EXTERNAL_POWER)
power.switch_to_output(value=True)
audio = audiobusio.I2SOut(board.I2S_BIT_CLOCK, board.I2S_WORD_SELECT, board.I2S_DATA)
mixer = audiomixer.Mixer(channel_count=1, sample_rate=44100, buffer_size=4096)

amp_env_slow = synthio.Envelope(
    attack_time=0.15,
    sustain_level=1.0,
    release_time=0.8
)
synth = synthio.Synthesizer(channel_count=1, sample_rate=44100,
envelope=amp_env_slow)

audio.play(mixer)
mixer.voice[0].play(synth)
mixer.voice[0].level = 0.6

# create sine, tri, saw & square single-cycle waveforms to act as oscillators
SAMPLE_SIZE = 512
SAMPLE_VOLUME = 32000  # 0-32767
half_period = SAMPLE_SIZE // 2
wave_sine = np.array(np.sin(np.linspace(0, 2*np.pi, SAMPLE_SIZE, endpoint=False)) * SAMPLE_VOLUME,
    dtype=np.int16)
wave_saw = np.linspace(SAMPLE_VOLUME, -SAMPLE_VOLUME, num=SAMPLE_SIZE,
dtype=np.int16)
```python
wave_tri = np.concatenate((np.linspace(-SAMPLE_VOLUME, SAMPLE_VOLUME, num=half_period, dtype=np.int16),
                           np.linspace(SAMPLE_VOLUME, -SAMPLE_VOLUME, num=half_period, dtype=np.int16)))

wave_square = np.concatenate((np.full(half_period, SAMPLE_VOLUME, dtype=np.int16),
                               np.full(half_period, -SAMPLE_VOLUME, dtype=np.int16)))

midi_note = 65

while True:
    note1 = synthio.Note(synthio.midi_to_hz(midi_note), waveform=wave_sine, amplitude=1)
    synth.press(note1)
    time.sleep(0.75)
    synth.release(note1)
    time.sleep(0.75)

    note1 = synthio.Note(synthio.midi_to_hz(midi_note), waveform=wave_tri, amplitude=0.7)
    synth.press(note1)
    time.sleep(0.75)
    synth.release(note1)
    time.sleep(0.75)

    note1 = synthio.Note(synthio.midi_to_hz(midi_note), waveform=wave_saw, amplitude=0.25)
    synth.press(note1)
    time.sleep(0.75)
    synth.release(note1)
    time.sleep(0.75)

    note1 = synthio.Note(synthio.midi_to_hz(midi_note), waveform=wave_square, amplitude=0.2)
    synth.press(note1)
    time.sleep(0.75)
    synth.release(note1)
    time.sleep(0.75)
```

Waveshape Morphing

DIY waveshapes are cool, but even cooler is the ability to morph between waveshapes in real time!

This is done by creating the Note object with an empty waveform buffer, and then instead of replacing that buffer, we copy the new wave into it with with

```
note.waveform[:] = new_wave
```
import digitalio
import synthio
import ulab.numpy as np

# for PWM audio with an RC filter
# import audiopwmio
# audio = audiopwmio.PWMAudioOut(board.GP10)

# for I2S audio with external I2S DAC board
import audiobusio

# I2S on Audio BFF or Amp BFF on QT Py:
# audio = audiobusio.I2SOut(bit_clock=board.A3, word_select=board.A2, data=board.A1)

# I2S audio on PropMaker Feather RP2040
power = digitalio.DigitalInOut(board.EXTERNAL_POWER)
power.switch_to_output(value=True)
audio = audiobusio.I2SOut(board.I2S_BIT_CLOCK, board.I2S_WORD_SELECT, board.I2S_DATA)

mixer = audiomixer.Mixer(channel_count=1, sample_rate=44100, buffer_size=4096)

amp_env_slow = synthio.Envelope(
    attack_time=0.25,
    sustain_level=1.0,
    release_time=0.8
)
synth = synthio.Synthesizer(channel_count=1, sample_rate=44100, envelope=amp_env_slow)

audio.play(mixer)
mixer.voice[0].play(synth)
mixer.voice[0].level = 0.6

# create sine, tri, saw & square single-cycle waveforms to act as oscillators
SAMPLE_SIZE = 512
SAMPLE_VOLUME = 32000  # 0-32767
half_period = SAMPLE_SIZE // 2
wave_sine = np.array(np.sin(np.linspace(0, 2*np.pi, SAMPLE_SIZE, endpoint=False)) * SAMPLE_VOLUME, dtype=np.int16)
wave_saw = np.linspace(SAMPLE_VOLUME, -SAMPLE_VOLUME, num=SAMPLE_SIZE, dtype=np.int16)
wave_tri = np.concatenate((np.linspace(-SAMPLE_VOLUME, SAMPLE_VOLUME, num=half_period, dtype=np.int16),
                           np.linspace(SAMPLE_VOLUME, -SAMPLE_VOLUME, num=half_period, dtype=np.int16)))
wave_square = np.concatenate((np.full(half_period, SAMPLE_VOLUME, dtype=np.int16),
                              np.full(half_period, -SAMPLE_VOLUME, dtype=np.int16)))

def lerp(a, b, t):  # function to morph shapes w linear interpolation
    return (1-t) * a + t * b

wave_empty = np.zeros(SAMPLE_SIZE, dtype=np.int16)  # empty buffer we use array slice copy "[:]" on
note1 = synthio.Note(frequency=440, waveform=wave_empty, amplitude=0.6)
synth.press(note1)

pos = 0
my_wave = wave_empty

while True:
    while pos <= 1.0:
        print(pos)
        pos += 0.01
        my_wave[:] = lerp(wave_sine, wave_saw, pos)
Detuning Oscillators for Fatter Sound

Since we have fine-grained control over a note's frequency with `note.frequency`, this means we can do a common technique for getting a "fatter" sound. When a note is played at a specific pitch, a second note object is created with a slightly shifted pitch, which adds organic "movement" and a sort of chorusing effect to the notes.

We can stack up a bunch of these progressively further detuned notes to create a huge wall of synth awesomeness!

```python
# SPDX-FileCopyrightText: 2023 John Park and @todbot / Tod Kurt
# SPDX-License-Identifier: MIT

import time
import board
import audiomixer
import digitalio
import synthio
import ulab.numpy as np

# for PWM audio with an RC filter
# import audiopwmio
# audio = audiopwmio.PWMAudioOut(board.GP10)

# for I2S audio with external I2S DAC board
import audiobusio

# I2S on Audio BFF or Amp BFF on QT Py:
# audio = audiobusio.I2SOut(bit_clock=board.A3, word_select=board.A2, data=board.A1)

# I2S on PropMaker Feather RP2040
power = digitalio.DigitalInOut(board.EXTERNAL_POWER)
power.switch_to_output(value=True)

audio = audiobusio.I2SOut(board.I2S_BIT_CLOCK, board.I2S_WORD_SELECT, board.I2S_DATA)
mixer = audiomixer.Mixer(channel_count=1, sample_rate=44100, buffer_size=4096)
amp_env_slow = synthio.Envelope(
    attack_time=0.65,
    sustain_level=1.0,
    release_time=0.8
)
synth = synthio.Synthesizer(channel_count=1, sample_rate=44100,
envelope=amp_env_slow)

audio.play(mixer)
mixer.voice[0].play(synth)
mixer.voice[0].level = 0.3
```
# create sine, tri, saw & square single-cycle waveforms to act as oscillators
SAMPLE_SIZE = 512
SAMPLE_VOLUME = 32000  # 0-32767
half_period = SAMPLE_SIZE // 2
wave_sine = np.array(np.sin(np.linspace(0, 2*np.pi, SAMPLE_SIZE, endpoint=False)) * SAMPLE_VOLUME, dtype=np.int16)
wave_saw = np.linspace(SAMPLE_VOLUME, -SAMPLE_VOLUME, num=SAMPLE_SIZE, dtype=np.int16)
wave_tri = np.concatenate((np.linspace(-SAMPLE_VOLUME, SAMPLE_VOLUME, num=half_period, dtype=np.int16),
                           np.linspace(SAMPLE_VOLUME, -SAMPLE_VOLUME, num=half_period, dtype=np.int16)))
wave_square = np.concatenate((np.full(half_period, SAMPLE_VOLUME, dtype=np.int16),
                              np.full(half_period, -SAMPLE_VOLUME, dtype=np.int16)))

# note1 = synthio.Note( frequency = 220, waveform = wave_sine, amplitude=0.3)
detune = 0.003  # how much to detune
num_oscs = 1
midi_note = 52

while True:
    print("num_oscs:", num_oscs)
    notes = []  # holds note objs being pressed
    # simple detune, always detunes up
    for i in range(num_oscs):
        f = synthio.midi_to_hz(midi_note) * (1 + i*detune)
        notes.append(synthio.Note(frequency=f, waveform=wave_saw))
synth.press(notes)
time.sleep(3.6)
synth.release(notes)
time.sleep(0.1)
    # increment number of detuned oscillators
    num_oscs = num_oscs+1 if num_oscs < 5 else 1

Wavetables

# SPDX-License-Identifier: MIT
# wavetable_midisynth_code_i2s.py -- simple wavetable synth that responds to MIDI
# 26 Jul 2023 - @todbot / Tod Kurt
# Demonstrate using wavetables to make a MIDI synth
# Needs WAV files from waveeditonline.com
# - BRAIDS01.WAV - http://waveeditonline.com/index-17.html

import time
import busio
import board
import audiomixer
import synthio
import digitalio
import audiobusio
import ulab.numpy as np
import adafruit_wave

import usb_midi
import adafruit_midi
from adafruit_midi.note_on import NoteOn
from adafruit_midi.note_off import NoteOff
from adafruit_midi.control_change import import ControlChange
auto_play = False  # set to true to have it play its own little song
auto_play_notes = [36, 38, 40, 41, 43, 45, 46, 48, 50, 52]
auto_play_speed = 0.9  # time in seconds between notes

midi_channel = 1

wavetable_fname = "wav/BRAIDS01.WAV"  # from http://waveeditonline.com/index-17.html
wavetable_sample_size = 256  # number of samples per wave in wavetable (256 is standard)
sample_rate = 44100
wave_lfo_min = 0  # which wavetable number to start from 10
wave_lfo_max = 6  # which wavetable number to go up to 25

# for PWM audio with an RC filter
# Pins used on QTPY RP2040:
# - board.MOSI - Audio PWM output (needs RC filter output)
# import audiopwmio
# audio = audiopwmio.PWMAudioOut(board.GP10)

# for I2S audio with external I2S DAC board
# import audiobusio

# I2S on Audio BFF or Amp BFF on QT Py:
# audio = audiobusio.I2SOut(bit_clock=board.A3, word_select=board.A2, data=board.A1)

# I2S audio on PropMaker Feather RP2040
power = digitalio.DigitalInOut(board.EXTERNAL_POWER)
power.switch_to_output(value=True)

audio = audiobusio.I2SOut(board.I2S_BIT_CLOCK, board.I2S_WORD_SELECT, board.I2S_DATA)
mixer = audiomixer.Mixer(buffer_size=4096, voice_count=1, sample_rate=sample_rate, channel_count=1, bits_per_sample=16, samples_signed=True)

mixer.voice[0].play(synth)  # attach synth to mixer

uart = busio.UART(tx=board.TX, rx=board.RX, baudrate=31250, timeout=0.001)
midi_uart = adafruit_midi.MIDI(midi_in=uart, in_channel=midi_channel-1)
midi_usb = adafruit_midi.MIDI(midi_in=uart, in_channel=midi_channel-1)

# mix between values a and b, works with numpy arrays too, t ranges 0-1
def lerp(a, b, t):
    return (1-t)*a + t*b

class Wavetable:
    """ A 'waveform' for synthio.Note that uses a wavetable w/ a scannable wave position."""
    def __init__(self, filepath, wave_len=256):
        self.w = adafruit_wave.open(filepath)
        self.wave_len = wave_len  # how many samples in each wave
        if self.w.getsampwidth() != 2 or self.w.getnchannels() != 1:
            raise ValueError("unsupported WAV format")
        self.waveform = np.zeros(wave_len, dtype=np.int16)  # empty buffer we'll copy into
        self.num_waves = self.w.getnframes() // self.wave_len
        self.set_wave_pos(0)

    def set_wave_pos(self, pos):
        """Pick where in wavetable to be, morphing between waves"""
        pos = min(max(pos, 0), self.num_waves-1)  # constrain
        samp_pos = int(pos) * self.wave_len  # get sample position
        self.w.setpos(samp_pos)
        waveA = np.frombuffer(self.w.readframes(self.wave_len), dtype=np.int16)
        one_up = self.w.setpos(samp_pos + self.wave_len)
        waveB = np.frombuffer(self.w.readframes(self.wave_len), dtype=np.int16)
pos_frac = pos - int(pos)  # fractional position between wave A & B
self.waveform[:] = lerp(waveA, waveB, pos_frac)  # mix waveforms A & B

wavetable1 = Wavetable(wavetable_fname, wave_len=wavetable_sample_size)

amp_env = synthio.Envelope(attack_level=0.2, sustain_level=0.2, attack_time=0.05, release_time=0.3, decay_time=.5)
wave_lfo = synthio.LFO(rate=0.2, waveform=np.array((0, 32767), dtype=np.int16))
lpf = synth.low_pass_filter(4000, 1)  # cut some of the annoying harmonics

synth.blocks.append(wave_lfo)  # attach wavelfo to global lfo runner since cannot attach to note

notes_pressed = {}  # keys = midi note num, value = synthio.Note,
def note_on(notenum):
    # release old note at this notenum if present
    if oldnote := notes_pressed.pop(notenum, None):
        synth.release(oldnote)

    if not auto_play:
        wave_lfo.retrigger()

    f = synthio.midi_to_hz(notenum)

    vibrato_lfo = synthio.LFO(rate=1, scale=0.01)
    note = synthio.Note(frequency=f, waveform=wavetable1.waveform,
                        envelope=amp_env, filter=lpf, bend=vibrato_lfo)
    synth.press(note)
    notes_pressed[notenum] = note

def note_off(notenum):
    if note := notes_pressed.pop(notenum, None):
        synth.release(note)


def set_wave_lfo_minmax(wmin, wmax):
    scale = (wmax - wmin)
    wave_lfo.scale = scale
    wave_lfo.offset = wmin


last_synth_update_time = 0
def update_synth():
    # pylint: disable=global-statement
    global last_synth_update_time
    # only update 100 times a sec to lighten the load
    if time.monotonic() - last_synth_update_time > 0.01:
        # last update time = time.monotonic()
        wavetable1.set_wave_pos( wave_lfo.value )


last_auto_play_time = 0
auto_play_pos = -1
def update_auto_play():
    # pylint: disable=global-statement
    global last_auto_play_time, auto_play_pos
    if auto_play and time.monotonic() - last_auto_play_time > auto_play_speed:
        last_auto_play_time = time.monotonic()
        note_off( auto_play_notes[ auto_play_pos ] )
        auto_play_pos = (auto_play_pos + 3) % len(auto_play_notes)
        note_on( auto_play_notes[ auto_play_pos ] )

set_wave_lfo_minmax(wave_lfo_min, wave_lfo_max)

def map_range(s, a1, a2, b1, b2):
    return b1 + ((s - a1) * (b2 - b1) / (a2 - a1))
print("wavetable midisynth i2s. auto_play:", auto_play)

while True:
    update_synth()
    update_auto_play()

    msg = midi_uart.receive() or midi_usb.receive()

    if isinstance(msg, NoteOn) and msg.velocity != 0:
        note_on(msg.note)
    elif isinstance(msg, NoteOff) or isinstance(msg, NoteOn) and msg.velocity == 0:
        note_off(msg.note)
    elif isinstance(msg, ControlChange):
        if msg.control == 21:  # mod wheel
            scan_low = map_range(msg.value, 0, 127, 0, 64)
            set_wave_lfo_minmax(scan_low, scan_low)

Larger Examples

Tyrell Desktop Synth
This synthio-based creation is an adaptation of Tod Kurt's eighties_dystopia() synth. It combines many of the concepts we've covered, including:

detuned oscillators
LFO modulation of filter cutoff
saw wave oscillators
user pitch input

For more info, check out the full Learn Guide().
Circle of Fifths Euclidean Synth

This synth by Liz Clark () celebrates all things circular: the circle of fifths, Euclidean rhythms, and rotary encoders. Four synth voices play random notes in a triad to the beat of a determined Euclidean rhythm animated on the 8x8 matrix. You can scroll through the circle of fifths () on each synth voice to change the triad for easy modulation between keys.

For more info, check out the full Learn Guide ().

Computer Perfection Synthesizer

This project takes the Computer Perfection and reuses its buttons and switches to trigger a polyphonic, multi-timbral wavetable synthesizer for all your spacey jam sessions. It includes ADSR envelopes and LFO modulation for a beautiful, otherworldly sound.

For more info, check out the full Learn Guide ().

Monosynth

Another awesome Tod Kurt creation, this is a monosynth created in synthio () that you can control with a MIDI keyboard.

You can plug it into your computer or other USB MIDI Host, or a keyboard via UART (classic DIN-5 or TRS MIDI).
Wavetable Polysynth
Tod's done it again. This is a syntio-based polyphonic synth that uses a Mutable Instruments Plaits wavetable for wave shape morphing source.

Mini Arpeggiator
This "todbot" character sure is prolific! Here's his self-contained arpeggiator synth (with controls for root note, BPM, pattern, and pattern octaves) called eighties_arp.