Circuit Playground Express USB MIDI Controller and Synthesizer

Created by Kevin Walters

https://learn.adafruit.com/cpx-midi-controller

Last updated on 2024-06-03 02:44:21 PM EDT

Table of Contents

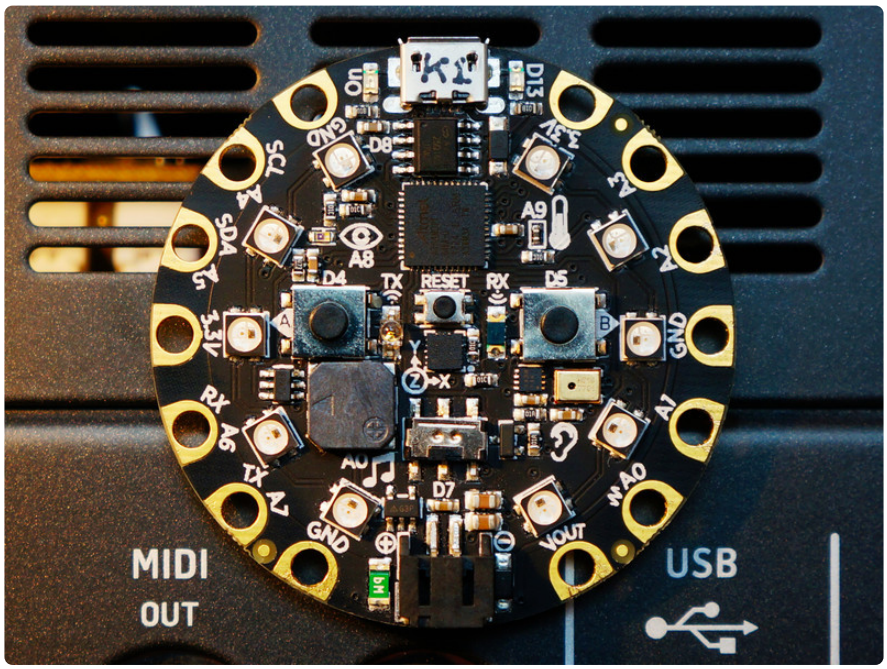
Overview

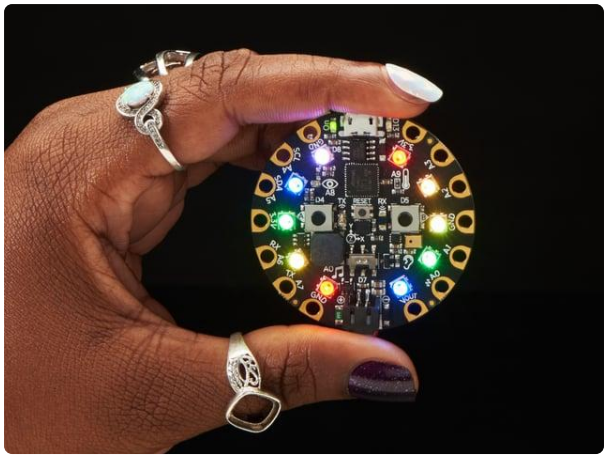
This project turns the Circuit Playground Express (CPX) board into a touch pad USB MIDI controller (https://adafru.it/EJT) with tilt control for modulation and pitch bend similar to the excellent Trellis M4 Expressive MIDI controller (https://adafru.it/EJU) project.

A second partner program creates a basic MIDI (https://adafru.it/EJV)-enabled synthesizer using the onboard speaker.

The USB MIDI features require CircuitPython 4.x or higher and a recent version of the `adafruit_midi` library.

No additional hardware is required beyond the Circuit Playground Express board. Multiple boards can be used to increase the note range on the controller or for simple polyphony (https://adafru.it/EJW) on the synthesizer.





Circuit Playground Express

Circuit Playground Express is the next step towards a perfect introduction to electronics and programming. We've taken the original Circuit Playground Classic and...

https://www.adafruit.com/product/3333

USB A/Micro Cable - 2m

This is your standard USB A-Plug to Micro-USB cable. It's 2 meters long so you'll have plenty of cord to work with for those longer extensions.

https://www.adafruit.com/product/2185

Controllers and Instruments

MIDI History

Early electronic instruments like the RCA Synthesizer Mark II (https://adafru.it/EJX) or Siemens Synthesizer (https://adafru.it/EJY) (above) from 1950s were built by one manufacturer with proprietary interconnections. The advent of modular synthesizers (https://adafru.it/EJZ) led to a simple interface with analogue signals in the form of a control voltage and gate (https://adafru.it/EJ-) signal pair which could connect keyboards (a controller) and sequencers to other electronic instruments.

In the early 1980s, manufacturers collaborated on the first MIDI standard which was published in 1983. From www.midi.org (https://adafru.it/EJV):

> MIDI is an industry standard music technology protocol that connects products from many different companies including digital musical instruments, computers, tablets, and smartphones. MIDI is used every day around the world by musicians, DJs, producers, educators, artists, and hobbyists to create, perform, learn, and share music and artistic works.
>
> MIDI Manufacturer's Association

Colin's Lab: MIDI (https://adafru.it/EK0) is a great video introduction to MIDI. There is also a transcript of the video.

Expressive Controllers

Some instruments allow more variation (modulation) of the note than others. For example, the main variation a pianist can achieve is through the velocity of the key strike which determines how the hammer hits the strings. Other instruments like the violin have more possibilities for continuous modulation, i.e. variation of the note while it's being played.

For electronic instruments, the key-less theremin (https://adafru.it/pFU) is perhaps the most well-known, expressive one. A less well-known early electronic instrument is the remarkable ondes Martenot (https://adafru.it/EK1) which had a continuous pitch ring, lateral movement for pitch on later keybeds and a pressure sensitive button ("lozenge") for varying volume (amplitude modulation). The pre-MIDI Yamaha CS-80 (https://adafru.it/EK2) was first synthesizer with polyphonic aftertouch (https://adafru.it/EK3), modulation achieved by placing additional pressure on the depressed key(s) after the initial strike.

ROLI (https://adafru.it/EK4) make the most common, modern, highly expressive keyboards (https://adafru.it/EK5) using the relatively new MIDI Polyphonic Expression (MPE) (https://adafru.it/stC) standard. Another example is the MI.MU Glove (https://adafru.it/Eqs) controller and its simpler cousin the (non-MIDI) MINI.MU Glove (http://adafru.it/4141).

CPX and MIDI

The `adafruit_midi` CircuitPython library includes support for sending and receiving MIDI messages (sometimes called events (https://adafru.it/EK6)) over USB. This library is used by the two programs in this guide.

MIDI controller

The first program creates a MIDI controller using the 7 capacitive touch pads on the CPX as keys and the onboard accelerometer to apply expressive effects by measuring tilt to control the pitch bend and modulation wheels. The two CPX buttons are used to change octave/semitone offset and the keyboard scale. The switch inhibits the tilt control and selects between octave and semitone change.

Basic Synthesizer

The second program creates a MIDI-enabled synth on the CPX which plays a sawtooth wave of the appropriate pitch and volume for the incoming notes. Pitch bending and a limited form of amplitude modulation are implemented.

Both programs uses the NeoPixels to represent the note pressed/played.

CircuitPython

CircuitPython 4 or higher is required to support USB MIDI.

If you are new to CircuitPython, see Welcome to CircuitPython! (https://adafru.it/cpy-welcome)

Adafruit suggests using the Mu editor to edit your code and have an interactive REPL in CircuitPython. You can learn about Mu and its installation in this tutorial (https://adafru.it/ANO).

Libraries

Download the latest set of libraries for CircuitPython to match the version of CircuitPython you are running. There is one library package for CircuitPython 3.x, 4.x, and so on. Click the box below and download the library bundle to your computer.

Click to go to the latest Adafruit CircuitPython Library Bundle Release Page

https://adafru.it/y8E

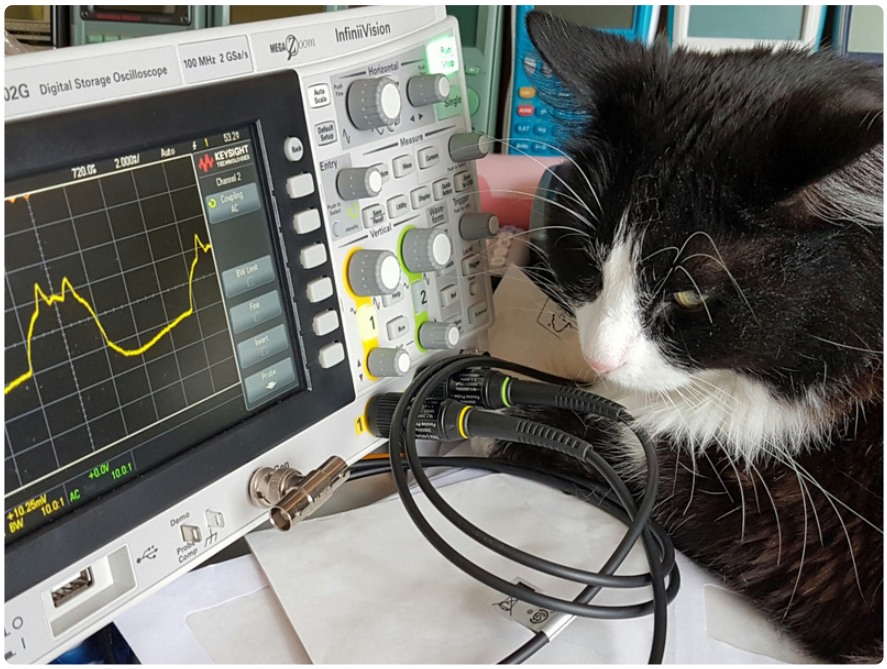
Open the library Zip file and copy the following files/directories onto the CPX **CIRCUITPY** drive in a directory called **/lib**. This library is the one used by the programs:

- `adafruit_midi` (must be adafruit-circuitpython-bundle-4.x-mpy-20190507.zip or later)

See the CircuitPython Libraries (https://adafru.it/Cqa) guide for additional details on how to add libraries.

This code has been tested on CircuitPython 4.0.0 rc1 and the adafruit-circuitpython-bundle-4.x-mpy-20190507 libraries.

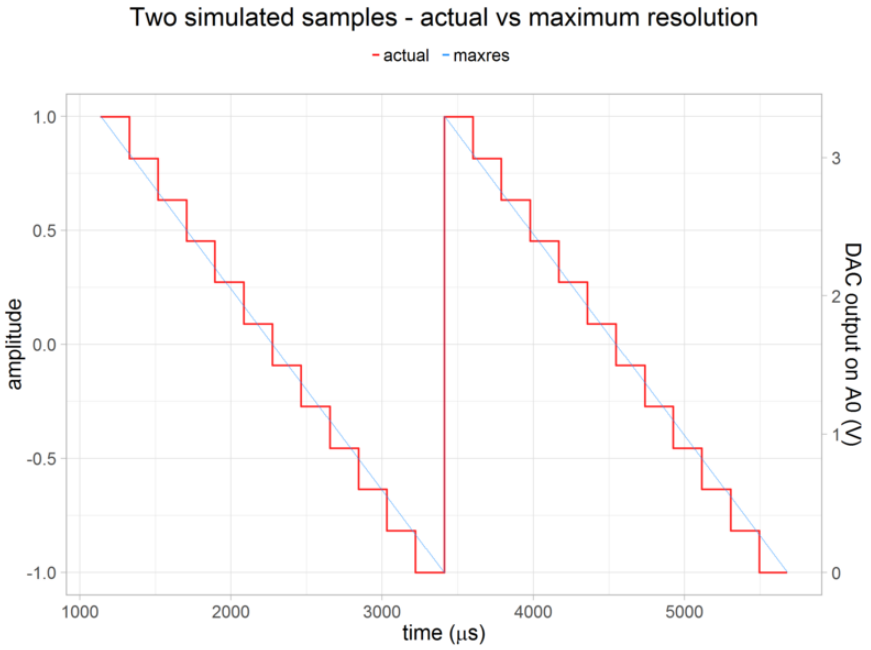
Waveforms



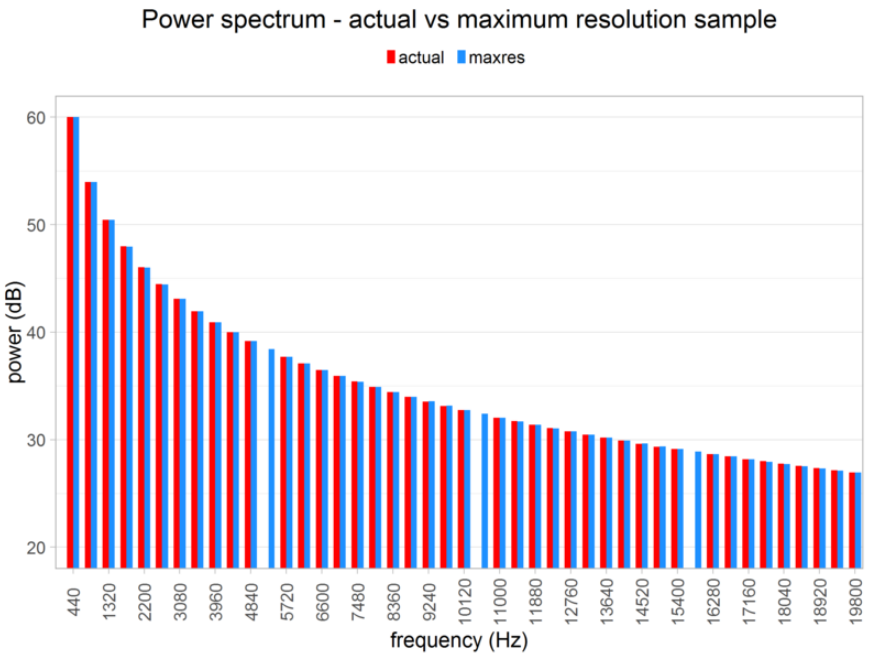
The CPX basic synthesizer uses a small sample made from a list of 12 values to create a low resolution sawtooth wave. This sample is played repeatedly as a loop at a playback rate adjusted to match the desired note frequency. For example, the note $\mathbf{A_4}$ (440Hz) would be played at 12 * 440 = 5280 samples per second.

The CPX has an upper playback limit of 350000 samples per second which dictates an upper frequency for this sample of 29167Hz, well into the inaudible, ultrasonic (https://adafru.it/EK7) range. This technique to change the audible frequency of a sample clearly has limitations with large samples.

Two cycles of the low resolution, "steppy (https://adafru.it/EK8)" sawtooth wave are shown below alongside the maximum resolution equivalent. CircuitPython uses values 0-65535 (16bit) to represent the output value/voltage. The CPX's SAMD21 processor's digital to analogue converter (DAC) (https://adafru.it/EK9) is only 10bit, reducing (quantizing (https://adafru.it/EKa)) the resolution to 1024 values.



The two different resolution sawtooth waves can be seen decomposed into multiple sine waves (https://adafru.it/EKb) of different frequencies on the audio spectra below. This is what would be seen on a spectrum analyzer (https://adafru.it/EKc) analysing either the the signal from the DAC output (touchpad **A0**) or that signal played through a high fidelity sound system. In this case, the bar graph was generated mathematically from the magnitude of the Discrete Fourier Transform (DFT) (https://adafru.it/EKd).



The expected highest (loudest) frequency is the fundamental occurring at 440Hz and then extra harmonics at 880Hz (2x) and 1320Hz (3x) and so on. The steppy quality of the low resolution sawtooth removes the 5280Hz, 10560Hz and 15840Hz harmonics, visible as absent red bars in the graph.

The graph stops at 20kHz, an oft-used upper limit for human hearing (https://adafru.it/EKe). The curve suggests the harmonics continue into the ultrasound range. This can be problematic if the audio signal is going to be used (sampled) by another digital device without filtering (https://adafru.it/EKf) to remove the higher frequencies.

The actual sound from the CPX is further affected by the characteristics of the onboard (Class D) amplifier (https://adafru.it/EKg) and the tiny 7.5mm speaker.

An explanation for why humans perceive sound mostly in terms of the frequencies of the constituent sine waves can be found in Chapter 1 of David Benson's Book, Music: A Mathematical Offering (https://adafru.it/EKh).

Simple Computer Sound Generation

It is trivial for a digital computer to generate square waves where the signal simply rises and falls with the pauses between determining the frequency and duty cycle (https://adafru.it/EKi). These are commonly used for computer clock signals (https://adafru.it/EjP) but can also be used for generating audio. This would seem like an obvious, straightforward choice for generating sounds, It was used for:

- the beeps from the Binatone TV Master IV (https://adafru.it/EKj),
- the primitive, mono 1bit output of the ZX Spectrum and the IBM PC, the latter being 8254 (https://adafru.it/EKk) driven,
- the two channel Atari (https://adafru.it/EKl) Television Interface Adapter (TIA) (https://adafru.it/EKm) used in the Atari 2600 (https://adafru.it/EKn),
- the multi-channel General Instrument AY 3 8910 (https://adafru.it/EKo) and Texas Instruments SN76489 (https://adafru.it/EKp) PSG (https://adafru.it/EKq) sound chips,
- and the Game Boy's Sharp LR35902 CPU (https://adafru.it/eCk) with integrated sound including one channel with a short user-defined waveform.

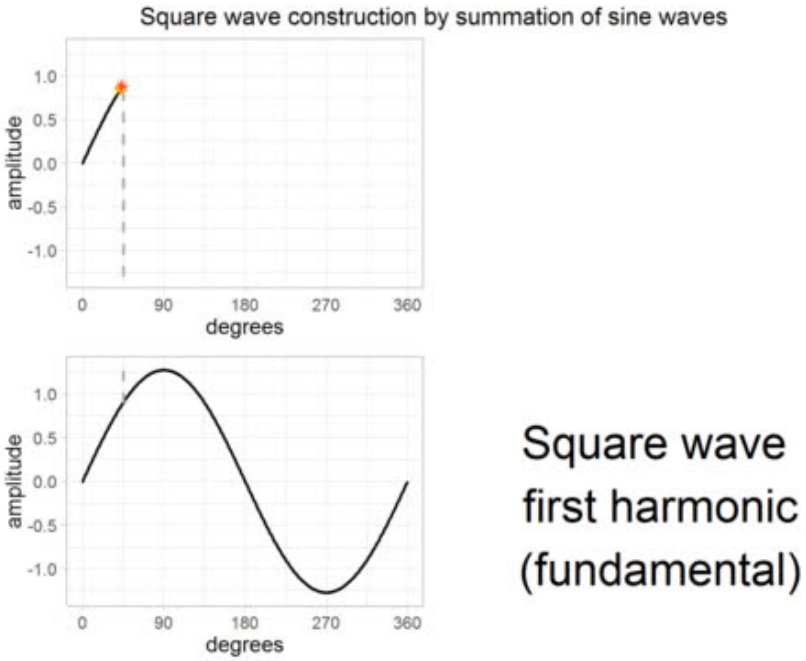
Waveforms can also be generated by composition. The next two sections compare the square and sawtooth wave to explore how the wave shape develops as more sine waves are added and how the two differ.

Square Wave Construction

The animation below shows construction of a square wave from the addition of 10 sine (https://adafru.it/EKr) waves. The overtones of a square wave occur at odd

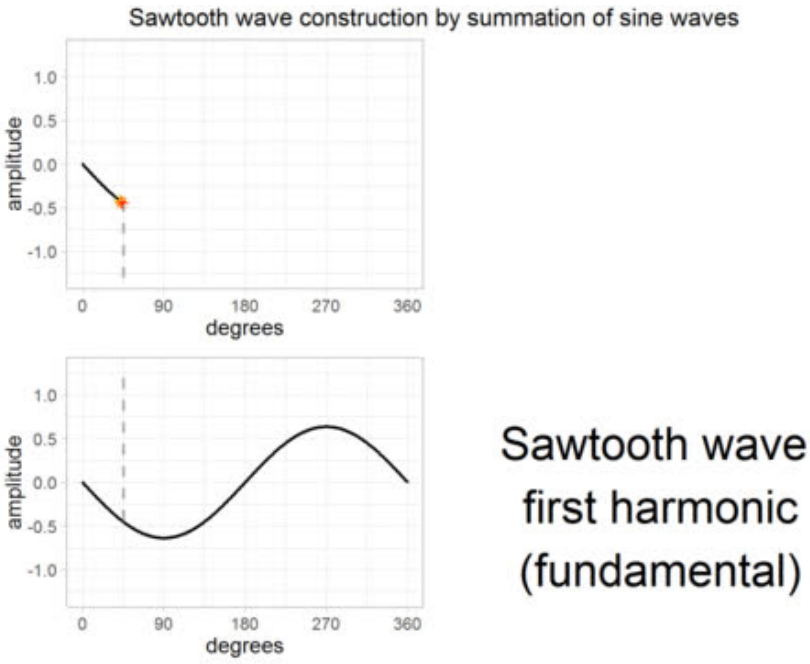
multiples of the fundamental frequency, this makes it less rich in harmonics than the sawtooth wave and gives it a different sound or timbre (https://adafru.it/EKs).

It's interesting to see how the shape of the final waveform becomes more straight/perpendicular as each sine wave is added. This is still far from perfect in terms of shape with ten harmonics but likely to be a better choice for digital audio applications to prevent undesirable aliasing (https://adafru.it/EKt) effects.



Sawtooth Wave Construction

A sawtooth wave made from 10 harmonics is shown below for comparison. This one is "upside down" compared to the one that the CPX is generating but this makes no difference to the magnitude of the frequencies of the harmonics.



The wikipedia pages for sawtooth wave (https://adafru.it/EKu) and square wave (https://adafru.it/EKv) have audio clips allowing for a comparison of the two different timbres.

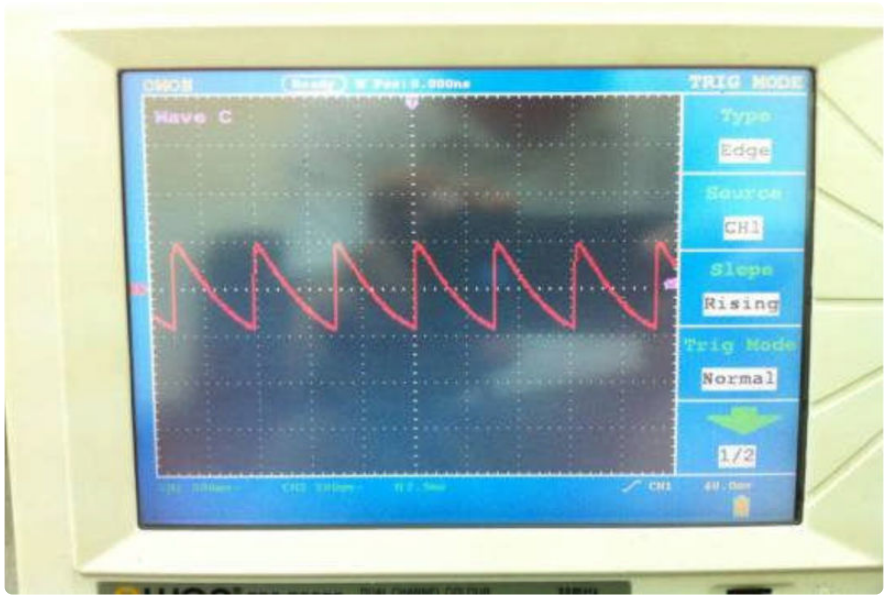
Jess Swanson's An Interactive Introduction to Fourier Transforms (https://adafru.it/EKw) is a fun and interactive way to explore waveforms more.

Analogue Synthesizer Waves

The Korg Volca Bass, part of the 2010s renaissance in analogue synthesizers, offers both sawtooth and square waves as the basis for creating sounds. The dual-purpose keys which allow waveform selection are shown below for the three (voltage controlled) oscillators (VCO) (https://adafru.it/EKx). The synthesizer's low-pass filter can be used to reduce higher harmonics, changing the timbre of the sound. This technique is referred to as subtractive synthesis (https://adafru.it/EKy). The opposite approach is additive synthesis (https://adafru.it/EKz) which creates sounds by adding sine waves like the animations above.

The low frequency oscillator (LFO) (below, on the left) is used to modulate the volume (amplitude) or pitch or filter cutoff frequency. This is offered in triangle (https://adafru.it/EKA) or square waveforms.

The actual sawtooth wave can be seen in an example of the audio output voltage from one oscillator in the service manual, shown below. The slightly steppy quality seen here is due to the resolution of the digital oscilloscope.



On an incidental note, the Korg Volca (and Monotron) synthesizers appear to have been designed to be easily modified by their owners - the circuit board has clearly labelled solder points (https://adafru.it/EKB). Hobbyist hardware modification for electronic instruments is referred to as circuit bending (https://adafru.it/EKC).

Going Further

Ideas for Areas to Explore

- Make a keyboard for the CPX MIDI controller:
    - "traditional" fruit: Circuit Playground Fruit Drums: Cirkey Cirkey (https://adafru.it/EKD),
    - crocodile clips only: Circuit Playground Fruit Drums: Tone Piano (https://adafru.it/EKE),
    - conductive paint: Bare Conductive: How to make a MIDI piano with the Touch Board (https://adafru.it/EKF) (silk screen printed, may be harded to hand paint),
    - copper tape / aluminium foil: Wendian Jiang: Automatic Notation System with Capacitive Touch Piano (https://adafru.it/EKG) (page 6).
- Enhance the note representation on the NeoPixels to include pitch bending.
- Change the synth waveform to a sine wave, a square wave or noise made from random values (audible result may surprise).
- Write a separate program to use the onboard microphone to record and process a very short sample for use in the synthesizer.
- Use the light sensor to modulate another MIDI cc.
- Implement MIDI "poly chaining" to send notes to the next CPX board when more than one note is played simultaneously.
- Explore the capabilities of the new audioio.Mixer (https://adafru.it/EKH) class - this could be used for limited polyphony and smoother pitch bending.
- Emulate a Leslie speaker (https://adafru.it/EKI) by mounting your CPX synth on a servo and wiggling it.

The boards based on the Cortex M0 processor (SAMD21) like the CPX have 32kB of memory. The M4 (SAMD51) boards with 192kB like the NeoTrellis M4 (http://adafru.it/4020) are likely to be a better choice for larger, more sophisticated programs using the `adafruit_midi` library. The NeoTrellis M4 also includes stereo 12bit DACs with 3.5mm audio output.

Related Projects

There are many projects using the NeoTrellis M4 (https://adafru.it/D1J), the ones below are just a small selection.

MIDI

- Trellis M4 Expressive MIDI Controller (https://adafru.it/EJU) - includes some useful background information on how MIDI works and how to use and connect iOS tablets.
- Grand Central USB MIDI Controller in CircuitPython (https://adafru.it/EKJ).
- NeoTrellis M4 MIDI File Synthesizer (https://adafru.it/DkO).

Non-MIDI

- Make It Sound (https://adafru.it/EKK) (CircuitPython and MakeCode examples).
- Circuit Playground Express: Piano in the Key of Lime (https://adafru.it/EKL).
- Touch Tone for Circuit Playground Express (https://adafru.it/EKM).
- Circuit Playground Express: Playground Drum Machine (https://adafru.it/BeH).
- Circuit Playground Fruit Drums (https://adafru.it/EKN) - this uses the Circuit Playground Classic which has one extra touch-capable pad but has a smaller CPU and **cannot run** CircuitPython.
- Circuit Playground Musical Glove (https://adafru.it/EKO) - this is inspired by the MINI.MU Glove Kit (http://adafru.it/4141) but uses MakeCode on the CPX.
- Matt Stanton's CPX Glove Synth (https://adafru.it/sTc) code (C++/Arduino).

Further Reading and Listening

- Collin's Lab: MIDI (https://adafru.it/EKP) (YouTube video with transcript).
- What is Web MIDI & BLE MIDI (https://adafru.it/EKQ) (BLE=Bluetooth Low Energy (https://adafru.it/Di6)).
- Gizmodo: A Beginner's Guide to the Synth (https://adafru.it/EKR).
- Circuit Playground Analog Input: Analog vs. Digital (https://adafru.it/xsb).
- The Curious Cases of Rutherford & Fry: An Instrumental Case: Why do instruments sound different? (https://adafru.it/EKS) (podcast, click on yellow Download button, main programme up to 00:21:30).
- The Synthesizer Academy (https://adafru.it/EKT) (online guides).
- Kenny McAlpine: Bits and Pieces - A History of Chiptunes (https://adafru.it/EKU) - a well researched book covering both the music and technology behind the Atari VCS, ZX Spectrum, Commodore 64, NES, Game Boy and more.
- Steve W. Smith: The Scientist and Engineer's Guide to (https://adafru.it/EKV) Digital Signal Processing (https://adafru.it/EKV) - an in-depth book available for free online.
- R code used to generate graphs (https://adafru.it/EKW) on the Waveforms page.

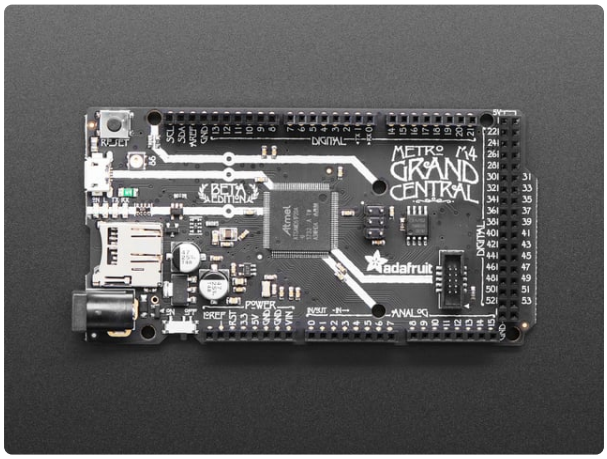
Selected Music

- Music From Mathematics album (https://adafru.it/EKX) (music made with IBM 7090 (https://adafru.it/EKY) in 1962).
- Pixelh8 (Matthew Applegate) - Obsolete? (bandcamp (https://adafru.it/EKZ)) - commissioned by The National Museum of Computing (https://adafru.it/EK-) (UK).
- Tristan Perich (https://adafru.it/EL0) - 1-Bit symphony (https://adafru.it/EL1) (bandcamp (https://adafru.it/EL2)).

Adafruit NeoTrellis M4 with Enclosure and Buttons Kit Pack

So you've got a cool/witty name for your band, a Soundcloud account, a 3D-printed Daft Punk...

https://www.adafruit.com/product/4020



Adafruit Grand Central M4 Express featuring the SAMD51

Are you ready? Really ready? Cause here comes the Adafruit Grand Central featuring the Microchip ATSAMD51. This dev board is so big, it's not...

https://www.adafruit.com/product/4064

Bare Conductive Paint - 50mL

Bare Conductive Paint is a multipurpose electrically conductive material perfect for all of your DIY projects! Bare Paint is water based, nontoxic and dries at room temperature.

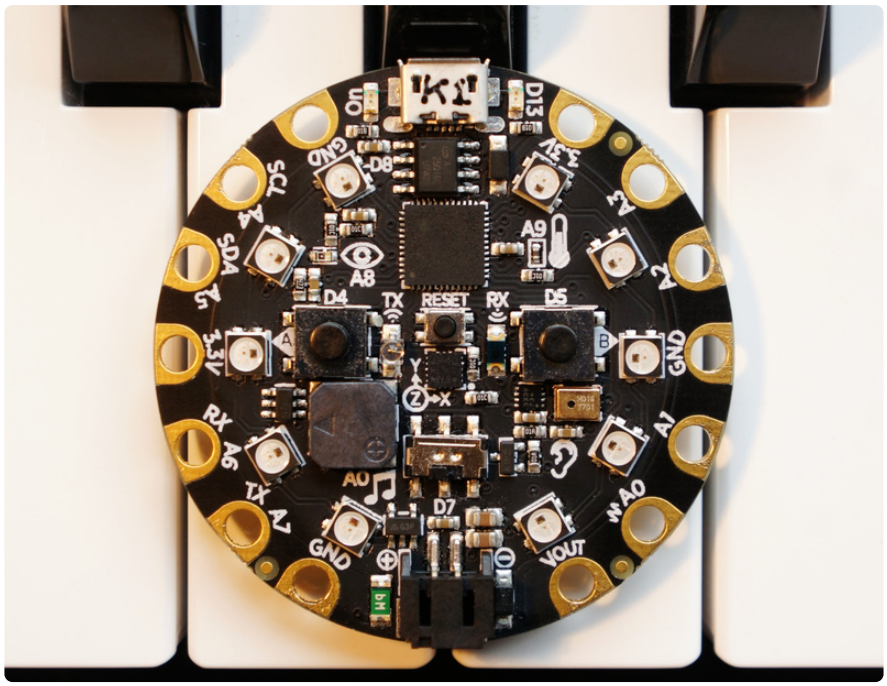
https://www.adafruit.com/product/1305

[Copper Foil Tape with Conductive Adhesive - 6mm x 15 meter roll]
Copper tape can be an interesting addition to your toolbox. The tape itself is made of thin pure copper so its extremely flexible and can take on nearly any shape. You can easily...
https://www.adafruit.com/product/1128

[Copper Foil Tape with Conductive Adhesive - 25mm x 15 meter roll]
Copper tape can be an interesting addition to your toolbox. The tape itself is made of thin pure copper so its extremely flexible and can take on nearly any shape. You can easily...
https://www.adafruit.com/product/1127

[Pimoroni MINI.MU Glove Kit - without micro:bit]
The MINI.MU is MI.MU's make-it-yourself musical glove for young...
https://www.adafruit.com/product/4141

MIDI Controller



Download the `cpx-expressive-midi-controller.py` file with the link below. Plug your Circuit Playground Express (CPX) into your computer via a known-good USB data cable. A flash drive named CIRCUITPY should appear in your file explorer/finder program. Copy `cpx-expressive-midi-controller.py` to the **CIRCUITPY** drive, renaming it code.py (https://adafru.it/EL3).

Scroll past the code below for two videos showing the CPX controlling some synthesizers and a discussion on selected parts of the program.

```
# SPDX-FileCopyrightText: 2019 Kevin J. Walters for Adafruit Industries
#
# SPDX-License-Identifier: MIT

### cpx-expressive-midi-controller v1.2
### CircuitPython (on CPX) MIDI controller using the seven touch pads
### and accelerometer for modulation (cc1) and pitch bend
### Left button adjusts octave (switch left) or semitone (switch right)
### Right button adjusts scale, major or chromatic
### Switch right also disables pitch bend and modulation

### Tested with CPX and CircuitPython and 4.0.0-beta.5

### Needs recent adafruit_midi module

### copy this file to CPX as code.py

### MIT License.

### Copyright (c) 2019 Kevin J. Walters

### Permission is hereby granted, free of charge, to any person obtaining a copy
### of this software and associated documentation files (the "Software"), to deal
### in the Software without restriction, including without limitation the rights
```

```
### to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
### copies of the Software, and to permit persons to whom the Software is
### furnished to do so, subject to the following conditions:

### The above copyright notice and this permission notice shall be included in all
### copies or substantial portions of the Software.

### THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
### IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
### FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
### AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
### LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
### OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
### SOFTWARE.

import time

import digitalio
import touchio
import busio
import board
import usb_midi
import neopixel

import adafruit_lis3dh
import adafruit_midi

from adafruit_midi.note_on          import NoteOn
from adafruit_midi.control_change   import ControlChange
from adafruit_midi.pitch_bend       import PitchBend

# MIDI defines middle C as 60 and modulation wheel is cc 1 by convention
midi_note_C4 = 60
midi_cc_modwheel = 1  # was const(1)

# 0x19 is the i2c address of the onboard accelerometer
acc_i2c = busio.I2C(board.ACCELEROMETER_SCL, board.ACCELEROMETER_SDA)
acc_int1 = digitalio.DigitalInOut(board.ACCELEROMETER_INTERRUPT)
acc = adafruit_lis3dh.LIS3DH_I2C(acc_i2c, address=0x19, int1=acc_int1)
acc.range = adafruit_lis3dh.RANGE_2_G
acc.data_rate = adafruit_lis3dh.DATARATE_10_HZ

# brightness 1.0 saves memory by removing need for a second buffer
# 10 is number of NeoPixels on CPX
numpixels = 10  # was const(10)
pixels = neopixel.NeoPixel(board.NEOPIXEL, numpixels, brightness=1.0)

# Turn NeoPixel on to represent a note using RGB x 10
# to represent 30 notes - doesn't do anything with pitch bend
def noteLED(pix, pnote, pvel):
    note30 = (pnote - midi_note_C4) % (3 * numpixels)
    pos = note30 % numpixels
    r, g, b = pix[pos]
    if pvel == 0:
        brightness = 0
    else:
        # max brightness will be 32
        brightness = round(pvel / 127 * 30 + 2)
    # Pick R/G/B based on range within the 30 notes
    if note30 < 10:
        r = brightness
    elif note30 < 20:
        g = brightness
    else:
        b = brightness
    pix[pos] = (r, g, b)

# white pulse used to indicate octave changes
flashbrightness = 20
```

```
def flashLED(pix, position):
    pos = position % numpixels
    t1 = time.monotonic()
    oldcolour = pix[pos]
    while time.monotonic() - t1 < 0.25:
        for i in range(0, flashbrightness, 2):
            pix[pos] = (i, i, i)
        for i in range(flashbrightness, 0, -2):
            pix[pos] = (i, i, i)
    pix[pos] = oldcolour

midi_channel = 1
midi = adafruit_midi.MIDI(midi_out=usb_midi.ports[1],
                          out_channel=midi_channel-1)

# CPX counter-clockwise order of touch capable pads (i.e. not A0)
pads = [board.A4,
        board.A5,
        board.A6,
        board.A7,
        board.A1,
        board.A2,
        board.A3]

# The touch pads calibrate themselves as they are created, just once here
touchpads = [touchio.TouchIn(pad) for pad in pads]
del pads  # done with that

pb_midpoint = 8192
pitch_bend_value = pb_midpoint  # mid point - no bend
min_pb_change = 250

mod_wheel = 0
min_mod_change = 5

# button A is on left (usb at top)
button_left = digitalio.DigitalInOut(board.BUTTON_A)
button_left.switch_to_input(pull=digitalio.Pull.DOWN)
button_right = digitalio.DigitalInOut(board.BUTTON_B)
button_right.switch_to_input(pull=digitalio.Pull.DOWN)
switch_left = digitalio.DigitalInOut(board.SLIDE_SWITCH)
switch_left.switch_to_input(pull=digitalio.Pull.UP)

# some example scales in semitones
scale_st = {"major": [0, 2, 4, 5, 7, 9, 11],
            "chromatic": [0, 1, 2, 3, 4, 5, 6]}
scales = ["major", "chromatic"]
scale_idx = 0
base_note = midi_note_C4  # C4 middle C

def make_scale(scale_name):
    return [semitone_offset + base_note
            for semitone_offset in scale_st[scale_name]]

midi_notes = make_scale(scales[scale_idx])
keydown = [False] * 7

velocity = 127
min_octave = -3
max_octave = +3
octave = 0
min_semitone = -11
max_semitone = +11
semitone = 0

# 1/10 = 10 Hz - review data_rate setting if this is changed
acc_read_t = time.monotonic()
acc_read_period = 1/10
# For accelerometer do nothing between 0 and 1.3 (ms-2)
```

```
acc_nullzone = 1.3
acc_range = 4.0

# Convert an accelerometer reading
# from min_msm2 to min_msm2+range to an int from 0 to value_range
# or return 0 or value_range outside those values
# The conversion is applied "symmetrically" to negative numbers
def scale_acc(acc_msm2, min_msm2, range_msm2, value_range):
    if acc_msm2 >= 0.0:
        sign_a_m = 1
        magn_acc_msm2 = acc_msm2
    else:
        sign_a_m = -1
        magn_acc_msm2 = abs(acc_msm2)

    adj_msm2 = magn_acc_msm2 - min_msm2

    # deal with out of bounds values else scale value
    # pylint: disable=no-else-return
    if adj_msm2 <= 0:
        return 0
    elif adj_msm2 >= range_msm2:
        return sign_a_m * value_range
    else:
        return sign_a_m * round(adj_msm2 / range_msm2 * value_range)

# Scan each pad and look for changes by comparing
# with keystate stored in keydown boolean list
# and send note on/off messages accordingly
# Send pitch bend and mod wheel cc based on tilt from accelerometer
# Change octave and semitone based on buttons
while True:
    for idx, touchpad in enumerate(touchpads):
        if touchpad.value != keydown[idx]:
            keydown[idx] = touchpad.value
            # 12 semitones in an octave
            note = midi_notes[idx] + octave * 12 + semitone
            if keydown[idx]:
                midi.send(NoteOn(note, velocity))
                noteLED(pixels, note, velocity)
            else:
                midi.send(NoteOn(note, 0))  # Using note on 0 for off
                noteLED(pixels, note, 0)

    # Perform rate limited checks on the accelerometer
    # if switch is to left
    now_t = time.monotonic()
    if switch_left.value and now_t - acc_read_t > acc_read_period:
        acc_read_t = time.monotonic()
        ax, ay, az = acc.acceleration

        # scale from 0 to 127 (maximum cc 7bit value)
        new_mod_wheel = abs(scale_acc(ay, acc_nullzone, acc_range, 127))
        if (abs(new_mod_wheel - mod_wheel) > min_mod_change
                or (new_mod_wheel == 0 and mod_wheel != 0)):
            midi.send(ControlChange(midi_cc_modwheel, new_mod_wheel))
            mod_wheel = new_mod_wheel

        # scale from 0 to +/- 8191 (almost maximum signed 14bit values)
        new_pitch_bend_value = (pb_midpoint
                                - scale_acc(ax, acc_nullzone, acc_range,
                                            pb_midpoint - 1))
        if (abs(new_pitch_bend_value - pitch_bend_value) > min_pb_change
                or (new_pitch_bend_value == pb_midpoint
                    and pitch_bend_value != pb_midpoint)):
            midi.send(PitchBend(new_pitch_bend_value))
            pitch_bend_value = new_pitch_bend_value

    # left button increase octave / semitones shift based on switch
```

```
    # does not currently clear playing notes (buglet)
    if button_left.value:
        if switch_left.value:
            octave += 1
            if octave > max_octave:
                octave = min_octave
            flashLED(pixels, octave)
        else:
            semitone += 1
            if semitone > max_semitone:
                semitone = min_semitone
            # semitone range is more than number of pixels!
            flashLED(pixels, semitone)

        while button_left.value:
            pass  # wait for button up

    # right button cycles through scales
    if button_right.value:
        scale_idx += 1
        if scale_idx >= len(scales):
            scale_idx = 0
        flashLED(pixels, scale_idx)
        midi_notes = make_scale(scales[scale_idx])

        while button_right.value:
            pass  # wait for button up
```

MIDI Controller Examples

The first video below shows the program running on a CPX board controlling Nikolay Tsenkov's Viktor NV-1 free software synthesizer (https://adafru.it/C-4). Note the movement of the pitch wheel and modulation wheel just left of the keyboard.

The second video uses the CPX to control the Gakken Pocket Miku (NSX-39) synthesizer. This synthesizer is normally used for its Vocaloid (https://adafru.it/EL4) functionality but here MIDI channel 2 is being used for its General MIDI (https://adafru.it/stE) implementation of a piano, part of its Yamaha XG (https://adafru.it/EL5) functionality.

Code Discussion

The main part of the program is a loop which sends certain MIDI messages based on various inputs, like:

- a change in touching of the touch pads which causes **note on** or **note off** messages to be sent,
- any significant change in accelerometer `x` and `y` values which sends **pitch bend change** or **change control** messages, these are limited by rate and inhibited by the switch in right position.

The loop also checks the button and switch values. These are used to change octave/semitone range and scale (between major and chromatic modes).

The program starts with a major scale at middle C. The touch pad at the top left of CPX will initially send a C_4 (MIDI note 60) and next one (counterclockwise) will send a D_4 (MIDI note 62) and so on. The code excerpt below shows this code, the MIDI note values for each touch pad are stored in `midi_notes[idx]`, these are based on the middle C value plus offset per scale. This is further adjusted by the current `octave` and `semitone` offset selected - a comment informs us of the significance of the "magic value (https://adafru.it/EL6)" of 12. The semitone offset can be used to change the key or in chromatic scale to offset a second CPX by +7 semitones to allow two CPXs with 14 notes to cover just over a full octave.

```
for idx, touchpad in enumerate(touchpads):
    if touchpad.value != keydown[idx]:
        keydown[idx] = touchpad.value
        # 12 semitones in an octave
        note = midi_notes[idx] + octave * 12 + semitone
        if keydown[idx]:
            midi.send(NoteOn(note, velocity))
            noteLED(pixels, note, velocity)
        else:
            midi.send(NoteOn(note, 0))  # Using note on 0 for off
            noteLED(pixels, note, 0)
```

The `keydown` list is tracking the previous state of the touchpad. This is needed to avoid sending unnecessary duplicate MIDI messages if nothing has changed.

One surprise may be the use of **note on** message for the user lifting their finger from the touch pad. When velocity is set to 0 this becomes equivalent to a **note off** for most MIDI devices. This is a useful memory economisation on an M0 processor-based board as each message is a separate python class and these consume memory when they are `import`'ed. The technique for importing the library with only the messages needed is shown below.

```
import adafruit_midi

from adafruit_midi.note_on          import NoteOn
from adafruit_midi.control_change   import ControlChange
from adafruit_midi.pitch_bend       import PitchBend
```

When a button is used to change a value the new value is shown by briefly flashing a NeoPixel white a few times with the `flashLED()` function and then restoring the NeoPixel to its previous value.

```
# white pulse used to indicate octave changes
flashbrightness = 20
def flashLED(pix, position):
```

```
    pos = position % numpixels
    t1 = time.monotonic()
    oldcolour = pix[pos]
    while time.monotonic() - t1 &lt; 0.25:
        for i in range(0, flashbrightness, 2):
            pix[pos] = (i, i, i)
        for i in range(flashbrightness, 0, -2):
            pix[pos] = (i, i, i)
    pix[pos] = oldcolour
```

This code works but it's worth discussing some minor issues with it. The `while` loop is running for a quarter (`0.25`) of a second, this would be more flexible if it was a function argument with a default value. The same could be said for the global variable `flashbrightness`. This would also make it easier to test.

Perhaps more importantly, inside the `while` loop there are two `for` loops which ramp the brightness of `pix[pos]` up and then down. These are not constrained by time but happen to execute at a desirable rate on the CPX board. This means the flashing rate of the NeoPixel is subject to the performance of the CircuitPython interpreter, the `neopixel` library and the processor - changes to any of those could alter the flash rate. A board using the faster M4 processor will inevitably make this flash much faster.

The code which sends the **control change** message is shown below. The `scale_acc()` function (https://adafru.it/EL7) is taking the accelerometer's `ay` (in ms-2 (https://adafru.it/EL8)) value and turning it into an integer value between 0 and 127. The `acc_nullzone` variable (set to `1.3`) keeps the value at 0 even when the board isn't quite flat or is gently nudged. The `acc_range` variable (set to `4.0`) determines the end of the range. i.e. an `ay` value of `acc_nullzone + acc_range` will return 127. It might be more natural to calculate and use the angle of the board instead of `ay` , but with the current ranges the movement feels appropriate for the modulation output. The use of `abs()` on the `scale_acc()` return value makes forward and backward tilt equivalent.

```
# scale from 0 to 127 (maximum cc 7bit value)
new_mod_wheel = abs(scale_acc(ay, acc_nullzone, acc_range, 127))
if (abs(new_mod_wheel - mod_wheel) &gt; min_mod_change
        or (new_mod_wheel == 0 and mod_wheel != 0)):
    midi.send(ControlChange(midi_cc_modwheel, new_mod_wheel))
    mod_wheel = new_mod_wheel
```

The `if` statement is determining whether the value has changed by a significant amount since the last value was sent. The value of 0 will always be sent to allow the modulation wheel to return to exactly 0 which often equates to no modulation.

The checks and message sending code for **control change** and **pitch bend change** messages are wrapped in an `if` statement to ensure they are only sent at a

maximum frequency, currently fixed at 10Hz. This keeps the MIDI message rate to a low rate suitable for all devices. This could be increased or made controllable but care would be needed to ensure the accelerometer readings are not noisy and give smooth variations.

The code for switch and button handling is getting a little lengthy and making the contents of the main `while` loop rather large. This can be a bit of a trap as further small additions accumulate and the code can get larger and larger. This can lead to a segment of code that's difficult to understand and more likely to be or become buggy. Moving some of the code into one or more functions is likely to make the code less unwieldy and more maintainable (https://adafru.it/EL9).

Capacitive Touch

The capacitive touch pads calibrate themselves when the program starts. The code may need to be restarted if the CPX board is moved to a different surface or if an external capactive keyboard is added.

From CircuitPython Cap Touch (https://adafru.it/DOe):

> If you get too many touch responses or not enough, reload your code through the serial console or eject the board and tap the reset button!

It's interesting to note that capacitive keyboards are far from a new invention. From the book Analog Days (https://adafru.it/ELa), referring to early 1960s Buchla 100 Series (https://adafru.it/ELb):

> "They [the ports] were all capacitance-sensitive touch-plates, or resistance-sensitive in some cases, organized in various sorts of array.
>
> "I saw no reason to borrow from a keyboard, which is a device invented to throw hammers at strings, later on, for operating switches for electronic organs and so-on."
>
> Don Buchla (https://adafru.it/ELc)

MIDI routing

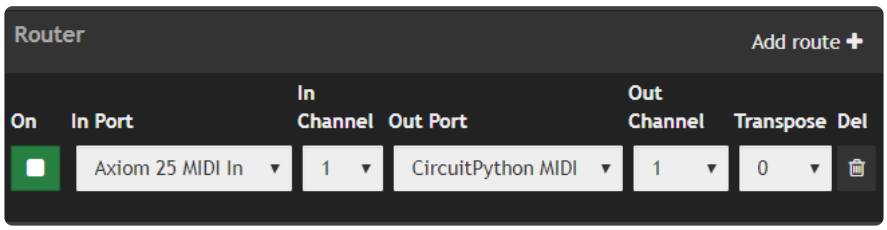
MIDI messages between different USB devices are not automatically forwarded by the operating system. An application is required to forward or route the messages.

Tommy van Leeuwen has written a very useful Web MIDI Sequencer, Router & Drum Machine (https://adafru.it/ELd) application which can be used for this.

Web MIDI (https://adafru.it/ELe) allows browser-based applications to communicate with MIDI devices. Web MIDI is only implemented (https://adafru.it/DcJ) in some browsers, for example Chrome and Opera.

The example in the video above shows the `CircuitPython MIDI` 's channel **1** being sent to the `NSX-39` 's channel **2**.

The example in the screenshot below shows an `Axiom 25 MIDI In` 's channel **1** being sent to the `CircuitPython MIDI` 's channel **1**.



Synthesizers

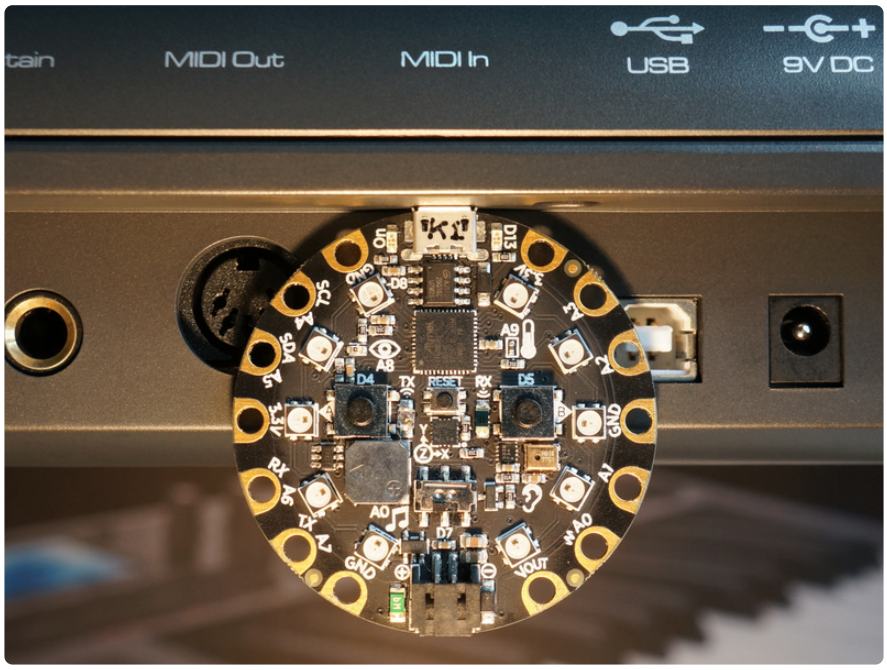
Everyone needs a synthesizer! If you don't have an EMS Synthi A at hand (picture below), there's a useful list of software synthesizers on Trellis M4 Expressive MIDI Controller guide (https://adafru.it/ELf) including some free ones.

MuTools MuLab (https://adafru.it/ELg) is another digital audio workstation (DAW) (https://adafru.it/ELh) which can be used in a restricted mode for free.

Plogue chipsounds (https://adafru.it/ELi) is a VST plugin (https://adafru.it/ELj) and standalone synthesizer with meticulous recreations of 1970s and 1980s era sound chips. It can be used for 4 minutes per session for free.

The next page has a program for a basic synthesizer for the CPX.

Basic Synthesizer



Download the `cpx-basic-synth.py` file with the link below and copy it to the **CIRCUITPY** drive renaming it code.py (https://adafru.it/EL3).

Scroll past the code below for a discussion on selected parts of the program and a video showing the CPX synthesizer being controlled by another CPX.

```
# SPDX-FileCopyrightText: 2019 Kevin J. Walters for Adafruit Industries
#
# SPDX-License-Identifier: MIT
```

```
### cpx-basic-synth v1.4
### CircuitPython (on CPX) synth module using internal speaker
### Velocity sensitive monophonic synth
### with crude amplitude modulation (cc1) and choppy pitch bend

### Tested with CPX and CircuitPython and 4.0.0-beta.7

### Needs recent adafruit_midi module

### copy this file to CPX as code.py

### MIT License.

### Copyright (c) 2019 Kevin J. Walters

### Permission is hereby granted, free of charge, to any person obtaining a copy
### of this software and associated documentation files (the "Software"), to deal
### in the Software without restriction, including without limitation the rights
### to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
### copies of the Software, and to permit persons to whom the Software is
### furnished to do so, subject to the following conditions:

### The above copyright notice and this permission notice shall be included in all
### copies or substantial portions of the Software.

### THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
### IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
### FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
### AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
### LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
### OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
### SOFTWARE.

import array
import time
import math

import digitalio
import audioio
import audiocore
import board
import usb_midi
import neopixel

import adafruit_midi

from adafruit_midi.midi_message     import note_parser

from adafruit_midi.note_on          import NoteOn
from adafruit_midi.note_off         import NoteOff
from adafruit_midi.control_change   import ControlChange
from adafruit_midi.pitch_bend       import PitchBend

# Turn the speaker on
speaker_enable = digitalio.DigitalInOut(board.SPEAKER_ENABLE)
speaker_enable.direction = digitalio.Direction.OUTPUT
speaker_on = True
speaker_enable.value = speaker_on

dac = audioio.AudioOut(board.SPEAKER)

# 440Hz is the standard frequency for A4 (A above middle C)
# MIDI defines middle C as 60 and modulation wheel is cc 1 by convention
A4refhz = 440  # was const(440)
midi_note_C4 = note_parser("C4")
midi_note_A4 = note_parser("A4")
midi_cc_modwheel = 1  # was const(1)
twopi = 2 * math.pi
```

```
# A length of 12 will make the sawtooth rather steppy
sample_len = 12
base_sample_rate = A4refhz * sample_len
max_sample_rate = 350000  # a CPX / M0 DAC limitation

midpoint = 32768

# A sawtooth function like math.sin(angle)
# 0 returns 1.0, pi returns 0.0, 2*pi returns -1.0
def sawtooth(angle):
    return 1.0 - angle % twopi / twopi * 2

# make a sawtooth wave between +/- each value in volumes
# phase shifted so it starts and ends near midpoint
# "H" arrays for RawSample looks more memory efficient
# see https://forums.adafruit.com/viewtopic.php?f=60&t=150894
def waveform_sawtooth(length, waves, volumes):
    for vol in volumes:
        waveraw = array.array("H",
                              [midpoint +
                               round(vol * sawtooth((idx + 0.5) / length
                                                    * twopi
                                                    + math.pi))
                               for idx in list(range(length))])
        waves.append((audiocore.RawSample(waveraw), waveraw))

# Make some square waves of different volumes volumes, generated with
# n=10;[round(math.sqrt(x)/n*32767*n/math.sqrt(n)) for x in range(1, n+1)]
# square root is for mapping velocity to power rather than signal amplitude
# n=15 throws MemoryError exceptions when a note is played :(
waveform_by_vol = []
waveform_sawtooth(sample_len,
                  waveform_by_vol,
                  [10362, 14654, 17947, 20724, 23170,
                   25381, 27415, 29308, 31086, 32767])

# brightness 1.0 saves memory by removing need for a second buffer
# 10 is number of NeoPixels on CPX
numpixels = 10  # was const(10)
pixels = neopixel.NeoPixel(board.NEOPIXEL, numpixels, brightness=1.0)

# Turn NeoPixel on to represent a note using RGB x 10
# to represent 30 notes - doesn't do anything with pitch bend
def noteLED(pix, pnote, pvel):
    note30 = (pnote - midi_note_C4) % (3 * numpixels)
    pos = note30 % numpixels
    r, g, b = pix[pos]
    if pvel == 0:
        brightness = 0
    else:
        # max brightness will be 32
        brightness = round(pvel / 127 * 30 + 2)
    # Pick R/G/B based on range within the 30 notes
    if note30 < 10:
        r = brightness
    elif note30 < 20:
        g = brightness
    else:
        b = brightness
    pix[pos] = (r, g, b)

# Calculate the note frequency from the midi_note with pitch bend
# of pb_st (float) semitones
# Returns float
def note_frequency(midi_note, pb_st):
    # 12 semitones in an octave
    return A4refhz * math.pow(2, (midi_note - midi_note_A4 + pb_st) / 12.0)
```

```
midi_channel = 1
midi = adafruit_midi.MIDI(midi_in=usb_midi.ports[0],
                          in_channel=midi_channel-1)

# pitchbendrange in semitones - often 2 or 12
pb_midpoint = 8192
pitch_bend_multiplier = 2 / pb_midpoint
pitch_bend_value = pb_midpoint  # mid point - no bend

wave = []  # current or last wave played
last_note = None

# Amplitude modulation frequency in Hz
am_freq = 16
mod_wheel = 0

# Read any incoming MIDI messages (events) over USB
# looking for note on, note off, pitch bend change
# or control change for control 1 (modulation wheel)
# Apply crude amplitude modulation using speaker enable
while True:
    msg = midi.receive()
    if isinstance(msg, NoteOn) and msg.velocity != 0:
        last_note = msg.note
        # Calculate the sample rate to give the wave form the frequency
        # which matches the midi note with any pitch bending applied
        pitch_bend = (pitch_bend_value - pb_midpoint) * pitch_bend_multiplier
        note_freq = note_frequency(msg.note, pitch_bend)
        note_sample_rate = round(base_sample_rate * note_freq / A4refhz)

        # Select the wave with volume for the note velocity
        # Value slightly above 127 together with int() maps the velocities
        # to equal intervals and avoids going out of bound
        wave_vol = int(msg.velocity / 127.01 * len(waveform_by_vol))
        wave = waveform_by_vol[wave_vol]

        if note_sample_rate > max_sample_rate:
            note_sample_rate = max_sample_rate
        wave[0].sample_rate = note_sample_rate  # must be integer
        dac.play(wave[0], loop=True)

        noteLED(pixels, msg.note, msg.velocity)

    elif (isinstance(msg, NoteOff) or
          isinstance(msg, NoteOn) and msg.velocity == 0):
        # Our monophonic "synth module" needs to ignore keys that lifted on
        # overlapping presses
        if msg.note == last_note:
            dac.stop()
            last_note = None

        noteLED(pixels, msg.note, 0)  # turn off NeoPixel

    elif isinstance(msg, PitchBend):
        pitch_bend_value = msg.pitch_bend  # 0 to 16383
        if last_note is not None:
            pitch_bend = (pitch_bend_value - pb_midpoint) * pitch_bend_multiplier
            note_freq = note_frequency(last_note, pitch_bend)
            note_sample_rate = round(base_sample_rate * note_freq / A4refhz)
            if note_sample_rate > max_sample_rate:
                note_sample_rate = max_sample_rate
            wave[0].sample_rate = note_sample_rate  # must be integer
            dac.play(wave[0], loop=True)

    elif isinstance(msg, ControlChange):
        if msg.control == midi_cc_modwheel:
            mod_wheel = msg.value  # msg.value is 0 (none) to 127 (max)

    if mod_wheel > 0:
```

```
        t1 = time.monotonic() * am_freq
        # Calculate a form of duty_cycle for enabling speaker for crude
        # amplitude modulation. Empirically the divisor needs to greater
        # than 127 as can't hear much when speaker is off more than half
        # 220 works reasonably well
        new_speaker_on = (t1 - int(t1)) > (mod_wheel / 220)
    else:
        new_speaker_on = True

    if speaker_on != new_speaker_on:
        speaker_enable.value = new_speaker_on
        speaker_on = new_speaker_on
```

Basic Synthesizer Example

The video below shows one CPX running the MIDI Controller program from the previous page and the second running the synthesizer code on this page. Tilting to the left (or right) bends the pitch and tilting up or down increases the mod wheel.

The video shows the limitations of the approach for pitch bending. There's a very perceivable delay when the sample is played at a new sample rate to change the frequency. This gives the pitch bend a certain unpleasant choppiness and means it can lag behind a rapid burst of **pitch bend change** messages. Attempts at portamento (https://adafru.it/ELk) are going to sound like a low quality glissando (https://adafru.it/ELl)!

A compiled, mid-level language like C is less easy to use than Python but can offer faster, more predictable performance and often allows more direct control over the audio hardware. The Soulsby miniATMEGATRON (https://adafru.it/ELm) shows what can be done with C code using an Arduino Uno based on the slower ATmega328P processor and an external low-pass filter (https://adafru.it/ELn).

Code Discussion

The main part of the program is a loop checking for any incoming MIDI messages and acting on these. At the end of the loop it applies an innovative form of crude amplitude modulation by rapidly disabling/enabling the speaker. The MIDI message actions are:

- **note on -** plays a pre-constructed sample at a sample rate which matches the note's pitch and a volume which approximates the note's velocity, note pitch is shown on the NeoPixels,
- **note off -** ends playing of the current sample and turns off the NeoPixel,
- **pitch bend change** - if a note is playing then the sample rate is adjusted for the new value of pitch bend,

- **control change** for cc1 - this represents the mod wheel and is used to control the duty cycle of the amplitude modulation for speaker output.

The sample used to play the note is a short list of values representing a sawtooth wave. The short length makes it low resolution and gives it a "steppy" appearance, see the next page for an in-depth look at waveforms. This single cycle wave is generated for a small range of volumes before the program's main loop with the code below. This pre-calculation uses a bit more memory but minimises the time it takes (latency) to start playing a note.

```
def waveform_sawtooth(length, waves, volumes):
    for vol in volumes:
        waveraw = array.array("H",
                              [midpoint +
                               round(vol * sawtooth((idx + 0.5) / length
                                                    * twopi
                                                    + math.pi))
                               for idx in list(range(length))])
    waves.append((audioio.RawSample(waveraw), waveraw))

waveform_by_vol = []
waveform_sawtooth(sample_len,
                  waveform_by_vol,
                  [10362, 14654, 17947, 20724, 23170,
                   25381, 27415, 29308, 31086, 32767])
```

The list of numbers representing volumes has been generated externally with:

```
n=10
[round(math.sqrt(x) / math.sqrt(n) * 32767)
 for x in range(1, n + 1)]
```

The numbers are not evenly spaced because they represent the maximum amplitude of the wave which is output as a voltage. If the voltage is doubled, then broadly speaking the current will double and since power is the product of these two quantities (https://adafru.it/CQa) the power will quadruple. This explains the role of the `math.sqrt()` to provide amplitudes which represent a linear increase in power. These will correspond to an increase in volume (sound pressure level (https://adafru.it/ELo)) when the velocity of a note is mapped linearly to a list element.

The array (https://adafru.it/ELp) type used for samples is a compact representation for numbers. "H" (https://adafru.it/ELq) selects unsigned 16 bit integers which is the representation most closely matching the native DAC values (https://adafru.it/ELr) - this helps with memory efficiency.

The NeoPixels are used to show the note playing with the `noteLED()` function shown below.

```
# Turn NeoPixel on to represent a note using RGB x 10
# to represent 30 notes - doesn't do anything with pitch bend
def noteLED(pix, pnote, pvel):
    note30 = (pnote - midi_note_C4) % (3 * numpixels)
    pos = note30 % numpixels
    r, g, b = pix[pos]
    if pvel == 0:
        brightness = 0
    else:
        # max brightness will be 32
        brightness = round(pvel / 127 * 30 + 2)
    # Pick R/G/B based on range within the 30 notes
    if note30 &lt; 10:
        r = brightness
    elif note30 &lt; 20:
        g = brightness
    else:
        b = brightness
pix[pos] = (r, g, b)
```

Modular division (https://adafru.it/ELs) is used to map the note to 30 values, the first ten notes will be red starting at middle C (C_4), the next ten green, the next ten blue and this then repeats both above and below that range. Multiple values will be merged, e.g. C_4 and $A\#_4$ keys pressed together will show as yellow (https://adafru.it/ELt).

The NeoPixels have been created with `brightness=1.0`. For the current library implementation this makes updates faster and uses less memory. This explains why the maximum value based on the note's velocity is set to just 32 and not the maximum value, 255.

The same `noteLED()` is used in the MIDI controller. The code could be kept in a separate file and that could be `import`'ed. This would make the code easier to maintain, promote reusability (https://adafru.it/ELu) and is a step towards creating a library (https://adafru.it/ELv).

At the end of the main loop is the code that applies amplitude modulation to the speaker output (only).

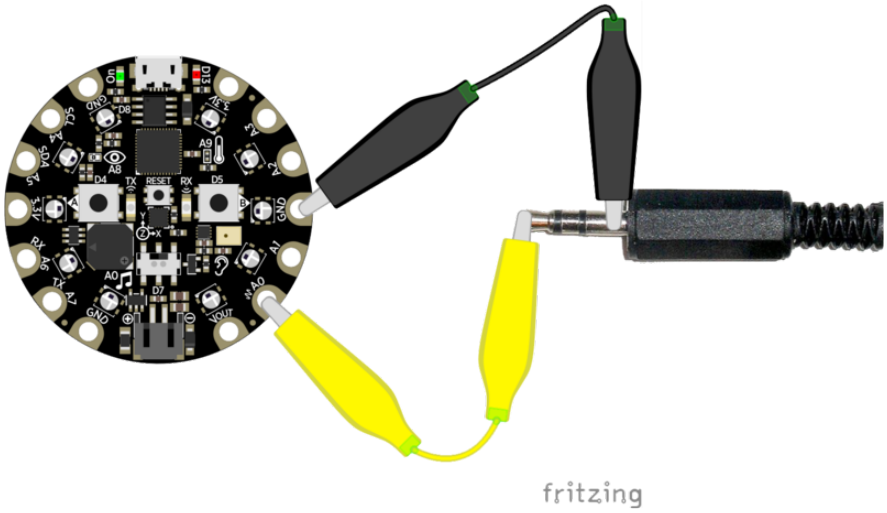
```
if mod_wheel &gt; 0:
        t1 = time.monotonic() * am_freq
        # Calculate a form of duty_cycle for enabling speaker for crude
        # amplitude modulation. Empirically the divisor needs to greater
        # than 127 as can't hear much when speaker is off more than half
        # 220 works reasonably well
        new_speaker_on = (t1 - int(t1)) &gt; (mod_wheel / 220)
    else:
        new_speaker_on = True
```

Any MIDI **control change** values received for cc1 are stored in the `mod_wheel` variable. For non-zero values this is used to control when the speaker is on or off. The

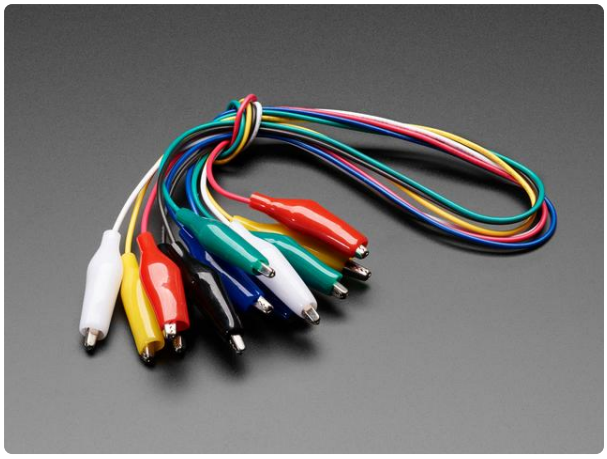
time measurement is used to set a regular period of on and off with `am_freq` set to a fixed value of 16 (Hz). The `mod_wheel` value then determines the duration of the off part. In synthesis terminology, this would be referred to as a free-running, square wave LFO (https://adafru.it/ELw) modulating the amplitude with maximum depth with the mod wheel determining the duty cycle.

External Audio

The CPX can be connected from touchpad **A0** to an amplifier or headphones but cannot drive a loudspeaker (https://adafru.it/ELx) without amplification. Using amplified speakers or a pair of headphones, connect your Circuit Playground Express as shown below. The speakers from Adafruit have a volume knob for easy adjustment. You can use a cell phone wall charger, computer, or cell phone external battery to power the speakers (and the Circuit Playground Express).



USB Powered Speakers
Add some extra boom to your audio project with these powered loudspeakers. We sampled half a dozen different models to find ones with a good frequency response, so you'll get...
https://www.adafruit.com/product/1363



Small Alligator Clip Test Lead (set of 6)
Connect this to that without soldering using these small alligator clip test leads. 18" long cables with color-coded alligator clips on both ends. You get 6 pieces in 6...
https://www.adafruit.com/product/4100

USB Battery Pack for Raspberry Pi - 10000mAh - 2 x 5V outputs
A large-sized rechargeable battery pack for your Raspberry Pi (or Arduino, or
https://www.adafruit.com/product/1566