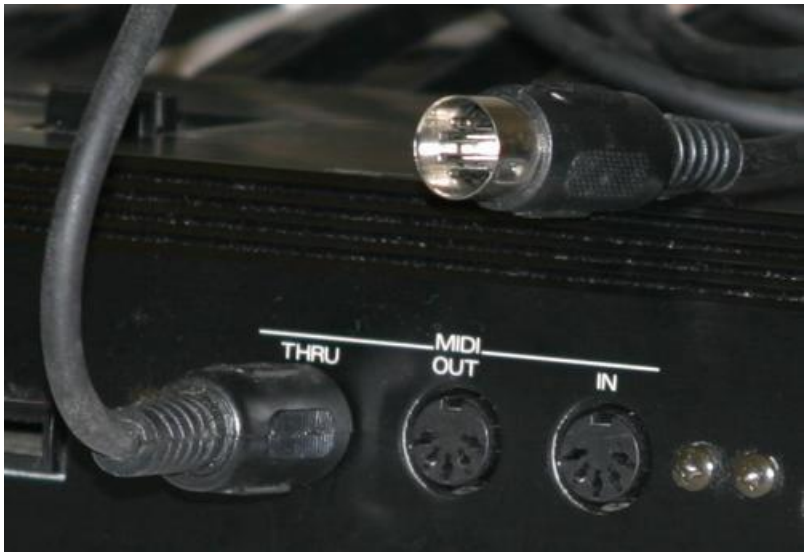
adafruit learning system

NeoTrellis M4 MIDI File Synthesizer

Created by Dave Astels



Last updated on 2019-03-27 03:15:21 PM UTC

Overview

There are guides for using the NeoTrellis M4 as a MIDI controller (https://adafru.it/D7S), beat sequencers (https://adafru.it/D4z), and sound boards (Star Trek (https://adafru.it/D7U), Christmas (https://adafru.it/D8C), and D&D).

This guide turns a NeoTrellis into a MIDI file synthesizer: **it takes a set of instrument samples (a clip for each note) and a MIDI file and plays the MIDI file using the samples.**

Parts

Your browser does not support the video tag.

Adafruit NeoTrellis M4 with Enclosure and Buttons Kit Pack

$59.95
IN STOCK

ADD TO CART

USB cable - USB A to Micro-B

$2.95
OUT OF STOCK

OUT OF STOCK

USB Powered Speakers

$9.95
IN STOCK

ADD TO CART

5V 2.5A Switching Power Supply with 20AWG MicroUSB Cable

$7.50
IN STOCK

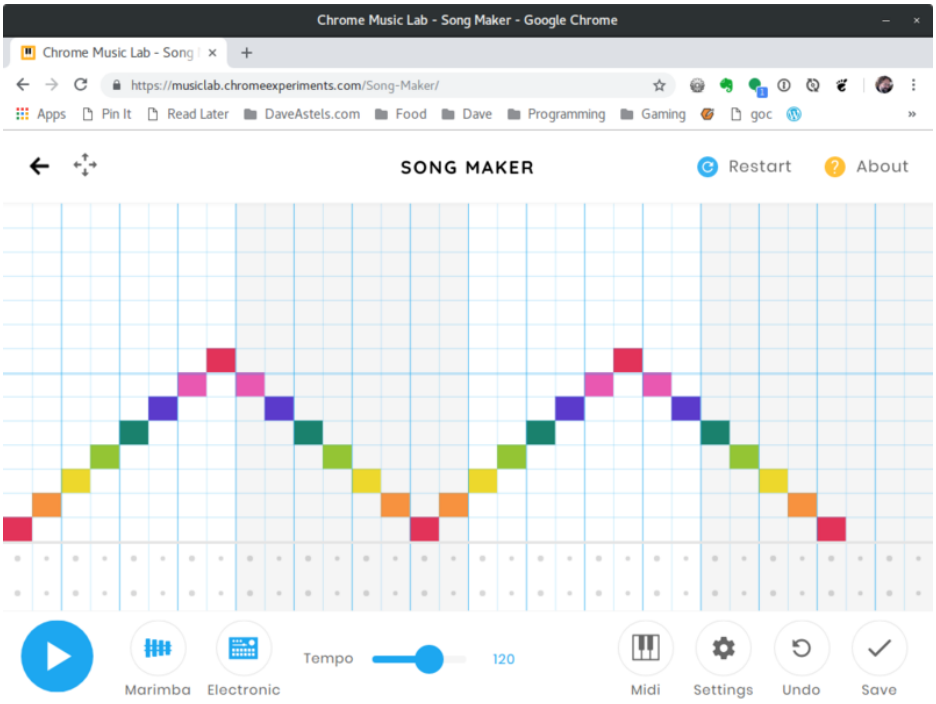
ADD TO CART

Thumbnail by Wikipedia user :en:Pretzelpaws (https://adafru.it/Df0) under GNU free documentation license

Files

MIDI Files

There are various ways to create MIDI files. One easy way to make simple ones is Song Maker from the Chrome Music Lab (https://adafru.it/Df1). It provides various ways to enter notes, including clicking on the sequencer grid to set notes. You can also play what you've entered. When you're satisfied you can save the MIDI file to your local drive.



Once you have some MIDI files, rename them something recognizable and put them onto your NeoTrellis. With your computer, plug the NeoTrellis in via a USB A to microB cable. In your file explorer/finder, you should see a new flash drive appear named **CIRCUITPY**. Create a new subdirectory on **CIRCUITPY** named **midi** and copy your MIDI files into the **midi** directory on your NeoTrellis **CIRCUITPY** drive. They'll be available in alphabetical order for selection so prefixing with two digit numbers gives you control of the ordering.

Samples

The code uses samples for each note. That means that for each instrument, you want to have available, you need to set up samples of it. There are various places to find instrument samples that can be used.
Basegorllia.com (https://adafru.it/Df2) has an extensive selection of free samples. The samples included here are taken from the 900+ Classic Synths Samples (https://adafru.it/Df3) package.

When you find one (or several) instruments you like, you will need to convert them to a form usable on the NeoTrellis. The samples used in this project are mono (not stereo) WAV files with a sample rate of 16000, at 16 bits per sample PCM. There's a guide on converting sound files (https://adafru.it/BvU).

All sample related files are stored in the **samples** directory on the **CIRCUITPY** drive. There is a file for each instrument that maps MIDI key codes to sample files. For organizational purposes, it's handy to put each set of samples in its own directory. For example, there is a set of samples from an Arp Odyssey synthesizer stored in the directory **samples/Arp**, with an associated file **samples/arp.txt**. See the above image for an example. This file must be named in all lowercase; this is a convention enforced by the code. This file is shown below.

```
36,Arp/Arp_C2.WAV
38,Arp/Arp_D2.WAV
40,Arp/Arp_E2.WAV
41,Arp/Arp_F2.WAV
43,Arp/Arp_G2.WAV
45,Arp/Arp_A2.WAV
47,Arp/Arp_B2.WAV
48,Arp/Arp_C3.WAV
50,Arp/Arp_D3.WAV
52,Arp/Arp_E3.WAV
53,Arp/Arp_F3.WAV
55,Arp/Arp_G3.WAV
57,Arp/Arp_A3.WAV
59,Arp/Arp_B3.WAV
60,Arp/Arp_C4.WAV
62,Arp/Arp_D4.WAV
64,Arp/Arp_E4.WAV
65,Arp/Arp_F4.WAV
67,Arp/Arp_G4.WAV
69,Arp/Arp_A4.WAV
71,Arp/Arp_B4.WAV
72,Arp/Arp_C5.WAV
74,Arp/Arp_D5.WAV
76,Arp/Arp_E5.WAV
77,Arp/Arp_F5.WAV
79,Arp/Arp_G5.WAV
81,Arp/Arp_A5.WAV
83,Arp/Arp_B5.WAV
84,Arp/Arp_C6.WAV
```

Example samples and midi directories can be downloaded with the following link.

https://adafru.it/Dj5

https://adafru.it/Dj5

Code

We'll be using CircuitPython for this project. Are you new to using CircuitPython? No worries, there is a full getting started guide here (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).

There's a guide to get you up and running with CircuitPython specifically for the NeoTrellis M4 (https://adafru.it/C-O). You should read it before starting to get the most recent CircuitPython build for the NeoTrellisM4 installed and running along with the required libraries.

Full code with links to github is on the Downloads page.

Navigating the NeoTrellis

To get your NeoTrellis M4 set up to run this project's code, first follow these steps:

1) Update the bootloader for NeoTrellis (https://adafru.it/C-N)from the NeoTrellis M4 guide

2) Install the latest CircuitPython for NeoTrellis (https://adafru.it/C-O)from the NeoTrellis M4 guide

3) Get the latest CircuitPython library pack (https://adafru.it/zB-), the version number of which should match your version of CircuitPython (4.x or greater), unzip it, and drag the libraries you need over into the **/lib** folder on **CIRCUITPY**. The latest library package includes support for NeoTrellis.
https://github.com/adafruit/Adafruit_CircuitPython_Bundle/releases/ (https://adafru.it/zB-)

For this project you will need the following libraries:

- **adafruit_trellism4.mpy**
- **neopixel.mpy**
- **adafruit_matrixkeypad.mpy**

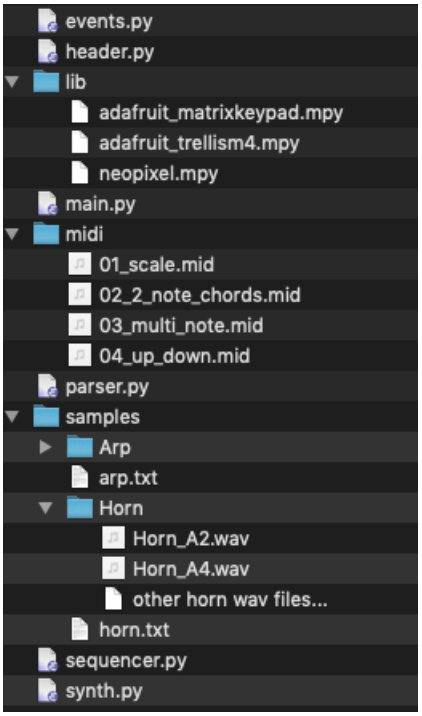
Overall Structure

The codebase is broken into several classes/files:

- **main.py** sets everything up and contains the main loop the checks for input from the user and acts accordingly
- **parser.py** contains the MIDI file parser
- **header.py** contains a class implementing the MIDI file header structure
- **events.py** contains several classes representing the various events that can be present in the MIDI files
- **sequencer.py** contains the sequencer which takes a MIDI track (a temporally ordered list of events) and executes them
- **synth.py** contains the Synthesizer class that plays samples as required

Click Download Project Zip below to get the full project with all the files

Below is a snapshot of what **CIRCUITPY** should look like.




```
"""
NeoTrellis M4 Express MIDI synth

Adafruit invests time and resources providing this open source code.
Please support Adafruit and open source hardware by purchasing
products from Adafruit!

Written by Dave Astels for Adafruit Industries
Copyright (c) 2018 Adafruit Industries
Licensed under the MIT license.
```

```
All text above must be included in any redistribution.
"""

import os
import parser
import sequencer
import synth
import adafruit_trellism4

trellis = adafruit_trellism4.TrellisM4Express(rotation=0)
trellis.pixels.brightness = 0.1
trellis.pixels.fill(0)

syn = synth.Synth()
seq = sequencer.Sequencer(syn)
p = parser.MidiParser()

voices = sorted([f.split('.')[0] for f in os.listdir('/samples')
                 if f.endswith('.txt') and not f.startswith('.')])
print('Voices found: ', voices)
tunes = sorted([f for f in os.listdir('/midi')
                if f.endswith('.mid') and not f.startswith('.')])
print('Midi files found: ', tunes)

selected_voice = None

def reset_voice_buttons():
    for i in range(len(voices)):
        trellis.pixels[(i, 0)] = 0x0000FF

def reset_tune_buttons():
    for i in range(len(tunes)):
        trellis.pixels[(i % 8, (i // 8) + 1)] = 0x00FF00

current_press = set()
reset_voice_buttons()
reset_tune_buttons()

while True:
    pressed = set(trellis.pressed_keys)
    just_pressed = pressed - current_press
    for down in just_pressed:
        if down[1] == 0:
            if down[0] < len(voices):                  # a voice selection
                selected_voice = down[0]
                reset_voice_buttons()
                trellis.pixels[down] = 0xFFFFFF
                syn.voice = voices[selected_voice]
        else:
            tune_index = (down[1] - 1) * 8 + down[0]
            if tune_index < len(tunes) and selected_voice is not None:
                trellis.pixels[down] = 0xFFFFFF
                header, tracks = p.parse('/midi/' + tunes[tune_index])
                for track in tracks:
                    seq.play(track)
                reset_tune_buttons()

    current_press = pressed
```

Code Walkthrough

You can browse the code files in GitHub here (https://adafru.it/Dj1)

main.py

`main.py` starts by initializing the hardware and class instances it will be using, collects the names of the sample and midi files, and blanks the display. A note about the file filtering. Clearly we only want `.txt` files for voices (as described earlier) and `.mid` for MIDI files. Additionally, we want to filter out files that start with `._` which MacOS will generate as part of its bookkeeping.

```
trellis = adafruit_trellism4.TrellisM4Express(rotation=0)
trellis.pixels.brightness = 0.1
trellis.pixels.fill(0)

syn = synth.Synth()
seq = sequencer.Sequencer(syn)
p = parser.MidiParser()

voices = sorted([f.split('.')[0] for f in os.listdir('/samples')
                 if f.endswith('.txt') and not f.startswith('.')])
print('Voices found: ', voices)
tunes = sorted([f for f in os.listdir('/midi')
                if f.endswith('.mid') and not f.startswith('.')])
print('Midi files found: ', tunes)
selected_voice = None
```

To set up the buttons there are two functions. These light up the top row with the valid instruments, while the rest are lit up for valid MIDI files.

```
def reset_voice_buttons():
    for i in range(len(voices)):
        trellis.pixels[(i, 0)] = 0x0000FF

def reset_tune_buttons():
    for i in range(len(tunes)):
        trellis.pixels[(i % 8, (i // 8) + 1)] = 0x00FF00

current_press = set()
reset_voice_buttons()
reset_tune_buttons()
```

The main loop waits for input from the user: instrument and MIDI file selection. If an instrument is selected (using the top row of buttons, those available are colored blue) the matching instrument name is sent to the `Synthesizer`. If a MIDI file is selected (the remaining 3 rows of buttons, usable ones are colored green) the file is parsed and the tracks sent to the `Sequencer` to be played.

The selected instrument is indicated by having its button turn white. While a MIDI file is being played, its button turns white as well, but returns to green when it finishes playing.

```
while True:
    pressed = set(trellis.pressed_keys)
    just_pressed = pressed - current_press
    for down in just_pressed:
        if down[1] == 0:
            if down[0] < len(voices):                  # a voice selection
                selected_voice = down[0]
                reset_voice_buttons()
                trellis.pixels[down] = 0xFFFFFF
                syn.voice = voices[selected_voice]
        else:
            tune_index = (down[1] - 1) * 8 + down[0]
            if tune_index < len(tunes) and selected_voice is not None:
                trellis.pixels[down] = 0xFFFFFF
                header, tracks = p.parse(tunes[tune_index])
                for track in tracks:
                    seq.play(track)
                reset_tune_buttons()

    current_press = pressed
```

parser.py

The `MidiParser` class does the job of reading a binary MIDI file and converting it into a header structure and a list of tracks. Each track is made up of events which are defined in `events.py` and discussed below.

The design and implementation of the parser was based on this format spec (https://adafru.it/Df4) from McGill University's music department (https://adafru.it/Df5).

The parser starts off with some methods that help with reading data from the MIDI file. These include methods to convert various sized bytearrays into numbers or a string.

```
    def _as_8(self, d):
        return d[0]

    def _as_16(self, d):
        return (d[0] << 8) | d[1]

    def _as_24(self, d):
        return (d[0] << 16) | (d[1] << 8) | d[2]

    def _as_32(self, d):
        return (d[0] << 24) | (d[1] << 16) | (d[2] << 8) | d[3]

    def _as_str(self, d):
        return str(d, encoding='utf8')
```

Next are methods for reading various numbers of bytes.

```
    def _read_bytes(self, f, count):
        val = f.read(count)
        return val

    def _read_1_byte(self, f):
        return self._read_bytes(f, 1)

    def _read_2_bytes(self, f):
        return self._read_bytes(f, 2)

    def _read_3_bytes(self, f):
        return self._read_bytes(f, 3)

    def _read_4_bytes(self, f):
        return self._read_bytes(f, 4)
```

The final set of reading methods read various sized numbers from the file. These sizes are specified in bits.

```
    def _read_8(self, f):
        return self._as_8(self._read_bytes(f, 1))

    def _read_16(self, f):
        return self._as_16(self._read_bytes(f, 2))

    def _read_24(self, f):
        return self._as_24(self._read_bytes(f, 3))

    def _read_32(self, f):
        return self._as_32(self._read_bytes(f, 4))
```

One thing that is used a lot in MIDI files is the idea of variable sized numbers. These are stored in the low 7 bits of each byte, with the highest bit indicating whether there are more bytes. As each byte is read, the high bit is checked. As long it is 1, the result is shifted left by 7 places and the low 7 bits of the new byte added. When the high bit is 0 it means that this byte contains the final 7 bits; the result is shifted 7 to the left and the final 7 bits are added.

```
    def _parse_variable_length_number(self, f):
        value = self._read_8(f)
        if value & 0x80:
            value &= 0x7F
            b = self._read_8(f)
            while b & 0x80:
                value = (value << 7) | (b & 0x7F)
                b = self._read_8(f)
            value = (value << 7) | b
        return value
```

Each MIDI file starts with a header that contains a few pieces of data, most importantly the number of tracks and some timing information. After opening the file, the first thing the parser does is parse the header. Based on the track count it finds there, if parses tracks.

```
    def parse(self, filename):
        with open(filename, 'rb') as f:
            tracks = []
            header = self._parse_header(f)
            for _ in range(header.number_of_tracks):
                tracks.append(self._parse_track(f))
        return (header, tracks)
```

The header starts with a tag of text `MThd`, followed by a data length (in 4 bytes). This has to have a value of `6`. If either of those things are wrong, `None` is returned to indicate a problem. If they're both good, the next 2 bytes are read and the timing information extracted from them. A `Header` object is constructed from the relevant information.

```
    def _parse_header(self, f):
        if self._read_4_bytes(f) != b'MThd':
            return None
        if self._read_32(f) != 6:
            return None
        midi_format = self._read_16(f)
        midi_number_of_tracks = self._read_16(f)
        d = self._read_2_bytes(f)
        if d[0] & 0x80:
            ticks_per_frame = d[1]
            negative_SMPTE_format = d[0] & 0x7F
            ticks_per_quarternote = None
        else:
            ticks_per_frame = None
            negative_SMPTE_format = None
            ticks_per_quarternote = (d[0] << 8) | d[1]
        return header.MidiHeader(midi_format, midi_number_of_tracks, ticks_per_frame, negative_SMPTE_form
```

Part of the data in the header is the number of tracks in the file. Once we have the header, we can parse that many tracks. Parsing a track is straight forward: there's a `MTrk` tag to start, followed by the number of events in the track (as a 32 bit number). Then it's just a matter of parsing that many events and appending them to the result list. As a fail safe, parsing stops when the end of track event is encountered.

```
    def _parse_track(self, f):
        if self._read_4_bytes(f) != b'MTrk':
            return None
        track_length = self._read_32(f)
        track_data = []
        for _ in range(track_length):
            event = self.parse_mtrk_event(f)
            if event is None:
                log('Error')
            track_data.append(event)
            if isinstance(event, events.EndOfTrackMetaEvent):
                return track_data
        return track_data
```

The next thing to look at is parsing events. This gets more complex fast since there are 4 different event types. The `parse_mtrk_event` method grabs the delta time and type of the event, which every event has as its initial two pieces of data. Based on the type, the appropriate parse method is called which gets more data and continues figuring out

what the event is before creating and returning it.

```
    def parse_mtrk_event(self, f):
        delta_time = self._parse_variable_length_number(f)
        event_type = self._read_8(f)
        if event_type == 0xF0:            #sysex event
            event = self._parse_F0_sysex_event(f, delta_time)
        elif event_type == 0xF7:          #sysex event
            event = self._parse_F7_sysex_event(f, delta_time)
        elif event_type == 0xFF:          #meta event
            event = self._parse_meta_event(f, delta_time)
        else:                          #regular midi event
            event = self._parse_midi_event(f, delta_time, event_type)
        log(event)
        return event
```

What's that `delta_time` all about? Each event in a MIDI track happens at a specific time. Instead of having an absolute timestamp, MIDI files use delta times. What that means is that the timestamp on a device is the time since the previous event. Also, it's not time-time, it's time relative to the tempo of the playing track. So it's more like a number of ticks rather than milliseconds or microseconds.

The different types of events encapsulate different types of commands. This code parses all of them, but only implements the more common ones. Notably, there are a few meta events like setting tempo, and marking the end of the track. Probably more interesting are some of the MIDI events (since they makes noises) `NoteOn` and `NoteOff` are the ones we implement.

The details of each event vary, so have a look through the code (on the downloads page) to see the details. To get you started, let's look at the code that parses note on and note off events.

```
    def _parse_midi_event(self, f, delta_time, status):
        if status & 0xF0 != 0xF0:
            command = (status & 0xF0) >> 4
            channel = status & 0x0F
            data_1 = self._read_8(f) & 0x7F
            data_2 = 0
            if command in [8, 9, 10, 11, 14]:
                data_2 = self._read_8(f) & 0x7F
            if command == 8:
                return events.NoteOffEvent(delta_time, channel, data_1, data_2)
            elif command == 9:
                if data_2 == 0:
                    return events.NoteOffEvent(delta_time, channel, data_1, data_2)
                return events.NoteOnEvent(delta_time, channel, data_1, data_2)
            .
            .
            .
```

Note on/off are MIDI events so are handled in the `_parse_midi_event` method. It's called with the timing value and what was the event type in the `_parse_mtrk_event` method. Here, in the context of a MIDI event, it's referred to as status. As much as possible, the terminology from the MIDI file format documentation has been adopted.

First of all, some MIDI events are channel independent and have a type/status with all 4 high bits on. I.e. they have values that are 0xFn. In that case, the event type is found in the lower 4 bits. Note on/off event, however, aren't of that

type; their type information (aka `command`) is in the high 4 bits. It's masked and shifted to extract it from the byte. The lower 4 bits are extracted by masking to provide the channel number. The first piece of data for the event is read next, and it's high bit removed. Some events require a second data byte, specifically those with command values of 9 through 11, and 14. If it is one of those (note on/off have command values of 8 and 9, respectively.

Now we know what type of event (from `command`), the time value, channel, and data bytes. Now the appropriate `Event` subclass can be constructed and returned. Other types of events are parsed similarly.

And that's the parser: binary MIDI file in, header object and list of tracks (each of which is a list of events) out.

header.py

The Header class is simple. It's essentially a constructor and a single property of interest: the number of tracks. More property getters could be added as more of it's data is needed, but for now this suffices. For debugging there's a description string method: `__str__`.

```
class MidiHeader(object):

    def __init__(self,
                 midi_format,
                 number_of_tracks,
                 ticks_per_frame,
                 negative_SMPTE_format,
                 ticks_per_quarternote):
        self._format = midi_format
        self._number_of_tracks = number_of_tracks
        self._ticks_per_frame = ticks_per_frame
        self._negative_SMPTE_format = negative_SMPTE_format
        self._ticks_per_quarternote = ticks_per_quarternote

    @property
    def number_of_tracks(self):
        return self._number_of_tracks

    def __str__(self):
        format_string = ('Header - format: {0}, '
                         'track count: {1}, '
                         'ticks per frame: {2}, '
                         'SMPTE: {3}, '
                         'ticks per quarternote: {4}')
        return format_string.format(self._format,
                                    self._number_of_tracks,
                                    self._ticks_per_frame,
                                    self._negative_SMPTE_format,
                                    self._ticks_per_quarternote)
```

events.py

This code file contains classes for all the MIDI events. The `Event` class is at the top of a small inheritance hierarchy. There are four subclass of `Event`: `F0SysexEvent`, `F7SysexEvent`, `MetaEvent`, and `MidiEvent`. The last two have a layer of subclasses under them for the actual events. Each event has a constructor and a *convert to string description* method. The events that are implemented also have an execute method. Each of those call a related method in the sequencer (which we'll discuss next) and return a boolean. They all must return `False` except for the `EndOfTrack` event that returns `True` to indicate that the track has completed.

The event classes provide a way to represent objects in an encapsulated way.

Consider the `NoteOnEvent`.

```
class NoteOnEvent(MidiEvent):

    def __init__(self, delta_time, channel, key, velocity):
        MidiEvent.__init__(self, delta_time, channel)
        self._key = key
        self._velocity = velocity

    def __str__(self):
        return '%d : Note On : key %d, velocity %d' % (self._delta_time, self._key, self._velocity)

    def execute(self, sequencer):
        sequencer.note_on(self._key, self._velocity)
        return False
```

It has a constructor that captures the arguments. The key and velocity (basically how hard/fast the key was pressed, ignored in our simple implementation) are stored directly while the time and channel are passed up the hierarchy.

As mentioned, each event has a `__str__` method that generates a readable description of the event for logging/debugging purposes.

Finally there is the execute method. The design of the system has this take a `Sequencer` object as its argument. The method then calls a corresponding method on the sequencer with the appropriate values (the key and pressure in this case). This is an example of double dispatch which allows the sequencer not to care what the events are, beyond having execute methods.

sequencer.py

The sequencer is pretty simple. There's a constructor that takes a synthesizer object that is used to make sounds and sets instance variables to sane defaults.

There's a `tick` method that just sleeps for an amount of time based on the tempo.

The `play` method takes a track and steps through it, calling execute on each event in order when it's time to. When executing an event returns `True`, the track is over and `play` exits.

```
    def play(self, track):
        for event in track:
            delta_time = 0
            while event.time > delta_time:
                delta_time += 1
                self._tick()
            print('Executing %s' % str(event))
            if event.execute(self):
                return
```

Finally there are methods that interface events with the system. In some cases this will do something with the synthesizer (e.g. starting and stopping notes) or with the sequencer itself (e.g. setting the tempo).

synth.py

This is a fun class. It's the one that makes sounds, after all.

As in the `Sequencer` class, the constructor here just initializes things. It doesn't do any *real work*. This is pretty commonly regarded as a *best practice*. See the section below for more on this important idea.

```
    def __init__(self):
        self._voice_name = None
        self._voice_file = None
        self._samples = [None] * 128
        self._channel_count = None
        self._bits_per_sample = None
        self._sample_rate = None
        self._audio = None
        self._mixer = None
        self._currently_playing = [{'key': None, 'voice' : x} for x in range(VOICE_COUNT)]
        self._voices_used = 0
```

Notably, even though the synthesizer uses the `audioio` module, it doesn't initialize it in the constructor. That's not done until it's first required. The work to do that is in the `_initialize_audio` method. Notice that the `_audio` instance variable is initialized to `None` in the constructor. If it's still `None` when `_initialize_audio` method is called, `_audio` and `_mixer` are initialized and connected. This is called *lazy initialization* and is a handy way to defer potentially expensive creation/initialization until it's actually required.

```
    def _initialize_audio(self):
        if self._audio is None:
            self._audio = audioio.AudioOut(board.A1)
            self._mixer = audioio.Mixer(voice_count=VOICE_COUNT,
                                        sample_rate=16000,
                                        channel_count=1,
                                        bits_per_sample=16,
                                        samples_signed=True)
            self._audio.play(self._mixer)
```

A key thing about the synthesizer is the instrument it uses. Here, that's called its voice. Notice that there's no instance variable called `_voice`. Instead it's implemented as a property and two instance variables: `_voice_name` and `_voice_file`.

```
    @property
    def voice(self):
        return self._voice_name

    @voice.setter
    def voice(self, v):
        self._initialize_audio()
        self._voice_name = utils.capitalize(v)
        self._voice_file = '/samples/%s.txt' % v.lower()
        first_note = None;
        with open(self._voice_file, "r") as f:
            for line in f:
                cleaned = line.strip()
                if len(cleaned) > 0 and cleaned[0] != '#':
                    key, filename = cleaned.split(',', 1)
                    self._samples[int(key)] = filename.strip()
                    if first_note is None:
                        first_note = filename.strip()
        sound_file = open(SAMPLE_FOLDER+first_note, 'rb')
        wav = audioio.WaveFile(sound_file)
        self._mixer.play(wav, voice=0, loop=False)
        time.sleep(0.5)
        self._mixer.stop_voice(0)
```

The getter is simple enough, it just returns the value of `_voice_name`. The setter is a bit more involved. The first thing is to now (finally) initialize the audio system. Next the associated instance variables are given values. `_voice_name` is set to the capitalized version of the string passed in. Notice that a `capitalize` function is included at the start of the module. CPython has `capitalize` as a standard part of it's `String` class, whereas CircuitPython (and apparently MicroPython) does not. The `_voice_file` variable is set to the name of the file that contains the MIDI key number to sample file mapping. That file is then loaded and parsed: putting the key-sample mapping into a `Dictionary`. Finally, a second of the first sample from that mapping file is played as feedback.

Next, we'll look at the business end of the class: the parts than play and stop notes. This synthesizer supports 8-note polyphony. That means that 8 different notes can be played at the same time. This is why it uses the `audioio.Mixer` class and not just the `audioio.AudioOut` class. When the mixer was constructed (see earlier) it had 8 voices allocated. You can see in `noteOn` how that's used.

```
    def note_on(self, key, velocity):
        fname = self._samples[key]
        if fname is not None:
            f = open(SAMPLE_FOLDER+fname, 'rb')
            wav = audioio.WaveFile(f)
            voice = self._find_usable_voice_for(key)
            if voice is not None:
                voice['key'] = key
                voice['file'] = f
                self._mixer.play(wav, voice=voice['voice'], loop=False)

    def note_off(self, key, velocity):
        if self._voices_used > 0:
            voice = self._find_voice_for(key)
            if voice is not None:
                self._voices_used -= 1
                self._mixer.stop_voice(voice['voice'])
                voice['file'].close()
                voice['file'] = None
                voice['key'] = None
```

As notes are played, they are assigned to a mixer voice. The _find_usable_voice_for method is used to find a mixer voice that isn't currently being used. That is then assigned to the note and the mixer is asked to play the sample.

Similarly, when a note is to be stopped the associated voice is found and the mixer is asked to stop it. That voice is then reclaimed for use later.

Here are the two voice finding methods:

```
    def _find_usable_voice_for(self, key):
        if self._voices_used == VOICE_COUNT:
            return None
        available = None
        for voice in self._currently_playing:
            if voice['key'] == key:
                return None
            if voice['key'] is None:
                available = voice
        if available is not None:
            self._voices_used += 1
            return available
        return None

    def _find_voice_for(self, key):
        for voice in self._currently_playing:
            if voice['key'] == key:
                return voice
        return None
```

A case against doing real work in a constructor

One of the better discussions of this is written by a colleague of the author's when he worked at Google: **Miško Hevery (https://adafru.it/Df6).** He argues that doing real work in a constructor:

Violates the Single Responsibility Principle (SRP): construction and internal initialization are inherently different

activities from creating other objects that will be interacted with. SRP postulates that a piece of code (a class, method, or function) should have a single responsibility: i.e. it should do one thing.

Makes testing directly difficult: Testing such constructors is difficult. To instantiate an object, the constructor must execute. And if that constructor does lots of work, you are forced to do that work when creating the object in tests. You may not have done testing for your CircuitPython code yet, but as the microcontrollers get more capable, the software we write for them can get more complex. It's a good idea to write tests for complex software. Therefore, we want to do anything we can to make testing easier and faster.

Forces collaborators on you: Using the constructor to create all the objects that might be interacted with makes construction take more time. You may not want to take that time up front if it slows the startup of your code. Also, it limits what collaborators your class has to those that are coded in the constructor. This makes reusing the class in the future more difficult. Passing objects that get used into the constructor is a better approach. You can see this in the way the `Sequencer`'s constructor takes the `Synthesizer` object to use.

Erases a "seam": Seams are places you can slice up your codebase to remove dependencies and instantiate small, focused objects and modules. If the constructor creates objects, those dependencies are hardcoded and it's harder to split the classes apart.