Guide Contents

Guide Contents

Overview
- MIDI CircuitPython
- UART MIDI
- Wood & Acrylic Case
- OLED & DisplayIO
- Parts List

Circuit Diagram
- Adafruit Library for Fritzing
- Wired Connections
- Powering

CAD Parts
- CAD Assembly
- CNC Milling
- Enclosure Parts Files
- Decal Files
- Design Source File

CircuitPython on Feather M4 Express
- Set up CircuitPython Quick Start!
- Further Information

CircuitPython Libraries
- The Adafruit CircuitPython Library Bundle
- Downloading the Adafruit CircuitPython Library Bundle
- The CircuitPython Community Library Bundle
- Downloading the CircuitPython Community Library Bundle
- Understanding the Bundle
- Example Files
- Copying Libraries to Your Board
- Understanding Which Libraries to Install
- Example: ImportError Due to Missing Library
- Library Install on Non-Express Boards
- Updating CircuitPython Libraries and Examples

Coding the MIDI Melody Maker
- Bitmap Sprite Sheet

CircuitPython Code Walkthrough
- Import the Libraries
- Turn Off the Onboard NeoPixel
- Setup the STEMMA OLED
- Create Text Objects for the OLED
- Setup the Blinka Tilegrid
- Setup MIDI
- Setup the Potentiometers and Switch Pins
- Create the MIDI Note Arrays
- MIDI Mode Arrays
- Key Name Strings for the OLED
- States and Default Array Indexes
- Beat Division Setup

The Loop
- Map the Analog Values
- Read the Potentiometers
- Beat Division
- Mode Selection

© Adafruit Industries
https://learn.adafruit.com/midi-melody-maker

Page 2 of 58
### Key Selection

- **BPM**
- **Run the MIDI Melody Maker**
- **Running the Modes: Fifths, Major Triads, Minor Triads and Pedal Tone**
- **Running Scale Mode**
- **Running Random Mode**
- **Stop the MIDI Melody Maker**

### Assembly

- Panel Mounting Parts
- Install OLED
- Install Slider
- Install Potentiometer
- Install Potentiometers (continued)
- Install Button
- Panel Mounted Parts

### Wiring

- Wiring Parts
- Wired Parts
- Wiring FeatherWing Doubler
- USB Extension Cable

### Final Build

- FeatherWing Doubler Standoffs
- Secure FeatherWing Doubler to Bottom Panel
- Feather M4 & MIDI FeatherWing
- Panel Mount USB Cable
- Plug-in USB Cable
- Installing Sides
- Install Acrylic Panels to Side Panels
- Installed Sides
- Final Build
Overview

**MIDI CircuitPython**
Build a custom MIDI instrument using CircuitPython! Use the MIDI FeatherWing and Feather M4 to power your musical projects using the classical 5-PIN DIN jacks. The 128x64 OLED with STEMMA QT makes it easy to add a screen with custom UI. Use potentiometers to change the modulation, key, beat division and modes to create an arrangement of MIDI notes.

**UART MIDI**
Using the MIDI library for CircuitPython, you can create MIDI notes and control MIDI data. Control synths with MIDI capabilities to create unique setups that does exactly what you want it to. The code is a great example of how to write CircuitPython code with MIDI control.

**Wood & Acrylic Case**
You can build a beautiful enclosure using acrylic and wood to make an elegant musical project. Use vinyl and a vinyl cutter to create custom decal graphics for labels. This project includes the vector SVG files and the 3D files.

**OLED & DisplayIO**
The OLED screen shows the BPM, beat division, key and mode selection. Adjusting potentiometers updates the values on screen in real time. Blinka the CircuitPython mascot nods her head along with the BPM acting as a metronome. This uses the displayio library for CircuitPython to display text, UI elements and animated bitmaps.
Parts used to build this project.

- MIDI FeatherWing [Link](https://adafru.it/NCh)
- Feather M4 Express [Link](https://adafru.it/Cmy)
- Doubler FeatherWing [Link](https://adafru.it/kBi)
- LED Button [Link](https://adafru.it/NCi)
- OLED Screen [Link](https://adafru.it/e3D)
- 10k Potentiometer Slider [Link](https://adafru.it/NCh)
- 10k Potentiometer Log [Link](https://adafru.it/NCh)
- USB Panel Mount [Link](https://adafru.it/JEO)
- 10-wire ribbon cable [Link](https://adafru.it/CJj)
- Slim metal knobs [Link](https://adafru.it/NCh)
- M2.5 Kit [Link](https://adafru.it/NCj)
- Stemma QT Cable [Link](https://adafru.it/FA-)

Monochrome 1.3” 128x64 OLED graphic display - STEMMA QT / Qwiic

These displays are small, only about 1.3” diagonal, but very readable due to the high contrast of an OLED display. This display is made of 128x64 individual white OLED pixels,...

$19.95

In Stock

Add to Cart

Adafruit MIDI FeatherWing Kit

Turn your Feather into a song-bird with this musically-enabled FeatherWing that adds MIDI input and output jacks to just about any Feather. You get both input and output DIN-5 MIDI...

$6.95

In Stock

Add to Cart

Adafruit Feather M4 Express - Featuring ATSAMD51

It's what you've been waiting for, the Feather M4 Express featuring ATSAMD51. This Feather is fast like a swift, smart like an owl, strong like a ox-bird (it's half ox,...

$22.95

In Stock

Add to Cart

FeatherWing Doubler - Prototyping Add-on For All Feather Boards

This is the FeatherWing Doubler - a prototyping add-on and more for all Feather boards. This is similar to our...
In Stock

Add to Cart

Your browser does not support the video tag.

**Slide Potentiometer with Plastic Knob - 45mm Long**
Slip slidin' away Slip slidin' away You know the nearer your resistance The more you're slip slidin' away If you're...
$1.95

In Stock

Add to Cart

1 x **16mm LED Pushbutton**
White Latching On/Off Switch

Add to Cart

4 x **Panel Mount 10K Log Potentiometer**
Breadboard Friendly

Add to Cart

4 x **Slim Metal Knobs**
10mm Diameter x 10mm - T18

Add to Cart

1 x **Panel Mount Extension USB Cable**
Micro B Male to Micro B Female

Add to Cart

1 x **10-Wire Silicone Cover Stranded-Core Ribbon Cable**
28AWG 1 Meter Long

Add to Cart

1 x **M2.5 Hardware Kit**
Black Nylon Standoffs, Screws and Hex Nuts

Add to Cart

1 x **STEMMA QT Cable**
JST SH 4-pin to Premium Male Headers Cable - 150mm Long

Add to Cart
Circuit Diagram

The diagram below provides a visual reference for wiring of the components. This diagram was created using the software package Fritzing (https://adafru.it/oEP).

Adafruit Library for Fritzing

Use Adafruit's Fritzing parts library to create circuit diagrams for your projects. Download the library or just grab individual parts. Get the library and parts from GitHub - Adafruit Fritzing Parts (https://adafru.it/AYZ).

Wired Connections

All parts share common ground and voltage. The ground and voltage lines are wired to the 3V and GND pins on the 128x64 OLED breakout. The 128x64 OLED breakout uses a STEMMA QT cable wired into the Doubler FeatherWing. All of the signals from the potentiometers are wired into the Doubler FeatherWing.

128x64 OLED breakout

- SDA to SDA
- SCL to SCL
- VCC to 3V
- GND to GND
**Button**

- Ground to GND
- Signal to D5
- LED anode to 3V (with 220ohm resistor)
- LED cathode to GND

**Potentiometers**

- Key to pot A1
- Mode pot A2
- Beat pot to A3
- BPM Slider to A4
- Modulation to A5

**Powering**

The Adafruit board can be powered via USB or JST using a 3.7v lipo battery.
CAD Parts

CAD Assembly
The OLED screen, five potentiometers and LED button are panel mounted to the top panel. A microUSB extension cable is panel mounted to the back panel. The doubler FeatherWing is panel mounted to the bottom panel using M2.5 x 6mm long standoffs.

CNC Milling
Manufacturing models were created for each part and feature stock setups using Fusion 360. Tool paths were generated using the tool library from Bantam Tools.

Enclosure Parts Files
The case can optionally be laser cut out of 1/8in thick acrylic. Download the zip file and setup the parts using your preferred software and manufacturing method. Each part is its own SVG file allowing for custom layouts and multi material use.

https://adafruit.it/NCn

https://adafruit.it/NCn
Decal Files
Additional labels are cut using a vinyl cutter. Vinyl is applied to transfer tape and placed onto acrylic panels. Graphics are included in the SVG zip file.

Design Source File
A STEP file is included and features sketches and solid bodies. The Fusion 360 file features adjustable user parameters and contains tool paths for CNC milling.
CircuitPython (https://adafru.it/tB7) is a derivative of MicroPython (https://adafru.it/BeZ) 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.

The following instructions will show you how to install CircuitPython. If you've already installed CircuitPython but are looking to update it or reinstall it, the same steps work for that as well!

Set up CircuitPython Quick Start!

Follow this quick step-by-step for super-fast Python power :)

Click the link above and download the latest UF2 file.
Download and save it to your desktop (or wherever is handy).

Plug your Feather M4 into your computer using a known-good USB cable.

A lot of people end up using charge-only USB cables and it is very frustrating! So make sure you have a USB cable you know is good for data sync.

Double-click the Reset button next to the USB connector on your board, and you will see the NeoPixel RGB LED turn green. If it turns red, check the USB cable, try another USB port, etc. **Note:** The little red LED next to the USB connector will pulse red. That's ok!

If double-clicking doesn't work the first time, try again. Sometimes it can take a few tries to get the rhythm right!
You will see a new disk drive appear called **FEATHERBOOT**.

Drag the `adafruit_circuitpython_etc.uf2` file to **FEATHERBOOT**.

The LED will flash. Then, the **FEATHERBOOT** drive will disappear and a new disk drive called **CIRCUITPY** will appear.

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

**Further Information**

For more detailed info on installing CircuitPython, check out [Installing CircuitPython](https://adafruit.it/Amd).
CircuitPython Libraries

As CircuitPython development continues and there are new releases, Adafruit will stop supporting older releases. Visit https://circuitpython.org/downloads to download the latest version of CircuitPython for your board. You must download the CircuitPython Library Bundle that matches your version of CircuitPython. Please update CircuitPython and then visit https://circuitpython.org/libraries to download the latest Library Bundle.

Each CircuitPython program you run needs to have a lot of information to work. The reason CircuitPython is so simple to use is that most of that information is stored in other files and works in the background. These files are called libraries. Some of them are built into CircuitPython. Others are stored on your CIRCUITPY drive in a folder called lib. Part of what makes CircuitPython so great is its ability to store code separately from the firmware itself. Storing code separately from the firmware makes it easier to update both the code you write and the libraries you depend.

Your board may ship with a lib folder already, it's in the base directory of the drive. If not, simply create the folder yourself. When you first install CircuitPython, an empty lib directory will be created for you.

CircuitPython libraries work in the same way as regular Python modules so the Python docs (https://adafruit.it/rar) are an excellent reference for how it all should work. In Python terms, you can place our library files in the lib directory because it's part of the Python path by default.

One downside of this approach of separate libraries is that they are not built in. To use them, one needs to copy them to the CIRCUITPY drive before they can be used. Fortunately, there is a library bundle.

The bundle and the library releases on GitHub also feature optimized versions of the libraries with the .mpy file extension. These files take less space on the drive and have a smaller memory footprint as they are loaded.

Due to the regular updates and space constraints, Adafruit does not ship boards with the entire bundle. Therefore, you will need to load the libraries you need when you begin working with your board. You can find example code in the guides for your board that depends on external libraries.
Either way, as you start to explore CircuitPython, you'll want to know how to get libraries on board.

The Adafruit CircuitPython Library Bundle

Adafruit provides CircuitPython libraries for much of the hardware they provide, including sensors, breakouts and more. To eliminate the need for searching for each library individually, the libraries are available together in the Adafruit CircuitPython Library Bundle. The bundle contains all the files needed to use each library.

Downloading the Adafruit CircuitPython Library Bundle

You can download the latest Adafruit CircuitPython Library Bundle release by clicking the button below. The libraries are being constantly updated and improved, so you'll always want to download the latest bundle.

**Match up the bundle version with the version of CircuitPython you are running.** For example, you would download the 6.x library bundle if you're running any version of CircuitPython 6, or the 7.x library bundle if you're running any version of CircuitPython 7, etc. If you mix libraries with major CircuitPython versions, you will get incompatible mpy errors due to changes in library interfaces possible during major version changes.

Download the bundle version that matches your CircuitPython firmware version. If you don't know the version, check the version info in `boot_out.txt` file on the CIRCUITPY drive, or the initial prompt in the CircuitPython REPL. For example, if you're running v7.0.0, download the 7.x library bundle.

There's also a `py` bundle which contains the uncompressed python files, you probably *don't* want that unless you are doing advanced work on libraries.

The CircuitPython Community Library Bundle

The CircuitPython Community Library Bundle is made up of libraries written and provided by members of the CircuitPython community. These libraries are often written when community members encountered hardware not supported in the Adafruit Bundle, or to support a personal project. The authors all chose to submit these libraries to the Community Bundle make them available to the community.

**These libraries are maintained by their authors and are not supported by Adafruit.** As you would with any library, if you run into problems, feel free to file an issue on the GitHub repo for the library. Bear in mind, though, that most of these libraries are supported by a single person and you should be patient about receiving a response. Remember, these folks are not paid by Adafruit, and are volunteering their personal time when possible to provide support.
Downloading the CircuitPython Community Library Bundle

You can download the latest CircuitPython Community Library Bundle release by clicking the button below. The libraries are being constantly updated and improved, so you'll always want to download the latest bundle.

https://adafruit.it/VCn

The link takes you to the latest release of the CircuitPython Community Library Bundle on GitHub. There are multiple versions of the bundle available. **Download the bundle version that matches your CircuitPython firmware version.** If you don't know the version, check the version info in `boot_out.txt` file on the CIRCUITPY drive, or the initial prompt in the CircuitPython REPL. For example, if you're running v7.0.0, download the 7.x library bundle.

Understanding the Bundle

After downloading the zip, extract its contents. This is usually done by double clicking on the zip. On Mac OSX, it places the file in the same directory as the zip.

Open the bundle folder. Inside you'll find two information files, and two folders. One folder is the lib bundle, and the other folder is the examples bundle.

Now open the lib folder. When you open the folder, you'll see a large number of `.mpy` files, and folders.
Example Files

All example files from each library are now included in the bundles in an **examples** directory (as seen above), as well as an examples-only bundle. These are included for two main reasons:

- Allow for quick testing of devices.
- Provide an example base of code, that is easily built upon for individualized purposes.

Copying Libraries to Your Board

First open the **lib** folder on your CIRCUITPY drive. Then, open the **lib** folder you extracted from the downloaded zip. Inside you’ll find a number of folders and .mpy files. Find the library you’d like to use, and copy it to the **lib** folder on CIRCUITPY.

If the library is a directory with multiple .mpy files in it, be sure to **copy the entire folder to CIRCUITPY/lib**.

This also applies to example files. Open the **examples** folder you extracted from the downloaded zip, and copy the applicable file to your CIRCUITPY drive. Then, rename it to **code.py** to run it.

Understanding Which Libraries to Install

You now know how to load libraries on to your CircuitPython-compatible microcontroller board. You may now be wondering, how do you know **which** libraries you need to install? Unfortunately, it's not always
straightforward. Fortunately, there is an obvious place to start, and a relatively simple way to figure out the rest. First up: the best place to start.

When you look at most CircuitPython examples, you'll see they begin with one or more `import` statements. These typically look like the following:

- `import library_or_module`

However, `import` statements can also sometimes look like the following:

- `from library_or_module import name`
- `from library_or_module.subpackage import name`
- `from library_or_module import name as local_name`

They can also have more complicated formats, such as including a `try / except` block, etc.

The important thing to know is that an `import` statement will always include the name of the module or library that you're importing.

Therefore, the best place to start is by reading through the `import` statements.

Here is an example import list for you to work with in this section. There is no setup or other code shown here, as the purpose of this section involves only the import list.

```python
import time
import board
import neopixel
import adafruit_lis3dh
import usb_hid
from adafruit_hid.consumer_control import ConsumerControl
from adafruit_hid.consumer_control_code import ConsumerControlCode
```

Keep in mind, not all imported items are libraries. Some of them are almost always built-in CircuitPython modules. How do you know the difference? Time to visit the REPL.

In the Interacting with the REPL section (https://adafruit.it/Awz) on The REPL page (https://adafruit.it/Awz) in this guide, the `help("modules")` command is discussed. This command provides a list of all of the built-in modules available in CircuitPython for your board. So, if you connect to the serial console on your board, and enter the REPL, you can run `help("modules")` to see what modules are available for your board. Then, as you read through the `import` statements, you can, for the purposes of figuring out which libraries to load, ignore the statement that import modules.

The following is the list of modules built into CircuitPython for the Feather RP2040. Your list may look similar or be anything down to a significant subset of this list for smaller boards.
Now that you know what you're looking for, it's time to read through the import statements. The first two, `time` and `board`, are on the modules list above, so they're built-in.

The next one, `neopixel`, is not on the module list. That means it's your first library! So, you would head over to the bundle zip you downloaded, and search for `neopixel`. There is a `neopixel.mpy` file in the bundle zip. Copy it over to the lib folder on your CIRCUITPY drive. The following one, `adafruit_lis3dh`, is also not on the module list. Follow the same process for `adafruit_lis3dh`, where you'll find `adafruit_lis3dh.mpy`, and copy that over.

The fifth one is `usb_hid`, and it is in the modules list, so it is built in. Often all of the built-in modules come first in the import list, but sometimes they don't! Don't assume that everything after the first library is also a library, and verify each import with the modules list to be sure. Otherwise, you'll search the bundle and come up empty!

The final two imports are not as clear. Remember, when import statements are formatted like this, the first thing after the `from` is the library name. In this case, the library name is `adafruit_hid`. A search of the bundle will find an `adafruit_hid folder`. When a library is a folder, you must copy the entire folder and its contents as it is in the bundle to the lib folder on your CIRCUITPY drive. In this case, you would copy the entire `adafruit_hid` folder to your CIRCUITPY/lib folder.

Notice that there are two imports that begin with `adafruit_hid`. Sometimes you will need to import more than one thing from the same library. Regardless of how many times you import the same library, you only need to load the library by copying over the `adafruit_hid` folder once.

That is how you can use your example code to figure out what libraries to load on your CircuitPython-compatible board!

There are cases, however, where libraries require other libraries internally. The internally required library is called a dependency. In the event of library dependencies, the easiest way to figure out what other libraries are required is to connect to the serial console and follow along with the `ImportError` printed there. The following is a very simple example of an `ImportError`, but the concept is the same for any missing library.
Example: **ImportError** Due to Missing Library

If you choose to load libraries as you need them, or you're starting fresh with an existing example, you may end up with code that tries to use a library you haven't yet loaded. This section will demonstrate what happens when you try to utilise a library that you don't have loaded on your board, and cover the steps required to resolve the issue.

This demonstration will only return an error if you do not have the required library loaded into the `lib` folder on your CIRCUITPY drive.

Let's use a modified version of the Blink example.

```python
import board
import time
import simpleio

led = simpleio.DigitalOut(board.LED)

while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
```

Save this file. Nothing happens to your board. Let's check the serial console to see what's going on.

![Serial Console](image)

You have an **ImportError**. It says there is no module named 'simpleio'. That's the one you just included in your code!

Click the link above to download the correct bundle. Extract the lib folder from the downloaded bundle file. Scroll down to find `simpleio.mpy`. This is the library file you're looking for! Follow the steps above to load an individual library file.

The LED starts blinking again! Let's check the serial console.
No errors! Excellent. You've successfully resolved an `ImportError`!

If you run into this error in the future, follow along with the steps above and choose the library that matches the one you're missing.

**Library Install on Non-Express Boards**

If you have an M0 non-Express board such as Trinket M0, Gemma M0, QT Py M0, or one of the M0 Trinkeys, you'll want to follow the same steps in the example above to install libraries as you need them. Remember, you don't need to wait for an `ImportError` if you know what library you added to your code. Open the library bundle you downloaded, find the library you need, and drag it to the `lib` folder on your `CIRCUITPY` drive.

You can still end up running out of space on your M0 non-Express board even if you only load libraries as you need them. There are a number of steps you can use to try to resolve this issue. You'll find suggestions on the [Troubleshooting page](https://adafruit.it/Den).

**Updating CircuitPython Libraries and Examples**

Libraries and examples are updated from time to time, and it's important to update the files you have on your `CIRCUITPY` drive.

To update a single library or example, follow the same steps above. When you drag the library file to your `lib` folder, it will ask if you want to replace it. Say yes. That's it!

A new library bundle is released every time there's an update to a library. Updates include things like bug fixes and new features. It's important to check in every so often to see if the libraries you're using have been updated.
Coding the MIDI Melody Maker

Once you've finished setting up your Feather M4 Express with CircuitPython, you can add these libraries to the `lib` folder:

- `adafruit_display_shapes`
- `adafruit_display_text`
- `adafruit_imageload`
- `adafruit_midi`
- `adafruit_displayio_ssd1306.mpy`
- `neopixel.mpy`
- `simpleio.mpy`

Then, you can click on the Download: Project Zip link above the code to download the code file and bitmap graphic.

```python
import time
from random import randint
import board
import simpleio
import busio
import terminalio
import neopixel
from digitalio import DigitalInOut, Direction, Pull
from analogio import AnalogIn
import displayio
import adafruit_imageload
from adafruit_display_text import label
import adafruit_displayio_ssd1306
# uncomment if using USB MIDI
# import usb_midi
from adafruit_display_shapes.rect import Rect
import adafruit_midi
from adafruit_midi.note_on          import NoteOn
from adafruit_midi.note_off         import NoteOff
from adafruit_midi.control_change   import ControlChange

displayio.release_displays()

oled_reset = board.D9

# turn off on-board neopixel
pixel = neopixel.NeoPixel(board.NEOPIXEL, 1, brightness=0)
pixel.fill((0, 0, 0))

# Use for I2C for STEMMA OLED
i2c = board.I2C()
display_bus = displayio.I2CDisplay(i2c, device_address=0x3D, reset=oled_reset)

# STEMMA OLED dimensions. can have height of 64, but 32 makes text larger
```
WIDTH = 128
HEIGHT = 32
BORDER = 0

# blinka sprite indexes
EMPTY = 0
BLINKA_1 = 1
BLINKA_2 = 2

# setup for STEMMA OLED
display = adafruit_displayio_ssd1306.SSD1306(display_bus, width=WIDTH, height=HEIGHT)

# create the displayio object
splash = displayio.Group()
display.show(splash)

# text for BPM
bpm_text = "BPM:    "
bpm_text_area = label.Label(terminalio.FONT, text=bpm_text, color=0xFFFFFF, x=4, y=6)
splash.append(bpm_text_area)
bpm_rect = Rect(0, 0, 50, 16, fill=None, outline=0xFFFFFF)
splash.append(bpm_rect)

# text for key
key_text = "Key:    "
key_text_area = label.Label(terminalio.FONT, text=key_text, color=0xFFFFFF, x=4, y=21)
splash.append(key_text_area)
key_rect = Rect(0, 15, 50, 16, fill=None, outline=0xFFFFFF)
splash.append(key_rect)

# text for mode
mode_text = "Mode:           "
mode_text_area = label.Label(terminalio.FONT, text=mode_text, color=0xFFFFFF, x=54, y=21)
splash.append(mode_text_area)
mode_rect = Rect(50, 15, 78, 16, fill=None, outline=0xFFFFFF)
splash.append(mode_rect)

# text for beat division
beat_text = "Div:       "
beat_text_area = label.Label(terminalio.FONT, text=beat_text, color=0xFFFFFF, x=54, y=6)
splash.append(beat_text_area)
beat_rect = Rect(50, 0, 78, 16, fill=None, outline=0xFFFFFF)
splash.append(beat_rect)

# Blinka sprite setup
(blinka, blinka_pal) = adafruit_imageload.load("/spritesWhite.bmp", © Adafruit Industries
https://learn.adafruit.com/midi-melody-maker
Page 24 of 58
blinka, blinka_pal = adafruit_imageload.load("/spritesWhite.bmp",
    bitmap=displayio.Bitmap,
    palette=displayio.Palette)

# creates a transparent background for Blinka
blinka_pal.make_transparent(7)
blinka_grid = displayio.TileGrid(blinka, pixel_shader=blinka_pal,
    width=1, height=1,
    tile_height=16, tile_width=16,
    default_tile=EMPTY)

blinka_grid.x = 112
blinka_grid.y = 0

splash.append(blinka_grid)

# imports MIDI

# USB MIDI:
# midi = adafruit_midi.MIDI(midi_out=usb_midi.ports[1], out_channel=0)
# UART MIDI:
midi = adafruit_midi.MIDI(midi_out=busio.UART(board.TX, board.RX, baudrate=31250),
    out_channel=0)

# potentiometer pin setup
key_pot = AnalogIn(board.A1)
mode_pot = AnalogIn(board.A2)
beat_pot = AnalogIn(board.A3)
bpm_slider = AnalogIn(board.A4)
mod_pot = AnalogIn(board.A5)

# run switch setup
run_switch = DigitalInOut(board.D5)
run_switch.direction = Direction.INPUT
run_switch.pull = Pull.UP

# arrays of notes in each key
key_of_C = [60, 62, 64, 65, 67, 69, 71, 72]
key_of_Csharp = [61, 63, 65, 66, 68, 70, 72, 73]
key_of_D = [62, 64, 66, 67, 69, 71, 73, 74]
key_of_Dsharp = [63, 65, 67, 68, 70, 72, 74, 75]
key_of_E = [64, 66, 68, 69, 71, 73, 75, 76]
key_of_F = [65, 67, 69, 70, 72, 74, 76, 77]
key_of_Fsharp = [66, 68, 70, 71, 73, 75, 77, 78]
key_of_G = [67, 69, 71, 72, 74, 76, 78, 79]
key_of_Gsharp = [68, 70, 72, 73, 75, 77, 79, 80]
key_of_A = [69, 71, 73, 74, 76, 78, 80, 81]
key_of_Asharp = [70, 72, 74, 75, 77, 79, 81, 82]
key_of_B = [71, 73, 75, 76, 78, 80, 82, 83]

# array of keys

# array of note indexes for modes
fifths = [0, 4, 3, 7, 2, 6, 4, 7]
major = [4, 2, 0, 3, 5, 7, 6, 4]
minor = [5, 7, 2, 4, 6, 5, 1, 3]
pedal = [5, 5, 5, 6, 5, 5, 5, 7]

# defining variables for key name strings
C_name = "C"
Csharp_name = "C#"
D_name = "D"
Dsharp_name = "D#"
E_name = "E"
F_name = "F"
Fsharp_name = "F#"
G_name = "G"
Gsharp_name = "G#"
A_name = "A"
Asharp_name = "A#"
B_name = "B"

# array of strings for key names for use with the display
key_names = [C_name, Csharp_name, D_name, Dsharp_name, E_name, F_name, Fsharp_name,
            G_name, Gsharp_name, A_name, Asharp_name, B_name]

# function for reading analog inputs
def val(voltage):
    return voltage.value

# comparitors for pots' values
mod_val2 = 0
beat_val2 = 0
bpm_val2 = 120
key_val2 = 0
mode_val2 = 0

# time.monotonic for running the modes
run = 0

# state for being on/off
run_state = False

# indexes for modes
r = 0
b = 0
f = 0
p = 0
maj = 0
mi = 0
random = 0

# mode states
play_pedal = False
play_fifths = False
play_maj = False
play_min = False
play_rando = False
play_scale = True

# state for random beat division
rando = False

# comparitors for states
last_r = 0
last_f = 0
last_maj = 0
last_min = 0
last_p = 0
last_random = 0
# index for random beat division
hit = 0
# default tempo
tempo = 60
# beat division
sixteenth = 15 / tempo
eighth = 30 / tempo
quarter = 60 / tempo
half = 120 / tempo
whole = 240 / tempo
# time.monotonic for blinka animation
slither = 0
# blinka animation sprite index
g = 1

# array for random beat division values
rando_div = [240, 120, 60, 30, 15]
# array of beat division values
beat_division = [whole, half, quarter, eighth, sixteenth]
# strings for beat division names
beat_division_name = ["1", "1/2", "1/4", "1/8", "1/16", "Random"]

while True:
    # mapping analog pot values to the different parameters
    # MIDI modulation 0-127
    mod_val1 = round(simpleio.map_range(val(mod_pot), 0, 65535, 0, 127))
    # BPM range 60-220
    bpm_val1 = simpleio.map_range(val(bpm_slider), 0, 65535, 60, 220)
    # 6 options for beat division
    beat_val1 = round(simpleio.map_range(val(beat_pot), 0, 65535, 0, 5))
    # 12 options for key selection
    key_val1 = round(simpleio.map_range(val(key_pot), 0, 65535, 0, 11))
    # 6 options for mode selection
    mode_val1 = round(simpleio.map_range(val(mode_pot), 0, 65535, 0, 5))

    # sending MIDI modulation
    if abs(mod_val1 - mod_val2) > 2:
        # updates previous value to hold current value
        mod_val2 = mod_val1
        # MIDI data has to be sent as an integer
        # this converts the pot data into an int
        modulation = int(mod_val2)
        # int is stored as a CC message
        modWheel = ControlChange(1, modulation)
        # CC message is sent
        midi.send(modWheel)
        print(modWheel)
        # delay to settle MIDI data
        time.sleep(0.001)

    # sets beat division
    if abs(beat_val1 - beat_val2) > 0:
        # updates previous value to hold current value
        beat_val2 = beat_val1
        print("beat div is", beat_val2)
        # updates display

# updates display
beat_text_area.text = "Div:%s" % beat_division_name[beat_val2]
# sets random beat division state
if beat_val2 == 5:
    rando = True
else:
    rando = False
time.sleep(0.001)

# mode selection
if abs(mode_val1 - mode_val2) > 0:
    # updates previous value to hold current value
    mode_val2 = mode_val1

# scale mode
if mode_val2 == 0:
    play_scale = True
    play_maj = False
    play_min = False
    play_fifths = False
    play_pedal = False
    play_rando = False
    # updates display
    mode_text_area.text = "Mode:Scale"
    print("scale")

# major triads mode
if mode_val2 == 1:
    play_scale = False
    play_maj = True
    play_min = False
    play_fifths = False
    play_pedal = False
    play_rando = False
    print("major chords")
    # updates display
    mode_text_area.text = "Mode:MajorTriads"

# minor triads mode
if mode_val2 == 2:
    play_scale = False
    play_maj = False
    play_min = True
    play_fifths = False
    play_pedal = False
    play_rando = False
    print("minor")
    # updates display
    mode_text_area.text = "Mode:MinorTriads"

# fifths mode
if mode_val2 == 3:
    play_scale = False
    play_maj = False
    play_min = False
    play_fifths = True
    play_pedal = False
    play_rando = False
    print("fifths")
    # updates display
    mode_text_area.text = "Mode:Fifths"
# pedal tone mode
if mode_val2 == 4:
    play_scale = False
    play_maj = False
    play_min = False
    play_fifths = False
    play_pedal = True
    play_rando = False
    print("play random")
    # updates display
    mode_text_area.text = 'Mode:Pedal'

# random mode
if mode_val2 == 5:
    play_scale = False
    play_maj = False
    play_min = False
    play_fifths = False
    play_pedal = False
    play_rando = True
    print("play random")
    # updates display
    mode_text_area.text = 'Mode:Random'
    time.sleep(0.001)

# key selection
if abs(key_val1 - key_val2) > 0:
    # updates previous value to hold current value
    key_val2 = key_val1
    # indexes the notes in each key array
    for k in keys:
        o = keys.index(k)
        octave = keys[o]
    # updates display
    key_text_area.text = 'Key:%s' % key_names[key_val2]
    print("o is", o)
    time.sleep(0.001)

# BPM adjustment
if abs(bpm_val1 - bpm_val2) > 1:
    # updates previous value to hold current value
    bpm_val2 = bpm_val1
    # updates tempo
    tempo = int(bpm_val2)
    # updates calculations for beat division
    sixteenth = 15 / tempo
    eighth = 30 / tempo
    quarter = 60 / tempo
    half = 120 / tempo
    whole = 240 / tempo
    # updates array of beat divisions
    beat_division = [whole, half, quarter, eighth, sixteenth]
    # updates display
    bpm_text_area.text = "BPM:%d" % tempo
    print("tempo is", tempo)
    time.sleep(0.05)

# if the run switch is pressed:

© Adafruit Industries  https://learn.adafruit.com/midi-melody-maker  Page 29 of 58
if run_switch.value:
    run_state = True
# if random beat division, then beat_division index is randomized with index hit
if rando:
    divide = beat_division[hit]
# if not random, then beat_division is the value of the pot
else:
    divide = beat_division[beat_val2]
# blinka animation in time with BPM and beat division
# she will slither every time a note is played
if (time.monotonic() - slither) >= divide:
    blinka_grid[0] = g
    g += 1
    slither = time.monotonic()
    if g > 2:
        g = 1
# holds key index
octave = keys[key_val2]
# fifths mode
if play_fifths:
    # tracks time divided by the beat division
    if (time.monotonic() - run) >= divide:
        # note index from mode, r counts index position
        f = fifths[r]
        # sends NoteOn
        midi.send(NoteOn(octave[f]))
        # turns previous note off
        midi.send(NoteOff(octave[last_f]))
        # print(octave[r])
        run = time.monotonic()
        # go to next note
        r += 1
        # updates previous value to hold current value
        if r > 0:
            last_r = r
            last_f = f
            hit = randint(2, 4)
        # resets note index position
        if r > 7:
            r = 0
            last_r = r
            last_f = f
            hit = randint(2, 4)
    # major triad mode
if play_maj:
    # tracks time divided by the beat division
    if (time.monotonic() - run) >= divide:
        # note index from mode, r counts index position
        maj = major[r]
        # sends NoteOn
        midi.send(NoteOn(octave[maj]))
        # turns previous note off
        midi.send(NoteOff(octave[last_maj]))
        # print(octave[r])
        run = time.monotonic()
        # go to next note
        r += 1
# updates previous value to hold current value
if r > 0:
    last_r = r
    last_maj = maj
    hit = randint(2, 4)
    # resets note index position
if r > 7:
    r = 0
    last_r = r
    last_maj = maj
    hit = randint(2, 4)
# minor triad mode
if play_min:
    # tracks time divided by the beat division
    if (time.monotonic() - run) >= divide:
        # note index from mode, r counts index position
        mi = minor[r]
        # sends NoteOn
        midi.send(NoteOn(octave[mi]))
        # turns previous note off
        midi.send(NoteOff(octave[last_min]))
        # print(octave[r])
        run = time.monotonic()
    # go to next note
    r += 1
    # updates previous value to hold current value
if r > 0:
    last_r = r
    last_min = mi
    hit = randint(2, 4)
    # resets note index position
if r > 7:
    r = 0
    last_r = r
    last_min = mi
    hit = randint(2, 4)
# pedal tone mode
if play_pedal:
    # tracks time divided by the beat division
    if (time.monotonic() - run) >= divide:
        # note index from mode, r counts index position
        p = pedal[r]
        # sends NoteOn
        midi.send(NoteOn(octave[p]))
        # turns previous note off
        midi.send(NoteOff(octave[last_p]))
        # print(octave[r])
        run = time.monotonic()
    # go to next note
    r += 1
    # updates previous value to hold current value
if r > 0:
    last_r = r
    last_p = p
    hit = randint(2, 4)
    # resets note index position
if r > 7:
if r > 7:
    r = 0
last_r = r
last_p = p
hit = randint(2, 4)
    # random note mode
    if play_rando:
        # randomizes note indexes in key
        r = randint(0, 7)
        # tracks time divided by the beat division
        if (time.monotonic() - run) >= divide:
            # sends NoteOn
            midi.send(NoteOn(octave[r]))
            # turns previous note off
            midi.send(NoteOff(octave[last_r]))
            # print(octave[r])
            run = time.monotonic()
            # updates previous value to hold current value
            if r > 0:
                last_r = r
                r = randint(0, 7)
                hit = randint(2, 4)
    # scale mode
    if play_scale:
        # tracks time divided by the beat division
        if (time.monotonic() - run) >= divide:
            # sends NoteOn
            midi.send(NoteOn(octave[r]))
            # turns previous note off
            midi.send(NoteOff(octave[last_r]))
            # print(octave[r])
            run = time.monotonic()
            # go to next note
            r += 1
            # updates previous value to hold current value
            if r > 0:
                last_r = r
                hit = randint(2, 4)
                # resets note index position
                if r > 7:
                    r = 0
                    last_r = r
        if not run_switch.value:
            if run_state:
                all_note_off = ControlChange(123, 0)
                # CC message is sent
                midi.send(all_note_off)
                run_state = False
                time.sleep(0.001)
    # delay to settle MIDI data
    time.sleep(0.005)

Bitmap Sprite Sheet
There is a bitmap sprite sheet, `spritesWhite.bmp`, that is also needed for the code. It will allow for an animation to play on the OLED of Blinka slithering along with the beat of your melodies.

Your Feather M4 CIRCUITPY drive should look like this after you load the libraries, bitmap graphic and `code.py` file:
CircuitPython Code Walkthrough

### Import the Libraries

First, import the libraries. If you're using USB MIDI, then be sure to uncomment `import usb_midi`.

```python
import time
from random import randint
import board
import simpleio
import busio
import terminalio
import neopixel
from digitalio import DigitalInOut, Direction, Pull
from analogio import AnalogIn
import displayio
import adafruit_imageload
from adafruit_display_text import label
import adafruit_displayio_ssd1306
# uncomment if using USB MIDI
# import usb_midi
from adafruit_display_shapes.rect import Rect
import adafruit_midi
from adafruit_midi.note_on          import NoteOn
from adafruit_midi.note_off         import NoteOff
from adafruit_midi.control_change   import ControlChange
```

### Turn Off the Onboard NeoPixel

The onboard NeoPixel is turned off so that it doesn't show through the acrylic case.

```python
# turn off on-board neopixel
pixel = neopixel.NeoPixel(board.NEOPIXEL, 1, brightness=0)
pixel.fill((0, 0, 0))
```

### Setup the STEMMA OLED

The STEMMA OLED is setup as the display with `displayio`. It uses I2C for communication. The actual height of the OLED is 64 pixels, but by cutting it in half to 32 pixels, the `terminalio` font appears larger on the screen.

The sprite indexes for the Blinka sprite that will be animated later in the code are also setup.
Create Text Objects for the OLED

Text objects are setup for the four different parameters that can be controlled with the MIDI Melody Maker: BPM (beats per minute), key, mode and beat division. As you adjust the parameters, their values will be updated live on the screen.
# text for BPM
bpm_text = "BPM:    
bpm_text_area = label.Label(
    terminalio.FONT, text=bpm_text, color=0xFFFFFF, x=4, y=6
)
splash.append(bpm_text_area)

bpm_rect = Rect(0, 0, 50, 16, fill=None, outline=0xFFFFFF)
splash.append(bpm_rect)

# text for key
key_text = "Key:    
key_text_area = label.Label(
    terminalio.FONT, text=key_text, color=0xFFFFFF, x=4, y=21
)
splash.append(key_text_area)

key_rect = Rect(0, 15, 50, 16, fill=None, outline=0xFFFFFF)
splash.append(key_rect)

# text for mode
mode_text = "Mode:           
mode_text_area = label.Label(
    terminalio.FONT, text=mode_text, color=0xFFFFFF, x=54, y=21
)
splash.append(mode_text_area)

mode_rect = Rect(50, 15, 78, 16, fill=None, outline=0xFFFFFF)
splash.append(mode_rect)

# text for beat division
beat_text = "Div:       
beat_text_area = label.Label(
    terminalio.FONT, text=beat_text, color=0xFFFFFF, x=54, y=6
)
splash.append(beat_text_area)

beat_rect = Rect(50, 0, 78, 16, fill=None, outline=0xFFFFFF)
splash.append(beat_rect)

Setup the Blinka Tilegrid

The Blinka sprite is setup using the adafruit_imageload library. It is setup as a tilegrid so you can iterate through the grid to create an animation of Blinka slithering.
# Blinka sprite setup
blinka, blinka_pal = adafruit_imageload.load("/spritesWhite.bmp",
    bitmap=displayio.Bitmap,
    palette=displayio.Palette)

# creates a transparent background for Blinka
blinka_pal.make_transparent(7)
blinka_grid = displayio.TileGrid(blinka, pixel_shader=blinka_pal,
    width=1, height=1,
    tile_height=16, tile_width=16,
    default_tile=EMPTY)

blinka_grid.x = 112
blinka_grid.y = 0

splash.append(blinka_grid)

Setup MIDI

MIDI communication is setup using the adafruit_midi library. You can either use USB MIDI or MIDI over UART with the MIDI FeatherWing’s 5-DIN ports.

# imports MIDI
# USB MIDI:
# midi = adafruit_midi.MIDI(midi_out=usb_midi.ports[1], out_channel=0)
# UART MIDI:
# midi = adafruit_midi.MIDI(midi_out=busio.UART(board.TX, board.RX, baudrate=31250),
#   out_channel=0)

Setup the Potentiometers and Switch Pins

The pins are setup for the five analog potentiometers and the switch.

# potentiometer pin setup
key_pot = AnalogIn(board.A1)
mode_pot = AnalogIn(board.A2)
beat_pot = AnalogIn(board.A3)
bpm_slider = AnalogIn(board.A4)
mod_pot = AnalogIn(board.A5)

# run switch setup
run_switch = DigitalInOut(board.D5)
run_switch.direction = Direction.INPUT
run_switch.pull = Pull.UP

Create the MIDI Note Arrays
The MIDI notes for each key (C through B) are setup as arrays. These arrays are then placed into another array called `keys` so that they can be accessed.

```python
# arrays of notes in each key
key_of_C = [60, 62, 64, 65, 67, 69, 71, 72]
key_of_Csharp = [61, 63, 65, 66, 68, 70, 72, 73]
key_of_D = [62, 64, 66, 67, 69, 71, 73, 74]
key_of_Dsharp = [63, 65, 67, 68, 70, 72, 74, 75]
key_of_E = [64, 66, 68, 69, 71, 73, 75, 76]
key_of_F = [65, 67, 69, 70, 72, 74, 76, 77]
key_of_Fsharp = [66, 68, 70, 71, 73, 75, 77, 78]
key_of_G = [67, 69, 71, 72, 74, 76, 78, 79]
key_of_Gsharp = [68, 70, 72, 73, 75, 77, 79, 80]
key_of_A = [69, 71, 73, 74, 76, 78, 80, 81]
key_of_Asharp = [70, 72, 74, 75, 77, 79, 81, 82]
key_of_B = [71, 73, 75, 76, 78, 80, 82, 83]

# array of keys
```

**MIDI Mode Arrays**

The different modes call on the array index location for different points in a key. This allows for you to play these different patterns in different keys automatically. These arrays are referencing note indexes ranging from 0 to 7.

You can create your own patterns by creating your own arrays of note indexes.

```python
# array of note indexes for modes
fifths = [0, 4, 3, 7, 2, 6, 4, 7]
major = [4, 2, 0, 3, 5, 7, 6, 4]
minor = [5, 7, 2, 4, 6, 5, 1, 3]
pedal = [5, 5, 6, 5, 5, 5, 7]
```

**Key Name Strings for the OLED**

Variables are created for the strings of key names. These variables are put into an array called `key_names` so that they can be displayed on the OLED.
# defining variables for key name strings
C_name = "C"
Csharp_name = "C#"
D_name = "D"
Dsharp_name = "D#"
E_name = "E"
F_name = "F"
Fsharp_name = "F#"
G_name = "G"
Gsharp_name = "G#"
A_name = "A"
Asharp_name = "A#"
B_name = "B"

# array of strings for key names for use with the display
key_names = [C_name, Csharp_name, D_name, Dsharp_name, E_name, F_name, Fsharp_name,
             G_name, Gsharp_name, A_name, Asharp_name, B_name]

States and Default Array Indexes

States and default array indexes are setup to be referenced later in the loop.
Beat Division Setup

Arrays are created for beat division. These values allow the MIDI Melody Maker to divide the BPM to play
different note values. The possible values are whole notes, half notes, quarter notes, eighth notes and sixteenth notes.

There is also an array of strings so that the beat division can be displayed on the OLED.

```python
# array for random beat division values
rando_div = [240, 120, 60, 30, 15]
# array of beat division values
beat_division = [whole, half, quarter, eighth, sixteenth]
# strings for beat division names
beat_division_name = ["1", "1/2", "1/4", "1/8", "1/16", "Random"]
```

### The Loop

#### Map the Analog Values

The loop begins by mapping the analog values from the potentiometers to the values needed for the different parameters.

```python
# mapping analog pot values to the different parameters
# MIDI modulation 0-127
mod_val1 = round(simpleio.map_range(val(mod_pot), 0, 65535, 0, 127))
# BPM range 60-220
bpm_val1 = simpleio.map_range(val(bpm_slider), 0, 65535, 60, 220)
# 6 options for beat division
beat_val1 = round(simpleio.map_range(val(beat_pot), 0, 65535, 0, 5))
# 12 options for key selection
key_val1 = round(simpleio.map_range(val(key_pot), 0, 65535, 0, 11))
# 6 options for mode selection
mode_val1 = round(simpleio.map_range(val(mode_pot), 0, 65535, 0, 5))
```

#### Read the Potentiometers

All of the parameters that are defined by analog potentiometers compare the last reading of the pot to the current reading to define whether the state has changed.

If the state has changed, then values are updated to affect how the MIDI Melody Maker is generating data. The display on the OLED is updated with the new value.

For modulation, a modulation MIDI CC message is sent.
# sending MIDI modulation
if abs(mod_val1 - mod_val2) > 2:
    # updates previous value to hold current value
    mod_val2 = mod_val1
    # MIDI data has to be sent as an integer
    # this converts the pot data into an int
    modulation = int(mod_val2)
    # int is stored as a CC message
    modWheel = ControlChange(1, modulation)
    # CC message is sent
    midi.send(modWheel)
    print(modWheel)
    # delay to settle MIDI data
    time.sleep(0.001)

Beat Division

For beat division, the text is updated and if the beat division is going to be randomized, the `rando` state is updated to `True`. The index for the beat division array is also stored in `beat_val2` and is referenced later in the loop.

```python
# sets beat division
if abs(beat_val1 - beat_val2) > 0:
    # updates previous value to hold current value
    beat_val2 = beat_val1
    print("beat div is", beat_val2)
    # updates display
    beat_text_area.text = "Div:%s" % beat_division_name[beat_val2]
    # sets random beat division state
    if beat_val2 == 5:
        rando = True
    else:
        rando = False
    time.sleep(0.001)
```

Mode Selection

Mode selection sets the different mode states to `True` depending on the array index and also updates the OLED’s text.

```python
# mode selection
if abs(mode_val1 - mode_val2) > 0:
    # updates previous value to hold current value
    mode_val2 = mode_val1
    # scale mode
    if mode_val2 == 0:
        play_scale = True
        play_maj = False
```

© Adafruit Industries  https://learn.adafruit.com/midi-melody-maker  Page 42 of 58
play_min = False
play_fifths = False
play_pedal = False
play_rando = False
# updates display
mode_text_area.text = "Mode:Scale"
print("scale")
# major triads mode
if mode_val2 == 1:
    play_scale = False
    play_maj = True
    play_min = False
    play_fifths = False
    play_pedal = False
    play_rando = False
    print("major chords")
    # updates display
    mode_text_area.text = "Mode:MajorTriads"
    # minor triads mode
    if mode_val2 == 2:
        play_scale = False
        play_maj = False
        play_min = True
        play_fifths = False
        play_pedal = False
        play_rando = False
        print("minor")
        # updates display
        mode_text_area.text = "Mode:MinorTriads"
    # fifths mode
    if mode_val2 == 3:
        play_scale = False
        play_maj = False
        play_min = False
        play_fifths = True
        play_pedal = False
        play_rando = False
        print("fifths")
        # updates display
        mode_text_area.text = "Mode:Fifths"
    # pedal tone mode
    if mode_val2 == 4:
        play_scale = False
        play_maj = False
        play_min = False
        play_fifths = False
        play_pedal = True
        play_rando = False
        print("play random")
        # updates display
        mode_text_area.text = 'Mode:Pedal'
    # random mode
    if mode_val2 == 5:
        play_scale = False
        play_maj = False
        play_min = False
        play_fifths = False
play_rando = True
print("play random")
# updates display
time.sleep(0.001)

Key Selection

Key selection defines which key is selected from the keys array with key_val2 holding the index. octave will be used later in the loop to access the key’s MIDI note array. The text for the key's name is also updated for the OLED.

```python
# key selection
if abs(key_val1 - key_val2) > 0:
    # updates previous value to hold current value
    key_val2 = key_val1
    # indexes the notes in each key array
    for k in keys:
        o = keys.index(k)
        octave = keys[o]
        # updates display
        key_text_area.text = 'Key:%s' % key_names[key_val2]
    print("o is", o)
    time.sleep(0.001)
```

BPM

BPM is adjusted with the sliding potentiometer. It’s value is stored as an integer in tempo. The tempo is divided in the beat division formulas to get the new beat division values. These values are updated in the beat_division array. The BPM’s text is updated for the OLED.
# BPM adjustment

```python
if abs(bpm_val1 - bpm_val2) > 1:
    # updates previous value to hold current value
    bpm_val2 = bpm_val1
    # updates tempo
    tempo = int(bpm_val2)
    # updates calculations for beat division
    sixteenth = 15 / tempo
    eighth = 30 / tempo
    quarter = 60 / tempo
    half = 120 / tempo
    whole = 240 / tempo
    # updates array of beat divisions
    beat_division = [whole, half, quarter, eighth, sixteenth]
    # updates display
    bpm_text_area.text = "BPM:%d" % tempo
    print("tempo is", tempo)
    time.sleep(0.05)
```

## Run the MIDI Melody Maker

If the `run_switch` is pressed, then the `run_state` is `True` and the MIDI Melody Maker begins generating MIDI data.

- `divide` holds the value of the beat division data and determines when a note is played. If the beat division is going to be randomized, then `hit` (which holds a random integer) acts as the index for `beat_division`. Otherwise, `beat_val2` (which holds the value of the beat division pot) defines the index.

The Blinka animation advances every time a note is played by comparing `time.monotonic()` to `divide`.

- `octave` allows access to the different keys’ MIDI note indexes so that the MIDI notes can be played in the different modes.
# if the run switch is pressed:
if run_switch.value:
    run_state = True
# if random beat division, then beat_division index is randomized with index hit
if rando:
    divide = beat_division[hit]
# if not random, then beat_division is the value of the pot
else:
    divide = beat_division[beat_val2]
# blinka animation in time with BPM and beat division
# she will slither every time a note is played
if (time.monotonic() - slither) >= divide:
    blinka_grid[0] = g
    g += 1
    slither = time.monotonic()
if g > 2:
    g = 1
# holds key index
octave = keys[key_val2]

Running the Modes: Fifths, Major Triads, Minor Triads and Pedal Tone

The modes with defined arrays (fifths, major triads, minor triads and pedal tone) work in the same way. If their state is True, then time.monotonic() is compared to divide to determine when the next note should be played.

A variable holds the index of the mode's array. That variable is then used as the index for the key's array to be able to play specific notes.

The next note is sent as a MIDI NoteOn message and the previous note is turned off with a MIDI NoteOff message.

Once the end of the array is reached, index positions are reset to the beginning to that the modes can continue playing.
# fifths mode
if play_fifths:
    # tracks time divided by the beat division
    if (time.monotonic() - run) >= divide:
        # note index from mode, r counts index position
        f = fifths[r]
        # sends NoteOn
        midi.send(NoteOn(octave[f]))
        # turns previous note off
        midi.send(NoteOff(octave[last_f]))
        # print(octave[r])
        run = time.monotonic()
        # go to next note
        r += 1
        # updates previous value to hold current value
        if r > 0:
            last_r = r
            last_f = f
            hit = randint(2, 4)
        # resets note index position
        if r > 7:
            r = 0
            last_r = r
            last_f = f
            hit = randint(2, 4)

Running Scale Mode

For scale mode, rather than referencing an array of defined notes, the keys' array of MIDI note numbers is played in ascending order. `r` holds the index value and increases by 1. `r` is reset when it is greater than 7, which is the end of the array.
# scale mode
if play_scale:
    # tracks time divided by the beat division
    if (time.monotonic() - run) >= divide:
        # sends NoteOn
        midi.send(NoteOn(octave[r]))
        # turns previous note off
        midi.send(NoteOff(octave[last_r]))
        # print(octave[r])
        run = time.monotonic()
        # go to next note
        r += 1
        # updates previous value to hold current value
        if r > 0:
            last_r = r
            hit = randint(2, 4)
        # resets note index position
        if r > 7:
            r = 0
            last_r = r

Running Random Mode

For random mode, the notes are played in a randomized order. In this case, \( r \) holds a random integer that is updated every time a note is played. \( r \) does not need to be reset since the random integer is constantly updated and is not constrained by an array.

# random note mode
if play_rando:
    # randomizes note indexes in key
    r = randint(0, 7)
    # tracks time divided by the beat division
    if (time.monotonic() - run) >= divide:
        # sends NoteOn
        midi.send(NoteOn(octave[r]))
        # turns previous note off
        midi.send(NoteOff(octave[last_r]))
        # print(octave[r])
        run = time.monotonic()
        # updates previous value to hold current value
        if r > 0:
            last_r = r
            hit = randint(2, 4)
            r = randint(0, 7)
        # resets note index position
        if r > 7:
            r = 0
            last_r = r

Stop the MIDI Melody Maker

If you turn off the MIDI Melody Maker by pressing the `run_switch`, then an all notes off message is sent
with MIDI CC message 123. This prevents any notes from accidentally hanging on, which can happen with some DAWs and hardware synths.

`run_state` is then set to `False` to reset for the next time you turn the MIDI Melody Maker on.

```python
if not run_switch.value:
    if run_state:
        all_note_off = ControlChange(123, 0)
        # CC message is sent
        midi.send(all_note_off)
        run_state = False
        time.sleep(0.001)
```
Assembly

Panel Mounting Parts
Gather up the parts for paneling mounting to the top acrylic.

Install OLED
Remove the protection film from the display. Insert the screws through the top side of the acrylic. Place the OLED breakout over the acrylic and fit the screws through the mounting holes. Install hex nuts onto the threads of the screws to secure.

- 4x m2.5 x 10mm screw
- 4x m2.5 hex but

Install Slider
Remove the tip from the slide by pulling it out of the stem. Place onto the bottom side of the top acrylic panel. Line up the mounting holes and insert the screws to secure.

- 2x M2 x 8mm flat head screw

If the screw is metal and too long it could short the ground and signal pins! Be careful, this can affect the functionality of the slider.
Install Potentiometer
Panel mount the 10k potentiometer into the holes in the top acrylic panel. The holes are sized to be 7.2mm in diameter.

Install Potentiometers (continued)
Proceed to panel mount the other potentiometers into the top acrylic panel. Use the washer and hex nut that came with the potentiometer to secure.

Install Button
Remove the hex nut washer from the button. Install the button by mounting it to the bigger hole in the top acrylic panel. Use the hex nut washer to secure to the acrylic panel.
Panel Mounted Parts
Take a moment to check the screws and hex nuts are tight and secure. Install the tip back onto the stem of the slider.
Wiring

Wiring Parts
Follow the circuit diagram and reference the wired connections. Use 10-wire silicone cover ribbon cable to make bundled connections.

Wired Parts
The parts all share common ground that is wired into the OLED breakout. Wire lengths are moderately sized to allow top and bottom panels to move around. The OLED display uses a STEMMA QT cable for easily plugging into the breakout.

Wiring FeatherWing Doubler
Voltage and ground connections from the OLED breakout are wired into the 3V and GND pins on the FeatherWing Doubler. The signal connections for the potentiometers and button are wired into the accompanying pins on the FeatherWing Doubler.
USB Extension Cable
A panel mounted microUSB extension cable is used to extend the microUSB port on the Feather M4. This cable includes screws for panel mounting to the back acrylic panel. It has been (optionally) resized using silicone ribbon cable to better fit into the enclosure.
Final Build

**FeatherWing Doubler Standoffs**
Install the hardware onto the mounting holes on the FeatherWing Doubler.

- 4x m2.5 x 6mm ff standoff
- 4x m2.5 x 6mm screw
- 4x m2.6 x 8mm screw

**Secure FeatherWing Doubler to Bottom Panel**
Place the FeatherWing Doubler onto the bottom acrylic panel and line up the mounting holes. Insert and fasten screws to secure the FeatherWing Doubler to the bottom acrylic panel.

**Feather M4 & MIDI FeatherWing**
Insert the Feather M4 and MIDI FeatherWing into the headers of the FeatherWing Doubler. Reference the photo for correct placement.
Panel Mount USB Cable
Place the microUSB cable onto the small hole on the back acrylic panel. Insert and fasten the screws through the panel to secure microUSB cable.

Plug-in USB Cable
Connect the microUSB extension cable to the Feather M4. Double check the orientation of the Feather M4 and MIDI FeatherWing are correct.

Installing Sides
Gather up the acrylic panels to fit into the left and right sides.
Install Acrylic Panels to Side Panels
Lay one of the sides onto your work surface with the slots facing up. Insert the tabs from the acrylic panels into the slots on the left and right sides.

Installed Sides
Carefully fit the second side onto the acrylic panels. Fit the tabs from the acrylic panels into the slots on the side piece. Firmly press pieces together.

Final Build
And there you have it! The parts are wired up and the enclosure is assembled. Plug in a good data USB cable to power up the Feather. Optionally plug in any 3.7v battery from the Adafruit shop to the Feather to go portable.