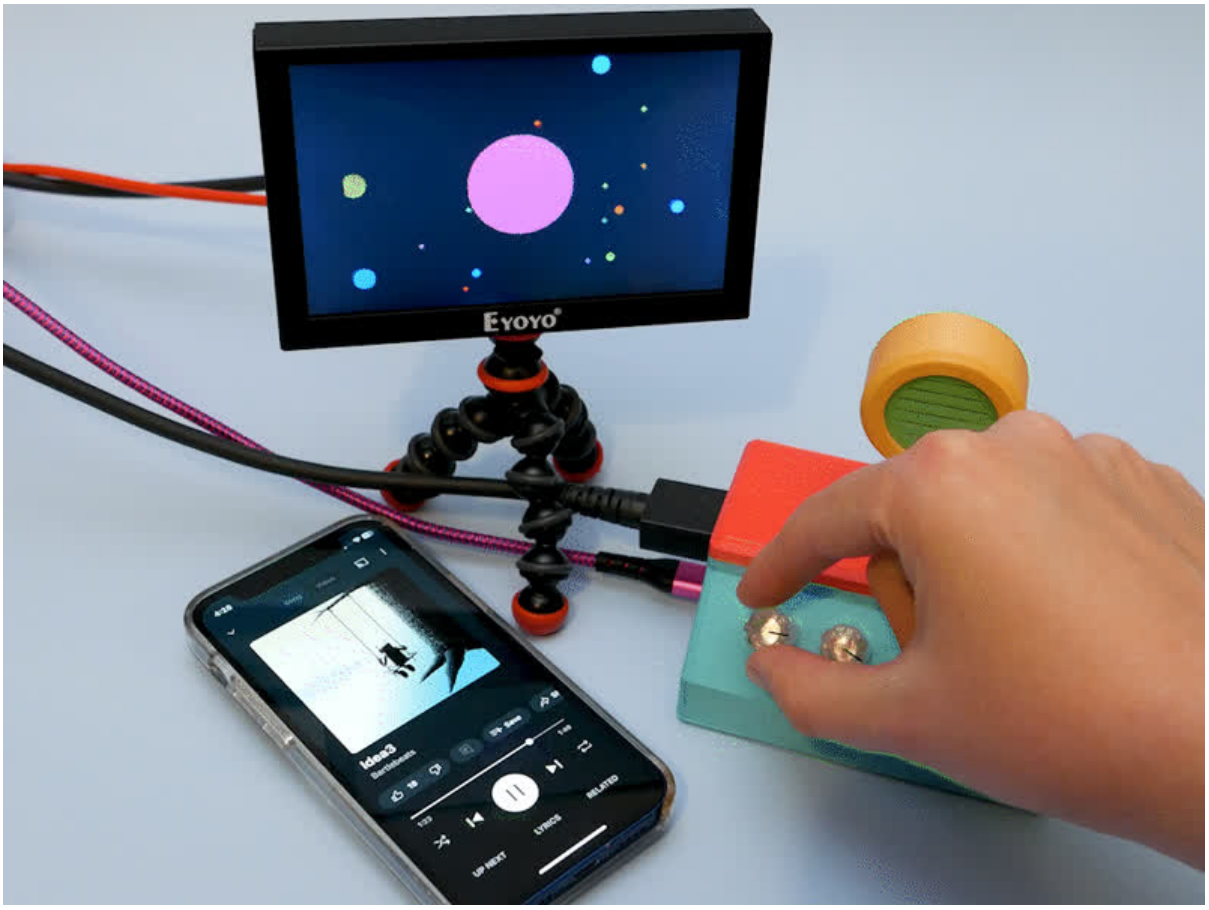
# Feather RP2350 Audio Reactive Video Synth

Created by Liz Clark




https://learn.adafruit.com/feather-rp2350-audio-reactive-video-synth

Last updated on 2025-01-13 04:13:00 PM EST

# Table of Contents


Overview 3

- Parts
- Fastener Hardware

Circuit Diagram 7

3D Printing 8

- 3D Printed Parts
- Build Volume
- Design Source Files

Install CircuitPython 9

- CircuitPython Quickstart
- Safe Mode
- Flash Resetting UF2

Code the Video Synth 13

- Upload the Code, Bitmap and Libraries to the Feather RP2350
- Code Inspiration
- How the Code Works
- seesaw Rotary Encoder
- PicoDVI
- Variables and States
- The Animations
- The Loop

Wiring 26

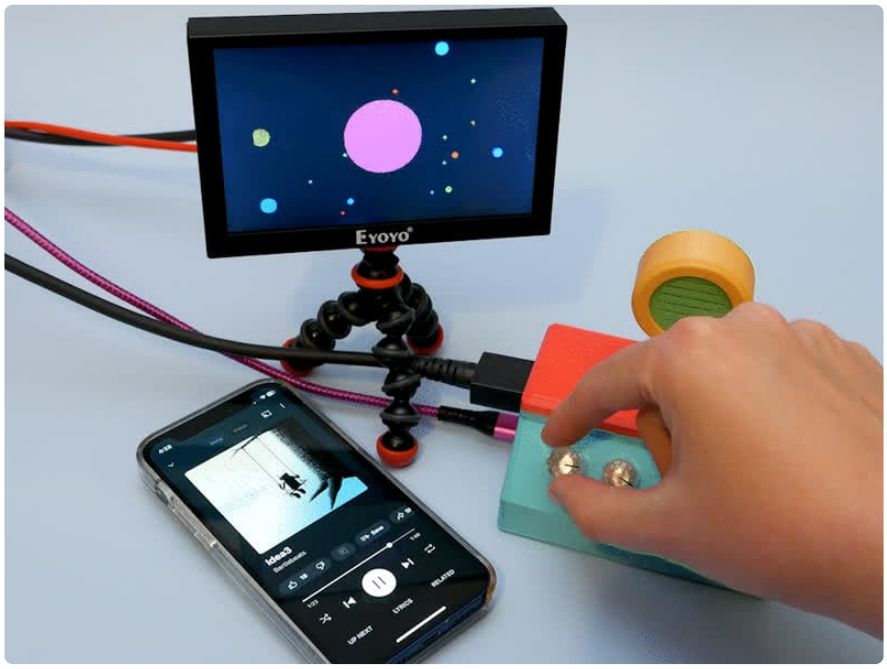
- Potentiometer 3
- Potentiometer 2
- Potentiometer 1
- Potentiometers to Feather Wiring
- PDM STEMMA Cable

Assembly 30

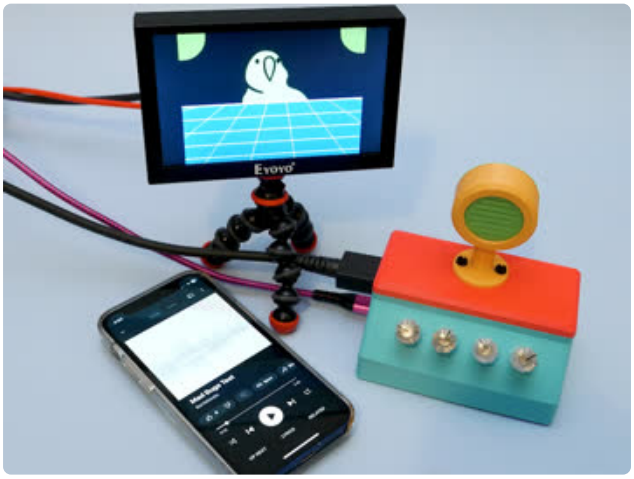
- Mount the Potentiometers
- Rotary Encoder
- Case Lid and Microphone Enclosure
- HSTX to DVI Breakout
- PDM Mic
- Final Touches

Use 36

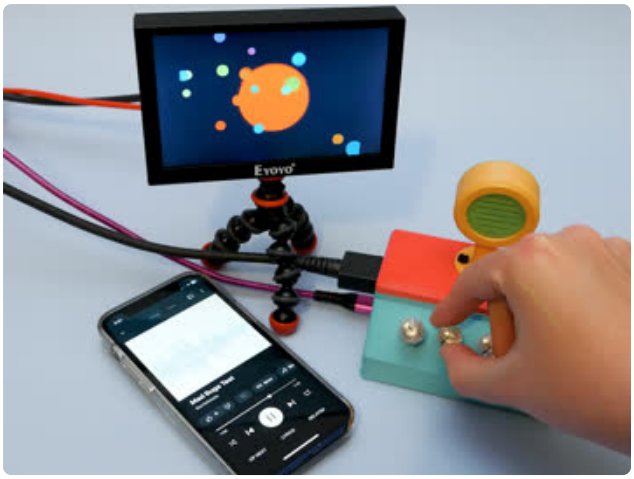
# Overview



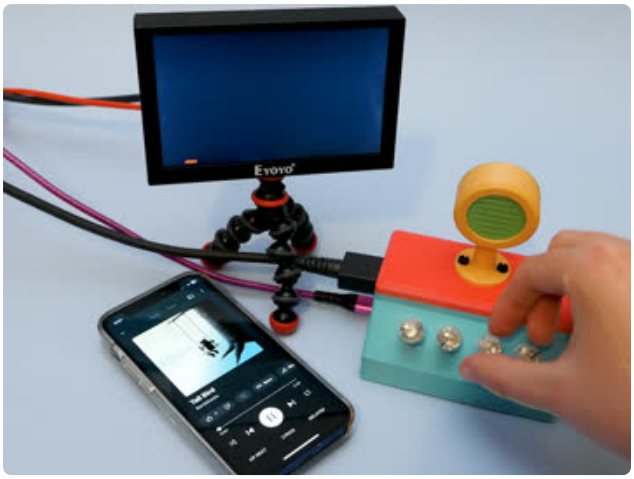

The Feather RP2350 with HSTX makes one of our project dreams come true: a CircuitPython-powered video synth! The Feather runs CircuitPython code that outputs graphics from HSTX to DVI. A PDM microphone takes in audio data to let the animations be audio reactive. There are three different animations to choose from. You can change which one is selected with the seesaw rotary encoder and control different parameters with the three analog potentiometers.




This video synth project makes adding reactive visuals to your music a breeze by using a PDM microphone to sample the incoming audio over time and analyze it with fast Fourier transform (FFT) (https://adafru.it/qMC). This means that the three animations can visualize in real-time the amplitude and frequencies of your music.

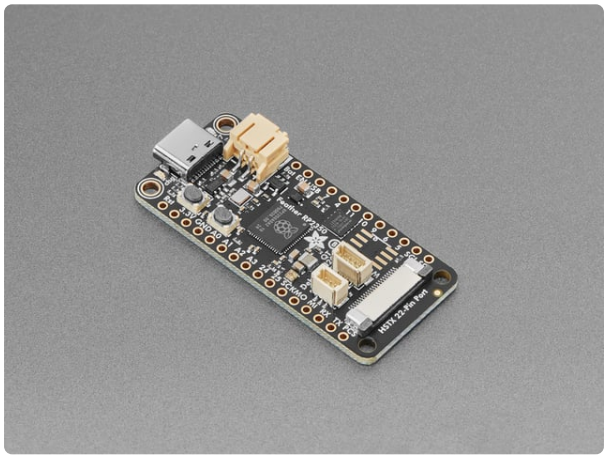


The analog potentiometers are super responsive with the animations. Each animation has three different parameters that can be controlled. You can control things like the color of a shape...

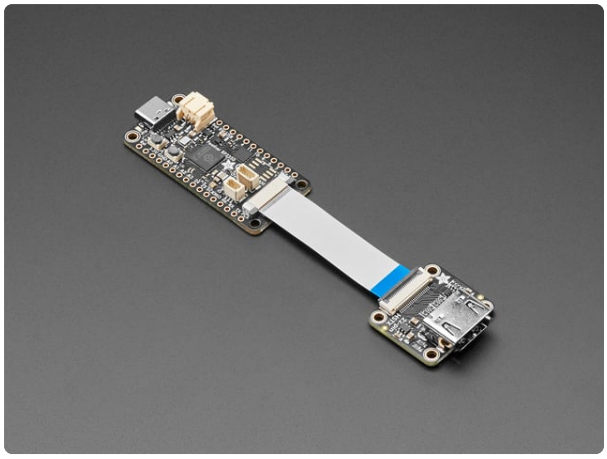


...or the noise floor for the FFT calculations aka how sensitive the PDM mic is to incoming sound. This can be handy for quieter music or noisier spaces with a lot of background noise.

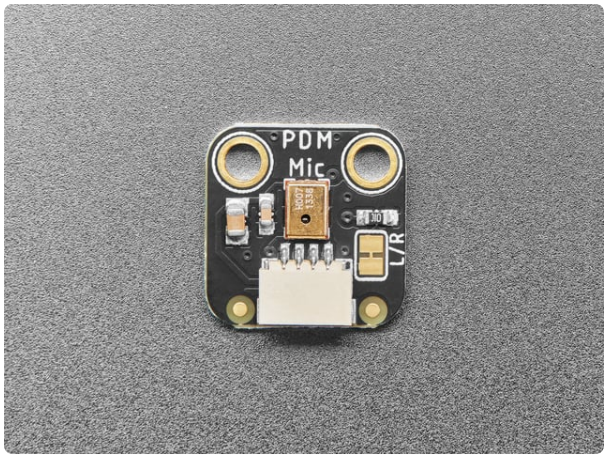
## Parts



Adafruit Feather RP2350 with HSTX Port
RP2350 flies high with the Feather format - now you can use any FeatherWings with this battery-powered dev board. It comes with 8MB of flash, 22pin HSTX output port,...
https://www.adafruit.com/product/6000

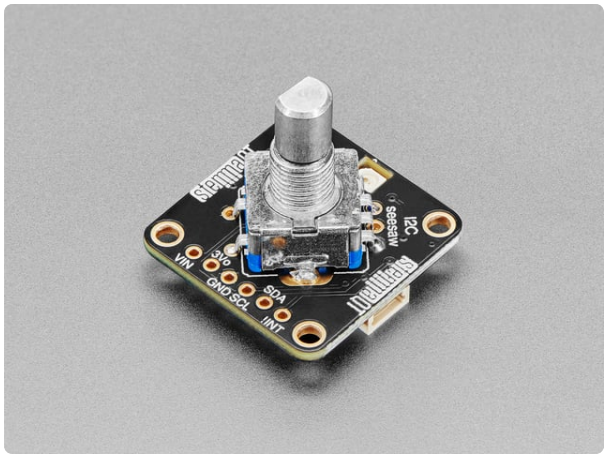


[Adafruit RP2350 22-pin FPC HSTX to DVI Adapter for HDMI Displays]
You may have noticed that our RP2350 Feather has an FPC output connector on the end for accessing the HSTX (High Speed...
https://www.adafruit.com/product/6055

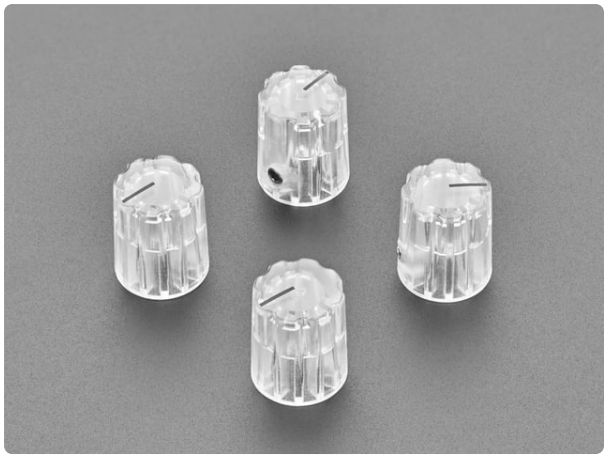



[Adafruit PDM Microphone Breakout with JST SH Connector]
An exotic new microphone has arrived in the Adafruit shop, a PDM MEMS Microphone! PDM is the 'third' kind of microphone you can integrate with electronics,...
https://www.adafruit.com/product/4346

[Panel Mount 10K Log Potentiometer (Breadboard Friendly)]
This potentiometer is a two-in-one, good in a breadboard or with a panel. It's a log taper 10K ohm potentiometer, with a grippy shaft. It's smooth and easy to...
https://www.adafruit.com/product/3391




[Adafruit I2C Stemma QT Rotary Encoder Breakout with Encoder]
Rotary encoders are soooo much fun! Twist em this way, then twist them that way. Unlike potentiometers, they go all the way around and often have little detents for tactile feedback....
https://www.adafruit.com/product/5880



Clear Micro Potentiometer Knob - 4 pack
Have you found yourself in a maze of twisty-turny knobs, all alike? These micro potentiometer (or rotary encoder) knobs offer a rainbow burst of color...
https://www.adafruit.com/product/5676

**1 x** FPC Flex Cable for DSI CSI or HSTX
22-pin 0.5mm pitch - 10 cm
https://www.adafruit.com/product/6035

**1 x** STEMMA QT / Qwiic JST SH Cable
Premium Male Headers Cable
https://www.adafruit.com/product/4209

**1 x** STEMMA QT / Qwiic JST SH 4-pin Cable
100mm Long
https://www.adafruit.com/product/4210

**1 x** Silicone Cover Stranded-Core Wire
30AWG in Various Colors
https://www.adafruit.com/product/2051

**1 x** Black Nylon Machine Screw and Stand-off Set
M2.5 Thread
https://www.adafruit.com/product/3299

**1 x** USB A to USB C Cable
2 meters long
https://www.adafruit.com/product/5044

**1 x** HDMI Cable
3 feet long
https://www.adafruit.com/product/2421

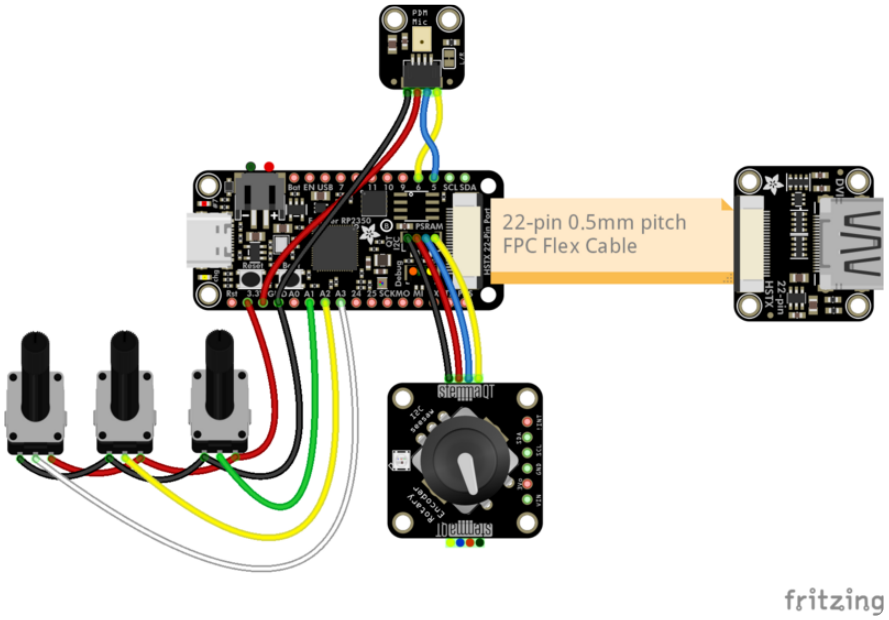
**1 x** 7" Display 1280x800 (720p)
HDMI
https://www.adafruit.com/product/1667

## Fastener Hardware

The following hardware screws and hex nuts will be needed to assemble the 3D printed enclosure

- 10x M2.5 x 6mm button head machine screws
- 2x M2.5 x 8mm button head machine screws
- 2x M2.5 hex nuts

# Circuit Diagram




## Potentiometers

- **Pot 1 wiper** to **Feather pin A1 (green wire)**
- **Pot 2 wiper** to **Feather pin A2 (yellow wire)**
- **Pot 3 wiper** to **Feather pin A3 (white wire)**
- **Pot 1, 2 and 3 positive pin** to **Feather 3.3V (red wire)**
- **Pot 1, 2 and 3 GND** to **Feather GND (black wire)**

## seesaw Rotary Encoder

- **Rotary encoder STEMMA QT port** to **Feather STEMMA QT port (STEMMA QT cable)**

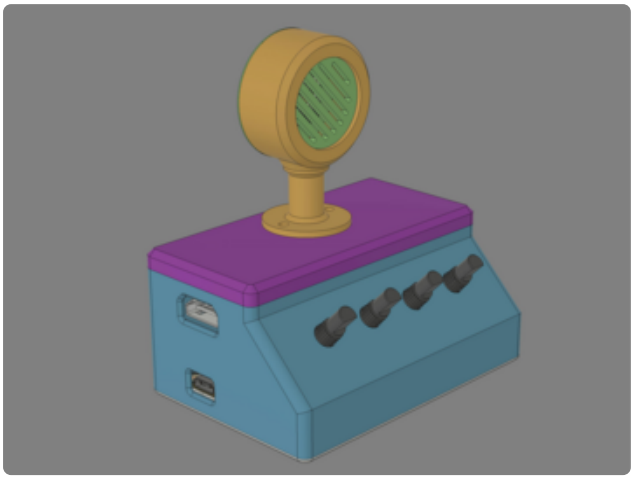
## PDM Mic

- **Mic GND** to **Feather GND (black wire)**
- **Mic VDD** to **Feather 3.3V (red wire)**
- **Mic CLK** to **Feather pin D6 (yellow wire)**
- **Mic DAT** to **Feather pin D5 (blue wire)**

## HSTX to DVI

- **Feather HSTX port** to **DVI breakout HSTX port (FPC cable)**

# 3D Printing



## 3D Printed Parts

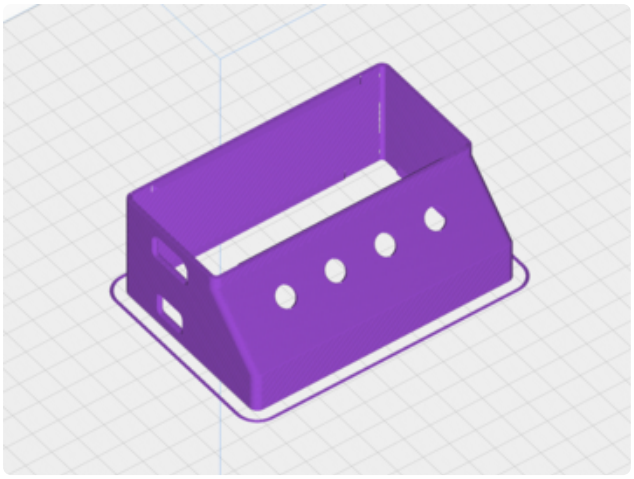
STL files for 3D printing are oriented to print "as-is" on FDM style machines. Parts are designed to 3D print without any support material using PLA filament. Original design source may be downloaded using the links below.

**Download CAD Source**

https://adafru.it/1aag

**Download STLs.zip**

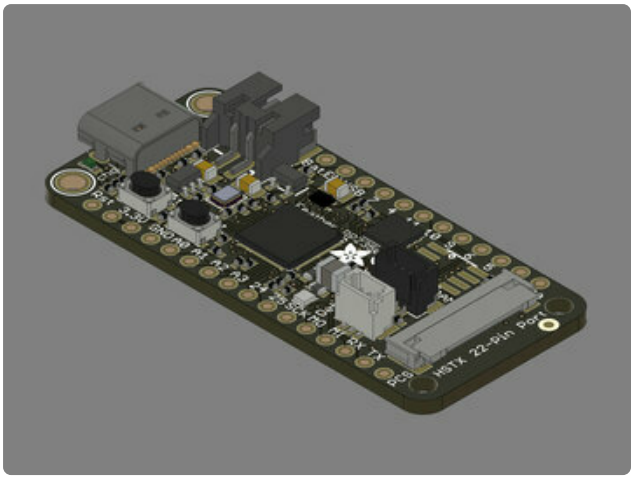
https://adafru.it/1aah



## Build Volume

The parts require a 3D printer with a minimum build volume of:

100mm (X) x 68mm (Y) x 40mm (Z)



## Design Source Files

The project assembly was designed in Fusion 360. This can be downloaded in different formats like STEP, STL and more.

Electronic components like Adafruit's boards, displays, connectors and more can be downloaded from the Adafruit CAD parts GitHub Repo (https://adafru.it/RvF).

---

# Install CircuitPython

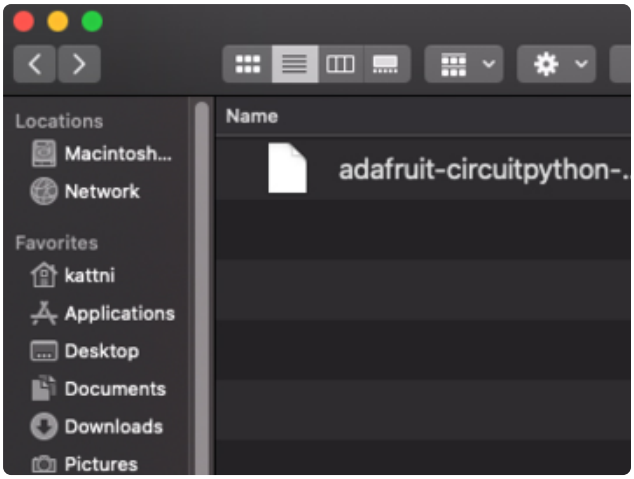
CircuitPython (https://adafru.it/tB7) is a derivative of MicroPython (https://adafru.it/BeZ) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files on the **CIRCUITPY** drive to iterate.

## CircuitPython Quickstart

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

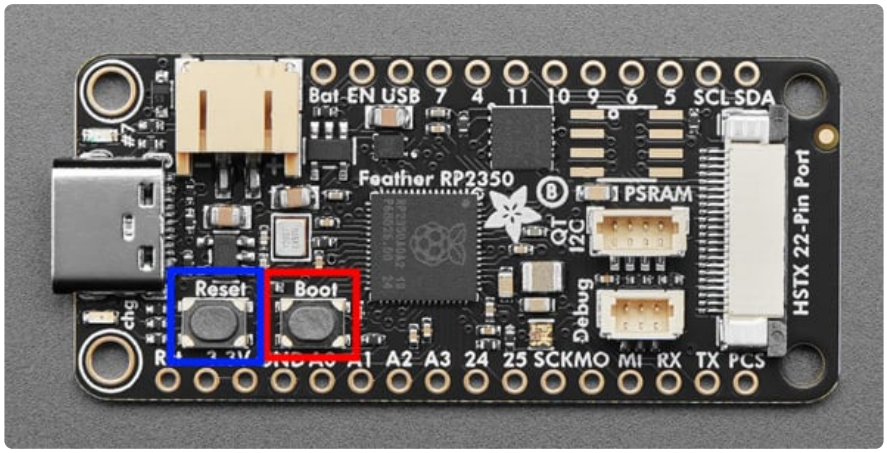
**Download the latest version of CircuitPython for this board via circuitpython.org**

https://adafru.it/1a7f




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

Save it wherever is convenient for you.

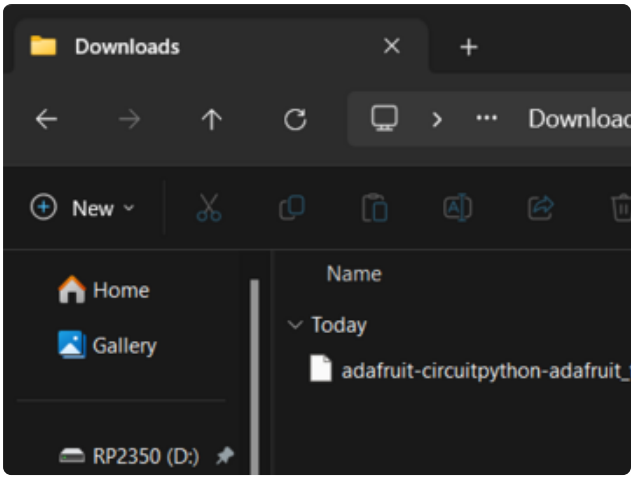



To enter the bootloader, hold down the **BOOT/BOOTSEL button** (highlighted in red above), and while continuing to hold it (don't let go!), press and release the **reset button** (highlighted in red or blue above). **Continue to hold the BOOT/BOOTSEL button until the RP2350 drive appears!**

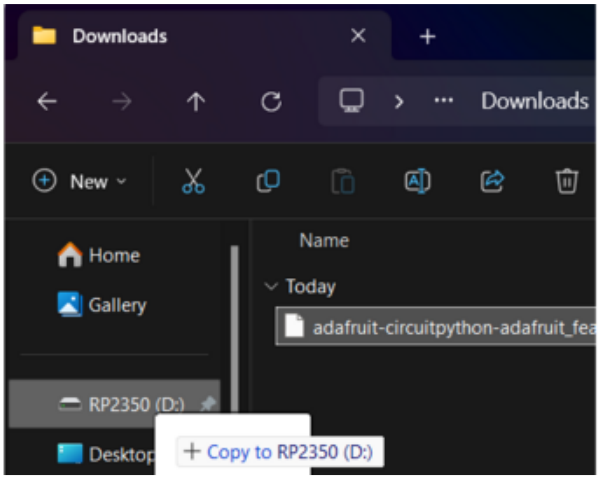
If the drive does not appear, release all the buttons, and then repeat the process above.

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

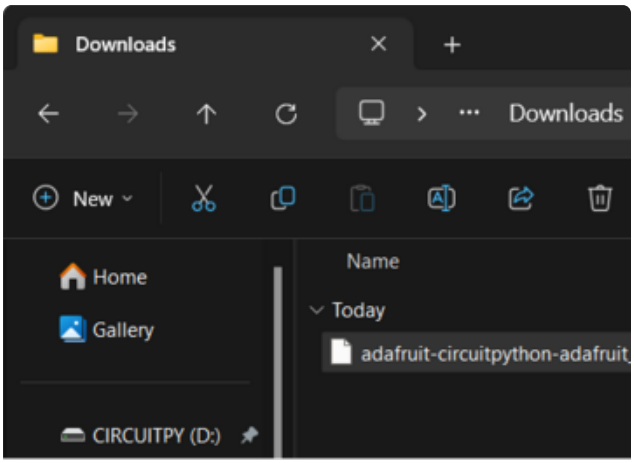
A lot of people end up using charge-only USB cables and it is very frustrating! **Make sure you have a USB cable you know is good for data sync.**




You will see a new disk drive appear called **RP2350**.




Drag the **adafruit_circuitpython_etc.uf2** file to **RP2350.**




The **RP2350** drive will disappear and a new disk drive called **CIRCUITPY** will appear.

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

## Safe Mode

You want to edit your **code.py** or modify the files on your **CIRCUITPY** drive, but find that you can't. Perhaps your board has gotten into a state where **CIRCUITPY** is read-only. You may have turned off the **CIRCUITPY** drive altogether. Whatever the reason, safe mode can help.

Safe mode in CircuitPython does not run any user code on startup, and disables auto-reload. This means a few things. First, safe mode bypasses any code in **boot.py** (where you can set **CIRCUITPY** read-only or turn it off completely). Second, it does

not run the code in **code.py**. And finally, it does not automatically soft-reload when data is written to the **CIRCUITPY** drive.

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

### Entering Safe Mode

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

### In Safe Mode

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

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

```
Auto-reload is off.
Running in safe mode! Not running saved code.

CircuitPython is in safe mode because you pressed the reset button during boot.
Press again to exit safe mode.

Press any key to enter the REPL. Use CTRL-D to reload.
```

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

## Flash Resetting UF2

If your board ever gets into a really weird state and CIRCUITPY doesn't show up as a disk drive after installing CircuitPython, try loading this 'nuke' UF2 to RP2350. which will do a 'deep clean' on your Flash Memory. **You will lose all the files on the board**, but at least you'll be able to revive it! After loading this UF2, follow the steps above to re-install CircuitPython.

**Download flash erasing "nuke" UF2**

https://adafru.it/1a7p

# Code the Video Synth

Once you've finished setting up your Feather RP2350 with CircuitPython, you can access the code and necessary libraries by downloading the Project Bundle.

To do this, click on the **Download Project Bundle** button in the window below. It will download to your computer as a zipped folder.

```
# SPDX-FileCopyrightText: 2024 Liz Clark for Adafruit Industries
#
# SPDX-License-Identifier: MIT

# FFT calculations based on Phil B.'s Audio Spectrum Light Show Code
# https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/
# Feather_Sense_Audio_Visualizer_13x9_RGB_LED_Matrix/audio_spectrum_lightshow/
code.py
import gc
from array import array
from math import log
from random import randint
import board
from audiobusio import PDMIn
import displayio
import picodvi
import framebufferio
import vectorio
from rainbowio import colorwheel
import adafruit_imageload
from adafruit_display_shapes.line import Line
from adafruit_ticks import ticks_ms, ticks_add, ticks_diff
from adafruit_seesaw import seesaw, rotaryio, digitalio
from analogio import AnalogIn
import simpleio
from ulab import numpy as np
try:
    from ulab.utils import spectrogram
except ImportError:
    from ulab.scipy.signal import spectrogram

# ------ POTENTIOMETER SETUP ------
pot1 = AnalogIn(board.A1)
pot2 = AnalogIn(board.A2)
pot3 = AnalogIn(board.A3)
read_pots = True
# ------ POTENTIOMETER MODE VALUES ------
mode0_values = [0, 0, 0]
mode1_values = [0, 0, 0]
mode2_values = [0, 0, 0]
mode_vals = [mode0_values, mode1_values, mode2_values]
# ------ POTENTIOMETER READ FUNCTION ------
def val(pin):
    return pin.value

# ------ SEESAW ENCODER ------
i2c = board.STEMMA_I2C()
seesaw = seesaw.Seesaw(i2c, addr=0x36)
seesaw_product = (seesaw.get_version() >> 16) & 0xFFFF
print("Found product {}".format(seesaw_product))
if seesaw_product != 4991:
    print("Wrong firmware loaded?  Expected 4991")
seesaw.pin_mode(24, seesaw.INPUT_PULLUP)
button = digitalio.DigitalIO(seesaw, 24)
button_held = False
```

```
encoder = rotaryio.IncrementalEncoder(seesaw)
last_position = 0

# ------ SHARED VARIABLES ------
fft_size = 512
low_bin = 15
high_bin = 75
low_band = (15, 75)
mid_band = (100, 120)
spectrum_size = fft_size // 2
spectrum_bits = log(spectrum_size, 2)
mode = 0
new_mode = True
states = {}

# ------ PICODVI SETUP ------
displayio.release_displays()
fb = picodvi.Framebuffer(320, 240, clk_dp=board.CKP, clk_dn=board.CKN,
                         red_dp=board.D0P, red_dn=board.D0N,
                         green_dp=board.D1P, green_dn=board.D1N,
                         blue_dp=board.D2P, blue_dn=board.D2N,
                         color_depth=8)
display = framebufferio.FramebufferDisplay(fb, auto_refresh=False)

# ------ PDM MIC ------
mic = PDMIn(board.D5, board.D6, sample_rate=44100, bit_depth=16)
rec_buf = array("H", [0] * fft_size)

# pylint: disable=too-many-locals, global-statement, too-many-statements, global-
variable-not-assigned

# ------ INITIALIZE BAR GRAPH ANIMATION ------
def initialize_bars():
    # based on Phil B.'s Audio Spectrum Light Show Code
    # https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/
    # Feather_Sense_Audio_Visualizer_13x9_RGB_LED_Matrix/audio_spectrum_lightshow/
code.py
    global states
    spectrum_group = displayio.Group()
    display.root_group = spectrum_group

    low_frac = log(low_bin, 2) / spectrum_bits
    frac_range = log(high_bin, 2) / spectrum_bits - low_frac
    num_columns = 16
    column_width = display.width // num_columns
    column_table = []
    moving_avg_buffer = [display.height] * num_columns
    smoothing_factor = 0.5
    height_multiplier = 5
    dynamic_level = 10
    noise_floor = 3.1

    for column in range(num_columns):
        lower = low_frac + frac_range * (column / num_columns * 0.95)
        upper = low_frac + frac_range * ((column + 1) / num_columns)
        mid = (lower + upper) * 0.5
        half_width = (upper - lower) * 0.5
        first_bin = int(2 ** (spectrum_bits * lower) + 1e-4)
        last_bin = int(2 ** (spectrum_bits * upper) + 1e-4)
        bin_weights = []
        for bin_index in range(first_bin, last_bin + 1):
            bin_center = log(bin_index + 0.5, 2) / spectrum_bits
            dist = abs(bin_center - mid) / half_width
            if dist < 1.0:
                dist = 1.0 - dist
                bin_weights.append(((3.0 - (dist * 2.0)) * dist) * dist)
        total = sum(bin_weights)
        bin_weights = [
            (weight / total) * (0.8 + idx / num_columns * 1.4)
```

```
            for idx, weight in enumerate(bin_weights)
        ]
        column_table.append(
            [
                first_bin - low_bin,
                bin_weights,
                colorwheel(225 * column / num_columns),
                display.height,
                0.0,
            ]
        )
        bar_pal = displayio.Palette(1)
        bar_pal[0] = colorwheel(225 * column / num_columns)
        rect = vectorio.Rectangle(
            pixel_shader=bar_pal,
            width=column_width,
            height=1,
            x=column * column_width,
            y=display.height,
        )
        spectrum_group.append(rect)

    peak_palette = displayio.Palette(1)
    peak_palette[0] = 0x808080
    for column in range(num_columns):
        column_table[column].append(display.height)
        column_table[column].append(0.0)
        peak_dot = vectorio.Rectangle(
            pixel_shader=peak_palette,
            width=column_width,
            height=5,
            x=column * column_width,
            y=display.height,
        )
        spectrum_group.append(peak_dot)

    states = {
        "spectrum_group": spectrum_group,
        "column_table": column_table,
        "moving_avg_buffer": moving_avg_buffer,
        "smoothing_factor": smoothing_factor,
        "height_multiplier": height_multiplier,
        "dynamic_level": dynamic_level,
        "noise_floor": noise_floor,
        "num_columns": num_columns,
        "column_width": column_width,
        "peak_pal": peak_palette,
    }

# ------ BAR GRAPH ANIMATION ------
def bars(pos, read):
    global states
    if read:
        states["peak_pal"][0] = colorwheel(pos[0])
        states["noise_floor"] = pos[1]
        states["smoothing_factor"] = pos[2]
    spectrum_group = states["spectrum_group"]
    column_table = states["column_table"]
    moving_avg_buffer = states["moving_avg_buffer"]
    smoothing_factor = states["smoothing_factor"]
    height_multiplier = states["height_multiplier"]
    dynamic_level = states["dynamic_level"]
    noise_floor = states["noise_floor"]
    num_columns = states["num_columns"]

    mic.record(rec_buf, fft_size)
    samples = np.array(rec_buf)

    spectrum = spectrogram(samples)[low_bin : high_bin + 1]
```

```
    spectrum = np.log(spectrum + 1e-7)
    spectrum = np.maximum(spectrum - noise_floor, 0)
    lower = max(np.min(spectrum), 4)
    upper = min(max(np.max(spectrum), lower + 12), 20)
    if upper > dynamic_level:
        dynamic_level = upper * 0.7 + dynamic_level * 0.3
    else:
        dynamic_level = dynamic_level * 0.5 + lower * 0.5
    states["dynamic_level"] = dynamic_level
    max_height = display.height
    data = (spectrum - lower) * (max_height / (dynamic_level - lower)) *
height_multiplier

    for column, element in enumerate(column_table):
        first_bin = element[0]
        column_top = display.height + 1
        for bin_offset, weight in enumerate(element[1]):
            if first_bin + bin_offset < len(data):
                column_top -= data[first_bin + bin_offset] * weight
        column_top = max(0, int(column_top))
        moving_avg_buffer[column] = (
            moving_avg_buffer[column] * (1 - smoothing_factor) +
            column_top * smoothing_factor
        )
        smoothed_top = int(moving_avg_buffer[column])
        rect = spectrum_group[column]
        rect.height = display.height - smoothed_top
        rect.y = smoothed_top

        if smoothed_top < element[3]:
            element[3] = smoothed_top - 1
            element[4] = 0
        else:
            element[3] += element[4]
            element[4] += 0.2
        peak_position = max(0, int(element[3]))
        peak_dot = spectrum_group[num_columns + column]
        peak_dot.y = peak_position
    display.refresh()
# ------ INITIALIZE CIRCLE ANIMATION ------
def initialize_circles():
    global states
    spectrum_group = displayio.Group()
    display.root_group = spectrum_group
    palette = displayio.Palette(1)
    palette[0] = 0xFFFFFF
    center_palette = displayio.Palette(1)
    center_palette[0] = 0xFF0000
    center_circle = (
    vectorio.Circle(pixel_shader=center_palette, radius=5,
                    x=display.width // 2, y=display.height // 2)
    )
    spectrum_group.append(center_circle)
    num_circles = 16
    smoothing_factor = 0.5
    decay_factor = 0.2
    circles = []
    directions = []
    smoothed_radii = [5] * num_circles

    for i in range(num_circles):
        radius = 5
        x_pos = randint(radius, display.width - radius)
        y_pos = randint(radius, display.height - radius)
        dx = randint(-2, 2) or 1
        dy = randint(-2, 2) or 1
        color_index = (255 * i) // num_circles
        circle_palette = displayio.Palette(1)
        circle_palette[0] = colorwheel(color_index)
```

```
        circle = vectorio.Circle(pixel_shader=circle_palette, radius=radius,
x=x_pos, y=y_pos)
        spectrum_group.append(circle)
        circles.append(circle)
        directions.append([dx, dy])
    states = {
        "spectrum_group": spectrum_group,
        "center_circle": center_circle,
        "center_radius_smoothed": 5,
        "circles": circles,
        "directions": directions,
        "smoothed_radii": smoothed_radii,
        "smoothing_factor": smoothing_factor,
        "decay_factor": decay_factor,
        "dynamic_level": 15,
        "noise_floor": 3.1,
        "max_radius": 100,
        "num_circles": num_circles,
        "center_palette": center_palette,
        "speed": 1,
    }
# ------ BOUNCING CIRCLES ANIMATION ------
def bouncing_circles(pos, read):
    global states
    if read:
        states["center_palette"][0] = colorwheel(pos[0])
        states["noise_floor"] = pos[1]
        states["smoothing_factor"] = pos[2]
        states["decay_factor"] = pos[2] - 0.3
    mic.record(rec_buf, fft_size)
    samples = np.array(rec_buf)
    spectrum = spectrogram(samples)[low_bin : high_bin + 1]
    spectrum = np.log(spectrum + 1e-7)
    spectrum = np.maximum(spectrum - states["noise_floor"], 0)
    lower = max(np.min(spectrum), 4)
    upper = min(max(np.max(spectrum), lower + 12), 20)
    if upper > states["dynamic_level"]:
        states["dynamic_level"] = upper * 0.7 + states["dynamic_level"] * 0.3
    else:
        states["dynamic_level"] = states["dynamic_level"] * 0.5 + lower * 0.5
    overall_amplitude = np.sum(spectrum)
    max_center_radius = 50
    target_center_radius = (
    int((overall_amplitude / (states["dynamic_level"] * states["num_circles"])) *
max_center_radius)
    )
    if target_center_radius < states["center_radius_smoothed"]:
        states["center_radius_smoothed"] = (
            states["center_radius_smoothed"] * (1 - states["decay_factor"]) +
            target_center_radius * states["decay_factor"]
        )
    else:
        states["center_radius_smoothed"] = (
            states["center_radius_smoothed"] * (1 - states["smoothing_factor"]) +
            target_center_radius * states["smoothing_factor"]
        )
    states["center_circle"].radius = int(states["center_radius_smoothed"])
    data = (spectrum - lower) * (states["max_radius"] / (states["dynamic_level"] -
lower))
    for i, circle in enumerate(states["circles"]):
        target_radius = max(2, int(data[i]))
        if target_radius < states["smoothed_radii"][i]:
            states["smoothed_radii"][i] = (
                states["smoothed_radii"][i] * (1 - states["decay_factor"]) +
                target_radius * states["decay_factor"]
            )
        else:
            states["smoothed_radii"][i] = (
                states["smoothed_radii"][i] * (1 - states["smoothing_factor"]) +
```

```
                target_radius * states["smoothing_factor"]
            )
        circle.radius = int(states["smoothed_radii"][i])
        dx, dy = states["directions"][i]
        circle.x += dx
        circle.y += dy
        if circle.x - circle.radius <= 0 or circle.x + circle.radius >=
display.width:
            states["directions"][i][0] *= -1  # Reverse x direction
        if circle.y - circle.radius <= 0 or circle.y + circle.radius >=
display.height:
            states["directions"][i][1] *= -1 # Reverse y direction
    display.refresh()
# ------ PARTY PARROT INIT ------
def initialize_party():
    global states
    spectrum_group = displayio.Group()
    display.root_group = spectrum_group
    bitmap, palette = adafruit_imageload.load(
        "/partyParrotsBig.bmp",
        bitmap=displayio.Bitmap,
        palette=displayio.Palette
    )
    parrot_grid = displayio.TileGrid(
        bitmap,
        pixel_shader=palette,
        tile_height=128,
        tile_width=132,
        x=(display.width - 128) // 2,
        y=0
    )
    spectrum_group.append(parrot_grid)
    line_group = displayio.Group()
    spectrum_group.append(line_group)
    pal_bg = displayio.Palette(1)
    pal_bg[0] = 0x0000FF
    palette_white = displayio.Palette(1)
    palette_white[0] = 0xFFFFFF
    pal = displayio.Palette(1)
    pal[0] = 0xFF00FF
    left_circle = vectorio.Circle(
        pixel_shader=pal, radius=5, x=5, y=5
    )
    right_circle = vectorio.Circle(
        pixel_shader=pal, radius=5, x=display.width - 5, y=5
    )
    spectrum_group.append(left_circle)
    spectrum_group.append(right_circle)
    ground = vectorio.Rectangle(
        pixel_shader=pal_bg,
        width=display.width,
        height=display.height - 128,
        x=0,
        y=128
    )
    line_group.append(ground)
    horizon_line = vectorio.Rectangle(
        pixel_shader=palette_white,
        width=display.width,
        height=1,
        x=0,
        y=128
    )
    line_group.append(horizon_line)
    slanted_lines_coords = [
        (0, 136, 34, 128),
        (0, 188, 76, 128),
        (34, 240, 113, 128),
        (117, 240, 148, 128),
```

```
        (198, 240, 182, 128),
        (294, 240, 216, 128),
        (320, 176, 255, 128),
        (320, 133, 297, 128)
    ]
    for coords in slanted_lines_coords:
        line = Line(coords[0], coords[1], coords[2], coords[3], 0xFFFFFF)
        line_group.append(line)
    horizontal_lines = []
    for _ in range(5):
        line_rect = vectorio.Rectangle(
            pixel_shader=palette_white,
            width=display.width,
            height=2,
            x=0,
            y=0
        )
        line_group.append(line_rect)
        horizontal_lines.append(line_rect)
    states = {
        "pal": pal,
        "pal_bg": pal_bg,
        "pal_line": palette_white,
        "spectrum_group": spectrum_group,
        "parrot_grid": parrot_grid,
        "line_group": line_group,
        "horizontal_lines": horizontal_lines,
        "frame_index": 0,
        "low_band_threshold": 3.4,
        "mid_band_threshold": 3.4,
        "smoothing_factor": 0.5,
        "last_i": 146,
        "dynamic_level": 15,
        "decay_factor": 2,
        "max_center_radius": 10,
        "center_radius_smoothed": 2,
        "left_circle": left_circle,
        "right_circle": right_circle,
        "clock_clock": ticks_ms(),
        "clock_time": int(0.01 * 1000),
        "i": 152
    }
# ------ PARTY PARROT ANIMATION ------
def party_parrot(pos, read):
    global states
    if read:
        states["pal"][0] = colorwheel(pos[0])
        states["max_center_radius"] = pos[1]
        states["low_band_threshold"] = pos[2]
        states["mid_band_threshold"] = pos[2]
    left_circle = states["left_circle"]
    right_circle = states["right_circle"]
    parrot_grid = states["parrot_grid"]
    horizontal_lines = states["horizontal_lines"]
    frame_index = states["frame_index"]
    last_i = states["last_i"]
    center_radius_smoothed = states["center_radius_smoothed"]
    dynamic_level = states["dynamic_level"]
    low_band_threshold = states["low_band_threshold"]
    mid_band_threshold = states["mid_band_threshold"]
    smoothing_factor = states["smoothing_factor"]
    decay_factor = states["decay_factor"]
    clock_clock = states["clock_clock"]
    palette_blue = states["pal_bg"]
    palette_white = states["pal_line"]
    i = states["i"]
    if i > 128:
        last_i = i
        i -= 1
```

```
    else:
        i = 152
    mic.record(rec_buf, fft_size)
    samples = np.array(rec_buf)
    spectrum = spectrogram(samples)
    spectrum = np.log(spectrum + 1e-7)
    spectrum = np.maximum(spectrum - 3.1, 0)
    low_band_avg = np.mean(spectrum[low_band[0]:low_band[1] + 1])
    mid_band_avg = np.mean(spectrum[mid_band[0]:mid_band[1] + 1])
    overall_amplitude = np.sum(spectrum)
    target_center_radius = (
    int((overall_amplitude / (dynamic_level * 16)) * states["max_center_radius"])
    )
    if target_center_radius < center_radius_smoothed:
        center_radius_smoothed = (
        center_radius_smoothed * (1 - decay_factor) + target_center_radius *
decay_factor
        )
    else:
        center_radius_smoothed = (
        center_radius_smoothed * (1 - smoothing_factor) + target_center_radius *
smoothing_factor
        )
    left_circle.radius = int(center_radius_smoothed)
    right_circle.radius = int(center_radius_smoothed)
    if low_band_avg >= low_band_threshold * 1.1 or mid_band_avg >=
mid_band_threshold * 1.1:
        frame_index = (frame_index + 1) % 10
        parrot_grid[0] = frame_index
    if ticks_diff(ticks_ms(), clock_clock) >= states["clock_time"]:
        for idx, offset in enumerate([0, 25, 50, 75, 100]):
            horizontal_lines[idx].y = last_i + offset
            horizontal_lines[idx].pixel_shader = palette_blue
        for idx, offset in enumerate([0, 25, 50, 75, 100]):
            horizontal_lines[idx].y = i + offset
            horizontal_lines[idx].pixel_shader = palette_white
        clock_clock = ticks_add(clock_clock, states["clock_time"])
    display.refresh()
    states["frame_index"] = frame_index
    states["last_i"] = last_i
    states["i"] = i
    states["center_radius_smoothed"] = center_radius_smoothed
# ------ THE LOOP ------
while True:
    # read encoder - if value != then change mode
    position = -encoder.position
    if position != last_position:
        if position > last_position:
            mode = (mode + 1) % 3
        else:
            mode = (mode - 1) % 3
        new_mode = True
        last_position = position
    # encode button - switch between reading potentiometer
    # to control animations or using default values
    if not button.value and not button_held:
        button_held = True
        read_pots = not read_pots
        print("Button pressed")
    if button.value and button_held:
        button_held = False
        print("Button released")
    # if a new mode is selected, run init function
    if new_mode:
        new_mode = False
        print(f"switching modes! {mode}")
        del states
        gc.collect()
        display.refresh()
```

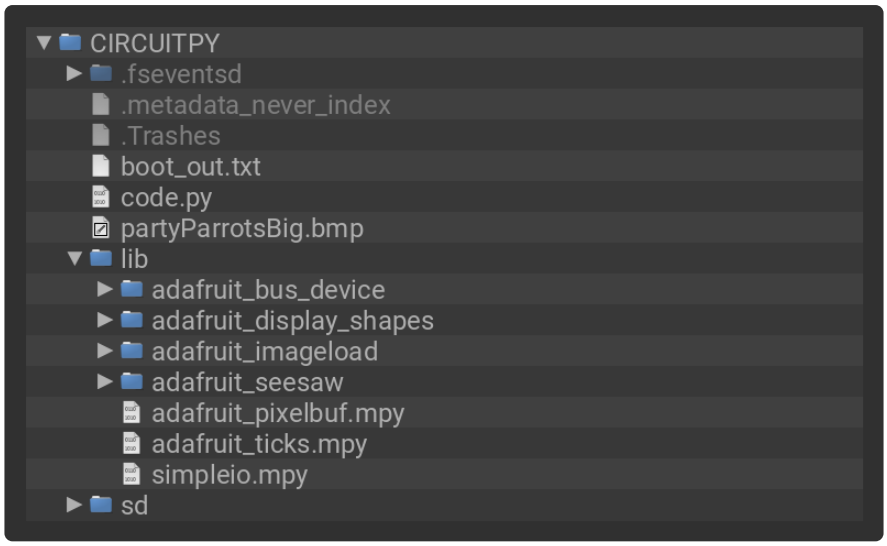
```
        if mode == 0:
            initialize_bars()
        if mode == 1:
            initialize_circles()
        if mode == 2:
            initialize_party()
    # mode 0 - bar graph visualizer
    if mode == 0:
        bars(mode0_values, read_pots)
        # pot 1 - control color of bouncing dot
        color = simpleio.map_range(val(pot1), 0, 65535, 0, 255)
        mode_vals[0][0] = color
        # pot 2 - raise/lower noise floor
        noise = simpleio.map_range(val(pot2), 0, 65535, 2.5, 4.5)
        mode_vals[0][1] = noise
        # pot 3 - smooth bar animation
        smooth = simpleio.map_range(val(pot3), 0, 65535, 0.5, 0.05)
        mode_vals[0][2] = smooth
    # mode 1 - bouncing circles visualizer
    if mode == 1:
        bouncing_circles(mode1_values, read_pots)
        # pot 1 - control color of center circle
        color = simpleio.map_range(val(pot1), 0, 65535, 0, 255)
        mode_vals[1][0] = color
        # pot 2 - raise/lower noise floor
        noise = simpleio.map_range(val(pot2), 0, 65535, 2.5, 4.5)
        mode_vals[1][1] = noise
        # pot 3 - smooth circle size change
        smooth = simpleio.map_range(val(pot3), 0, 65535, 0.8, 0.4)
        mode_vals[1][2] = smooth
    # mode 2 - party parrot synth wave
    if mode == 2:
        party_parrot(mode2_values, read_pots)
        # pot 1 - control color of side circles
        color = simpleio.map_range(val(pot1), 0, 65535, 0, 255)
        mode_vals[2][0] = color
        # pot 2 - mid & low end noise floor aka parrot sensitivity
        dynamic = simpleio.map_range(val(pot2), 0, 65535, 3.0, 6.0)
        mode_vals[2][1] = dynamic
        # pot 3 - size of side circles
        rad = simpleio.map_range(val(pot3), 0, 65535, 2, 12)
        mode_vals[2][2] = rad
```

## Upload the Code, Bitmap and Libraries to the Feather RP2350

After downloading the Project Bundle, plug your Feather RP2350 into the computer's USB port with a known good USB data+power cable. You should see a new flash drive appear in the computer's File Explorer or Finder (depending on your operating system) called **CIRCUITPY**. Unzip the folder and copy the following items to the Feather RP2350's **CIRCUITPY** drive.

- **lib** folder
- **code.py**
- **partyParrotsBig.bmp**

Your Feather RP2350 **CIRCUITPY** drive should look like this after copying the **lib** folder, bitmap and the **code.py** file.




## Code Inspiration

The code for this project uses fast Fourier transform (FFT) (https://adafru.it/qMC) to sample the incoming audio over time and turn the samples into a list of the different frequencies that are present in the sound. The foundation for the video synth was based on the Mini LED Matrix Audio Visualizer project code by Phil B (https://adafru.it/1aai)., which uses FFT for a bar graph audio visualizer on an LED matrix. This animation was ported to this project and laid the foundation for the bouncing circles animation and the party parrot animation.

Mini LED Matrix Audio Visualizer
By Liz Clark
Code the Audio Spectrum Light Show

https://learn.adafruit.com/mini-led-matrix-audio-visualizer/code-the-audio-spectrum-light-show

## How the Code Works

The three potentiometers are instantiated as analog inputs. The values from the potentiometers determine different visual parameters for each of the three animations (modes). These values are stored in lists called `mode#_values[]`. These three lists are then stored in the `mode_vals[]` list.

```
# ------ POTENTIOMETER SETUP ------
pot1 = AnalogIn(board.A1)
pot2 = AnalogIn(board.A2)
pot3 = AnalogIn(board.A3)
read_pots = True
```

```
# ------ POTENTIOMETER MODE VALUES ------
mode0_values = [0, 0, 0]
mode1_values = [0, 0, 0]
mode2_values = [0, 0, 0]
mode_vals = [mode0_values, mode1_values, mode2_values]
# ------ POTENTIOMETER READ FUNCTION ------
def val(pin):
    return pin.value
```

## seesaw Rotary Encoder

The mode is changed with the seesaw rotary encoder. The button on the rotary encoder controls the state of `read_pots`. If `read_pots` is `True`, then the potentiometers control parameters in the animations. If its `False`, then the default parameters run and the potentiometers do not control any of the parameters.

```
# ------ SEESAW ENCODER ------
i2c = board.STEMMA_I2C()
seesaw = seesaw.Seesaw(i2c, addr=0x36)
seesaw_product = (seesaw.get_version() &gt;&gt; 16) &amp; 0xFFFF
print("Found product {}".format(seesaw_product))
if seesaw_product != 4991:
    print("Wrong firmware loaded?  Expected 4991")
seesaw.pin_mode(24, seesaw.INPUT_PULLUP)
button = digitalio.DigitalIO(seesaw, 24)
button_held = False
encoder = rotaryio.IncrementalEncoder(seesaw)
last_position = 0
```

## PicoDVI

The PicoDVI framebuffer is setup using the HSTX pins and the PDM mic is instantiated along with its recording buffer.

```
# ------ PICODVI SETUP ------
displayio.release_displays()
fb = picodvi.Framebuffer(320, 240, clk_dp=board.CKP, clk_dn=board.CKN,
                         red_dp=board.D0P, red_dn=board.D0N,
                         green_dp=board.D1P, green_dn=board.D1N,
                         blue_dp=board.D2P, blue_dn=board.D2N,
                         color_depth=8)
display = framebufferio.FramebufferDisplay(fb, auto_refresh=False)

# ------ PDM MIC ------
mic = PDMIn(board.D5, board.D6, sample_rate=44100, bit_depth=16)
rec_buf = array("H", [0] * fft_size)
```

## Variables and States

A few shared variables are created that are used across all three animations.

```
# ------ SHARED VARIABLES ------
fft_size = 512
low_bin = 15
high_bin = 75
low_band = (15, 75)
mid_band = (100, 120)
```

```
spectrum_size = fft_size // 2
spectrum_bits = log(spectrum_size, 2)
mode = 0
new_mode = True
states = {}
```

## The Animations

There are three animations: bars, bouncing circles and party parrots. Each animation has two associated functions: an initialization and the actual animation. The initialization is run once when the mode is changed to populate a dictionary with needed values. The actual animation function runs a single frame of the animation and runs during an associated mode. Mode 0 is bars, 1 is bouncing circles and 2 is party parrots.

## The Loop

In the loop, the rotary encoder is polled to control the value of `mode`. The value is set to either `0`, `1` or `2`.

```
# ------ THE LOOP ------
while True:
    # read encoder - if value != then change mode
    position = -encoder.position
    if position != last_position:
        if position &gt; last_position:
            mode = (mode + 1) % 3
        else:
            mode = (mode - 1) % 3
        new_mode = True
        last_position = position
```

The button on the rotary encoder controls the state of `read_pots`. This state affects whether the animations are controlled by the three analog potentiometers or if they use their default values.

```
# encode button - switch between reading potentiometer
    # to control animations or using default values
    if not button.value and not button_held:
        button_held = True
        read_pots = not read_pots
        print("Button pressed")
    if button.value and button_held:
        button_held = False
        print("Button released")
```

If the rotary encoder selects a new mode, then the associated initialize function is run to start up the new animation.

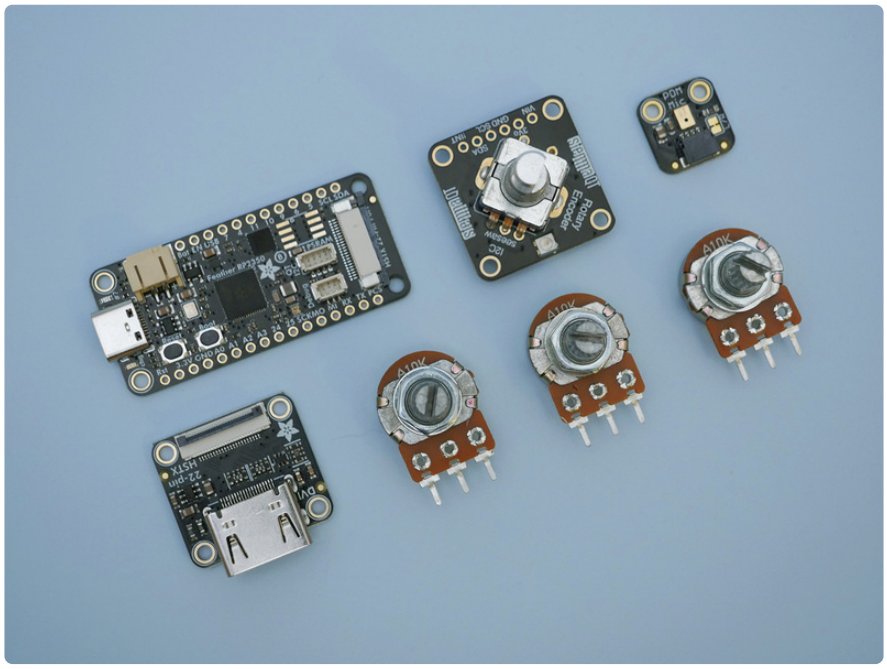
```
# if a new mode is selected, run init function
    if new_mode:
        new_mode = False
        print(f"switching modes! {mode}")
        del states
```

```
        gc.collect()
        display.refresh()
        if mode == 0:
            initialize_bars()
        if mode == 1:
            initialize_circles()
        if mode == 2:
            initialize_party()
```

While the `mode` is set, the associated animation runs. Each potentiometer is mapped to a specific set of values. The list of values and the state of `read_pots` is passed to each animation function.

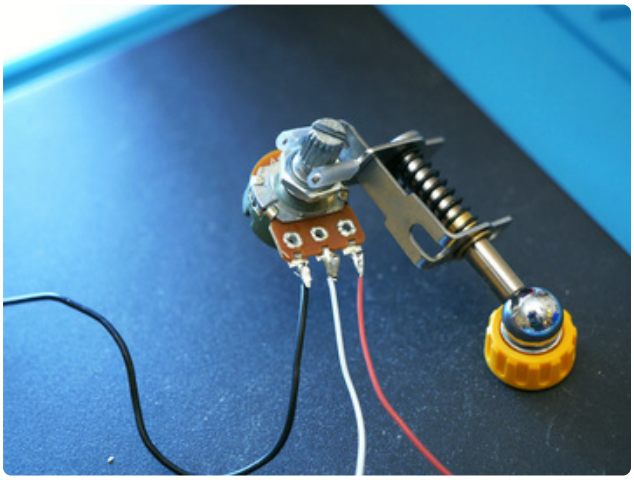
```
# mode 0 - bar graph visualizer
    if mode == 0:
        bars(mode0_values, read_pots)
        # pot 1 - control color of bouncing dot
        color = simpleio.map_range(val(pot1), 0, 65535, 0, 255)
        mode_vals[0][0] = color
        # pot 2 - raise/lower noise floor
        noise = simpleio.map_range(val(pot2), 0, 65535, 2.5, 4.5)
        mode_vals[0][1] = noise
        # pot 3 - smooth bar animation
        smooth = simpleio.map_range(val(pot3), 0, 65535, 0.5, 0.05)
        mode_vals[0][2] = smooth
    # mode 1 - bouncing circles visualizer
    if mode == 1:
        bouncing_circles(mode1_values, read_pots)
        # pot 1 - control color of center circle
        color = simpleio.map_range(val(pot1), 0, 65535, 0, 255)
        mode_vals[1][0] = color
        # pot 2 - raise/lower noise floor
        noise = simpleio.map_range(val(pot2), 0, 65535, 2.5, 4.5)
        mode_vals[1][1] = noise
        # pot 3 - smooth circle size change
        smooth = simpleio.map_range(val(pot3), 0, 65535, 0.8, 0.4)
        mode_vals[1][2] = smooth
    # mode 2 - party parrot synth wave
    if mode == 2:
        party_parrot(mode2_values, read_pots)
        # pot 1 - control color of side circles
        color = simpleio.map_range(val(pot1), 0, 65535, 0, 255)
        mode_vals[2][0] = color
        # pot 2 - mid &amp;amp; low end noise floor aka parrot sensitivity
        dynamic = simpleio.map_range(val(pot2), 0, 65535, 3.0, 6.0)
        mode_vals[2][1] = dynamic
        # pot 3 - size of side circles
        rad = simpleio.map_range(val(pot3), 0, 65535, 2, 12)
        mode_vals[2][2] = rad
```

# Wiring

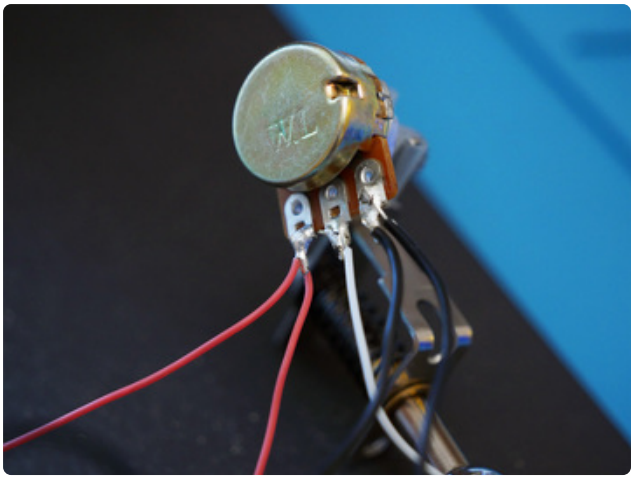


The potentiometers are soldered in the reverse order from how they will be mounted in the case, starting with potentiometer 3.

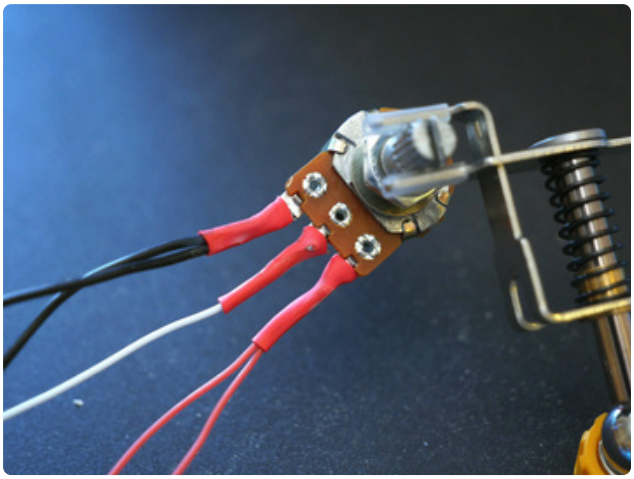
## Potentiometer 3



Cut, splice and tin three pieces of wire. Solder a piece of wire to the positive, wiper and GND pins on the potentiometer.

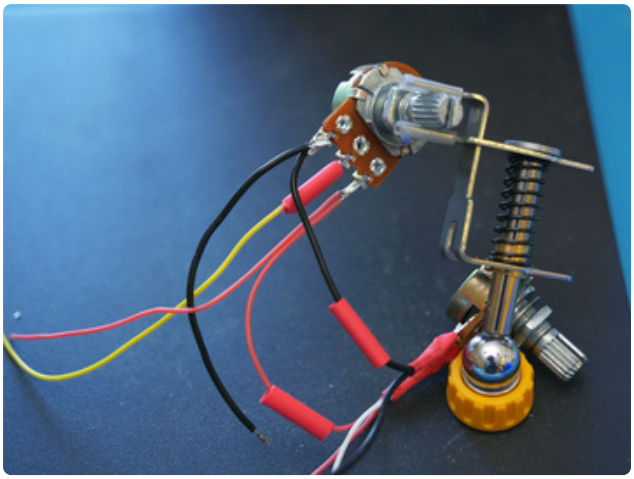


Cut, splice and tin two more pieces of wire. Solder a piece to the positive pin and the other to the GND pin. These two wires will daisy chain the power and GND connections to the next potentiometer.

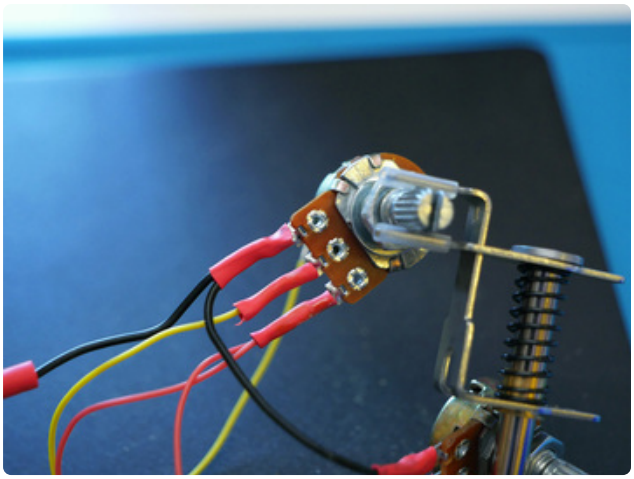


Apply heat shrink to the three solder joints.

## Potentiometer 2

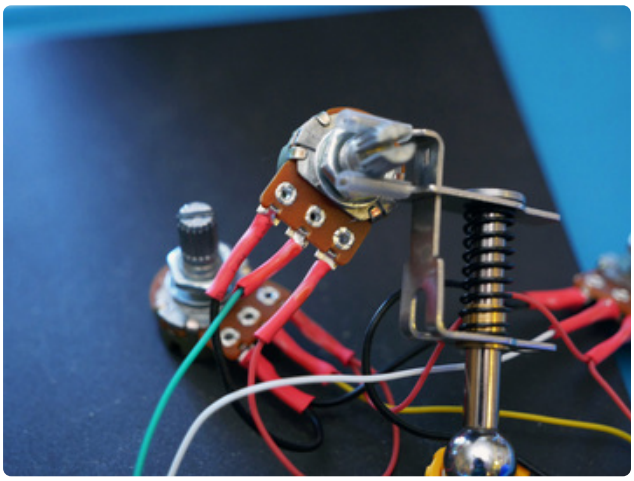


1. Slide a piece of heat shrink onto the daisy chain GND and power wires **before** soldering them to potentiometer 2.
2. Then, solder the daisy chain GND wire to the GND pin and the power wire to the positive pin.
3. Solder a wire to the wiper pin.
4. Solder a new piece of wire to the positive pin and another new piece to the GND pin. These will bring power and ground to potentiometer 1.



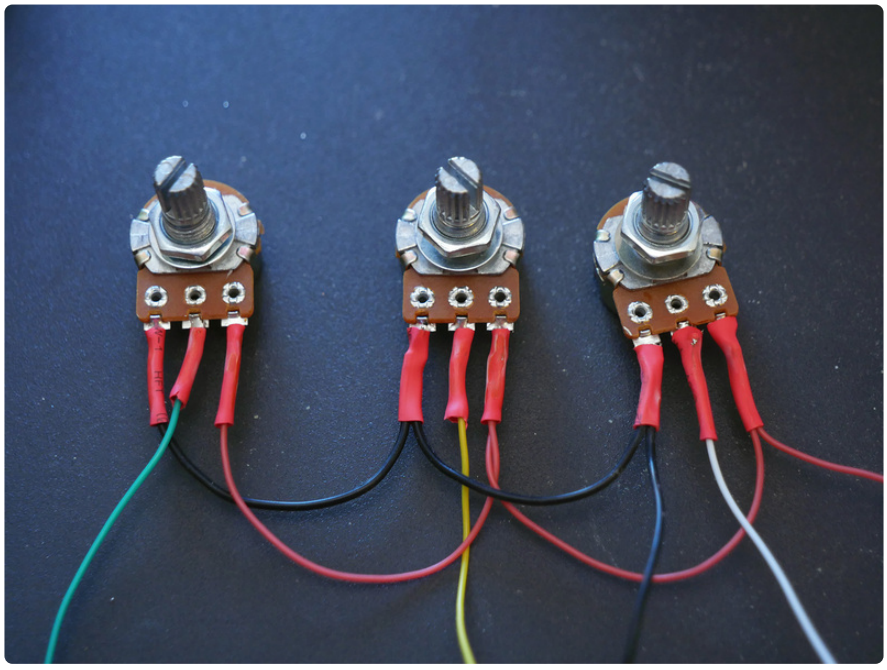
Slip the two new wires into the heat shrink for the positive and GND pins. Apply heat shrink to all three pins.

## Potentiometer 1

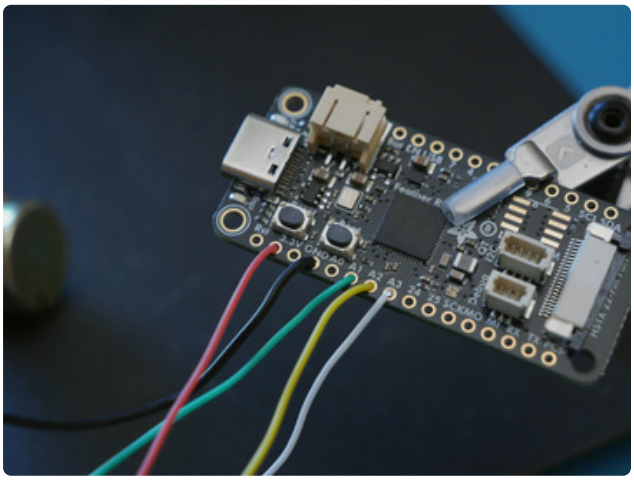


1. Slide a piece of heat shrink onto the daisy chain GND and power wires **before** soldering them to potentiometer 1.
2. Solder the GND wire to the GND pin and the power wire to the positive pin.
3. Solder a wire to the wiper pin.
4. Apply heat shrink to all three pins.

Your three potentiometers should look like the photo below with daisy chained power and ground connections between them.

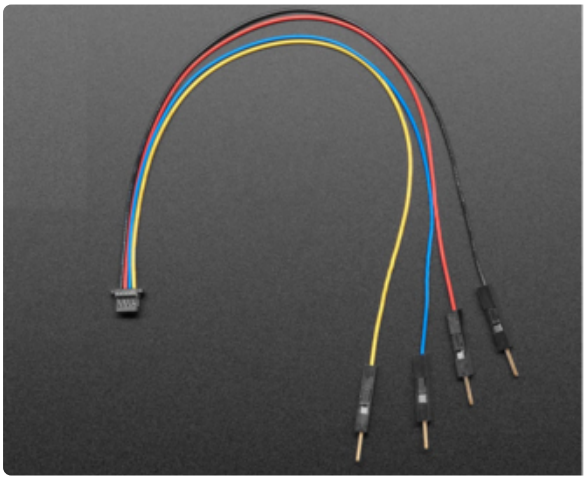


## Potentiometers to Feather Wiring

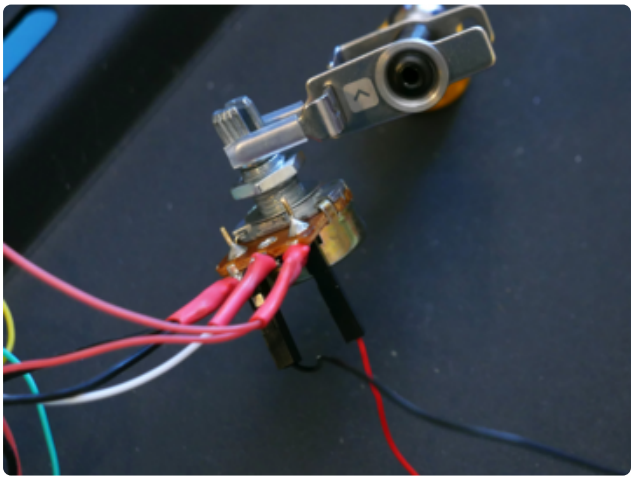


1. Solder the power input wire from potentiometer 1 to 3.3V on the Feather (red wire).
2. Solder the GND input wire from potentiometer 1 to GND on the Feather (black wire).
3. Solder potentiometer 1 wiper pin to Feather pin A1 (green wire).
4. Solder potentiometer 2 wiper pin to Feather pin A2 (yellow wire).
5. Solder potentiometer 3 wiper pin to Feather pin A3 (white wire).

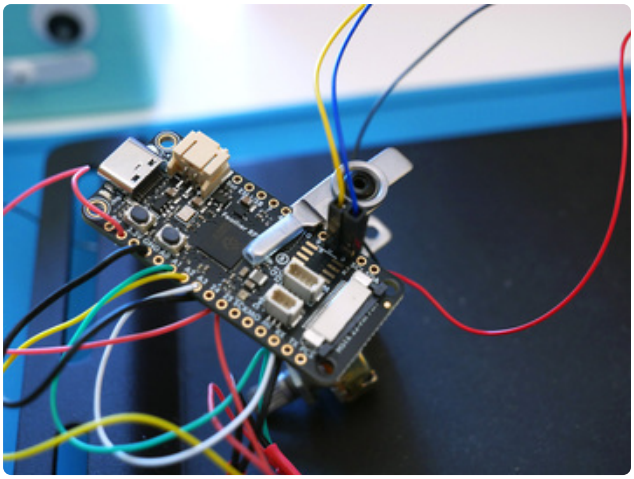
## PDM STEMMA Cable



The PDM mic will connect to the Feather using a STEMMA QT to plug cable. **Note that the mic does not plug into the STEMMA port!**



Solder the red plug wire from the STEMMA QT cable to the positive line on potentiometer 3. Solder the black plug wire from the STEMMA QT cable to the GND line on potentiometer 3. You can use the round solder pads as a solder point.

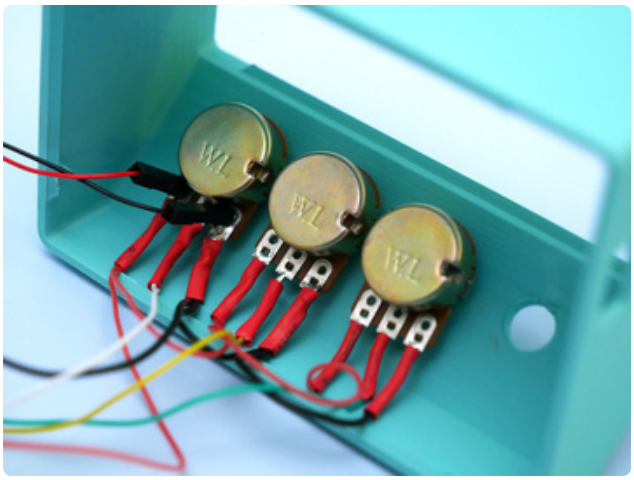


Solder the yellow plug wire from the STEMMA QT cable to pin D6 on the Feather. Solder the blue plug wire from the STEMMA QT cable to pin D5 on the Feather.

---

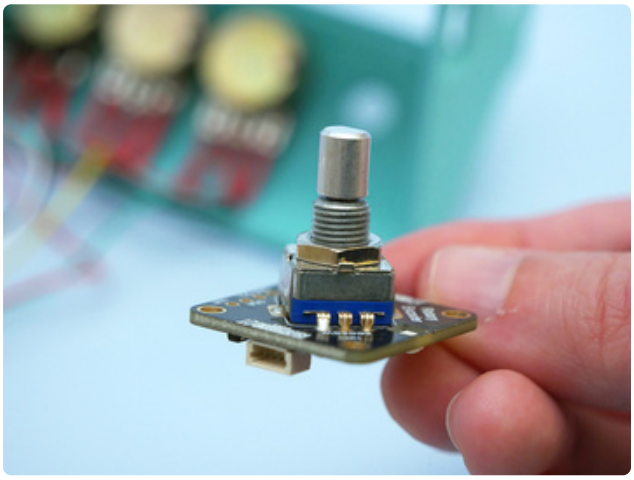
# Assembly

## Mount the Potentiometers

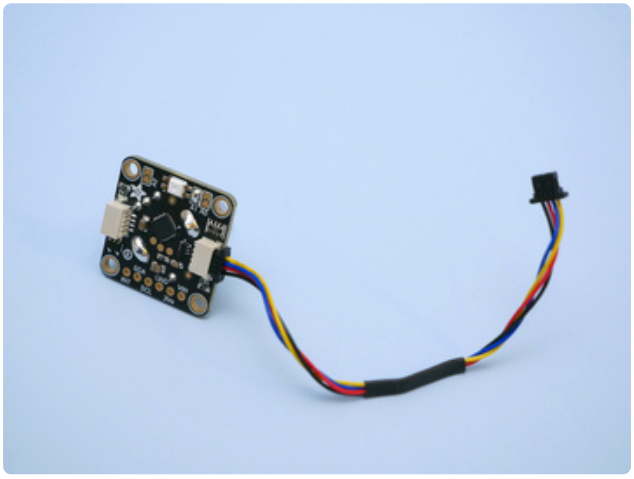


Mount the three potentiometers into the main body of the case. Potentiometer 1 should be mounted in the second hole from the left, potentiometer 2 in the third hole from the left and potentiometer 3 in the last hole on the right.

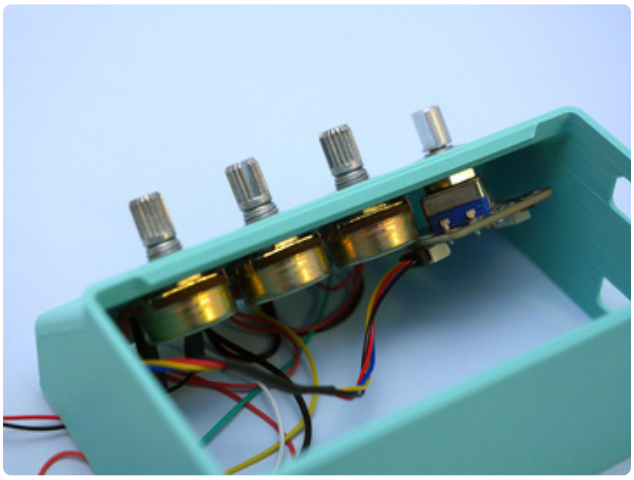
## Rotary Encoder



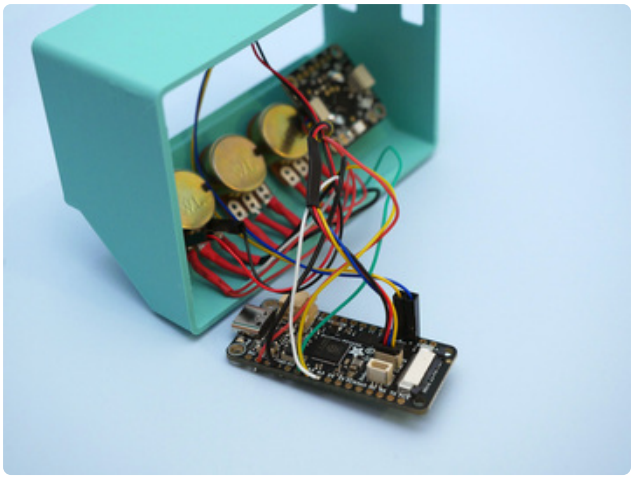
Add a nut to the rotary encoder shaft. This will act as a spacer when mounting it in the case so that it does not interfere with the potentiometer.



Plug a STEMMA QT cable into the seesaw rotary encoder.



Mount the rotary encoder into the remaining mounting hole on the case.

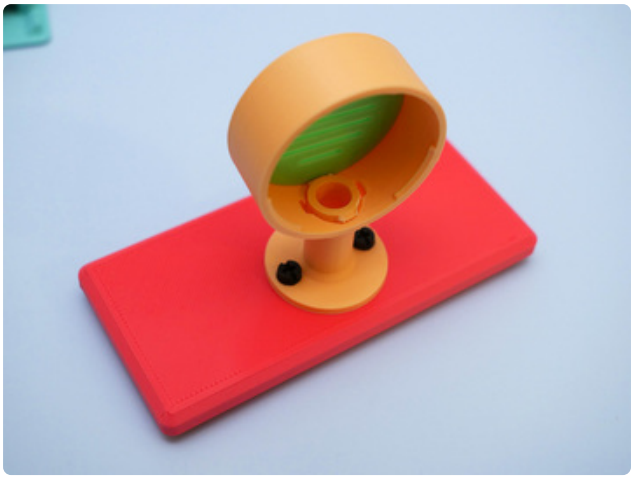


Plug the rotary encoder STEMMA QT cable into the STEMMA port on the Feather.

## Case Lid and Microphone Enclosure

Attach the microphone base to the case lid with two M2.5 screws and nuts.

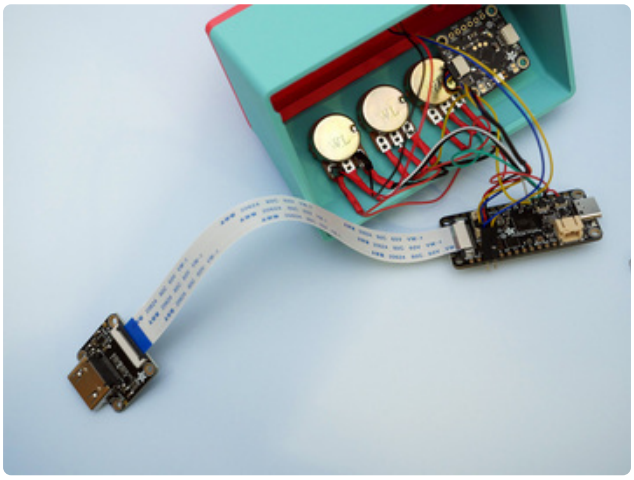
Insert the microphone cover into the microphone case.



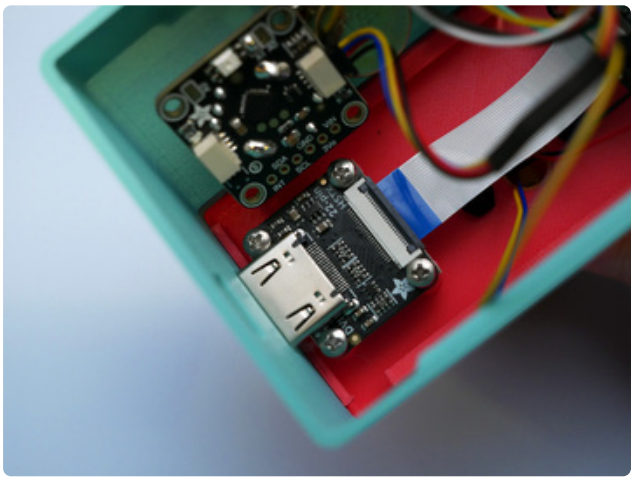
Attach the microphone case to the microphone base by placing it over the notches and turning it so that it is facing out towards the longer edge of the case lid.

Feed the microphone STEMMA cable up through the microphone base and plug it into the PDM mic.

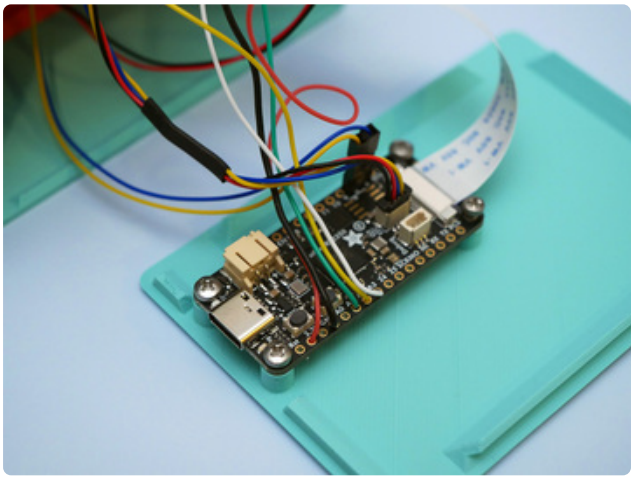
## HSTX to DVI Breakout



Attach the HSTX to DVI breakout to the Feather with the 22-pin FPC flex cable. The breakout end of the cable should have the blue side facing up. The Feather end of the cable should have the blue side facing down.

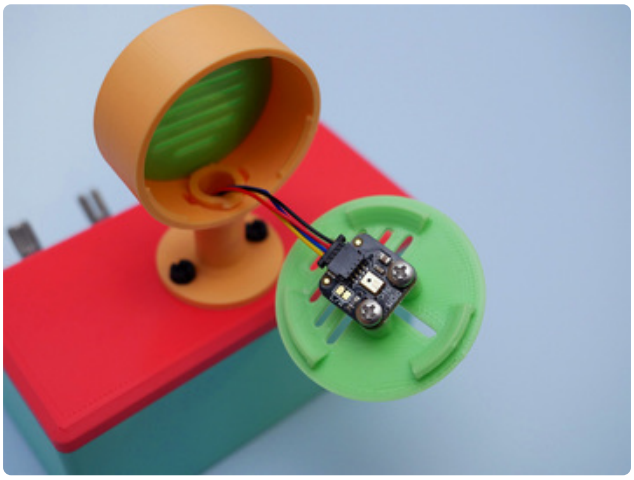


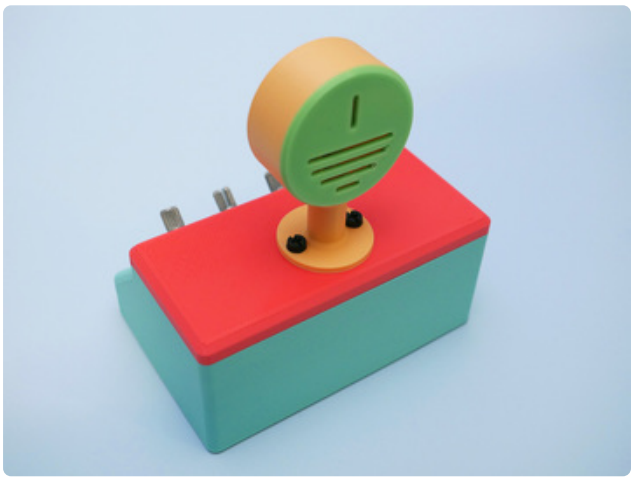
Attach the HSTX to DVI breakout into the standoffs on the lid of the case with four M2.5 screws.



Attach the Feather to the mounting holes on the bottom lid of the case with four M2.5 screws.

## PDM Mic





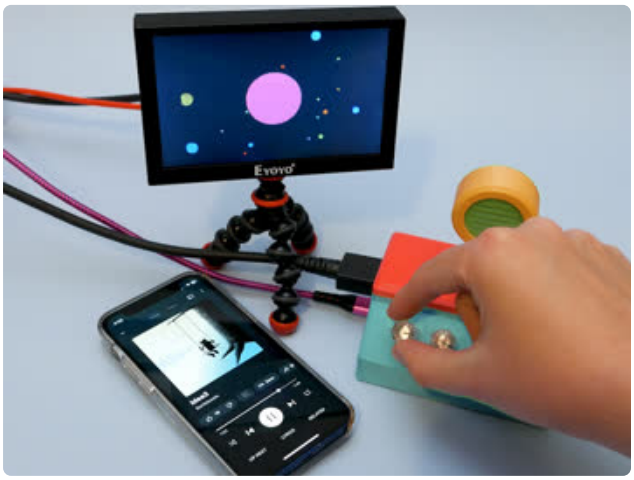
Attach the PDM mic to the back lid of the microphone enclosure with two M2.5 screws. Close the microphone enclosure by snapping in the back lid.

## Final Touches

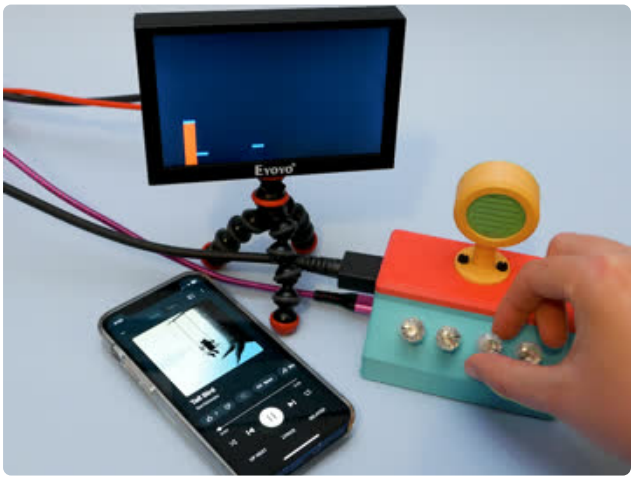
Finish up the assembly by adding some potentiometer knobs to the rotary encoder and potentiometers.

# Use

After assembly, power the Feather via the USB input and plug the DVI output into a DVI/HDMI monitor with an HDMI cable. When it powers up, you should see the bouncing bar animation on the display.

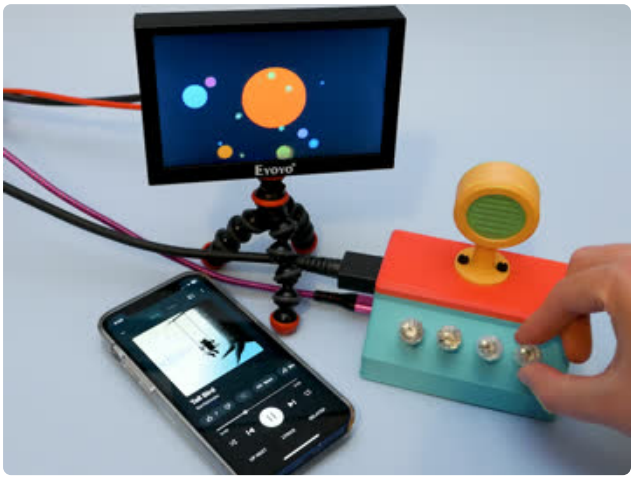


Use the rotary encoder to change between the three animation modes.



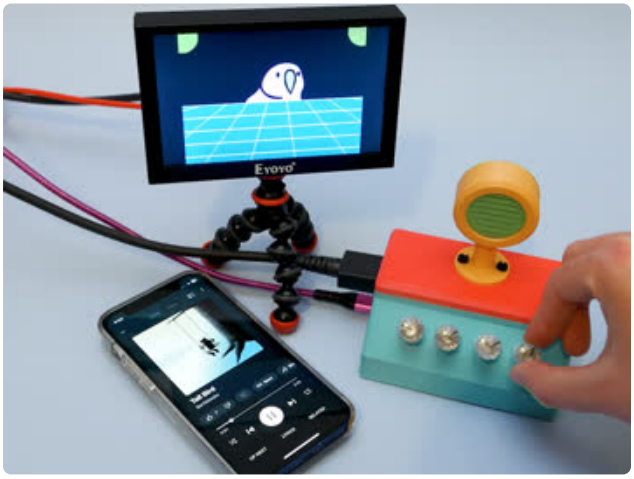
The bouncing bar animation is an audio EQ visualizer. The 16 bars are associated with 16 different frequency bands.

The first potentiometer changes the color of the bouncing dot at the top of the bars. The second potentiometer controls the noise floor aka the minimum amplitude that reads an an input with the PDM mic. The third potentiometer smooths the animation. You can have the bars be very jumpy as they react to the audio or do more of a slide.



The bouncing circles animation builds on the bouncing bar animation. 16 circles bounce around the display. Each circle is assigned to an audio frequency. When that frequency is read, the circle increases in size depending on how loud it is. The circle in the center changes size depending on the overall amplitude.

The first potentiometer changes the color of the center circle. The second potentiometer controls the noise floor threshold for the animation. The third potentiometer affects how quickly the circles go back to their default small size.



The third and final animation is the party parrot synthwave vibe. A party parrot sits atop a scrolling synthwave inspired grid landscape. In the upper corners, two circles change shape depending on the overall amplitude of the incoming audio. The party parrot has a twist: it advances frames when a midband and/or low end frequency are registered. These two frequencies are often where snare drums and bass drums sit in a music mix. As a result, the party parrot dances with the percussion.

The first potentiometer changes the color of the corner circles. The second potentiometer controls the noise floor threshold for the party parrot frequencies. If you turn it all the way down, the party parrot will automatically advance frames and basically play as an animated sprite. The third potentiometer affects the maximum size of the corner circles.