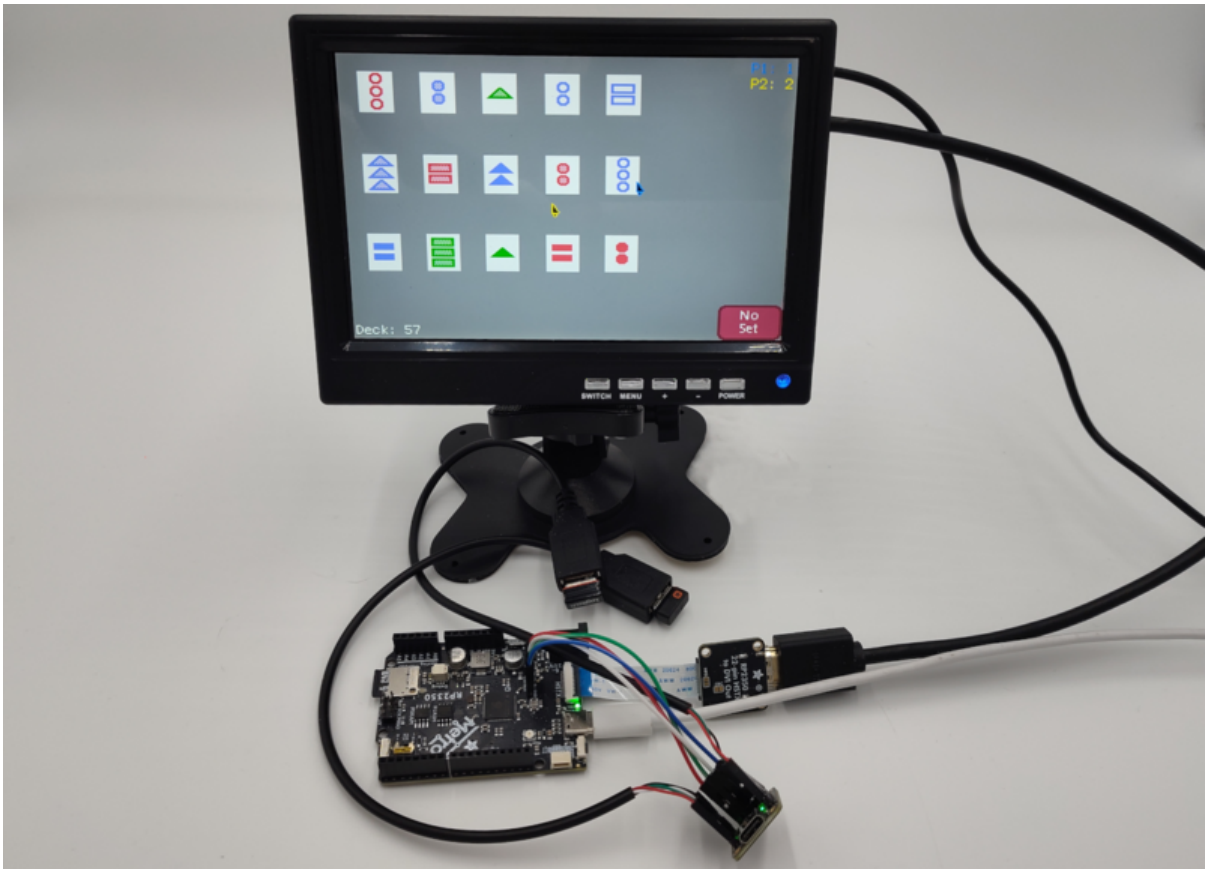
Match3 Game on the Adafruit Metro RP2350

Created by Tim C

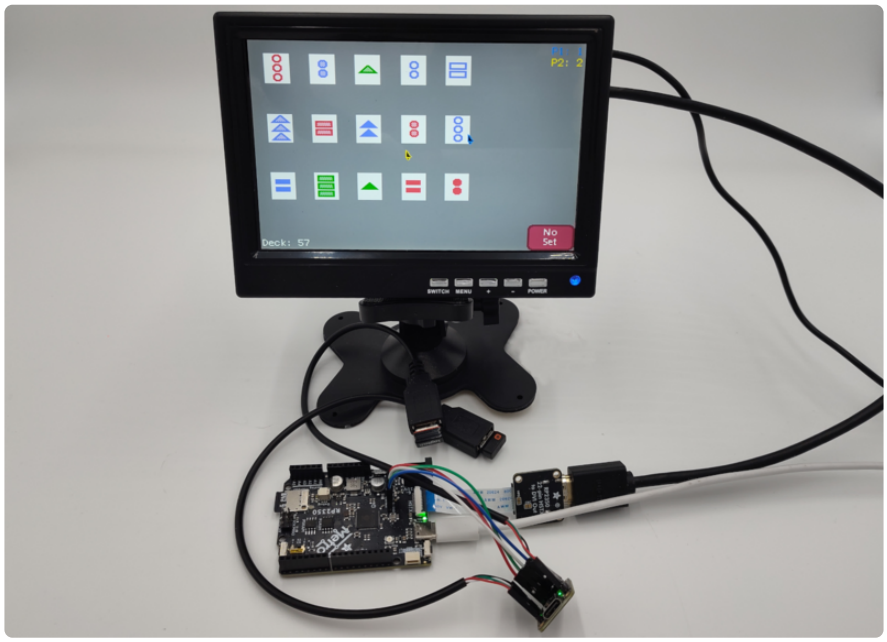


https://learn.adafruit.com/match3-game-on-metro-rp2350

Last updated on 2025-05-20 10:49:55 AM EDT

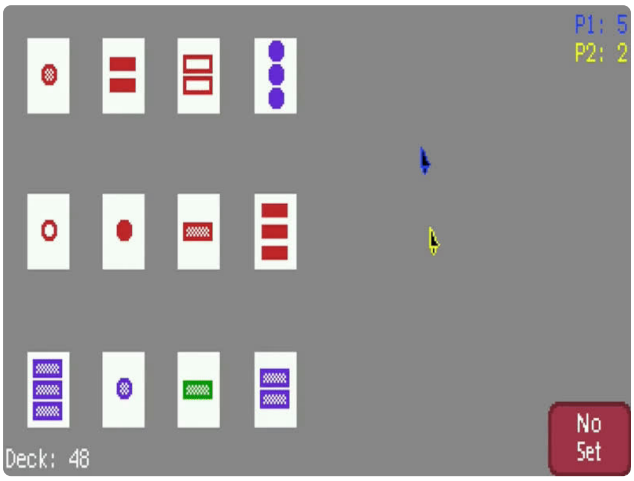
Table of Contents

Overview



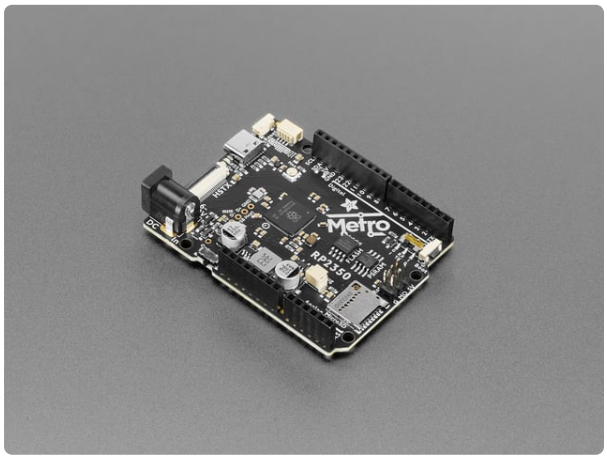
This CircuitPython project brings a game inspired by the Set card game (https://adafru.it/1ahx) to the Metro RP2350. A CH334F USB Hub (http://adafru.it/5999) allows two mice to be connected to the Metro at once.

The game can be displayed on any HDMI compatible display using the HSTX ribbon connector on the Metro and a DVI breakout.

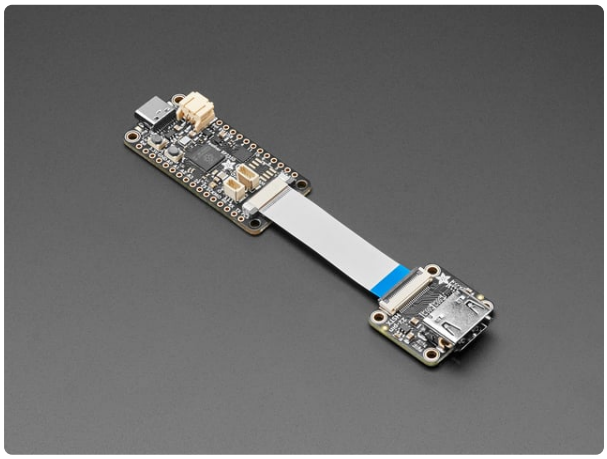


Players face off with each other searching for sets of 3 cards that either all match or all differ in each of four different properties: shape, color, count, and fill. Players use right click on their mouse to buzz in, declaring that they've found a set. Then they click on the 3 cards they've found. If it was a valid set the cards are removed, the player is awarded a point and more cards are dealt. If all players indicate that they cannot find any sets then new cards are dealt.

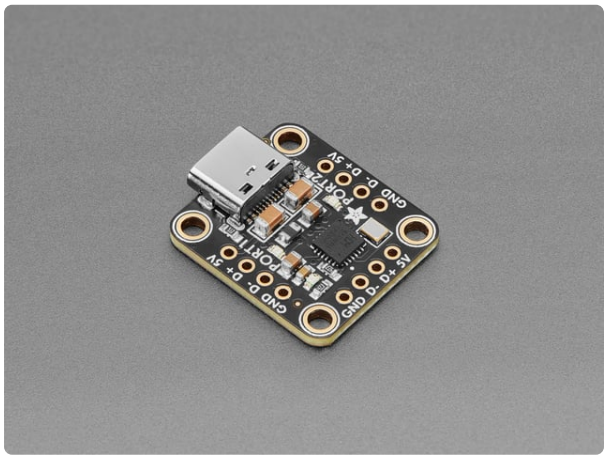
Parts



Adafruit Metro RP2350 with PSRAM
Choo! Choo! This is the RP2350 Metro Line, making all station stops at "Dual Cortex M33 mountain", "528K RAM round-about" and "16 Megabytes of Flash town"...
https://www.adafruit.com/product/6267

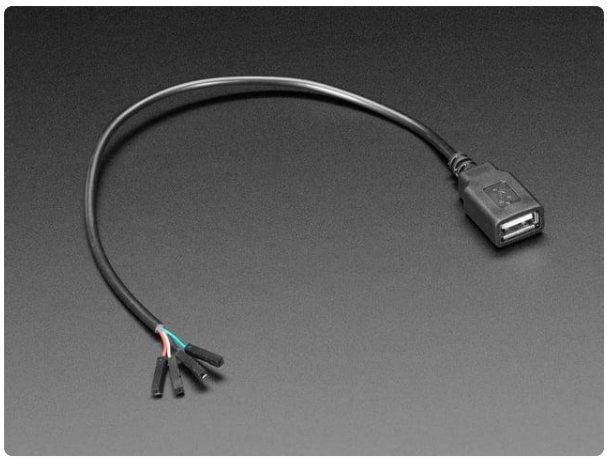


Adafruit RP2350 22-pin FPC HSTX to DVI Adapter for HDMI Displays
You may have noticed that our RP2350 Feather has an FPC output connector on the end for accessing the HSTX (High Speed...
https://www.adafruit.com/product/6055

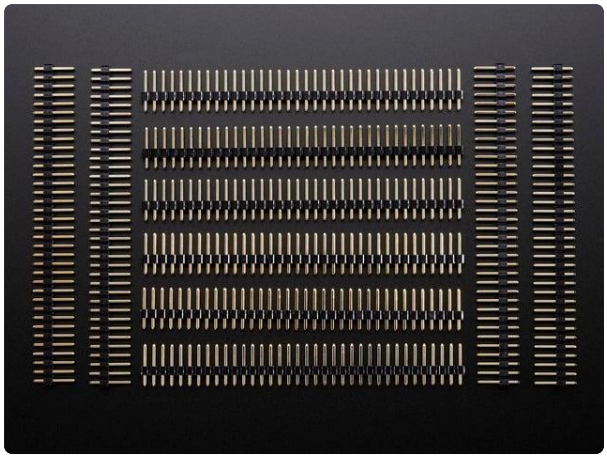


Adafruit CH334F Mini 2-Port USB Hub Breakout
Sometimes, you've got something with a USB host, like an embedded Linux board, and you want to connect more than one thing. Or maybe you want to turn something like a keyboard into...
https://www.adafruit.com/product/5999

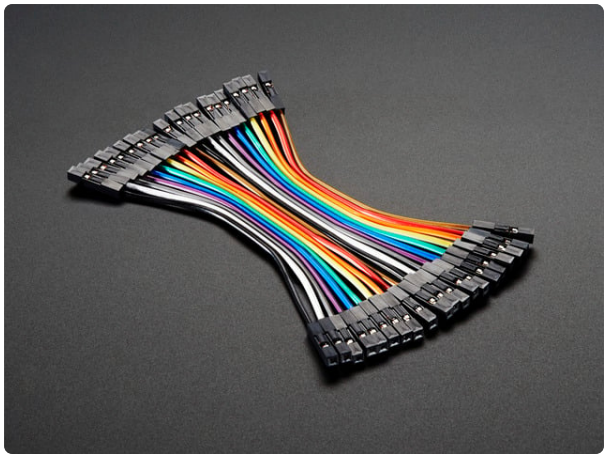
2x USB Type A Jack Breakout:



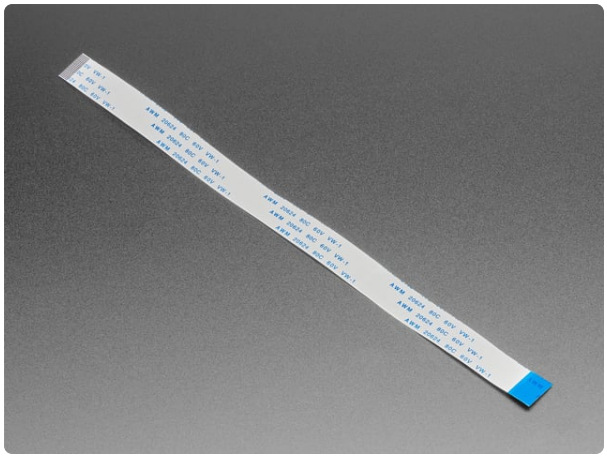
[USB Type A Jack Breakout Cable with Premium Female Jumpers]
If you'd like to connect a USB-host-capable chip to your USB peripheral, this cable will make the task very simple. There is no converter chip in this...
https://www.adafruit.com/product/4449



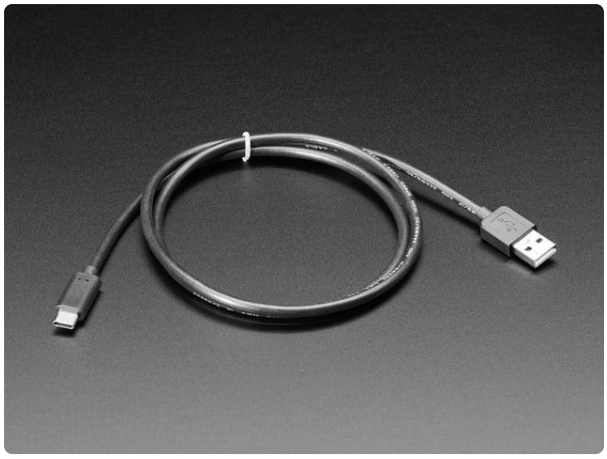
[Break-away 0.1" 36-pin strip male header - Black - 10 pack]
Breakaway header is like the duct tape of electronics. It's great for connecting things together, soldering to perf-boards, fits into any breakout or breadboard, etc. We go through...
https://www.adafruit.com/product/392



[Premium Female/Female Jumper Wires - 20 x 3" (75mm)]
These female-female premium jumper wires are handy for making wire harnesses or jumpering between headers on PCB's. They're 3" (75mm) long and come in a...
https://www.adafruit.com/product/1951



[22-pin 0.5mm pitch FPC Flex Cable for DSI CSI or HSTX - 20cm]
Connect this to that when a 22-pin FPC connector is needed. This 20 cm long cable is made of a flexible PCB. It's A-B style, meaning that pin one on one side will match with pin...
https://www.adafruit.com/product/6036



USB Type A to Type C Cable - approx 1 meter / 3 ft long

As technology changes and adapts, so does Adafruit. This USB Type A to Type C cable will help you with the transition to USB C, even if you're still...

https://www.adafruit.com/product/4474

HDMI Cable - 1 meter

Connect two HDMI devices together with this basic HDMI cable. It has nice molded grips for easy installation, and is 1 meter long (about 3 feet). This is a HDMI 1.3...

https://www.adafruit.com/product/608

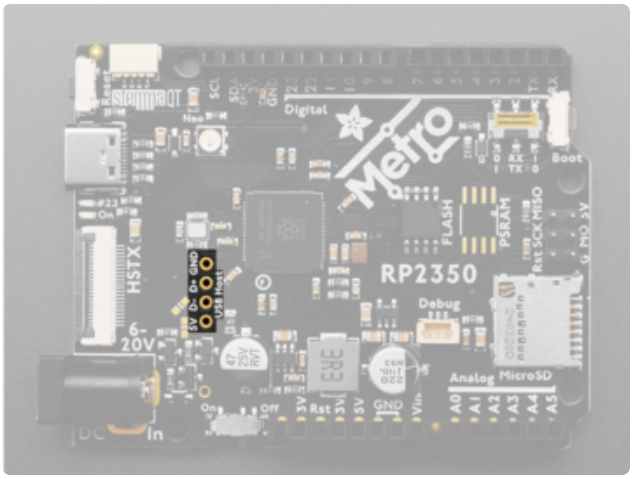
USB Wired Mouse - Two Buttons plus Wheel

This is a mouse. A nice, simple mouse. No bells or whistles. Just a mouse.But that doesn't mean it's not the best simple mouse! We compared...

https://www.adafruit.com/product/2025

Preparing the Metro RP2350

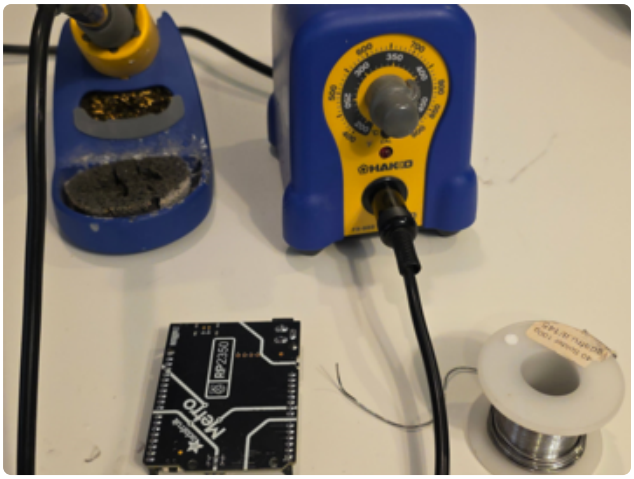
The USB Host port on the Metro RP2350, and the pins on the CH334F are the only parts of this project that require soldering.

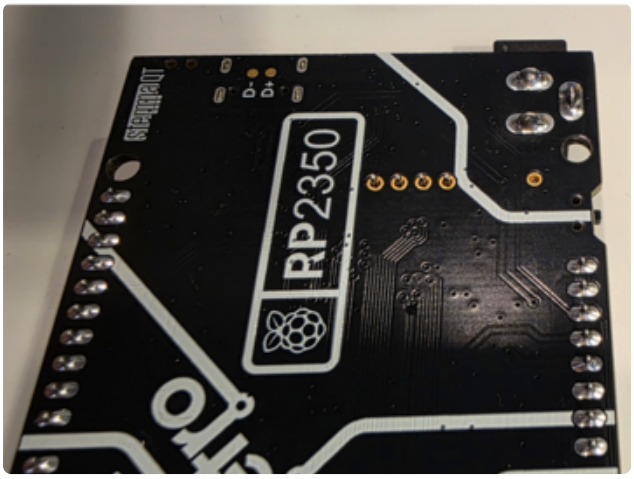


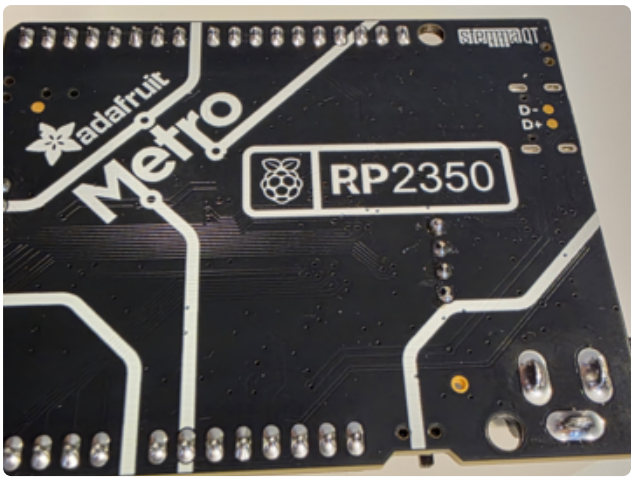
The USB Host pin connections are highlighted on the Metro image to the left. You will need a small piece of standard 0.1 inch male header, with 4 pins, to fit the holes.

You can cut header with diagonal cutters or break them with pliers or even your fingers.

Be sure to wear eye protection as header pieces can fly when cut.

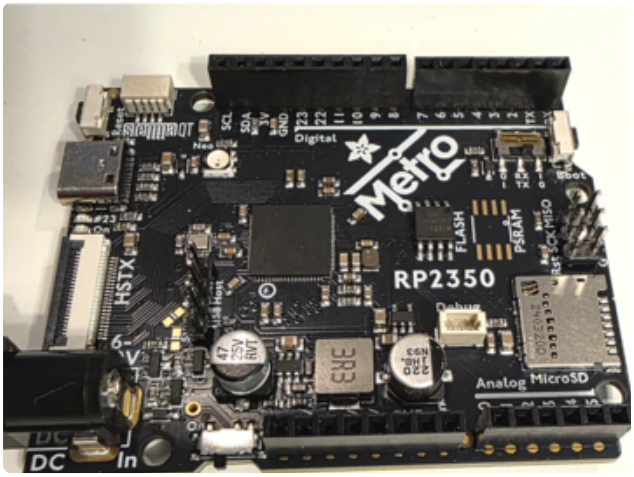






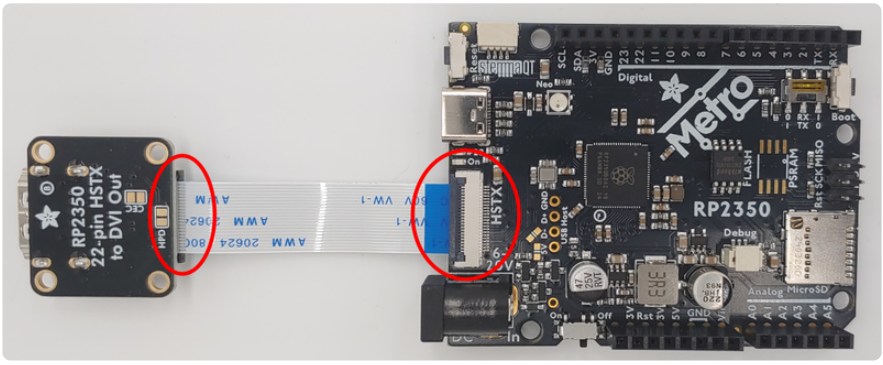
Put the short end of the header into the holes in the Metro marked USB Host and secure them with putty, blutack, tape, etc. Turn the Metro over and you should see the header barely poking out of the bottom of the board. If the pins stick through a great deal you may have the header pins upside down, double check the short end is sticking into the board.

Solder the 4 pin "nubbins" to the board.



Turn the board over and remove the material securing the pins. Now there is a new 4-pin header, which will get connected to the USB Hub on the next page.

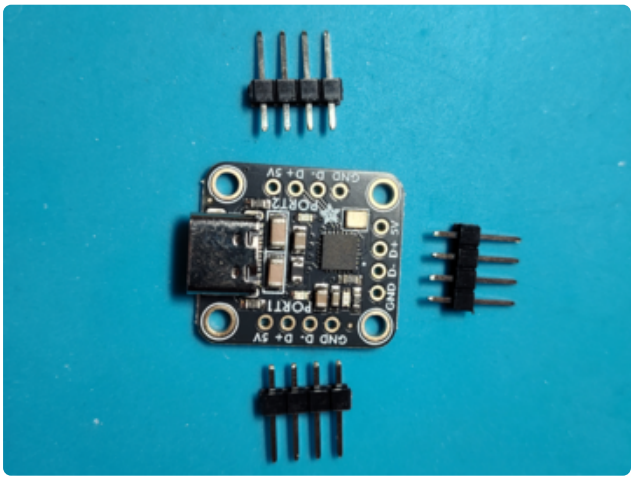
HSTX Connection to DVI

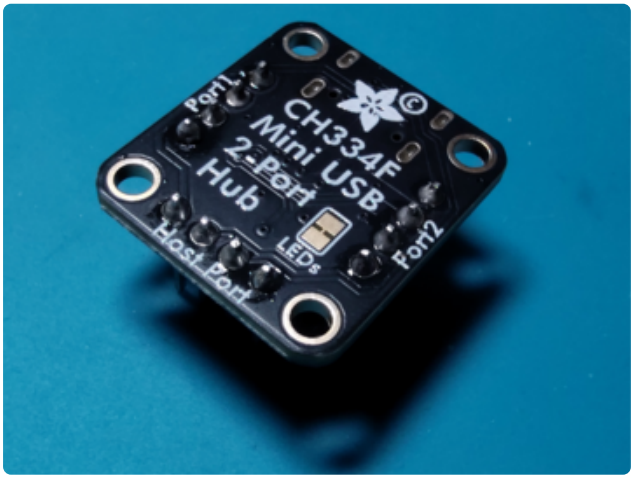


Get the HSTX cable. Any length Adafruit sells is fine. CAREFULLY lift the dark grey bar up on the Metro, insert the cable silver side down, blue side up, then put the bar CAREFULLY down, ensuring it locks. If it feels like it doesn't want to go, do not force it.

Do the same with the other end and the DVI breakout. Note that the DVI breakout will be inverted/upside down when compared to the Metro - this is normal for these boards and the Adafruit cables.

USB Hub

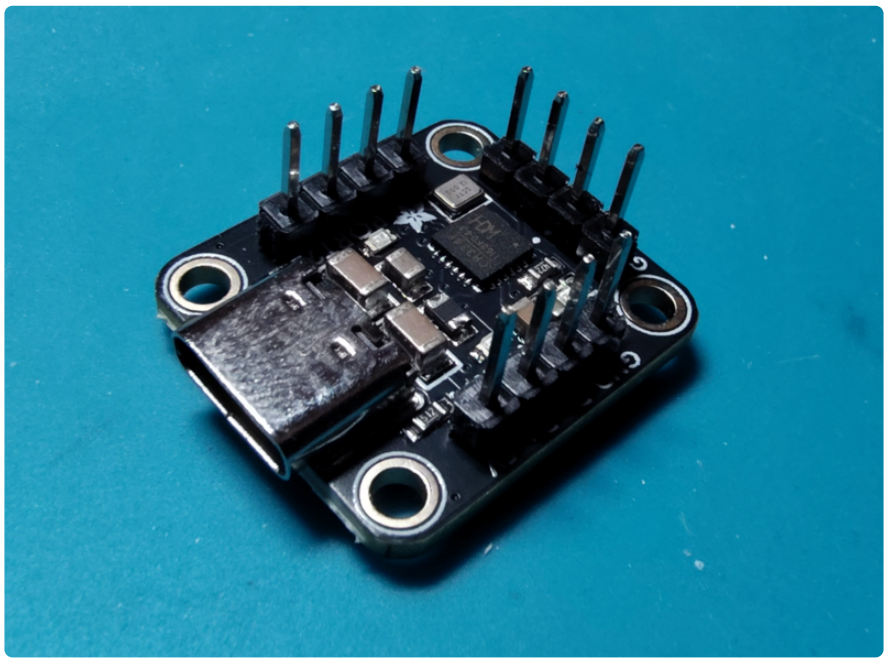


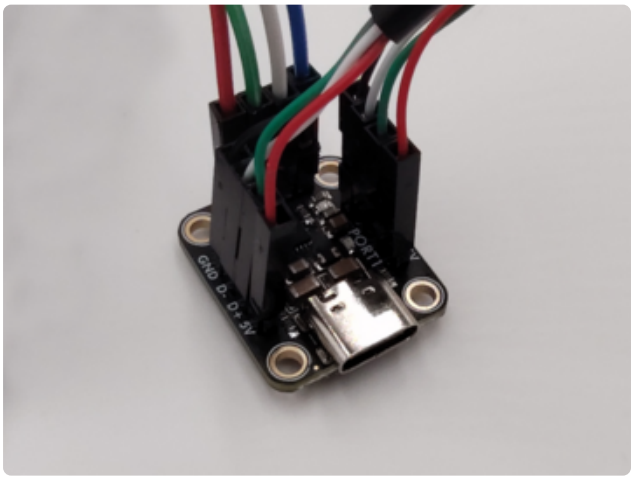


This project uses the CH334F USB Hub breakout to connect two mice to the Metro RP2350. In order to prepare the hub breakout for this project there are 3 rows of 4 header pins that need to be soldered to the breakout.

Each row of header pins will get soldered to the 4 pin holes along the sides of the breakout. The two sides next to the USB C plug are the two ports on the hub. The side opposite the USB C plug is the connection back to the Metro acting as USB Host.

Soldering the pins with the long sides going upwards makes connecting to them easier because you can see the labels on the silkscreen, though having them go downwards instead could work as well.

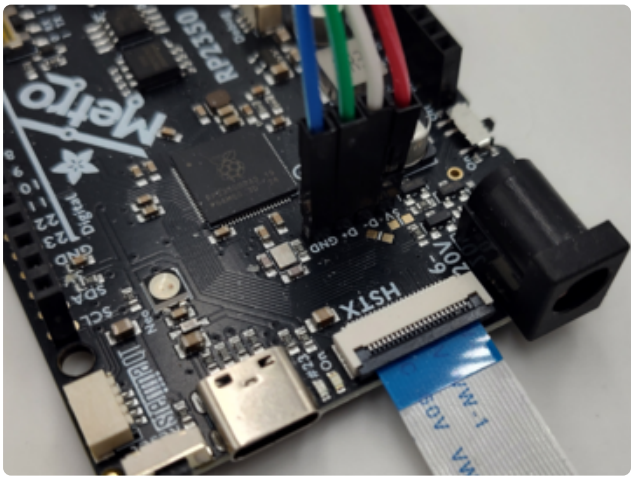




Next connect the two USB Port Breakout cables and 4 female header wires to each of the 3 rows of pins. Be sure to match the following wiring for the USB Host connectors:

GND to **Black** or **Blue**
D+ to **Green**
D- to **White**
5V to **Red**

I used a **Blue** header wire for **GND** instead of **Black**.



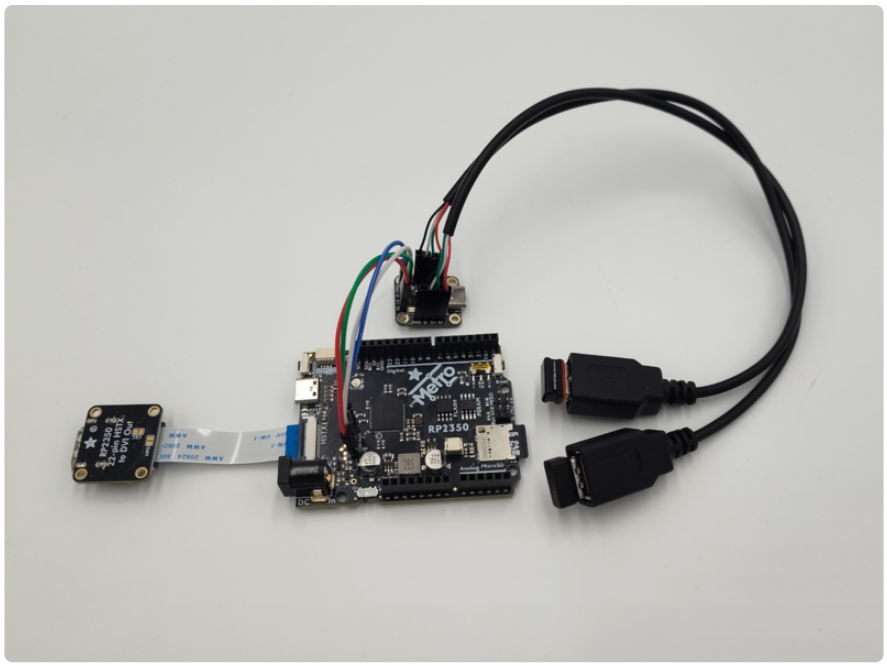
Connect the other end of the header wires to the USB Host pins soldered to the Metro RP2350 on the previous page.

GND to **Black** or **Blue**
D+ to **Green**
D- to **White**
5V to **Red**

Note that the data pins are swapped on the Metro compared to the CH334F breakout. On the Metro **D-** is next to **5V**, whereas on the CH334F **D-** is next to **GND**.

Be sure to connect **D-** on the breakout to **D-** on the Metro, and **D+** on the breakout to **D+** on the Metro.

With everything connected it will look like this:



Install CircuitPython

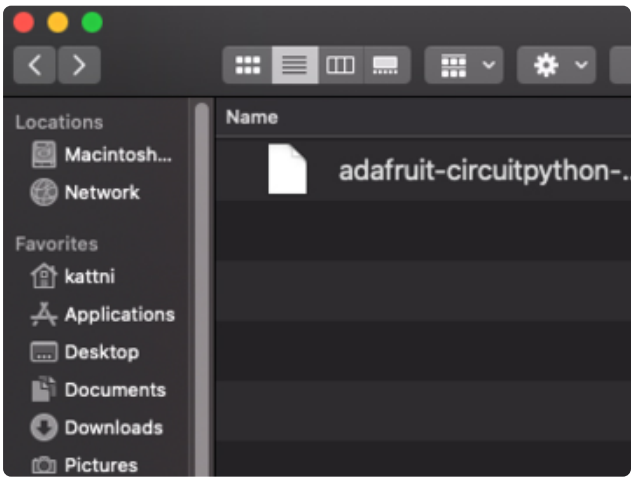
CircuitPython (https://adafru.it/tB7) is a derivative of MicroPython (https://adafru.it/BeZ) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files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

CircuitPython Quickstart

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

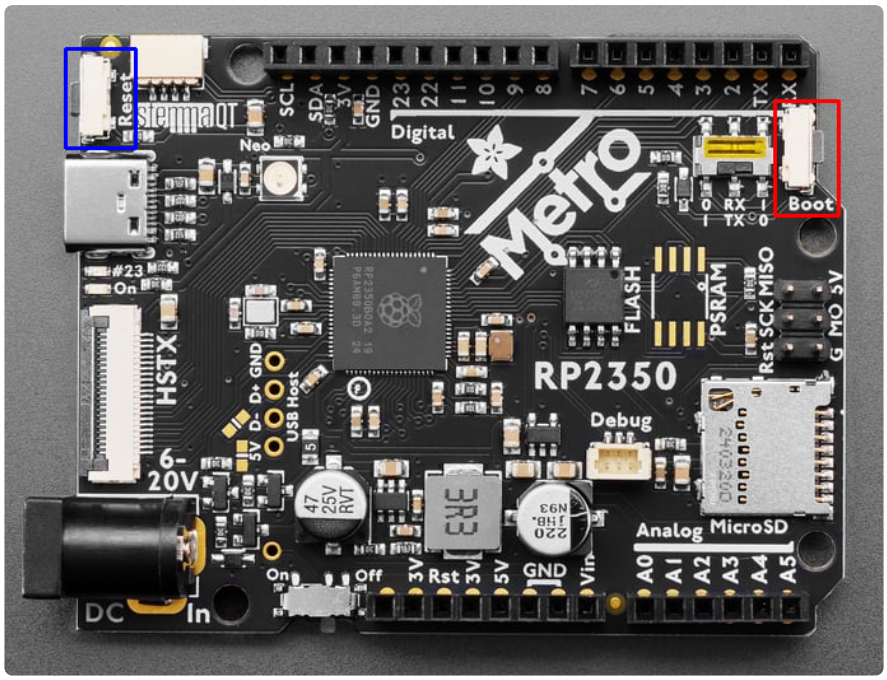
Download the latest version of CircuitPython for this board via circuitpython.org

https://adafru.it/1aeL



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

Save it wherever is convenient for you.

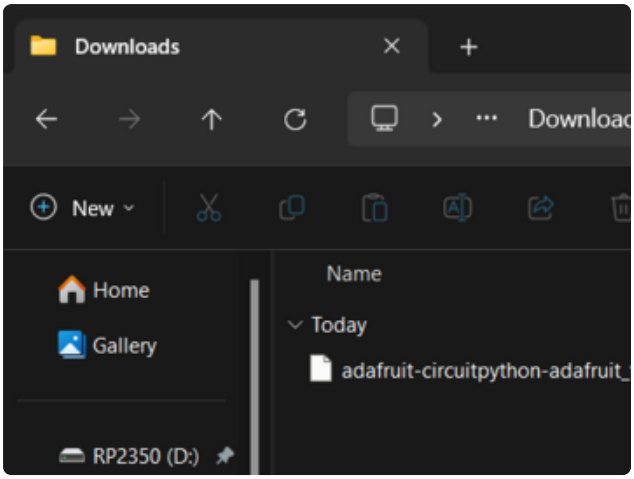


To enter the bootloader, hold down the **BOOT/BOOTSEL button** (highlighted in red above), and while continuing to hold it (don't let go!), press and release the **reset button** (highlighted in red or blue above). **Continue to hold the BOOT/BOOTSEL button until the RP2350 drive appears!**

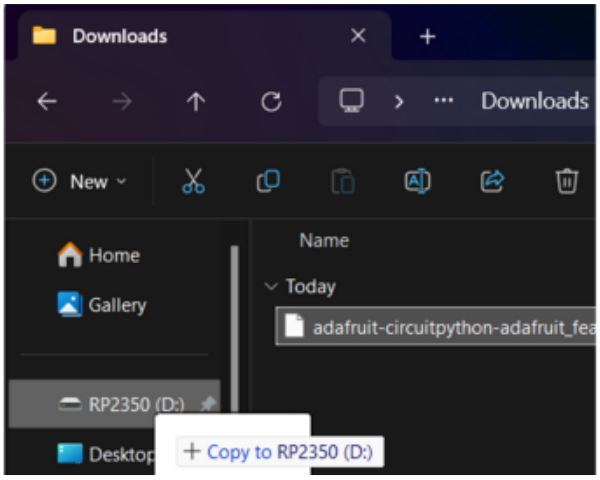
If the drive does not appear, release all the buttons, and then repeat the process above.

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.

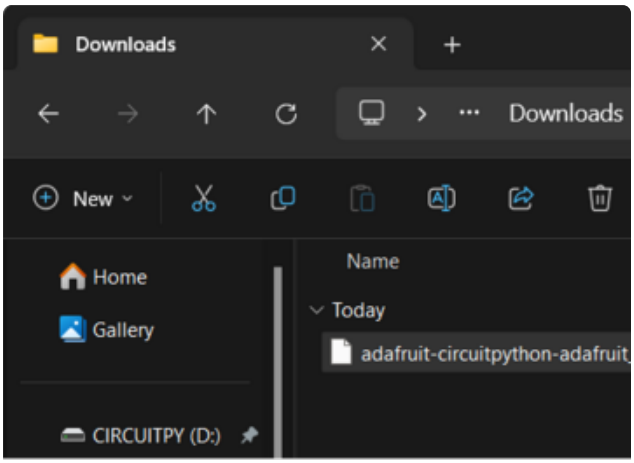
A lot of people end up using charge-only USB cables and it is very frustrating! **Make sure you have a USB cable you know is good for data sync.**



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/1afi

Wrangling Two Mice

The Match3 game uses two different USB mice for input from the players, each player gets their own mouse. This builds on top of the basic code for handling a single mouse. If you are unfamiliar with the concepts involved, first read the basic single Mouse Input guide page (https://adafru.it/1ahy) then return here.

To use two mice, start with with the example below. It illustrates finding the connected mice, initializing them, reading data, and moving two cursors around the display.

The basic mouse example is embedded below. Click the 'Download Project Bundle' button to download a zip containing this example and the associated image file and required libraries. Unzip and copy them to the **CIRCUITPY** drive on the Metro which shows up in your computer file application (Explorer or Finder) to run the example.

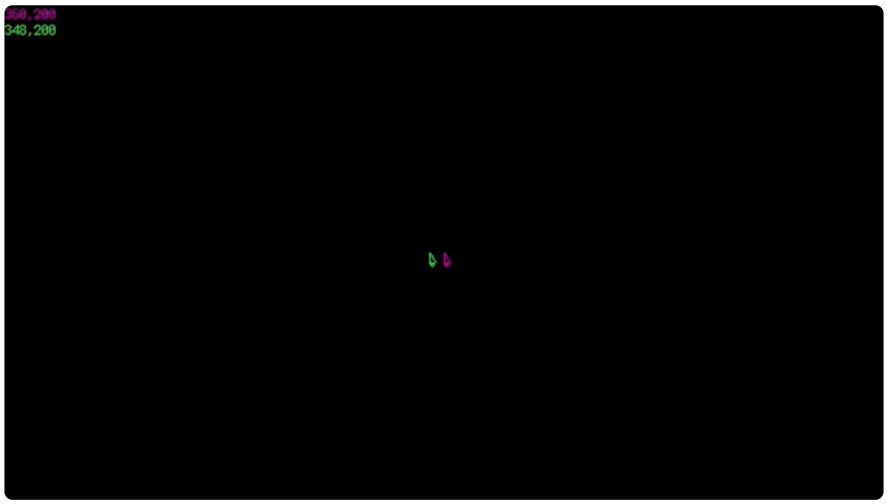
> There is active development work underway for USB Host support. If you are having trouble with your mice, try upgrading your device to CircuitPython 10.0.0-alpha.6 or newer.

```
# SPDX-FileCopyrightText: Copyright (c) 2025 Tim Cocks for Adafruit Industries
#
# SPDX-License-Identifier: MIT
import array
import supervisor
import terminalio
import usb.core
from adafruit_display_text.bitmap_label import Label
from displayio import Group, OnDiskBitmap, TileGrid, Palette

import adafruit_usb_host_descriptors

# use the default built-in display,
# the HSTX / PicoDVI display for the Metro RP2350
display = supervisor.runtime.display

# a group to hold all other visual elements
main_group = Group()

# set the main group to show on the display
display.root_group = main_group

# load the cursor bitmap file
mouse_bmp = OnDiskBitmap("mouse_cursor.bmp")

# lists for labels, mouse tilegrids, and palettes.
# each mouse will get 1 of each item. All lists
# will end up with length 2.
output_lbls = []
mouse_tgs = []
palettes = []

# the different colors to use for each mouse cursor
# and labels
```

```
colors = [0xFF00FF, 0x00FF00]

for i in range(2):
    # create a palette for this mouse
    mouse_palette = Palette(3)
    # index zero is used for transparency
    mouse_palette.make_transparent(0)
    # add the palette to the list of palettes
    palettes.append(mouse_palette)

    # copy the first two colors from mouse palette
    for palette_color_index in range(2):
        mouse_palette[palette_color_index] =
mouse_bmp.pixel_shader[palette_color_index]

    # replace the last color with different color for each mouse
    mouse_palette[2] = colors[i]

    # create a TileGrid for this mouse cursor.
    # use the palette created above
    mouse_tg = TileGrid(mouse_bmp, pixel_shader=mouse_palette)

    # move the mouse tilegrid to near the center of the display
    mouse_tg.x = display.width // 2 - (i * 12)
    mouse_tg.y = display.height // 2

    # add this mouse tilegrid to the list of mouse tilegrids
    mouse_tgs.append(mouse_tg)

    # add this mouse tilegrid to the main group so it will show
    # on the display
    main_group.append(mouse_tg)

    # create a label for this mouse
    output_lbl = Label(terminalio.FONT, text=f"{mouse_tg.x},{mouse_tg.y}",
color=colors[i], scale=1)
    # anchored to the top left corner of the label
    output_lbl.anchor_point = (0, 0)

    # move to op left corner of the display, moving
    # down by a static amount to static the two labels
    # one below the other
    output_lbl.anchored_position = (1, 1 + i * 13)

    # add the label to the list of labels
    output_lbls.append(output_lbl)

    # add the label to the main group so it will show
    # on the display
    main_group.append(output_lbl)

# lists for mouse interface indexes, endpoint addresses, and USB Device instances
# each of these will end up with length 2 once we find both mice
mouse_interface_indexes = []
mouse_endpoint_addresses = []
mice = []

# scan for connected USB devices
for device in usb.core.find(find_all=True):
    # check for boot mouse endpoints on this device
    mouse_interface_index, mouse_endpoint_address = (
        adafruit_usb_host_descriptors.find_boot_mouse_endpoint(device)
    )
    # if a boot mouse interface index and endpoint address were found
    if mouse_interface_index is not None and mouse_endpoint_address is not None:
        # add the interface index to the list of indexes
        mouse_interface_indexes.append(mouse_interface_index)
        # add the endpoint address to the list of addresses
        mouse_endpoint_addresses.append(mouse_endpoint_address)
```

```
        # add the device instance to the list of mice
        mice.append(device)

        # print details to the console
        print(f"mouse interface: {mouse_interface_index} ", end="")
        print(f"endpoint_address: {hex(mouse_endpoint_address)}")

        # detach device from kernel if needed
        if device.is_kernel_driver_active(0):
            device.detach_kernel_driver(0)

        # set the mouse configuration so it can be used
        device.set_configuration()

# This is ordered by bit position.
BUTTONS = ["left", "right", "middle"]

# list of buffers, will hold one buffer for each mouse
mouse_bufs = []
for i in range(2):
    # Buffer to hold data read from the mouse
    mouse_bufs.append(array.array("b", [0] * 8))


def get_mouse_deltas(buffer, read_count):
    """
    Given a buffer and read_count return the x and y delta values
    :param buffer: A buffer containing data read from the mouse
    :param read_count: How many bytes of data were read from the mouse
    :return: tuple x,y delta values
    """
    if read_count == 4:
        delta_x = buffer[1]
        delta_y = buffer[2]
    elif read_count == 8:
        delta_x = buffer[2]
        delta_y = buffer[4]
    else:
        raise ValueError(f"Unsupported mouse packet size:
{read_count}, must be 4 or 8")
    return delta_x, delta_y

# main loop
while True:
    # for each mouse instance
    for mouse_index, mouse in enumerate(mice):
        # try to read data from the mouse
        try:
            count = mouse.read(
                mouse_endpoint_addresses[mouse_index], mouse_bufs[mouse_index],
timeout=10
            )

        # if there is no data it will raise USBTimeoutError
        except usb.core.USBTimeoutError:
            # Nothing to do if there is no data for this mouse
            continue

        # there was mouse data, so get the delta x and y values from it
        mouse_deltas = get_mouse_deltas(mouse_bufs[mouse_index], count)

        # update the x position of this mouse cursor using the delta value
        # clamped to the display size
        mouse_tgs[mouse_index].x = max(
            0, min(display.width - 1, mouse_tgs[mouse_index].x + mouse_deltas[0])
        )
        # update the y position of this mouse cursor using the delta value
        # clamped to the display size
        mouse_tgs[mouse_index].y = max(
```

```
            0, min(display.height - 1, mouse_tgs[mouse_index].y + mouse_deltas[1])
        )

        # output string with the new cursor position
        out_str = f"{mouse_tgs[mouse_index].x},{mouse_tgs[mouse_index].y}"

        # loop over possible button bit indexes
        for i, button in enumerate(BUTTONS):
            # check each bit index to determin if the button was pressed
            if mouse_bufs[mouse_index][0] & (1 << i) != 0:
                # if it was pressed, add the button to the output string
                out_str += f" {button}"

        # set the output string into text of the label
        # to show it on the display
        output_lbls[mouse_index].text = out_str
```



The code contains comments describing the purpose of each section. An overview of the code functionality can be found below. Reading both will give you a good understanding of one of the core principals that will get used by the Match3 game.

Default Display

This demo code uses the built-in display which is the HSTX ribbon connector PicoDVI display for the Metro RP2350. In older versions of CircuitPython, the display was not automatically initialized. If you get errors dealing with `None` trying to access the display size or setting its `root_group`, then update your Metro RP2350 to the latest stable release of CircuitPython listed on [the downloads page](https://adafru.it/1aeL).

Visual Elements Setup

`main_group` is a `Group` created to hold all of the other visual elements, it gets set as `display.root_display`. `mouse_bmp` holds the loaded mouse cursor bitmap image file. For the rest of the display related objects there will be two of each type, one for each of the mice. The objects are held in lists that will contain 2 elements each after the setup is complete. `output_lbls` list holds the `Label` objects that are used to display the mouse positions and buttons pressed. `mouse_tgs` list holds the `TileGrid` objects that will draw the visual mouse cursors. `palettes` list holds a

`Palette` object for each cursor. The palettes both contain the same colors as the mouse cursor bitmap for indexes `0` and `1`, and each has their own color at index `2` which is the color that the outline of the cursor will be drawn with. In the demo code the colors are pink and green.

Finding Connected Mice

There are three additional lists declared that will hold two elements, one for each of the USB mice found connected via the USB Host API (https://adafru.it/1ahz). `mouse_interface_indexes` list will hold the interface index within connected USB device. `mouse_endpoint_addresses` will hold the address of the mouse endpoint within the USB device. Finally, the `mice` list will hold a USB `Device` object for each mouse which will be used to read data from that device during the main loop.

The code scans for connected USB devices with `usb.core.find(find_all=True)` and iterates over the returned device objects. Each device object is passed to `adafruit_usb_host_descriptors.find_boot_mouse_endpoint()` to check if it represents a boot mouse and find its interface index and endpoint address if so. If the device is found to be a mouse the appropriate values and object will get added to the respective lists.

For each mouse device object it is also detached from the kernel if needed, and its configuration is set so that data can be read from it.

Mouse Deltas Helper Function

The code defines a helper function `get_mouse_deltas()` which accepts a `buffer` and a `read_count` as arguments. This function will check the appropriate indexes within the buffer based on the `read_count` and return the delta x and y values contained within the buffer. These values represent how far the mouse has moved, and in which directions.

Main Loop

Inside the main loop is a `for` loop that iterates over each mouse device object. Inside of that, the code attempts to read data from the mouse device object. If there is no data to read, the `read()` call will raise the `USBTimeoutError` which is caught by the code and causes it to skip the rest of the loop and go on to check the next mouse object.

If there was data read, the x and y delta values will be found using the helper function and then the mouse `TileGrid` has its `x` and `y` properties updated based on the delta values, clamped to the display size.

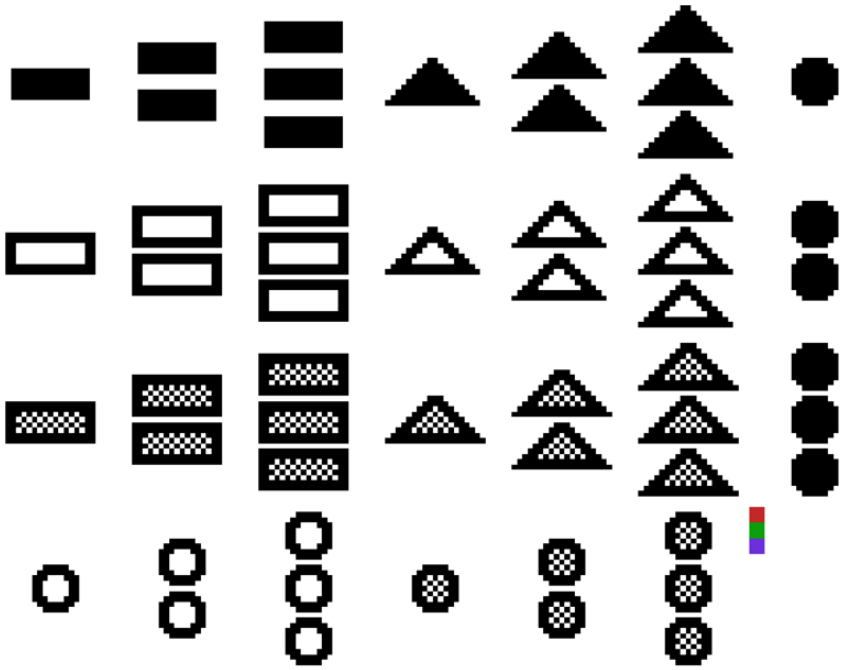
Sprites & Colors

The Match3 game makes use of the new `tilepalettemapper (https://adafru.it/1agm)` core module (https://adafru.it/1agm) which was added in CircuitPython 9.2.5. You must update your device if its version of CircuitPython is 9.2.4 or older. It is generally recommended to run the latest stable version of CircuitPython.

This demo program will illustrate how `tilepalettemapper` works and how the Match3 game code uses it to have one set of card sprites and re-map them to the 3 different possible colors, instead of needing one sprite for each of the different colors which would mean needing 3x as many sprites.

Spritesheet

This is a zoomed in look at the spritesheet used by the Match3 game.



There are a few important things to note about the spritesheet:

- The sprites are arranged into a 7x4 grid of tiles. There are 28 total sprites in the full sheet.
- 27 of the sprites are Match3 game cards. Do not worry about the different shapes and types just yet, they'll be used by the game but aren't important to this demo.
- All of the game card sprites have a white background with black foreground pixels, drawing its shape and fill type.

- The last sprite in the bottom right corner isn't a game card. It has 3 small colored blocks within it. Those 3 colors are the possible colors that the cards will get mapped to when drawn on the display: red, green, and purple. Any pixel in a card that is black in the spritesheet will be mapped one of those other 3 colors when shown with a `TileGrid`.

This is the palette for the spritesheet image. It contains the following colors at their respective indexes.

0: white
1: black
2: purple
3: red
4: green

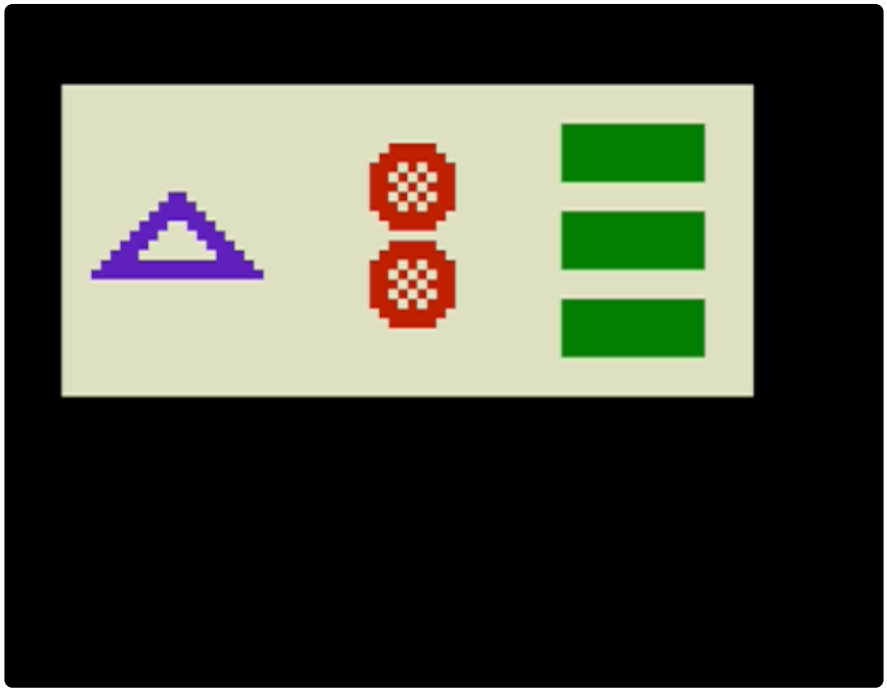
The `TilePaletteMapper` object maps the indexes within the palette for each tile within a `TileGrid` to different chosen indexes. For the Match3 game, and in this demo script the code maps index `1` to use one of indexes `2`, `3`, or `4`. Since black is at index `1` and the card designs are drawn in black in the spritesheet, this will have the effect of changing the card to either purple, red, or green depending on which index is chosen for the mapping.

Demo Code

The demo code is shown below followed by a screenshot of the cards that it displays. If you want to run it on your own device click the "Download Project Bundle" button, unzip the project files and copy them to the CIRCUITPY drive.

```
# SPDX-FileCopyrightText: Copyright (c) 2025 Tim Cocks for Adafruit Industries
#
# SPDX-License-Identifier: MIT
import sys

import supervisor
from displayio import Group, OnDiskBitmap, TileGrid
from tilepalettemapper import TilePaletteMapper

# use the default built-in display,
# the HSTX / PicoDVI display for the Metro RP2350
display = supervisor.runtime.display

# a group to hold all other visual elements
main_group = Group(scale=4, x=30, y=30)

# set the main group to show on the display
display.root_group = main_group

# load the sprite sheet bitmap
spritesheet_bmp = OnDiskBitmap("match3_cards_spritesheet.bmp")
```

```
# create a TilePaletteMapper
if sys.implementation.version[0] == 9:
    tile_palette_mapper = TilePaletteMapper(
        spritesheet_bmp.pixel_shader,  # input pixel_shader
        5,  # input color count
        3,  # grid width
        1,  # grid height
    )
elif sys.implementation.version[0] >= 10:
    tile_palette_mapper = TilePaletteMapper(
        spritesheet_bmp.pixel_shader,  # input pixel_shader
        5,  # input color count
    )

# create a TileGrid to show some cards
cards_tilegrid = TileGrid(
    spritesheet_bmp,
    pixel_shader=tile_palette_mapper,
    width=3,
    height=1,
    tile_width=24,
    tile_height=32,
)

# set each tile in the grid to a different sprite index
cards_tilegrid[0, 0] = 10
cards_tilegrid[1, 0] = 25
cards_tilegrid[2, 0] = 2

# re-map each tile in the grid to use a different color for index 1
# all other indexes remain their default values
tile_palette_mapper[0, 0] = [0, 2, 2, 3, 4]
tile_palette_mapper[1, 0] = [0, 3, 2, 3, 4]
tile_palette_mapper[2, 0] = [0, 4, 2, 3, 4]

# add the tilegrid to the main group
main_group.append(cards_tilegrid)

# wait forever so it remains visible on the display
while True:
    pass
```



TilePaletteMapper Usage

The code creates a `TilePalleteMapper` instance, supplying an input color count, grid width, and grid height as arguments.

```
tile_palette_mapper = TilePaletteMapper(
    spritesheet_bmp.pixel_shader,  # input pixel_shader
    5,  # input color count
    3,  # grid width
    1   # grid height
)
```

`tile_palette_mapper` is then used as the `pixel_shader` for a `TileGrid`

```
cards_tilegrid = TileGrid(spritesheet_bmp, pixel_shader=tile_palette_mapper,
                          width=3, height=1, tile_width=24, tile_height=32)
```

To re-map the colors the code sets new color index lists at the desired tile locations.

```
# re-map each tile in the grid to use a different color for index 1
# all other indexes remain their default values
tile_palette_mapper[0, 0] = [0, 2, 2, 3, 4]
tile_palette_mapper[1, 0] = [0, 3, 2, 3, 4]
tile_palette_mapper[2, 0] = [0, 4, 2, 3, 4]
```

To change the foreground color of each card a new color index is supplied for the second value in the list. By default it would be index `1` to match its own index within the list. The code sets the 3 different tile positions to `2`, `3`, and `4` which are the indexes for purple, red, and green respectively.

Game Mechanics: Autosave & Resume

The Match3 game supports auto-save and resume if an SD Card is inserted into the Metro. Without an SD Card, the game will work as normal, but progress will be lost if the device loses power or resets.

In order to understand how the auto-save and resume mechanics work, first consider this simplified demo script.

Demo Code

The demo code is shown below followed by a screenshot and explanation. If you want to run it on your own device click the "Download Project Bundle" button, unzip the project files and copy them to the **CIRCUITPY** drive.

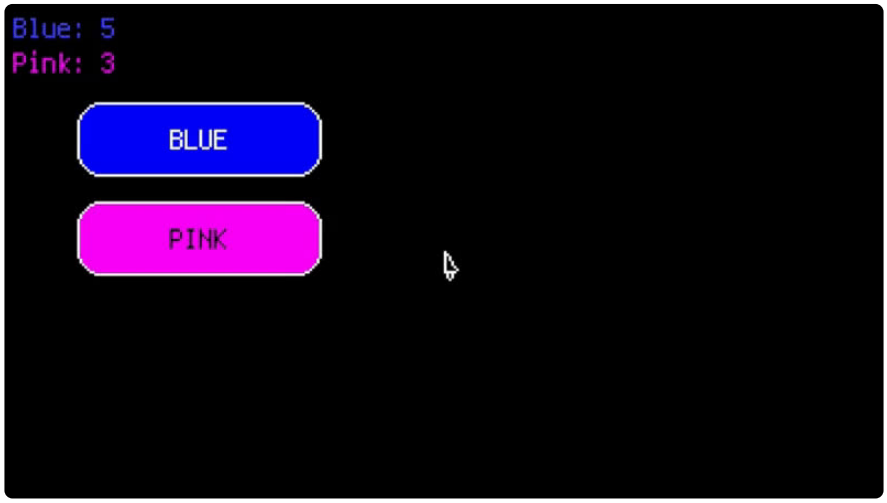
```
# SPDX-FileCopyrightText: 2024 Tim Cocks for Adafruit Industries
# SPDX-License-Identifier: MIT
"""
This example demonstrates basic autosave and resume functionality. There are two
```

```
buttons
that can be clicked to increment respective counters. The number of clicks is stored
in a game_state dictionary and saved to a data file on the SDCard. When the code
first
launches it will read the data file and load the game_state from it.
"""
import array
from io import BytesIO
import os

import board
import busio
import digitalio
import displayio
import msgpack
import storage
import supervisor
import terminalio
import usb

import adafruit_sdcard
from adafruit_display_text.bitmap_label import Label
from adafruit_button import Button

# use the default built-in display
display = supervisor.runtime.display

# button configuration
BUTTON_WIDTH = 100
BUTTON_HEIGHT = 30
BUTTON_STYLE = Button.ROUNDRECT

# game state object will get loaded from SDCard
# or a new one initialized as a dictionary
game_state = None

save_to = None

# boolean variables for possible SDCard states
sd_pins_in_use = False

# The SD_CS pin is the chip select line.
SD_CS = board.SD_CS

# try to Connect to the sdcard card and mount the filesystem.
try:
    # initialze CS pin
    cs = digitalio.DigitalInOut(SD_CS)
except ValueError:
    # likely the SDCard was auto-initialized by the core
    sd_pins_in_use = True

try:
    # if sd CS pin was not in use
    if not sd_pins_in_use:
        # try to initialize and mount the SDCard
        sdcard = adafruit_sdcard.SDCard(
            busio.SPI(board.SD_SCK, board.SD_MOSI, board.SD_MISO), cs
        )
        vfs = storage.VfsFat(sdcard)
        storage.mount(vfs, "/sd")

    # check for the autosave data file
    if "autosave_demo.dat" in os.listdir("/sd/"):
        # if the file is found read data from it into a BytesIO buffer
        buffer = BytesIO()
        with open("/sd/autosave_demo.dat", "rb") as f:
            buffer.write(f.read())
        buffer.seek(0)
```

```
        # unpack the game_state object from the read data in the buffer
        game_state = msgpack.unpack(buffer)
        print(game_state)

    # if placeholder.txt file does not exist
    if "placeholder.txt" not in os.listdir("/sd/"):
        # if we made it to here then /sd/ exists and has a card
        # so use it for save data
        save_to = "/sd/autosave_demo.dat"
except OSError as e:
    # sdcard init or mounting failed
    raise OSError(
        "This demo requires an SDCard. Please power off the device " +
        "insert an SDCard and then plug it back in."
    ) from e

# if no saved game_state was loaded
if game_state is None:
    # create a new game state dictionary
    game_state = {"pink_count": 0, "blue_count": 0}

# Make the display context
main_group = displayio.Group()
display.root_group = main_group

# make buttons
blue_button = Button(
    x=30,
    y=40,
    width=BUTTON_WIDTH,
    height=BUTTON_HEIGHT,
    style=BUTTON_STYLE,
    fill_color=0x0000FF,
    outline_color=0xFFFFFF,
    label="BLUE",
    label_font=terminalio.FONT,
    label_color=0xFFFFFF,
)

pink_button = Button(
    x=30,
    y=80,
    width=BUTTON_WIDTH,
    height=BUTTON_HEIGHT,
    style=BUTTON_STYLE,
    fill_color=0xFF00FF,
    outline_color=0xFFFFFF,
    label="PINK",
    label_font=terminalio.FONT,
    label_color=0x000000,
)

# add them to a list for easy iteration
all_buttons = [blue_button, pink_button]

# Add buttons to the display context
main_group.append(blue_button)
main_group.append(pink_button)

# make labels for each button
blue_lbl = Label(
    terminalio.FONT, text=f"Blue: {game_state['blue_count']}", color=0x3F3FFF
)
blue_lbl.anchor_point = (0, 0)
blue_lbl.anchored_position = (4, 4)
pink_lbl = Label(
    terminalio.FONT, text=f"Pink: {game_state['pink_count']}", color=0xFF00FF
)
```

```
pink_lbl.anchor_point = (0, 0)
pink_lbl.anchored_position = (4, 4 + 14)
main_group.append(blue_lbl)
main_group.append(pink_lbl)

# load the mouse cursor bitmap
mouse_bmp = displayio.OnDiskBitmap("mouse_cursor.bmp")

# make the background pink pixels transparent
mouse_bmp.pixel_shader.make_transparent(0)

# create a TileGrid for the mouse, using its bitmap and pixel_shader
mouse_tg = displayio.TileGrid(mouse_bmp, pixel_shader=mouse_bmp.pixel_shader)

# move it to the center of the display
mouse_tg.x = display.width // 2
mouse_tg.y = display.height // 2

# add the mouse tilegrid to main_group
main_group.append(mouse_tg)

# scan for connected USB device and loop over any found
for device in usb.core.find(find_all=True):
    # print device info
    print(f"{device.idVendor:04x}:{device.idProduct:04x}")
    print(device.manufacturer, device.product)
    print(device.serial_number)
    # assume the device is the mouse
    mouse = device

# detach the kernel driver if needed
if mouse.is_kernel_driver_active(0):
    mouse.detach_kernel_driver(0)

# set configuration on the mouse so we can use it
mouse.set_configuration()

# buffer to hold mouse data
# Boot mice have 4 byte reports
buf = array.array("b", [0] * 4)


def save_game_state():
    """
    msgpack the game_state and save it to the autosave data file
    :return:
    """
    b = BytesIO()
    msgpack.pack(game_state, b)
    b.seek(0)
    with open(save_to, "wb") as savefile:
        savefile.write(b.read())


# main loop
while True:
    try:
        # attempt to read data from the mouse
        # 10ms timeout, so we don't block long if there
        # is no data
        count = mouse.read(0x81, buf, timeout=10)
    except usb.core.USBTimeoutError:
        # skip the rest of the loop if there is no data
        continue

    # update the mouse tilegrid x and y coordinates
    # based on the delta values read from the mouse
    mouse_tg.x = max(0, min(display.width - 1, mouse_tg.x + buf[1]))
    mouse_tg.y = max(0, min(display.height - 1, mouse_tg.y + buf[2]))
```

```
    # if left click is pressed
    if buf[0] & (1 << 0) != 0:
        # get the current cursor coordinates
        coords = (mouse_tg.x, mouse_tg.y, 0)

        # loop over the buttons
        for button in all_buttons:
            # if the current button contains the mouse coords
            if button.contains(coords):
                # if the button isn't already in the selected state
                if not button.selected:
                    # enter selected state
                    button.selected = True

                    # if it is the pink button
                    if button == pink_button:
                        # increment pink count
                        game_state["pink_count"] += 1
                        # update the label
                        pink_lbl.text = f"Pink: {game_state['pink_count']}"

                    # if it is the blue button
                    elif button == blue_button:
                        # increment blue count
                        game_state["blue_count"] += 1
                        # update the label
                        blue_lbl.text = f"Blue: {game_state['blue_count']}"

                    # save the new game state
                    save_game_state()

            # if the click is not on the current button
            else:
                # set this button as not selected
                button.selected = False

                # left click is not pressed
    else:
        # set all buttons as not selected
        for button in all_buttons:
            button.selected = False
```



Auto-save & Resume Functionality

Before the code can read and write files on the SD Card, it must be initialized and mounted. In version 10.0, CircuitPython will begin initializing and mounting the SD

Card automatically. For any versions prior to that, the user code must handle the initialization and mounting. This demo script supports both by first checking if the SD Card is already mounted and only trying to initialize and mount it if it wasn't.

```
# The SD_CS pin is the chip select line.
SD_CS = board.SD_CS

# try to Connect to the sdcard card and mount the filesystem.
try:
    # initialze CS pin
    cs = digitalio.DigitalInOut(SD_CS)
except ValueError:
    # likely the SDCard was auto-initialized by the core
    sd_pins_in_use = True

try:
    # if sd CS pin was not in use
    if not sd_pins_in_use:
        # try to initialize and mount the SDCard
        sdcard = adafruit_sdcard.SDCard(busio.SPI(board.SD_SCK, board.SD_MOSI,
board.SD_MISO), cs)
        vfs = storage.VfsFat(sdcard)
        storage.mount(vfs, "/sd")
```

Next the code will attempt to read the **autosave_demo.dat** file and load `game_state` from it. For storing the `game_state` more efficiently, the [msgpack module](https://adafru.it/1ahA) is used.

```
# check for the autosave data file
if "autosave_demo.dat" in os.listdir("/sd/"):
    # if the file is found read data from it into a BytesIO buffer
    buffer = BytesIO()
    with open("/sd/autosave_demo.dat", "rb") as f:
        buffer.write(f.read())
    buffer.seek(0)

    # unpack the game_state object from the read data in the buffer
    game_state = msgpack.unpack(buffer)
    print(game_state)
```

If there was no autosave file, but the SD Card was mounted successfully, then the code will set up the `save_to` variable holding a file path to the location where the autosave file will get written.

If no game state was loaded, then a new one is initialized with `0` values for the button counts.

```
# if placeholder.txt file does not exist
    if "placeholder.txt" not in os.listdir("/sd/"):
        # if we made it to here then /sd/ exists and has a card
        # so use it for save data
        save_to = "/sd/autosave_demo.dat"
except OSError:
    # sdcard init or mounting failed
    raise OSError(
        "This demo requires an SDCard. Please power off the device insert an SDCard
and then plug it back in.")
```

```
# if no saved game_state was loaded
if game_state is None:
    # create a new game state dictionary
    game_state = {'pink_count': 0, 'blue_count': 0}
```

Saving Game State

A helper function is defined that saves the game state when it is called.

```
def save_game_state():
    """
    msgpack the game_state and save it to the autosave data file
    :return:
    """
    b = BytesIO()
    msgpack.pack(game_state, b)
    b.seek(0)
    with open(save_to, "wb") as f:
        f.write(b.read())
```

Click For Points

The rest of the code handles the mouse movement and waits for the buttons to be clicked. Once a button is clicked, the `game_state` is updated by incrementing the count for the corresponding color, and then the `game_state` is saved by calling the helper function.

Code

CircuitPython Usage

To use the game, you need to update **code.py** with the game program to the **CIRCUITPY** drive.

Thankfully, this can be done in one go. In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file.

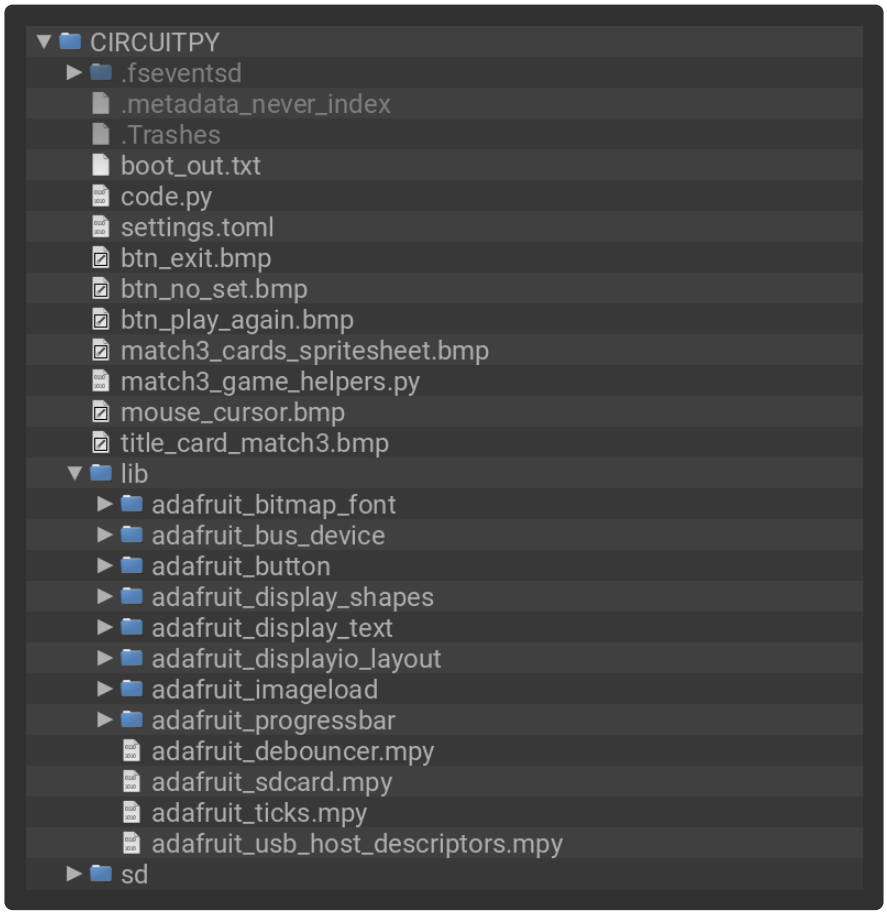
Connect your board to your computer via a known good data+power USB cable. The board should show up in your File Explorer/Finder (depending on your operating system) as a flash drive named **CIRCUITPY**.

Extract the contents of the zip file, copy the **lib** directory files to **CIRCUITPY/lib**. Copy the **code.py** file to your **CIRCUITPY** drive. The program should self start.

There is active development work underway for USB Host support. If you are having trouble with your mice, try upgrading your device to CircuitPython 10.0.0-alpha.6 or newer.

Drive Structure

After copying the files, your drive should look like the listing below. It can contain other files as well, but must contain these at a minimum.



Note that the **CIRCUITPY/sd** directory is required.

Code

The **code.py** for the Memory game is shown below.

```
# SPDX-FileCopyrightText: Copyright (c) 2025 Tim Cocks for Adafruit Industries
#
# SPDX-License-Identifier: MIT
"""
Match3 game inspired by the Set card game. Two players compete
to find sets of cards that share matching or mis-matching traits.
"""
import array
import atexit
```

```
import io
import os
import sys
import time

import board
import busio
import digitalio
import supervisor
import terminalio
import usb
from tilepalettemapper import TilePaletteMapper
from displayio import TileGrid, Group, Palette, OnDiskBitmap, Bitmap
from adafruit_display_text.text_box import TextBox
import adafruit_usb_host_descriptors
from adafruit_debouncer import Debouncer
import adafruit_sdcard
import msgpack
import storage
from match3_game_helpers import (
    Match3Game,
    STATE_GAMEOVER,
    STATE_PLAYING_SETCALLED,
    GameOverException,
)

original_autoreload_val = supervisor.runtime.autoreload
supervisor.runtime.autoreload = False

AUTOSAVE_FILENAME = "match3_game_autosave.dat"

main_group = Group()
display = supervisor.runtime.display

# set up scale factor of 2 for larger display resolution
scale_factor = 1
if display.width > 360:
    scale_factor = 2
    main_group.scale = scale_factor

save_to = None
game_state = None
try:
    # check for autosave file on CPSAVES drive
    if AUTOSAVE_FILENAME in os.listdir("/saves/"):
        savegame_buffer = io.BytesIO()
        with open(f"/saves/{AUTOSAVE_FILENAME}", "rb") as f:
            savegame_buffer.write(f.read())
        savegame_buffer.seek(0)
        game_state = msgpack.unpack(savegame_buffer)
        print(game_state)

    # if we made it to here then /saves/ exist so use it for
    # save data
    save_to = f"/saves/{AUTOSAVE_FILENAME}"
except OSError as e:
    # no /saves/ dir likely means no CPSAVES
    pass

sd_pins_in_use = False

if game_state is None:
    # try to use sdcard for saves
    # The SD_CS pin is the chip select line.
    SD_CS = board.SD_CS

    # Connect to the card and mount the filesystem.
    try:
        cs = digitalio.DigitalInOut(SD_CS)
```

```
    except ValueError:
        sd_pins_in_use = True

    print(f"sd pins in use: {sd_pins_in_use}")
    try:
        if not sd_pins_in_use:
            sdcard = adafruit_sdcard.SDCard(
                busio.SPI(board.SD_SCK, board.SD_MOSI, board.SD_MISO), cs
            )
            vfs = storage.VfsFat(sdcard)
            storage.mount(vfs, "/sd")

        if "set_game_autosave.dat" in os.listdir("/sd/"):
            savegame_buffer = io.BytesIO()
            with open("/sd/set_game_autosave.dat", "rb") as f:
                savegame_buffer.write(f.read())
            savegame_buffer.seek(0)
            game_state = msgpack.unpack(savegame_buffer)
            print(game_state)

        if "placeholder.txt" not in os.listdir("/sd/"):
            # if we made it to here then /sd/ exists and has a card
            # so use it for save data
            save_to = "/sd/set_game_autosave.dat"
    except OSError:
        # no SDcard
        pass

# background color
bg_bmp = Bitmap(
    display.width // scale_factor // 10, display.height // scale_factor // 10, 1
)
bg_palette = Palette(1)
bg_palette[0] = 0x888888
bg_tg = TileGrid(bg_bmp, pixel_shader=bg_palette)
bg_group = Group(scale=10)
bg_group.append(bg_tg)
main_group.append(bg_group)

# create Game helper object
match3_game = Match3Game(
    game_state=game_state,
    display_size=(display.width // scale_factor, display.height // scale_factor),
    save_location=save_to,
)
main_group.append(match3_game)

# create a group to hold the game over elements
game_over_group = Group()

# create a TextBox to hold the game over message
game_over_label = TextBox(
    terminalio.FONT,
    text="",
    color=0xFFFFFF,
    background_color=0x111111,
    width=display.width // scale_factor // 2,
    height=110,
    align=TextBox.ALIGN_CENTER,
)
# move it to the center top of the display
game_over_label.anchor_point = (0, 0)
game_over_label.anchored_position = (
    display.width // scale_factor // 2 - (game_over_label.width) // 2,
    40,
)

# make it hidden, we'll show it when the game is over.
game_over_group.hidden = True
```

```
# add the game over lable to the game over group
game_over_group.append(game_over_label)

# load the play again, and exit button bitmaps
play_again_btn_bmp = OnDiskBitmap("btn_play_again.bmp")
exit_btn_bmp = OnDiskBitmap("btn_exit.bmp")

# create TileGrid for the play again button
play_again_btn = TileGrid(
    bitmap=play_again_btn_bmp, pixel_shader=play_again_btn_bmp.pixel_shader
)

# transparent pixels in the corners for the rounded corner effect
play_again_btn_bmp.pixel_shader.make_transparent(0)

# centered within the display, offset to the left
play_again_btn.x = (
    display.width // scale_factor // 2 - (play_again_btn_bmp.width) // 2 - 30
)

# inside the bounds of the game over label, so it looks like a dialog visually
play_again_btn.y = 100

# create TileGrid for the exit button
exit_btn = TileGrid(bitmap=exit_btn_bmp, pixel_shader=exit_btn_bmp.pixel_shader)

# transparent pixels in the corners for the rounded corner effect
exit_btn_bmp.pixel_shader.make_transparent(0)

# centered within the display, offset to the right
exit_btn.x = display.width // scale_factor // 2 - (exit_btn_bmp.width) // 2 + 30

# inside the bounds of the game over label, so it looks like a dialog visually
exit_btn.y = 100

# add the play again and exit buttons to the game over group
game_over_group.append(play_again_btn)
game_over_group.append(exit_btn)
main_group.append(game_over_group)

# wait a second for USB devices to be ready
time.sleep(1)

# load the mouse bitmap
mouse_bmp = OnDiskBitmap("mouse_cursor.bmp")

# make the background pink pixels transparent
mouse_bmp.pixel_shader.make_transparent(0)

# list for mouse tilegrids
mouse_tgs = []
# list for palette mappers, one for each mouse
palette_mappers = []
# list for mouse colors
colors = [0x2244FF, 0xFFFF00]

# remap palette will have the 3 colors from mouse bitmap
# and the two colors from the mouse colors list
remap_palette = Palette(3 + len(colors))
# index 0 is transparent
remap_palette.make_transparent(0)

# copy the 3 colors from the mouse bitmap palette
for i in range(3):
    remap_palette[i] = mouse_bmp.pixel_shader[i]

# copy the 2 colors from the mouse colors list
for i in range(2):
```

```
    remap_palette[i + 3] = colors[i]

# create tile palette mappers
for i in range(2):
    if sys.implementation.version[0] == 9:
        palette_mapper = TilePaletteMapper(remap_palette, 3, 1, 1)
    elif sys.implementation.version[0] >= 10:
        palette_mapper = TilePaletteMapper(remap_palette, 3)

    palette_mappers.append(palette_mapper)

    # create tilegrid for each mouse
    mouse_tg = TileGrid(mouse_bmp, pixel_shader=palette_mapper)
    mouse_tg.x = display.width // scale_factor // 2 - (i * 12)
    mouse_tg.y = display.height // scale_factor // 2
    mouse_tgs.append(mouse_tg)

    # remap index 2 to each of the colors in mouse colors list
    palette_mapper[0] = [0, 1, i + 3]

# USB info lists
mouse_interface_indexes = []
mouse_endpoint_addresses = []
kernel_driver_active_flags = []
# USB device object instance list
mice = []
# buffers list for mouse packet data
mouse_bufs = []
# debouncers list for debouncing mouse left clicks
mouse_debouncers = []

# scan for connected USB devices
for device in usb.core.find(find_all=True):
    # check if current device is has a boot mouse endpoint
    mouse_interface_index, mouse_endpoint_address = (
        adafruit_usb_host_descriptors.find_boot_mouse_endpoint(device)
    )
    if mouse_interface_index is not None and mouse_endpoint_address is not None:
        # if it does have a boot mouse endpoint then add information to the
        # usb info lists
        mouse_interface_indexes.append(mouse_interface_index)
        mouse_endpoint_addresses.append(mouse_endpoint_address)

        # add the mouse device instance to list
        mice.append(device)
        print(
            f"mouse interface: {mouse_interface_index} "
            + f"endpoint_address: {hex(mouse_endpoint_address)}"
        )

        # detach kernel driver if needed
        kernel_driver_active_flags.append(device.is_kernel_driver_active(0))
        if device.is_kernel_driver_active(0):
            device.detach_kernel_driver(0)

        # set the mouse configuration so it can be used
        device.set_configuration()


def is_mouse1_left_clicked():
    """
    Check if mouse 1 left click is pressed
    :return: True if mouse 1 left click is pressed
    """
    return is_left_mouse_clicked(mouse_bufs[0])


def is_mouse2_left_clicked():
    """
```

```
    Check if mouse 2 left click is pressed
    :return: True if mouse 2 left click is pressed
    """
    return is_left_mouse_clicked(mouse_bufs[1])


def is_left_mouse_clicked(buf):
    """
    Check if a mouse is pressed given its packet buffer
    filled with read data
    :param buf: the buffer containing the packet data
    :return: True if mouse left click is  pressed
    """
    val = buf[0] & (1 << 0) != 0
    return val


def is_right_mouse_clicked(buf):
    """
    check if a mouse right click is pressed given its packet buffer
    :param buf: the buffer containing the packet data
    :return: True if mouse right click is pressed
    """
    val = buf[0] & (1 << 1) != 0
    return val


# print(f"addresses: {mouse_endpoint_addresses}")
# print(f"indexes: {mouse_interface_indexes}")

for mouse_tg in mouse_tgs:
    # add the mouse to the main group
    main_group.append(mouse_tg)

    # Buffer to hold data read from the mouse
    # Boot mice have 4 byte reports
    mouse_bufs.append(array.array("b", [0] * 8))

# create debouncer objects for left click functions
mouse_debouncers.append(Debouncer(is_mouse1_left_clicked))
mouse_debouncers.append(Debouncer(is_mouse2_left_clicked))

# set main_group as root_group, so it is visible on the display
display.root_group = main_group

# variable to hold winning player
winner = None


def get_mouse_deltas(buffer, read_count):
    """
    Given a mouse packet buffer and a read count of number of bytes read,
    return the delta x and y values of the mouse.
    :param buffer: the buffer containing the packet data
    :param read_count: the number of bytes read from the mouse
    :return: tuple containing x and y delta values
    """
    if read_count == 4:
        delta_x = buffer[1]
        delta_y = buffer[2]
    elif read_count == 8:
        delta_x = buffer[2]
        delta_y = buffer[4]
    else:
        raise ValueError(f"Unsupported mouse packet size:
{read_count}, must be 4 or 8")
    return delta_x, delta_y
```

```
def atexit_callback():
    """
    re-attach USB devices to kernel if needed, and set
    autoreload back to the original state.
    :return:
    """
    for _i, _mouse in enumerate(mice):
        if kernel_driver_active_flags[_i]:
            if not _mouse.is_kernel_driver_active(0):
                _mouse.attach_kernel_driver(0)
    supervisor.runtime.autoreload = original_autoreload_val


atexit.register(atexit_callback)

# main loop
while True:

    # if set has been called
    if match3_game.cur_state == STATE_PLAYING_SETCALLED:
        # update the progress bar ticking down
        match3_game.update_active_turn_progress()

    # loop over the mice objects
    for i, mouse in enumerate(mice):
        mouse_tg = mouse_tgs[i]
        # attempt mouse read
        try:
            # read data from the mouse, small timeout so we move on
            # quickly if there is no data
            data_len = mouse.read(
                mouse_endpoint_addresses[i], mouse_bufs[i], timeout=10
            )
            mouse_deltas = get_mouse_deltas(mouse_bufs[i], data_len)
            # if we got data, then update the mouse cursor on the display
            # using min and max to keep it within the bounds of the display
            mouse_tg.x = max(
                0,
                min(
                    display.width // scale_factor - 1, mouse_tg.x +
mouse_deltas[0] // 2
                ),
            )
            mouse_tg.y = max(
                0,
                min(
                    display.height // scale_factor - 1,
                    mouse_tg.y + mouse_deltas[1] // 2,
                ),
            )

        # timeout error is raised if no data was read within the allotted timeout
        except usb.core.USBTimeoutError:
            pass

        # update the mouse debouncers
        mouse_debouncers[i].update()

        try:
            # if the current mouse is right-clicking
            if is_right_mouse_clicked(mouse_bufs[i]):
                # let the game object handle the right-click
                match3_game.handle_right_click(i)

            # if the current mouse left-clicked
            if mouse_debouncers[i].rose:
                # get the current mouse coordinates
                coords = (mouse_tg.x, mouse_tg.y, 0)
```

```
            # if the current state is GAMEOVER
            if match3_game.cur_state != STATE_GAMEOVER:
                # let the game object handle the click event
                match3_game.handle_left_click(i, coords)
            else:
                # if the mouse point is within the play again
                # button bounding box
                if play_again_btn.contains(coords):
                    # set next code file to this one
                    supervisor.set_next_code_file(__file__)
                    # reload
                    supervisor.reload()

                # if the mouse point is within the exit
                # button bounding box
                if exit_btn.contains(coords):
                    supervisor.reload()

    # if the game is over
    except GameOverException:
        # check for a winner
        winner = None
        if match3_game.scores[0] > match3_game.scores[1]:
            winner = 0
        elif match3_game.scores[0] < match3_game.scores[1]:
            winner = 1

        # if there was a winner
        if winner is not None:
            # show a message with the winning player
            message = f"\nGame Over\nPlayer{winner + 1} Wins!"
            game_over_label.color = colors[winner]
            game_over_label.text = message

        else:  # there wasn't a winner
            # show a tie game message
            message = "\nGame Over\nTie Game Everyone Wins!"

        # make the gameover group visible
        game_over_group.hidden = False

        # delete the autosave file.
        os.remove(save_to)
```

Code Explanation

The code for the Match3 game is thoroughly commented with explanations of what each line or section are for. This page will provide a higher level summary of the major components.

Hardware Principals

This game is designed around a few primary hardware peripherals: the HSTX connector with a DVI breakout for the display, a CH334F USB Hub and two USB Mice connected for player input.

HSTX Display

The code will use the built-in display from `supervisor.runtime.display`. Depending on what version of CircuitPython you have, the default size that the

display is configured for can be different. The code for this game checks the size of the display and automatically scales up the `main_group` by `2` if it finds that it is running on the larger display resolution. That allows the game to look visually the same on the screen no matter which size it is configured for.

```
scale_factor = 1
if display.width > 360:
    scale_factor = 2
    main_group.scale = scale_factor
```

The `scale_factor` variable is also used in various other calculations within the code to account for the differences in resolution.

USB Mice

The USB mouse setup, data reading, and cursor movement are all documented on the Wrangling Two Mice guide page (https://adafru.it/1ahB). See that page for more details.

Helper Classes

Inside of **match3_game_helpers.py** there are 4 class definitions and 2 functions which help with various parts of the game. A brief description of each can be found below. The code for each contains comments explaining them in more depth.

- `Match3Card` - This class encapsulates the graphics and functionality for a single Match3 card. It holds the `TileGrid` and `TilePaletteMapper` that work together to draw the card visually. See the Sprites & Colors guide page (https://adafru.it/1ahC) for more details on how they are used. When initialized, `Match3Card` is passed `card_tuple` as an argument. `card_tuple` contains four numbers that each range from `0` to `2`, this is a numerical representation of the unique card within the deck. The four numbers represent: color, shapes, fill, and count respectively. There is a commented block in the code which contains a table showing what all the possible values mean. `Match3Card` also has a `contains()` function which accepts x,y coordinates and returns `True` if the coordinate point is within the bounding box of the card.
- `Match3Game` - The `Match3Game` class manages most of the functionality of the game. It extends `Group` and holds all of the other primary visual elements of the game. It uses a state machine with four possible states.
    - Title State - Showing the title screen, waiting for the user to click either the resume or new game buttons.
    - Open Playing State - One of two game play states. This one is when the play is open meaning any player may call set by right clicking their mouse. During this state players may also click the "no set" button to indicate they

can't find a set and would like more cards dealt. Both players must click before the cards are actually dealt.
    - Playing Set Called State - Once set is called a countdown with a visible progress bar begins and the player who called must click three cards that they believe makes a set before it runs out.
    - Game Over State - This state is used at the end of the game once there are no more cards in the deck and both players have declared they can find no more sets.

`Match3Game` uses a `GridLayout` to arrange the cards on the display. Mouse click events are sent in to the Game object with the functions `handle_right_click()` and `handle_left_click()` which carry out the appropriate action based on the current state. There are functions for saving and loading the game state which use the same technique shown on the Game Mechanics: Autosave & Resume guide page (https://adafru.it/1ahD).

- `GameOverException` - A simple `Exception` wrapper class that gets raised when the game has ended.
- `Match3TitleScreen` - A class that extends `Group` and contains all of the visual elements used for the title screen. The class holds visual elements only, it does not handle functionality of the title screen.

Helper Functions

The two helper functions inside of **match3_game_helpers.py** are used by `Match3Game`. They are:

- `random_selection()` - A function that accepts a `list` and an int `count` as arguments. It selects `count` items from `list` and removes them from the `list` then returns them. It gets used when the game needs to deal cards from the deck.
- `validate_set()` - This function accepts three `Match3Card` instances and returns True if they make up a valid set. It uses matrix addition and modulus on the `card_tuple` attribute values to determine whether the cards are a valid set. This technique is detailed in an excellent episode of Numberphile on Youtube (https://adafru.it/1ahE). The video was instrumental in the making of this implementation of the game. Specifically the details of the numerical representation, and mathematical way to check for sets start around 5:40 into the video, but the whole thing is well worth a watch.

Usage

Ensure that the CH334F USB Hub has two USB mice plugged into it, and is connected to the USB Host port on the Metro RP2350 that was wired up previously. Reset the board by cycling power or pressing the Reset button if you happen to plug in a mouse after power up.

Be sure you connect the DVI breakout to an HDMI monitor and the monitor is on. You might need a long cable if your monitor is not near the Metro RP2350 (like a television). The cables are standard and may be obtained from any trusted retail outlet. Also reset the Metro if you plug in HDMI after powering the Metro.

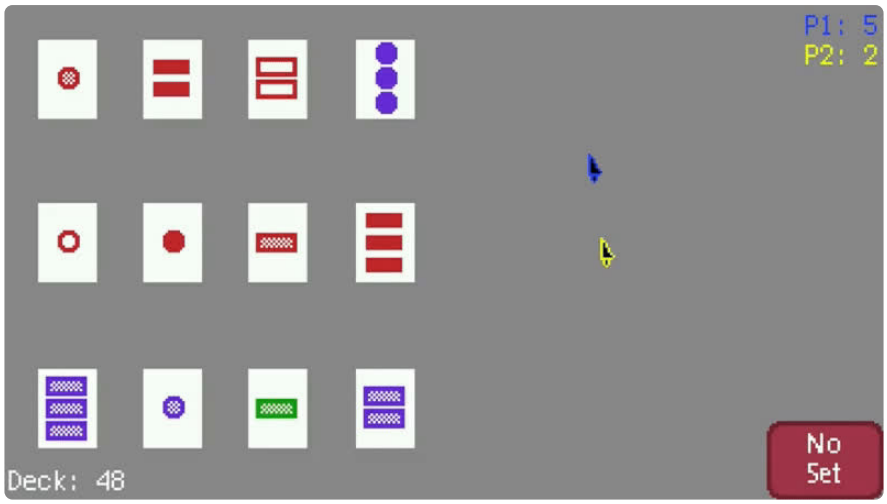
Once connections are all set, power the Metro RP2350 either via USB C (5 volts) or the barrel connection (5.5 to 17 volts DC, center positive).

Gameplay

When the game first launches the title screen is shown. The title screen always contains a "New Game" button which will clear out any previously saved data and begin a new game. If there is previously saved data, then the title screen will also have a "Resume" button which can be clicked to pick up where the previous game left off.

During the game, players look at the cards on the board, trying to find valid sets of 3. Once a player spots a set, they click the right mouse to buzz in and begin their countdown. They then click the 3 cards that make up the set they found. If the set was valid, they are awarded a point and the cards are removed. If it wasn't a set, or they run out of time, a point is deducted.

During open play, while no one has called set, the players may click on the "No Set" button in the bottom right corner to indicate they can't find a set and want more cards to be dealt. Once both players click the button, the game will will deal an additional 3 cards, if there is room on the board, or shuffle the cards on the board back in and deal 12 new ones if the board was full.



If both the deck and board are empty, then the game is over and winner is determined by the high score. A more likely game ending occurs if the deck is empty, some cards remain on the board, but both players have clicked "No Set" indicating they cannot find any sets in the cards that do remain.