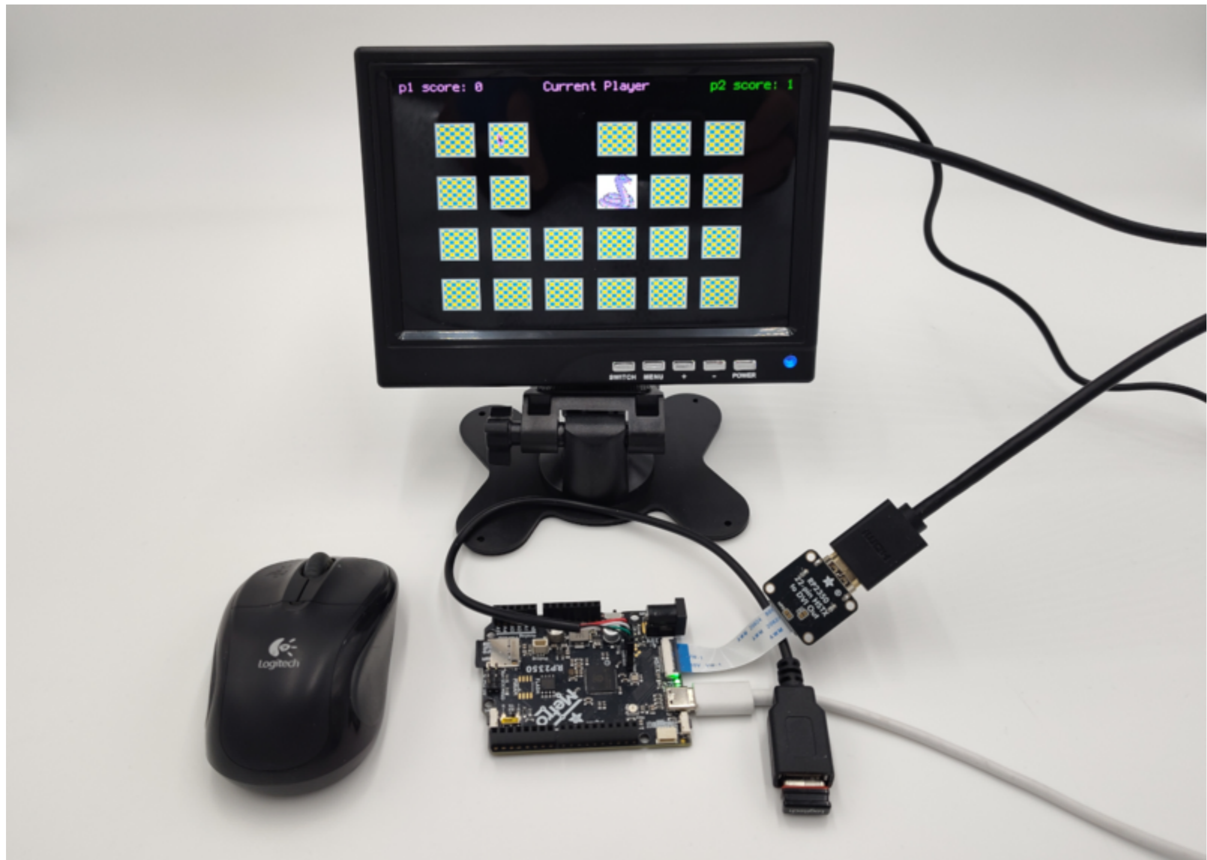
Create a Memory Game on Metro RP2350

Created by Tim C



https://learn.adafruit.com/create-a-memory-game-on-metro-rp2350

Last updated on 2025-04-03 10:10:02 PM EDT

Table of Contents

Overview 3

- Parts

Preparing the Metro RP2350 7

- HSTX Connection to DVI

Install CircuitPython 10

- CircuitPython Quickstart
- Safe Mode
- Flash Resetting UF2

Mouse Input 13

- Display Setup
- Visual Elements Setup
- USB Mouse Device Setup
- Main Loop

Game Mechanics: Multiplayer Turns 17

- Visual Elements Setup
- Player Turn Setup & Logic
- check_winner() Function
- Main Loop

Code 23

- CircuitPython Usage
- Display Size
- Drive Structure
- Code

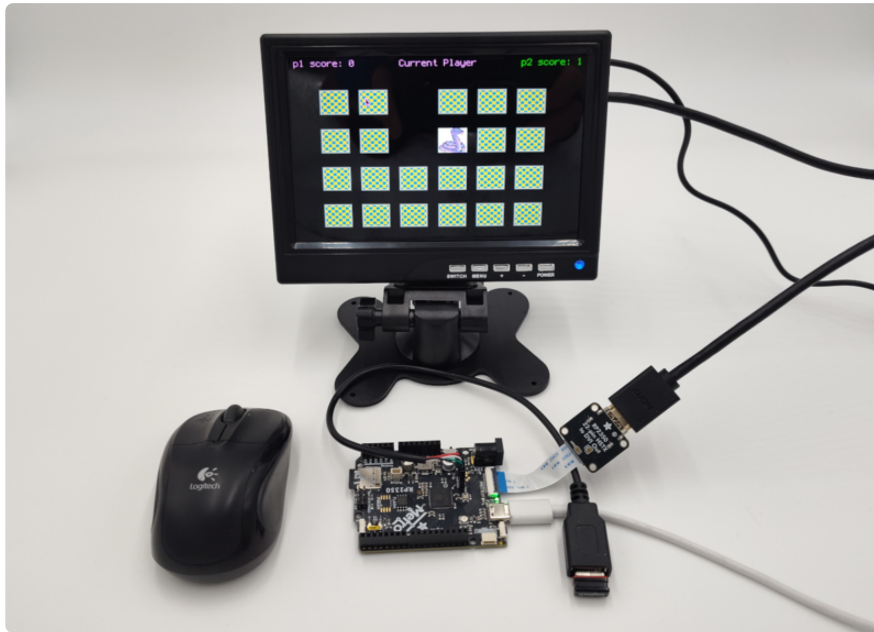
Code Explanation 32

- Hardware Principals
- USB Mouse
- Helper Functions
- State Machine
- Player Specific Variables
- Initializing Cards
- Flipping Cards
- Check If the Game Is Over

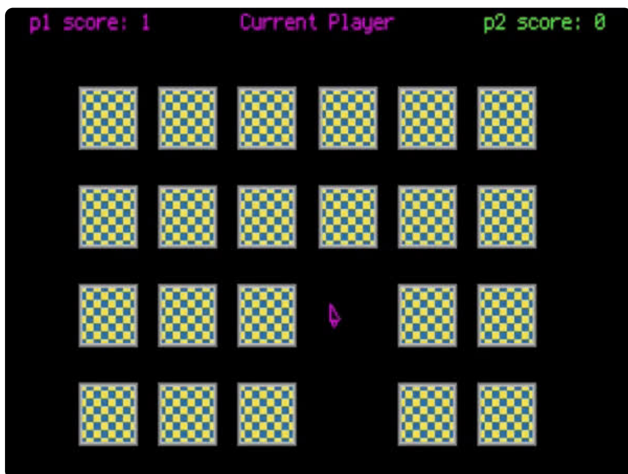
Usage 36

- Gameplay

Overview



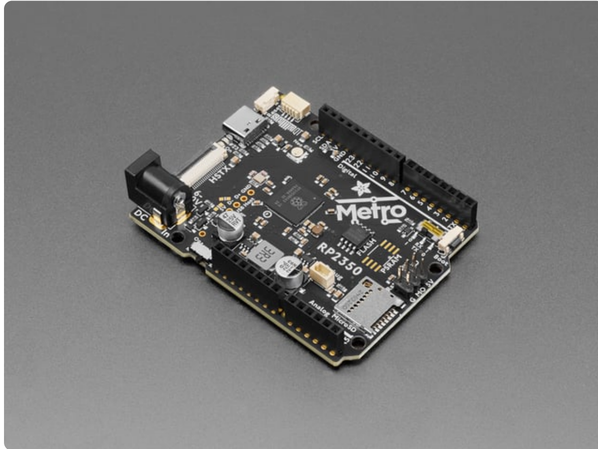
The Metro RP2350 makes a perfect little game console. The on-board HSTX combined with a DVI breakout can output to a standard television or computer monitor for the display. The USB host connection makes it easy to take input from a mouse to control the game.



This game is an implementation of the classic game Memory (https://adafru.it/1ag9) also known as Concentration. A USB mouse is used by the players to alternate turns clicking on cards to flip them over, revealing the front. Players are rewarded with a point and an extra turn for finding two matching cards.

Play continues until all matches have been found. The player with the most points wins.

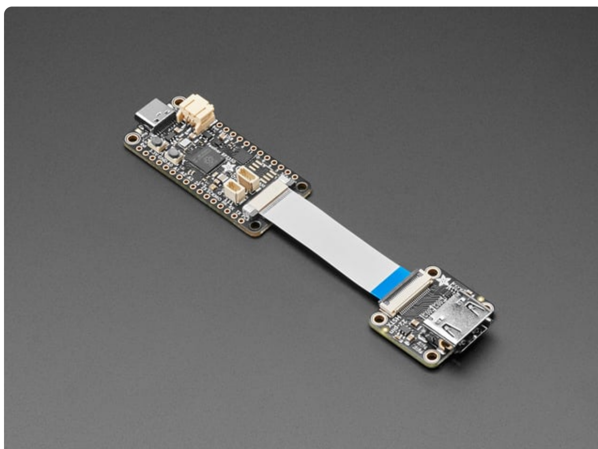
Parts



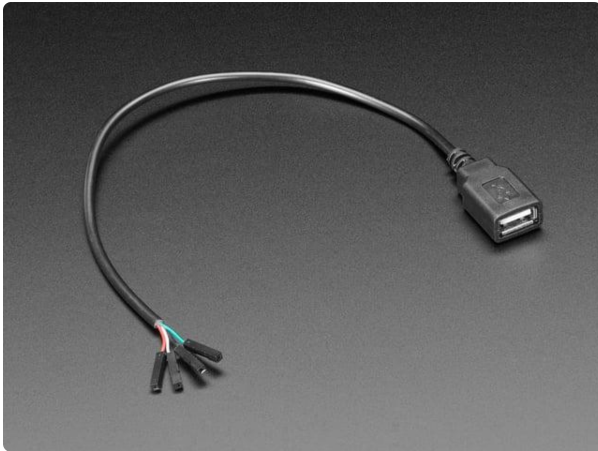
Adafruit Metro RP2350
Choo! Choo! This is the RP2350 Metro Line, making all station stops at "Dual Cortex M33 mountain", "528K RAM round-about" and "16 Megabytes of Flash...
https://www.adafruit.com/product/6003

Or

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



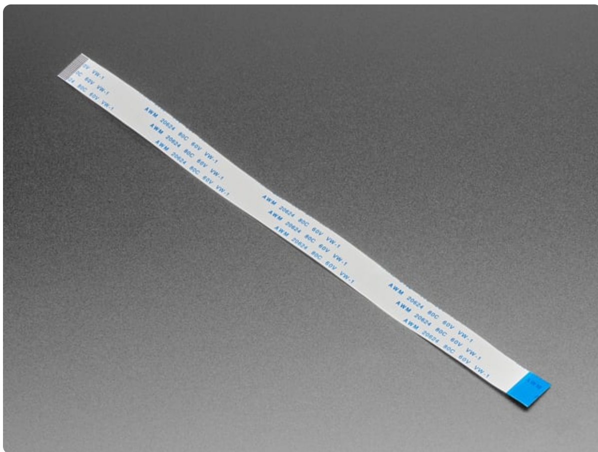
Adafruit RP2350 22-pin FPC HSTX to DVI Adapter for HDMI Displays
You may have noticed that our RP2350 Feather has an FPC output connector on the end for accessing the HSTX (High Speed...
https://www.adafruit.com/product/6055



[USB Type A Jack Breakout Cable with Premium Female Jumpers](https://www.adafruit.com/product/4449)

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...

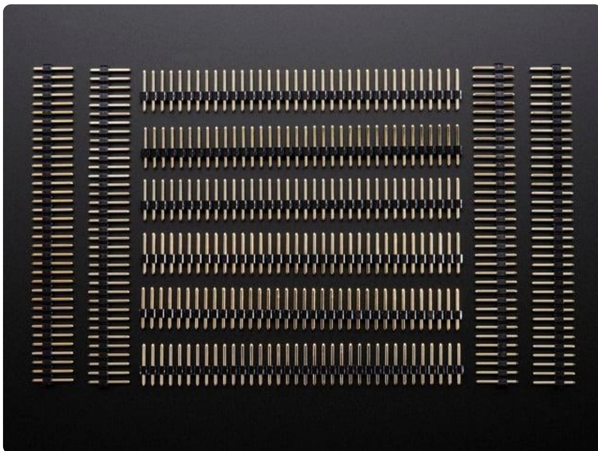
https://www.adafruit.com/product/4449



[22-pin 0.5mm pitch FPC Flex Cable for DSI CSI or HSTX - 20cm](https://www.adafruit.com/product/6036)

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...

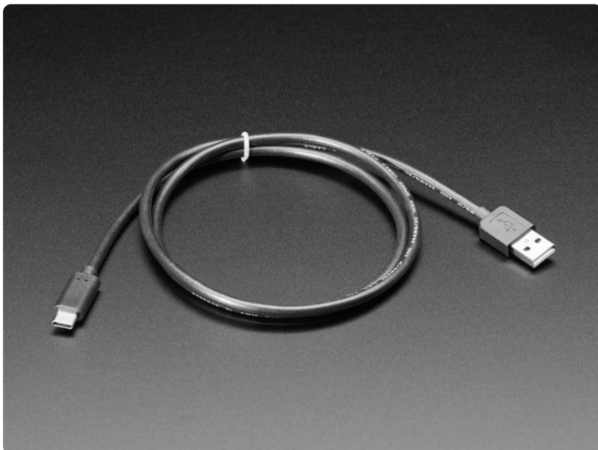
https://www.adafruit.com/product/6036



[Break-away 0.1" 36-pin strip male header - Black - 10 pack](https://www.adafruit.com/product/392)

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



[USB Type A to Type C Cable - approx 1 meter / 3 ft long](https://www.adafruit.com/product/4474)

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

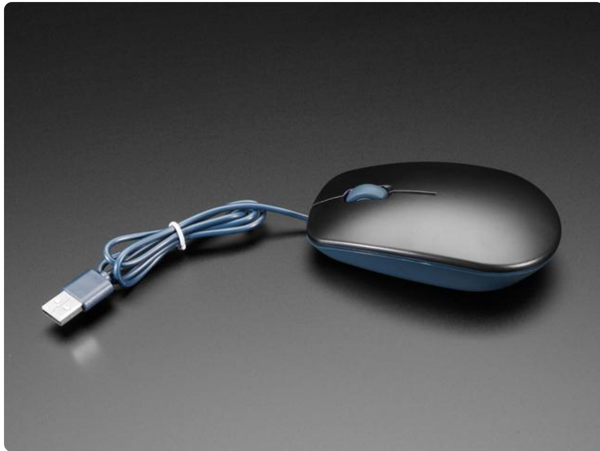
[HDMI Cable - 1 meter](https://www.adafruit.com/product/608)
Connect two HDMI devices together with this basic HDMI cable. It has nice molded grips for easy installation, and is 1 meter long (about 3 feet). This is a HDMI 1.3...
https://www.adafruit.com/product/608

\- or -

HDMI Flat Cable - 1 foot / 30cm long
Connect two HDMI devices together and save space with this basic flat HDMI 1.4 cable. It has nice molded grips for easy installation, and is 1 foot long (~30 cm).This cable is...
https://www.adafruit.com/product/2197

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

\- or -



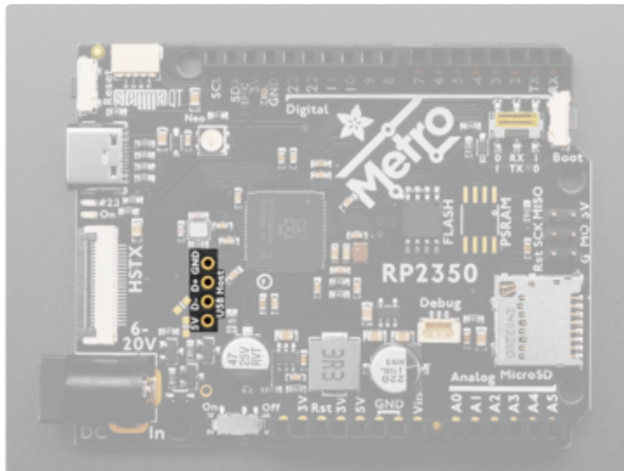
[Official Raspberry Pi USB Optical Mouse - Black and Gray](https://www.adafruit.com/product/4113)

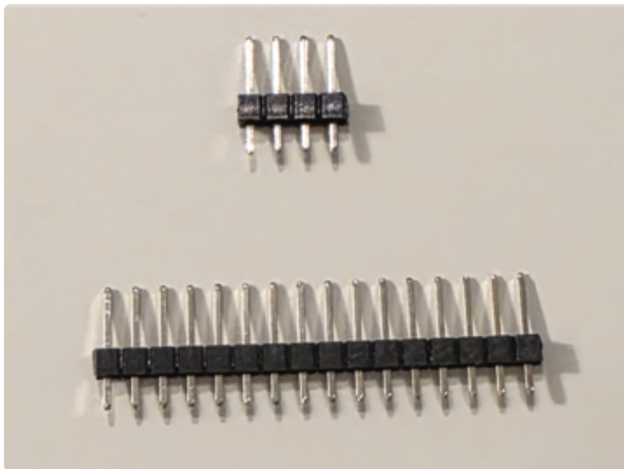
This is a mouse. But, hark, it's no generic mouse. It's the Official Raspberry Pi Mouse!It's optical for good resolution and precision, three...

https://www.adafruit.com/product/4113

Preparing the Metro RP2350

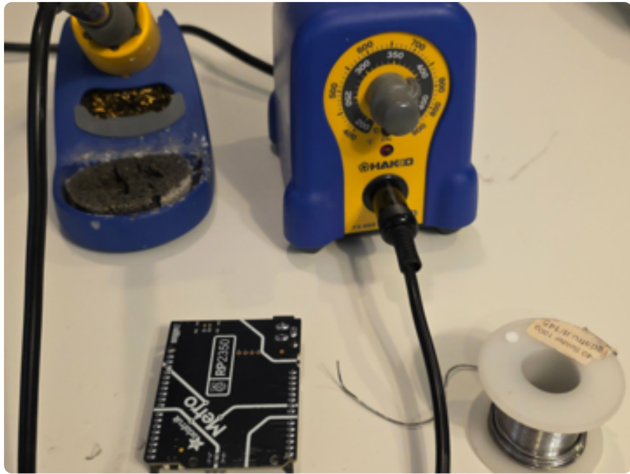
The USB Host port is the only part of this project that required soldering.

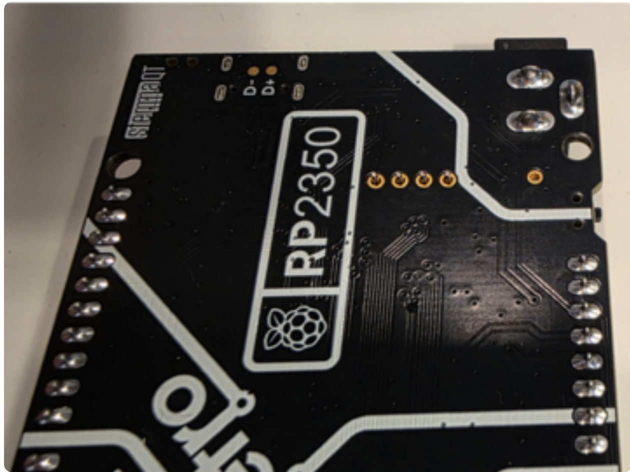


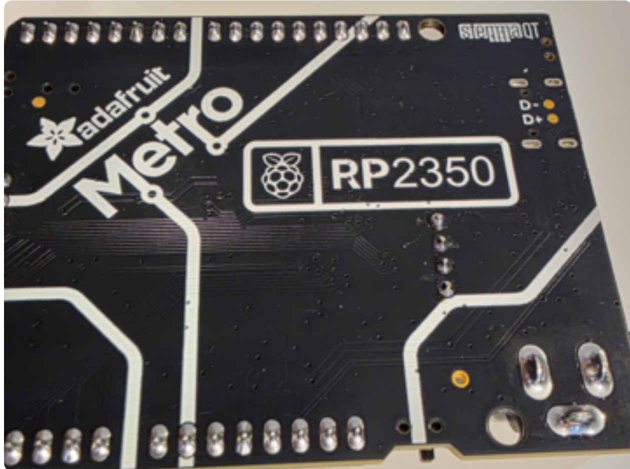


The USB Host pin connections are highlighted on the Metro image to the left. You will need a small piece of standard 0.1 inch male header, with 4 pins, to fit the holes.

You can cut header with diagonal cutters or break them with pliers or even your fingers. Just be sure to wear eye protection as they can fly when cut.

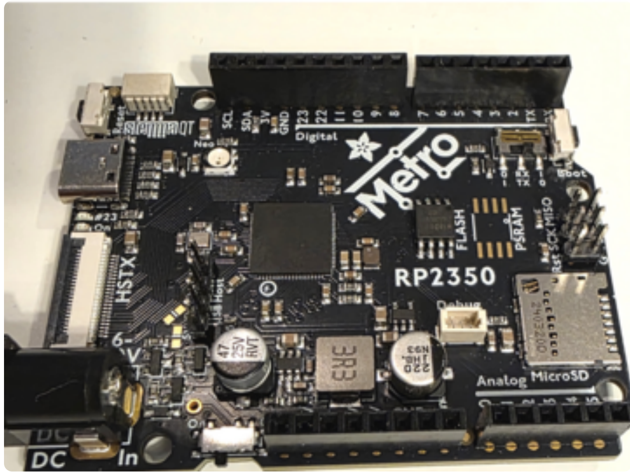






Put the short end of the header into the holes in the Metro marked USB Host and secure them with putty, blutack, tape, etc. Turn the Metro over and you should see the header barely poking out of the bottom of the board. If the pins stick through a great deal you may have the header pins upside down,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

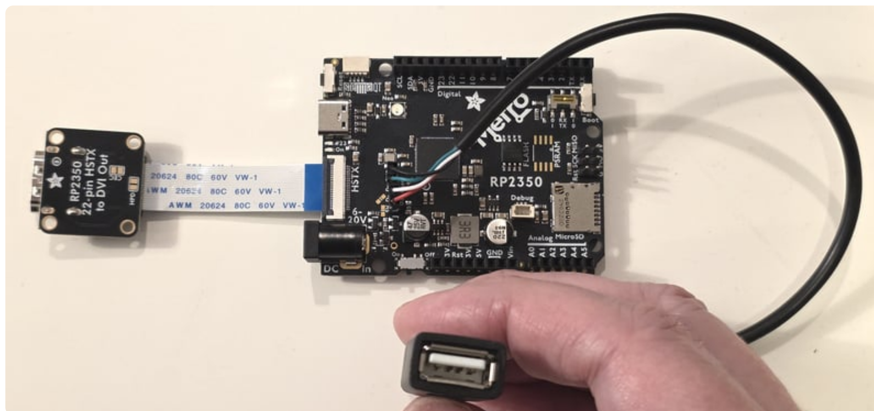
Get the USB Host cable and wire as follows:

GRD to **Black**

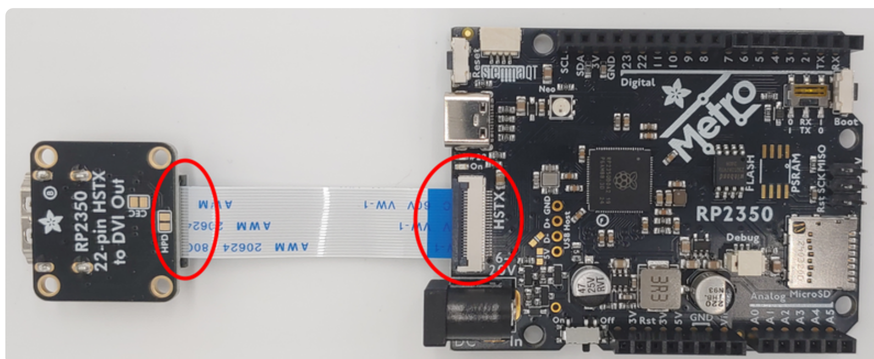
D+ to **Green**

D- to **White**

5V to **Red**



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.

Install CircuitPython

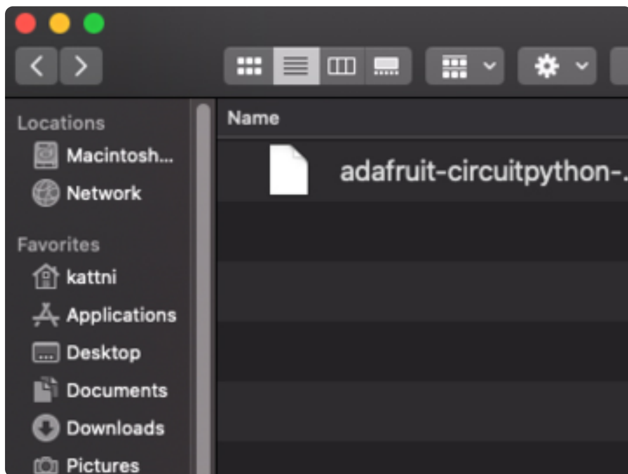
[CircuitPython](https://adafru.it/tB7) (https://adafru.it/tB7) is a derivative of [MicroPython](https://adafru.it/BeZ) (https://adafru.it/BeZ) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop 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.

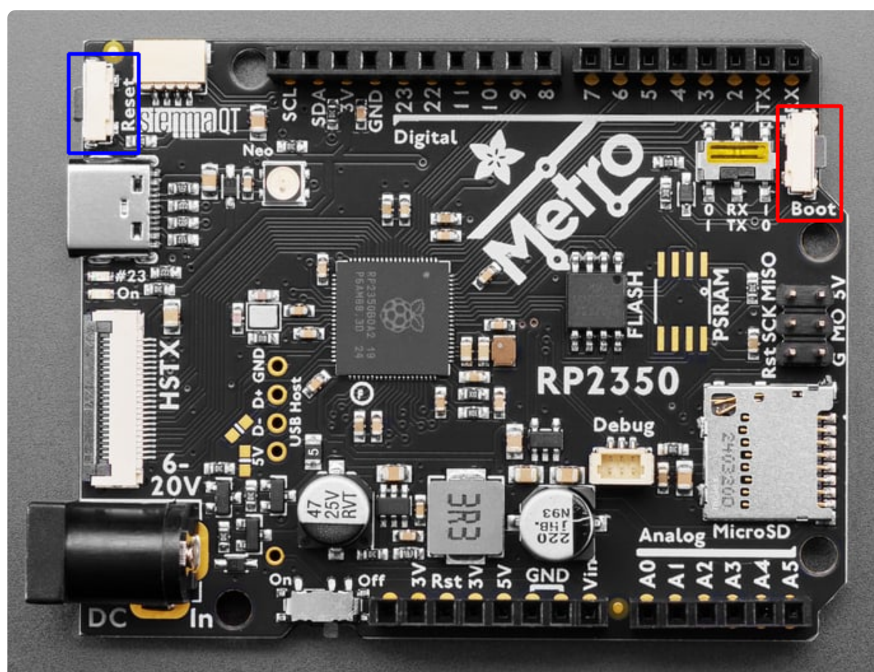
Download the latest version of CircuitPython for this board via circuitpython.org

https://adafru.it/1aeL



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

Save it wherever is convenient for you.

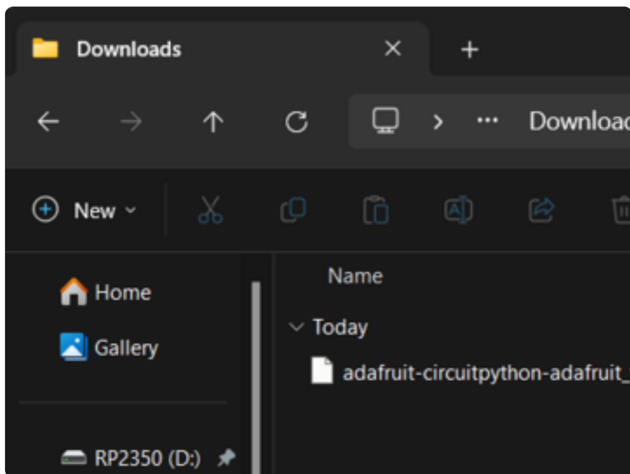


To enter the bootloader, hold down the **BOOT/BOOTSEL button** (highlighted in red above), and while continuing to hold it (don't let go!), press and release the **reset button** (highlighted in red or blue above). **Continue to hold the BOOT/BOOTSEL button until the RP2350 drive appears!**

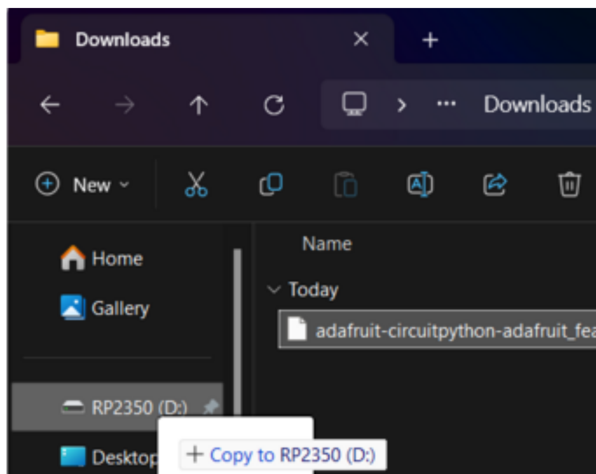
If the drive does not appear, release all the buttons, and then repeat the process above.

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.

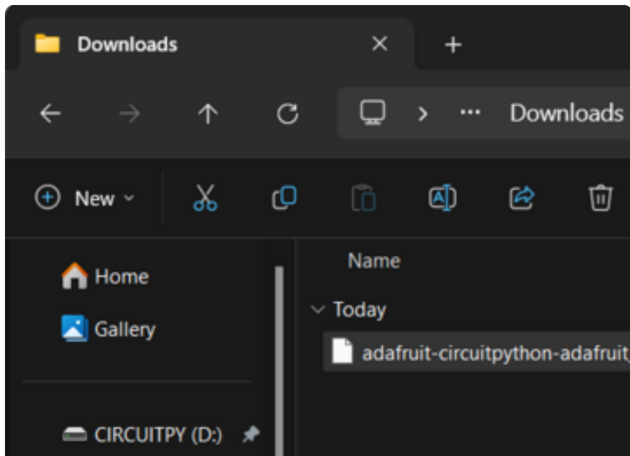
A lot of people end up using charge-only USB cables and it is very frustrating! **Make sure you have a USB cable you know is good for data sync.**



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

Mouse Input

The Memory game will make use of a USB mouse for input from the players. To start, look at the following basic example of initializing the mouse, reading data from it, and plotting a cursor on the screen.

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 your **CIRCUITPY** drive to run the example.

```
# SPDX-FileCopyrightText: 2025 Tim Cocks for Adafruit Industries
# SPDX-License-Identifier: MIT
"""
This example is made for a basic Microsoft optical mouse with
two buttons and a wheel that can be pressed.

It assumes there is a single mouse connected to USB Host,
and no other devices connected.
"""
import array
from displayio import Group, OnDiskBitmap, TileGrid
from adafruit_display_text.bitmap_label import Label
import supervisor
import terminalio
import usb.core

# pylint: disable=ungrouped-imports
if hasattr(supervisor.runtime, "display") and supervisor.runtime.display is not
None:
```

```
    # use the built-in HSTX display for Metro RP2350
    display = supervisor.runtime.display
else:
    # pylint: disable=ungrouped-imports
    from displayio import release_displays
    import picodvi
    import board
    import framebufferio

    # initialize display
    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=16,
    )
    display = framebufferio.FramebufferDisplay(fb)

# group to hold visual elements
main_group = Group()

# make the group visible on the display
display.root_group = main_group

# load the mouse cursor bitmap
mouse_bmp = 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 = 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

# text label to show the x, y coordinates on the screen
output_lbl = Label(
    terminalio.FONT, text=f"{mouse_tg.x},{mouse_tg.y}", color=0xFFFFFF, scale=1
)

# move it to the upper left corner
output_lbl.anchor_point = (0, 0)
output_lbl.anchored_position = (1, 1)

# add it to the main group
main_group.append(output_lbl)

# add the mouse tile grid to the main group
main_group.append(mouse_tg)

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

# 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)

# 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]))

    # string with updated coordinates for the text label
    out_str = f"{mouse_tg.x},{mouse_tg.y}"

    # loop over the button names
    for i, button in enumerate(BUTTONS):
        # check if each button is pressed using bitwise AND shifted
        # to the appropriate index for this button
        if buf[0] & (1 << i) != 0:
            # append the button name to the string to show if
            # it is being clicked.
            out_str += f" {button}"

    # update the text label with the new coordinates
    # and buttons being pressed
    output_lbl.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 Memory game.

Display Setup

First the code checks whether there is a built-in display. For the Metro RP2350 starting with CircuitPython 9.2.5 the HSTX connector counts as the built-in display, so it will get used by this script by default. If a display is not found, then it will attempt to initialize one manually using the default HSTX pins.

Visual Elements Setup

Next the code creates a `main_group` to put all visual elements into and sets it as the `root_group` on the display so it is shown on the screen. Then it creates an `OnDiskBitmap` to load the **mouse_cursor.bmp** file.

This files contains a pink color in the palette index 0 which is treated as transparency by calling `mouse_bmp.pixel_shader.make_transparent(0)`. A TileGrid is created and stored in the variable `mouse_tg`. Later on when reading mouse data, the program will use it to move `mouse_tg` around the screen. The mouse cursor is put into the center of the screen to start with.

A `Label` named `output_lbl` is created and added to the `main_group` after being placed in the top left corner of the screen. This will be used to show the current mouse coordinates and any buttons that are pressed.

The `mouse_tg` is added to `main_group` last so that it will be visually in front of everything else.

USB Mouse Device Setup

The list `BUTTONS` is created with values `"left"`, `"right"`, and `"middle"`. The order and indexes of these values align with the mouse protocol which will use bits in the same positions in order to denote whether each button is being pressed or not.

Next the code scans for connected USB devices and sets the first one found to the `mouse` variable. Once found, the connected mouse is detached from the kernel driver, if needed. Next `mouse.set_configuration()` is called in order to get the mouse ready to send data.

An `array` of bytes that can hold 4 elements is created to store the data that is read from the mouse device.

Main Loop

Inside the main loop, `mouse.read(0x81, buf, timeout=10)` is called to read data from the mouse. `0x81` is the default endpoint address for basic HID mice, `buf` is the 4 byte buffer array that will get filled with the data that is read. `timeout` is how many milliseconds to wait before raising a `USBTimeoutError` if there is no data to read. This code uses a low value of `10` milliseconds to illustrate how the main loop can do other things if the timeout is kept low. Higher timeout values result in the `read()` call blocking other code execution for longer times.

If there is no data, the `USBTimeoutError` is raised and the code skips to the next iteration with `continue`.

If there is data, the code reads the delta x and y values from the buffer indexes `1` and `2`. These delta values represent how far the mouse has moved in each direction. It will have negative values for up/left, and positive values for right/down.

The delta values are used to move the `mouse_tg` to a new location. `min()` and `max()` are used to clamp the mouse cursor to the bounds of the display. The mouse itself is not aware of these bounds, the code enforces staying on the display after reading raw data from the mouse.

The `out_str` is updated with the current x and y coordinates of the `mouse_tg`.

Next the code checks for button presses by looping over the `BUTTONS` list and checking the bits in the respective positions within the byte at index `0` of the buffer. Any buttons that are pressed have their name added to the `out_str`.

Finally `out_str` is set as the text on the `output_lbl` in order to show the current values on the screen.

Game Mechanics: Multiplayer Turns

In the Memory game, multiple players take turns playing the game. Each player will need to use the mouse to select cards to flip over and when their turn is over, the mouse is passed to the other player.

The code will need to keep track of which turn it is to assign points to the appropriate player when matching cards are found.

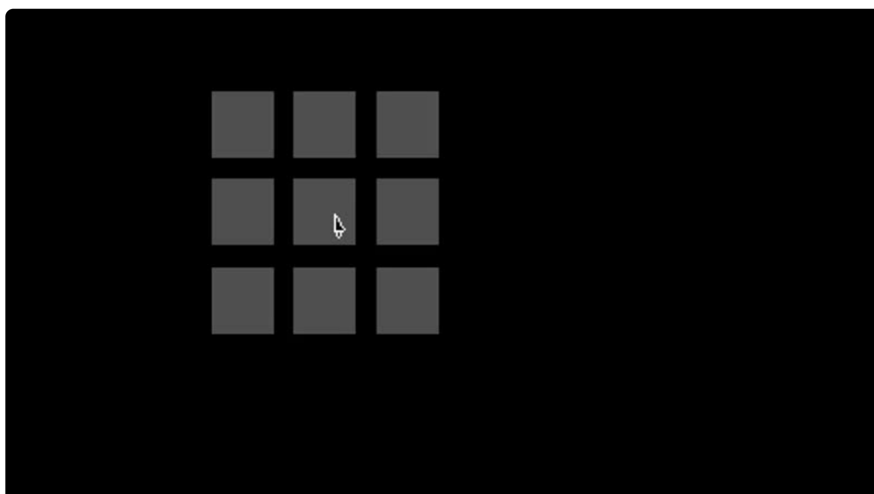
Before getting into all of the complexity that comes with the rest of the Memory game, a look at something simpler: an implementation of Tic-Tac-Toe. This Tic-Tac-Toe game will use the same core mechanism to keep track of player turns as the Memory game. As an added bonus, it also uses the same `GridLayout` technique to position its visual elements on the screen as the Memory game.

```
# SPDX-FileCopyrightText: 2025 Tim Cocks for Adafruit Industries
# SPDX-License-Identifier: MIT
"""
This example is made for a basic Microsoft optical mouse with
two buttons and a wheel that can be pressed.

It assumes there is a single mouse connected to USB Host,
and no other devices connected.

It illustrates multi-player turn based logic with a very
basic implementation of tic-tac-toe.
"""
import array
import random
from displayio import Group, OnDiskBitmap, TileGrid
from adafruit_display_text.bitmap_label import Label
from adafruit_displayio_layout.layouts.grid_layout import GridLayout
import supervisor
import terminalio
import usb.core

# pylint: disable=ungrouped-imports
if hasattr(supervisor.runtime, "display") and supervisor.runtime.display is not
None:
    # use the built-in HSTX display for Metro RP2350
    display = supervisor.runtime.display
else:
    from displayio import release_displays
    import picodvi
    import board
    import framebufferio

    # initialize display
    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=16,
    )
    display = framebufferio.FramebufferDisplay(fb)

# group to hold visual elements
main_group = Group()

# make the group visible on the display
display.root_group = main_group

# load the mouse cursor bitmap
mouse_bmp = 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 = 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
```

```
# text label to show the x, y coordinates on the screen
output_lbl = Label(terminalio.FONT, text="", color=0xFFFFFF, scale=1)

# move it to the right side of the screen
output_lbl.anchor_point = (0, 0)
output_lbl.anchored_position = (180, 40)

# add it to the main group
main_group.append(output_lbl)

# 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)

# set up a 3x3 grid for the tic-tac-toe board
board_grid = GridLayout(x=40, y=40, width=128, height=128, grid_size=(3, 3))

# load the tic-tac-toe spritesheet
tictactoe_spritesheet = OnDiskBitmap("tictactoe_spritesheet.bmp")

# X is index 1 in the spritesheet, O is index 2 in the spritesheet
player_icon_indexes = [1, 2]

# current player variable.
# When this equlas 0 its X's turn,
# when it equals 1 it is O's turn.
current_player_index = random.randint(0, 1)  # randomize the initial player

# loop over rows
for y in range(3):
    # loop over columns
    for x in range(3):
        # create a TileGrid for this cell
        new_tg = TileGrid(
            bitmap=tictactoe_spritesheet,
            default_tile=0,
            tile_height=32,
            tile_width=32,
            height=1,
            width=1,
            pixel_shader=tictactoe_spritesheet.pixel_shader,
        )

        # add the new TileGrid to the board grid at the current position
        board_grid.add_content(new_tg, grid_position=(x, y), cell_size=(1, 1))

# add the board grid to the main group
main_group.append(board_grid)

# add the mouse tile grid to the main group
main_group.append(mouse_tg)
```

```python
def check_for_winner():
    """
    check if a player has won

    :return: the player icon index of the winning player,
        None if no winner and game continues, -1 if game ended in a tie.
    """
    found_empty = False

    # check rows
    for row_idx in range(3):
        # if the 3 cells in this row match
        if (
            board_grid[0 + (row_idx * 3)][0] != 0
            and board_grid[0 + (row_idx * 3)][0]
            == board_grid[1 + (row_idx * 3)][0]
            == board_grid[2 + (row_idx * 3)][0]
        ):
            return board_grid[0 + (row_idx * 3)][0]

        # if any of the cells in this row are empty
        if 0 in (
            board_grid[0 + (row_idx * 3)][0],
            board_grid[1 + (row_idx * 3)][0],
            board_grid[2 + (row_idx * 3)][0],
        ):
            found_empty = True

    # check columns
    for col_idx in range(3):
        # if the 3 cells in this column match
        if (
            board_grid[0 + col_idx][0] != 0
            and board_grid[0 + col_idx][0]
            == board_grid[3 + col_idx][0]
            == board_grid[6 + col_idx][0]
        ):
            return board_grid[0 + col_idx][0]

        # if any of the cells in this column are empty
        if 0 in (
            board_grid[0 + col_idx][0],
            board_grid[3 + col_idx][0],
            board_grid[6 + col_idx][0],
        ):
            found_empty = True

    # check diagonals
    if (
        board_grid[0][0] != 0
        and board_grid[0][0] == board_grid[4][0] == board_grid[8][0]
    ):
        return board_grid[0][0]

    if (
        board_grid[2][0] != 0
        and board_grid[2][0] == board_grid[4][0] == board_grid[6][0]
    ):
        return board_grid[2][0]

    if found_empty:
        # return None if there is no winner and the game continues
        return None
    else:
        # return -1 if it's a tie game with no winner
        return -1
```

```
# 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 button clicked
    if buf[0] & (1 << 0) != 0:
        # get the mouse pointer coordinates accounting for the offset of
        # the board grid location
        coords = (mouse_tg.x - board_grid.x, mouse_tg.y - board_grid.y, 0)

        # loop over all cells in the board
        for cell_tg in board_grid:

            # if the current cell is blank, and contains the clicked coordinates
            if cell_tg[0] == 0 and cell_tg.contains(coords):
                # set the current cell tile index to the current player's icon
                cell_tg[0] = player_icon_indexes[current_player_index]

                # change to the next player
                current_player_index = (current_player_index + 1) % 2

                # print out which player's turn it is
                print(f"It is now {'X' if current_player_index == 0 else 'O'}'s
turn")

                # check for a winner
                result = check_for_winner()

                # if Xs or Os have won
                if result == 1:
                    output_lbl.text = "X is the winner!"
                elif result == 2:
                    output_lbl.text = "O is the winner!"

                # if it was a tie game
                elif result == -1:
                    output_lbl.text = "Tie game, no winner."
```



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

The code makes use of things detailed on the [Mouse Input guide page](https://adafru.it/1aga) (https://adafru.it/1aga), see that page for more information about how the mouse data is read and the cursor drawn and moved around the display.

Visual Elements Setup

In addition to the mouse cursor, this code also creates `board_grid`, a `GridLayout` object, which will hold the cells that make up the Tic-Tac-Toe board. The `GridLayout` takes a `height` and `width` in pixels and a `grid_size` when it is initialized and will automatically distribute the content items added into the cells of the grid using the values specified to determine the appropriate placement.

The **tictactoe_spritesheet.bmp** is loaded into an `OnDiskBitmap`.

A set of nested for loops with `y` and `x` counter variables are used to create a `TileGrid` for each cell on the board and add it to the `board_grid`. Each `TileGrid` is set to `default_tile=0` which is the index of the blank cell within the spritesheet.

Player Turn Setup & Logic

An integer variable, `current_player_index`, is created to store a number which will keep track of which player's turn it is. Tic-Tac-Toe and Memory are both 2 player games, but the same approach can be used for games with more than two players as well.

For Memory and Tic-Tac-Toe, the `current_player_index` will always be either `0` or `1`. When the time comes to change turns, the code will swap from the current value to the other.

Initially the value of this variable is set to a random number using `random.randint(0, 1)`.

Now that we have an integer variable that holds whose turn it is, any other player specific data or variables can be stored inside of a list and use the indexes of the list to match up with the `current_player_index` values. In the Tic-Tac-Toe code, it creates the list `player_icon_indexes` to hold the indexes within the spritesheet of the X and O tiles respectively. `current_player_index` of `0` matches up to the X sprite tile, and `current_player_index` of `1` matches up to the O sprite tile.

Whenever the current player clicks a tile to play their turn, the code sets the tile sprite by looking up the appropriate sprite index with `player_icon_indexes[current_player_index]`

Other lists can be created and used similarly to store and retrieve information specific to each player. The Memory game will make use of a few lists like this.

`check_winner()` Function

This function will check all of the horizontal, vertical and diagonal lines within the game board to find if any player has 3 in a row and thus has won. If a player does have 3 in a row then `check_winner()` will return their `current_player_index` value, i.e. if X wins then `0` will be returned, and if O wins then `1` will be returned. If there is no winner and there are still empty cells then `None` is returned. If there is no winner and there are no empty cells remaining then `-1` is returned to indicate it is a tie game.

Main Loop

The first part of the main loop reads data from the mouse and moves the `mouse_tg` just as was shown in the mouse demo on the previous page.

`if buf[0] & (1 << 0) != 0` is used to check if the left mouse button has been pressed. Nothing else happens until that button does get pressed.

When the left button is pressed, the code puts the current `mouse_tg` coordinates into a the tuple `coords` then it loops over all of the cell `TileGrid`s that are in the `board_grid` and checks if the mouse coordinates are contained within the bounding box of each by calling `cell_tg.contains(coords)`. If the click was not inside of any of the cell `TileGrid`s then nothing else happens.

If the click was inside of a cell, `TileGrid` the code ensures that the cell is currently empty by checking the sprite index. If it is empty then the current player's sprite is put into the cell `TileGrid`. Afterward the turn is changed by updating `current_player_index`.

Next `check_for_winner()` is called, if there is a winner the `output_lbl` is updated to say which player won. If it's a tie game, the `output_lbl` is updated to reflect that. If there is no winner and the game is continuing, the program goes to the next iteration of the main loop.

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.

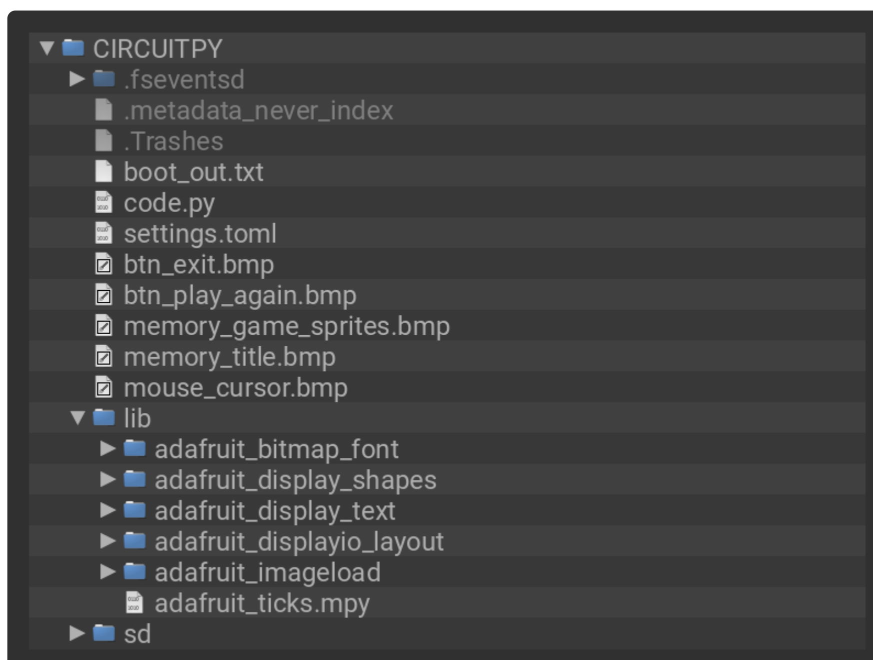
Display Size

To ensure the display gets automatically configured for the 320x240 resolution which the Memory game is made for, add a `CIRCUITPY_DISPLAY_WIDTH` variable with value `320` to the **settings.toml** file in the root directory of the **CIRCUITPY** drive. If you do not already have a **settings.toml** file, follow the instructions on [this guide page](https://adafru.it/18f9) (https://adafru.it/18f9) to create one.

```
# This file is where you keep private settings, passwords, and tokens!
# If you put them in the code you risk committing that info or sharing it

CIRCUITPY_DISPLAY_WIDTH=320
```

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.



Code

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

```
# SPDX-FileCopyrightText: 2025 Tim Cocks for Adafruit Industries
# SPDX-License-Identifier: MIT
"""
An implementation of the card game memory. Players take turns flipping
over two cards trying to find pairs. After the turn any non-pairs are
flipped face down so the players must try to remember where they are.

Players trade off using the USB mouse to play their turns.
"""
import array
import random
import time
from displayio import Group, OnDiskBitmap, TileGrid
from adafruit_display_text.bitmap_label import Label
from adafruit_display_text.text_box import TextBox
from adafruit_displayio_layout.layouts.grid_layout import GridLayout
from adafruit_ticks import ticks_ms
import supervisor
import terminalio
import usb.core


def random_selection(lst, count):
    """
    Select items randomly from a list of items.

    returns a list of length count containing the selected items.
    """
    if len(lst) <= count:
        raise ValueError("Count must be less than or equal to length of list")
    iter_copy = list(lst)
    selection = set()
    while len(selection) < count:
        selection.add(iter_copy.pop(random.randrange(len(iter_copy))))
    return list(selection)


def update_score_text():
    """
    Update the score text on the display for each player
    """
    for _ in range(2):
        out_str = f"p{_+1} score: {player_scores[_]}"
        score_lbls[_].text = out_str


# state machine constants

# title state, shows title screen waits for click
STATE_TITLE = 0

# playing state alternates players flipping cards to play the game
STATE_PLAYING = 1

# shows the game over message and waits for a button to be clicked
STATE_GAMEOVER = 2

# initial state is title screen
CUR_STATE = STATE_TITLE

# pylint: disable=ungrouped-imports
if hasattr(supervisor.runtime, "display") and supervisor.runtime.display is not
None:
```

```
    # use the built-in HSTX display for Metro RP2350
    display = supervisor.runtime.display
else:
    # pylint: disable=ungrouped-imports
    from displayio import release_displays
    import picodvi
    import board
    import framebufferio

    # initialize display
    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=16,
    )
    display = framebufferio.FramebufferDisplay(fb)

# main group will hold all the visual elements
main_group = Group()

# make main group visible on the display
display.root_group = main_group

# list of Label instances for player scores
score_lbls = []

# list of colors, one representing each player
colors = [0xFF00FF, 0x00FF00]

# randomly choose the first player
current_turn_index = random.randrange(0, 2)

# list that holds up to 2 cards that have been flipped over
# on the current turn
cards_flipped_this_turn = []

# list that holds the scores of each player
player_scores = [0, 0]

# size of the grid of cards to layout
grid_size = (6, 4)

# create a grid layout to help place cards neatly
# into a grid on the display
card_grid = GridLayout(x=10, y=10, width=260, height=200, grid_size=grid_size)

# these indexes within the spritesheet contain the
# card front sprites, there are 8 different cards total.
CARD_FRONT_SPRITE_INDEXES = {1, 2, 3, 4, 5, 6, 7, 9}

# pool of cards to deal them onto the board from
# starts with 2 copies of each of 8 different cards
pool = list(CARD_FRONT_SPRITE_INDEXES) + list(CARD_FRONT_SPRITE_INDEXES)

# select 4 cards at random that will be duplicated
duplicates = random_selection(CARD_FRONT_SPRITE_INDEXES, 4)

# add 2 copies each of the 4 selected duplicate cards
# this brings the pool to 24 cards total
```

```
pool += duplicates + duplicates

# list that represents the order the cards are randomly
# dealt out into. The board is a two-dimensional grid,
# but this list is one dimension where
# index in the list = y * width + x in the grid.
card_locations = []

# load the spritesheet for the cards
sprites = OnDiskBitmap("memory_game_sprites.bmp")

# list to hold TileGrid instances for each card
card_tgs = []

# loop over 4 rows
for y in range(4):
    # loop over 6 columns
    for x in range(6):
        # i = y * 6 + x

        # create a TileGrid
        new_tg = TileGrid(
            bitmap=sprites,
            default_tile=10,
            tile_height=32,
            tile_width=32,
            height=1,
            width=1,
            pixel_shader=sprites.pixel_shader,
        )

        # add it to the list of card tilegrids
        card_tgs.append(new_tg)

        # add it to the grid layout at the current x,y position
        card_grid.add_content(new_tg, grid_position=(x, y), cell_size=(1, 1))

        # choose a random index of a card in the pool
        random_choice = random.randrange(0, len(pool) - 1) if len(pool) > 1 else 0

        # remove the chosen card from the pool, and add it
        # to the card locations list at the current location
        card_locations.append(pool.pop(random_choice))

# center the card grid layout horizontally
card_grid.x = display.width // 2 - card_grid.width // 2

# move the card grid layout towards the bottom of the screen
card_grid.y = display.height - card_grid.height

# add the card grid to the main group
main_group.append(card_grid)

# 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=0x222222,
    width=display.width // 2,
    height=80,
    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 // 2 - game_over_label.width // 2,
    40,
)

# make it hidden, it will 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 // 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 = 80

# 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 // 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 = 80

# 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)

# add the game over group to the main group
main_group.append(game_over_group)

# create score label for each player
for i in range(2):
    # create a new label to hold score
    score_lbl = Label(terminalio.FONT, text="", color=colors[i], scale=1)
    if i == 0:
        # if it's player 1 put it in the top left
        score_lbl.anchor_point = (0, 0)
        score_lbl.anchored_position = (4, 1)
    else:
        # if it's player 2 put it tin the top right
        score_lbl.anchor_point = (1.0, 0)
        score_lbl.anchored_position = (display.width - 4, 1)

    # add the label to list of score labels
    score_lbls.append(score_lbl)

    # add the label to the main group
    main_group.append(score_lbl)

# initialize the text in the score labels to show 0
update_score_text()
```

```
# create a label to indicate which player's turn it is
current_player_lbl = Label(
    terminalio.FONT, text="Current Player", color=colors[current_turn_index],
scale=1
)

# place it centered horizontally at the top of the screen
current_player_lbl.anchor_point = (0.5, 0)
current_player_lbl.anchored_position = (display.width // 2, 1)

# add the score label to the main group
main_group.append(current_player_lbl)

# load the title screen bitmap
title_screen_bmp = OnDiskBitmap("memory_title.bmp")

# create a TileGrid for the title screen
title_screen_tg = TileGrid(
    bitmap=title_screen_bmp, pixel_shader=title_screen_bmp.pixel_shader
)

# add it to the main group
main_group.append(title_screen_tg)

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

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

# create a TileGrid for the mouse
mouse_tg = TileGrid(mouse_bmp, pixel_shader=mouse_bmp.pixel_shader)

# place it in the center of the display
mouse_tg.x = display.width // 2
mouse_tg.y = display.height // 2

# add the mouse to the main group
main_group.append(mouse_tg)

# variable for the mouse USB device instance
mouse = None

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

# scan for connected USB devices
for device in usb.core.find(find_all=True):
    # print information about the found devices
    print(f"{device.idVendor:04x}:{device.idProduct:04x}")
    print(device.manufacturer, device.product)
    print(device.serial_number)

    # assume this device is the mouse
    mouse = device

    # detach from kernel driver if active
    if mouse.is_kernel_driver_active(0):
        mouse.detach_kernel_driver(0)

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


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

# timestamp in the future to wait until before
```

```
# awarding points for a pair, or flipping cards
# back over and changing turns
WAIT_UNTIL = 0

# bool indicating whether the code is waiting to reset flipped
# cards and change turns or award points and remove
# cards. Will be True if the code is waiting to take action,
# False otherwise.
waiting_to_reset = False

# main loop
while True:
    # timestamp of the current time
    now = ticks_ms()

    # attempt mouse read
    try:
        # try to read data from the mouse, small timeout so the code will move on
        # quickly if there is no data
        data_len = mouse.read(0x81, buf, timeout=10)

        # if there was 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 - 1, mouse_tg.x + buf[1] // 2))
        mouse_tg.y = max(0, min(display.height - 1, mouse_tg.y + buf[2] // 2))

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

    # if the current state is title screen
    if CUR_STATE == STATE_TITLE:
        # if the left mouse button was clicked
        if buf[0] & (1 << 0) != 0:
            # change the current state to playing
            CUR_STATE = STATE_PLAYING
            # hide the title screen
            title_screen_tg.hidden = True
            # change the mouse cursor color to match the current player
            mouse_bmp.pixel_shader[2] = colors[current_turn_index]

    # if the current state is playing
    elif CUR_STATE == STATE_PLAYING:

        # if the code is  waiting to reset, and it's time to take action
        if waiting_to_reset and now >= WAIT_UNTIL:
            # this means that there are already 2 cards flipped face up.
            # The code needs to either award points, or flip the cards
            # back over and change to the next players turn.

            # change variable to indicate the code is no longer waiting to take
action
            waiting_to_reset = False

            # if both cards were the same i.e. they found a match
            if (
                card_tgs[cards_flipped_this_turn[0]][0]
                == card_tgs[cards_flipped_this_turn[1]][0]
            ):

                # set the cards tile index to show a blank spot instead of a card
                card_tgs[cards_flipped_this_turn[0]][0] = 8
                card_tgs[cards_flipped_this_turn[1]][0] = 8

                # award a point to the player
                player_scores[current_turn_index] += 1

                # refresh the score texts to show the new score
```

```
                update_score_text()

                # if the total of both players scores is equal to half the amount

# of cards then the code knows the game is over because each pair is worth 1
                # point
                if (
                    player_scores[0] + player_scores[1]
                    >= (grid_size[0] * grid_size[1]) // 2
                ):

                    # if the player's scores are equal
                    if player_scores[0] == player_scores[1]:
                        # set the game over message to tie game
                        game_over_label.text = "Game Over\nTie Game"

                    else:  # player scores are not equal

                        # if player 2 score is larger than player 1
                        if player_scores[0] < player_scores[1]:
                            # set the game over message to indicate player 2 victory
                            game_over_label.text = "Game Over\nPlayer 2 Wins"
                            game_over_label.color = colors[1]

                        else:  # player 1 score is larger than player 2
                            # set the game over message to indicate player 1 victory
                            game_over_label.text = "Game Over\nPlayer 1 Wins"
                            game_over_label.color = colors[0]

                    # set the game over group to visible
                    game_over_group.hidden = False

                    # change the state to gameover
                    CUR_STATE = STATE_GAMEOVER

            else:  # the two cards were different i.e. they did not find a match
                # set both cards tile index to the card back sprite to flip it back
over
                card_tgs[cards_flipped_this_turn[0]][0] = 10
                card_tgs[cards_flipped_this_turn[1]][0] = 10

                # go to the next players turn
                current_turn_index = (current_turn_index + 1) % 2

                # update the color of the current player indicator
                current_player_lbl.color = colors[current_turn_index]

                # update the color of the mouse cursor
                mouse_bmp.pixel_shader[2] = colors[current_turn_index]

            # empty out the cards flipped this turn list
            cards_flipped_this_turn = []

        # ignore any clicks while the code is waiting to take reset cards
        if now >= WAIT_UNTIL:
            # left btn pressed
            if buf[0] & (1 << 0) != 0:

                # loop over all cards
                for card_index, card in enumerate(card_tgs):
                    # coordinates of the mouse taking into account
                    # the offset from the card_grid position
                    coords = (mouse_tg.x - card_grid.x, mouse_tg.y - card_grid.y, 0)

                    # if this is a face down card, and the mouse coordinates
                    # are within its bounding box
                    if card[0] == 10 and card.contains(coords):
                        # flip the card face up by setting its tile index
                        # to the appropriate value from the card_locations list
```

```
                    card[0] = card_locations[card_tgs.index(card)]

                    # add this card index to the cards flipped this turn list
                    cards_flipped_this_turn.append(card_index)

                    # if 2 cards have been flipped this turn
                    if len(cards_flipped_this_turn) == 2:
                        # set the wait until time to a little bit in the future
                        WAIT_UNTIL = ticks_ms() + 1500
                        # set the waiting to reset flag to True
                        waiting_to_reset = True

    # if the current state is gameover
    elif CUR_STATE == STATE_GAMEOVER:
        # left btn pressed
        if buf[0] & (1 << 0) != 0:
            # get the coordinates of the mouse cursor point
            coords = (mouse_tg.x, mouse_tg.y, 0)

            # 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):
                # break to exit out of this script
                break
```

Code Explanation

The code for the Memory 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 two primary hardware peripherals: the HSTX connector with a DVI breakout for the display, and a basic USB mouse for the player control input.

HSTX Display

First the code will attempt to use the built-in display from `supervisor.runtime.display`. If that doesn't work, then it will initialize the display using the built-in core modules `picodvi`, and `framebufferio`. These modules support a few different resolutions and color depths. This project is made for the 320x240 resolution with 16 bit color depth. The pixels are automatically doubled before being pushed to the display, so it will come out as 640x480, depending on your monitor or TV, it may further upscale it to fit the screen.

If you haven't already, be sure to add `CIRCUITPY_DISPLAY_WIDTH=320` to your **settings.toml** file to ensure the correct configuration.

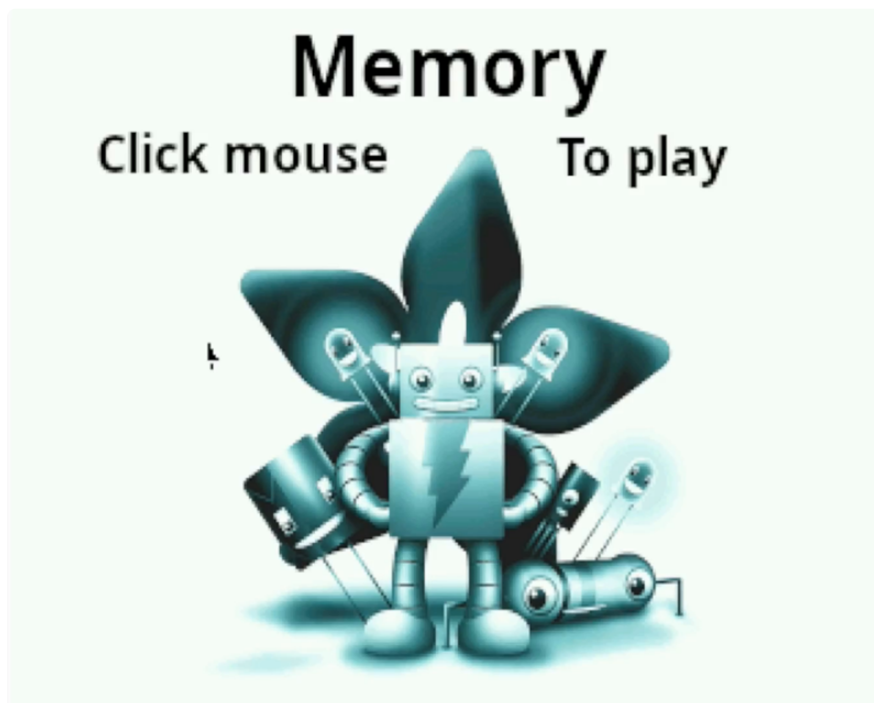
USB Mouse

The USB mouse setup, data reading, and cursor movement are all documented on the [Mouse Input guide page](https://adafru.it/1aga) (https://adafru.it/1aga). See that page for more details.

Helper Functions

The code contains two helper functions: `random_selection()`, and `update_score_text()`.

- `random_selection()` Accepts list and count arguments. It returns a list of items randomly selected from the given list with a length of `count`. This is used to select a number of cards at random out of the pool of all possible cards.
- `update_score_text()` Is a display helper function that will update the visual score text on the display of both players to reflect their current score.



State Machine

The code is based upon a [state machine](https://adafru.it/DtL) (https://adafru.it/DtL). There are 3 states that it can be in:

- Title state - shows the title screen and waits for a click of the mouse anywhere on the screen. When click occurs the state is changed to playing.

- Playing state - The primary gameplay occurs during this state. Players take turns flipping cards. Once there are no more cards to flip, the state is changed to gameover.
- Gameover state - The gameover message is shown along with buttons that can be clicked to play again or quit.

Player Specific Variables

There are a number of player specific variables stored in lists indexed with the `current_player_index` values as described on the [multi-player turns mechanic page](https://adafru.it/1agb) (https://adafru.it/1agb).

- `score_lbls` list will contain a Label instance for each player which will get added to `main_group` to be shown on the display. Each player's `score_lbl` will show their score during the gameplay.
- `colors` list will contain a hex color code for each player. By default these are pink `0xff00ff` for player index `0`, and green `0x00ff00` for player index `1`. Feel free to change the colors to suit your own style. The mouse cursor and text indicator at the top of the screen will be set to these colors to indicate which player's turn it is.
- `player_scores` list will contain the numerical score of each player.

Initializing Cards

First the code builds up `pool`, a list of indexes within the spritesheet, each index representing one of the possible cards. Initially the list is created with two copies of each possible card index. Two of each are desired so that there are guaranteed to be matches for every card used.

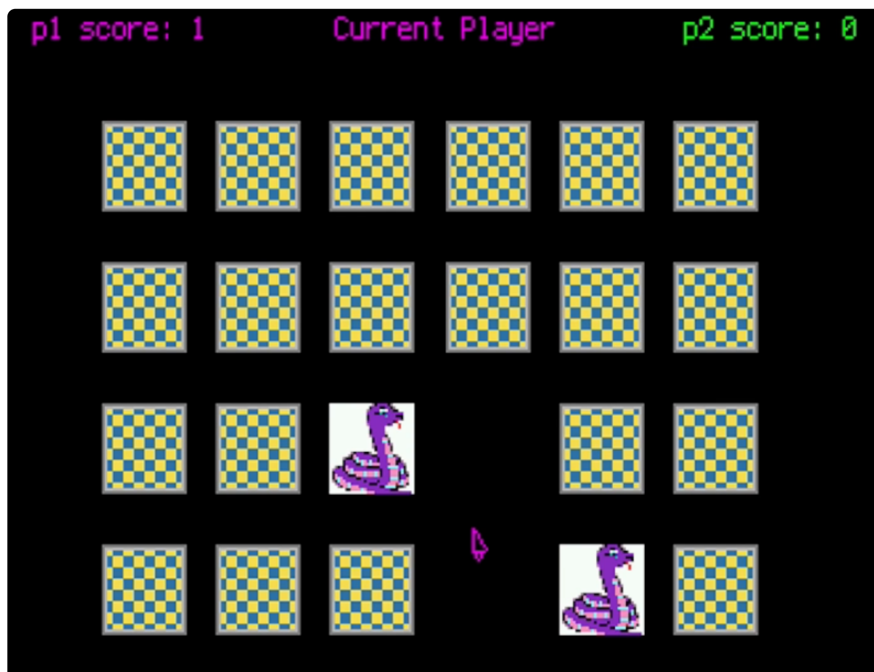
There are only 8 different cards, which makes 16 total cards once they are each paired up with a match. The grid is 6x4 which is 24 cards, so an additional 8 cards or 4 pairs are needed in order to fill out the rest of the grid. 4 card indexes are chosen at random using the `random_selection()` function, then those 4 plus a duplicate of each are added to the `pool` list to bring it up to a total of 24 card indexes.

The cards are arranged visually using a `GridLayout` which takes a `height` and `width` in pixels, as well as a `grid_size` in cells and automatically arranges the content added to it in a grid. A nested for loop of 4 rows by 6 columns is used to create a `TileGrid` for each card and add it to the grid at the appropriate cell. All `TileGrid`s are created with `default_tile=10` which is the face down card tile index within the spritesheet.

Within the nested loop the code also chooses and removes a random card index from the `pool` and adds it to the `card_locations` list. `card_locations` serves as the "key" to the location of the cards. It keeps track of which card is where on the board so that when a card is clicked we can change to the appropriate sprite to reveal which card it was.

Flipping Cards

A list named `cards_flipped_this_turn` stores the `TileGrid` indexes of the cards that have been flipped over this turn. When the first card is clicked it's index within the grid is added to this list, when the second card is clicked its index is added as well, then the code compares the sprite indexes of the `TileGrid`s at the grid indexes stored within `cards_flipped_this_turn`. If they match, a point is awarded to the current player and the cards are removed. If they don't match, there is a small delay before they are flipped back over by changing their `TileGrid` sprite index back to `10`, the face down card sprite. The `WAIT_UNTIL` variable is used to store the timestamp of time in the future when the cards will get flipped back down. The delay gives the players time to see the cards and try to commit their location to memory.



Check If the Game Is Over

To determine whether the game is over, the code will see if the sum of both players score is equal to the total size of the grid divided by two. For the default board of 6x4, this means the code is looking for the sum of scores to be `12`. When it is, the code knows that the game is over because twelve pairs have been found which accounts for all 24 cards we used.

The code checks if one score is higher than the other and sets the `gameover` message accordingly, including a message for tie game if the scores are the same.

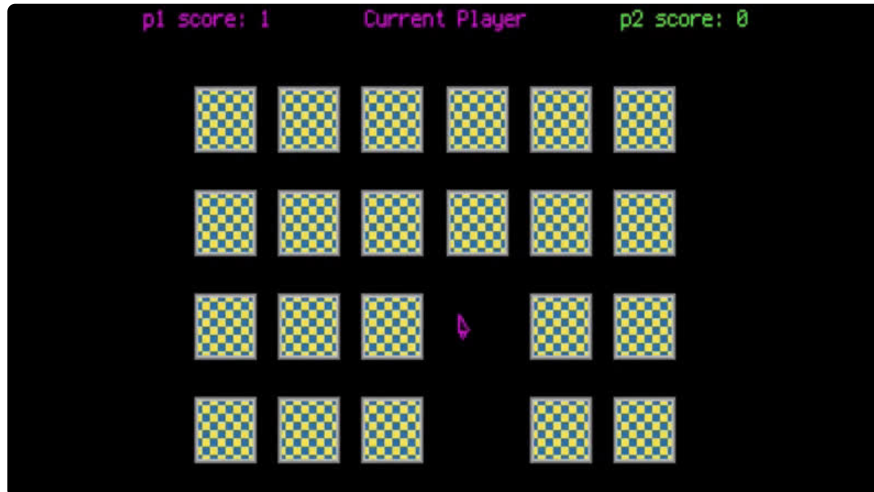
Usage

Ensure a USB mouse is plugged into the USB Host port 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



Players alternate using the mouse to take turns clicking on two cards to flip over. If the cards weren't a match, then they are flipped back face down and it becomes the next player's turn. If the two cards revealed are a match, then the player is awarded a point, the cards are removed from the board, and that player gets to take another turn, repeating until they fail to find a match.

The mouse color and an indicator label at the top of the screen are changed to indicate which player's turn it is currently.

Once the game has ended, there are buttons to play again or quit.