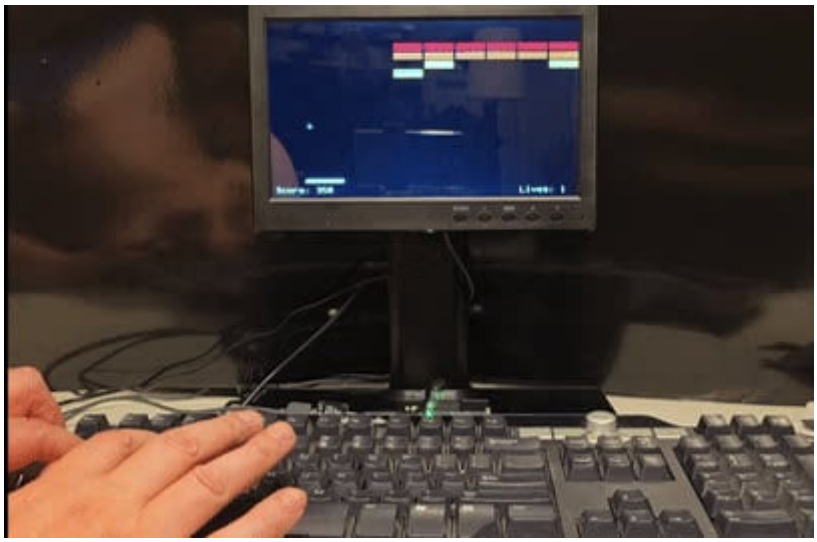
Breakout Game on the Metro RP2350

Created by Anne Barela

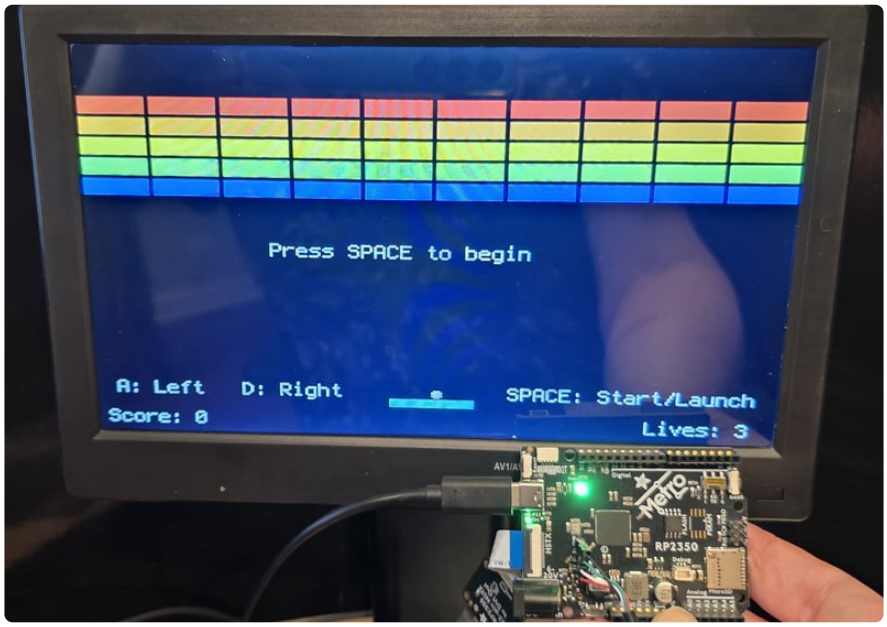


https://learn.adafruit.com/breakout-game-on-metro-rp2350

Last updated on 2025-04-09 11:26:11 AM EDT

Table of Contents

Overview

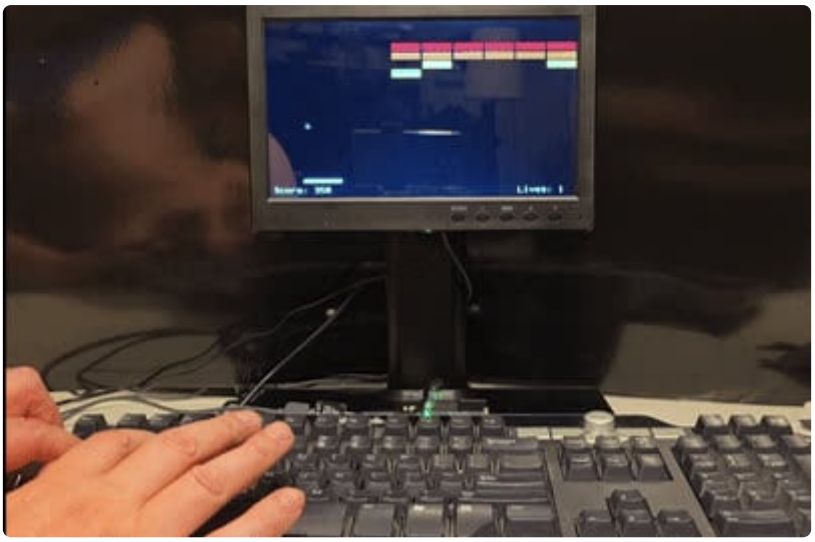


This guide presents the classic video game Breakout which uses an HDMI monitor for video out and a computer keyboard connected via USB for game control. A piezo speaker provides sound.

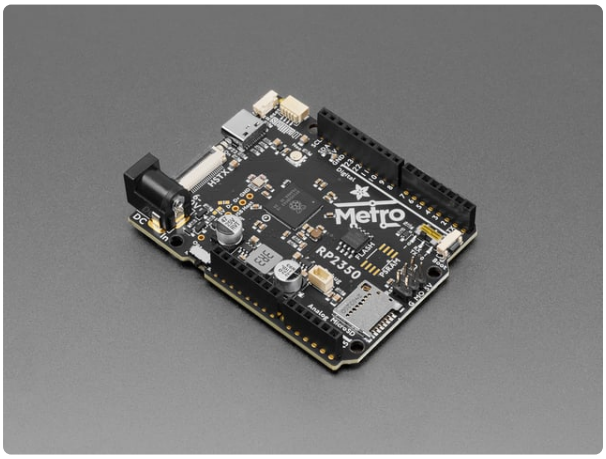
The Adafruit Metro RP2350 contains the RP2350B in a familiar "Arduino Uno" form factor with many peripherals included on the board. These include an HSTX connector that, when connected to a DVI breakout board, can output video compatible with modern HDMI monitors.

The board also has pins for a USB Host port to connect a keyboard or game controller.

We'll use these to provide input and output for gaming.



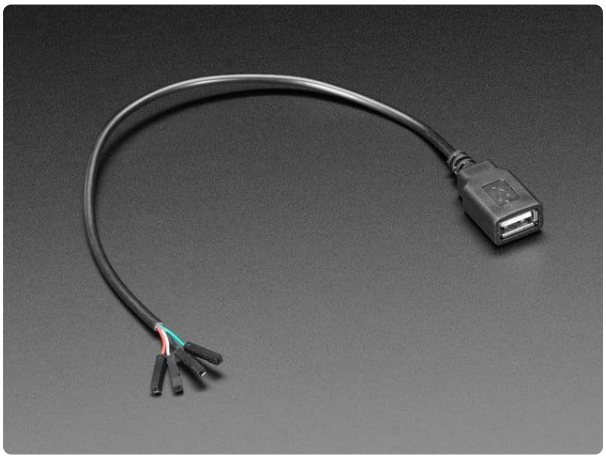
Parts



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...

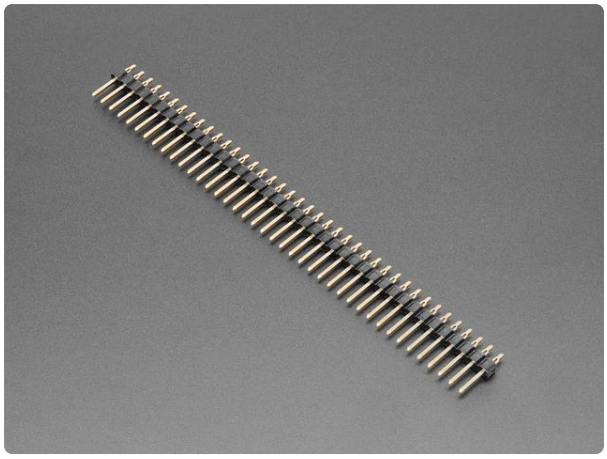
https://www.adafruit.com/product/6003



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...

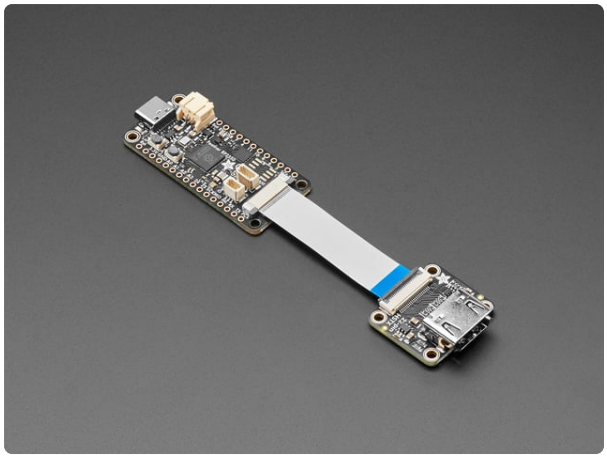
https://www.adafruit.com/product/4449



[Solderless Press-Fit Male Pin Header - 2.54mm / 0.1" Pitch]

If your soldering isn't quite up to scratch, or you yet don't own a soldering iron, then these nifty 0.1"/2.54mm standard pitch press-fit...

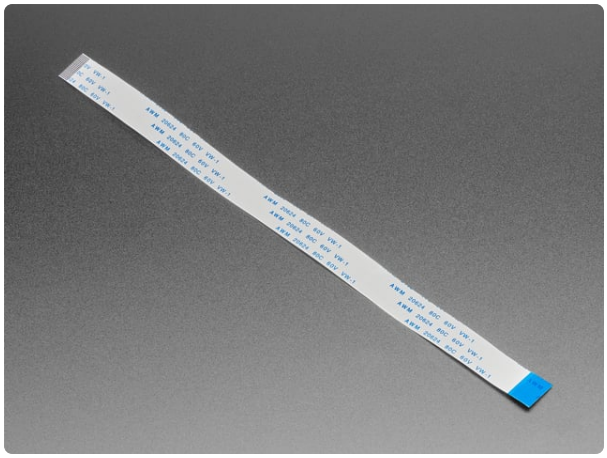
https://www.adafruit.com/product/5938



[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...

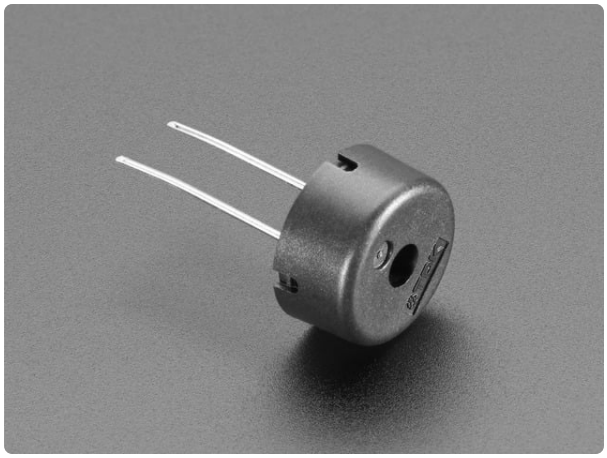
https://www.adafruit.com/product/6055



[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



[Piezo Buzzer]

Piezo buzzers are used for making beeps, tones and alerts. This one is petite but loud! Drive it with 3-30V peak-to-peak square wave. To use, connect one pin to ground (either one) and...

https://www.adafruit.com/product/160

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

- and -

You'll need a USB cable to hook to your PC for either solution. If your computer has a USB C port, you'll want the USB C to USB C cable.

1 x USB A to USB C Cable https://www.adafruit.com/product/4474
Approx 1 meter / 3 ft long

- or -

1 x USB C to USB C Cable https://www.adafruit.com/product/4199
USB 3.1 Gen 4 with E-Mark - 1 meter long

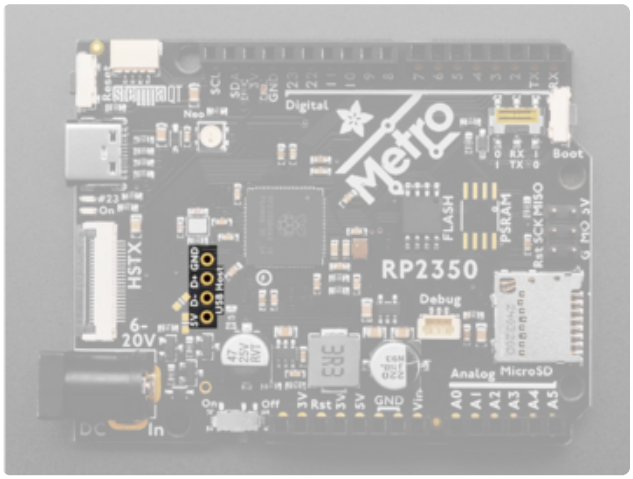
It's suggested you use an HDMI display you have around, such as a computer monitor or television. If you'd like to get a dedicated monitor:

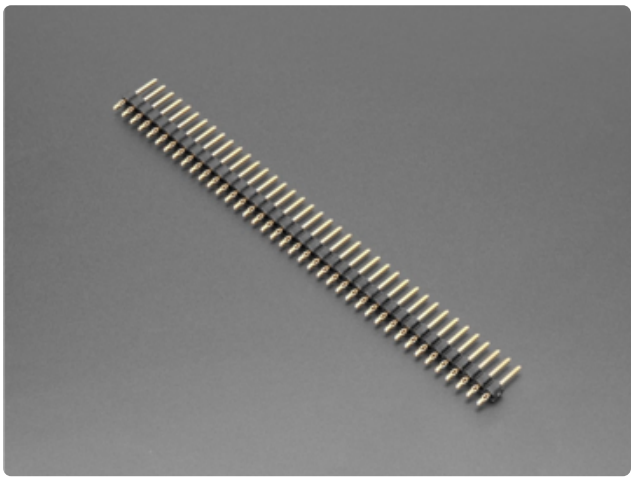
Raspberry Pi Monitor - Red and White Colorway
The Raspberry Pi Monitor is a 15.6" full HD computer display. It's the perfect compact, versatile, user-friendly desktop display companion for Raspberry Pi computers and other...
https://www.adafruit.com/product/6088

Preparing the Metro RP2350

The USB Host port is the only part of this project that required soldering and only if you use standard header pins. If you use the Solderless Press-Fit Male Pin

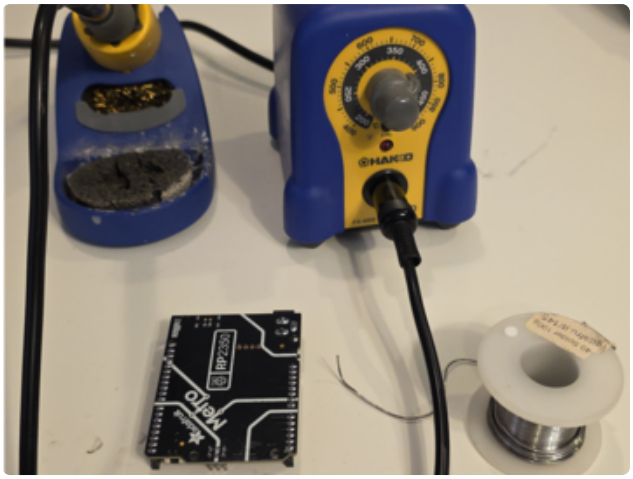
Header (http://adafru.it/5938) in the Parts list, soldering may not be required if they are installed correctly.

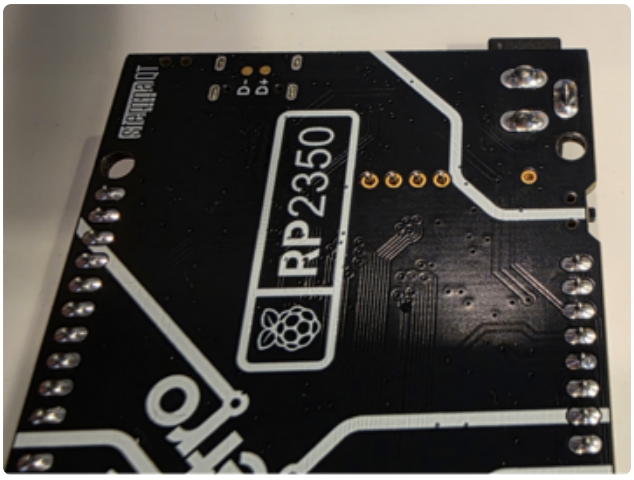


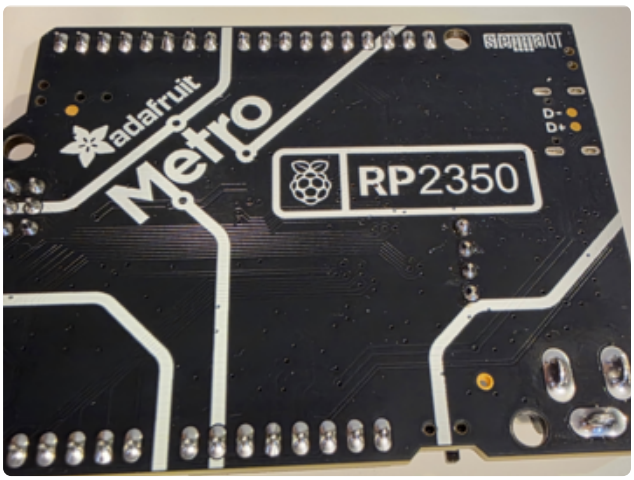


The USB Host pin connections are highlighted on the Metro image to the left. You will need a small piece of 0.1 inch male header, with 4 pins, to fit the holes.

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.







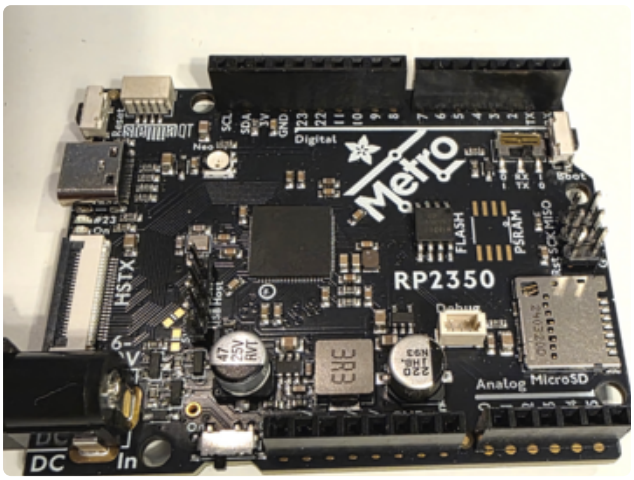
Put the short end of the header into the holes in the Metro marked USB Host.

If you are using solderless header then they are press fit into the holes. You might need some pressure to get them in. While they are designed to make electrical contact, you might want to solder them to be sure.

If using standard header, 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.

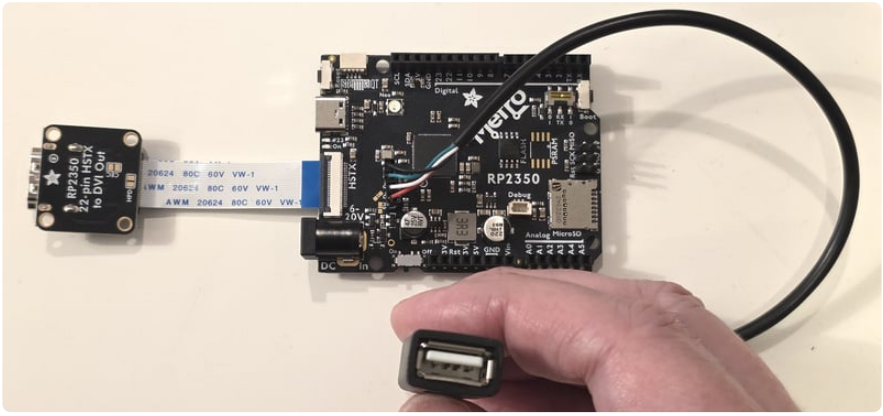
Get the USB Host cable and wire as follows:

GRD to **Black**

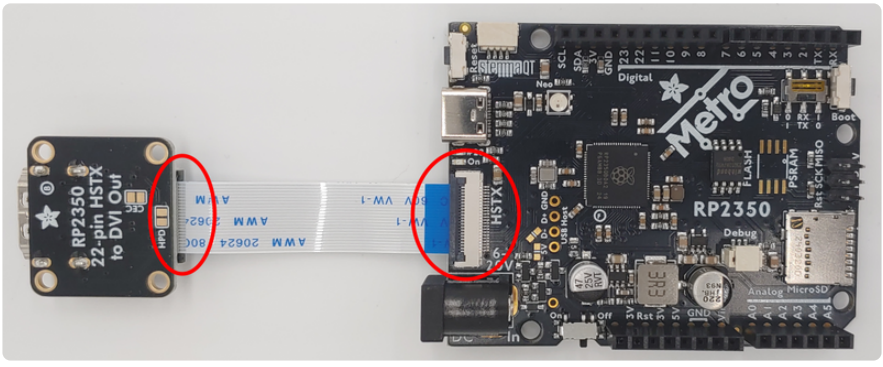
D+ to **Green**

D- to **White**

5V to **Red**



HSTX Connection to DVI

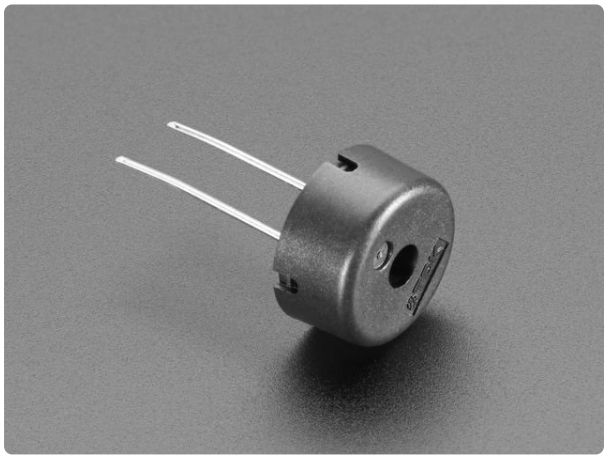


Get the HSTX cable. Any length Adafruit sells is fine. CAREFULLY lift the dark grey bar up on the Metro, insert the cable silver side down, blue side up, then put the bar CAREFULLY down, ensuring it locks. If it feels like it doesn't want to go, do not force it.

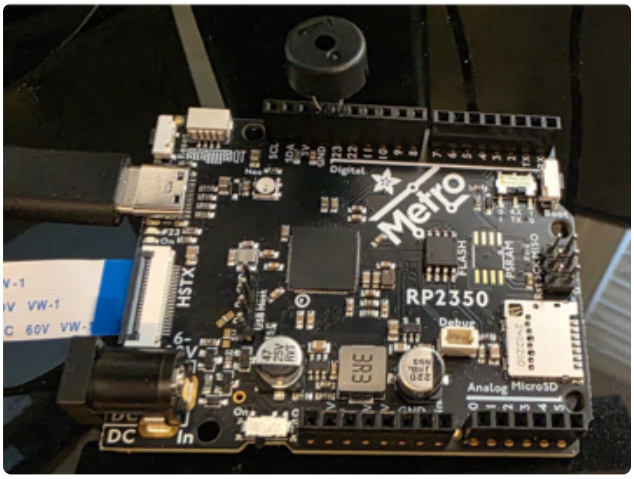
Do the same with the other end and the DVI breakout. Note that the DVI breakout will be inverted/upside down when compared to the Metro - this is normal for these boards and the Adafruit cables.

Piezo Speaker (optional)

Games always are better with some sound. This project uses a small Piezo speaker pressed to the headers.



Piezo Buzzer
Piezo buzzers are used for making beeps, tones and alerts. This one is petite but loud! Drive it with 3-30V peak-to-peak square wave. To use, connect one pin to ground (either one) and...
https://www.adafruit.com/product/160



Connect the Piezo leads into pins 23 and GND, which are adjacent, on the female header pins. No soldering is needed.

Install CircuitPython

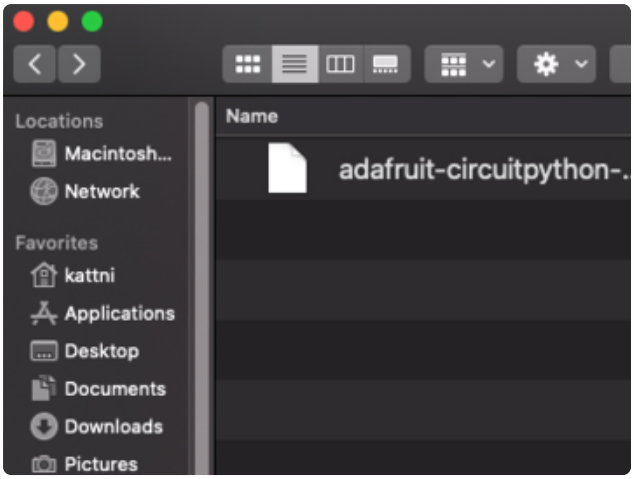
CircuitPython (https://adafru.it/tB7) is a derivative of MicroPython (https://adafru.it/BeZ) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files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

CircuitPython Quickstart

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

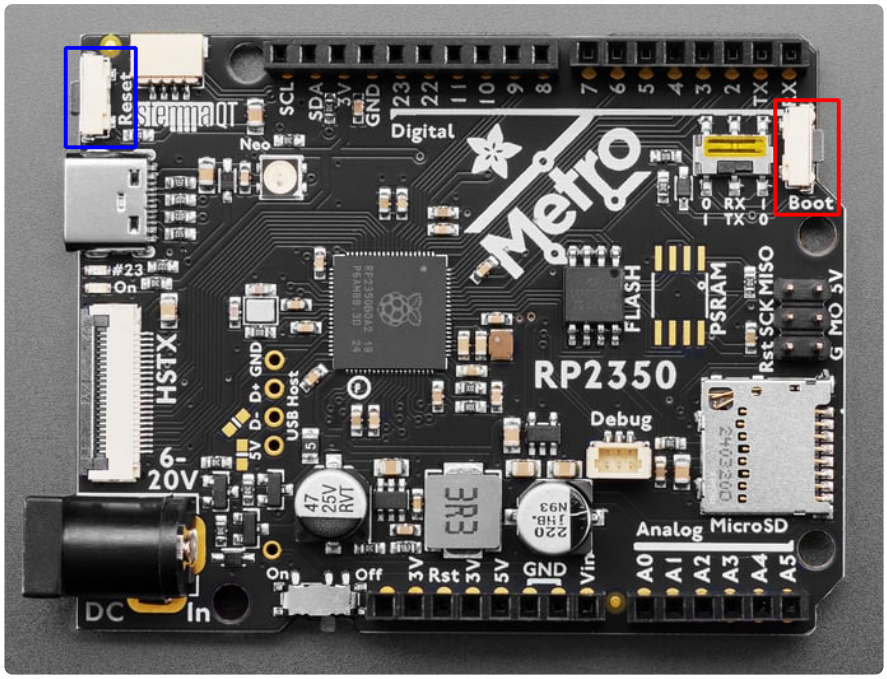
Download the latest version of CircuitPython for this board via circuitpython.org

https://adafru.it/1aeL



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

Save it wherever is convenient for you.

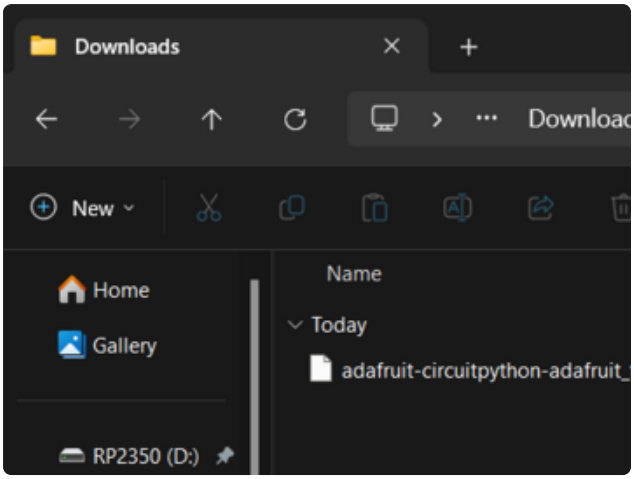


To enter the bootloader, hold down the **BOOT/BOOTSEL button** (highlighted in red above), and while continuing to hold it (don't let go!), press and release the **reset button** (highlighted in red or blue above). **Continue to hold the BOOT/BOOTSEL button until the RP2350 drive appears!**

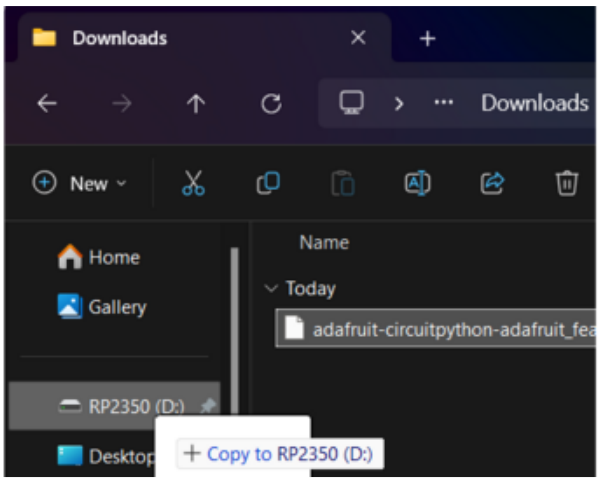
If the drive does not appear, release all the buttons, and then repeat the process above.

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.

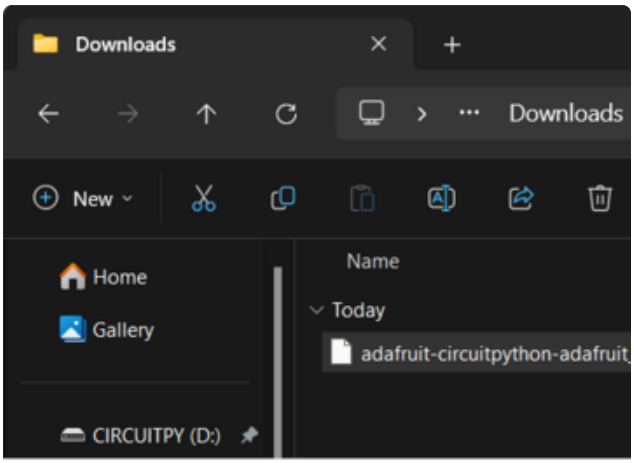
A lot of people end up using charge-only USB cables and it is very frustrating! **Make sure you have a USB cable you know is good for data sync.**



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

Code

CircuitPython Usage

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

Thankfully, we can do this 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.


```
# SPDX-FileCopyrightText: 2025 Anne Barela for Adafruit Industries
#
# SPDX-License-Identifier: MIT
#
# Breakout
# Original https://github.com/davepl/Grok3-Breakout by Grok3
# Converted to CircuitPython by GitHub Copilot 2/20/2025
# Further improved with keyboard controls via serial input, sound
# with Claude Sonnet 3.7

import sys
import time
import random
import board
import displayio
import supervisor
from adafruit_display_shapes.rect import Rect
from adafruit_display_shapes.circle import Circle
from adafruit_display_text import label
import terminalio
import picodvi
```

```
import framebufferio
import neopixel
import pwmio

displayio.release_displays()  # Ensure there are no active displays

# Initialize status LED
leds = neopixel.NeoPixel(board.NEOPIXEL, 1, brightness=0.4)
leds.fill((0, 40, 0))  # Initial status: green

# Initialize piezo speaker, connect piezo D23 to ground
try:
    piezo = pwmio.PWMOut(board.D23, frequency=440, duty_cycle=0,
                         variable_frequency=True)
except Exception as e:  # pylint: disable=broad-except
    print("Piezo initialization error:", e)
    piezo = None

# Sound effect functions
def play_tone(frequency, duration):
    """Play a tone at the specified frequency for the specified duration"""
    if piezo is None:
        return

    piezo.frequency = frequency
    piezo.duty_cycle = 32767  # 50% duty cycle
    time.sleep(duration)
    piezo.duty_cycle = 0

def play_paddle_hit():
    """Sound for ball hitting paddle"""
    if piezo is None:
        return

    play_tone(440, 0.05)  # A4, short duration

def play_wall_hit():
    """Sound for ball hitting wall"""
    if piezo is None:
        return

    play_tone(330, 0.03)  # E4, very short

def play_brick_hit():
    """Sound for ball hitting brick"""
    if piezo is None:
        return

    play_tone(660, 0.05)  # E5, short

def play_game_over():
    """Sound for game over"""
    if piezo is None:
        return

    # Descending tone
    for i in range(5):
        play_tone(330 - i*30, 0.1)

def play_level_win():
    """Sound for winning the level"""
    if piezo is None:
        return

    # Victory fanfare
    sequence = [440, 440, 440, 587, 440, 587, 659]
    durations = [0.1, 0.1, 0.1, 0.3, 0.1, 0.1, 0.4]

    for freq, dur in zip(sequence, durations):
```

```
        play_tone(freq, dur)
        time.sleep(0.01)  # Small gap between notes

def play_life_lost():
    """Sound for losing a life"""
    if piezo is None:
        return

    play_tone(220, 0.1)
    time.sleep(0.05)
    play_tone(196, 0.2)

def play_game_start():
    """Sound for game start"""
    if piezo is None:
        return

    play_tone(440, 0.1)
    time.sleep(0.05)
    play_tone(554, 0.1)
    time.sleep(0.05)
    play_tone(659, 0.2)

# Initialize display
try:
    fb = picodvi.Framebuffer(
        320, 240,
        clk_dp=board.CKP, clk_dn=board.CKN,
        red_dp=board.D0P, red_dn=board.D0N,
        green_dp=board.D1P, green_dn=board.D1N,
        blue_dp=board.D2P, blue_dn=board.D2N,
        color_depth=8
    )
except ValueError as e:
    print("Error: ", e)
    # Per picodvi/Framebuffer_RP2350.c only 320x240 at 8 & 16 bits work
    leds.fill((255, 0, 0))  # Error status: red
    sys.exit(e)
display = framebufferio.FramebufferDisplay(fb)

# Game constants
PADDLE_WIDTH = 40
PADDLE_HEIGHT = 5
BALL_RADIUS = 2
BRICK_ROWS = 5
BRICK_COLS = 10
BRICK_WIDTH = 30
BRICK_HEIGHT = 10
BRICK_GAP = 2
PADDLE_SPEED = 6.0         # Change to make paddle speed up or slow down
BALL_SPEED_INITIAL = 2.75  # Change initial ball speed
BRICK_COLORS = [0xFF4136, 0xFF851B, 0xFFDC00, 0x2ECC40, 0x0074D9]
DISPLAY_LINE = 20  # Score is on the bottom of the display 20 px tall

# Game state
game_active = False
game_over = False
brick_count = BRICK_ROWS * BRICK_COLS

# Keyboard input state
key_left_pressed = False
key_right_pressed = False
key_space_pressed = False
space_key_released = True  # Track space key release to prevent multiple actions

# Track paddle position with float for smoother movement
paddle_pos_x = 0.0

def check_keys():
```

```
    """
    Check for keyboard input via supervisor.runtime.serial_bytes_available
    Returns tuple of (left_pressed, right_pressed, space_pressed, any_input)
    """
    l_pressed = False
    r_pressed = False
    s_pressed = False
    any_key = False

    # Check if serial data is available
    if supervisor.runtime.serial_bytes_available:
        any_key = True
        try:
            key = sys.stdin.read(1)
            if key in ('a', 'A'):    # Left movement
                l_pressed = True
            elif key in ('d', 'D'):  # Right movement
                r_pressed = True
            elif key == ' ':         # Space for start/launch
                s_pressed = True
        except Exception as e:       # pylint: disable=broad-except
            print("Input error:", e)

    return (l_pressed, r_pressed, s_pressed, any_key)

# pylint: disable=redefined-outer-name
def create_game_elements():
    """Create and return all game display elements"""
    game_group = displayio.Group()

    # Paddle
    paddle = Rect(
        (display.width - PADDLE_WIDTH) // 2,
        display.height - PADDLE_HEIGHT - DISPLAY_LINE,
        PADDLE_WIDTH,
        PADDLE_HEIGHT,
        fill=0x00FFFF
    )
    game_group.append(paddle)

    # Ball
    ball = Circle(
        display.width // 2,
        display.height - PADDLE_HEIGHT - DISPLAY_LINE - BALL_RADIUS * 3,
        BALL_RADIUS,
        fill=0xFFFFFF
    )
    game_group.append(ball)

    # Bricks
    bricks = []
    for row in range(BRICK_ROWS):
        for col in range(BRICK_COLS):
            # Calculate brick position with appropriate spacing
            brick_x = col * (BRICK_WIDTH + BRICK_GAP) + \
                      (display.width - (BRICK_COLS * (BRICK_WIDTH + BRICK_GAP)
                       - BRICK_GAP)) // 2
            brick_y = row * (BRICK_HEIGHT + BRICK_GAP) + 30
# Start a bit down from top

            brick = Rect(
                brick_x,
                brick_y,
                BRICK_WIDTH,
                BRICK_HEIGHT,
                fill=BRICK_COLORS[row % len(BRICK_COLORS)]
            )
            bricks.append(brick)
            game_group.append(brick)
```

```
    # Score and lives
    score_label = label.Label(
        terminalio.FONT,
        text="Score: 0",
        color=0xFFFFFF,
        x=5,
        y=display.height - 10
    )
    lives_label = label.Label(
        terminalio.FONT,
        text="Lives: 3",
        color=0xFFFFFF,
        x=display.width - 60,
        y=display.height - 10
    )
    game_group.append(score_label)
    game_group.append(lives_label)

    # Controls info
    controls_label = label.Label(
        terminalio.FONT,
        text="A: Left   D: Right",
        color=0xFFFFFF,
        x=10,
        y=display.height - 30
    )
    game_group.append(controls_label)

    # Start/game over message
    message_label = label.Label(
        terminalio.FONT,
        text="Press SPACE to begin",
        color=0xFFFFFF,
        x=(display.width - 130) // 2,
        y=display.height // 2
    )
    game_group.append(message_label)

    return (game_group, paddle, ball, bricks, score_label,
            lives_label, message_label, controls_label)

# Create initial game elements
main_group, paddle, ball, bricks, score_label, lives_label, message_label, \
    controls_label = create_game_elements()
display.root_group = main_group

# Game variables
score = 0
lives = 3
ball_dx = 0.0
ball_dy = 0.0
ball_speed = BALL_SPEED_INITIAL
last_hit_brick = None

# Track actual ball position with floats
ball_pos_x = float(ball.x)
ball_pos_y = float(ball.y)

# Initialize paddle position tracking
paddle_pos_x = float((display.width - PADDLE_WIDTH) // 2)

def reset_ball():
    """Reset the ball position and set it initially stationary"""
    global ball_dx, ball_dy, ball_pos_x, ball_pos_y  # pylint: disable=global-
statement
    ball_pos_x = float(display.width // 2)
    ball_pos_y = float(display.height - PADDLE_HEIGHT - DISPLAY_LINE - BALL_RADIUS *
3)
```

```
    ball.x = int(ball_pos_x)
    ball.y = int(ball_pos_y)
    ball_dx = 0
    ball_dy = 0

def launch_ball():
    """Launch the ball in a random upward direction"""
    global ball_dx, ball_dy, ball_speed  # pylint: disable=global-statement
    ball_speed = BALL_SPEED_INITIAL
    angle = random.uniform(0.5, 0.8)  # Launch at angle between 45-65 degrees
    ball_dx = ball_speed * random.choice([-angle, angle])
    # Ensure consistent speed
    ball_dy = -ball_speed * (1 - abs(ball_dx/ball_speed))
    controls_label.hidden = True

def update_message(text):
    """Update the message text"""
    message_label.text = text
    # Center text (approximate width)
    message_label.x = (display.width - len(text) * 6) // 2

# Game loop
reset_ball()

# Track last collision for sound effect throttling
last_wall_hit_time = 0
sound_cooldown = 0.05  # Minimum time between similar sound effects

# Play startup sound
play_game_start()

while True:
    current_time = time.monotonic()

    # Check keyboard input
    left_pressed, right_pressed, space_pressed, any_input = check_keys()

    # Apply paddle movement ONLY if keys are currently pressed
    if left_pressed and paddle_pos_x > 0:
        paddle_pos_x -= PADDLE_SPEED
        # Ensure paddle doesn't go offscreen
        if paddle_pos_x < 0:
            paddle_pos_x = 0

    if right_pressed and paddle_pos_x < display.width - PADDLE_WIDTH:
        paddle_pos_x += PADDLE_SPEED
        # Ensure paddle doesn't go offscreen
        if paddle_pos_x > display.width - PADDLE_WIDTH:
            paddle_pos_x = display.width - PADDLE_WIDTH

    # Update paddle position with integer conversion
    paddle.x = int(paddle_pos_x)

    # Handle space bar for start/launch with debouncing
    if space_pressed and space_key_released:
        space_key_released = False  # Prevent multiple triggers until released

        if game_over:
            # Reset everything for a new game
            (main_group, paddle, ball, bricks, score_label,
             lives_label, message_label, controls_label) = create_game_elements()
            display.root_group = main_group
            score = 0
            lives = 3
            brick_count = BRICK_ROWS * BRICK_COLS
            game_over = False
            reset_ball()
            update_message("")
            # Reset paddle position tracker
```

```
            paddle_pos_x = float((display.width - PADDLE_WIDTH) // 2)

            # Play restart sound
            play_game_start()

        if not game_active and not game_over:
            game_active = True
            launch_ball()
            update_message("")

            # Play ball launch sound
            play_tone(587, 0.1)  # D5, short

    # Reset space key released state if no key is pressed
    if not space_pressed:
        space_key_released = True

    # Handle game logic when active
    if game_active and not game_over:
        # Update ball position (using float variables first)
        ball_pos_x += ball_dx
        ball_pos_y += ball_dy
        # Update the actual ball object with integer positions
        ball.x = int(ball_pos_x)
        ball.y = int(ball_pos_y)

        # Ball collision with walls with sound
        wall_hit = False

        if ball_pos_x - BALL_RADIUS <= 0:
            ball_pos_x = BALL_RADIUS + 1  # Prevent sticking to wall
            ball_dx = abs(ball_dx)  # Move right
            wall_hit = True
        elif ball_pos_x + BALL_RADIUS >= display.width:
            ball_pos_x = display.width - BALL_RADIUS - 1  # Prevent sticking to wall
            ball_dx = -abs(ball_dx)  # Move left
            wall_hit = True

        if ball_pos_y - BALL_RADIUS <= 0:
            ball_pos_y = BALL_RADIUS + 1  # Prevent sticking to wall
            ball_dy = abs(ball_dy)  # Move down
            wall_hit = True

        # Play wall hit sound with cooldown to prevent sound overlap
        if wall_hit and current_time - last_wall_hit_time > sound_cooldown:
            play_wall_hit()
            last_wall_hit_time = current_time

        # Ball fell below paddle
        if ball_pos_y + BALL_RADIUS > display.height - DISPLAY_LINE:
            lives -= 1
            lives_label.text = f"Lives: {lives}"

            # Play life lost sound
            play_life_lost()

            if lives <= 0:
                game_active = False
                game_over = True
                update_message("GAME OVER - Press SPACE to play again")

                # Play game over sound
                play_game_over()
            else:
                reset_ball()
                game_active = False
                update_message("Press SPACE to continue")

        # Ball collision with paddle
```

```
        paddle_collision = (
            ball_pos_y + BALL_RADIUS >= paddle.y and
            ball_pos_y - BALL_RADIUS <= paddle.y + PADDLE_HEIGHT and
            ball_pos_x + BALL_RADIUS >= paddle.x and
            ball_pos_x - BALL_RADIUS <= paddle.x + PADDLE_WIDTH
        )

        if paddle_collision:
            # Determine bounce angle based on where ball hit paddle
            # Center of paddle gives straight bounce, edges give angled bounce
            hit_position = (ball_pos_x - paddle.x) / PADDLE_WIDTH
            angle_factor = 2 * (hit_position - 0.5)  # -1 to 1 based on position

            # Set new ball direction with slight speed increase
            ball_speed = min(ball_speed * 1.05, 4.5)  # Speed up slightly, max at
4.5
            ball_dx = ball_speed * angle_factor
            # Ensure upward movement
            ball_dy = -ball_speed * (1 - 0.8 * abs(angle_factor))

            # Ensure ball is above paddle (prevent sticking)
            ball_pos_y = paddle.y - BALL_RADIUS - 1
            ball.y = int(ball_pos_y)

            # Play paddle hit sound
            play_paddle_hit()

        # Ball collision with bricks
        for brick in bricks:
            brick_collision = (
                brick in main_group and
                brick.x <= ball_pos_x + BALL_RADIUS and
                brick.x + BRICK_WIDTH >= ball_pos_x - BALL_RADIUS and
                brick.y <= ball_pos_y + BALL_RADIUS and
                brick.y + BRICK_HEIGHT >= ball_pos_y - BALL_RADIUS
            )

            if brick_collision:
                # Skip if this is the same brick we just hit (prevent double hits)
                if brick == last_hit_brick:
                    continue

                last_hit_brick = brick
                main_group.remove(brick)
                brick_count -= 1

                # Determine which side of the brick was hit
                dx1 = ball_pos_x - brick.x  # Distance from left edge
                dx2 = brick.x + BRICK_WIDTH - ball_pos_x  # Distance from right edge
                dy1 = ball_pos_y - brick.y  # Distance from top edge
                dy2 = brick.y + BRICK_HEIGHT - ball_pos_y  # Distance from bottom
edge

                # Find the minimum distance
                min_dist = min(dx1, dx2, dy1, dy2)

                # Bounce based on which side was hit
                if min_dist in (dy1, dy2):  # Top or bottom hit
                    ball_dy = -ball_dy
                else:  # Left or right hit
                    ball_dx = -ball_dx

                # Update score
                score += 10
                score_label.text = f"Score: {score}"

                # Play brick hit sound
                play_brick_hit()
```

```
                # Check win condition
                if brick_count <= 0:
                    game_active = False
                    game_over = True
                    update_message("YOU WIN! Press SPACE to play again")

                    # Play victory sound
                    play_level_win()

                # Only process one brick collision per frame
                break

        # Reset last_hit_brick if we're not touching it anymore
        if last_hit_brick:
            still_touching = (
                last_hit_brick.x <= ball_pos_x + BALL_RADIUS and
                last_hit_brick.x + BRICK_WIDTH >= ball_pos_x - BALL_RADIUS and
                last_hit_brick.y <= ball_pos_y + BALL_RADIUS and
                last_hit_brick.y + BRICK_HEIGHT >= ball_pos_y - BALL_RADIUS
            )
            if not still_touching:
                last_hit_brick = None

    elif not game_active and not game_over:
        # Ball sticks to paddle when not active
        ball.x = paddle.x + PADDLE_WIDTH // 2
        ball_pos_x = float(ball.x)

    # Refresh display
    display.refresh()
    time.sleep(0.016)  # ~60fps
```

Code Principles

USB Host Input

USB Host is relatively new to CircuitPython, first coming on Raspberry Pi RP2040-based boards.

Typically, one would access a USB port by:

- Establishing the USB connection
- Reading USB Reports, sections of bytes sent when an action occurs on the peripheral like a key is pressed or joystick moved.
- Parsing the reports and providing meaningful input to the program.

Python has the concept of standard input and output streams, similar to those in Linux/Unix and other operating systems. CircuitPython has this capability and through a lot of behind the scenes code, presents a USB keyboard as a stdin input device. The code to get USB Host Keyboard characters and echo them to serial out is as follows:

```
import supervisor
import sys
```

```
while True:
    available = supervisor.runtime.serial_bytes_available
    if available:
        c = sys.stdin.read(available)
        print(c, end='')
```

While not every key on the keyboard has a single character representation, for simple gameplay this works very well! Using the gaming standard WASD keys for movement and the spacebar for Start is easily done and what is used in this project.

Video Output

Video for this project is output via the Raspberry Pi HSTX peripheral. CircuitPython has the `picodvi` library which can set up the DVI graphics as a standard CircuitPython framebuffer usable by `displayio`. There are only certain display resolutions and color bit depths supported at present with lower resolutions "pixel doubled" to fit higher resolution HDMI displays. All that happens behind the scenes.

Usage

Ensure a USB keyboard 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 keyboard after power up.

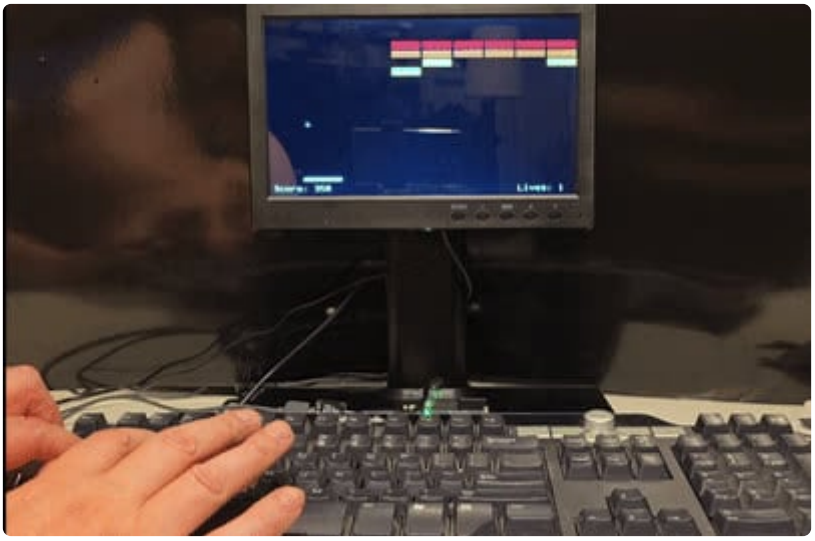
Be sure you connect the DVI breakout to an HDMI monitor and the monitor is on. You might need a long cable if your monitor is not near the Metro RP2350 (like a television). The cables are standard and may be obtained from any trusted retail outlet. Also reset the Metro if you plug in HDMI after powering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

The game should auto start if all the software was loaded in the previous step. The colored blocks at the top, the paddle below.

Gameplay

On-screen keys used in the game are displayed. The typical WASD directional keys are used (caps lock or not, no matter) and since the paddle only goes left and right, only A and D are used. The spacebar is used to start the game and when a turn is lost.



The goal is to keep hitting the bouncing ball, aiming to "break" (eliminate) all the colored blocks.

While the ball is falling, position the paddle so the ball will hit it and bounce up again.

If the ball hits the end of the paddle it may bounce in a shallow angle (glancing blow). If the paddle is moving when the ball hits it, the ball may not bounce at an equal and opposite angle but have "some english" from the paddle momentum.

Sound effects are played with each action on screen (if you have the piezo installed).

For each colored brick removed, you score 10 points. You win by eliminating all the bricks with the lives you have, scoring 500.

Changing the Code

There are values in the code to adjust various parameters in the game if you want to simplify things or make things more challenging:

`lives` is the number of chances you have to get all the bricks cleared and is set by default at `3`.

`BALL_SPEED_INITIAL` is the speed the ball moves at the start. Increase or decrease for faster or slower play.

`PADDLE_WIDTH` is the width of the paddle in pixels. Making the paddle smaller makes things more challenging.

`PADDLE_SPEED` affects the speed at which the paddle can move when the A and D keys are pressed.

`BALL_RADIUS` changes the ball size.