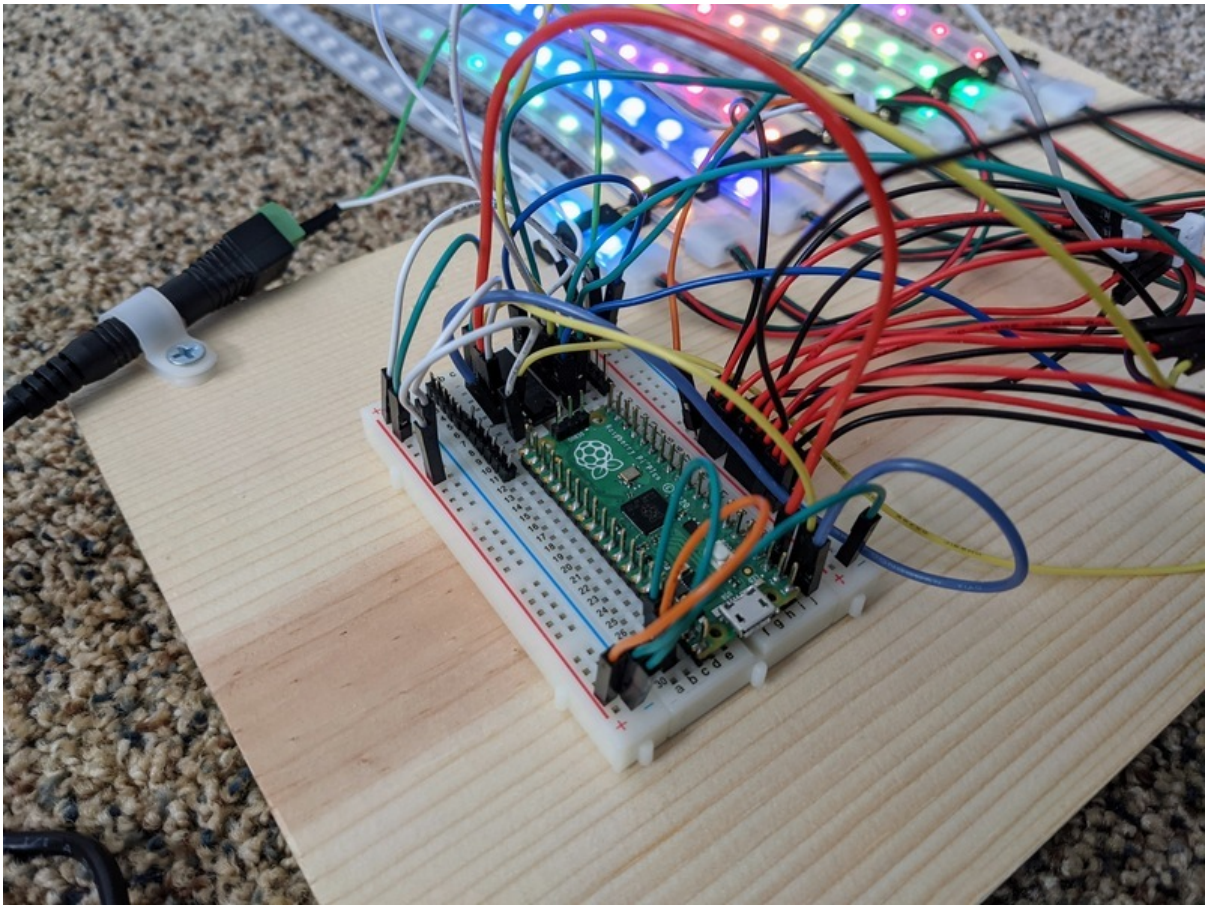
# NeoPIO: Drive lots of LEDs with Raspberry Pi Pico

Created by Jeff Epler

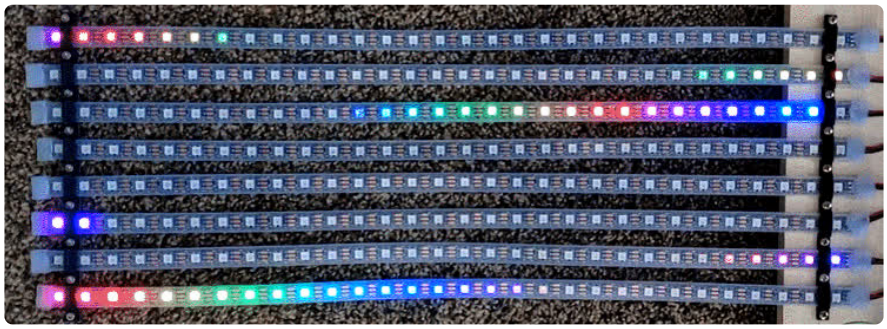


https://learn.adafruit.com/neopio-drive-lots-of-leds-with-raspberry-pi-pico

Last updated on 2025-03-04 10:16:41 AM EST

# Table of Contents

# Overview



The new RP2040 microcontroller from Raspberry Pi Foundation has a powerful feature: the PIO module, a sort of accelerated I/O processor.

In this guide, you will learn how you can use CircuitPython and PIO together to drive 8 separate NeoPixel LED strands using just 3 GPIO pins and a 74HC595 Shift Register (https://adafru.it/QEa).

One advantage of driving your NeoPixel LEDs this way is that less time is spent actually sending the pixel data to the LEDs—It is about 7.5x faster to `show()` your NeoPixels with the parallel method. Depending on your project, having parallel strips as opposed to chained strips can also make for better wiring.

Mostly, this guide is just a cool way to show off some of the capabilities of PIO itself. When it comes to wiring up NeoPixels, especially lots of them, check out some in-depth information in the UberGuide (https://adafru.it/Bej) including learning about level-shifting (https://adafru.it/19E0) and powering them (https://adafru.it/DCq).

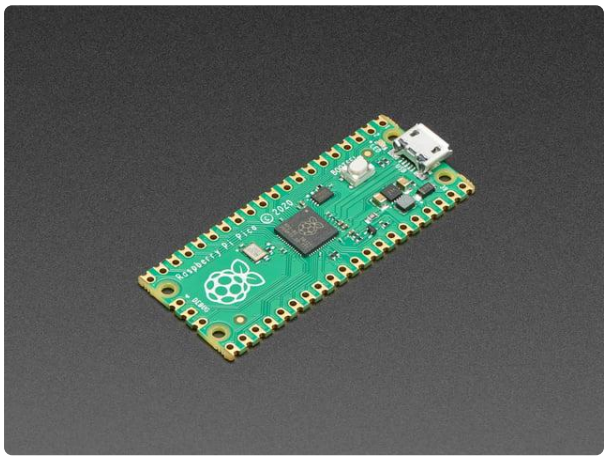
## Benchmark

The numbers below compare the speed of the two methods of refreshing NeoPixels, the traditional with a single 960-pixel chain, or 8 independent 120-pixel strands thanks to using PIO. It excludes the time to actually update animations on the strips, so it is a "best case" scenario.

```
960 pixels, 100 refreshes
Neopixel show(): 2.93s,  34.1 FPS,  32757 pixels/second
NeoPIO   show(): 0.43s, 232.7 FPS, 223418 pixels/second
```

Check out the following pages to learn more about PIO and how to use it in CircuitPython.

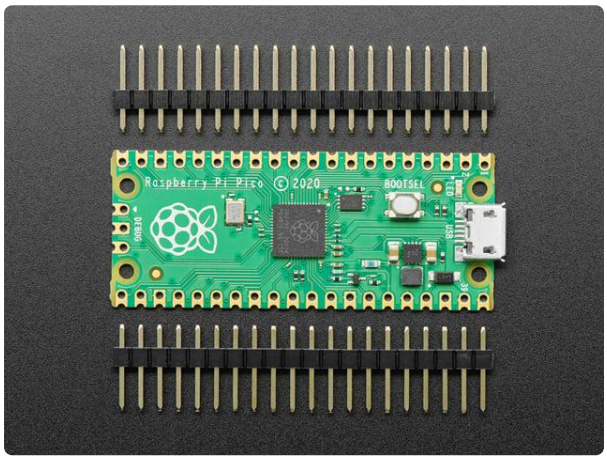
## Parts



Raspberry Pi Pico RP2040

The Raspberry Pi foundation changed single-board computing when they released the Raspberry Pi computer, now they're ready to...

https://www.adafruit.com/product/4864



Raspberry Pi Pico RP2040 with Loose Unsoldered Headers

The Raspberry Pi foundation changed single-board computing when they released the Raspberry Pi computer, now they're...

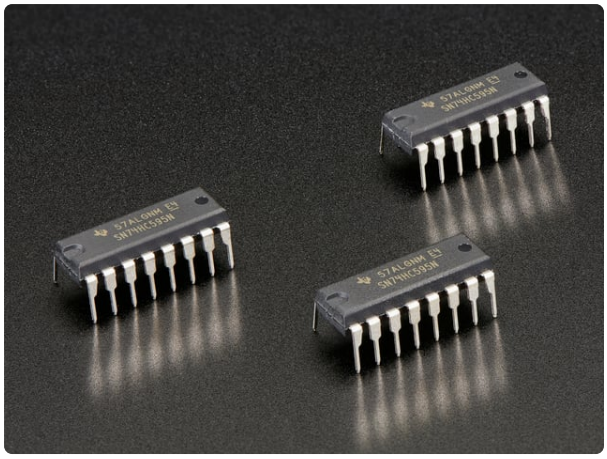
https://www.adafruit.com/product/4883

Adafruit NeoPixel LED Strip with 3-pin JST PH 2mm Connector

Plug in and glow, this Adafruit NeoPixel LED Strip with JST PH Connector has 30 total LEDs in a "60 LED per meter" spacing,...

https://www.adafruit.com/product/3919

You can make this project with anywhere from 2 to 8 NeoPixel LED strands. Each strand should be the same length.



74HC595 Shift Register - 3 pack

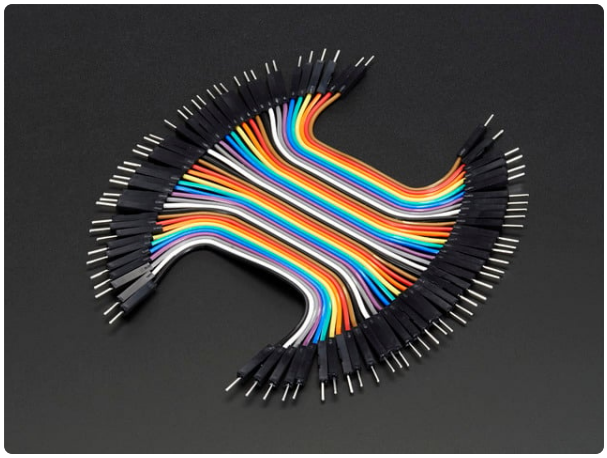
Add lots more outputs to a microcontroller system with chainable shift registers. These chips take a serial input (SPI) of 1 byte (8 bits) and then output those digital bits onto 8...

https://www.adafruit.com/product/450

5V 2A (2000mA) switching power supply - UL Listed

This is an FCC/CE certified and UL listed power supply. Need a lot of 5V power? This switching supply gives a clean regulated 5V output at up to 2000mA. 110 or 240 input, so it works...

https://www.adafruit.com/product/276



Premium Male/Male Jumper Wires - 40 x 3" (75mm)

Handy for making wire harnesses or jumpering between headers on PCB's. These premium jumper wires are 3" (75mm) long and come in a 'strip' of 40 (4 pieces of each of...

https://www.adafruit.com/product/759

Female DC Power adapter - 2.1mm jack to screw terminal block

If you need to connect a DC power wall wart to a board that doesn't have a DC jack - this adapter will come in very handy! There is a 2.1mm DC jack on one end, and a screw terminal...

https://www.adafruit.com/product/368

# Installing 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

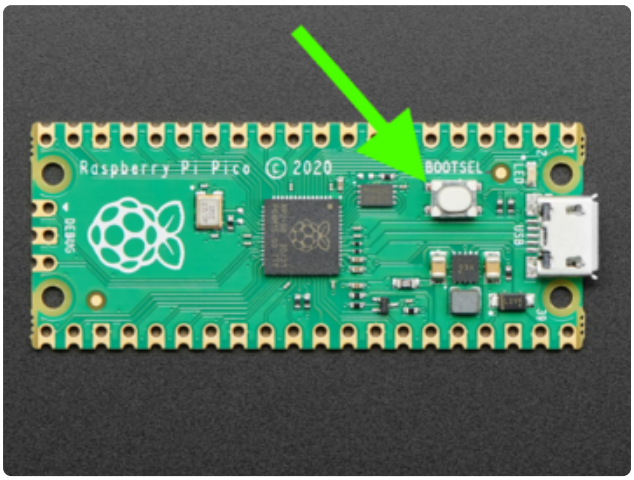
Follow this step-by-step to quickly get CircuitPython working on your board.

**Download the latest version of CircuitPython for the Raspberry Pi Pico from circuitpython.org**

https://adafru.it/QaP


**Click the link above and download the latest UF2 file.**

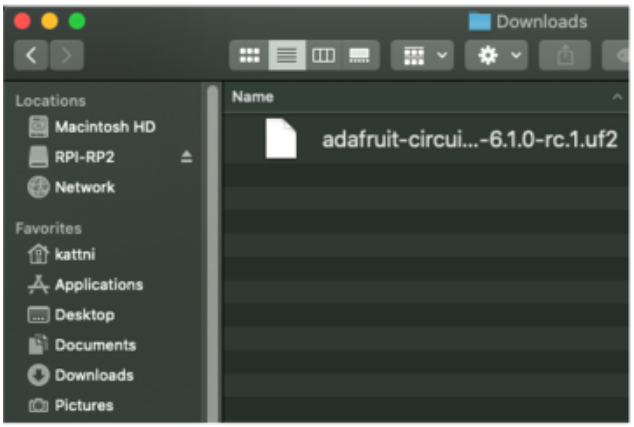
Download and save it to your desktop (or wherever is handy).



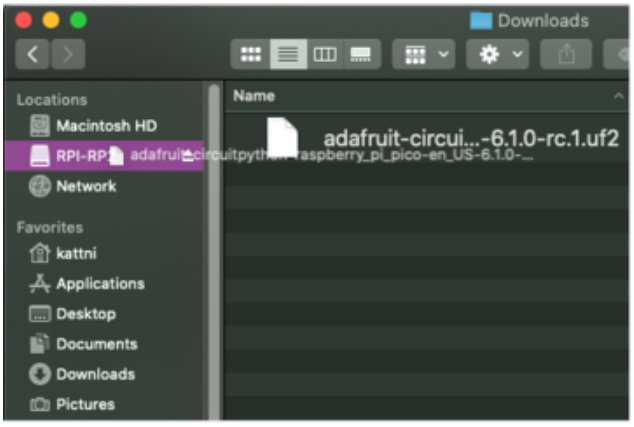

Start with your Pico unplugged from USB. Hold down the **BOOTSEL** button, and while continuing to hold it (don't let go!), plug the Pico into USB. **Continue to hold the BOOTSEL button until the RPI-RP2 drive appears!**

If the drive does not appear, unplug your Pico and go through the above process again.

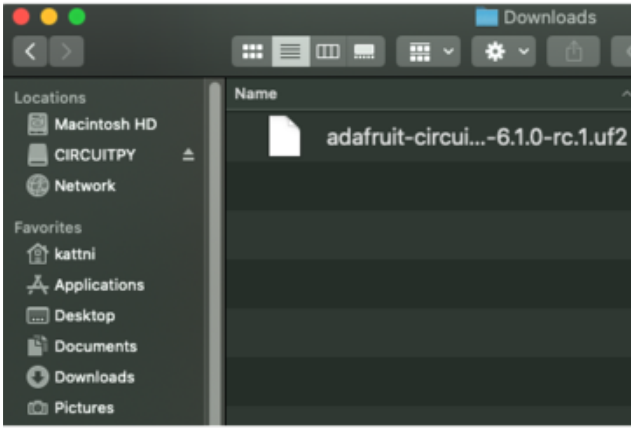
A lot of people end up using charge-only USB cables and it is very frustrating! **So make sure you have a USB cable you know is good for data sync.**




You will see a new disk drive appear called **RPI-RP2**.




Drag the **adafruit_circuitpython_etc.uf2** file to **RPI-RP2.**




The **RPI-RP2** drive will disappear and a new disk drive called **CIRCUITPY** will appear.

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

## Flash Resetting UF2

If your Pico ever gets into a really weird state and doesn't even show up as a disk drive when installing CircuitPython, try installing this 'nuke' UF2 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 nuking, re-install CircuitPython

**flash_nuke.uf2**

https://adafru.it/1afi

# Installing the Mu Editor

Mu is a simple code editor that works with the Adafruit CircuitPython boards. It's written in Python and works on Windows, MacOS, Linux and Raspberry Pi. The serial console is built right in so you get immediate feedback from your board's serial output!

Mu is our recommended editor - please use it (unless you are an experienced coder with a favorite editor already!).

## Download and Install Mu

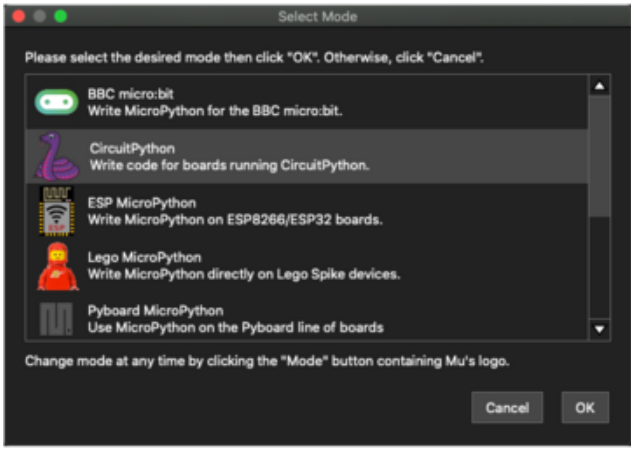

Download Mu from https://codewith.mu (https://adafru.it/Be6).

Click the **Download** link for downloads and installation instructions.

Click **Start Here** to find a wealth of other information, including extensive tutorials and and how-to's.

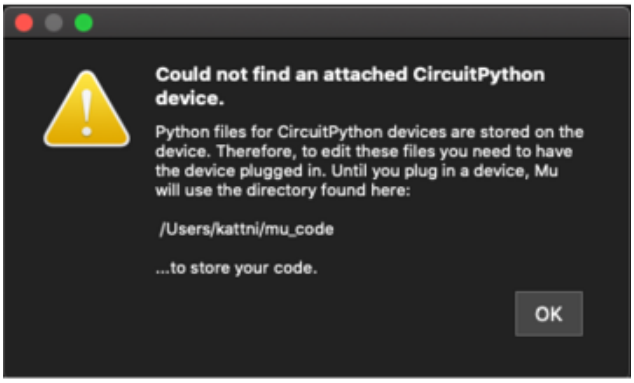
Windows users: due to the nature of MSI installers, please remove old versions of Mu before installing the latest version.

## Starting Up Mu




The first time you start Mu, you will be prompted to select your 'mode' - you can always change your mind later. For now please select **CircuitPython**!

The current mode is displayed in the lower right corner of the window, next to the "gear" icon. If the mode says "Microbit" or something else, click the **Mode** button in the upper left, and then choose "CircuitPython" in the dialog box that appears.




Mu attempts to auto-detect your board on startup, so if you do not have a CircuitPython board plugged in with a **CIRCUITPY** drive available, Mu will inform you where it will store any code you save until you plug in a board.

To avoid this warning, plug in a board and ensure that the **CIRCUITPY** drive is mounted before starting Mu.

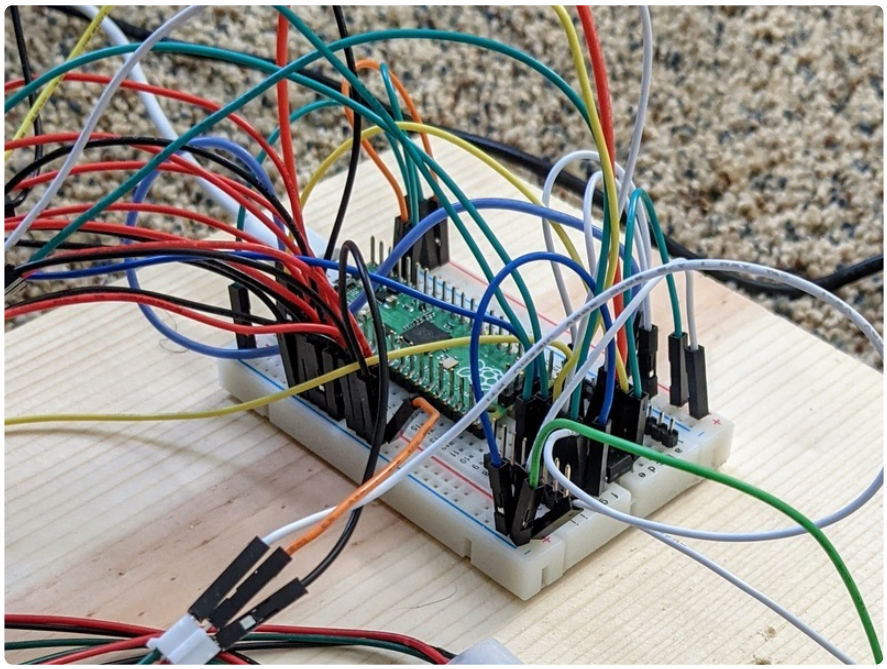
## Using Mu

You can now explore Mu! The three main sections of the window are labeled below; the button bar, the text editor, and the serial console / REPL.


Now you're ready to code! Let's keep going...

---

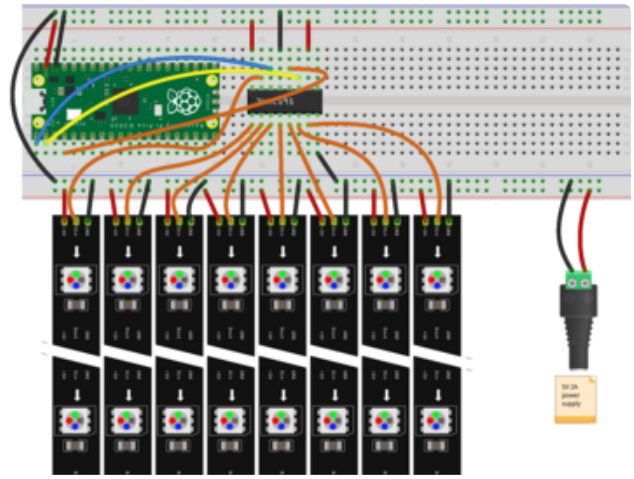
# Wiring and Code

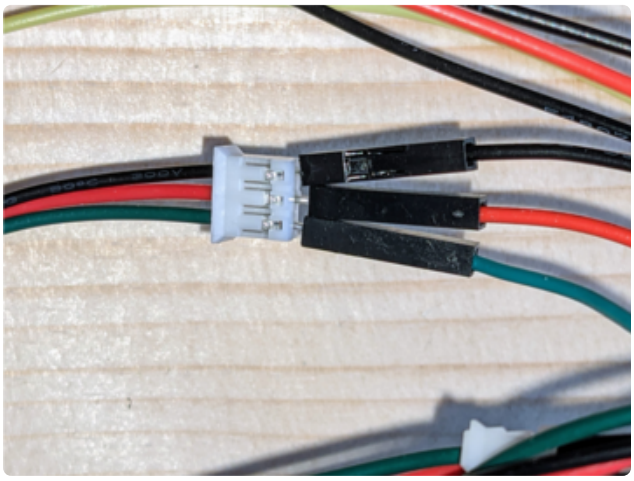


## CircuitPython Wiring

First, wire up the 74HC595 to your Raspberry Pi Pico. Then, connect all the NeoPixel strips. Once you've double-checked the wiring, you can connect the 5V 2A power supply and then connect the Pico to your computer using a USB cable.

To make this diagram clearer, it is shown on a full-size breadboard. However, a half-size breadboard has plenty of room for the connections you need.





**Pico VSYS** to **TOP SIDE power rail on breadboard**
**Pico GND** to **TOP SIDE ground rail on breadboard**
**TOP SIDE Ground rail on breadboard** to **BOTTOM SIDE ground rail on breadboard**
**74HC595 Pin 16** to **TOP SIDE power rail on breadboard**
**74HC595 Pin 10** to **TOP SIDE power rail on breadboard**
**74HC595 Pin 13** to **either ground rail on breadboard**
**74HC595 Pin 8** to **either ground rail on breadboard**
**74HC595 Pin 14** to **Pico GP0**
**74HC595 Pin 11** to **Pico GP1**
**74HC595 Pin 12** to **Pico GP2**
**5V 2A "+"** to **BOTTOM SIDE power rail**
**5V 2A "-"** to **BOTTOM SIDE ground rail**
**8 x NeoPixel GND** to **ground rail on breadboard**
**8 x NeoPixel VCC** to **BOTTOM SIDE power rail on breadboard**
**74HC595 Pins 1-7, 15** to **8 × NeoPixel DATA pins**

Depending on your NeoPixel strip, you may have to solder wires to the strip itself or use an adapter cable. The suggested strips have JST PH connectors, and you can insert male jumper wires into them to create the connections.

You can do this project with fewer than 8 strands. If you have fewer strands, connect them to the pins in this order: 15, 1, 2, 3, 4, 5, 6, 7.

If you want to power the Pico from the 5V 2A power supply, you can connect the **TOP SIDE power rail** and the **BOTTOM SIDE power rail together**. This way, you don't need an additional USB cable except when you need to connect the Pico to the computer to make changes to the code.

## Copying & Running the code


Use the "Download Project Zip" link below and unzip the files onto the **CIRCUITPY** drive that appears when you plug the Pico into your computer via a known good power + data USB cable.

CircuitPython will automatically re-load and start displaying 8 independent rainbow chases on your 8 LED strands. If you run into trouble, double-check your connections and if necessary open the REPL/serial in Mu to see any Python errors that may have occurred.

From here, you can apply any of the animations from the Adafruit LED Animations Library (https://adafru.it/LZF)! Treat the LEDs as a single strand, or independent strands using PixelMap.

To discover more about how the code works behind the scenes, continue on with this guide.

```
# SPDX-FileCopyrightText: 2021 Jeff Epler for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import random
from adafruit_led_animation.animation.rainbowcomet import RainbowComet
from adafruit_led_animation.helper import PixelMap
from adafruit_led_animation.group import AnimationGroup
from neopio import NeoPIO
import board

# Customize for your strands here
num_strands = 8
strand_length = 30
```

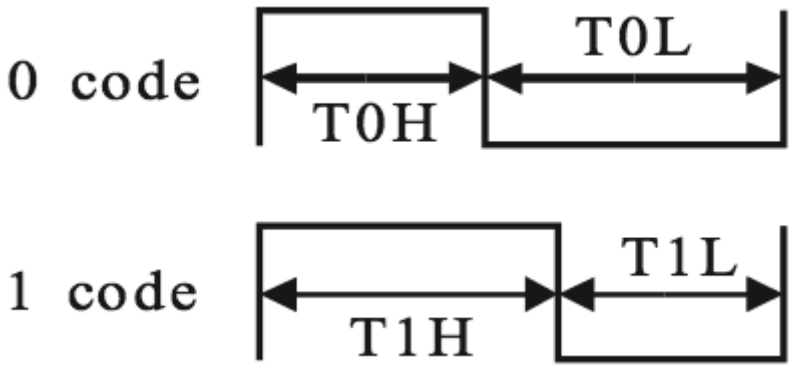
```
# Make the object to control the pixels
pixels = NeoPIO(board.GP0, board.GP1, board.GP2, num_strands*strand_length,
    num_strands=num_strands, auto_write=False, brightness=.18)

# Make a virtual PixelMap so that each strip can be controlled independently
strips = [PixelMap(pixels, range(i*strand_length, (i+1)*strand_length),
individual_pixels=True)
    for i in range(num_strands)]

# This function makes a comet animation with slightly random settings
def make_animation(strip):
    speed = (1+random.random()) * 0.02
    length = random.randrange(18, 22)
    bounce = random.random() > .5
    offset = random.randint(0, 255)
    return RainbowComet(strip, speed=speed, tail_length=length, bounce=bounce,
        colorwheel_offset=offset)

# Make an animation for each virtual strip
animations = [make_animation(strip) for strip in strips]

# Put the animations into a group so that we can animate them together
group = AnimationGroup(*animations, )

# Show the animations forever
while True:
    group.animate()
```

---

# Code Walkthrough: PIO

Let's review how data is transmitted to a NeoPixel (https://adafru.it/jFu): A "1"-bit is sent as a LONGER period of HIGH voltage followed by a SHORTER period of LOW voltage. A "0"-bit is sent as a SHORTER period of HIGH voltage followed by a LONGER period of LOW voltage.

The code on this page uses ratios of 1:2 and 2:1. **This isn't the most compatible waveform possible**, but it works on most strips that identify themselves as NeoPixel-compatible.




Since each pixel is made up of 24 bits of red, green, and blue data, this cycle repeats 24 times for each pixel, and 720 times for a strand of 30 pixels. All together, that takes just under 1 millisecond.

When running CircuitPython, the `neopixel_write` module has optimized code (often in assembler) to produce exactly the required waveform. On the Raspberry Pi Pico with RP2040 microcontroller, this is actually done with the PIO programmable I/O block.

However, the PIO is so powerful that it laughs at the simplicity of running a single LED strip and boasts that (with the addition of the lowly 74HC595 shift register) it can run 8 LED strips from just 3 I/O pins.

First, let's look at the code that works on the PIO programmable I/O block:

```
.program neopio
.side_set 2

.wrap_target
    set x, 7            side 2
    pull

bitloop0:
    set pins, 1         side 0
    jmp x--, bitloop0   side 1
    set x, 7            side 2
bitloop1:
    out pins, 1         side 0
    jmp x--, bitloop1   side 1
    set x, 7            side 2
bitloop2:
    set pins, 0         side 0
    jmp x--, bitloop2   side 1
.wrap
```

The code is divided into 3 loops ( `bitloop0` , `bitloop1` , and `bitloop2` ), plus a little housekeeping code. The full loop will run in about 1.25 microseconds, and transfers 1 bit for each of the 8 NeoPixel strips.

The register **x** is used as a loop variable. The loop instruction **jmp x--** checks if **x** is zero, then updates x by subtracting 1, and depending on the previous value of **x** jumps to the given label (if nonzero) or continues to the next line (if zero). By storing 7 in **x** to begin with, each loop executes 8 times (this may make more sense to you if you're a robot)

In the first loop, the first instruction is "set pins, 1" which turns the first 'set pin' to **HIGH**.

In the second loop, the first instruction is "out pins, 1" which turns the first 'out pin' to either **HIGH** or **LOW** depending on the next bit of data sent to it from the data buffer in Python. Note that the 'set pin' and 'out pin' are actually the same pin in this case.

In the third loop, the first instruction is "set pins, 0" which turns the 'set pin' to **LOW**.

See how the structure of the three loops reflect the structure of a NeoPixel bit transmission? The first part is always **HIGH**, the second part depends on the data to be transmitted, and the third part is always **LOW**.

Next: to understand how each bit gets received by the shift register with concentration on the middle bitloop:

```
bitloop1:
    out pins, 1         side 0
    jmp x--, bitloop1   side 1
    set x, 7            side 2
```

This time, concentrate on the part of each line that says **side N**. This instruction updates the value of the side-set pins. There are two side-set pins, and the possible side values are:

- **0:** turn both pins **LOW**
- **1**: turn first pin **HIGH** and second pin **LOW**
- **2**: turn first pin **LOW** and second pin **HIGH**
- **3**: turn both pins **HIGH** (not used in this program)

Side-set pin 1 is connected to the shift register's clock pin, and side-set pin 2 is connected to the shift register's parallel load pin. Each of those functions is activated by a rising edge (from **LOW** to **HIGH**). This means that in the loop the data pin is updated first with its new value (`out pins`) with clock low (`side 0`) and then the clock pin is activated (`side 1`).

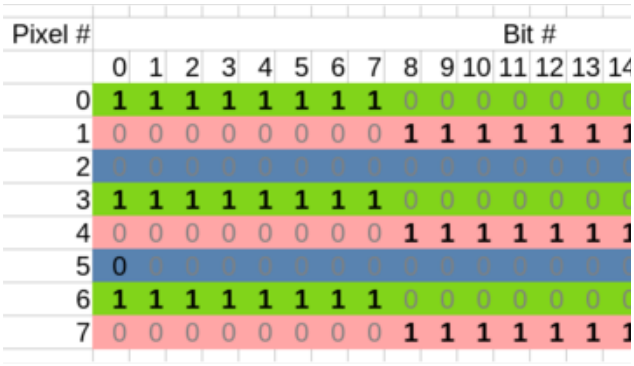
When the loop finishes, the parallel load pin is activated (`side 2`) and the 8 values just sent out appear on the 8 output pins of the '595.

---

# Code Walkthrough: Bit Manipulation

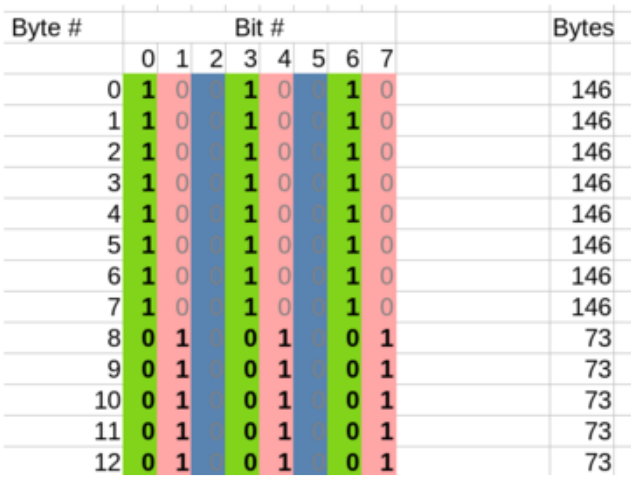
So far you've seen how to use PIO to send data to NeoPixels, but the data also has to be in the correct format: Each 8 bits sent has to describe a part of 8 different pixels, one in each strip.

This means re-formatting the bits that make up the pixels. There is a function do this quickly in the CircuitPython core, but below will explain it in this section with diagrams and Python code.



| Pixel # | Bit # | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Normally, color data for NeoPixels is stored in memory as a series of 24-bit (sometimes 32-bit) values, with one pixel after another.



| Byte # | Bit # | | | | | | | | Bytes |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 146 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 146 |
| 2 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 146 |
| 3 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 146 |
| 4 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 146 |
| 5 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 146 |
| 6 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 146 |
| 7 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 146 |
| 8 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 73 |
| 9 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 73 |
| 10 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 73 |
| 11 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 73 |
| 12 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 73 |

See how the bits for each color now have a vertical orientation? After rearrangement, it's like the data has been rotated 90 degrees (or "transposed", to use a more mathematical term.

When the shift register receives each group of data, it sends the "bit 0" column to the first strand, the "bit 1" column to the second strand, and so forth.

The Python code shown below transforms the data of 8 strands into data for the shift register:

```
n = len(strand_data[0])
buf = bytearray(n*8)
for output_idx in range(n*8):    # For each output byte, ...
    input_idx = output_idx &gt;&gt; 3
    input_bit = 0x80 &gt;&gt; (output_idx &amp; 0x7)
    b = 0
    for strand_idx in range(8):  # Find 8 input bits
        if strand_data[strand_idx][input_idx] &amp; input_bit:
            b |= 1&lt;&lt;strand_idx   # Place them in the right spot
    buf[output_idx] = b
```

However, since our goal is raw speed, we don't want to actually perform these steps using interpreted Python code. On RP2040-based boards like the Raspberry Pi Pico there is a function `bitops.bit_transpose` which contains highly optimized code to perform this transformation.

# Code Walkthrough: Being a PixelBuf

This is all great, but you probably have existing code using the Animations library that you'd like to use with NeoPIO. How can we make the two work together? By implementing a subclass of PixelBuf! Select parts of the code are listed below.

Instead of taking a single I/O pin, the constructor takes three: `data`, `clock`, and `strobe`. It also requires that you specify the number of strands (`num_strands`), from 2 to 8. The first step is some necessary error checking:

```
class NeoPIO(adafruit_pixelbuf.PixelBuf):
    def __init__(
        self, data, clock, strobe, n, *, num_strands=8, bpp=3, brightness=1.0,
auto_write=True, pixel_order=None
    ):
        if not _pin_directly_follows(data, clock):
            raise ValueError("clock pin must directly follow data pin")
        if not _pin_directly_follows(clock, strobe):
            raise ValueError("strobe pin must directly follow clock pin")

        if n % num_strands:
            raise ValueError("Length must be a multiple of num_strands")
```

Next, the steps to construct the underlying PixelBuf object, just like NeoPixel:

```
…
        if not pixel_order:
            pixel_order = GRB if bpp == 3 else GRBW
        else:
            if isinstance(pixel_order, tuple):
                order_list = [RGBW[order] for order in pixel_order]
                pixel_order = "".join(order_list)

        super().__init__(
            n, brightness=brightness, byteorder=pixel_order, auto_write=auto_write
        )
```

Last, we need to store the number of strands so we can use that number later, create some scratch memory for the transposed bits, and then (the interesting part!) create the PIO state machine:

```
…
        self._transposed = bytearray(bpp*n*8//num_strands)
        self._num_strands = num_strands

        self._sm = rp2pio.StateMachine(
            _assembled,
            frequency=800_000 * 52,
            init=adafruit_pioasm.assemble("set pindirs 7"),
            first_out_pin=data,
            out_pin_count=1,
            first_set_pin=data,
            set_pin_count=3,
            first_sideset_pin=clock,
            sideset_pin_count=2,
            auto_pull=True,
```

```
            out_shift_right=False,
            pull_threshold=8,
        )
```

There are some other definitions to supply that are identical to NeoPixel, or are not very interesting, such as the method `deinit` and the property `num_strands`.

The last thing to provide is the `_transmit` method, to blast out those pixel values. There are two steps—transposing the bits, then writing them out—yet It's almost anticlimactic how simple this is:

```
…
    def _transmit(self, buffer):
        bitops.bit_transpose(buffer, self._transposed, self._num_strands)
        self._sm.write(self._transposed)
```