An Introduction to RP2040 PIO with CircuitPython

Created by Jeff Epler

```
1. program = ""
2. .program ws2812
3. .side_set 1
4. .wrap_target
5. bitloop:
6.   out x 1   side 0 [1]; Side-set still takes
   place when instruction stalls
7.   jmp !x do_zero side 1 [1]; Branch on the bit shifted
   out. Positive pulse
8. do_one:
9.   jmp bitloop side 1 [1]; Continue driving high,
   for a long pulse
10. do_zero:
11.   nop   side 0 [1]; Or drive low, for a short
       pulse
12. .wrap
13. ""
```

Last updated on 2021-11-05 01:09:05 PM EDT
Guide Contents

Guide Contents

Overview 3
- Major differences between adafruit_pioasm and standard pioasm 3
  .wrap and .wrap_target are not supported 3
- mov operator restrictions 4
- Expressions are not supported 5
Products 7
Installing CircuitPython 8
- CircuitPython Quickstart 8
- Flash Resetting UF2 9
Installing the Mu Editor 11
- Download and Install Mu 11
- Starting Up Mu 11
- Using Mu 12
Installing Libraries 13
Using PIO to turn an LED on and off 14
- Code Walkthrough 15
Using PIO to control LED brightness 18
- Code Walkthrough 19
Using PIO to blink a LED quickly or slowly 22
- Code Walkthrough 23
Using PIO to drive a NeoPixel 26
- Parts 26
  Full Code Listing 27
  Code Walk-through 29
API Documentation: adafruit_pioasm 32
API Documentation: rp2pio 33
Overview

A feature that sets the Raspberry Pi Foundation RP2040 microcontroller apart from other microcontrollers is "PIO". The RP2040 datasheet says that the "programmable input/output block (PIO) is a versatile hardware interface. It can support a variety of IO standards... PIO is programmable in the same sense as a processor."

In this guide, you'll learn how to write and use PIO programs from CircuitPython. The official datasheet (https://adafruit.it/QLC) (chapter 3), the book "Get Started with MicroPython on Raspberry Pi Pico" (https://adafruit.it/QLD) and pico-examples (https://adafruit.it/Qa2) (pio folder) are helpful resources too, but CircuitPython sometimes deviates from the way that PIO is used in other environments like C or MicroPython.

Examples in this guide require CircuitPython 6.2.0-beta.3 or later, or the Absolute Latest version.

Major differences between adafruit_pioasm and standard pioasm

.wrap and .wrap_target are not supported

Instead, an implicit .wrap is at the bottom of every program, and an implicit .wrap_target is at the top of every program. You can add a jmp instruction instead of using .wrap but remember that this slightly changes the instruction length and the timing of your pio program.
mov operator restrictions

The `mov` instruction can accept an optional argument, called an operator, to reverse (`::`) or invert (`~`) its argument. In adafruit_pioasm, one or more spaces must come between the `::` or `~` and the operand. These spaces are not required by the official dialect.
The official dialect allows "!" to mean the same as "\~". This is not accepted by adafruit_pioasm.

Expressions are not supported

Standard pioasm supports expressions like \([T1 + 1]\). In pioasm, calculations are not supported. Instead,
use string formatting operations to insert the computed value where necessary, e.g.,

```python
f"...{T1+ 1}...""
```
Products

These examples were designed for the Raspberry Pi Pico but can be adapted to the range of boards with the RP2040 microcontroller. The QT Py RP2040 requires the addition of an external LED and current-limiting resistor for the standard LED examples.

**Adafruit Feather RP2040**
A new chip means a new Feather, and the Raspberry Pi RP2040 is no exception. When we saw this chip we thought "this chip is going to be awesome when we give it the Feather..."
$11.95
In Stock
Add to Cart

**Adafruit ItsyBitsy RP2040**
A new chip means a new ItsyBitsy, and the Raspberry Pi RP2040 is no exception. When we saw this chip we thought "this chip is going to be awesome when we give it the ItsyBitsy..."
$9.95
In Stock
Add to Cart

**Adafruit QT Py RP2040**
What a cutie pie! Or is it... a QT Py? This diminutive dev board comes with one of our new favorite chip, the RP2040. It's been made famous in the new
$9.95
In Stock
Add to Cart

**Raspberry Pi Pico RP2040**
The Raspberry Pi foundation changed single-board computing when they released the Raspberry Pi computer, now they're ready to...
$4.00
In Stock
Add to Cart

© Adafruit Industries
https://learn.adafruit.com/intro-to-rp2040-pio-with-circuitpython
**Installing CircuitPython**

CircuitPython ([https://adafruit.it/tB7](https://adafruit.it/tB7)) is a derivative of MicroPython ([https://adafruit.it/BeZ](https://adafruit.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.

[Click the link above and download the latest UF2 file.](https://adafruit.it/QaP)

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

https://adafruit.it/QAJ

https://adafruit.it/QAJ
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 how-to's.

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...
Installing Libraries

You'll need to install the Adafruit_CircuitPython_Pioasm library on your CircuitPython board.

First make sure you are running the latest version of Adafruit CircuitPython (https://adafruit.it/Amd) for your board.

Next you'll need to install the necessary libraries to use the hardware--carefully follow the steps to find and install these libraries from Adafruit's CircuitPython library bundle (https://adafruit.it/uap). Our CircuitPython starter guide has a great page on how to install the library bundle (https://adafruit.it/ABU).

Manually install the necessary library from the bundle:

- adafruit_pioasm.mpy

Before continuing, make sure your board's lib folder or root filesystem has the adafruit_pioasm.mpy file copied over.
Using PIO to turn an LED on and off

Normally, you’d use `DigitalInOut` to turn an LED on and off in CircuitPython. However, as a simple introduction to PIO, it can also be used to turn an LED on or off.

Here’s a CircuitPython program to do just that:
import time
import board
import rp2pio
import adafruit_pioasm

hello = ""
  .program hello
  loop:
    pull
    out pins, 1
  jmp loop
"

assembled = adafruit_pioasm.assemble(hello)

sm = rp2pio.StateMachine(
    assembled,
    frequency=2000,
    first_out_pin=board.LED,
)
print("real frequency", sm.frequency)

while True:
    sm.write(bytes((1,)))
    time.sleep(0.5)
    sm.write(bytes((0,)))
    time.sleep(0.5)

Save the file below as code.py and transfer it to your CIRCUITPY drive. Your device should automatically restart and run the code. Shortly, a LED will blink on and off about once every second. If it doesn't, use Mu to connect to the Serial REPL (https://adafruit.it/Awz) of your device and you'll be able to see any errors that occurred.

Code Walkthrough

The full program is shown above, but let's look at the interesting bits a few lines at a time:
PIO programs are included within your CircuitPython source as strings, and then converted into a program with the `assemble` function.

```python
sm = rp2pio.StateMachine(
    assembled,
    frequency=10000,
    first_out_pin=board.LED,
)
```

The PIO peripheral contains several "state machines", which are the units that run PIO programs. The `StateMachine` constructor takes the assembled program as well as some additional information:

- `frequency` says how quickly each pio instruction executes. If you have a task that you need to take "exactly X microseconds" or "execute at exactly Y kHz", this will allow you to determine the right frequency value.
- `first_out_pin` names the first pin that will be updated by out instructions in the PIO program. This program only affects a single pin.

```python
while True:
    sm.write(bytes((1,)))
    time.sleep(0.5)
    sm.write(bytes((0,)))
    time.sleep(0.5)
```

The forever-loop of our Python code alternates between sending the byte 1 and the byte 0 to the PIO state machine. Each time a byte is sent, the PIO program acts on it.

```
loop:
    pull
    out pins, 1
    jmp loop
```

Every PIO instruction is documented in the RP2040 datasheet, so while this guide will give a high-level description of what is happening, it eliminates many details in order to keep the descriptions sort.
The first line, `loop:`, is a label. This line, together with the last line `jmp loop` create a forever-loop.

The second line contains the first instruction, `pull`, which waits until a value is sent to the State Machine (A value is sent by the `sm.write` function calls in our Python code). The value is stored in a location (register) called the **OSR (Output Shift Register)**.

The next line contains another instruction, `out pins, 1`. This is our first instruction with operands. The first operand, `pins`, says where the data is being transferred to. The second operand, `1`, says how many bits are being transferred. The source of the data is the **OSR**, the same as the implicit destination of the `pull` instruction.

The final line contains the last instruction, `jmp loop`. A `jmp` instruction makes the program continue at the named location instead of the next line.

The net effect of this program is to turn the related pin HIGH if the number sent is even and the pin LOW if the number sent is odd. And that's why the Python forever-loop makes the Pico's LED turn off and on about once per second.
Using PIO to control LED brightness

Normally, you’d use `PWMOut` to control the brightness of an LED connected to a GPIO pin. However, as you may expect, you can also use PIO to create a PWM-like effect.

Here’s a CircuitPython program to do just that:
Code Walkthrough

The full program is shown above, but let's look at the interesting bits a few lines at a time. In this example,
there are three different PIO programs, each one for a different brightness level:

```python
led_quarter_brightness = adafruit_pioasm.assemble(""" set pins, 0 [2] set pins, 1 """)
led_half_brightness = adafruit_pioasm.assemble(""" set pins, 0 set pins, 1 """)
led_full_brightness = adafruit_pioasm.assemble(""" set pins, 1 """)
```

The "full brightness" program is most self-explanatory: at each moment, it sets its corresponding pin to 1.

The "half brightness" program alternately sets its pin to 0 and then to 1. When it changes rapidly between these two states, the LED appears lit, but dim, to a human eye. With a few exceptions, each instruction takes the same length of time. For one instruction the LED is off and for one instruction the LED is on, so the LED is turned on half the time.

The "quarter brightness" program needs the most explanation. The new element on the first set line, [2], indicates that after the set command, there is an additional delay of 2 cycles before going to the next instruction. That means that the pin is 0 for 3 cycles and 1 for 1 cycle, giving a ratio of 1/4.

```python
while True:
    sm = rp2pio.StateMachine(led_quarter_brightness,
                               frequency=10000, first_set_pin=board.LED)
    time.sleep(1)
    sm.deinit()

    sm = rp2pio.StateMachine(led_half_brightness,
                               frequency=10000, first_set_pin=board.LED)
    time.sleep(1)
    sm.deinit()

    sm = rp2pio.StateMachine(led_full_brightness,
                               frequency=10000, first_set_pin=board.LED)
    time.sleep(1)
    sm.deinit()
```

The CircuitPython forever loop cycles among the three programs, for one second each. frequency=10000, or 10kHz, means that the on-off cycle of the LED is far too fast to be seen by a human eye; the quarter-brightness LED turns on and off 2500 times per second, or 100 times faster than a "24fps" film.
Because the LED is turned fully off for a short time between programs, you may see a flicker when the brightness changes.
Using PIO to blink a LED quickly or slowly

The next example program introduces new concepts: the pull instruction, which receives data from the CircuitPython program; and registers, which are similar to variables in that they can store values and some simple operations can be performed on those values. However, the uses of PIO registers are much more restricted than the use of variables in Python.
import array
import time
import board
import rp2pio
import adafruit_pioasm

blink = adafruit_pioasm.assemble(".program blink
   pull block    ; These two instructions take the blink duration
   out y, 32     ; and store it in y
forever:
   mov x, y
   set pins, 1   ; Turn LED on
lp1:
   jmp x-- lp1   ; Delay for (x + 1) cycles, x is a 32 bit number
   mov x, y
   set pins, 0   ; Turn LED off
lp2:
   jmp x-- lp2   ; Delay for the same number of cycles again
   jmp forever  ; Blink forever!
""
)

while True:
    for freq in [5, 8, 30]:
        with rp2pio.StateMachine(blink,
           frequency=125_000_000,
           first_set_pin=board.LED,
           wait_for_txstall=False,
        ) as sm:
            data = array.array("I", [sm.frequency // freq])
            sm.write(data)
            time.sleep(3)
            time.sleep(0.5)

Code Walkthrough

pull block    ; These two instructions take the blink duration
   out y, 32     ; and store it in y

The instruction "pull block" says to wait until a value sent from CircuitPython is available ("block"), and then
to pull that value into a holding area known as OSR, or Output Shift Register. Then, all 32 bits of OSR are stored in the register called Y. Later, the CircuitPython program will send in a number that represents how long the LED spends in an on or off state, so remember that this is what the register Y now holds.

```
forever:
  mov x, y
  set pins, 1   ; Turn LED on
lp1:
  jmp x-- lp1   ; Delay for (x + 1) cycles, x is a 32 bit number
```

The next few lines' purpose is to turn the LED on, then wait for the desired length of time.

Working up from the bottom of this section of code, the last two lines create a loop that delays for (x+1) cycles. lp1: is a label, and a jmp instruction can skip to it instead of continuing to the next instruction. In this case, the jump is conditional on x--, which means "if X is not zero, jump to lp1. In any case, decrease X by 1." Because the delay numbers are large and because the program will change the delay value without changing the PIO program itself, it cannot use the [#] notation to delay as in the previous program.

The set instruction to turn on the LED should be familiar by now.

The mov instruction takes the value in Y and copies it to X. If not, and the loop used jmp y--, then it would lose the original delay value. But the value is needed each time the program has a delay. Happily, there are two register X and Y so the program can just take a copy of the original delay value each time.

Remember that there's a forever: label here, it will be used later.

```
mov x, y
  set pins, 0   ; Turn LED off
lp2:
  jmp x-- lp2   ; Delay for the same number of cycles again
  jmp forever   ; Blink forever!
```

The next block is very much like the previous block, except that set is used to turn the LED off before the delay.

The final line, jmp forever, sends us back to the first delay. If it instead relied on the automatic wrap back to the first instruction, there would only be a single on-off blink before the program went back to the pull block instruction and waited for a new blink duration to be sent in, which would not give the desired result.
while True:
    for freq in [5, 8, 30]:
        with rp2pio.StateMachine(blink,
            frequency=125_000_000,
            first_set_pin=board.LED,
            wait_for_txstall=False,
        ) as sm:
            data = array.array("I", [sm.frequency // freq])
            sm.write(data)
            time.sleep(3)
            time.sleep(0.5)

The Python forever-loop repeatedly cycles through the frequencies 5, 8, and 30.

For each frequency, it creates a state machine with our program, calculates and send the required delay value to it, and waits 3 seconds. Then, before continuing with the next blink pattern, it delays a half a second.

Of course, your CircuitPython program doesn't need to sleep while PIO is making the LED blink, it could be doing calculations, updating an LCD display, reading button presses. The PIO program keeps running independently of what CircuitPython is doing.

In many applications of PIO, such as sending data out to NeoPixels, a write call needs to wait until the PIO program has completed. Since PIO programs run endlessly, there needs to be some definition of "completed". The usual definition is "the PIO program (through a pull instruction) requested fresh data from CircuitPython, but none was available". This is called a "transmit stall" or "txstall". Thus, the line

```python
wait_for_txstall=False
```

means that CircuitPython does not wait for this condition before the `sm.write(data)` returns.
Using PIO to drive a NeoPixel

If you are using a board with a built-in NeoPixel, the example on this page will use it. If you have another board, such as a Raspberry Pi Pico, you'll need to connect it to an external NeoPixel:

- Connect Pico VSYS to NeoPixel + or VCC
- Connect Pico GND to NeoPixel GND
- Connect Pico GP16 to NeoPixel In or DIN

You can also select whatever pin you like and change the CircuitPython code accordingly.

You can use any NeoPixel (including a NeoPixel strip) but this code example only drives a single NeoPixel at a time.

Parts

If you have the Raspberry Pi Pico or another RP2040 board which does not have a built-in NeoPixel, you'll need the parts below to follow this example. Adafruit's RP2040-based boards include a built-in NeoPixel.

Breadboard-friendly RGB Smart NeoPixel - Pack of 4
This is the easiest way possible to add small, bright RGB pixels to your project. We took the same technology from our Flora NeoPixels and made them breadboard friendly, with two rows...

$7.95
In Stock
Add to Cart
Half-Size Breadboard with Mounting Holes
This cute 3.2" × 2.1" (82 × 53mm) solderless half-size breadboard has four bus lines and 30 rows of pins, our favorite size of solderless breadboard for...
$5.00
In Stock

Breadboarding wire bundle
75 flexible stranded core wires with stiff ends molded on in red, orange, yellow, green, blue, brown, black and white. These are a major improvement over the "box of bent... $4.95
In Stock

Full Code Listing
import time
import rp2pio
import board
import microcontroller
import adafruit_pioasm

# NeoPixels are 800khz bit streams. Zeroes are 1/3 duty cycle (~416ns) and ones
# are 2/3 duty cycle (~833ns).
program = ""
    .program ws2812
    .side_set 1
    .wrap_target
    bitloop:
        out x 1  side 0 [1]; Side-set still takes place when instruction stalls
        jmp !x do_zero side 1 [1]; Branch on the bit we shifted out. Positive pulse
        do_one:
            jmp bitloop  side 1 [1]; Continue driving high, for a long pulse
        do_zero:
            nop  side 0 [1]; Or drive low, for a short pulse
    .wrap
""

assembled = adafruit_pioasm.assemble(program)

# If the board has a designated neopixel, then use it. Otherwise use
# GPIO16 as an arbitrary choice.
if hasattr(board, "NEOPIXEL"):
    NEOPIXEL = board.NEOPIXEL
else:
    NEOPIXEL = microcontroller.pin.GPIO16

sm = rp2pio.StateMachine(
    assembled,
    frequency=800000 * 6,  # 800khz * 6 clocks per bit
    first_sideset_pin=NEOPIXEL,
    auto_pull=True,
    out_shift_right=False,
    pull_threshold=8,
)
print("real frequency", sm.frequency)

for i in range(30):
    sm.write(b"\x0a\x00\x00")
    time.sleep(0.1)
    sm.write(b"\x00\x0a\x00")
    time.sleep(0.1)
    sm.write(b"\x00\x00\x0a")
    time.sleep(0.1)
print("writes done")

time.sleep(2)
Code Walk-through

Exactly how does a NeoPixel expect to receive data? Here's how. Each original bit needs to be sent in approximately 1200 microseconds. In the transmission program, 1200 microseconds are divided into three equal portions.

To transmit a "0", the pin is set \textbf{HIGH} during the first third, and then \textbf{LOW} during the other two thirds.

To transmit a "1", the pin is set \textbf{HIGH} during the first two thirds and low during the last third.

This is repeated 24 times for each RGB NeoPixel, and a strip repeats these 24 cycles once for each pixel—once a pixel has received its own data, it sends any more data to the next pixel using its Data Out (\texttt{DOUT}) pin.

A short delay without any pulses makes the pixels ready to receive fresh values.

```python
program = ""
 .program ws2812
 .side_set 1
 .wrap_target
 bitloop:
   out x 1 side 0 [1]; Side-set still takes place when instruction stalls
   jmp !x do_zero side 1 [1]; Branch on the bit shifted out. Positive pulse
.do_one:
   jmp bitloop side 1 [1]; Continue driving high, for a long pulse
.do_zero:
   nop side 0 [1]; Or drive low, for a short pulse
 .wrap
 ""
```

The first new concept is "side set". So far, PIO has changed a pin value using "out"; with "side set" PIO can change the value of a pin while also doing some other activity. This is from the English idiom "do something on the side", which means in addition to one's regular job or duties.

When an instruction has \texttt{side 0} next to it, the corresponding output is set \texttt{LOW}, and when it has \texttt{side 1} next to it, the corresponding output is set \texttt{HIGH}. There can be up to 5 side-set pins, in which case \texttt{side N} is interpreted as a binary number.

The first instruction, \texttt{out x 1}, transfers the next NeoPixel data bit into the x register. It also ensures the data out pin is \texttt{LOW} until the data bit is available. This creates the delay necessary between refreshes of the NeoPixel strip, as well as the \texttt{LOW} period at the end of each transmitted bit.
Because the state machine is created with `auto_pull=True`, there's no need for a `pull` instruction.

Next, `jmp !x do_zero side 1` sets the output pin `HIGH` and then continues either at the next line (if X is nonzero) or at `do_zero` (if it is zero). `!x` means "if X is zero" and is taken from the syntax of C/C++/Arduino code.

Depending whether execution continues at `do_one` or `do_zero`, the middle portion of the signal is transmitted as `HIGH` or `LOW`. `nop` indicates that "no operation" (except the side-set operation) is performed by the instruction.

Either way, PIO continues from `bitloop`, setting the output pin `LOW` and then getting the next pixel data into X. If there's more pixel data waiting to be transmitted, it will continue immediately on to the next lines; otherwise, it will wait until more pixel data is available.

If you consider each possible path through a single loop (X is zero; X is nonzero) and count the number of instructions plus the number of `[N]` delays, you will see that there are 6 clocks before the execution returns to `bitloop`.

```python
sm = rp2pio.StateMachine(
    assembled,
    frequency=800000 * 6,  # 800khz * 6 clocks per bit
    first_sideset_pin=NEOPIXEL,
    auto_pull=True,
    out_shift_right=False,
    pull_threshold=8,
)
print("real frequency", sm.frequency)
```

Accordingly, the StateMachine is constructed with a requested frequency based on the desired bit rate (800kHz) times the number of cycles per bit (6).

As discussed above, since the `out x` instruction should wait until data is available and then automatically pull it in from CircuitPython, `auto_pull=True` is specified.

`out_shift_right` controls how multi-bit values are sent in. NeoPixels expect the "most significant bits" first, which is the order you get when `out_shift_right=False`.

`first_sideset_pin` controls the pin(s) which are set by side-set operations.

`pull_threshold` controls the minimum number of bits that have to be available for an auto-pull to complete. Since neopixels are a sequence of bytes, set the value to 8, the number of bits in a byte.

PIO can't exactly provide any requested frequency. In this case, instead of the exact value 4800000Hz a slightly different value of 4799760Hz is provided. This is well within the tolerance of NeoPixels. When a device has to be controlled at a very specific frequency, it's important to check that your program is
running at a rate that is close enough to the required rate.

```python
for i in range(30):
    sm.write(b"\x0a\x00\x00")
    time.sleep(0.1)
    sm.write(b"\x00\x0a\x00")
    time.sleep(0.1)
    sm.write(b"\x00\x00\x0a")
    time.sleep(0.1)
```

It's finally time to light up your NeoPixel by directly specifying the bytes to send to it. For most RGB NeoPixels, this program will send a sequence of green, red, and blue pixels, with 30 repetitions.

On the RP2040, the standard `neopixel` module works very much in the way shown here, but it's ready to work with the other CircuitPython libraries you may already know and love, like the LED animations library [here](https://adafruit.it/LZF).

For a more sophisticated example of driving 8 NeoPixel strips from just 3 GPIO pins using PIO, there's a dedicated guide [here](https://adafruit.it/QMd).
API Documentation: rp2pio

[API Documentation: rp2pio](https://adafruit.it/QMf)