An Introduction to RP2040 PIO with CircuitPython

Created by Jeff Epler

```
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
"""
```

https://learn.adafruit.com/intro-to-rp2040-pio-with-circuitpython

Last updated on 2022-12-01 03:08:23 PM EST

Table of Contents

Overview

```
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
"""
```

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 () (chapter 3), the book "Get Started with MicroPython on Raspberry Pi Pico" () and pico-examples () (pio folder) are helpful resources too, but CircuitPython sometimes deviates from the way that PIO is used in other environments like C or MicroPython.

Major differences between pio in CircuitPython compared to standard pioasm

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

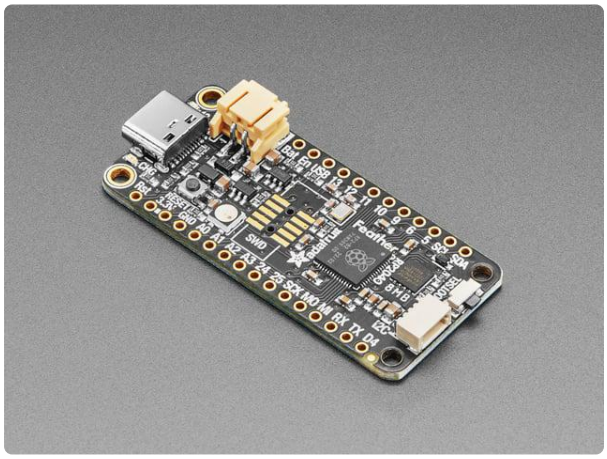
Standard pioasm supports expressions like `[T1 + 1]`. In pioasm, calculations are not supported. Instead, use string formatting operations to insert the computed value where necessary, e.g., `f"""…[{T1+ 1}]…"""`

Interrupts are not supported

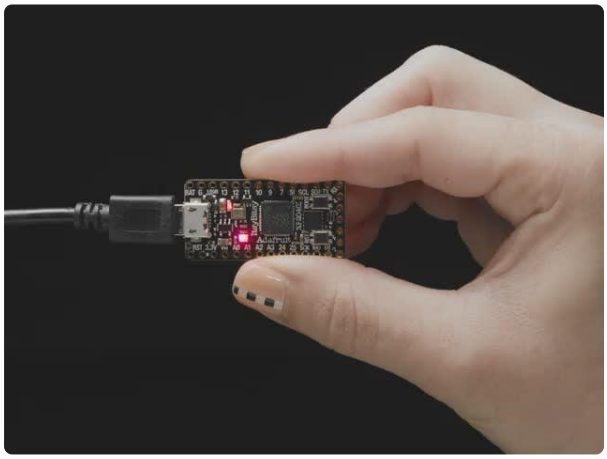
The IRQ method of sending signals out of a PIO program is not supported in CircuitPython.

Products

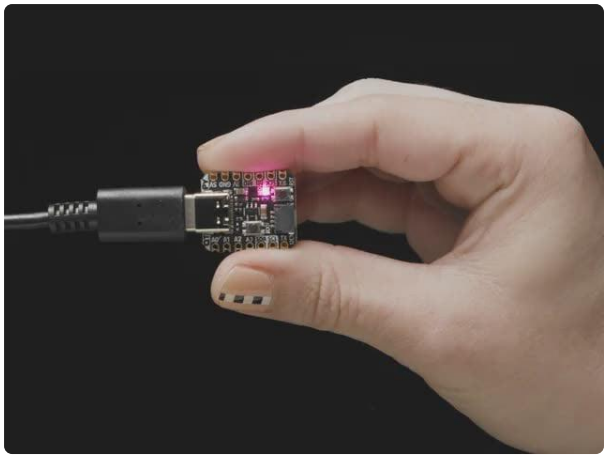
Except as otherwise noted, these examples were designed for the Raspberry Pi Pico but can be adapted to the range of boards with the RP2040 microcontroller. The QT Py RP2040 requires the addition of an external LED and current-limiting resistor for the standard LED examples.



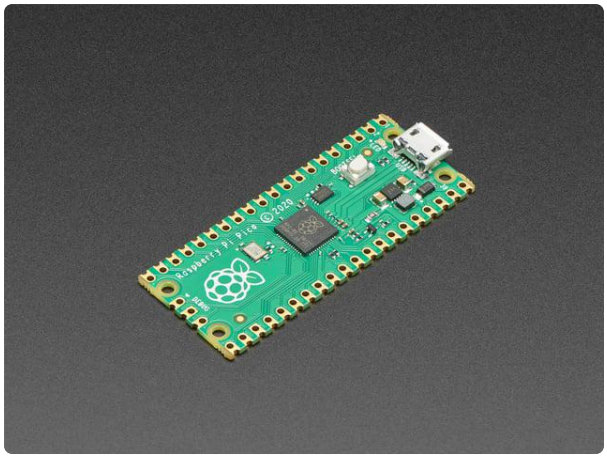
Adafruit Feather RP2040
A new chip means a new Feather, and the Raspberry Pi RP2040 is no exception. When we saw this chip we thought "this chip is going to be awesome when we give it the Feather...
https://www.adafruit.com/product/4884



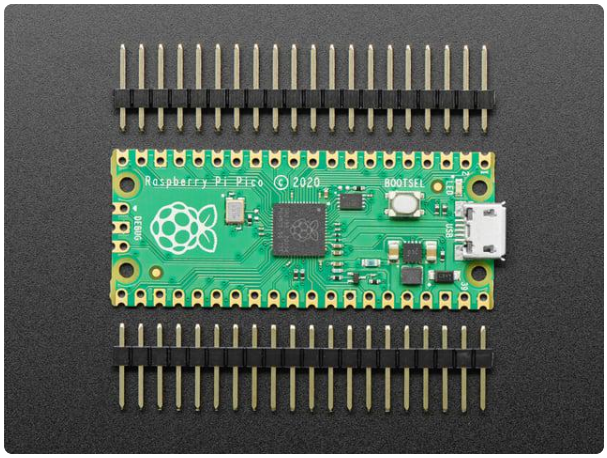
Adafruit ItsyBitsy RP2040
A new chip means a new ItsyBitsy, and the Raspberry Pi RP2040 is no exception. When we saw this chip we thought "this chip is going to be awesome when we give it the ItsyBitsy...
https://www.adafruit.com/product/4888



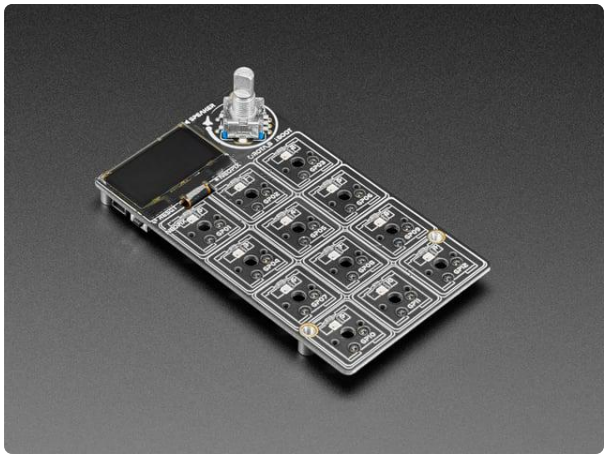
[Adafruit QT Py RP2040]
What a cutie pie! Or is it... a QT Py? This diminutive dev board comes with one of our new favorite chip, the RP2040. It's been made famous in the new
https://www.adafruit.com/product/4900



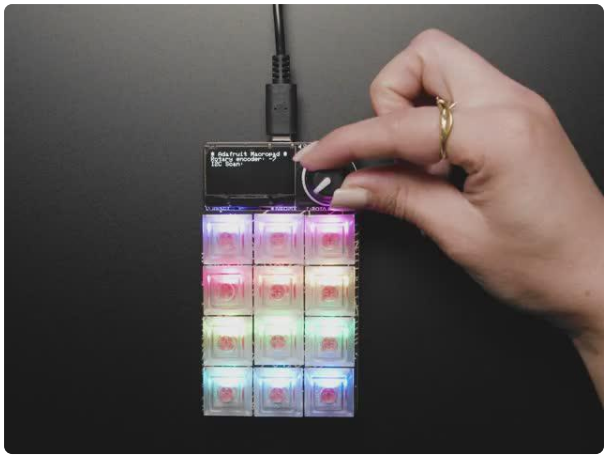
[Raspberry Pi Pico RP2040]
The Raspberry Pi foundation changed single-board computing when they released the Raspberry Pi computer, now they're ready to...
https://www.adafruit.com/product/4864



[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 MACROPAD RP2040 Bare Bones - 3x4 Keys + Encoder + OLED]
Strap yourself in, we're launching in T-minus 10 seconds...Destination? A new Class M planet called MACROPAD! M here, stands for Microcontroller because this 3x4 keyboard...
https://www.adafruit.com/product/5100



Adafruit MacroPad RP2040 Starter Kit - 3x4 Keys + Encoder + OLED

Strap yourself in, we're launching in T-minus 10 seconds...Destination? A new Class M planet called MACROPAD! M here stands for Microcontroller because this 3x4 keyboard controller...

https://www.adafruit.com/product/5128

Installing CircuitPython

CircuitPython () is a derivative of MicroPython () 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

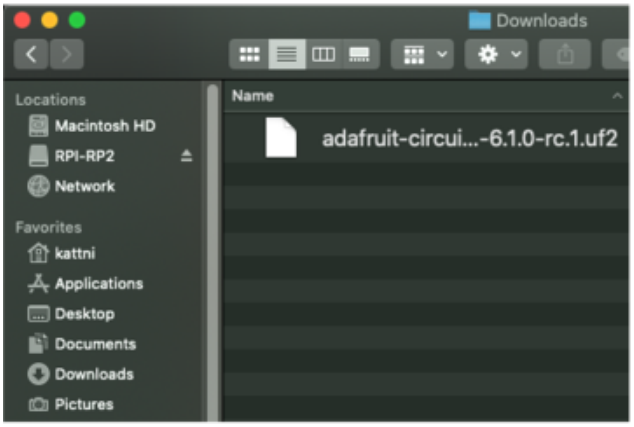
Click the link above and download the latest UF2 file.

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

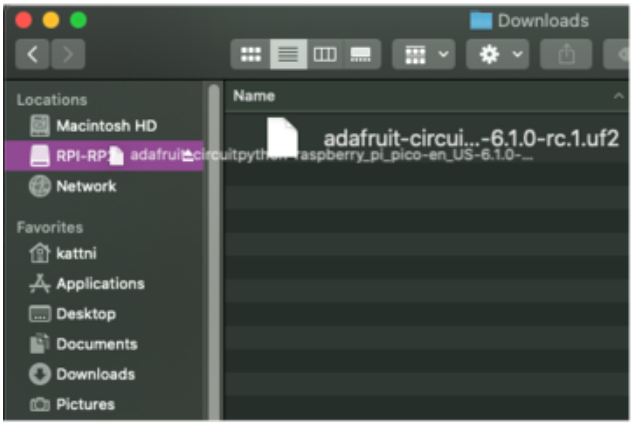
Start with your Pico unplugged from USB. Hold down the BOOTSEL button, and while continuing to hold it (don't let go!), plug the Pico into USB. Continue to hold the BOOTSEL button until the RPI-RP2 drive appears!

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

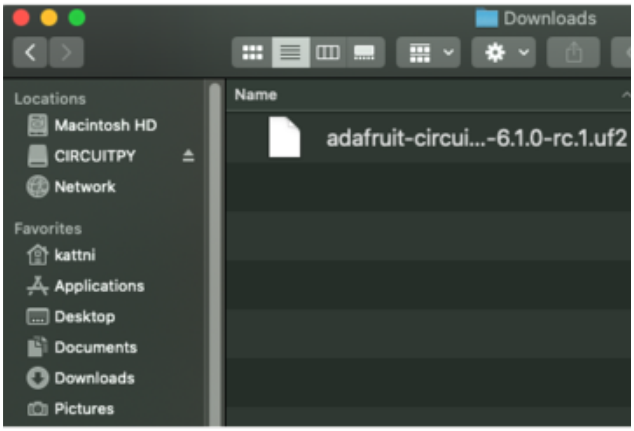
A lot of people end up using charge-only USB cables and it is very frustrating! So make sure you have a USB cable you know is good for data sync.



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

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 ().

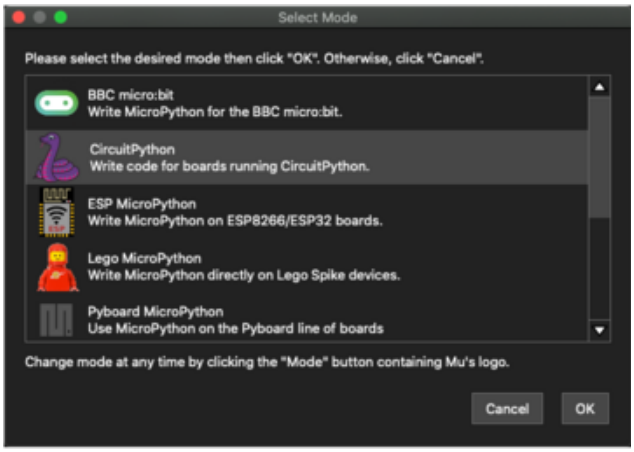
Click the Download link for downloads and installation instructions.

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.

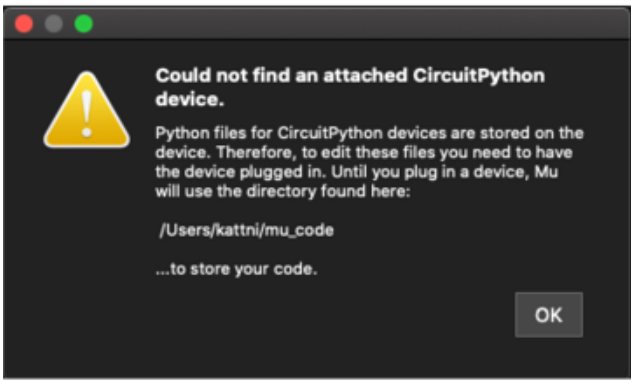
Ubuntu users: Mu currently (checked May 4, 2022) does not install properly on Ubuntu 22.04. See https://github.com/mu-editor/mu/issues to track this issue. See https://learn.adafruit.com/welcome-to-circuitpython/recommended-editors and https://learn.adafruit.com/welcome-to-circuitpython/pycharm-and-circuitpython for other editors to use.

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.

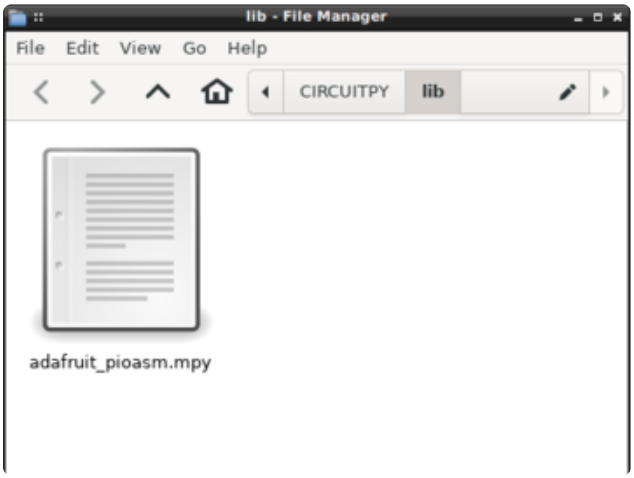
Now you're ready to code! Let's keep going...

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 () for your board.

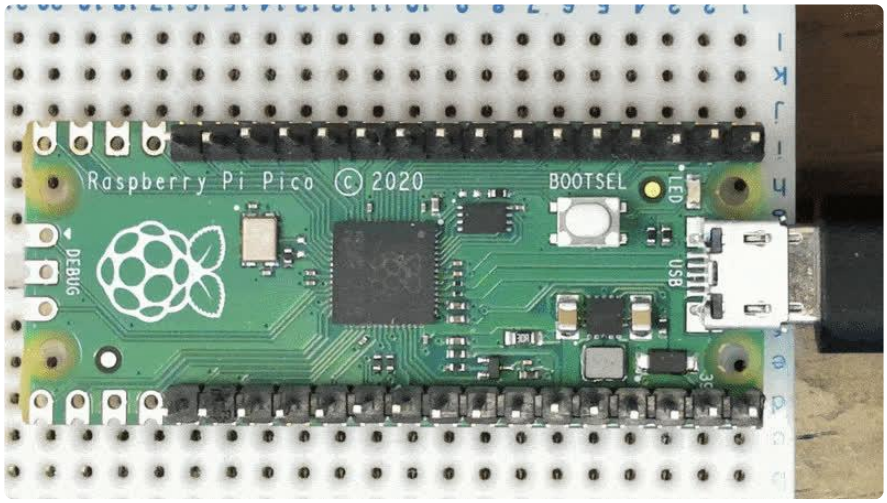
Next you'll need to install the necessary libraries to use the hardware--carefully follow the steps to find and install these libraries from Adafruit's CircuitPython library bundle (). Our CircuitPython starter guide has a great page on how to install the library bundle ().



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:

```
# SPDX-FileCopyrightText: 2021 Jeff Epler, written for Adafruit Industries
#
# SPDX-License-Identifier: MIT
#
# Adapted from the example https://github.com/raspberrypi/pico-examples/tree/master/
pio/hello_pio

import time
import board
import rp2pio
```

```
import adafruit_pioasm

hello = """
.program hello
loop:
    pull
    out pins, 1
; This program uses a 'jmp' at the end to follow the example.  However,
; in a many cases (including this one!) there is no jmp needed at the end
; and the default "wrap" behavior will automatically return to the "pull"
; instruction at the beginning.
    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 () 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:

```
hello = """
.program hello
loop:
    pull
    out pins, 1
; This program uses a 'jmp' at the end to follow the example. However,
; in a many cases (including this one!) there is no jmp needed at the end
; and the default "wrap" behavior will automatically return to the "pull"
; instruction at the beginning.
    jmp loop
"""
```

PIO programs are included within your CircuitPython source as strings, and then converted into a program with the `assemble` function.

```
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.

```
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.

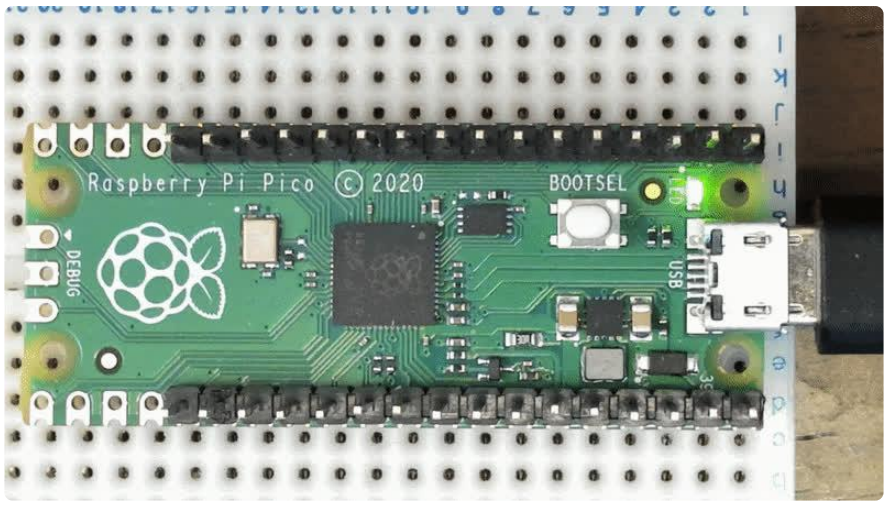
The second line contains the first instruction, `pull`, which waits until a value is sent to the State Machine (A value is sent by the `sm.write` function calls in our Python code). The value is stored in a location (register) called the OSR (Output Shift Register).

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:

```
# SPDX-FileCopyrightText: 2021 Jeff Epler, written for Adafruit Industries
#
# SPDX-License-Identifier: MIT
#
# Adapted from the an example in Appendix C of RPi_PiPico_Digital_v10.pdf

import time
import board
import rp2pio
import adafruit_pioasm

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
"""
)

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()
```

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:

```
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.

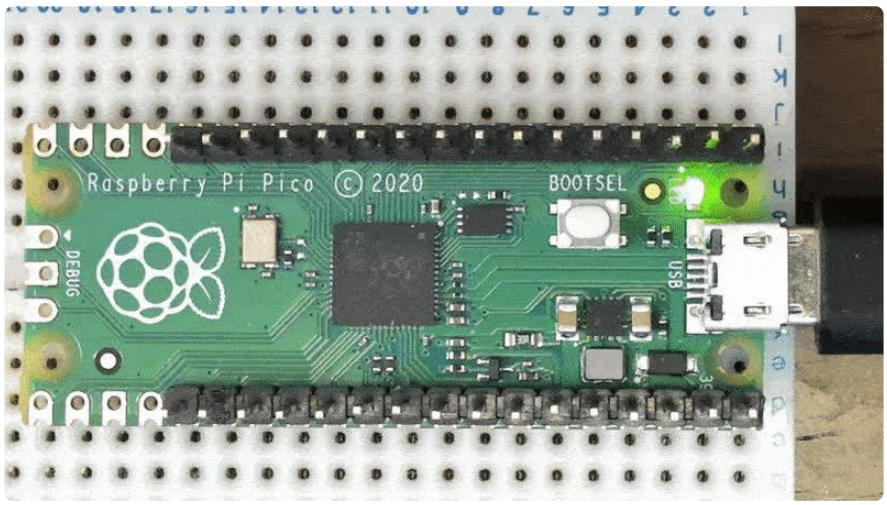
The "quarter brightness" program needs the most explanation. The new element on the first set line, `[2]`, indicates that after the set command, there is an additional delay of 2 cycles before going to the next instruction. That means that the pin is 0 for 3 cycles and 1 for 1 cycle, giving a ratio of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

```
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.

```
# SPDX-FileCopyrightText: 2021 Jeff Epler, written for Adafruit Industries
#
# SPDX-License-Identifier: MIT
#
# Adapted from the example https://github.com/raspberrypi/pico-examples/tree/master/
pio/pio_blink

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 to 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)
```

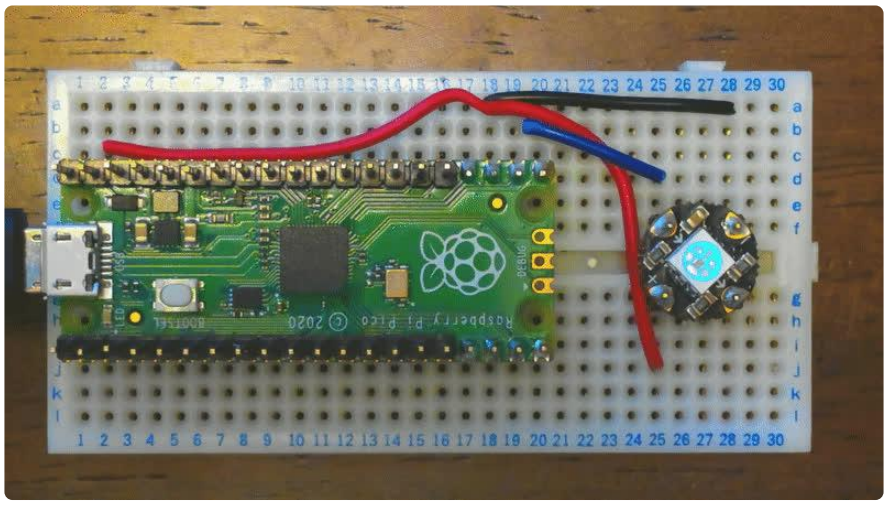
The Python forever-loop repeatedly cycles through the frequencies 5, 8, and 30.

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.

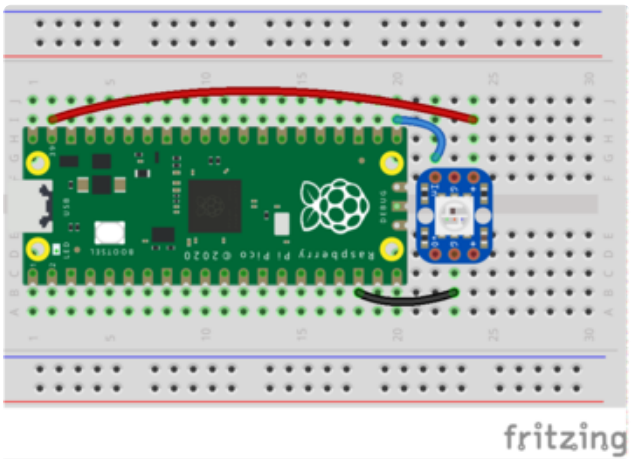
Of course, your CircuitPython program doesn't need to sleep while PIO is making the LED blink, it could be doing calculations, updating an LCD display, reading button presses. The PIO program keeps running independently of what CircuitPython is doing.

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 `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:

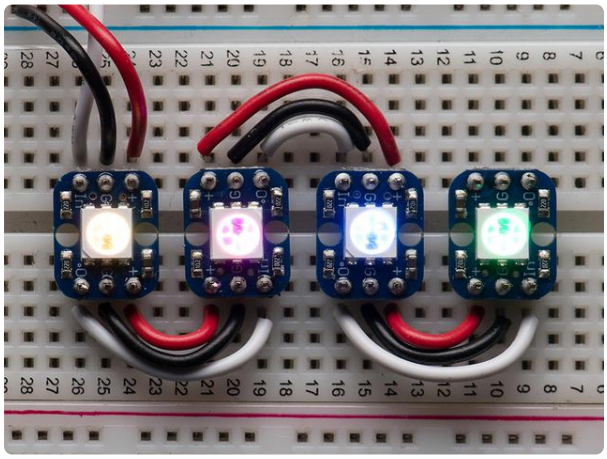


Connect Pico VSYS to NeoPixel + or VCC
Connect Pico GND to NeoPixel GND
Connect Pico GP16 to NeoPixel In or DIN
You can also select whatever pin you like and change the CircuitPython code accordingly.

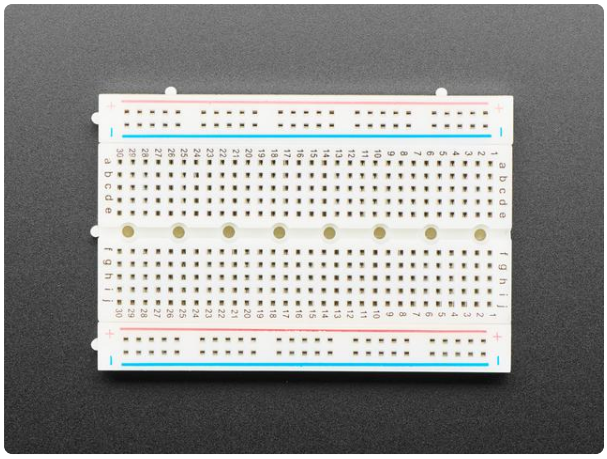
This code is tested with the RGB NeoPixel linked below. The topic of neopixel timing is actually quite complex, and this program may not work with all kinds of neopixels.

Parts

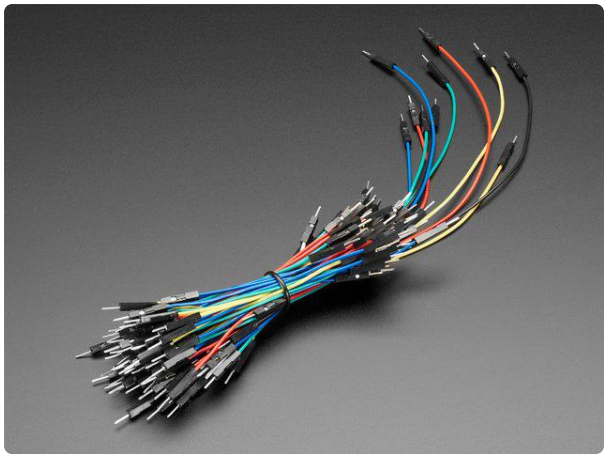
If you have the Raspberry Pi Pico or another RP2040 board which does not have a built-in NeoPixel, you'll need the parts below to follow this example. Adafruit's RP2040-based boards include a built-in NeoPixel.



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... https://www.adafruit.com/product/1312



[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...
https://www.adafruit.com/product/4539



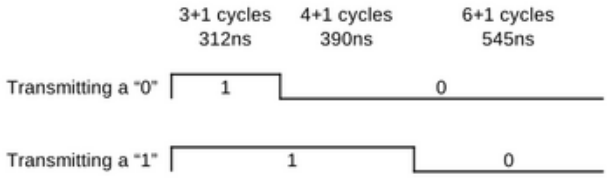
[Breadboarding wire bundle]
75 flexible stranded core wires with stiff ends molded on in red, orange, yellow, green, blue, brown, black and white. These are a major improvement over the "box of bent...
https://www.adafruit.com/product/153

Full Code Listing

```
# SPDX-FileCopyrightText: 2021 Scott Shawcroft, written for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import time
import rp2pio
import board
import microcontroller
import adafruit_pioasm

# NeoPixels are 800khz bit streams. We are choosing zeros as <312ns hi, 936 lo>
# and ones as <700 ns hi, 556 ns lo>.
# The first two instructions always run while only one of the two final
# instructions run per bit. We start with the low period because it can be
# longer while waiting for more data.
program = """
.program ws2812
.side_set 1
.wrap_target
bitloop:
   out x 1        side 0 [6]; Drive low. Side-set still takes place before
instruction stalls.
   jmp !x do_zero side 1 [3]; Branch on the bit we shifted out previous delay.
Drive high.
 do_one:
   jmp  bitloop   side 1 [4]; Continue driving high, for a one (long pulse)
 do_zero:
   nop            side 0 [4]; Or drive low, for a zero (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=12_800_000,  # to get appropriate sub-bit times in PIO program
    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 this program transmit data? Each original bit is sent in approximately 1250 microseconds. In the transmission program, 1250 microseconds are divided into three unequal portions.

To transmit a "0", the pin is set HIGH during the first portion, and then LOW during the other portions.

To transmit a "1", the pin is set HIGH during the first two porttions and low during the last portion.

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 (DOUT) pin.

This is the most compatible timing for NeoPixels that we could find, and it is believed to work with all pixels and strips sold by Adafruit. It is more compatible than the "equal thirds" timing that many sources (including older versions of this page!) describe.

A short delay (approximately 300 microseconds) without any pulses makes the pixels ready to receive fresh values.

```
program="""
.program ws2812
.side_set 1
.wrap_target
bitloop:
   out x 1        side 0 [6]; Drive low. Side-set still takes place before
instruction stalls.
   jmp !x do_zero side 1 [3]; Branch on the bit we shifted out previous delay.
Drive high.
 do_one:
   jmp  bitloop   side 1 [4]; Continue driving high, for a one (long pulse)
 do_zero:
   nop            side 0 [4]; Or drive low, for a zero (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 `side 0` next to it, the corresponding output is set LOW, and when it has `side 1` next to it, the corresponding output is set HIGH. There can be up to 5 side-set pins, in which case `side N` is interpreted as a binary number.

The first instruction, `out x 1`, transfers the next NeoPixel data bit into the x register. It also ensures the data out pin is LOW until the data bit is available. This creates the delay necessary between refreshes of the NeoPixel strip, as well as the 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 16 clocks before the execution returns to `bitloop`.

```
sm = rp2pio.StateMachine(
    assembled,
    frequency=12_800_000,  # to get appropriate sub-bit times in PIO program
    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 (16).

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 12.8MHz a slightly different value 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.

```
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 ().

For a more sophisticated example of driving 8 NeoPixel strips from just 3 GPIO pins using PIO, there's a dedicated guide ().

Advanced: Generating Morse Code with Background Writes

As of CircuitPython 7.3, it is possible to send data to one or more PIO peripherals while Python code keeps running. Depending on how the `background_write` function is called, it can send a block of data just once, or repeatedly until it's directed otherwise.

To introduce this capability, the following example shows morse code messages on the board's LED. It works on any RP2040 board with `board.LED`. You can also use it on a Raspberry Pi Pico, but you'll have to add an external LED and series resistor, then modify the code to use the right `board.GP##` connection.

```
# SPDX-FileCopyrightText: 2022 Jeff Epler, written for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""Demonstrate background writing, including loop writing, with morse code.

On any rp2040 board with board.LED, this will alternately send 'SOS' and 'TEST'
via the LED, demonstrating that Python code continues to run while the morse
code data is transmitted. Alternately, change one line below to make it send
'TEST' forever in a loop, again while Python code continues to run.

The combination of "LED status" and duration is sent to the PIO as 16-bit number:
The top bit is 1 if the LED is turned on and 0 otherwise.  The other 15 bits form a
delay
value from 1 to 32767. A subset of the morse code 'alphabit' is created, with
everthing
based on the 'DIT duration' of about 128ms (1MHz / 32 / 4000).

https://en.wikipedia.org/wiki/Morse_code
"""

import array
import time
from board import LED
from rp2pio import StateMachine
from adafruit_pioasm import Program

# This program turns the LED on or off depending on the first bit of the value,
# then delays a length of time given by the next 15 bits of the value.
# By correctly choosing the durations, a message in morse code can be sent.
pio_code = Program(
    """
        out x, 1
        mov pins, x
        out x, 15
    busy_wait:
        jmp x--, busy_wait [31]
        """
)


# The top bit of the command is the LED value, on or off
LED_ON = 0x8000
LED_OFF = 0x0000

# The other 15 bits are a delay duration.
# It must be the case that 4 * DIT_DURATION < 32768
DIT_DURATION = 4000
DAH_DURATION = 3 * DIT_DURATION

# Build up some elements of morse code, based on the wikipedia article.
DIT = array.array("H", [LED_ON | DIT_DURATION, LED_OFF | DIT_DURATION])
DAH = array.array("H", [LED_ON | DAH_DURATION, LED_OFF | DIT_DURATION])
# That is, two more DAH-length gaps for a total of three
LETTER_SPACE = array.array("H", [LED_OFF | (2 * DAH_DURATION)])
# That is, four more DAH-length gaps (after a letter space) for a total of seven
WORD_SPACE = array.array("H", [LED_OFF | (4 * DIT_DURATION)])

# Letters and words can be created by concatenating ("+") the elements
E = DIT + LETTER_SPACE
O = DAH + DAH + DAH + LETTER_SPACE
S = DIT + DIT + DIT + LETTER_SPACE
T = DAH + LETTER_SPACE
SOS = S + O + S + WORD_SPACE
TEST = T + E + S + T + WORD_SPACE

sm = StateMachine(
    pio_code.assembled,
```

```
    frequency=1_000_000,
    first_out_pin=LED,
    pull_threshold=16,
    auto_pull=True,
    out_shift_right=False,
)

# To simply repeat 'TEST' forever, change to 'if True':
if False:  # pylint: disable=using-constant-test
    print("Sending out TEST forever", end="")
    sm.background_write(loop=TEST)
    while True:
        print(end=".")
        time.sleep(0.1)

# But instead, let's alternate SOS and TEST, forever:

while True:
    for plain, morse in (
        ("SOS", SOS),
        ("TEST", TEST),
    ):
        print(f"Sending out {plain}", end="")
        sm.background_write(morse)
        sm.clear_txstall()
        while not sm.txstall:
            print(end=".")
            time.sleep(0.1)
        print()
        print("Message all sent to StateMachine (including emptying FIFO)")
        print()
```

Now to focus on some specific parts of the program. First, take a look at the PIO program itself. This program reads the new LED state first, sets it on the pin, reads a delay, then loops until the delay has been completed:

```
pio_code = Program(
    """
        out x, 1
        mov pins, x
        out x, 15
    busy_wait:
        jmp x--, busy_wait [31]
        """
)
```

Next, according to the necessary arrangement of bits, a small vocabulary in Morse code is built up:

```
# Letters and words can be created by concatenating ("+") the elements
E = DAH + LETTER_SPACE
O = DAH + DAH + DAH + LETTER_SPACE
S = DIT + DIT + DIT + LETTER_SPACE
T = DIT + LETTER_SPACE
SOS = S + O + S + WORD_SPACE
TEST = T + E + S + T + WORD_SPACE
```

Now, of course the Python code could simply write the data to the State Machine and wait for it to finish with `sm.write(SOS)`. However, by using this method, control

won't return to the Python code until all the data is handled. In the case where this is undesirable, use a background write so that the operation continues in the background while CircuitPython code continues working in the foreground:

```
sm.background_write(morse)
sm.clear_txstall()
while not sm.txstall:
    print(end=".")
    time.sleep(0.1)
print()
print("Message all sent to StateMachine (including emptying FIFO)")
print()
```

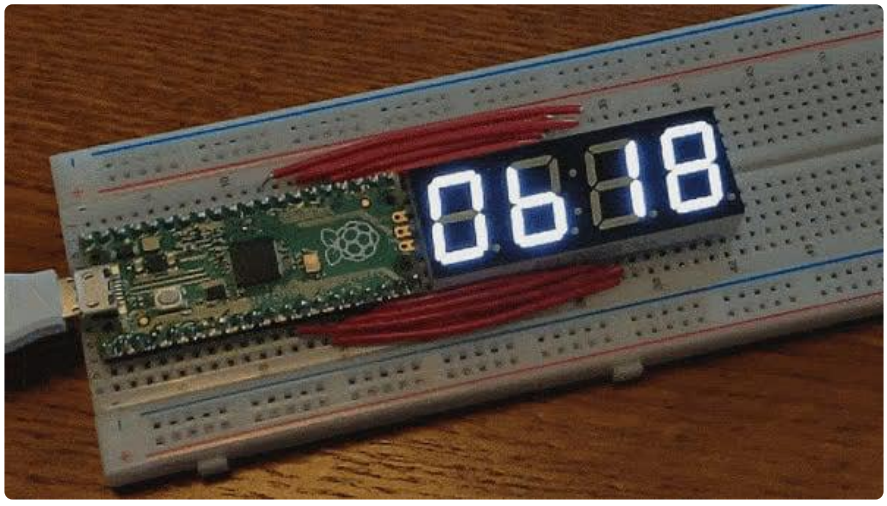
By clearing and then monitoring the `txstall` flag, the code waits for the whole message to be played before continuing.

Using the `loop` keyword argument, a single message can be made to loop forever:

```
print("Sending out TEST forever", end="")
sm.background_write(loop=TEST)
while True:
    print(end=".")
    time.sleep(0.1)
```

That's one LED, but what if you have greater ambitions? Head to the next page: Driving 7-segment displays with Background Writes ().

Advanced: Driving 7-segment displays with Background Writes



The following example shows a counter on a 4-digit 7-segment LED without a separate driver chip. Don't forget you need CircuitPython 7.3 or later.

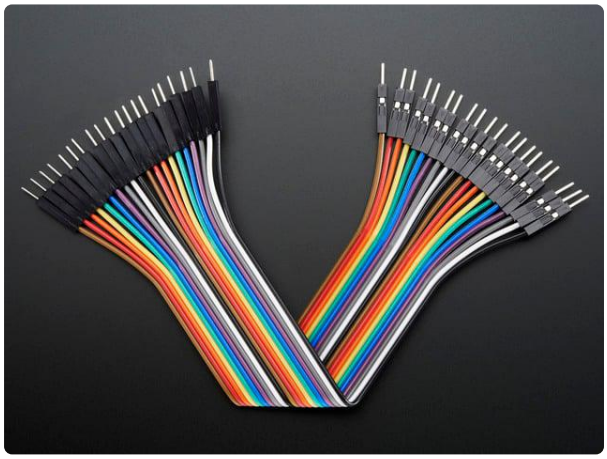
The main program just counts up forever, looping back to 0000 after 9999.

This example is designed for a Raspberry Pi Pico and bare LED display. For simplicity, it is wired without any current limiting resistors, instead relying on a combination of the RP2040's pin drive strength and the 1/4 duty cycle to limit LED current to an acceptable level, and longevity of the display was not a priority. Also, we're feeling punk today!

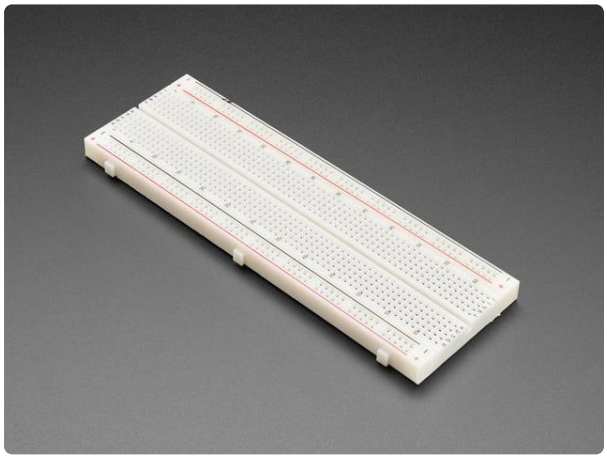
Before integrating a variant of this example code in a project, evaluate whether the design needs to add current-limiting resistors.

Parts

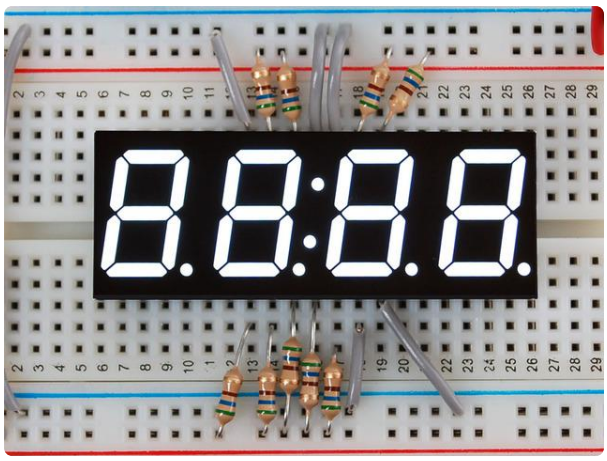
For this example, you need additional parts:



Premium Male/Male Jumper Wires - 20 x 6" (150mm)
These Male/Male Jumper Wires are handy for making wire harnesses or jumpering between headers on PCB's. These premium jumper wires are 6" (150mm) long and come in a...
https://www.adafruit.com/product/1957



Full Sized Premium Breadboard - 830 Tie Points
This is a 'full-size' premium quality breadboard, 830 tie points. Good for small and medium projects. It's 2.2" x 7" (5.5 cm x 17 cm) with a standard double-strip...
https://www.adafruit.com/product/239

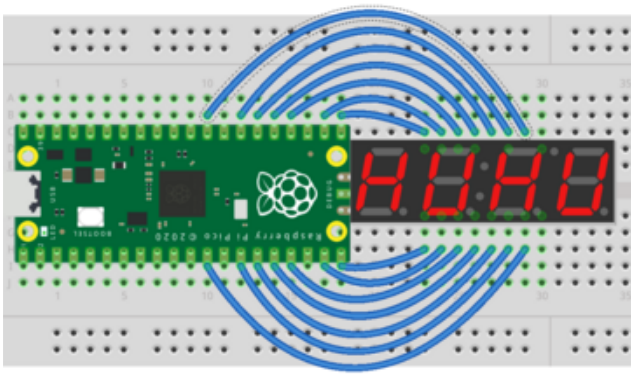


White 7-segment clock display - 0.56" digit height

Design a clock, timer or counter into your next project using our pretty 4-digit seven-segment display. These bright crisp displays are good for adding numeric output. Besides the four...

https://www.adafruit.com/product/1001

Wiring



Place the Pico and the 7-segment display on the breadboard, noting the orientation of the decimal points.

Make the following connections:

- Pico GP15 to LED matrix 1 (E SEG)
- Pico GP14 to LED matrix 2 (D SEG)
- Pico GP13 to LED matrix 3 (DP SEG)
- Pico GP12 to LED matrix 4 (C SEG)
- Pico GP11 to LED matrix 5 (G SEG)
- Pico GP10 to LED matrix 6 (COM4)
- Pico GP9 to LED matrix 7 (COLON COM)
- Pico GP22 to LED matrix 8 (COLON SEG)
- Pico GP21 to LED matrix 9 (B SEG)
- Pico GP20 to LED matrix 10 (COM3)
- Pico GP19 to LED matrix 11 (COM2)
- Pico GP18 to LED matrix 12 (F SEG)
- Pico GP17 to LED matrix 13 (A SEG)
- Pico GP16 to LED matrix 14 (COM1)

Next, load the code below on your Raspberry Pi Pico's CIRCUITPY drive:

```
# SPDX-FileCopyrightText: 2022 Jeff Epler, written for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""Drive a 7-segment display entirely from the PIO peripheral

By updating the buffer being written to the display, the shown digits can be
changed.

The main program just counts up, looping back to 0000 after 9999.

This example is designed for a Raspberry Pi Pico and bare LED display. For
simplicity, it is wired without any current limiting resistors, instead relying
on a combination of the RP2040's pin drive strength and the 1/4 duty cycle to
limit LED current to an acceptable level, and longevity of the display was not
a priority.

Before integrating a variant of this example code in a project, evaluate
whether your design needs to add current-limiting resistors.

https://www.adafruit.com/product/4864
https://www.adafruit.com/product/865

Wiring:
 * Pico GP15 to LED matrix 1 (E SEG)
 * Pico GP14 to LED matrix 2 (D SEG)
 * Pico GP13 to LED matrix 3 (DP SEG)
 * Pico GP12 to LED matrix 4 (C SEG)
 * Pico GP11 to LED matrix 5 (G SEG)
 * Pico GP10 to LED matrix 6 (COM4)
 * Pico GP9 to LED matrix 7 (COLON COM)
 * Pico GP22 to LED matrix 8 (COLON SEG)
 * Pico GP21 to LED matrix 9 (B SEG)
 * Pico GP20 to LED matrix 10 (COM3)
 * Pico GP19 to LED matrix 11 (COM2)
 * Pico GP18 to LED matrix 12 (F SEG)
 * Pico GP17 to LED matrix 13 (A SEG)
 * Pico GP16 to LED matrix 14 (COM1)
"""

import array
import time
import board
import rp2pio
import adafruit_pioasm

_program = adafruit_pioasm.Program(
    """
    out pins, 14       ; set the pins to their new state
    """
)

# Display Pins 1-7 are GP 15-9
# Display Pins 8-12 are GP 22-16
COM1_WT = 1 << 7
COM2_WT = 1 << 10
COM3_WT = 1 << 11
COM4_WT = 1 << 1
COMC_WT = 1 << 0

SEGA_WT = 1 << 8
SEGB_WT = 1 << 12
SEGC_WT = 1 << 3
SEGD_WT = 1 << 5
SEGE_WT = 1 << 6
```

```
SEGF_WT = 1 << 9
SEGG_WT = 1 << 2

SEGDP_WT = 1 << 4
SEGCOL_WT = 1 << 13

ALL_COM = COM1_WT | COM2_WT | COM3_WT | COM4_WT | COMC_WT

SEG_WT = [
    SEGA_WT,
    SEGB_WT,
    SEGC_WT,
    SEGD_WT,
    SEGE_WT,
    SEGF_WT,
    SEGG_WT,
    SEGDP_WT,
    SEGCOL_WT,
]
COM_WT = [COM1_WT, COM2_WT, COM3_WT, COM4_WT, COMC_WT]

DIGITS = [
    0b0111111,  # 0
    0b0000110,  # 1
    0b1011011,  # 2
    0b1001111,  # 3
    0b1100110,  # 4
    0b1101101,  # 5
    0b1111100,  # 6
    0b0000111,  # 7
    0b1111111,  # 8
    0b1101111,  # 9
]


def make_digit_wt(v):
    val = ALL_COM
    seg = DIGITS[v]
    for i in range(8):
        if seg & (1 << i):
            val |= SEG_WT[i]
    return val


DIGITS_WT = [make_digit_wt(i) for i in range(10)]


class SMSevenSegment:
    def __init__(self, first_pin=board.GP9):
        self._buf = array.array("H", (DIGITS_WT[0] & ~COM_WT[i] for i in range(4)))
        self._sm = rp2pio.StateMachine(
            _program.assembled,
            frequency=2000,
            first_out_pin=first_pin,
            out_pin_count=14,
            auto_pull=True,
            pull_threshold=14,
            **_program.pio_kwargs,
        )
        self._sm.background_write(loop=self._buf)

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        self.deinit()

    def deinit(self):
        self._sm.deinit()
```

```
    def __setitem__(self, i, v):
        if v is None:
            self._buf[i] = 0
        else:
            self._buf[i] = DIGITS_WT[v] & ~COM_WT[i]

    def set_number(self, number):
        for j in range(4):
            self[3 - j] = number % 10
            number //= 10


def count(start=0):
    val = start
    while True:
        yield val
        val += 1


def main():
    with SMSevenSegment(board.GP9) as s:
        for i in count():
            s.set_number(i)
            time.sleep(0.05)


if __name__ == "__main__":
    main()
```

Let's focus on some specific parts of the program. First, take a look at the PIO program itself. This program simply reads each value then sends it to the output pins:

```
_program = adafruit_pioasm.Program(
    """
    out pins, 14       ; set the pins to their new state
    """
)
```

To display a particular digit on a particular position of the display,

- Each SEG pin must be set HIGH if it is to be turned on, and LOW otherwise
- Each COM pin must be set LOW if the corresponding digit is to be turned on, and HIGH otherwise

To display a different digit in each position, the program needs to repeatedly loop through the correct pin values for each digit.

Next, some constants are built up to describe the function of each pin, and finally the list `DIGITS_WT` is created with the correct bits set to show a particular digit, 0 to 9, plus the bits for each `COM` (common) pin are turned on:

```
def make_digit_wt(v):
    val = ALL_COM
    seg = DIGITS[v]
```

```
    for i in range(8):
        if seg &amp;amp; (1 &amp;lt;&amp;lt; i):
            val |= SEG_WT[i]
    return val

DIGITS_WT = [make_digit_wt(i) for i in range(10)]
```

A `SMSevenSegment` object continually scans out the digits to the display. It operates at a clock speed of 2000Hz, and it displays each digit for 1 cycle at a time, so the refresh rate of the display is 500kHz. In almost all situations, this is enough for the digit to appear 'solid' to the human eye.

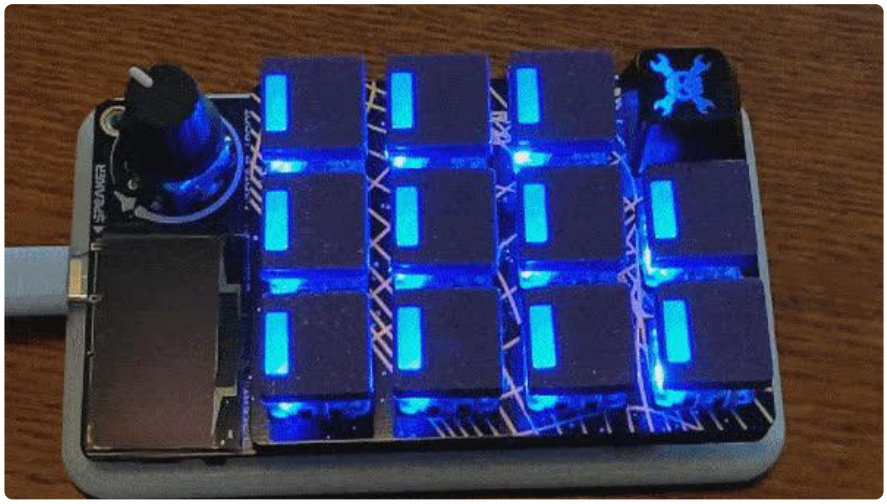
```
class SMSevenSegment:
    def __init__(self, first_pin=board.GP9):
        self._buf = array.array("H", (DIGITS_WT[0] &amp;amp; ~COM_WT[i] for i in
range(4)))
        self._sm = rp2pio.StateMachine(
            _program.assembled,
            frequency=2000,
            first_out_pin=first_pin,
            out_pin_count=14,
            auto_pull=True,
            pull_threshold=14,
            **_program.pio_kwargs,
        )
        self._sm.background_write(loop=self._buf)
```

To display a particular digit at a particular position, we take the `DIGITS_WT` for that value and turn OFF the single `COM` bit for the position in question. A 4-digit number can be displayed by putting each of its digits in one of the positions. Updating the number from right to left makes it easy to use division and modulo to compute each digit:

```
def __setitem__(self, i, v):
    if v is None:
        self._buf[i] = 0
    else:
        self._buf[i] = DIGITS_WT[v] &amp;amp; ~COM_WT[i]

def set_number(self, number):
    for j in range(4):
        self[3 - j] = number % 10
        number //= 10
```

Ready for more? Bring some color into your life with PIO and NeoPixels: writing to NeoPixels LEDS "in the background" while your CircuitPython code continues to run ().

Advanced: Using PIO to drive NeoPixels "in the background"



The following example implements a NeoPixel-compatible class that performs the actual writing of data to the LEDs "in the background", allowing your program to continue processing while LED data is being transmitted. Background PIO requires CircuitPython 7.3 or later.

It's designed as a library you can incorporate into your own program, or as a demo you can load on an Adafruit MacroPad. You can also modify the demo to work on other devices or with different numbers of NeoPixels.

The example is designed to be used as a library: Copy it to CIRCUITPY and use `from pioasm_neopixel_bg import NeoPixelBackground` to import it. If placed on a MacroPad as code.py, it shows a demo. Adapt the demo to another device by modifying the connection used (`board.NEOPIXEL`) and the number of LEDs (12) to match your setup.

```
# SPDX-FileCopyrightText: 2022 Jeff Epler, written for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""Demonstrate background writing with NeoPixels

The NeoPixelBackground class defined here is largely compatible with the
standard NeoPixel class, except that the ``show()`` method returns immediately,
writing data to the LEDs in the background, and setting `auto_write` to true
causes the data to be continuously sent to the LEDs all the time.

Writing the LED data in the background will allow more time for your
Python code to run, so it may be possible to slightly increase the refresh
rate of your LEDs or do more complicated processing.

Because the pixelbuf storage is also being written out 'live', it is possible
(even with auto-show 'false') to experience tearing, where the LEDs are a
combination of old and new values at the same time.
```

```
The demonstration code, under ``if __name__ == '__main__':`` is intended
for the Adafruit MacroPad, with 12 NeoPixel LEDs. It shows a cycling rainbow
pattern across all the LEDs.
"""

import struct
import adafruit_pixelbuf
from rp2pio import StateMachine
from adafruit_pioasm import Program

# Pixel color order constants
RGB = "RGB"
"""Red Green Blue"""
GRB = "GRB"
"""Green Red Blue"""
RGBW = "RGBW"
"""Red Green Blue White"""
GRBW = "GRBW"
"""Green Red Blue White"""

# NeoPixels are 800khz bit streams. We are choosing zeros as <312ns hi, 936 lo>
# and ones as <700 ns hi, 556 ns lo>.
_program = Program(
    """
.side_set 1 opt
.wrap_target
    pull block          side 0
    out y, 32           side 0      ; get count of NeoPixel bits

bitloop:
    pull ifempty        side 0      ; drive low
    out x 1             side 0 [5]
    jmp !x do_zero      side 1 [3]  ; drive high and branch depending on bit val
    jmp y--, bitloop    side 1 [4]  ; drive high for a one (long pulse)
    jmp end_sequence    side 0      ; sequence is over

do_zero:
    jmp y--, bitloop    side 0 [4]  ; drive low for a zero (short pulse)

end_sequence:
    pull block          side 0      ; get fresh delay value
    out y, 32           side 0      ; get delay count
wait_reset:
    jmp y--, wait_reset side 0      ; wait until delay elapses
.wrap
        """
)


class NeoPixelBackground(  # pylint: disable=too-few-public-methods
    adafruit_pixelbuf.PixelBuf
):
    def __init__(
        self, pin, n, *, bpp=3, brightness=1.0, auto_write=True, pixel_order=None
    ):
        if not pixel_order:
            pixel_order = GRB if bpp == 3 else GRBW
        elif isinstance(pixel_order, tuple):
            order_list = [RGBW[order] for order in pixel_order]
            pixel_order = "".join(order_list)

        byte_count = bpp * n
        bit_count = byte_count * 8
        padding_count = -byte_count % 4

        # backwards, so that dma byteswap corrects it!
        header = struct.pack(">L", bit_count - 1)
        trailer = b"\0" * padding_count + struct.pack(">L", 3840)
```

```
        self._sm = StateMachine(
            _program.assembled,
            auto_pull=False,
            first_sideset_pin=pin,
            out_shift_right=False,
            pull_threshold=32,
            frequency=12_800_000,
            **_program.pio_kwargs,
        )

        self._first = True
        super().__init__(
            n,
            brightness=brightness,
            byteorder=pixel_order,
            auto_write=False,
            header=header,
            trailer=trailer,
        )

        self._auto_write = False
        self._auto_writing = False
        self.auto_write = auto_write

    @property
    def auto_write(self):
        return self._auto_write

    @auto_write.setter
    def auto_write(self, value):
        self._auto_write = bool(value)
        if not value and self._auto_writing:
            self._sm.background_write()
            self._auto_writing = False
        elif value:
            self.show()

    def _transmit(self, buf):
        if self._auto_write:
            if not self._auto_writing:
                self._sm.background_write(loop=memoryview(buf).cast("L"), swap=True)
                self._auto_writing = True
        else:
            self._sm.background_write(memoryview(buf).cast("L"), swap=True)


if __name__ == "__main__":
    import board
    import rainbowio
    import supervisor

    NEOPIXEL = board.NEOPIXEL
    NUM_PIXELS = 12
    pixels = NeoPixelBackground(NEOPIXEL, NUM_PIXELS)
    while True:
        # Around 1 cycle per second
        pixels.fill(rainbowio.colorwheel(supervisor.ticks_ms() // 4))
```

Let's focus on some specific parts of the program. First, take a look at the PIO program itself. This program first fetches the number of bits of data to be sent to the neopixel. Then, while there are still bits left to send, it performs a bit transmission just like we already saw for NeoPixels in this guide (). Finally, it performs a delay loop to

ensure that multiple data transmissions have a gap between them, as required by the NeoPixel protocol.

```
_program = Program(
    """
.side_set 1 opt
.wrap_target
    pull block          side 0
    out y, 32           side 0      ; get count of NeoPixel bits

bitloop:
    pull ifempty        side 0      ; drive low
    out x 1             side 0 [5]
    jmp !x do_zero      side 1 [3]  ; drive high and branch depending on bit val
    jmp y--, bitloop    side 1 [4]  ; drive high for a one (long pulse)
    jmp end_sequence    side 0      ; sequence is over

do_zero:
    jmp y--, bitloop    side 0 [4]  ; drive low for a zero (short pulse)

end_sequence:
    pull block          side 0      ; get fresh delay value
    out y, 32           side 0      ; get delay count
wait_reset:
    jmp y--, wait_reset side 0      ; wait until delay elapses
.wrap
        """
)
```

Next, a `NeoPixel`-like class is implemented. It needs to send the bit count first, then the pixel data, and finally the delay amount; for this purpose, the header and trailer arguments for `PixelBuf` are used.

Because of how the pixel data is organized, it has to be "byte-swapped" before sending to the NeoPixels. Logic in `_transmit` also enables background writing if `auto_write` is requested, otherwise it writes just once.

```
def _transmit(self, buf):
    if self._auto_write:
        if not self._auto_writing:
            self._sm.background_write(loop=memoryview(buf).cast("L"), swap=True)
            self._auto_writing = True
    else:
        self._sm.background_write(memoryview(buf).cast("L"), swap=True)
```

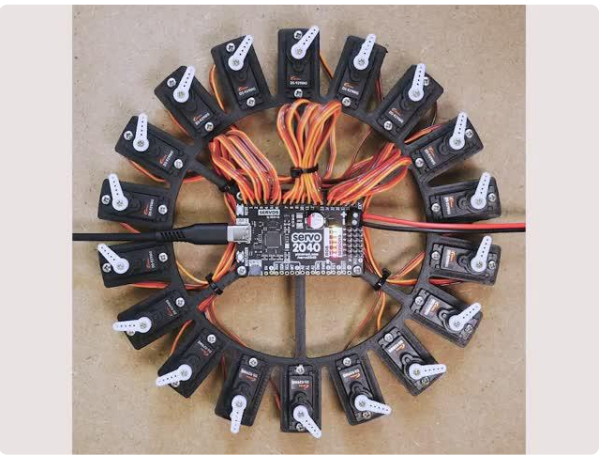
When used as a main program, it simply shows a rainbow on the configured NeoPixel LEDs:

```
NEOPIXEL = board.NEOPIXEL
NUM_PIXELS = 12
pixels = NeoPixelBackground(NEOPIXEL, NUM_PIXELS)
while True:
    # Around 1 cycle per second
    pixels.fill(rainbowio.colorwheel(supervisor.ticks_ms() // 4))
```

Finish up our tour of the background_write feature with an example for controlling a large number of RC Servo motors with a single State Machine ().

Advanced: Using PIO to control Servos with Background Writes

The `background_write` feature added in CircuitPython 7.3 is especially handy to take full advantage of the Servo 2040 board from Pimoroni: (Not familiar with servos yet? Check out this guide () to learn about servos and how to use them in CircuitPython)



Pimoroni Servo 2040 - RP2040 18 Channel Servo Controller
Build the hexapod/robot arm/other articulated contraption of your dreams with this all-in-one RP2040 powered servo controller with current measurement, sensor headers, and RGB...
https://www.adafruit.com/product/5437

This type of servo motor needs a control pulse of around 1ms to 2ms in length, repeated every 20ms. A standard PWM peripheral can control one motor, but RP2040 only has 8 PWM peripherals. This presents a challenge: How to control all 18 at once?

Pimoroni has their own library for Arduino and MicroPython. Since it's open source, the author of this guide peeked inside and saw that it could be possible in CircuitPython too, with the addition of `StateMachine.background_write`. Here's example code which will control 18 servo motors in oscillating fashion (though it is also perfectly OK to do without plugging motors into every position)

To reduce the potential for electrical damage, only plug or un-plug servo motors when the power is off!

```
# SPDX-FileCopyrightText: 2022 Jeff Epler, written for Adafruit Industries
#
# SPDX-License-Identifier: MIT
#
# Heavy inspiration from Pimoroni's "PWM Cluster":
# https://github.com/pimoroni/pimoroni-pico/blob/main/drivers/pwm/pwm_cluster.cpp
# https://github.com/pimoroni/pimoroni-pico/blob/main/drivers/pwm/pwm_cluster.pio

import array
```

```
import board
import rp2pio
import adafruit_ticks
import ulab.numpy as np
from adafruit_motor import servo


import adafruit_pioasm

_cycle_count = 3
_program = adafruit_pioasm.Program(
    """
.wrap_target
    out pins, 32            ; Immediately set the pins to their new state
    out y, 32               ; Set the counter
count_check:
    jmp y-- delay           ; Check if the counter is 0, and if so wrap around.
                            ; If not decrement the counter and jump to the delay
.wrap

delay:
    jmp count_check [1]     ; Wait a few cycles then jump back to the loop
"""
)


class PulseItem:
    def __init__(self, group, index, phase, maxval):
        self._group = group
        self._index = index
        self._phase = phase
        self._value = 0
        self._maxval = maxval
        self._turn_on = self._turn_off = None
        self._mask = 1 << index

    @property
    def frequency(self):
        return self._group.frequency

    @property
    def duty_cycle(self):
        return self._value

    @duty_cycle.setter
    def duty_cycle(self, value):
        if value < 0 or value > self._maxval:
            raise ValueError(f"value must be in the range(0, {self._maxval+1})")
        self._value = value
        self._recalculate()

    @property
    def phase(self):
        return self._phase

    @phase.setter
    def phase(self, phase):
        if phase < 0 or phase >= self._maxval:
            raise ValueError(f"phase must be in the range(0, {self._maxval})")
        self._phase = phase
        self._recalculate()

    def _recalculate(self):
        self._turn_on = self._get_turn_on()
        self._turn_off = self._get_turn_off()
        self._group._maybe_update()  # pylint: disable=protected-access

    def _get_turn_on(self):
```

```
        maxval = self._maxval
        if self._value == 0:
            return None
        if self._value == self._maxval:
            return 0
        return self.phase % maxval

    def _get_turn_off(self):
        maxval = self._maxval
        if self._value == 0:
            return None
        if self._value == self._maxval:
            return None
        return (self._value + self.phase) % maxval

    def __str__(self):
        return f"<PulseItem: {self.duty_cycle=} {self.phase=} {self._turn_on=}
{self._turn_off=}>"


class PulseGroup:
    def __init__(
        self,
        first_pin,
        pin_count,
        period=0.02,
        maxval=65535,
        stagger=False,
        auto_update=True,
    ):  # pylint: disable=too-many-arguments
        """Create a pulse group with the given characteristics"""
        self._frequency = round(1 / period)
        pio_frequency = round((1 + maxval) * _cycle_count / period)
        self._sm = rp2pio.StateMachine(
            _program.assembled,
            frequency=pio_frequency,
            first_out_pin=first_pin,
            out_pin_count=pin_count,
            auto_pull=True,
            pull_threshold=32,
            **_program.pio_kwargs,
        )
        self._auto_update = auto_update
        self._items = [
            PulseItem(self, i, round(maxval * i / pin_count) if stagger else 0,
maxval)
            for i in range(pin_count)
        ]
        self._maxval = maxval

    @property
    def frequency(self):
        return self._frequency

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        self.deinit()

    def deinit(self):
        self._sm.deinit()
        del self._items[:]

    def __getitem__(self, i):
        """Get an individual pulse generator"""
        return self._items[i]

    def __len__(self):
```

```
        return len(self._items)

    def update(self):
        changes = {0: [0, 0]}

        for i in self._items:
            turn_on = i._turn_on  # pylint: disable=protected-access
            turn_off = i._turn_off  # pylint: disable=protected-access
            mask = i._mask  # pylint: disable=protected-access

            if turn_on is not None:
                this_change = changes.get(turn_on)
                if this_change:
                    this_change[0] |= mask
                else:
                    changes[turn_on] = [mask, 0]

                # start the cycle 'on'
                if turn_off is not None and turn_off < turn_on:
                    changes[0][0] |= mask

            if turn_off is not None:
                this_change = changes.get(turn_off)
                if this_change:
                    this_change[1] |= mask
                else:
                    changes[turn_off] = [0, mask]

        def make_sequence():
            sorted_changes = sorted(changes.items())
            # Note that the first change time is always 0! Loop over range(len) is
            # to reduce allocations
            old_time = 0
            value = 0
            for time, (turn_on, turn_off) in sorted_changes:
                if time != 0:  # never occurs on the first iteration
                    yield time - old_time - 1
                old_time = time

                value = (value | turn_on) & ~turn_off
                yield value

            # the final delay value
            yield self._maxval - old_time

        buf = array.array("L", make_sequence())

        self._sm.background_write(loop=buf)

    def _maybe_update(self):
        if self._auto_update:
            self.update()

    @property
    def auto_update(self):
        return self.auto_update

    @auto_update.setter
    def auto_update(self, value):
        self.auto_update = bool(value)

    def __str__(self):
        return f"<PulseGroup({len(self)})>"


class CyclicSignal:
    def __init__(self, data, phase=0):
        self._data = data
        self._phase = 0
```

```
        self.phase = phase
        self._scale = len(self._data) - 1

    @property
    def phase(self):
        return self._phase

    @phase.setter
    def phase(self, value):
        self._phase = value % 1

    @property
    def value(self):
        idxf = self._phase * len(self._data)
        idx = int(idxf)
        frac = idxf % 1
        idx1 = (idx + 1) % len(self._data)
        val = self._data[idx]
        val1 = self._data[idx1]
        return val + (val1 - val) * frac

    def advance(self, delta):
        self._phase = (self._phase + delta) % 1


if __name__ == "__main__":
    pulsers = PulseGroup(board.SERVO_1, 18, auto_update=False)
    # Set the phase of each servo so that servo 0 starts at offset 0ms, servo 1
    # at offset 2.5ms, ...
    # For up to 8 servos, this means their duty cycles do not overlap.  Otherwise,
    # servo 9 is also at offset 0ms, etc.
    for j, p in enumerate(pulsers):
        p.phase = 8192 * (j % 8)

    servos = [servo.Servo(p) for p in pulsers]

    sine = np.sin(np.linspace(0, 2 * np.pi, 50, endpoint=False)) * 0.5 + 0.5
    print(sine)

    signals = [CyclicSignal(sine, j / len(servos)) for j in range(len(servos))]

    t0 = adafruit_ticks.ticks_ms()
    while True:
        t1 = adafruit_ticks.ticks_ms()
        for servo, signal in zip(servos, signals):
            signal.advance((t1 - t0) / 8000)
            servo.fraction = signal.value
        pulsers.update()
        print(adafruit_ticks.ticks_diff(t1, t0), "ms")
        t0 = t1
```

The key technique, which the author became aware of through Pimoroni's open source code, is to organize the PIO's data as pairs of numbers: First, 32 bits to give the new value of up to 32 output pins; Second, an additional 32 bits to give the length of time the pins should be held with this value.

The Python code simply needs to consider all the individual PWM signals in turn, and assemble a list of steps that mean something like "Turn off everything. Wait 1.5ms, then turn on output 1. Wait 1 ms, then turn off output 1 and turn on output 2. Wait 1.75ms", and so on. In order to meet the requirements of servo motors, the whole list of steps is carefully controlled to take exactly 20 milliseconds to follow. Then, Python

sends the list of steps to the PIO module to be looped "forever", or until the next list of steps is calculated and sent.

Note that the "list of steps" are sent as data to the PIO. This is distinct from the pio program, which in effect interprets the list of steps.

```
_program = adafruit_pioasm.Program(
    """
.wrap_target
    out pins, 32            ; Immediately set the pins to their new state
    out y, 32               ; Set the counter
count_check:
    jmp y-- delay           ; Check if the counter is 0, and if so wrap around.
                            ; If not decrement the counter and jump to the delay
.wrap

delay:
    jmp count_check [1]     ; Wait a few cycles then jump back to the loop
"""
)
```

To try out the code, simply copy it to a Pimoroni Servo 2040 board loaded with CircuitPython 7.3.0 or newer (or other RP2040 board, just change the code to use a different starting pin instead of `board.SERVO_1`) and hook up some RC servo motors to the appropriate headers.

The code is quite lengthy, but provides a class that can be useful in other code. A `PulseGroup` acts like a collection of PWM objects, each of which behaves similarly enough to `pulseio.PWMOut` to work with the Adafruit-CircuitPython-Motor library. Even better, by setting the `phase` of each PWM output separately, the overlap of the pulses for different motors can be minimized or eliminated, which may decrease peak current usage.

PulseGroup could be used for other tasks as well, like controlling the brightness of multiple LEDs.

There are bound to be other uses for the background write capability. Why not code one up and submit a new example on our GitHub ()?

API Documentation: adafruit_pioasm

API Documentation: adafruit_pioasm ()

API Documentation: rp2pio

API Documentation: rp2pio ()