Pico Python SDK

A MicroPython environment
for RP2040 microcontrollers

Colophon

build-date: 2021-01-12
build-version: 4c83a5a-clean

Legal Disclaimer Notice

Table of Contents

Chapter 1. The MicroPython Environment

TO DO: Talk about C vs Micro Python here? Need to mention how to use the stuff in the BootRom from Python here.

MicroPython implements the entire Python 3.4 syntax (including exceptions, `with`, `yield from`, etc., and additionally `async` `/await` keywords from Python 3.5). The following core datatypes are provided: `str` (including basic Unicode support), `bytes`, `bytearray`, `tuple`, `list`, `dict`, `set`, `frozenset`, `array.array`, `collections.namedtuple`, classes and instances. Builtin modules include `sys`, `time`, and `struct`, etc. Select ports have support for `_thread` module (multithreading). Note that only a subset of Python 3 functionality is implemented for the data types and modules.

MicroPython can execute scripts in textual source form or from precompiled bytecode, in both cases either from an on-device filesystem or "frozen" into the MicroPython executable.

1.1. Getting MicroPython for the RP2040

Pre-built Binary

A pre-built binary of the MicroPython firmware is available from the Pico Getting Started pages.

❗ **IMPORTANT**

The following instructions assume that you are using a Raspberry Pi Pico and some details may differ if you are using a different RP2040-based board. They also assume you are using Raspberry Pi OS running on a Raspberry Pi 4, or an equivalent Debian-based Linux distribution running on another platform.

If you do not want to download the pre-built release binary from the Pico Getting Started pages, and instead want to build your own binary.

Start by creating a pico directory to keep all pico related checkouts in. These instructions create a pico directory at `/home/pi/pico`.

```
$ cd ~/
$ mkdir pico
$ cd pico
```

Then clone the `micropython` git repository.

```
$ git clone -b pico git@github.com:raspberrypi/micropython.git
```

To build the Raspberry Pi Pico MicroPython port you'll also need to install some extra tools. To build projects you'll need CMake, a cross-platform tool used to build the software, and the GNU Embedded Toolchain for Arm. You can install both these via `apt` from the command line. Anything you already have installed will be ignored by `apt`.

```
$ sudo apt update
$ sudo apt install cmake gcc-arm-none-eabi
```

1.2. Building MicroPython for the RP2040

To build the port, you should change directory into `micropython` check out, and then update the Git submodules, before proceeding to build the MicroPython port.

```
$ cd micropython
$ git submodule update --init --recursive
$ make -C mpy-cross
$ cd ports/rp2
$ make
```

Amongst other targets, we have now built:

- `firmware.elf`, which is used by the debugger

- `firmware.uf2`, which can dragged onto the RP2040 USB Mass Storage Device

you can find these in the `ports/rp2/build/` directory.

1.3. Installing MicroPython on the Raspberry Pi Pico

The simplest method to load software onto a RP2040-based board is by mounting it as a USB Mass Storage Device. Doing this allows you to drag the `firmware.uf2` onto the board to program the flash. Go ahead and connect the Raspberry Pi Pico to your Raspberry Pi using a micro-USB cable, making sure that you hold down the `BOOTSEL` button to force it into USB Mass Storage Mode.

ℹ **NOTE**

If you are not following these instructions on a Raspberry Pi Pico, you may not have a `BOOTSEL` button. If this is the case, you should check if there is some other way of grounding the flash CS pin, such as a jumper, to tell RP2040 to enter the USB boot mode on boot. If there is no such method, you can load code using the Serial Wire Debug interface (see the **Getting started with Raspberry Pi Pico** book for more details).

Chapter 2. Connecting to the MicroPython REPL

The MicroPython port for Raspberry Pi Pico and other RP2040-based boards supports Serial-over-USB.

2.1. Connecting from a Raspberry Pi over USB

Once the MicroPython port has been installed, you can install `minicom`:

```
$ sudo apt install minicom
```

and then open the serial port:

```
$ minicom -b 115200 -o -D /dev/ttyACM0
```

Toggling the power to Raspberry Pi Pico you should see,

```
MicroPython v1.12-725-g315e7f50c-dirty on 2020-08-21; Raspberry Pi PICO with cortex-m0plus
Type "help()" for more information.
>>>
```

printed to the console when the Raspberry Pi Pico is first powered on.

💡 **TIP**

When in `minicom` pressing `CTRL-A` followed by `U` will add carriage returns to the serial output so that each print will end with a newline. To exit `minicom`, use `CTRL-A` followed by `X`.

ℹ️ **NOTE**

In the rare case where you can't connect to Raspberry Pi Pico you may have to reboot your Raspberry Pi.

2.2. Connecting from a Raspberry Pi using GPIO

The MicroPython port for RP2040 also provides the REPL over the Raspberry Pi Pico `UART0`, so alternatively you can connect to the REPL via this mechanism. The first thing you'll need to do is to enable UART serial on the Raspberry Pi. To do so, run `raspi-config`,

```
$ sudo raspi-config
```

and go to `Interfacing Options` → `Serial` and select "No" when asked "Would you like a login shell to be accessible over serial?" and "Yes" when asked "Would you like the serial port hardware to be enabled?" You should see something like

Figure 1. Enabling a serial UART using `raspi-config` *on the Raspberry Pi.*
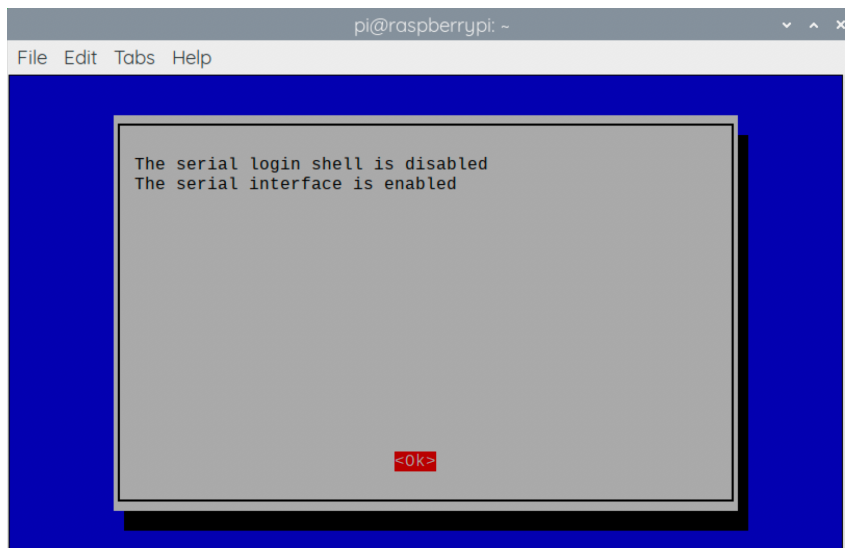
Leaving `raspi-config` you should choose "Yes" and reboot your Raspberry Pi to enable the serial port.

You should then wire the Raspberry Pi and the Raspberry Pi Pico together with the following mapping:

Raspberry Pi	Raspberry Pi Pico
GND	GND
GPIO15 (UART_RX0)	GPIO0 (UART0_TX)
GPIO14 (UART_TX0)	GPOI1 (UART0_RX)

See

Figure 2. A Raspberry Pi 4 and the Raspberry Pi Pico with UART0 connected together.
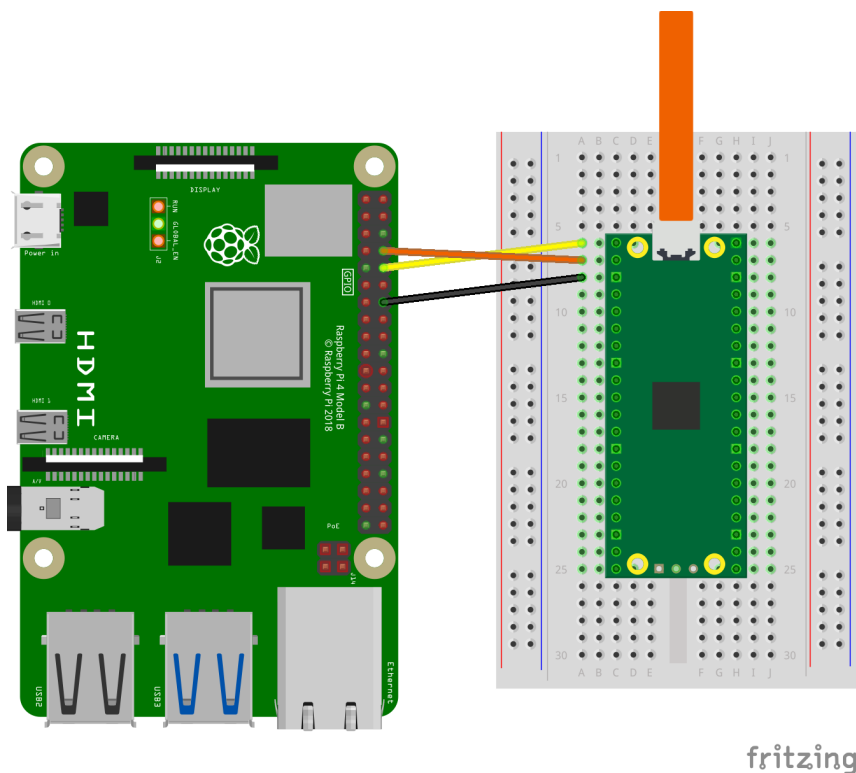
then connect to the board using `minicom` connected to `/dev/serial0`,

```
$ minicom -b 115200 -o -D /dev/serial0
```

2.3. Connecting from a Mac using USB

So long as you're using a recent version of macOS like Catalina, drivers should already be loaded. Otherwise see the manufacturers' website for FTDI Chip Drivers. Then you should use a Terminal program, e.g. Serial or similar to connect to the serial port.

ⓘ NOTE

Serial also includes driver support if needed.

The Serial-over-USB port will show up as `/dev/tty.usbmodem0000000000001`. If you're using Serial this will be shown as "Board in FS mode — CDC" in the port selector window when you open the application. Connect to this port and hit Return and you should see the REPL prompt.

2.4. Saying "Hello World" from REPL

Once connected you can check that everything is working by typing a simple "Hello World" into the REPL,

```
>>> print('hello pico!')
hello pico!
>>>>
```

Chapter 3. The RP2040 Port

Currently supported features include:

- REPL over USB and UART (on GP0/GP1).

- 128kB filesystem using `littlefs2` on the internal flash.

- `utime` module with sleep and ticks functions.

- `ubinascii` modile.

- `machine` module with some basic functions.

    - `machine.Pin` class.

    - `machine.Timer` class.

    - `machine.ADC` class.

    - `machine.I2C` and `machine.SoftI2C` classes.

    - `machine.SPI` and `machine.SoftSPI` classes.

    - `machine.WDT` class.

    - `machine.PWM` class.

    - `machine.UART` class.

- `rp2` specific module.

    - Support for PIO.

- Multicore support exposed via the standard `_thread` module

Documentation around MicroPython is available from https://docs.micropython.org.

<mark>TO DO: Talk about the chip-specific features here</mark>

3.1. Blinking an LED in MicroPython

The on-board LED on the Raspberry Pi Pico is connected to GPIO pin 25. You can blink this on and off from the REPL. When you see the REPL prompt enter the following,

```
>>> from machine import Pin
>>> led = Pin(25, Pin.OUT)
```

then you can turn the LED on with,

```
>>> led.value(1)
```

and off again with,

```
>>> led.value(0)
```

Alternatively you can use a timer to blink the on-board LED.

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/blink/blink.py Lines 1 - 9

```
1 from machine import Pin, Timer
2
3 led = Pin(25, Pin.OUT)
4 tim = Timer()
5 def tick(timer):
6     global led
7     led.toggle()
8
9 tim.init(freq=2.5, mode=Timer.PERIODIC, callback=tick)
```

3.2. UART

USB serial is available from MicroPython, but the REPL is also available over UART0 by default. The default setting for UART are taken from the C SDK.

Table 1. Default UART

Function	Default
UART_BAUDRATE	115,200
UART_BITS	8
UART_STOP	1
UART0_TX	Pin 0
UART0_RX	Pin 1
UART1_TX	Pin 4
UART1_RX	Pin 5

3.3. Interrupts

You can set an IRQ like this:

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/irq/irq.py Lines 1 - 5

```
1 from machine import Pin
2
3 p2 = Pin(2, Pin.IN, Pin.PULL_UP)
4 p2.irq(lambda pin: print("IRQ with flags:", pin.irq().flags()),
5 Pin.IRQ_FALLING)
```

It should print out something when GP2 has a falling edge.

3.4. Multicore Support

Example usage:

```
 1 import time, _thread, machine
 2
 3 def task(n, delay):
 4     led = machine.Pin(25, machine.Pin.OUT)
 5     for i in range(n):
 6         led.high()
 7         time.sleep(delay)
 8         led.low()
 9         time.sleep(delay)
10     print('done')
11
12 _thread.start_new_thread(task, (10, 0.5))
```

Only one thread can be started/running at any one time, because there is no `RTOS` just a second core. The `GIL` is not enabled so both `core0` and `core1` can run Python code concurrently, with care to use locks for shared data.

3.5. I2C

Example usage:

```
 1 from machine import Pin, I2C
 2
 3 i2c = I2C(0, scl=Pin(9), sda=Pin(8), freq=100000)
 4 i2c.scan()
 5 i2c.writeto(76, b'123')
 6 i2c.readfrom(76, 4)
 7
 8 i2c = I2C(1, scl=Pin(7), sda=Pin(6), freq=100000)
 9 i2c.scan()
10 i2c.writeto_mem(76, 6, b'456')
11 i2c.readfrom_mem(76, 6, 4)
```

I2C can be constructed without specifying the frequency, if you just want all the defaults.

```
 1 from machine import I2C
 2
 3 i2c = I2C(0)  # defaults to SCL=Pin(9), SDA=Pin(8), freq=400000
```

🚫 **WARNING**

> There may be some bugs reading/writing to device addresses that do not respond, the hardware seems to lock up in some cases.

Table 2. Default I2C pins

Function	Default
I2C Frequency	400,000
I2C0 SCL	Pin 9

I2C0 SDA	Pin 8
I2C1 SCL	Pin 7
I2C1 SDA	Pin 6

3.6. SPI

Example usage:

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/sqi/sqi.py Lines 1 - 11

```
 1 from machine import SPI
 2
 3 spi = SPI(0)
 4 spi = SPI(0, 100_000)
 5 spi = SPI(0, 100_000, polarity=1, phase=1)
 6
 7 spi.write('test')
 8 spi.read(5)
 9
10 buf = bytearray(3)
11 spi.write_readinto('out', buf)
```

> **ℹ NOTE**
>
> The chip select must be managed separately using a `machine.Pin`.

Table 3. Default SPI pins

Function	Default
SPI_BAUDRATE	1,000,000
SPI_POLARITY	0
SPI_PHASE	0
SPI_BITS	8
SPI_FIRSTBIT	MSB
SPI0_SCK	Pin 6
SPI0_MOSI	Pin 7
SPI0_MISO	Pin 4
SPI1_SCK	Pin 10
SPI1_MOSI	Pin 11
SPI1_MISO	Pin 8

3.7. PWM

Example of using PWM to fade an LED:

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/pwm/pwm_fade.py Lines 1 - 25

```
1  # Example using PWM to fade an LED.
2
3  import time
4  from machine import Pin, PWM
5
6
7  # Construct PWM object, with LED on Pin(25).
8  pwm = PWM(Pin(25))
9
10 # Set the PWM frequency.
11 pwm.freq(1000)
12
13 # Fade the LED in and out a few times.
14 duty = 0
15 direction = 1
16 for _ in range(8 * 256):
17     duty += direction
18     if duty > 255:
19         duty = 255
20         direction = -1
21     elif duty < 0:
22         duty = 0
23         direction = 1
24     pwm.duty_u16(duty * duty)
25     time.sleep(0.001)
```

3.8. PIO Support

The current status of PIO support

The current development status of PIO support can be found in this Github issue. Support for PIO is preliminary and may be unstable.

Current support allows you to define Programmable IO (PIO) Assembler blocks and using them in the PIO peripheral, more documentation around PIO can be found in Chapter 3 of the **RP2040 Datasheet** and Chapter 4 of the **Pico C/C++ SDK** book.

The Raspberry Pi Pico MicroPython por a new `@rp2.asm_pio` decorator, along with a `rp2.PIO` class. The definition of a PIO program, and the configuration of the state machine, into 2 logical parts:

- The program definition, including how many pins are used and if they are in/out pins. This goes in the `@rp2.asm_pio` definition. This is close to what the `pioasm` tool from the Pico SDK would generate from a `.pio` file (but here it's all defined in Python).

- The program instantiation, which sets the frequency of the state machine and which pins to bind to. These get set when setting a SM to run a particular program.

The aim was to allow a program to be defined once and then easily instantiated multiple times (if needed) with different GPIO. Another aim was to make it easy to basic things without getting weighed down in too much PIO/SM configuration.

Example usage, to blink the on-board LED connected to GPIO 25,

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/pio/pio_blink.py Lines 1 - 28

```
1  import time
```

```
 2 from rp2 import PIO, asm_pio
 3 from machine import Pin
 4
 5 # Define the blink program.  It has one GPIO to bind to on the set instruction, which is an
   output pin.
 6 # Use lots of delays to make the blinking visible by eye.
 7 @asm_pio(set_init=rp2.PIO.OUT_LOW)
 8 def blink():
 9     wrap_target()
10     set(pins, 1)    [31]
11     nop()           [31]
12     nop()           [31]
13     nop()           [31]
14     nop()           [31]
15     set(pins, 0)    [31]
16     nop()           [31]
17     nop()           [31]
18     nop()           [31]
19     nop()           [31]
20     wrap()
21
22 # Instantiate a state machine with the blink program, at 1000Hz, with set bound to Pin(25)
   (LED on the rp2 board)
23 sm = rp2.StateMachine(0, blink, freq=1000, set_base=Pin(25))
24
25 # Run the state machine for 3 seconds.  The LED should blink.
26 sm.active(1)
27 time.sleep(3)
28 sm.active(0)
```

or via explicit exec.

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/pio/pio_exec.py Lines 1 - 27

```
 1 # Example using PIO to turn on an LED via an explicit exec.
 2 #
 3 # Demonstrates:
 4 #   - using set_init and set_base
 5 #   - using StateMachine.exec
 6
 7 import time
 8 from machine import Pin
 9 import rp2
10
11 # Define an empty program that uses a single set pin.
12 @rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
13 def prog():
14     pass
15
16
17 # Construct the StateMachine, binding Pin(25) to the set pin.
18 sm = rp2.StateMachine(0, prog, set_base=Pin(25))
19
20 # Turn on the set pin via an exec instruction.
21 sm.exec("set(pins, 1)")
22
23 # Sleep for 500ms.
24 time.sleep(0.5)
25
26 # Turn off the set pin via an exec instruction.
27 sm.exec("set(pins, 0)")
```

Some points to note,

- All program configuration (eg autopull) is done in the `@asm_pio` decorator. Only the frequency and base pins are set in the StateMachine constructor.

- `[n]` is used for delay, `.set(n)` used for sideset

- The assembler will automatically detect if sideset is used everywhere or only on a few instructions, and set the `SIDE_EN` bit automatically

The idea is that for the 4 sets of pins (`in`, `out`, `set`, `sideset`, excluding `jmp`) that can be connected to a state machine, there's the following that need configuring for each set:

1. base GPIO

2. number of consecutive GPIO

3. initial GPIO direction (in or out pin)

4. initial GPIO value (high or low)

In the design of the Python API for PIO these 4 items are split into "declaration" (items 2-4) and "instantiation" (item 1). In other words, a program is written with items 2-4 fixed for that program (eg a WS2812 driver would have 1 output pin) and item 1 is free to change without changing the program (eg which pin the WS2812 is connected to).

So in the `@asm_pio` decorator you declare items 2-4, and in the `StateMachine` constructor you say which base pin to use (item 1). That makes it easy to define a single program and instantiate it multiple times on different pins (you can't really change items 2-4 for a different instantiation of the same program, it doesn't really make sense to do that).

And the same keyword arg (in the case about it's `sideset_pins`) is used for both the declaration and instantiation, to show that they are linked.

To declare multiple pins in the decorator (the count, ie item 2 above), you use a tuple/list of values. And each item in the tuple/list specified items 3 and 4. For example:

```
1 @asm_pio(set_pins=(PIO.OUT_LOW, PIO.OUT_HIGH, PIO.IN_LOW), sideset_pins=PIO.OUT_LOW)
2 def foo():
3     ....
4
5 sm = StateMachine(0, foo, freq=10000, set_pins=Pin(15), sideset_pins=Pin(22))
```

In this example:

- there are 3 set pins connected to the SM, and their initial state (set when the StateMachine is created) is: output low, output high, input low (used for open-drain)

- there is 1 sideset pin, initial state is output low

- the 3 set pins start at Pin(15)

- the 1 sideset pin starts at Pin(22)

The reason to have the constants `OUT_LOW`, `OUT_HIGH`, `IN_LOW` and `IN_HIGH` is so that the pin value and dir are automatically set before the start of the PIO program (instead of wasting instruction words to do `set(pindirs, 1)` etc at the start).

3.8.1. IRQ

There is support for PIO IRQs, e.g.

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/pio/pio_irq.py Lines 1 - 25

```
1 import time
2 import rp2
```

```
 3
 4 @rp2.asm_pio()
 5 def irq_test():
 6     wrap_target()
 7     nop()          [31]
 8     nop()          [31]
 9     nop()          [31]
10     nop()          [31]
11     irq(0)
12     nop()          [31]
13     nop()          [31]
14     nop()          [31]
15     nop()          [31]
16     irq(1)
17     wrap()
18
19
20 rp2.PIO(0).irq(lambda pio: print(pio.irq().flags()))
21
22 sm = rp2.StateMachine(0, irq_test, freq=1000)
23 sm.active(1)
24 time.sleep(1)
25 sm.active(0)
```

An example program that blinks at 1Hz and raises an IRQ at 1Hz to print the current millisecond timestamp,

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/pio/pio_1hz.py Lines 1 - 33

```
 1 # Example using PIO to blink an LED and raise an IRQ at 1Hz.
 2
 3 import time
 4 from machine import Pin
 5 import rp2
 6
 7
 8 @rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
 9 def blink_1hz():
10     # Cycles: 1 + 1 + 6 + 32 * (30 + 1) = 1000
11     irq(rel(0))
12     set(pins, 1)
13     set(x, 31)                  [5]
14     label("delay_high")
15     nop()                       [29]
16     jmp(x_dec, "delay_high")
17
18     # Cycles: 1 + 7 + 32 * (30 + 1) = 1000
19     set(pins, 0)
20     set(x, 31)                  [6]
21     label("delay_low")
22     nop()                       [29]
23     jmp(x_dec, "delay_low")
24
25
26 # Create the StateMachine with the blink_1hz program, outputting on Pin(25).
27 sm = rp2.StateMachine(0, blink_1hz, freq=2000, set_base=Pin(25))
28
29 # Set the IRQ handler to print the millisecond timestamp.
30 sm.irq(lambda p: print(time.ticks_ms()))
31
32 # Start the StateMachine.
33 sm.active(1)
```

or to wait for a pin change and raise an IRQ.

```python
1  # Example using PIO to wait for a pin change and raise an IRQ.
2  #
3  # Demonstrates:
4  #   - PIO wrapping
5  #   - PIO wait instruction, waiting on an input pin
6  #   - PIO irq instruction, in blocking mode with relative IRQ number
7  #   - setting the in_base pin for a StateMachine
8  #   - setting an irq handler for a StateMachine
9  #   - instantiating 2x StateMachine's with the same program and different pins
10
11 import time
12 from machine import Pin
13 import rp2
14
15
16 @rp2.asm_pio()
17 def wait_pin_low():
18     wrap_target()
19
20     wait(0, pin, 0)
21     irq(block, rel(0))
22     wait(1, pin, 0)
23
24     wrap()
25
26
27 def handler(sm):
28     # Print a (wrapping) timestamp, and the state machine object.
29     print(time.ticks_ms(), sm)
30
31
32 # Instantiate StateMachine(0) with wait_pin_low program on Pin(16).
33 pin16 = Pin(16, Pin.IN, Pin.PULL_UP)
34 sm0 = rp2.StateMachine(0, wait_pin_low, in_base=pin16)
35 sm0.irq(handler)
36
37 # Instantiate StateMachine(1) with wait_pin_low program on Pin(17).
38 pin17 = Pin(17, Pin.IN, Pin.PULL_UP)
39 sm1 = rp2.StateMachine(1, wait_pin_low, in_base=pin17)
40 sm1.irq(handler)
41
42 # Start the StateMachine's running.
43 sm0.active(1)
44 sm1.active(1)
45
46 # Now, when Pin(16) or Pin(17) is pulled low a message will be printed to the REPL.
```

3.8.2. WS2812 LED (NeoPixel)

While a WS2812 LED (NeoPixel) can be driven via the following program,

```python
1  # Example using PIO to drive a set of WS2812 LEDs.
2
3  import array, time
```

```
 4 from machine import Pin
 5 import rp2
 6
 7 # Configure the number of WS2812 LEDs.
 8 NUM_LEDS = 8
 9
10
11 @rp2.asm_pio(sideset_init=rp2.PIO.OUT_LOW, out_shiftdir=rp2.PIO.SHIFT_LEFT, autopull=True,
   pull_thresh=24)
12 def ws2812():
13     T1 = 2
14     T2 = 5
15     T3 = 3
16     wrap_target()
17     label("bitloop")
18     out(x, 1)               .side(0)    [T3 - 1]
19     jmp(not_x, "do_zero")   .side(1)    [T1 - 1]
20     jmp("bitloop")          .side(1)    [T2 - 1]
21     label("do_zero")
22     nop()                   .side(0)    [T2 - 1]
23     wrap()
24
25
26 # Create the StateMachine with the ws2812 program, outputting on Pin(22).
27 sm = rp2.StateMachine(0, ws2812, freq=8_000_000, sideset_base=Pin(22))
28
29 # Start the StateMachine, it will wait for data on its FIFO.
30 sm.active(1)
31
32 # Display a pattern on the LEDs via an array of LED RGB values.
33 ar = array.array("I", [0 for _ in range(NUM_LEDS)])
34
35 # Cycle colours.
36 for i in range(4 * NUM_LEDS):
37     for j in range(NUM_LEDS):
38         r = j * 100 // (NUM_LEDS - 1)
39         b = 100 - j * 100 // (NUM_LEDS - 1)
40         if j != i % NUM_LEDS:
41             r >>= 3
42             b >>= 3
43         ar[j] = r << 16 | b
44     sm.put(ar, 8)
45     time.sleep_ms(50)
46
47 # Fade out.
48 for i in range(24):
49     for j in range(NUM_LEDS):
50         ar[j] >>= 1
51     sm.put(ar, 8)
52     time.sleep_ms(50)
```

3.8.3. UART TX

A UART TX example,

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/pio/pio_uart_tx.py Lines 1 - 42

```
1 # Example using PIO to create a UART TX interface
2
3 from machine import Pin
```

```
 4 from rp2 import PIO, StateMachine, asm_pio
 5
 6 UART_BAUD = 115200
 7 PIN_BASE = 10
 8 NUM_UARTS = 8
 9
10
11 @asm_pio(sideset_init=PIO.OUT_HIGH, out_init=PIO.OUT_HIGH, out_shiftdir=PIO.SHIFT_RIGHT)
12 def uart_tx():
13     # Block with TX deasserted until data available
14     pull()
15     # Initialise bit counter, assert start bit for 8 cycles
16     set(x, 7)   .side(0)        [7]
17     # Shift out 8 data bits, 8 execution cycles per bit
18     label("bitloop")
19     out(pins, 1)               [6]
20     jmp(x_dec, "bitloop")
21     # Assert stop bit for 8 cycles total (incl 1 for pull())
22     nop()       .side(1)        [6]
23
24
25 # Now we add 8 UART TXs, on pins 10 to 17. Use the same baud rate for all of them.
26 uarts = []
27 for i in range(NUM_UARTS):
28     sm = StateMachine(
29         i, uart_tx, freq=8 * UART_BAUD, sideset_base=Pin(PIN_BASE + i), out_base=Pin
   (PIN_BASE + i)
30     )
31     sm.active(1)
32     uarts.append(sm)
33
34 # We can print characters from each UART by pushing them to the TX FIFO
35 def pio_uart_print(sm, s):
36     for c in s:
37         sm.put(ord(c))
38
39
40 # Print a different message from each UART
41 for i, u in enumerate(uarts):
42     pio_uart_print(u, "Hello from UART {}!\n".format(i))
```

ℹ **NOTE**

You need to specify an intial OUT pin state in your program in order to be able to pass OUT mapping to your SM instantiation, even though in this program it is redundant because the mappings overlap.

3.8.4. SPI

An SPI example.

TO DO: 8 bit only for now because we need to set pullthresh per **program** not per SM

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/pio/pio_spi.py Lines 1 - 48

```
 1 from machine import Pin
 2
 3 @rp2.asm_pio(out_shiftdir=0, autopull=True, pull_thresh=8, autopush=True, push_thresh=8,
   sideset_init=(rp2.PIO.OUT_LOW, rp2.PIO.OUT_HIGH), out_init=rp2.PIO.OUT_LOW)
 4 def spi_cpha0():
```

```
 5      # Note X must be preinitialised by setup code before first byte, we reload after sending
    each byte
 6      # Would normally do this via exec() but in this case it's in the instruction memory and is
    only run once
 7      set(x, 6)
 8      # Actual program body follows
 9      wrap_target()
10      pull(ifempty)           .side(0x2)   [1]
11      label("bitloop")
12      out(pins, 1)            .side(0x0)   [1]
13      in_(pins, 1)            .side(0x1)
14      jmp(x_dec, "bitloop")   .side(0x1)
15
16      out(pins, 1)            .side(0x0)
17      set(x, 6)               .side(0x0) # Note this could be replaced with mov x, y for
    programmable frame size
18      in_(pins, 1)            .side(0x1)
19      jmp(not_osre, "bitloop") .side(0x1) # Fallthru if TXF empties
20
21      nop()                   .side(0x0)   [1] # CSn back porch
22      wrap()
23
24
25 class PIOSPI:
26
27     def __init__(self, sm_id, pin_mosi, pin_miso, pin_sck, cpha=False, cpol=False, freq
    =1000000):
28         assert(not(cpol or cpha))
29         self._sm = rp2.StateMachine(sm_id, spi_cpha0, freq=4*freq, sideset_base=Pin(
    pin_sck), out_base=Pin(pin_mosi), in_base=Pin(pin_sck))
30         self._sm.active(1)
31
32     # Note this code will die spectacularly cause we're not draining the RX FIFO
33     def write_blocking(wdata):
34         for b in wdata:
35             self._sm.put(b << 24)
36
37     def read_blocking(n):
38         data = []
39         for i in range(n):
40             data.append(self._sm.get() & 0xff)
41         return data
42
43     def write_read_blocking(wdata):
44         rdata = []
45         for b in wdata:
46             self._sm.put(b << 24)
47             rdata.append(self._sm.get() & 0xff)
48         return rdata
```

ⓘ **NOTE**

This SPI program supports programmable frame sizes (by holding the reload value for X counter in the Y register) but currently this can't be used, because the autopull threshold is associated with the program, instead of the SM instantiation.

3.8.5. PWM

A PWM example,

Pico MicroPython Examples: *https://github.com/raspberrypi/pico-micropython-examples/tree/master/pio/pio_pwm.py* *Lines 1 - 43*

```python
1  # Example of using PIO for PWM, and fading the brightness of an LED
2
3  from machine import Pin
4  from rp2 import PIO, StateMachine, asm_pio
5  from time import sleep
6
7
8  @asm_pio(sideset_init=PIO.OUT_LOW)
9  def pwm_prog():
10     pull(noblock) .side(0)
11     mov(x, osr) # Keep most recent pull data stashed in X, for recycling by noblock
12     mov(y, isr) # ISR must be preloaded with PWM count max
13     label("pwmloop")
14     jmp(x_not_y, "skip")
15     nop()          .side(1)
16     label("skip")
17     jmp(y_dec, "pwmloop")
18
19
20 class PIOPWM:
21     def __init__(self, sm_id, pin, max_count, count_freq):
22         self._sm = StateMachine(sm_id, pwm_prog, freq=2 * count_freq, sideset_base=Pin(pin))
23         # Use exec() to load max count into ISR
24         self._sm.put(max_count)
25         self._sm.exec("pull()")
26         self._sm.exec("mov(isr, osr)")
27         self._sm.active(1)
28         self._max_count = max_count
29
30     def set(self, value):
31         # Minimum value is -1 (completely turn off), 0 actually still produces narrow pulse
32         value = max(value, -1)
33         value = min(value, self._max_count)
34         self._sm.put(value)
35
36
37 # Pin 25 is LED on Pico boards
38 pwm = PIOPWM(0, 25, max_count=(1 << 16) - 1, count_freq=10_000_000)
39
40 while True:
41     for i in range(256):
42         pwm.set(i ** 2)
43         sleep(0.01)
```

3.8.6. Using `pioasm`

As well as writing PIO code inline in your MicroPython script you can use the `pioasm` tool from the C/C++ SDK to generate a Python file.

```
$ pioasm -o python input (output)
```

For more information on `pioasm` see the **Pico C/C++ SDK** book which talks about the C/C++ SDK.

Chapter 4. Using an Integrated Development Environment (IDE)

The MicroPython port to Raspberry Pi Pico and other RP2040-based boards works with commonly used development environments.

4.1. Using Thonny

The current status of Thonny

Version 3.3.0 with the `thonny-pico` backend installed is ready for launch, except for module stubs for code completion. See the Github issue for details.

TO DO: Remove ahead of launch

Packages are available for Linux, MS Windows, and macOS. After installation, using the Thonny development environment is the same across all three platforms. The latest release (version 3.3.0) of Thonny can be downloaded from Github at https://github.com/thonny/thonny/releases/tag/v3.3.0.

> ℹ **NOTE**
>
> After the launch date Thonny should be downloaded from thonny.org

Alternatively if you are working on a Raspberry Pi you should install Thonny using `apt` from the command line,

```
$ sudo apt install thonny
```

this will add a Thonny icon to the Raspberry Pi desktop menu. Go ahead and select Raspberry Pi → Programming → Thonny Python IDE to open the development environment.

> ℹ **NOTE**
>
> When opening Thonny for the first time select "Standard Mode." For some versions this choice will be made via a popup when you first open Thonny. However for the Raspberry Pi release you should click on the text in the top right of the window to switch to "Regular Mode."

Download the Pico backend wheel from Github, https://github.com/raspberrypi/thonny-pico/releases/download/v0.1-post/thonny_rpi_pico-0.1-py3-none-any.whl. This wheel file can be installed into an existing Thonny installation (version 3.3.0b6 or later)

Start Thonny and navigate to "Tools → Manage plug-ins" and click on the link to "Install from local file" in the right hand panel, and select the Pico backend wheel (see Figure 3). Hit the "Close" button to finish. Afterwards you should **quit and restart Thonny**.

Figure 3. Installing the Raspberry Pi Pico Wheel file.
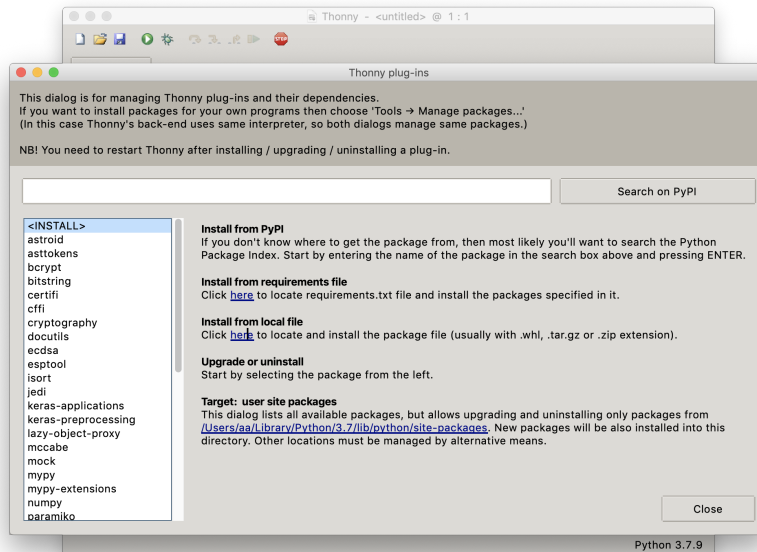
TO DO: Rewrite this section ahead of launch

4.1.1. Connecting to the Raspberry Pi Pico from Thonny

Connect your computer and the Raspberry Pi Pico together, see Chapter 2. Then open up the Run menu and select Run → Select Interpreter, picking "MicroPython (Raspberry Pi Pico)" from the drop down, see Figure 4.

Figure 4. Selecting the correct MicroPython interpreter inside the Thonny environment.
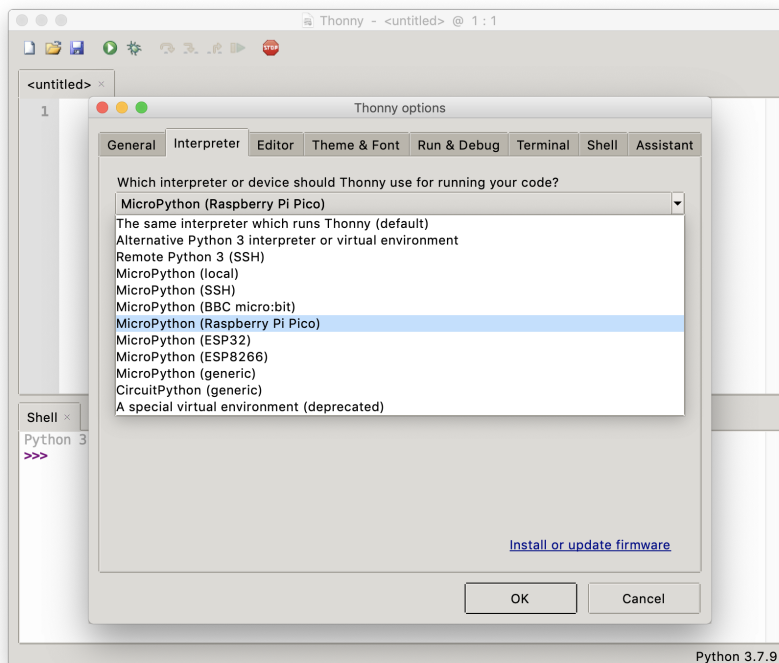
Hit "OK". If your Raspberry Pi Pico is plugged in and running MicroPython Thonny should automatically connect to the REPL.

If this doesn't happen go to Tools → Options menu item, and select your serial port in the drop down on the "Interpreter" tab. On the Raspberry Pi the serial port will be "Board in FS Mode — Board CDC (/dev/ttyACM0)" this should automatically connect you to the REPL of your Raspberry Pi Pico. Afterwards go to the "View" menu and select the "Variables" option to open the variables panel.
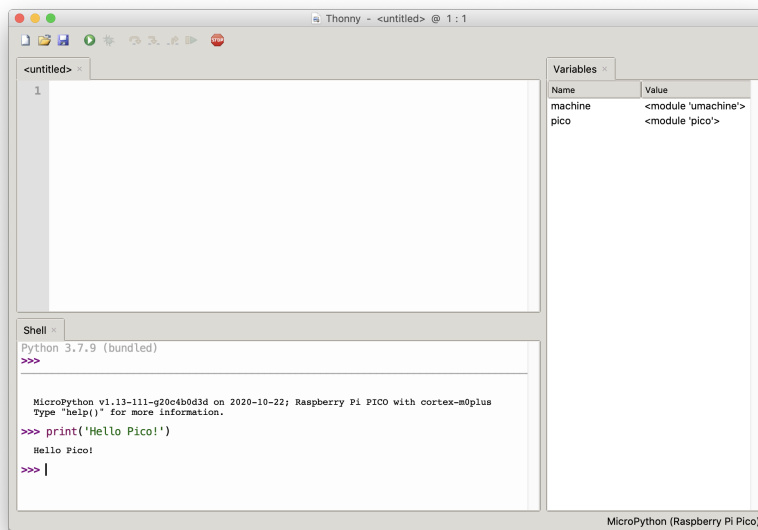
> **ℹ NOTE**
>
> In the rare case where you can't connect to Raspberry Pi Pico you may have to reboot your Raspberry Pi.

You can now access the REPL from the Shell panel,

```
>>> print('Hello Pico!')
Hello Pico!
>>>
```

see Figure 5.

Figure 5. Saying "Hello Pico!" from the MicroPython REPL inside the Thonny environment.


4.1.2. Blinking the LED from Thonny

You can use a timer to blink the on-board LED.

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/blink/blink.py Lines 1 - 9

```python
1 from machine import Pin, Timer
2
3 led = Pin(25, Pin.OUT)
4 tim = Timer()
5 def tick(timer):
6     global led
7     led.toggle()
8
9 tim.init(freq=2.5, mode=Timer.PERIODIC, callback=tick)
```

Enter the code in the main panel, then click on the green run button. Thonny will present you with a popup, click on "MicroPython device" and enter "test.py" to save the code to the Raspberry Pi Pico, see Figure 6.

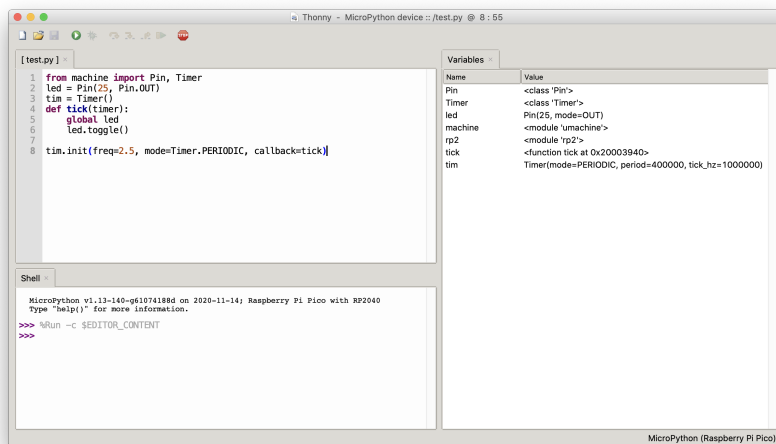*Figure 6. Saving code to the Raspberry Pi Pico inside the Thonny environment.*
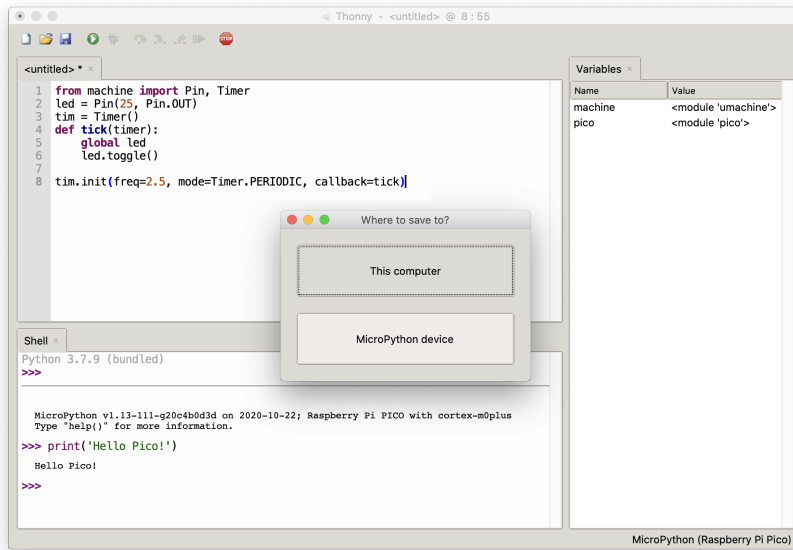
> **ⓘ NOTE**
>
> If you "save a file to the device" and give it the special name `main.py`, then MicroPython starts running that script as soon as power is supplied to Raspberry Pi Pico in the future.

The program should uploaded to the Raspberry Pi Pico using the REPL, and automatically start running. You should see the onboard LED start blinking, connected to GPIO pin 25, and the variables change in the Thonny variable window, see Figure 7.

Figure 7. Blinking an LED using a timer from the Thonny environment.

4.2. Using `rshell`

The Remote Shell for MicroPython (`rshell`) is a simple shell which runs on the host and uses MicroPython's REPL to send python code to the Raspberry Pi Pico in order to get filesystem information, and to copy files to and from MicroPython's own filesystem.

You can install rshell using,

```
$ sudo apt install python3-pip
$ sudo pip3 install rshell
```

Assuming that the UART is connected to your Raspberry Pi, see Section 2.1, then you can connect to Raspberry Pi Pico using,

```
$ rshell -p /dev/serial0
Connecting to /dev/serial0 (buffer-size 512)...
Trying to connect to REPL  connected
Testing if sys.stdin.buffer exists ... N
Retrieving root directories ...
Setting time ... Aug 21, 2020 15:35:18
Evaluating board_name ... pyboard
Retrieving time epoch ... Jan 01, 2000
Welcome to rshell. Use Control-D (or the exit command) to exit rshell.
/home/pi>
```

Full documentation of `rshell` can be found on the project's Github repository.
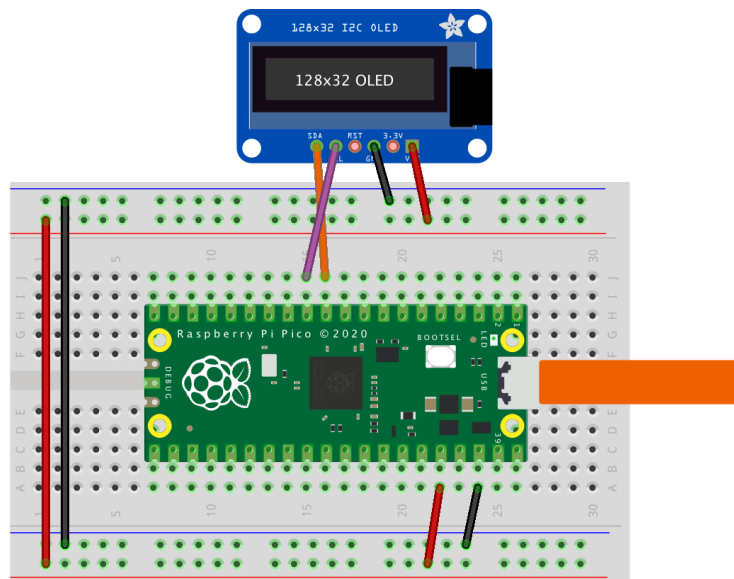
Appendix A: App Notes

Using a SSD1306-based OLED graphics display

Display an image and text on I2C driven SSD1306-based OLED graphics display.

Wiring information

See Figure 8 for wiring instructions.

Figure 8. Wiring the OLED to Pico using I2C

List of Files

A list of files with descriptions of their function;

i2c_1306oled_using_defaults.py

> The example code.

Pico MicroPython Examples: *https://github.com/raspberrypi/pico-micropython-examples/tree/master/i2c/1306oled/i2c_1306oled_using_defaults.py* *Lines 1 - 33*

```
1   # Display Image & text on I2C driven ssd1306 OLED display
2   from machine import Pin, I2C
3   from ssd1306 import SSD1306_I2C
4   import framebuf
5
6   WIDTH  = 128                            # oled display width
7   HEIGHT = 32                             # oled display height
8
9   i2c = I2C(0)                            # Init I2C using I2C0 defaults,
    SCL=Pin(GP9), SDA=Pin(GP8), freq=400000
10  print("I2C Address      : "+hex(i2c.scan()[0].upper()) # Display device address
11  print("I2C Configuration: "+str(i2c))          # Display I2C config
12
```

```
13
14 oled = SSD1306_I2C(WIDTH, HEIGHT, i2c)                # Init oled display
15
16 # Raspberry Pi logo as 32x32 bytearray
17 buffer = bytearray(b"\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00|?\x00\x01\x86
   @\x80\x01\x01\x80\x80\x01\x11\x88\x80\x01\x05\xa0\x80\x00\x83\xc1\x00\x00C\xe3\x00\x00
   ~\xfc\x00\x00L'\x00\x00\x9c\x11\x00\x00\xbf\xfd\x00\x00\xe1\x87\x00\x01\xc1\x83\x80\x02A
   \x82@\x02A\x82@\x02\xc1\xc2@\x02\xf6>\xc0\x01\xfc
   =\x80\x01\x18\x18\x80\x01\x88\x10\x80\x00\x8c!\x00\x00\x87\xf1\x00\x00\x7f\xf6\x00\x00
   8\x1c\x00\x00\x0c \x00\x00\x03\xc0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00")
18
19 # Load the raspberry pi logo into the framebuffer (the image is 32x32)
20 fb = framebuf.FrameBuffer(buffer, 32, 32, framebuf.MONO_HLSB)
21
22 # Clear the oled display in case it has junk on it.
23 oled.fill(0)
24
25 # Blit the image from the framebuffer to the oled display
26 oled.blit(fb, 96, 0)
27
28 # Add some text
29 oled.text("Raspberry Pi",5,5)
30 oled.text("Pico",5,15)
31
32 # Finally update the oled display so the image & text is displayed
33 oled.show()
```

i2c_1306oled_with_freq.py

The example code, explicitly sets a frequency.

```
 1 # Display Image & text on I2C driven ssd1306 OLED display
 2 from machine import Pin, I2C
 3 from ssd1306 import SSD1306_I2C
 4 import framebuf
 5
 6 WIDTH  = 128                                  # oled display width
 7 HEIGHT = 32                                   # oled display height
 8
 9 i2c = I2C(0, scl=Pin(9), sda=Pin(8), freq=200000)       # Init I2C using pins GP8 & GP9
   (default I2C0 pins)
10 print("I2C Address      : "+hex(i2c.scan()[0]).upper()) # Display device address
11 print("I2C Configuration: "+str(i2c))                   # Display I2C config
12
13
14 oled = SSD1306_I2C(WIDTH, HEIGHT, i2c)                # Init oled display
15
16 # Raspberry Pi logo as 32x32 bytearray
17 buffer = bytearray(b"\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00|?\x00\x01\x86
   @\x80\x01\x01\x80\x80\x01\x11\x88\x80\x01\x05\xa0\x80\x00\x83\xc1\x00\x00C\xe3\x00\x00
   ~\xfc\x00\x00L'\x00\x00\x9c\x11\x00\x00\xbf\xfd\x00\x00\xe1\x87\x00\x01\xc1\x83\x80\x02A
   \x82@\x02A\x82@\x02\xc1\xc2@\x02\xf6>\xc0\x01\xfc
   =\x80\x01\x18\x18\x80\x01\x88\x10\x80\x00\x8c!\x00\x00\x87\xf1\x00\x00\x7f\xf6\x00\x00
   8\x1c\x00\x00\x0c \x00\x00\x03\xc0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00")
18
19 # Load the raspberry pi logo into the framebuffer (the image is 32x32)
20 fb = framebuf.FrameBuffer(buffer, 32, 32, framebuf.MONO_HLSB)
21
22 # Clear the oled display in case it has junk on it.
23 oled.fill(0)
```

```
24
25 # Blit the image from the framebuffer to the oled display
26 oled.blit(fb, 96, 0)
27
28 # Add some text
29 oled.text("Raspberry Pi",5,5)
30 oled.text("Pico",5,15)
31
32 # Finally update the oled display so the image & text is displayed
33 oled.show()
```

Bill of Materials

Table 4. A list of materials required for the example

| Item	Quantity	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	http://raspberrypi.org/
Monochrome 128x32 I2C OLED Display	1	https://www.adafruit.com/product/931

Using PIO to drive a set of NeoPixel Ring (WS2812 LEDs)

Combination of the PIO WS2812 demo with the Adafruit 'essential' NeoPixel example code to show off color fills, chases and of course a rainbow swirl on a 16-LED ring.

Wiring information

See Figure 9 for wiring instructions.

Figure 9. Wiring the 16-LED NeoPixel Ring to Pico
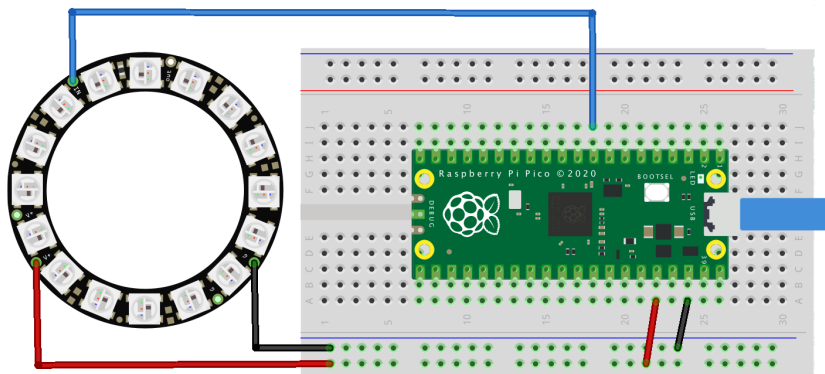
List of Files

A list of files with descriptions of their function;

neopixel_ring.py

    The example code.

```python
1  # Example using PIO to drive a set of WS2812 LEDs.
2
3  import array, time
4  from machine import Pin
5  import rp2
6
7  # Configure the number of WS2812 LEDs.
8  NUM_LEDS = 16
9  PIN_NUM = 6
10 brightness = 0.2
11
12 @rp2.asm_pio(sideset_init=rp2.PIO.OUT_LOW, out_shiftdir=rp2.PIO.SHIFT_LEFT, autopull=True,
   pull_thresh=24)
13 def ws2812():
14     T1 = 2
15     T2 = 5
16     T3 = 3
17     wrap_target()
18     label("bitloop")
19     out(x, 1)               .side(0)    [T3 - 1]
20     jmp(not_x, "do_zero")   .side(1)    [T1 - 1]
21     jmp("bitloop")          .side(1)    [T2 - 1]
22     label("do_zero")
23     nop()                   .side(0)    [T2 - 1]
24     wrap()
25
26
27 # Create the StateMachine with the ws2812 program, outputting on pin
28 sm = rp2.StateMachine(0, ws2812, freq=8_000_000, sideset_base=Pin(PIN_NUM))
29
30 # Start the StateMachine, it will wait for data on its FIFO.
31 sm.active(1)
32
33 # Display a pattern on the LEDs via an array of LED RGB values.
34 ar = array.array("I", [0 for _ in range(NUM_LEDS)])
35
36 ########################################################################
37 def pixels_show():
38     dimmer_ar = array.array("I", [0 for _ in range(NUM_LEDS)])
39     for i,c in enumerate(ar):
40         r = int(((c >> 8) & 0xFF) * brightness)
41         g = int(((c >> 16) & 0xFF) * brightness)
42         b = int((c & 0xFF) * brightness)
43         dimmer_ar[i] = (g<<16) + (r<<8) + b
44     sm.put(dimmer_ar, 8)
45     time.sleep_ms(10)
46
47 def pixels_set(i, color):
48     ar[i] = (color[1]<<16) + (color[0]<<8) + color[2]
49
50 def pixels_fill(color):
51     for i in range(len(ar)):
52         pixels_set(i, color)
53
54 def color_chase(color, wait):
55     for i in range(NUM_LEDS):
56         pixels_set(i, color)
57         time.sleep(wait)
58         pixels_show()
59     time.sleep(0.2)
60
```

```python
61 def wheel(pos):
62     # Input a value 0 to 255 to get a color value.
63     # The colours are a transition r - g - b - back to r.
64     if pos < 0 or pos > 255:
65         return (0, 0, 0)
66     if pos < 85:
67         return (255 - pos * 3, pos * 3, 0)
68     if pos < 170:
69         pos -= 85
70         return (0, 255 - pos * 3, pos * 3)
71     pos -= 170
72     return (pos * 3, 0, 255 - pos * 3)
73
74
75 def rainbow_cycle(wait):
76     for j in range(255):
77         for i in range(NUM_LEDS):
78             rc_index = (i * 256 // NUM_LEDS) + j
79             pixels_set(i, wheel(rc_index & 255))
80         pixels_show()
81         time.sleep(wait)
82
83 BLACK = (0, 0, 0)
84 RED = (255, 0, 0)
85 YELLOW = (255, 150, 0)
86 GREEN = (0, 255, 0)
87 CYAN = (0, 255, 255)
88 BLUE = (0, 0, 255)
89 PURPLE = (180, 0, 255)
90 WHITE = (255, 255, 255)
91 COLORS = (BLACK, RED, YELLOW, GREEN, CYAN, BLUE, PURPLE, WHITE)
92
93 print("fills")
94 for color in COLORS:
95     pixels_fill(color)
96     pixels_show()
97     time.sleep(0.2)
98
99 print("chases")
100 for color in COLORS:
101     color_chase(color, 0.01)
102
103 print("rainbow")
104 rainbow_cycle(0)
```

Bill of Materials

Table 5. A list of materials required for the example

Item	Quantity	Details
Breadboard	1	generic part
Raspberry Pi Pico	1	http://raspberrypi.org/
NeoPixel Ring	1	https://www.adafruit.com/product/1463