Pico Python SDK

A MicroPython environment for the RP2040 microcontroller

Colophon

build-date: 2021-01-04
build-version: githash: 1f2b413-dirty (pico-sdk: 04605c3-clean pico-examples: cc4c1c7-clean)

Legal Disclaimer Notice

TECHNICAL AND RELIABILITY DATA FOR RASPBERRY PI PRODUCTS (INCLUDING DATASHEETS) AS MODIFIED FROM TIME TO TIME ("RESOURCES") ARE PROVIDED BY RASPBERRY PI (TRADING) LTD ("RPTL) "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW IN NO EVENT SHALL RPTL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THE RESOURCES, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

RPTL reserves the right to make any enhancements, improvements, corrections or any other modifications to the RESOURCES or any products described in them at any time and without further notice.

The RESOURCES are intended for skilled users with suitable levels of design knowledge. Users are solely responsible for their selection and use of the RESOURCES and any application of the products described in them. User agrees to indemnify and hold RPTL harmless against all liabilities, costs, damages or other losses arising out of their use of the RESOURCES.

RPTL grants users permission to use the RESOURCES solely in conjunction with the Raspberry Pi products. All other use of the RESOURCES is prohibited. No licence is granted to any other RPTL or other third party intellectual property right.

HIGH RISK ACTIVITIES. Raspberry Pi products are not designed, manufactured or intended for use in hazardous environments requiring fail safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, weapons systems or safety-critical applications (including life support systems and other medical devices), in which the failure of the products could lead directly to death, personal injury or severe physical or environmental damage ("High Risk Activities"). RPTL specifically disclaims any express or implied warranty of fitness for High Risk Activities and accepts no liability for use or inclusions of Raspberry Pi products in High Risk Activities.

Raspberry Pi products are provided subject to RPTL's Standard Terms. RPTL's provision of the RESOURCES does not expand or otherwise modify RPTL's Standard Terms including but not limited to the disclaimers and warranties expressed in them.

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

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.

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

Pre-built Binary

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

To build the port, cd into `micropython` directory and,

```
$ 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.

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`.

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.

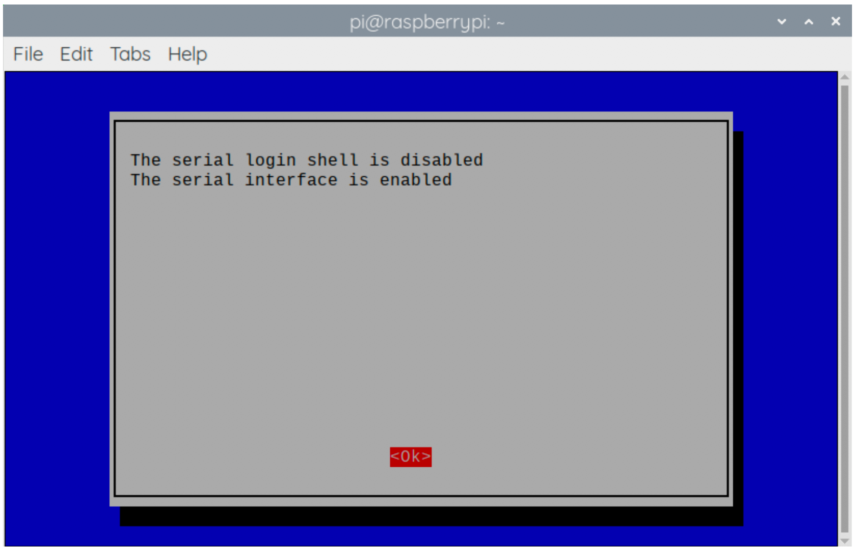


Figure 1. Enabling a serial UART using `raspi-config` on the Raspberry Pi.

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.

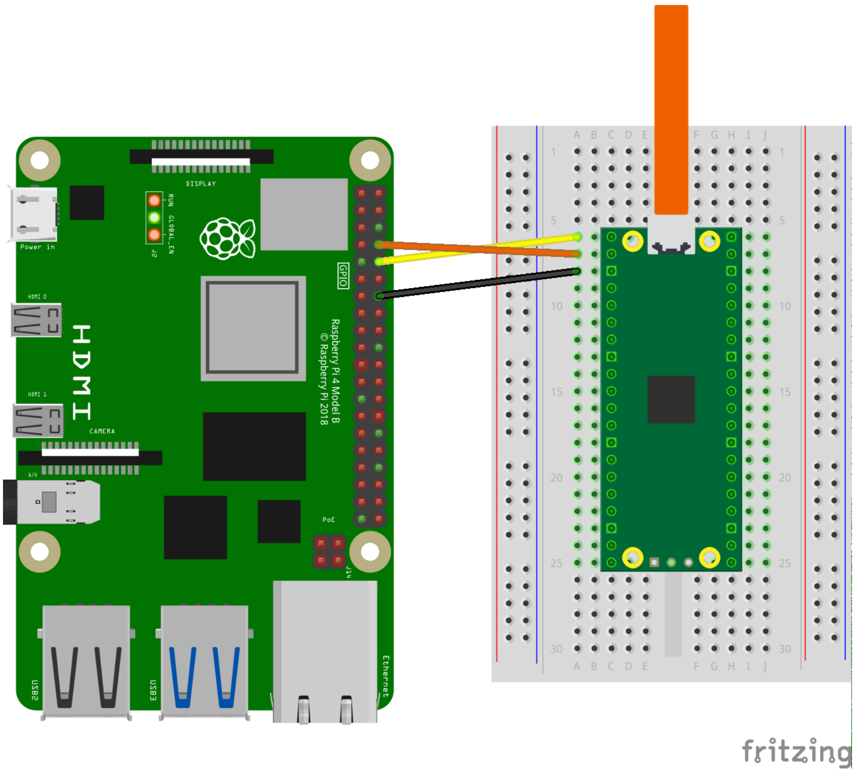


Figure 2. A Raspberry Pi 4 and the Raspberry Pi Pico with UART0 connected together.

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.

TO DO: Talk about the chip-specific features here

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

```
from machine import Pin, Timer

led = Pin(25, Pin.OUT)
tim = Timer()
def tick(timer):
    global led
    led.toggle()

tim.init(freq=2.5, mode=Timer.PERIODIC, callback=tick)
```

3.2. 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

```
from machine import Pin

p2 = Pin(2, Pin.IN, Pin.PULL_UP)
p2.irq(lambda pin: print("IRQ with flags:", pin.irq().flags()),
Pin.IRQ_FALLING)
```

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

3.3. Multicore Support

Example usage:

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/multicore/multicore.py Lines 1 - 12

```
import time, _thread, machine

def task(n, delay):
    led = machine.Pin(25, machine.Pin.OUT)
    for i in range(n):
        led.high()
        time.sleep(delay)
        led.low()
        time.sleep(delay)
    print('done')

_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.

3.4. I2C

Example usage:

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/i2c/i2c.py Lines 1 - 11

```
from machine import Pin, I2C

i2c = I2C(0, scl=Pin(9), sda=Pin(8), freq=100000)
i2c.scan()
i2c.writeto(76, b'123')
i2c.readfrom(76, 4)

i2c = I2C(1, scl=Pin(7), sda=Pin(6), freq=100000)
i2c.scan()
i2c.writeto_mem(76, 6, b'456')
i2c.readfrom_mem(76, 6, 4)
```

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

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/i2c/i2c_without_freq.py Lines 1 - 3

```
from machine import I2C

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.

3.5. SPI

Example usage:

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

```
from machine import SPI

spi = SPI(0)
spi = SPI(0, 100_000)
spi = SPI(0, 100_000, polarity=1, phase=1)

spi.write('test')
spi.read(5)

buf = bytearray(3)
spi.write_readinto('out', buf)
```

NOTE

The chip select must be managed separately using a `machine.Pin`.

3.6. 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

```python
# Example using PWM to fade an LED.

import time
from machine import Pin, PWM


# Construct PWM object, with LED on Pin(25).
pwm = PWM(Pin(25))

# Set the PWM frequency.
pwm.freq(1000)

# Fade the LED in and out a few times.
duty = 0
direction = 1
for _ in range(8 * 256):
    duty += direction
    if duty > 255:
        duty = 255
        direction = -1
    elif duty < 0:
        duty = 0
        direction = 1
    pwm.duty_u16(duty * duty)
    time.sleep(0.001)
```

3.7. 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*

```python
import time
from rp2 import PIO, asm_pio
from machine import Pin

# Define the blink program.  It has one GPIO to bind to on the set instruction, which is an output pin.
# Use lots of delays to make the blinking visible by eye.
@asm_pio(set_init=rp2.PIO.OUT_LOW)
def blink():
    wrap_target()
    set(pins, 1)   [31]
    nop()          [31]
    nop()          [31]
    nop()          [31]
    nop()          [31]
    set(pins, 0)   [31]
    nop()          [31]
    nop()          [31]
    nop()          [31]
    nop()          [31]
    wrap()

# Instantiate a state machine with the blink program, at 1000Hz, with set bound to Pin(25) (LED on the rp2 board)
sm = rp2.StateMachine(0, blink, freq=1000, set_base=Pin(25))

# Run the state machine for 3 seconds.  The LED should blink.
sm.active(1)
time.sleep(3)
sm.active(0)
```

or via explict exec.

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

```python
# Example using PIO to turn on an LED via an explicit exec.
#
# Demonstrates:
#   - using set_init and set_base
#   - using StateMachine.exec

import time
from machine import Pin
import rp2

# Define an empty program that uses a single set pin.
@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
def prog():
    pass


# Construct the StateMachine, binding Pin(25) to the set pin.
sm = rp2.StateMachine(0, prog, set_base=Pin(25))

# Turn on the set pin via an exec instruction.
sm.exec("set(pins, 1)")

# Sleep for 500ms.
time.sleep(0.5)
```

```python

# Turn off the set pin via an exec instruction.
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:

```python
@asm_pio(set_pins=(PIO.OUT_LOW, PIO.OUT_HIGH, PIO.IN_LOW), sideset_pins=PIO.OUT_LOW)
def foo():
    ....

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).

3.7.1. 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

```
import time
import rp2

@rp2.asm_pio()
def irq_test():
    wrap_target()
    nop()          [31]
    nop()          [31]
    nop()          [31]
    nop()          [31]
    irq(0)
    nop()          [31]
    nop()          [31]
    nop()          [31]
    nop()          [31]
    irq(1)
    wrap()


rp2.PIO(0).irq(lambda pio: print(pio.irq().flags()))

sm = rp2.StateMachine(0, irq_test, freq=1000)
sm.active(1)
time.sleep(1)
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

```
# Example using PIO to blink an LED and raise an IRQ at 1Hz.

import time
from machine import Pin
import rp2


@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
def blink_1hz():
    # Cycles: 1 + 1 + 6 + 32 * (30 + 1) = 1000
    irq(rel(0))
    set(pins, 1)
    set(x, 31)                  [5]
    label("delay_high")
    nop()                       [29]
    jmp(x_dec, "delay_high")

    # Cycles: 1 + 7 + 32 * (30 + 1) = 1000
    set(pins, 0)
    set(x, 31)                  [6]
    label("delay_low")
    nop()                       [29]
    jmp(x_dec, "delay_low")


# Create the StateMachine with the blink_1hz program, outputting on Pin(25).
sm = rp2.StateMachine(0, blink_1hz, freq=2000, set_base=Pin(25))

# Set the IRQ handler to print the millisecond timestamp.
```

```
sm.irq(lambda p: print(time.ticks_ms()))

# Start the StateMachine.
sm.active(1)
```

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

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/pio/pio_pinchange.py Lines 1 - 46

```
# Example using PIO to wait for a pin change and raise an IRQ.
#
# Demonstrates:
#   - PIO wrapping
#   - PIO wait instruction, waiting on an input pin
#   - PIO irq instruction, in blocking mode with relative IRQ number
#   - setting the in_base pin for a StateMachine
#   - setting an irq handler for a StateMachine
#   - instantiating 2x StateMachine's with the same program and different pins

import time
from machine import Pin
import rp2


@rp2.asm_pio()
def wait_pin_low():
    wrap_target()

    wait(0, pin, 0)
    irq(block, rel(0))
    wait(1, pin, 0)

    wrap()


def handler(sm):
    # Print a (wrapping) timestamp, and the state machine object.
    print(time.ticks_ms(), sm)


# Instantiate StateMachine(0) with wait_pin_low program on Pin(16).
pin16 = Pin(16, Pin.IN, Pin.PULL_UP)
sm0 = rp2.StateMachine(0, wait_pin_low, in_base=pin16)
sm0.irq(handler)

# Instantiate StateMachine(1) with wait_pin_low program on Pin(17).
pin17 = Pin(17, Pin.IN, Pin.PULL_UP)
sm1 = rp2.StateMachine(1, wait_pin_low, in_base=pin17)
sm1.irq(handler)

# Start the StateMachine's running.
sm0.active(1)
sm1.active(1)

# Now, when Pin(16) or Pin(17) is pulled low a message will be printed to the REPL.
```

3.7.2. WS2812 LED (NeoPixel)

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

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/pio/pio_ws2812.py Lines 1 - 52

```python
# Example using PIO to drive a set of WS2812 LEDs.

import array, time
from machine import Pin
import rp2

# Configure the number of WS2812 LEDs.
NUM_LEDS = 8


@rp2.asm_pio(sideset_init=rp2.PIO.OUT_LOW, out_shiftdir=rp2.PIO.SHIFT_LEFT, autopull=True, pull_thresh=24)
def ws2812():
    T1 = 2
    T2 = 5
    T3 = 3
    wrap_target()
    label("bitloop")
    out(x, 1)               .side(0)    [T3 - 1]
    jmp(not_x, "do_zero")   .side(1)    [T1 - 1]
    jmp("bitloop")          .side(1)    [T2 - 1]
    label("do_zero")
    nop()                   .side(0)    [T2 - 1]
    wrap()


# Create the StateMachine with the ws2812 program, outputting on Pin(22).
sm = rp2.StateMachine(0, ws2812, freq=8_000_000, sideset_base=Pin(22))

# Start the StateMachine, it will wait for data on its FIFO.
sm.active(1)

# Display a pattern on the LEDs via an array of LED RGB values.
ar = array.array("I", [0 for _ in range(NUM_LEDS)])

# Cycle colours.
for i in range(4 * NUM_LEDS):
    for j in range(NUM_LEDS):
        r = j * 100 // (NUM_LEDS - 1)
        b = 100 - j * 100 // (NUM_LEDS - 1)
        if j != i % NUM_LEDS:
            r >>= 3
            b >>= 3
        ar[j] = r << 16 | b
    sm.put(ar, 8)
    time.sleep_ms(50)

# Fade out.
for i in range(24):
    for j in range(NUM_LEDS):
        ar[j] >>= 1
    sm.put(ar, 8)
    time.sleep_ms(50)
```

3.7.3. 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

```python
# Example using PIO to create a UART TX interface

from machine import Pin
from rp2 import PIO, StateMachine, asm_pio

UART_BAUD = 115200
PIN_BASE = 10
NUM_UARTS = 8


@asm_pio(sideset_init=PIO.OUT_HIGH, out_init=PIO.OUT_HIGH, out_shiftdir=PIO.SHIFT_RIGHT)
def uart_tx():
    # Block with TX deasserted until data available
    pull()
    # Initialise bit counter, assert start bit for 8 cycles
    set(x, 7)  .side(0)       [7]
    # Shift out 8 data bits, 8 execution cycles per bit
    label("bitloop")
    out(pins, 1)              [6]
    jmp(x_dec, "bitloop")
    # Assert stop bit for 8 cycles total (incl 1 for pull())
    nop()      .side(1)       [6]


# Now we add 8 UART TXs, on pins 10 to 17. Use the same baud rate for all of them.
uarts = []
for i in range(NUM_UARTS):
    sm = StateMachine(
        i, uart_tx, freq=8 * UART_BAUD, sideset_base=Pin(PIN_BASE + i), out_base=Pin(PIN_BASE + i)
    )
    sm.active(1)
    uarts.append(sm)

# We can print characters from each UART by pushing them to the TX FIFO
def pio_uart_print(sm, s):
    for c in s:
        sm.put(ord(c))


# Print a different message from each UART
for i, u in enumerate(uarts):
    pio_uart_print(u, "Hello from UART {}!\n".format(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.

3.7.4. 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

```python
from machine import Pin

@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)
def spi_cpha0():
    # Note X must be preinitialised by setup code before first byte, we reload after sending each byte
    # Would normally do this via exec() but in this case it's in the instruction memory and is only run once
    set(x, 6)
    # Actual program body follows
    wrap_target()
    pull(ifempty)            .side(0x2)   [1]
    label("bitloop")
    out(pins, 1)             .side(0x0)   [1]
    in_(pins, 1)             .side(0x1)
    jmp(x_dec, "bitloop")    .side(0x1)

    out(pins, 1)             .side(0x0)
    set(x, 6)                .side(0x0) # Note this could be replaced with mov x, y for programmable frame size
    in_(pins, 1)             .side(0x1)
    jmp(not_osre, "bitloop") .side(0x1) # Fallthru if TXF empties

    nop()                    .side(0x0)   [1] # CSn back porch
    wrap()


class PIOSPI:

    def __init__(self, sm_id, pin_mosi, pin_miso, pin_sck, cpha=False, cpol=False, freq=1000000):
        assert(not(cpol or cpha))
        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))
        self._sm.active(1)

    # Note this code will die spectacularly cause we're not draining the RX FIFO
    def write_blocking(wdata):
        for b in wdata:
            self._sm.put(b << 24)

    def read_blocking(n):
        data = []
        for i in range(n):
            data.append(self._sm.get() & 0xff)
        return data

    def write_read_blocking(wdata):
        rdata = []
        for b in wdata:
            self._sm.put(b << 24)
            rdata.append(self._sm.get() & 0xff)
        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.

3.7.5. PWM

A PWM example,

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

```python
# Example of using PIO for PWM, and fading the brightness of an LED

from machine import Pin
from rp2 import PIO, StateMachine, asm_pio
from time import sleep


@asm_pio(sideset_init=PIO.OUT_LOW)
def pwm_prog():
    pull(noblock) .side(0)
    mov(x, osr) # Keep most recent pull data stashed in X, for recycling by noblock
    mov(y, isr) # ISR must be preloaded with PWM count max
    label("pwmloop")
    jmp(x_not_y, "skip")
    nop()         .side(1)
    label("skip")
    jmp(y_dec, "pwmloop")


class PIOPWM:
    def __init__(self, sm_id, pin, max_count, count_freq):
        self._sm = StateMachine(sm_id, pwm_prog, freq=2 * count_freq, sideset_base=Pin(pin))
        # Use exec() to load max count into ISR
        self._sm.put(max_count)
        self._sm.exec("pull()")
        self._sm.exec("mov(isr, osr)")
        self._sm.active(1)
        self._max_count = max_count

    def set(self, value):
        # Minimum value is -1 (completely turn off), 0 actually still produces narrow pulse
        value = max(value, -1)
        value = min(value, self._max_count)
        self._sm.put(value)


# Pin 25 is LED on Pico boards
pwm = PIOPWM(0, 25, max_count=(1 << 16) - 1, count_freq=10_000_000)

while True:
    for i in range(256):
        pwm.set(i ** 2)
        sleep(0.01)
```

3.7.6. 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.

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.

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**.

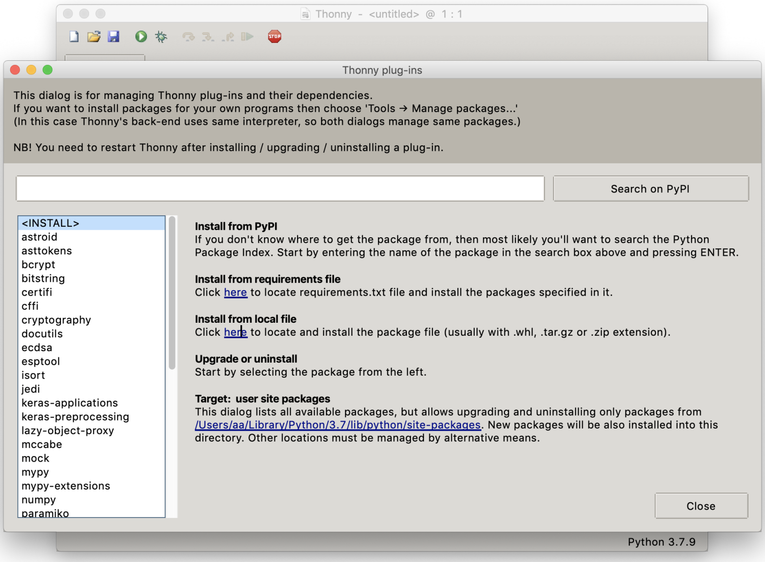


Figure 3. Installing the Raspberry Pi Pico Wheel file.

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.

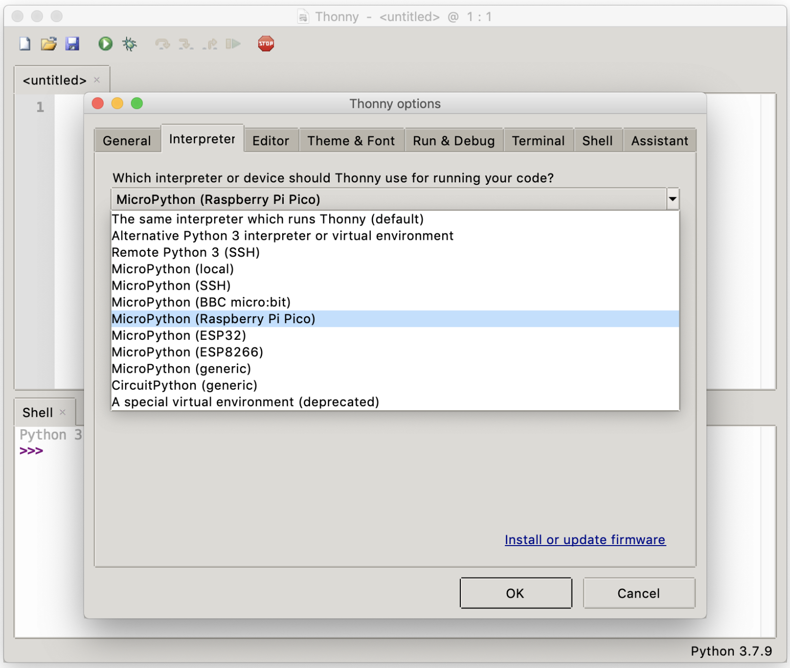


Figure 4. Selecting the correct MicroPython interpreter inside the Thonny environment.

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.

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.

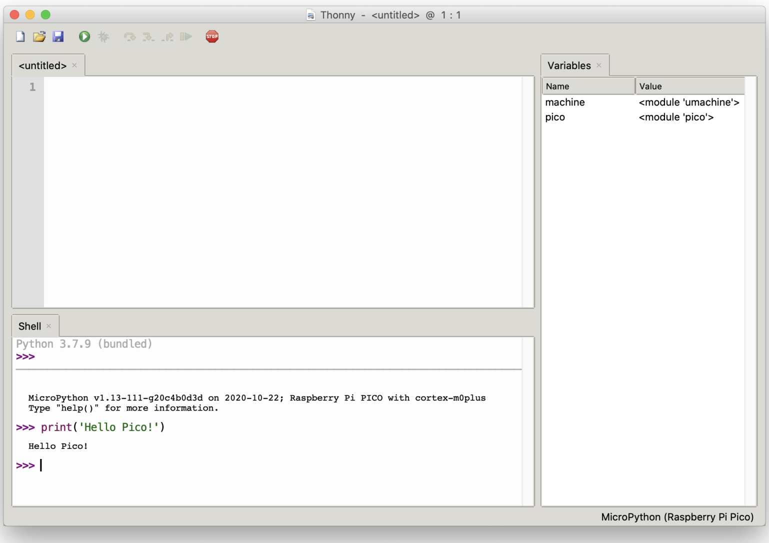


Figure 5. Saying "Hello Pico!" from the MicroPython REPL inside the Thonny environment.

4.1.2. Blinking the LED from Thonny

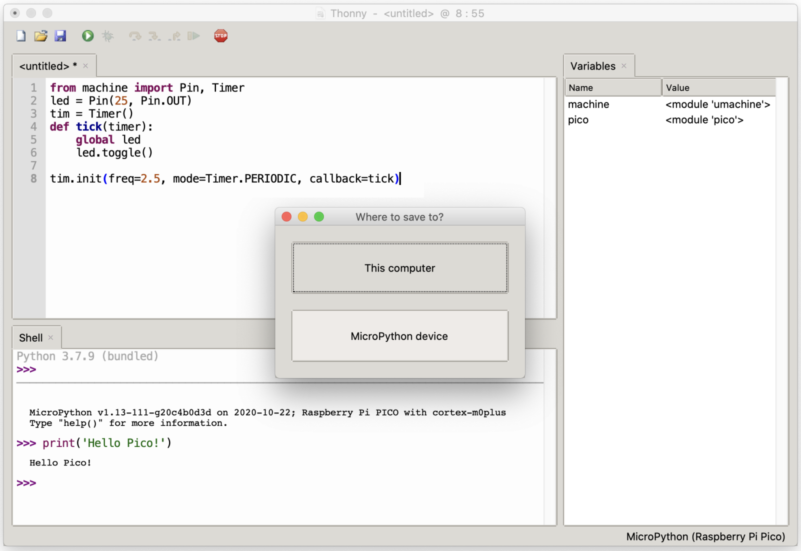
You can use a timer to blink the on-board LED.

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

```
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.

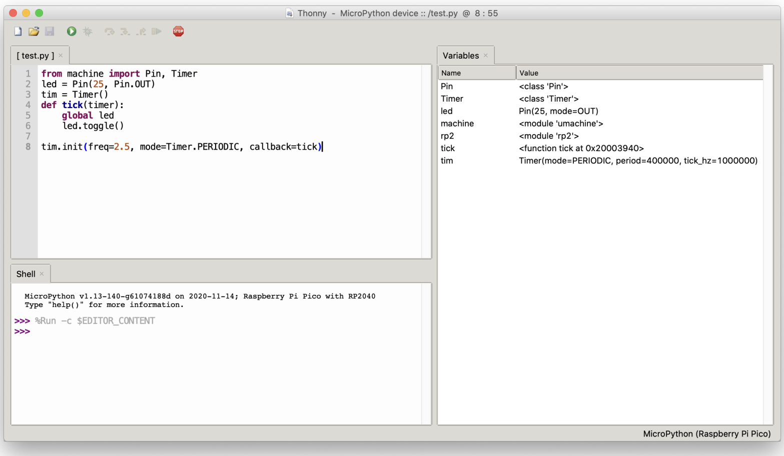


NOTE

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.

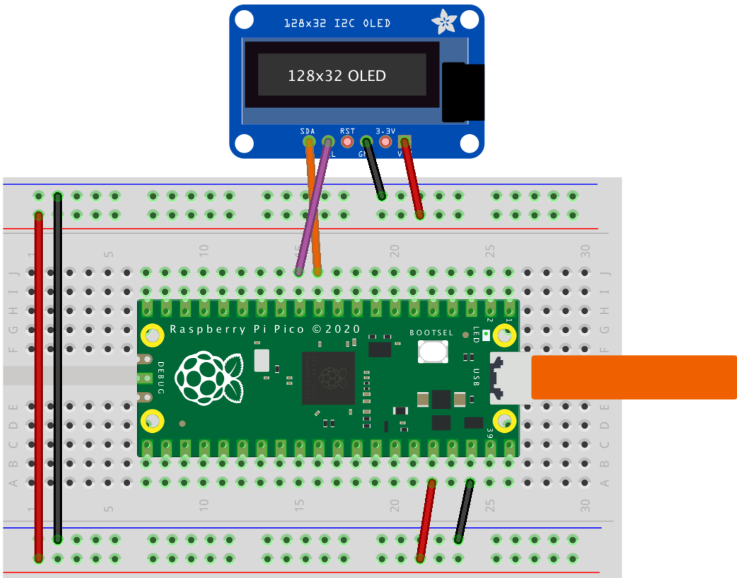


Figure 8. Wiring the OLED to Pico using I2C

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

```
 # Display Image & text on I2C driven ssd1306 OLED display
from machine import Pin, I2C
from ssd1306 import SSD1306_I2C
import framebuf

WIDTH  = 128                                            # oled display width
HEIGHT = 32                                             # oled display height

i2c = I2C(0)                                            # Init I2C using I2C0 defaults, SCL=Pin(GP9), SDA=Pin(GP8), freq=400000
print("I2C Address      : "+hex(i2c.scan()[0]).upper()) # Display device address
print("I2C Configuration: "+str(i2c))                   # Display I2C config

```

```

oled = SSD1306_I2C(WIDTH, HEIGHT, i2c)                  # Init oled display

# Raspberry Pi logo as 32x32 bytearray
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\x008\x1c\x00\x00\x0c \x00\x00\x03\xc0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00")

# Load the raspberry pi logo into the framebuffer (the image is 32x32)
fb = framebuf.FrameBuffer(buffer, 32, 32, framebuf.MONO_HLSB)

# Clear the oled display in case it has junk on it.
oled.fill(0)

# Blit the image from the framebuffer to the oled display
oled.blit(fb, 96, 0)

# Add some text
oled.text("Raspberry Pi",5,5)
oled.text("Pico",5,15)

# Finally update the oled display so the image & text is displayed
oled.show()
```

i2c_1306oled_with_freq.py

The example code, explicitly sets a frequency.

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/i2c/1306oled/i2c_1306oled_with_freq.py Lines 1 - 33

```
# Display Image & text on I2C driven ssd1306 OLED display
from machine import Pin, I2C
from ssd1306 import SSD1306_I2C
import framebuf

WIDTH  = 128                                            # oled display width
HEIGHT = 32                                             # oled display height

i2c = I2C(0, scl=Pin(9), sda=Pin(8), freq=200000)       # Init I2C using pins GP8 & GP9 (default I2C0 pins)
print("I2C Address      : "+hex(i2c.scan()[0]).upper()) # Display device address
print("I2C Configuration: "+str(i2c))                   # Display I2C config


oled = SSD1306_I2C(WIDTH, HEIGHT, i2c)                  # Init oled display

# Raspberry Pi logo as 32x32 bytearray
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\x008\x1c\x00\x00\x0c \x00\x00\x03\xc0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00")

# Load the raspberry pi logo into the framebuffer (the image is 32x32)
fb = framebuf.FrameBuffer(buffer, 32, 32, framebuf.MONO_HLSB)

# Clear the oled display in case it has junk on it.
oled.fill(0)
```

```python

# Blit the image from the framebuffer to the oled display
oled.blit(fb, 96, 0)

# Add some text
oled.text("Raspberry Pi",5,5)
oled.text("Pico",5,15)

# Finally update the oled display so the image & text is displayed
oled.show()
```

Bill of Materials

Table 1. 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.

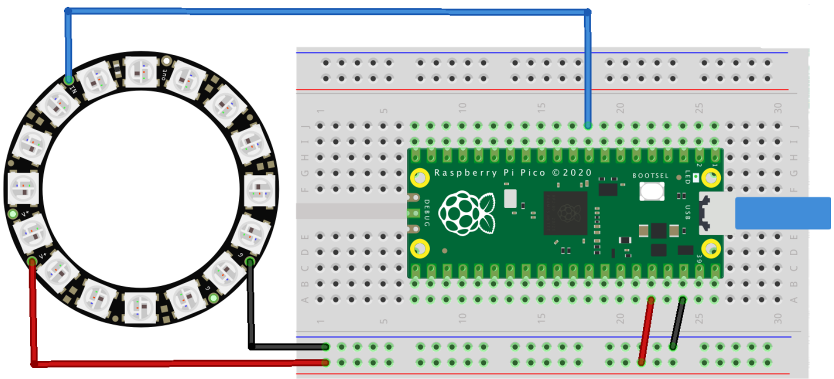


Figure 9. Wiring the 16-LED NeoPixel Ring to Pico

List of Files

A list of files with descriptions of their function;

neopixel_ring.py

The example code.

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/pio/neopixel_ring/neopixel_ring.py Lines 1 - 104

```python
# Example using PIO to drive a set of WS2812 LEDs.

import array, time
from machine import Pin
import rp2

# Configure the number of WS2812 LEDs.
NUM_LEDS = 16
PIN_NUM = 6
brightness = 0.2

@rp2.asm_pio(sideset_init=rp2.PIO.OUT_LOW, out_shiftdir=rp2.PIO.SHIFT_LEFT, autopull=True, pull_thresh=24)
def ws2812():
    T1 = 2
    T2 = 5
    T3 = 3
    wrap_target()
    label("bitloop")
    out(x, 1)               .side(0)    [T3 - 1]
    jmp(not_x, "do_zero")   .side(1)    [T1 - 1]
    jmp("bitloop")          .side(1)    [T2 - 1]
    label("do_zero")
    nop()                   .side(0)    [T2 - 1]
    wrap()


# Create the StateMachine with the ws2812 program, outputting on pin
sm = rp2.StateMachine(0, ws2812, freq=8_000_000, sideset_base=Pin(PIN_NUM))

# Start the StateMachine, it will wait for data on its FIFO.
sm.active(1)

# Display a pattern on the LEDs via an array of LED RGB values.
ar = array.array("I", [0 for _ in range(NUM_LEDS)])

##########################################################################
def pixels_show():
    dimmer_ar = array.array("I", [0 for _ in range(NUM_LEDS)])
    for i,c in enumerate(ar):
        r = int(((c >> 8) & 0xFF) * brightness)
        g = int(((c >> 16) & 0xFF) * brightness)
        b = int((c & 0xFF) * brightness)
        dimmer_ar[i] = (g<<16) + (r<<8) + b
    sm.put(dimmer_ar, 8)
    time.sleep_ms(10)

def pixels_set(i, color):
    ar[i] = (color[1]<<16) + (color[0]<<8) + color[2]

def pixels_fill(color):
    for i in range(len(ar)):
        pixels_set(i, color)

def color_chase(color, wait):
    for i in range(NUM_LEDS):
        pixels_set(i, color)
        time.sleep(wait)
        pixels_show()
    time.sleep(0.2)

```

```
def wheel(pos):
    # Input a value 0 to 255 to get a color value.
    # The colours are a transition r - g - b - back to r.
    if pos < 0 or pos > 255:
        return (0, 0, 0)
    if pos < 85:
        return (255 - pos * 3, pos * 3, 0)
    if pos < 170:
        pos -= 85
        return (0, 255 - pos * 3, pos * 3)
    pos -= 170
    return (pos * 3, 0, 255 - pos * 3)


def rainbow_cycle(wait):
    for j in range(255):
        for i in range(NUM_LEDS):
            rc_index = (i * 256 // NUM_LEDS) + j
            pixels_set(i, wheel(rc_index & 255))
        pixels_show()
        time.sleep(wait)

BLACK = (0, 0, 0)
RED = (255, 0, 0)
YELLOW = (255, 150, 0)
GREEN = (0, 255, 0)
CYAN = (0, 255, 255)
BLUE = (0, 0, 255)
PURPLE = (180, 0, 255)
WHITE = (255, 255, 255)
COLORS = (BLACK, RED, YELLOW, GREEN, CYAN, BLUE, PURPLE, WHITE)

print("fills")
for color in COLORS:
    pixels_fill(color)
    pixels_show()
    time.sleep(0.2)

print("chases")
for color in COLORS:
    color_chase(color, 0.01)

print("rainbow")
rainbow_cycle(0)
```

Bill of Materials

Table 2. 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

Raspberry Pi

Raspberry Pi Trading Ltd