Raspberry Pi Pico Python SDK
A MicroPython environment for RP2040 microcontrollers
About the SDK

Throughout the text ‘the SDK’ refers to our Raspberry Pi Pico SDK. More details about the SDK can be found in the Raspberry Pi Pico C/C++ SDK book.

Release History

<table>
<thead>
<tr>
<th>Release</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>21/Jan/2021</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>
| 1.1     | 26/Jan/2021| * Minor corrections  
* Extra information about using DMA with ADC  
* Clarified M0+ and SIO CPUID registers  
* Added more discussion of Timers  
* Update Windows and macOS build instructions  
* Renamed books and optimised size of output PDFs |
| 1.2     | 01/Feb/2021| * Minor corrections  
* Small improvements to PIO documentation  
* Added missing TIMER2 and TIMER3 registers to DMA  
* Explained how to get MicroPython REPL on UART  
* To accompany the V1.0.1 release of the C SDK |


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

Colophon ........................................................................................................... 1
Release History .................................................................................................... 1
Legal Disclaimer Notice ....................................................................................... 1
1. The MicroPython Environment ..................................................................... 4
   1.1. Getting MicroPython for RP2040 ......................................................... 4
   1.2. Installing MicroPython on Raspberry Pi Pico ..................................... 4
   1.3. Building MicroPython From Source .................................................. 5
2. Connecting to the MicroPython REPL ......................................................... 8
   2.1. Connecting from a Raspberry Pi over USB ......................................... 8
   2.2. Connecting from a Raspberry Pi using UART ............................... 9
   2.3. Connecting from a Mac using USB .................................................. 11
   2.4. Say “Hello World” ........................................................................... 11
   2.5. Blink an LED ..................................................................................... 11
   2.6. What next? ....................................................................................... 12
3. The RP2040 Port ............................................................................................ 13
   3.1. Blinking an LED Forever (Timer) ....................................................... 13
   3.2. UART ............................................................................................... 14
   3.3. ADC ............................................................................................... 14
   3.4. Interrupts ......................................................................................... 15
   3.5. Multicore Support .......................................................................... 15
   3.6. I2C .................................................................................................. 16
   3.7. SPI ................................................................................................. 17
   3.8. PWM ............................................................................................... 17
   3.9. PIO Support ..................................................................................... 18
      3.9.1. IRQ ........................................................................................... 20
      3.9.2. WS2812 LED (NeoPixel) ............................................................ 22
      3.9.3. UART TX ................................................................................ 23
      3.9.4. SPI .......................................................................................... 24
      3.9.5. PWM ....................................................................................... 25
      3.9.6. Using pioasm ......................................................................... 26
4. Using an Integrated Development Environment (IDE) ............................ 27
   4.1. Using Thonny ................................................................................... 27
      4.1.1. Connecting to the Raspberry Pi Pico from Thonny .................. 28
      4.1.2. Blinking the LED from Thonny .................................................. 29
   4.2. Using rshell ..................................................................................... 30
Appendix A App Notes ....................................................................................... 32
   Using a SSD1306-based OLED graphics display .................................... 32
      Wiring information .............................................................................. 32
      List of Files ......................................................................................... 32
      Bill of Materials .................................................................................. 34
   Using a SH1106-based OLED graphics display .................................... 34
      Wiring information .............................................................................. 34
      List of Files ......................................................................................... 35
      Bill of Materials .................................................................................. 40
   Using PIO to drive a set of NeoPixel Ring (WS2812 LEDs) .................. 40
      Wiring information .............................................................................. 40
      List of Files ......................................................................................... 41
      Bill of Materials .................................................................................. 42
Chapter 1. The MicroPython Environment

Python is the fastest way to get started with embedded software on Raspberry Pi Pico. This book is about the official MicroPython port for RP2040-based microcontroller boards.

MicroPython is a Python 3 implementation for microcontrollers and small embedded systems. Because MicroPython is highly efficient, and RP2040 is designed with a disproportionate amount of system memory and processing power for its price, MicroPython is a serious tool for embedded systems development, which does not compromise on approachability.

For exceptionally demanding pieces of software, you can fall back on the SDK (covered in Getting started with Raspberry Pi Pico and Raspberry Pi Pico C/C++ SDK), or an external C module added to your MicroPython firmware, to wring out the very last drop of performance. For every other project, MicroPython handles a lot of heavy lifting for you, and lets you focus on writing the code that adds value to your project. The accelerated floating point libraries in RP2040's on-board ROM storage are used automatically by your Python code, so you should find arithmetic performance quite snappy.

Most on-chip hardware is exposed through the standard `machine` module, so existing MicroPython projects can be ported without too much trouble. The second processor core is exposed through the `thread` module.

RP2040 has some unique hardware you won't find on other microcontrollers, with the programmable I/O system (PIO) being the prime example of this: a versatile hardware subsystem that lets you create new I/O interfaces and run them at high speed. In the `rp2` module you will find a comprehensive PIO library which lets you write new PIO programs at the MicroPython prompt, and interact with them in real time, to develop interfaces for new or unusual pieces of hardware (or indeed if you just find yourself wanting an extra few serial ports).

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`, `frozenset`, `array.array`, `collections.namedtuple`, classes and instances. Built-in modules include `sys`, `time`, and `struct`, etc. 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 (.py files) or from precompiled bytecode, in both cases either from an on-device filesystem or “frozen” into the MicroPython executable.

1.1. Getting MicroPython for RP2040

Pre-built Binary

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

The fastest way to get MicroPython is to download the pre-built release binary from the Pico Getting Started pages. If you can’t or don’t want to use the pre-built release — for example, if you want to develop a C module for MicroPython — you can follow the instructions in Section 1.3 to get the source code for MicroPython, which you can use to build your own MicroPython firmware binary.

1.2. Installing MicroPython on Raspberry Pi Pico

Raspberry Pi Pico has a BOOTSEL mode for programming firmware over the USB port. Holding the BOOTSEL button when powering up your board will put it into a special mode where it appears as a USB Mass Storage Device. First make sure your Raspberry Pi Pico is not plugged into any source of power: disconnect the micro USB cable if plugged in, and disconnect any other wires that might be providing power to the board, e.g. through the VSYS or VBUS pin. Now hold
down the BOOTSEL button, and plug in the micro USB cable (which hopefully has the other end plugged into your computer).

A drive called RPI-RP2 should pop up. Go ahead and drag the MicroPython firmware.uf2 file onto this drive. This programs the MicroPython firmware onto the flash memory on your Raspberry Pi Pico.

It should take a few seconds to program the UF2 file into the flash. The board will automatically reboot when finished, causing the RPI-RP2 drive to disappear, and boot into MicroPython.

By default, MicroPython doesn't do anything when it first boots. It sits and waits for you to type in further instructions. Chapter 2 shows how you can connect with the MicroPython firmware now running on your board. You can read on to see how a custom MicroPython firmware file can be built from the source code.

The Getting started with Raspberry Pi Pico book has detailed instructions on getting your Raspberry Pi Pico into BOOTSEL mode and loading UF2 files, in case you are having trouble. There is also a section going over loading ELF files with the debugger, in case your board doesn't have an easy way of entering BOOTSEL, or you would like to debug a MicroPython C module you are developing.

### 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 BOOTSEL mode on boot. If there is no such method, you can load code using the Serial Wire Debug interface.

## 1.3. Building MicroPython From Source

The prebuilt binaries on the Pico Getting Started pages should serve most use cases, but you can build your own MicroPython firmware from source if you'd like to customise its low-level aspects.

### TIP

If you have already downloaded and installed a prebuilt MicroPython UF2 file, you can skip ahead to Chapter 2 to start using your board.

### IMPORTANT

These instructions for getting and building MicroPython assume you are using Raspberry Pi OS running on a Raspberry Pi 4, or an equivalent Debian-based Linux distribution running on another platform.

It's a good idea to create a pico directory to keep all pico-related checkouts in. These instructions create a pico directory at /home/pi/pico.

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

Then clone the micropython git repository. These instructions will fetch the latest version of the source code.

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

Once the download has finished, the source code for MicroPython should be in a new directory called micropython. The MicroPython repository also contains pointers (submodules) to specific versions of libraries it needs to run on a particular board, like the SDK in the case of RP2040. We need to explicitly fetch these too:
To build the RP2040 MicroPython port, you'll 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, which turns MicroPython's C source code into a binary program RP2040's processors can understand. build-essential is a bundle of tools you need to build code native to your own machine — this is needed for some internal tools in MicroPython and the SDK. You can install all of these via apt from the command line. Anything you already have installed will be ignored by apt.

```bash
$ sudo apt update
$ sudo apt install cmake gcc-arm-none-eabi libnewlib-arm-none-eabi build-essential
```

To build the port, you first need to change directory into the micropython repository containing the source. If you've been following along with the instructions, you'll need to go up two directories.

```bash
$ cd ../../
```

First we need to bootstrap a special tool for MicroPython builds, that ships with the source code:

```bash
$ make -C mpy-cross
```

We can now build the port we need for RP2040, that is, the version of MicroPython that has specific support for our chip.

```bash
$ cd ports/rp2
$ make
```

If everything went well, there will be a new directory called build (ports/rp2/build relative to the micropython directory), which contains the new firmware binaries. The most important ones are:

- **firmware.uf2**: A UF2 binary file which can be dragged onto the RPI-RP2 drive that pops up once your Raspberry Pi Pico is in BOOTSEL mode. The firmware binaries you will find on Pico Getting Started pages are UF2 files, because they're the easiest to install.

- **firmware.elf**: A different type of binary file, which can be loaded by a debugger (such as gdb with openocd) over RP2040's SWD debug port. This is useful for debugging either a native C module you've added to MicroPython, or the MicroPython core interpreter itself. The actual binary contents is the same as firmware.uf2.

You can take a look inside your new firmware.uf2 using picotool, see the Appendix B in the Getting started with Raspberry Pi Pico Python SDK.
Pi Pico book for details, e.g.

$ picotool info -a build/firmware.uf2
File /home/pi/pico/micropython/ports/rp2/build/firmware.uf2:

Program Information
  name: MicroPython
  version: v1.13-288-g3ce8f14e0
  features: USB REPL
            thread support
  frozen modules: _boot, rp2, ds18x20, onewire, uasyncio, uasyncio/core,
                  uasyncio/event, uasyncio/funcs, uasyncio/lock, uasyncio/stream
  binary start: 0x10000000
  binary end: 0x10038be4
  embedded drive: 0x100a0000-0x10200000 (1408K): MicroPython

Fixed Pin Information
  none

Build Information
  sdk version: 1.0.0
  pico_board: pico
  build date: Jan 21 2021
  build attributes: MinSizeRel
Chapter 2. Connecting to the MicroPython REPL

When MicroPython boots for the first time, it will sit and wait for you to connect and tell it what to do. You can load a `.py` file from your computer onto the board, but a more immediate way to interact with it is through what is called the read-evaluate-print loop, or REPL (often pronounced similarly to "ripple").

- **Read**: MicroPython waits for you to type in some text, followed by the enter key.
- **Evaluate**: Whatever you typed is interpreted as Python code, and runs immediately.
- **Print**: Any results of the last line you typed are printed out for you to read.
- **Loop**: Go back to the start — prompt you for another line of code.

There are two ways to connect to this REPL, so you can communicate with the MicroPython firmware on your board: over USB, and over the UART serial port on Raspberry Pi Pico GPIOs.

### 2.1. Connecting from a Raspberry Pi over USB

The MicroPython firmware is equipped with a virtual USB serial port which is accessed through the micro USB connector on Raspberry Pi Pico. Your computer should notice this serial port and list it as a character device, most likely `/dev/ttyACM0`.

**Tip**

You can run `ls /dev/tty*` to list your serial ports. There may be quite a few, but MicroPython’s USB serial will start with `/dev/ttyACM`. If in doubt, unplug the micro USB connector and see which one disappears. If you don’t see anything, you can try rebooting your Raspberry Pi.

You can install `minicom` to access the serial port:

```
$ sudo apt install minicom
```

and then open it as such:

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

Where the `-D /dev/ttyACM0` is pointing `minicom` at MicroPython’s USB serial port, and the `-o` flag essentially means "just do it". There’s no need to worry about baud rate, since this is a virtual serial port.

Press the enter key a few times in the terminal where you opened `minicom`. You should see this:

```
>>> 
```

This is a prompt. MicroPython wants you to type something in, and tell it what to do.

If you press `CTRL-D` on your keyboard whilst the `minicom` terminal is focused, you should see a message similar to this:
This key combination tells MicroPython to reboot. You can do this at any time. When it reboots, MicroPython will print out a message saying exactly what firmware version it is running, and when it was built. Your version number will be different from the one shown here.

2.2. Connecting from a Raspberry Pi using UART

**NOTE**

REPL over UART is disabled by default.

The MicroPython port for RP2040 does not expose REPL over a UART port by default. However this default can be changed in the `mpconfigport.h` source file.

Go ahead and download the MicroPython source (see Section 1.3) and in `ports/rp2/mpconfigport.h` change `MICROPY_HW_ENABLE_UART_REPL` to 1 to enable it.

```c
#define MICROPY_HW_ENABLE_UART_REPL (1) // useful if there is no USB
```

Then continue to follow the instructions in Section 1.3 to build your own MicroPython UF2 firmware.

This will allow the REPL to be accessed over a UART port, through two GPIOs pin. By default on Raspberry Pi Pico this is on GPIO0 (TX, MicroPython output) and GPIO1 (RX, MicroPython input), and the speed is 115200 baud. This alternative interface is handy if you have trouble with USB, if you don't have any free USB ports, or if you are using some other RP2040-based board which doesn't have an exposed USB connector.

**NOTE**

This initially occupies the UART0 peripheral on RP2040. The UART1 peripheral is free for you to use in your Python code as a second UART.

To connect, 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.
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:

<table>
<thead>
<tr>
<th>Raspberry Pi</th>
<th>Raspberry Pi Pico</th>
</tr>
</thead>
<tbody>
<tr>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>GPIO15 (UART_RX0)</td>
<td>GPIO0 (UART0_TX)</td>
</tr>
<tr>
<td>GPIO14 (UART_TX0)</td>
<td>GPIO11 (UART0_RX)</td>
</tr>
</tbody>
</table>

**IMPORTANT**

RX matches to TX, and TX matches to RX. You mustn’t connect the two opposite TX pins together, or the two RX pins. This is because MicroPython needs to listen on the channel that the Raspberry Pi transmits on, and vice versa.

See Figure 2.

Then connect to the board using `minicom` connected to `/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 to connect to Serial-over-USB (USB CDC). The serial port will show up as /dev/tty.usbmodem0000000000001

If you don't already have a Terminal program installed you can install minicom using Homebrew,

```bash
brew install minicom
```

and connect to the board as below.

```bash
minicom -b 115200 -o -D /dev/tty.usbmodem0000000000001
```

**NOTE**

Other applications like CoolTerm or Serial can also be used.

2.4. Say "Hello World"

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

```python
>>> print("Hello, Pico!")
Hello, Pico!
```  

2.5. Blink an LED

The on-board LED on 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,

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

The `machine` module is used to control on-chip hardware. This is standard on all MicroPython ports, and you can read more
about it in the MicroPython documentation. Here we are using it to take control of a GPIO, so we can drive it high and low. If you type this in,

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

The LED should turn on. You can turn it off again with

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

### 2.6. What next?

At this point you should have MicroPython installed on your board, and have tested your setup by typing short programs into the prompt to print some text back to you, and blink an LED.

You can read on to the next chapter, which goes into the specifics of MicroPython on RP2040, and where it differs from other platforms. Chapter 3 also has some short examples of the different APIs offered to interact with the hardware.

You can learn how to set up an integrated development environment (IDE) in Chapter 4, so you don’t have to type programs in line by line.

You can dive straight into Appendix A if you are eager to start connecting wires to a breadboard.
Chapter 3. The RP2040 Port

Currently supported features include:

- REPL over USB and UART (on GP0/GP1).
- 1600 kB filesystem using littlefs2 on the on-board flash. (Default size for Raspberry Pi Pico)
- utime module with sleep and ticks functions.
- ubinascii module.
- machine module with some basic functions.
  - machine.Pin class.
  - machine.Timer class.
  - machine.ADC class.
  - machine.SPI and machine.SoftSPI classes.
  - machine.WDT class.
  - machine.PWM class.
  - machine.UART class.
- rp2 platform-specific module.
  - PIO hardware access library
  - PIO program assembler
  - Raw flash read/write access
- Multicore support exposed via the standard _thread module
- Accelerated floating point arithmetic using the RP2040 ROM library and hardware divider (used automatically)

Documentation around MicroPython is available from https://docs.micropython.org. For example the machine module, which can be used to access a lot of RP2040's on-chip hardware, is standard, and you will find a lot of the information you need in the online documentation for that module.

This chapter will give a very brief tour of some of the hardware APIs, with code examples you can either type into the REPL (Chapter 2) or load onto the board using a development environment installed on your computer (Chapter 4).

3.1. Blinking an LED Forever (Timer)

In Chapter 2 we saw how the machine.Pin class could be used to turn an LED on and off, by driving a GPIO high and low.

```python
>>> from machine import Pin
>>> led = Pin(25, Pin.OUT)
>>> led.value(1)
>>> led.value(0)
```

This is, to put it mildly, quite a convoluted way of turning a light on and off. A light switch would work better. The machine.Timer class, which uses RP2040's hardware timer to trigger callbacks at regular intervals, saves a lot of typing if we want the light to turn itself on and off repeatedly, thus bringing our level of automation from 'mechanical switch' to "555 timer".
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)

Typing this program into the REPL will cause the LED to start blinking, but the prompt will appear again:

>>> The Timer we created will run in the background, at the interval we specified, blinking the LED. The MicroPython prompt is still running in the foreground, and we can enter more code, or start more timers.

3.2. UART

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>UART_BAUDRATE</td>
<td>115200</td>
</tr>
<tr>
<td>UART_BITS</td>
<td>8</td>
</tr>
<tr>
<td>UART_STOP</td>
<td>1</td>
</tr>
<tr>
<td>UART0_TX</td>
<td>Pin 0</td>
</tr>
<tr>
<td>UART0_RX</td>
<td>Pin 1</td>
</tr>
<tr>
<td>UART1_TX</td>
<td>Pin 4</td>
</tr>
<tr>
<td>UART1_RX</td>
<td>Pin 5</td>
</tr>
</tbody>
</table>

3.3. ADC

An analogue-to-digital converter (ADC) measures some analogue signal and encodes it as a digital number. The ADC on RP2040 measures voltages.

An ADC has two key features: its resolution, measured in digital bits, and its channels, or how many analogue signals it can accept and convert at once. The ADC on RP2040 has a resolution of 12-bits, meaning that it can transform an analogue signal into a digital signal as a number ranging from 0 to 4095 – though this is handled in MicroPython transformed to a 16-bit number ranging from 0 to 65,535, so that it behaves the same as the ADC on other MicroPython microcontrollers.

RP2040 has five ADC channels total, four of which are brought out to chip GPIOs: GP26, GP27, GP28 and GP29. On Raspberry Pi Pico, the first three of these are brought out to GPIO pins, and the fourth can be used to measure the VSYS voltage on the board.

The ADC’s fifth input channel is connected to a temperature sensor built into RP2040.
You can specify which ADC channel you're using by pin number, e.g.

```
adc = machine.ADC(26) # Connect to GP26, which is channel 0
```

or by channel,

```
adc = machine.ADC(4) # Connect to the internal temperature sensor
adc = machine.ADC(0) # Connect to channel 0 (GP26)
```

An example reading the fourth analogue-to-digital (ADC) converter channel, connected to the internal temperature sensor:

```
import machine
import utime

sensor_temp = machine.ADC(4)
conversion_factor = 3.3 / (65535)

while True:
    reading = sensor_temp.read_u16() * conversion_factor
    temperature = 27 - (reading - 0.706) / 0.001721
    print(temperature)
    utime.sleep(2)
```

### 3.4. Interrupts

You can set an IRQ like this:

```
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.

### 3.5. Multicore Support

Example usage:

```
import time, _thread, machine
def task(n, delay):
    led = machine.Pin(25, machine.Pin.OUT)
```
```python
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.

### 3.6. I2C

Example usage:

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

```python
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](https://github.com/raspberrypi/pico-micropython-examples/tree/master/i2c/i2c_without_freq.py) Lines 1 - 3

```python
from machine import I2C
i2c = I2C(8)  # 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>
<thead>
<tr>
<th>Function</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2C Frequency</td>
<td>400,000</td>
</tr>
<tr>
<td>I2C0 SCL</td>
<td>Pin 9</td>
</tr>
<tr>
<td>I2C0 SDA</td>
<td>Pin 8</td>
</tr>
<tr>
<td>I2C1 SCL</td>
<td>Pin 7</td>
</tr>
<tr>
<td>I2C1 SDA</td>
<td>Pin 6</td>
</tr>
</tbody>
</table>
3.7. SPI

Example usage:

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/spi/spi.py Lines 7 - 11

```python
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)
```

NOTE

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPI_BAUDRATE</td>
<td>1,000,000</td>
</tr>
<tr>
<td>SPI_POLARITY</td>
<td>0</td>
</tr>
<tr>
<td>SPI_PHASE</td>
<td>0</td>
</tr>
<tr>
<td>SPI_BITS</td>
<td>8</td>
</tr>
<tr>
<td>SPI_FIRSTBIT</td>
<td>MSB</td>
</tr>
<tr>
<td>SPI0_SCK</td>
<td>Pin 6</td>
</tr>
<tr>
<td>SPI0_MOSI</td>
<td>Pin 7</td>
</tr>
<tr>
<td>SPI0_MISO</td>
<td>Pin 4</td>
</tr>
<tr>
<td>SPI1_SCK</td>
<td>Pin 10</td>
</tr>
<tr>
<td>SPI1_MOSI</td>
<td>Pin 11</td>
</tr>
<tr>
<td>SPI1_MISO</td>
<td>Pin 8</td>
</tr>
</tbody>
</table>

3.8. PWM

Example of using PWM to fade an LED:


```python
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)

3.9. PIO Support

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 Raspberry Pi Pico C/C++ SDK book.

The Raspberry Pi Pico MicroPython introduces 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 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,
# 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 explicit exec.


```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
class rp2:
    @asm_pio
    def prog():
        pass

    @rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
def prog():
    pass

# 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)

# 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).

### 3.9.1. IRQ

There is support for PIO IRQs, e.g.


```python
import time
import rp2

def irq_test():
    wrap_target()
    nop() [31]
    nop() [31]
    nop() [31]
    nop() [31]
    irq(0)
    nop() [31]
    nop() [31]
    nop() [31]
    irq(1)
    nop() [31]
    wrap()  
```
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

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

10 import time
11 from machine import Pin
12 import rp2
13
14 @rp2.asm_pio()
15 def wait_pin_low():
16     wrap_target()
17     wait(0, pin, 0)
18     irq(block, rel(0))
19     wait(1, pin, 0)
20     wrap()
21
22 def handler(sm):
23     # Print a (wrapping) timestamp, and the state machine object.
24     print(time.ticks_ms(), sm)
25
26 # Instantiate StateMachine(0) with wait_pin_low program on Pin(16).
27 pin16 = Pin(16, Pin.IN, Pin.PULL_UP)
28 sm0 = rp2.StateMachine(0, wait_pin_low, in_base=pin16)
29 sm0.irq(handler)
30
31 # Instantiate StateMachine(1) with wait_pin_low program on Pin(17).
32 pin17 = Pin(17, Pin.IN, Pin.PULL_UP)
33 sm1 = rp2.StateMachine(1, wait_pin_low, in_base=pin17)
34 sm1.irq(handler)
35
36 # Start the StateMachine’s running.
37 sm0.active(1)
38 sm1.active(1)
39
40 # Now, when Pin(16) or Pin(17) is pulled low a message will be printed to the REPL.

3.9.2. WS2812 LED (NeoPixel)

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


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 @rp2.asm_pio(sideset_init=rp2.PIO.OUT_LOW, out_shiftdir=rp2.PIO.SHIFT_LEFT, autopull=True, pull_thresh=24)
11 def ws2812():
12     T1 = 2
13     T2 = 5
14     T3 = 3

22
# Example using PIO to create a UART TX interface

```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]
```

3.9. PIO Support

3.9.3. UART TX

A UART TX example,

# 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 initial 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.

#### 3.9.4. SPI

An SPI example.


```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(0x8) [1]
in_(pins, 1) .side(0x1)
jmp(x_dec, "bitloop") .side(0x1)
out(pins, 1) .side(0x8)
set(x, 6) .side(0x8) # Note this could be replaced with mov x, y for
```
programmable frame size

```python
    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()
```

```python
    23
class PIOSPI:
    24    def __init__(self, sm_id, pin_mosi, pin_miso, pin_sck, cpha=False, cpol=False, freq=1000000):
    25        assert not (cpol or cpha)
    26        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))
    27        self._sm.active(1)
    28
    29        # Note this code will die spectacularly cause we're not draining the RX FIFO
    30    def write_blocking(self, wdata):
    31        for b in wdata:
    32            self._sm.put(b << 24)
    33    def read_blocking(self, n):
    34        data = []
    35        for i in range(n):
    36            data.append(self._sm.get() & 0xff)
    37        return data
    38    def write_read_blocking(self, wdata):
    39        rdata = []
    40        for b in wdata:
    41            self._sm.put(b << 24)
    42            rdata.append(self._sm.get() & 0xff)
    43        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.

#### 3.9.5. PWM

A PWM example,

```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    @asm_pio(sideset_init=PIO.OUT_LOW)
    8    def pwm_prog():
    9        pull(noblock) .side(0)
   10    mov(x, osr) # Keep most recent pull data stashed in X, for recycling by noblock
   11    mov(y, isr) # ISR must be preloaded with PWM count max
```
3.9.6. Using \texttt{pioasm}

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

\begin{verbatim}
$ pioasm -o python input (output)
\end{verbatim}

For more information on \texttt{pioasm} see the \textit{Raspberry Pi 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

Thonny 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 of Thonny can 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/latest. This wheel file can be installed into Thonny 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.](image)
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,

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

see Figure 5.
4.1.2. Blinking the LED from Thonny

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


```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.
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 be 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.

### 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

You can then connect to Raspberry Pi Pico using,

$ rshell --buffer-size=512 -p /dev/ttyACM0
Connecting to /dev/ttyACM0 (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.

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

```python
# 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
```
# 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
fb = framebuf.FrameBuffer(buffer, 32, 32, framebuf.MONO_HLSB)

# 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()
# Blit the image from the framebuffer to the oled display

```python
oled.blit(fb, 96, 0)
```

# Add some text

```python
oled.text("Raspberry Pi", 5, 5)
oled.text("Pico", 5, 15)
```

# Finally update the oled display so the image & text is displayed

```python
oled.show()
```

## Bill of Materials

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadboard</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>Raspberry Pi Pico</td>
<td>1</td>
<td><a href="http://raspberrypi.org/">http://raspberrypi.org/</a></td>
</tr>
<tr>
<td>Monochrome 128x32 I2C OLED Display</td>
<td>1</td>
<td><a href="https://www.adafruit.com/product/931">https://www.adafruit.com/product/931</a></td>
</tr>
</tbody>
</table>

## Using a SH1106-based OLED graphics display

Display an image and text on I2C driven SH1106-based OLED graphics display such as the Pimoroni Breakout Garden 1.12” Mono OLED [https://shop.pimoroni.com/products/1-12-oled-breakout?variant=29421050757203](https://shop.pimoroni.com/products/1-12-oled-breakout?variant=29421050757203).

### Wiring information

See [Figure 8](#) for wiring instructions.
List of Files

A list of files with descriptions of their function;

i2c_1106oled_using_defaults.py

The example code.

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/i2c/1106oled/i2c_1106oled_using_defaults.py Lines 1 - 34

```
1 # Display Image & text on I2C driven SH1106 OLED display
2 from machine import I2C, ADC
3 from sh1106 import SH1106_I2C
4 import framebuf
5
6 WIDTH = 128          # oled display width
7 HEIGHT = 128          # oled display height
9 i2c = I2C(0)          # Init I2C using I2C0 defaults, SCL=Pin(GP9), SDA=Pin(GP8), freq=400000
11 print("I2C Address : " + hex(i2c.scan()[0]).upper()) # Display device address
11 print("I2C Configuration: " + str(i2c)) # Display I2C config
13
15 oled = SH1106_I2C(WIDTH, HEIGHT, i2c)          # Init oled display
16
17 buffer = bytearray(b"\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00")
19 fb = framebuf.FrameBuffer(buffer, 32, 32, framebuf.MONO_HLSB)
22
26 oled.blit(fb, 96, 0)  # Blit the image from the framebuffer to the oled display
27
29 # Add some text
30 oled.text("Raspberry Pi", 5, 5)
31 oled.text("Pico", 5, 15)
32
34 oled.show()          # Finally update the oled display so the image & text is displayed
```

i2c_1106oled_with_freq.py

The example code, explicitly sets a frequency.

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/i2c/1106oled/i2c_1106oled_with_freq.py Lines 1 - 33

```
```
HEIGHT = 32  # oled display height

def I2C(0, scl=Pin(9), sda=Pin(8), freq=200000):  # Init I2C using pins GP8 & GP9
    print("I2C Address : "+hex(i2c.scan()[0]).upper())  # Display device address
    print("I2C Configuration: "+str(i2c))  # Display I2C config
oled = SH1106_I2C(WIDTH, HEIGHT, i2c)  # Init oled display

Raspberry Pi Pico Python SDK

Using a SH1106-based OLED graphics display

sh1106.py

SH1106 Driver Obtained from https://github.com/robert-hh/SH1106

Pico MicroPython Examples: https://github.com/raspberrypi/pico-micropython-examples/tree/master/i2c/1106oled/sh1106.py Lines 1 - 227
Sample code sections

# Pin Map SPI

# 3V - xxxxxx - Vcc
# G  - xxxxxx - Gnd
# D7 - GPIO 13 - Din / MOSI fixed
# D5 - GPIO 14 - Clk / Sck fixed
# D8 - GPIO 4 - CS (optional, if the only connected device)
# D2 - GPIO 5 - D/C
# D1 - GPIO 2 - Res

# for CS, D/C and Res other ports may be chosen.

from machine import Pin, SPI
import sh1106

spi = SPI(1, baudrate=1000000)
display = sh1106.SH1106_SPI(128, 64, spi, Pin(5), Pin(2), Pin(4))
display.sleep(False)
display.fill(0)
display.text('Testing 1', 0, 0, 1)
display.show()

# Pin Map I2C

# 3V - xxxxxx - Vcc
# G  - xxxxxx - Gnd
# D2 - GPIO 5 - SCK / SCL
# D1 - GPIO 4 - DIN / SDA
# D0 - GPIO 16 - Res
# G  - xxxxxx - CS
# G  - xxxxxx - D/C

# Pin's for I2C can be set almost arbitrary

from machine import Pin, I2C
import sh1106

i2c = I2C(scl=Pin(5), sda=Pin(4), freq=400000)
display = sh1106.SH1106_I2C(128, 64, i2c, Pin(16), 0x3c)
display.sleep(False)
display.fill(0)
display.text('Testing 1', 0, 0, 1)
display.show()
class SH1106:

def __init__(self, width, height, external_vcc):
    self.width = width
    self.height = height
    self.external_vcc = external_vcc
    self.pages = self.height // 8

    buffer = bytearray(self.pages * self.width)
    fb = framebuf.FrameBuffer(buffer, self.width, self.height, 
                            framebuf.MVLSB)

    self.framebuf = fb

    # set shortcuts for the methods of framebuffer
    self.fill = fb.fill
    self.fill_rect = fb.fill_rect
    self.hline = fb.hline
    self.vline = fb.vline
    self.line = fb.line
    self.rect = fb.rect
    self.pixel = fb.pixel
    self.scroll = fb.scroll
    self.text = fb.text
    self.blit = fb.blit

    self.init_display()

    def init_display(self):
        self.reset()
        self.fill(0)
        self.poweron()
        self.show()

    def poweroff(self):
        self.write_cmd(_SET_DISP | 0x00)

    def poweron(self):
        self.write_cmd(_SET_DISP | 0x01)

    def rotate(self, flag, update=True):
        if flag:
            self.write_cmd(_SET_SEG_REMAP | 0x01)  # mirror display vertically
            self.write_cmd(_SET_SCAN_DIR | 0x08)  # mirror display hor.
        else:
            self.write_cmd(_SET_SEG_REMAP | 0x00)
            self.write_cmd(_SET_SCAN_DIR | 0x00)

        if update:
            self.show()

    def sleep(self, value):
        self.write_cmd(_SET_DISP | (not value))

    def contrast(self, contrast):
        self.write_cmd(_SET_CONTRAST)
        self.write_cmd(contrast)

    def invert(self, invert):
        self.write_cmd(_SET_NORM_INV | (invert & 1))

    def show(self):
        for page in range(self.height // 8):
            self.write_cmd(_SET_PAGE_ADDRESS | page)
            self.write_cmd(_LOW_COLUMN_ADDRESS | 2)
Using a SH1106-based OLED graphics display

```python
self.write_cmd(_HIGH_COLUMN_ADDRESS | 0)
self.write_data(self.buffer[
    self.width * page:self.width * page + self.width
])

def reset(self, res):
    if res is not None:
        res(1)
    time.sleep_ms(1)
    res(0)
    time.sleep_ms(20)
    res(1)
    time.sleep_ms(20)

class SH1106_I2C(SH1106):
    def __init__(self, width, height, i2c=None, addr=0x3c,
                 external_vcc=False):
        self.i2c = i2c
        self.addr = addr
        self.res = res
        self.temp = bytearray(2)
        if res is not None:
            res.init(res.OUT, value=1)
        super().__init__(width, height, external_vcc)

    def write_cmd(self, cmd):
        self.temp[0] = 0x80  # Co=1, D/C#=0
        self.temp[1] = cmd
        self.i2c.writeto(self.addr, self.temp)

    def write_data(self, buf):
        self.i2c.writeto(self.addr, b'\x40'+buf)

    def reset(self):
        super().reset(self.res)

class SH1106_SPI(SH1106):
    def __init__(self, width, height, spi, dc, res=None, cs=None,
                 external_vcc=False):
        self.rate = 10 * 1000 * 1000
        dc.init(dc.OUT, value=0)
        if res is not None:
            res.init(res.OUT, value=0)
        if cs is not None:
            cs.init(cs.OUT, value=1)
        self.spi = spi
        self.dc = dc
        self.res = res
        self.cs = cs
        super().__init__(width, height, external_vcc)

    def write_cmd(self, cmd):
        self.spi.init(baudrate=self.rate, polarity=0, phase=0)
        if self.cs is not None:
            self.cs(1)
        self.dc(0)
        self.cs(0)
        self.spi.write(bytearray([cmd]))
        self.cs(1)

    else:
        self.dc(0)
```

Using a SH1106-based OLED graphics display
def write_data(self, buf):
    self.spi.init(baudrate=self.rate, polarity=0, phase=0)
    if self.cs is not None:
        self.cs(1)
        self.dc(1)
        self.cs(0)
        self.spi.write(buf)
        self.cs(1)
    else:
        self.dc(1)
        self.spi.write(buf)

def reset(self):
    super().reset(self.res)

Bill of Materials

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadboard</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>Raspberry Pi Pico</td>
<td>1</td>
<td><a href="http://raspberrypi.org/">http://raspberrypi.org/</a></td>
</tr>
<tr>
<td>Monochrome 128x128 I2C OLED Display</td>
<td>1</td>
<td><a href="https://shop.pimoroni.com/products/1-12-oled-breakout?variant=29421050757203">https://shop.pimoroni.com/products/1-12-oled-breakout?variant=29421050757203</a></td>
</tr>
</tbody>
</table>

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 10 for wiring instructions.
# Example using PIO to drive a set of WS2812 LEDs.

```python
# 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)])

    # Display a pattern on the LEDs via an array of LED RGB values.
    # 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)

    # Display a pattern on the LEDs via an array of LED RGB values.
    # def pixels_set(i, color):

    # Display a pattern on the LEDs via an array of LED RGB values.
    # def pixels_fill(color):
    #     for i in range(len(ar)):
    #         pixels_set(i, color)
```

---

Raspberry Pi Pico Python SDK

Using PIO to drive a set of NeoPixel Ring (WS2812 LEDs)
```python
53 def color_chase(color, wait):
54     for i in range(NUM_LEDS):
55         pixels_set(i, color)
56         time.sleep(wait)
57     pixels_show()
58     time.sleep(0.2)
59
def wheel(pos):
60     # Input a value 0 to 255 to get a color value.
61     # The colours are a transition r - g - b - back to r.
62     if pos < 0 or pos > 255:
63         return (0, 0, 0)
64     if pos < 85:
65         return (255 - pos * 3, pos * 3, 0)
66     if pos < 170:
67         pos -= 85
68         return (0, 255 - pos * 3, pos * 3)
69         pos -= 170
70         return (pos * 3, 0, 255 - pos * 3)
71
def rainbow_cycle(wait):
72     for j in range(255):
73         for i in range(NUM_LEDS):
74             rc_index = (i * 256 // NUM_LEDS) + j
75             pixels_set(i, wheel(rc_index & 255))
76         pixels_show()
77         time.sleep(wait)
78
79 BLACK = (0, 0, 0)
80 RED = (255, 0, 0)
81 YELLOW = (255, 150, 0)
82 GREEN = (0, 255, 0)
83 CYAN = (0, 255, 255)
84 BLUE = (0, 0, 255)
85 PURPLE = (180, 0, 255)
86 WHITE = (255, 255, 255)
87 COLORS = (BLACK, RED, YELLOW, GREEN, CYAN, BLUE, PURPLE, WHITE)
88
def print("fills")
89 for color in COLORS:
90     pixels_fill(color)
91     pixels_show()
92     time.sleep(0.2)
93
def print("chases")
94 for color in COLORS:
95     color_chase(color, 0.01)
96
def print("rainbow")
97 rainbow_cycle(0)
```

### Bill of Materials

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadboard</td>
<td>1</td>
<td>generic part</td>
</tr>
<tr>
<td>NeoPixel Ring</td>
<td>1</td>
<td><a href="https://www.adafruit.com/product/1463">https://www.adafruit.com/product/1463</a></td>
</tr>
</tbody>
</table>

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