Connecting to the Internet with Raspberry Pi Pico W

Getting Raspberry Pi Pico W online with C/C++or MicroPython

Colophon

build-date: 2023-06-14
build-version: a6fe703-clean

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. Source code included in the documentation is Copyright © 2020-2023 Raspberry Pi Ltd (formerly Raspberry Pi (Trading) Ltd.) and licensed under the 3-Clause BSD license.

Legal disclaimer notice

Table of contents

Chapter 1. About Raspberry Pi Pico W

Raspberry Pi Pico W is a microcontroller board based on the Raspberry Pi RP2040 microcontroller chip.

Figure 1. The Raspberry Pi Pico W Rev3 board.
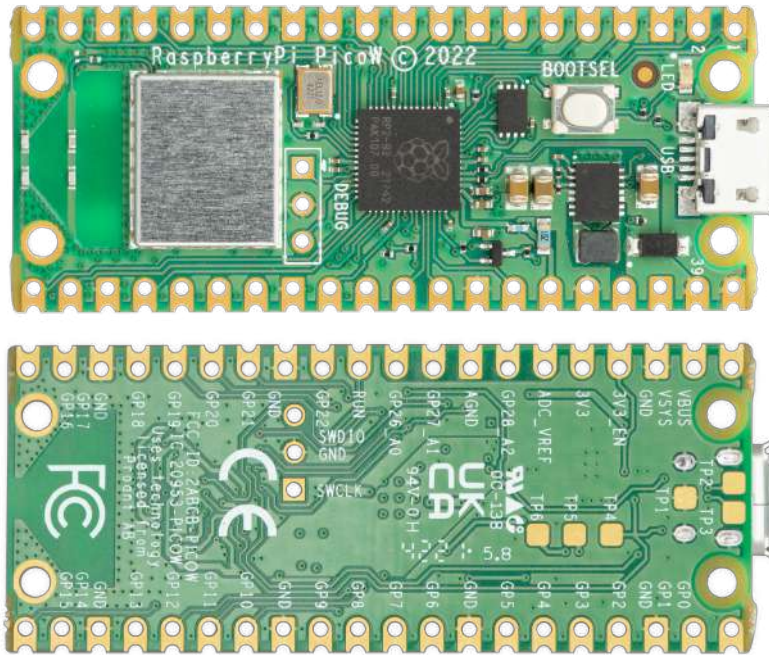
Raspberry Pi Pico W has been designed to be a low cost yet flexible development platform for RP2040, with the addition of a 2.4GHz wireless interface and the following key features:

- RP2040 microcontroller with 2MB of external flash

- On board 2.4GHz wireless (802.11n) interface

- On board Bluetooth interface

    - Support for Bluetooth LE Central and Peripheral roles

    - Support for Bluetooth Classic

- Micro-USB port for power and data (and for reprogramming the flash)

- 40-pin 21mm×51mm 'DIP' style 1mm thick PCB with 0.1 inch through-hole pins also with edge castellations

    - Exposes 26 multi-function 3.3V general purpose I/O (GPIO)

    - 23 GPIO are digital-only, with 3 more which also support ADC

    - Can be surface-mounted as a module

Apart from the addition of wireless networking, Raspberry Pi Pico W is very similar to Raspberry Pi Pico and, like all RP2040-based boards, shares the same development environment. If you have not previously used an RP2040-based board you can get started by reading Getting started with Raspberry Pi Pico if you're intending to use our C SDK, or Raspberry Pi Pico Python SDK if you're thinking about using MicroPython.

> **ⓘ NOTE**
>
> Full details of the Raspberry Pi Pico W can be found in the Raspberry Pi Pico W Datasheet.

> **ⓘ NOTE**
>
> By default `libcyw43` is licensed for non-commercial use, but Pico W users, and anyone else who builds their product around RP2040 and CYW43439, benefit from a free commercial-use license.

Chapter 2. Getting on the internet with the C SDK

Wireless support for Raspberry Pi Pico W has been added to the C/C++ SDK.

> ℹ **NOTE**
>
> If you have not previously used an RP2040-based board you can get started by reading Getting started with Raspberry Pi Pico, while further details about the SDK along with API-level documentation can be found in the Raspberry Pi Pico C/C++ SDK book.

2.1. Installing the SDK and examples

For full instructions on how to get started with the SDK see the Getting started with Raspberry Pi Pico book.

```
$ git clone https://github.com/raspberrypi/pico-sdk.git --branch master
$ cd pico-sdk
$ git submodule update --init
$ cd ..
$ git clone https://github.com/raspberrypi/pico-examples.git --branch master
```

> ⊖ **WARNING**
>
> If you have not initialised the `tinyusb` submodule in your `pico-sdk` checkout, then USB CDC serial, and other USB functions and example code, will not work as the SDK will contain no USB functionality. Similarly, if you have not initialised the `cyw43-driver` and `lwip` submodules in your checkout, then network-related functionality will not be enabled.

2.2. Building an SDK example

Building the SDK examples, and other wireless code, requires you to specify your network SSID and password, like this:

```
$ cd pico-examples
$ mkdir build
$ cd build
$ export PICO_SDK_PATH=../../pico-sdk
$ cmake -DPICO_BOARD=pico_w -DWIFI_SSID="Your Network" -DWIFI_PASSWORD="Your Password" ..
Using PICO_SDK_PATH from environment ('../../pico-sdk')
PICO_SDK_PATH is /home/pi/pico/pico-sdk
  .
  .
  .
-- Build files have been written to: /home/pi/pico/pico-examples/build
$
```

> The command line flags `-DWIFI_SSID="Your Network" -DWIFI_PASSWORD="Your Password"` are used by the `pico-examples` to set the SSID and password to the call to `cyw43_arch_wifi_connect_xxx()` to connect to your wireless network.

To then build a basic example for Raspberry Pi Pico W that will scan for nearby wireless networks, you can do:

```
$ cd pico_w/wifi_scan
$ make
PICO_SDK_PATH is /home/pi/pico-sdk
PICO platform is rp2040.
Build type is Release
PICO target board is pico_w.
    .
    .
    .
[100%] Built target picow_scan_test_background
$
```

Along with other targets, we have now built a binary called `picow_scan_test_background.uf2`, which can be dragged onto the RP2040 USB mass storage device

This binary will scan for wireless networks using the Raspberry Pi Pico W's wireless chip.

The fastest method to load software onto a RP2040-based board for the first time is by mounting it as a USB mass storage device. Doing this allows you to drag a file onto the board to program the flash. Go ahead and connect the Raspberry Pi Pico W to your Raspberry Pi using a micro-USB cable, making sure that you hold down the BOOTSEL button as you do so, to force it into USB mass storage mode.

If you are running the Raspberry Pi Desktop the Raspberry Pi Pico W should automatically mount as a USB mass storage device. From here, you can drag-and-drop the UF2 file onto the mass storage device. RP2040 will reboot, unmounting itself as a mass storage device, and start to run the flashed code.

By default the code will report its results via serial UART.

ⓘ **IMPORTANT**

> The default UART pins are configured on a per-board basis using board configuration files. The default Raspberry Pi Pico W UART TX pin (out from Pico W) is pin GP0, and the UART RX pin (in to Pico W) is pin GP1.

To see the text, you will need to enable UART serial communications on the Raspberry Pi host. 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?", then "Yes" when asked "Would you like the serial port hardware to be enabled?" You should see something like Figure 2.

Figure 2. Enabling a serial UART using `raspi-config` on the Raspberry Pi.
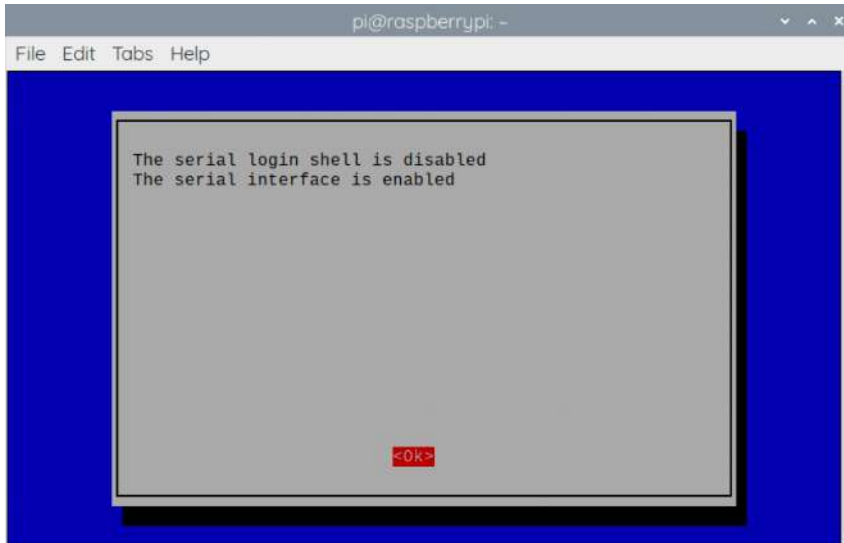
Leaving `raspi-config` you should choose "Yes" and reboot your Raspberry Pi to enable the serial port.

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

Raspberry Pi	Raspberry Pi Pico W
GND (Pin 14)	GND (Pin 3)
GPIO15 (UART_RX0, Pin 10)	GP0 (UART0_TX, Pin 1)
GPIO14 (UART_TX0, Pin 8)	GP1 (UART0_RX, Pin 2)

See Figure 3.

Figure 3. A Raspberry Pi 4 and the Raspberry Pi Pico with UART0 connected together.
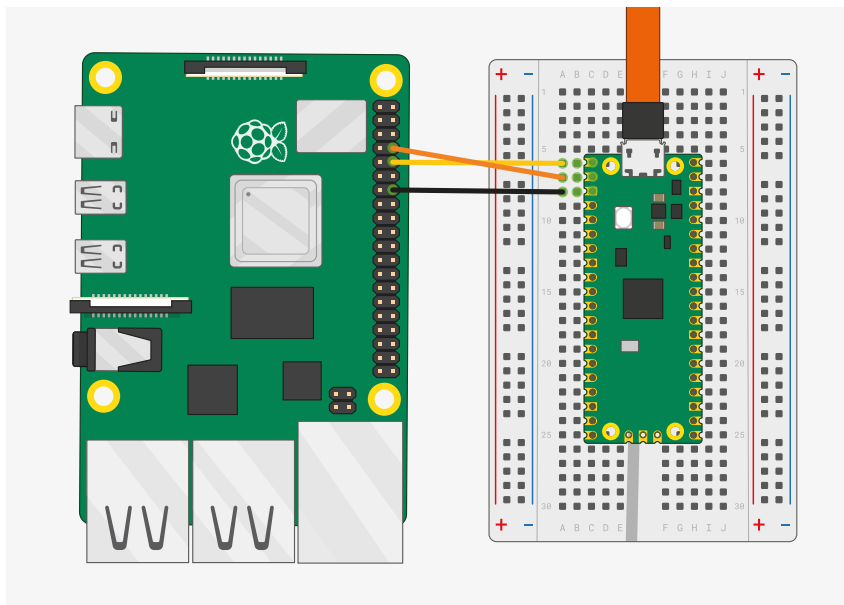
Once the two boards are wired together you should install `minicom` if you have not already done so:

```
$ sudo apt install minicom
```

and open the serial port:

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

You should see the results of our wireless scanning being printed to the console, see Figure 4.

💡 **TIP**

To exit minicom, use `CTRL-A` followed by `X`.

Figure 4. Results of our wireless scanning in the console

2.3. Creating your own project

Go ahead and create a directory to house your test project sitting alongside the `pico-sdk` directory,

```
$ ls -la
total 16
drwxr-xr-x   7 aa  staff   224  6 Apr 10:41 ./
drwx------@ 27 aa  staff   864  6 Apr 10:41 ../
drwxr-xr-x  10 aa  staff   320  6 Apr 09:29 pico-examples/
drwxr-xr-x  13 aa  staff   416  6 Apr 09:22 pico-sdk/
$ mkdir test
$ cd test
```

and then create a `test.c` file in the directory,

```
 1  #include <stdio.h>
 2
 3  #include "pico/stdlib.h"
 4  #include "pico/cyw43_arch.h"
 5
 6  char ssid[] = "A Network";①
 7  char pass[] = "A Password";②
 8
 9  int main() {
10      stdio_init_all();
11
```

```
12      if (cyw43_arch_init_with_country(CYW43_COUNTRY_UK)) {
13          printf("failed to initialise\n");
14          return 1;
15      }
16      printf("initialised\n");
17
18      cyw43_arch_enable_sta_mode();
19
20      if (cyw43_arch_wifi_connect_timeout_ms(ssid, pass, CYW43_AUTH_WPA2_AES_PSK, 10000)) {
21          printf("failed to connect\n");
22          return 1;
23      }
24      printf("connected\n");
25 }
```

1. Replace `A Network` with the SSID name of your wireless network.

2. Replace `A Password` with the password for your wireless network.

along with a `CMakeLists.txt` file,

```
cmake_minimum_required(VERSION 3.13)

include(pico_sdk_import.cmake)

project(test_project C CXX ASM)
set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)
pico_sdk_init()

add_executable(test
    test.c
)

pico_enable_stdio_usb(test 1)
pico_enable_stdio_uart(test 1)

pico_add_extra_outputs(test)

target_include_directories(test PRIVATE ${CMAKE_CURRENT_LIST_DIR} )

target_link_libraries(test pico_cyw43_arch_lwip_threadsafe_background pico_stdlib)
```

Then copy the `pico_sdk_import.cmake` file from the `external` folder in your `pico-sdk` installation to your test project folder,

```
$ cp ../pico-sdk/external/pico_sdk_import.cmake .
```

along with the `lwipopts.h` file needed by the lwIP stack.

```
$ cp ../pico-examples/pico_w/lwipopts_examples_common.h lwipopts.h
```

You should now have something that looks like this,

```
$ ls -la
total 32
drwxr-xr-x  6 aa  staff   192B 29 Jun 18:11 ./
drwxr-xr-x  7 aa  staff   224B 29 Jun 16:57 ../
-rw-r--r--@ 1 aa  staff   379B 29 Jun 18:10 CMakeLists.txt
-rw-rw-r--@ 1 aa  staff   3.3K 15 Jun 00:34 lwipopts.h
-rw-rw-r--@ 1 aa  staff   3.1K 15 Jun 00:34 pico_sdk_import.cmake
-rw-r--r--@ 1 aa  staff   427B 29 Jun 17:03 test.c
```

and can build it as we did before with our previous example in the last section.

```
$ mkdir build
$ cd build
$ export PICO_SDK_PATH=../../pico-sdk
$ cmake -DPICO_BOARD=pico_w ..
$ make
```

Afterwards unplug your Raspberry Pi Pico W from your computer if it is plugged in already. Then push and hold the BOOTSEL button while plugging it back into your computer. Then drag and drop the `test.uf2` binary onto the RPI-RP2 mass storage volume which will mount on your desktop.

Open the serial port:

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

and you should see the a message indicating that the Pico W has connected to your wireless network.

2.3.1. Going further

More information on the C SDK can be found in the Raspberry Pi Pico C/C++ SDK book. While information around lwIP can be found on the project's website. Example code can be found as part of the pico-examples Github repository.

2.4. Which hardware am I running on?

There is no direct method in the C SDK that can be called to allow software to discover whether it is running on a Raspberry Pi Pico or a Pico W. However, it is possible to indirectly discover the type of underlying hardware. If the board is powered via USB or `VSYS`, so `3v3_EN` is not pulled low externally, with GPIO25 low, ADC3 will be around 0V for Raspberry Pi Pico W and approximately 1/3 of `VSYS` for Raspberry Pi Pico.

Creating a `test.c` file,

```
 1  #include <stdio.h>
 2
 3  #include "pico/stdlib.h"
 4  #include "hardware/gpio.h"
 5  #include "hardware/adc.h"
 6
 7  int main() {
 8      stdio_init_all();
 9
10      adc_init();
```

```
11      adc_gpio_init(29);
12      adc_select_input(3);
13      const float conversion_factor = 3.3f / (1 << 12);
14      uint16_t result = adc_read();
15      printf("ADC3 value: 0x%03x, voltage: %f V\n", result, result * conversion_factor);
16
17      gpio_init(25);
18      gpio_set_dir(25, GPIO_IN);
19      uint value = gpio_get(25);
20      printf("GP25 value: %i", value);
21 }
```

alongside the following `CMakeLists.txt` file,

```
cmake_minimum_required(VERSION 3.13)

include(pico_sdk_import.cmake)

project(test_project C CXX ASM)
set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)
pico_sdk_init()

add_executable(test
    test.c
)

target_link_libraries(test pico_stdlib hardware_adc hardware_gpio)

pico_enable_stdio_usb(test 1)
pico_enable_stdio_uart(test 1)

pico_add_extra_outputs(test)
```

in a project directory gives this for Raspberry Pi Pico W,

```
ADC3 value: 0x01c, voltage: 0.022559 V
GP25 value: 0
```

and this for an original Raspberry Pi Pico board,

```
ADC3 value: 0x2cd, voltage: 0.577661 V
GP25 value: 0
```

Chapter 3. Getting on the internet with MicroPython

Wireless support for Raspberry Pi Pico W has been added to MicroPython. A pre-built binary, which can be downloaded from the MicroPython section of the documentation website, should serve most use cases and comes with `micropython-lib` pre-integrated into the binary.

> ℹ️ **NOTE**
>
> If you have not previously used an RP2040-based board you can get started by reading Raspberry Pi Pico Python SDK book.

3.1. Getting MicroPython for Raspberry Pi Pico W

Pre-built Binary

A pre-built binary of the latest MicroPython firmware is available from the MicroPython section of the documentation site.

The fastest way to get MicroPython is to download the pre-built release binary from the Documentation 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 Appendix A> to get the source code for MicroPython, which you can use to build your own MicroPython firmware binary.

3.2. Installing MicroPython on Raspberry Pi Pico W

Raspberry Pi Pico W 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 W 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 its 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 W.

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.

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.

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

> **ⓘ NOTE**
>
> The MicroPython port for RP2040 does not expose a REPL over a UART port by default, please see Raspberry Pi Pico Python SDK for more details of how to configure MicroPython to allow you to connect to the REPL over UART.

3.3. 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 W. 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:

```
MPY: soft reboot
MicroPython v1.18-524-g22474d25d on 2022-05-25; Raspberry Pi Pico W with RP2040
Type "help()" for more information.
>>>
```

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.

> **❶ NOTE**
>
> If you are working on an Apple Mac, 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.usbmodem` with a number appended to the end.
>
> If you don't already have a Terminal program installed you can install `minicom` using Homebrew:
>
> ```
> $ brew install minicom
> ```
>
> and connect to the board as below.
>
> ```
> $ minicom -b 115200 -o -D /dev/tty.usbmodem0000000000001
> ```
>
> Other Terminal applications like CoolTerm or Serial can also be used.

3.3.1. Using an integrated development environment (IDE)

The MicroPython port to Raspberry Pi Pico W and other RP2040-based boards works with commonly-used development environments. Thonny is the recommended editor. 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.

For full details on how to use the Thonny editor, see the section on using a development environment in the Raspberry Pi Pico Python SDK book.

3.3.2. Remote access via serial port

It's suggested you use the `mpremote` tool to access the device via the serial port.

```
$ pip install mpremote
$ mpremote connect list
/dev/cu.Bluetooth-Incoming-Port None 0000:0000 None None
/dev/cu.usbmodem22201 e660583883807e27 2e8a:0005 MicroPython Board in FS mode
$ mpremote connect port:/dev/cu.usbmodem22201
Connected to MicroPython at /dev/cu.usbmodem22201
Use Ctrl-] to exit this shell
>>>
```

With this you can run a script from your local machine directly on Raspberry Pi Pico W.

```
$ mpremote connect port:/dev/cu.usbmodem22201
$ mpremote run hello_world.py
```

> **ℹ NOTE**
>
> For more information on `mpremote` see the documentation.

3.4. The on-board LED

Unlike the original Raspberry Pi Pico, the on-board LED on Pico W is not connected to a pin on RP2040, but instead to a GPIO pin on the wireless chip. MicroPython has been modified accordingly. This means that you can now do:

```
>>> import machine
>>> led = machine.Pin("LED", machine.Pin.OUT)
>>> led.off()
>>> led.on()
```

or even:

```
>>> led.toggle()
```

to change the current state. However, if you now look at the `led` object:

```
>>> led
Pin(WL_GPIO0, mode=OUT)
>>>
```

You can also do the following:

```
>>> led = machine.Pin("LED", machine.Pin.OUT, value=1)
```

which will configure the `led` object, associate it with the on-board LED **and** turn the LED on.

> **ℹ NOTE**
>
> Full details of the Raspberry Pi Pico W can be found in the Raspberry Pi Pico W Datasheet. `WL_GPIO1` is connected to the `PS/SYNC` pin on the RT6154A to allow selection of different operating modes, while `WL_GPIO2` can be used to monitor USB `VBUS`.

3.5. Installing modules

You can use the `upip` tool to install modules that are not present in the default MicroPython installation.

```
>>> import upip
>>> upip.install("micropython-pystone_lowmem")
>>> import pystone_lowmem
>>> pystone_lowmem.main()
Pystone(1.2) time for 500 passes = 402ms
This machine benchmarks at 1243 pystones/second
```

```
>>>
```

3.6. Connecting to a wireless network

We're using the `network` library to talk to the wireless hardware:

```
1 import network
2 import time
3
4 wlan = network.WLAN(network.STA_IF)
5 wlan.active(True)
6 wlan.connect('Wireless Network', 'The Password')
7
8 while not wlan.isconnected() and wlan.status() >= 0:
9     print("Waiting to connect:")
10    time.sleep(1)
11
12 print(wlan.ifconfig())
```

although more correctly, you should wait for the connection to succeed or fail in your code, and handle any connection errors that might occur.

```
1 import time
2 import network
3
4 ssid = 'Wireless Network'
5 password = 'The Password'
6
7 wlan = network.WLAN(network.STA_IF)
8 wlan.active(True)
9 wlan.connect(ssid, password)
10
11 # Wait for connect or fail
12 max_wait = 10
13 while max_wait > 0:
14     if wlan.status() < 0 or wlan.status() >= 3:
15         break
16     max_wait -= 1
17     print('waiting for connection...')
18     time.sleep(1)
19
20 # Handle connection error
21 if wlan.status() != 3:
22     raise RuntimeError('network connection failed')
23 else:
24     print('connected')
25     status = wlan.ifconfig()
26     print( 'ip = ' + status[0] )
```

You can also disconnect and then connect to a different wireless network.

```
1 # Connect to another network
2 wlan.disconnect();
3 wlan.connect('Other Network', 'The Other Password')
```

For more information on the `network.WLAN` library see the library documentation.

3.6.1. Connection status codes

The values returned by the `wlan.status()` call are defined in the CYW43 wireless driver, and are passed directly through to user-code.

```
// Return value of cyw43_wifi_link_status
#define CYW43_LINK_DOWN        (0)
#define CYW43_LINK_JOIN        (1)
#define CYW43_LINK_NOIP        (2)
#define CYW43_LINK_UP          (3)
#define CYW43_LINK_FAIL        (-1)
#define CYW43_LINK_NONET       (-2)
#define CYW43_LINK_BADAUTH     (-3)
```

3.6.2. Setting the country

By default, the country setting for the wireless network is unset. This means that the driver will use a default world-wide safe setting, which may mean some channels are unavailable.

```
>>> import rp2
>>> rp2.country()
'\x00\x00'
>>>
```

This can cause problems on some wireless networks. If you find that your Raspberry Pi Pico W does not connect to your wireless network you may want to try setting the country code, e.g.

```
>>> rp2.country('GB')
```

3.6.3. Power-saving mode

By default the wireless chip will active power-saving mode when it is idle, which might lead it to being less responsive. If you are running a server or need more responsiveness, you can change this by toggling the power mode.

```
1 import network
2
3 wlan = network.WLAN(network.STA_IF)
4 wlan.active(True)
5 wlan.config(pm = 0xa11140)
```

3.7. The MAC address

The MAC is stored in the wireless chip OTP.

```python
 1 import network
 2 import ubinascii
 3
 4 wlan = network.WLAN(network.STA_IF)
 5 wlan.active(True)
 6 mac = ubinascii.hexlify(network.WLAN().config('mac'),':').decode()
 7 print(mac)
 8
 9 # Other things you can query
10 print(wlan.config('channel'))
11 print(wlan.config('essid'))
12 print(wlan.config('txpower'))
```

> ℹ **NOTE**
>
> We have to set the wireless active (which loads the firmware) before we can get the MAC address.

3.8. Making HTTP requests

You can take a low-level approach to HTTP requests using raw sockets, or a high-level approach using the `urequests` library.

3.8.1. HTTP with sockets

```python
 1 # Connect to network
 2 import network
 3
 4 wlan = network.WLAN(network.STA_IF)
 5 wlan.active(True)
 6 wlan.connect('Wireless Network', 'The Password')
 7
 8 # Should be connected and have an IP address
 9 wlan.status() # 3 == success
10 wlan.ifconfig()
11
12 # Get IP address for google.com
13 import socket
14 ai = socket.getaddrinfo("google.com", 80)
15 addr = ai[0][-1]
16
17 # Create a socket and make a HTTP request
18 s = socket.socket()
19 s.connect(addr)
20 s.send(b"GET / HTTP/1.0\r\n\r\n")
21
22 # Print the response
23 print(s.recv(512))
```

3.8.2. HTTP with urequests

It is much simpler to use the `urequests` library to make an HTTP connection.

```
 1  # Connect to network
 2  import network
 3  wlan = network.WLAN(network.STA_IF)
 4  wlan.active(True)
 5  wlan.connect('Wireless Network', 'The Password')
 6
 7  # Make GET request
 8  import urequests
 9  r = urequests.get("http://www.google.com")
10  print(r.content)
11  r.close()
```

Support has been added for redirects.

```
1  import urequests
2  r = urequests.get("http://www.raspberrypi.com")
3  print(r.status_code) # redirects to https
4  r.close()
```

ℹ️ **NOTE**

HTTPS works, but you should be aware that SSL verification is currently disabled.

The `urequests` library comes with limited JSON support.

```
>>> r = urequests.get("http://date.jsontest.com")
>>> r.json()
{'milliseconds_since_epoch': 1652188199441, 'date': '05-10-2022', 'time': '01:09:59 PM'}
>>>>
```

For more information on `urequests` see the library documentation.

🛑 **IMPORTANT**

You must close the returned response object after making a request using the `urequests` library using `response.close()`. If you do not, the object will not be garbage-collected, and if the request is being made inside a loop this will quickly lead to a crash.

3.8.3. Ensuring robust connections

This partial example illustrates a more robust approach to connecting to a network using `urequests`.

```
1  import time
2  import network
3  import urequests as requests
4
5  ssid = 'A Network'
6  password = 'The Password'
```

```python
 7
 8 wlan = network.WLAN(network.STA_IF)
 9 wlan.active(True)
10 wlan.connect(ssid, password)
11
12 # Wait for connect or fail
13 max_wait = 10
14 while max_wait > 0:
15     if wlan.status() < 0 or wlan.status() >= 3:
16         break
17     max_wait -= 1
18     print('waiting for connection...')
19     time.sleep(1)
20
21 # Handle connection error
22 if wlan.status() != 3:
23     raise RuntimeError('network connection failed')
24 else:
25     print('connected')
26     status = wlan.ifconfig()
27     print( 'ip = ' + status[0] )
28
29 while True:
30
31     # Do things here, perhaps measure something using a sensor?
32
33     # ...and then define the headers and payloads
34     headers = ...
35     payload = ...
36
37     # Then send it in a try/except block
38     try:
39         print("sending...")
40         response = requests.post("A REMOTE END POINT", headers=headers, data=payload)
41         print("sent (" + str(response.status_code) + "), status = " + str(wlan.status()) )
42         response.close()
43     except:
44         print("could not connect (status =" + str(wlan.status()) + ")")
45         if wlan.status() < 0 or wlan.status() >= 3:
46             print("trying to reconnect...")
47             wlan.disconnect()
48             wlan.connect(ssid, password)
49             if wlan.status() == 3:
50                 print('connected')
51             else:
52                 print('failed')
53
54     time.sleep(5)
```

Here we handle the possibility that we lose connection to our wireless network and then will seek to reconnect.

3.9. Building HTTP servers

You can build synchronous or asynchronous web servers.

3.9.1. A simple server for static pages

You can use the `socket` library to build a simple web server.

```python
1  import network
2  import socket
3  import time
4
5  from machine import Pin
6
7  led = Pin(15, Pin.OUT)
8
9  ssid = 'A Network'
10 password = 'A Password'
11
12 wlan = network.WLAN(network.STA_IF)
13 wlan.active(True)
14 wlan.connect(ssid, password)
15
16 html = """<!DOCTYPE html>
17 <html>
18     <head> <title>Pico W</title> </head>
19     <body> <h1>Pico W</h1>
20         <p>Hello World</p>
21     </body>
22 </html>
23 """
24
25 # Wait for connect or fail
26 max_wait = 10
27 while max_wait > 0:
28     if wlan.status() < 0 or wlan.status() >= 3:
29         break
30     max_wait -= 1
31     print('waiting for connection...')
32     time.sleep(1)
33
34 # Handle connection error
35 if wlan.status() != 3:
36     raise RuntimeError('network connection failed')
37 else:
38     print('connected')
39     status = wlan.ifconfig()
40     print( 'ip = ' + status[0] )
41
42 # Open socket
43 addr = socket.getaddrinfo('0.0.0.0', 80)[0][-1]
44
45 s = socket.socket()
46 s.bind(addr)
47 s.listen(1)
48
49 print('listening on', addr)
50
51 # Listen for connections
52 while True:
53     try:
54         cl, addr = s.accept()
55         print('client connected from', addr)
56         cl_file = cl.makefile('rwb', 0)
57         while True:
58             line = cl_file.readline()
```

```
59              if not line or line == b'\r\n':
60                  break
61          response = html
62          cl.send('HTTP/1.0 200 OK\r\nContent-type: text/html\r\n\r\n')
63          cl.send(response)
64          cl.close()
65
66      except OSError as e:
67          cl.close()
68          print('connection closed')
```

ℹ️ **NOTE**

> This example is synchronous, for more robust request handling you should implement the server to handle requests asynchronously.

3.9.2. Controlling an LED via a web server

Going further, we can implement a RESTful web server that will allow us to control an LED.

Figure 5. The Raspberry Pi Pico W with an LED on GP15.
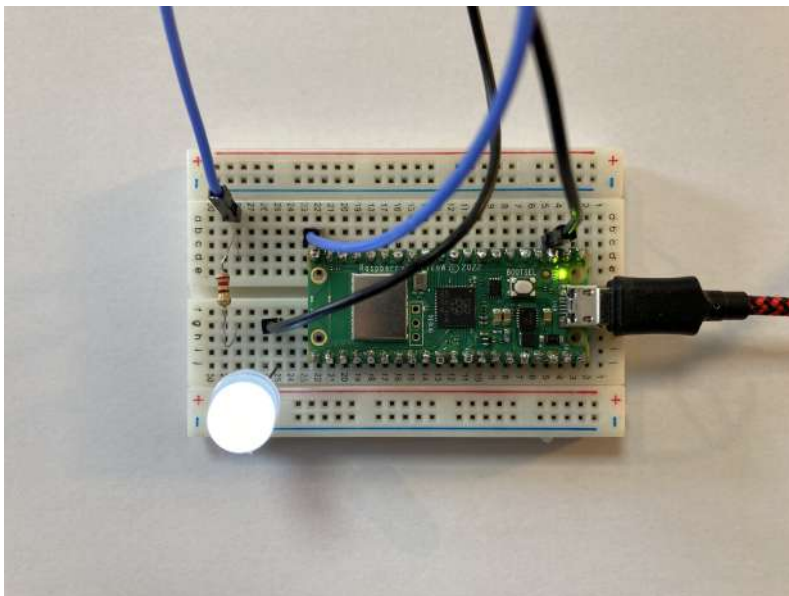
Connecting an LED to GP15 we can turn the LED on and off by using HTTP GET. We can do this by going to http://192.168.1.X/light/on to turn the LED on, and http://192.168.1.X/light/off to turn the LED off, in our web browser; where 192.168.1.X is the IP address of our Pico W, which will be printed in the console after it connects to the network.

```
 1 import network
 2 import socket
 3 import time
 4
 5 from machine import Pin
 6
 7 led = Pin(15, Pin.OUT)
 8
 9 ssid = 'A Network'
10 password = 'A Password'
11
12 wlan = network.WLAN(network.STA_IF)
13 wlan.active(True)
```

```
14 wlan.connect(ssid, password)
15
16 html = """<!DOCTYPE html>
17 <html>
18     <head> <title>Pico W</title> </head>
19     <body> <h1>Pico W</h1>
20         <p>%s</p>
21     </body>
22 </html>
23 """
24
25 # Wait for connect or fail
26 max_wait = 10
27 while max_wait > 0:
28     if wlan.status() < 0 or wlan.status() >= 3:
29         break
30     max_wait -= 1
31     print('waiting for connection...')
32     time.sleep(1)
33
34 # Handle connection error
35 if wlan.status() != 3:
36     raise RuntimeError('network connection failed')
37 else:
38     print('connected')
39     status = wlan.ifconfig()
40     print( 'ip = ' + status[0] )
41
42 # Open socket
43 addr = socket.getaddrinfo('0.0.0.0', 80)[0][-1]
44
45 s = socket.socket()
46 s.bind(addr)
47 s.listen(1)
48
49 print('listening on', addr)
50
51 # Listen for connections
52 while True:
53     try:
54         cl, addr = s.accept()
55         print('client connected from', addr)
56         request = cl.recv(1024)
57         print(request)
58
59         request = str(request)
60         led_on = request.find('/light/on')
61         led_off = request.find('/light/off')
62         print( 'led on = ' + str(led_on))
63         print( 'led off = ' + str(led_off))
64
65         if led_on == 6:
66             print("led on")
67             led.value(1)
68             stateis = "LED is ON"
69
70         if led_off == 6:
71             print("led off")
72             led.value(0)
73             stateis = "LED is OFF"
74
75         response = html % stateis
76
77         cl.send('HTTP/1.0 200 OK\r\nContent-type: text/html\r\n\r\n')
```

```
78        cl.send(response)
79        cl.close()
80
81    except OSError as e:
82        cl.close()
83        print('connection closed')
```

Running the code, we can see the response in our browser.
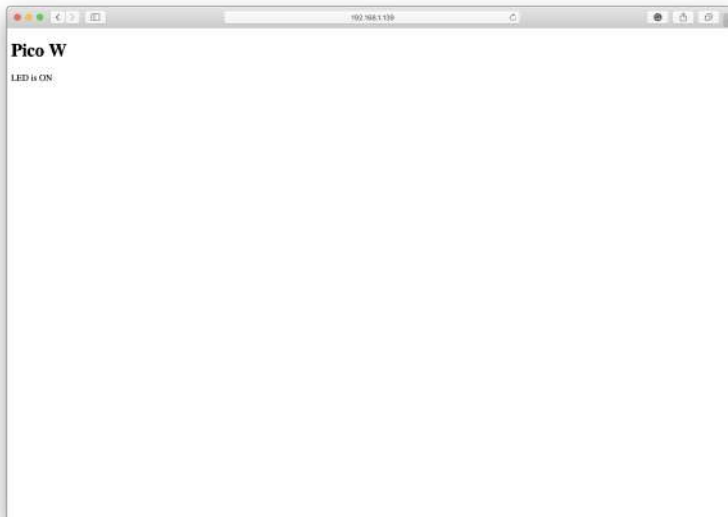
3.9.3. An asynchronous web server

We can use the `uasyncio` module to implement the same server, but in this case it will handle HTTP requests asynchronously rather than blocking.

```
 1 import network
 2 import socket
 3 import time
 4
 5 from machine import Pin
 6 import uasyncio as asyncio
 7
 8 led = Pin(15, Pin.OUT)
 9 onboard = Pin("LED", Pin.OUT, value=0)
10
11 ssid = 'A Network'
12 password = 'A Password'
13
14 html = """<!DOCTYPE html>
15 <html>
16     <head> <title>Pico W</title> </head>
17     <body> <h1>Pico W</h1>
18         <p>%s</p>
19     </body>
20 </html>
21 """
22
23 wlan = network.WLAN(network.STA_IF)
24
25 def connect_to_network():
```

```
26        wlan.active(True)
27        wlan.config(pm = 0xa11140)   # Disable power-save mode
28        wlan.connect(ssid, password)
29
30        max_wait = 10
31        while max_wait > 0:
32            if wlan.status() < 0 or wlan.status() >= 3:
33                break
34            max_wait -= 1
35            print('waiting for connection...')
36            time.sleep(1)
37
38        if wlan.status() != 3:
39            raise RuntimeError('network connection failed')
40        else:
41            print('connected')
42            status = wlan.ifconfig()
43            print('ip = ' + status[0])
44
45    async def serve_client(reader, writer):
46        print("Client connected")
47        request_line = await reader.readline()
48        print("Request:", request_line)
49        # We are not interested in HTTP request headers, skip them
50        while await reader.readline() != b"\r\n":
51            pass
52
53        request = str(request_line)
54        led_on = request.find('/light/on')
55        led_off = request.find('/light/off')
56        print( 'led on = ' + str(led_on))
57        print( 'led off = ' + str(led_off))
58
59        stateis = ""
60        if led_on == 6:
61            print("led on")
62            led.value(1)
63            stateis = "LED is ON"
64
65        if led_off == 6:
66            print("led off")
67            led.value(0)
68            stateis = "LED is OFF"
69
70        response = html % stateis
71        writer.write('HTTP/1.0 200 OK\r\nContent-type: text/html\r\n\r\n')
72        writer.write(response)
73
74        await writer.drain()
75        await writer.wait_closed()
76        print("Client disconnected")
77
78    async def main():
79        print('Connecting to Network...')
80        connect_to_network()
81
82        print('Setting up webserver...')
83        asyncio.create_task(asyncio.start_server(serve_client, "0.0.0.0", 80))
84        while True:
85            onboard.on()
86            print("heartbeat")
87            await asyncio.sleep(0.25)
88            onboard.off()
89            await asyncio.sleep(5)
```

```
90
91 try:
92     asyncio.run(main())
93 finally:
94     asyncio.new_event_loop()
```

3.10. Running iperf

You can install `iperf` using the `upip` tool:

```
>>> import network
>>> wlan = network.WLAN(network.STA_IF)
>>> wlan.active(True)
>>> wlan.connect('Wireless Network', 'The Password')
>>> import upip
>>> upip.install("uiperf3")
```

and start an `iperf3` client.

> **ℹ️ NOTE**
>
> The `iperf` server should be running on another machine.

```
>>> import uiperf3
>>> uiperf3.client('10.3.15.xx)

CLIENT MODE: TCP sending
Connecting to ('10.3.15.234', 5201)
Interval            Transfer      Bitrate
  0.00-1.00   sec  48.4 KBytes    397 Kbits/sec
  1.00-2.00   sec  48.4 KBytes    397 Kbits/sec
  2.00-3.00   sec  80.5 KBytes    659 Kbits/sec
  3.00-4.00   sec   100 KBytes    819 Kbits/sec
  4.00-5.00   sec   103 KBytes    845 Kbits/sec
  5.00-6.00   sec  22.7 KBytes    186 Kbits/sec
  6.00-7.00   sec  0.00 Bytes  0.00 bits/sec
  7.00-8.00   sec  0.00 Bytes  0.00 bits/sec
  8.00-9.00   sec  45.3 KBytes    371 Kbits/sec
  9.00-10.00  sec  89.1 KBytes    729 Kbits/sec
 10.00-10.01  sec  0.00 Bytes  0.00 bits/sec
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  0.00-10.01  sec   538 KBytes    440 Kbits/sec  sender
>>>
```

3.11. Which hardware am I running on?

There is no direct method for software written in MicroPython to discover whether it is running on a Raspberry Pi Pico or a Pico W by looking at the hardware. However, you can tell indirectly by looking to see if network functionality is included in your particular MicroPython firmware:

```
1 import network
2 if hasattr(network, "WLAN"):
3     # the board has WLAN capabilities
```

Alternatively, you can inspect the MicroPython firmware version to check whether it was compiled for Raspberry Pi Pico or for Pico W using the `sys` module.

```
>>> import sys
>> sys.implementation
(name='micropython', version=(1, 19, 1), _machine='Raspberry Pi Pico W with RP2040', _mpy=4102)
```

So `if 'Pico W' in sys.implementation._machine` can be used to detect whether your firmware was compiled for Pico W.

Chapter 4. About Bluetooth

Raspberry Pi Pico W onboard Bluetooth interface has support for both Bluetooth LE Central and Peripherals roles, along with support for Bluetooth Classic, and is configurable so you can enable both LE and Classic at the same time, or either of them individually.

> ℹ **NOTE**

> Full details of supported Bluetooth protocols and profiles are availabile on the Blue Kitchen BTStack Github repository. In addition to the standard BTstack licensing terms, a supplemental licence which covers commercial use of BTstack with Raspberry Pi Pico W or Raspberry Pi Pico WH is provided.

4.1. More about Bluetooth LE

Bluetooth LE divides the world into peripheral and central devices. Peripheral devices are things like sensors — they're typically small, low-powered, and resource-constrained. Central devices are things like mobile phones or laptops, although these often also operate in peripheral mode.

> ℹ **NOTE**

> The Bluetooth LE specification is a sprawling mess of interlocking documents that runs to thousands of pages; the core standards document is over 2,700 pages just on its own. Proceed with caution.

Peripherals can operate in two modes: either by broadcasting, or when directly connected to a central device. The broadcast mechanism is one of the big differences between Bluetooth LE and "classic" Bluetooth. Using it, data can be sent out by the peripheral to any device within range.

This means that a Bluetooth LE peripheral device doesn't necessarily need to be paired — in Bluetooth LE we'd speak of it as "connected", rather than "paired" as we did with Bluetooth 2.1 — to a central device to transfer data. In broadcast mode, the peripheral will periodically send out advertising packets, available to anyone that's looking for them, for devices that are acting as "observers".

The standard advertising packet describes the broadcasting device and its capabilities, but it is also capable of including custom information — sensor data for instance — that you might want to broadcast.

Broadcasting data from your peripheral is a good choice if you're building something like a weather station, where the data isn't sensitive. There is, however, no provision for security when broadcasting, so for personal data, the central device should connect to the peripheral.

Connections are exclusive. This means that a peripheral cannot be connected to more than one central device at a time. When a central device connects to a peripheral, the peripheral will stop advertising itself. Other devices will not be able to see it, or connect to it, until the first connection is terminated. While a peripheral can only be connected to one central device, a central device can be connected to more than peripheral at the same time.

If you need to exchange data between the peripheral and the central device, then you need to establish a connection between the two devices.

4.1.1. Protocols and profiles

On top of the protocols that make up the Bluetooth LE standard, the specification offers what are called "profiles". These are either the basic modes of operation needed by all Bluetooth LE devices, for instance the Generic Access Profile (GAP) and Generic Attribute Profile (GATT), or profiles covering specific use cases such as the Heart Rate Profile.

4.1.2. The GAP

The Generic Access Profile (GAP) is the profile that defines roles for devices, including the peripheral and central roles we mentioned in the last section, alongside advertising and discovery.

There are two ways to advertise data using GAP: advertising data and scan response packets. While both packets use the same payload format, and consist of up to 31 bytes of data, only the advertising data packet is mandatory. It is sent out at a preset advertising interval; the longer the interval, the less power is used. On receipt, listening devices can request the scan response packet with additional data if it exists. For instance, using custom advertisement data in the broadcast packets is how Bluetooth beacon standards are implemented.

Once a connection with the peripheral has been made, you will use GATT services and characteristics to communicate with the peripheral device, and advertising will stop until the connection is terminated.

4.1.3. The GATT

The Generic Attribute Profile (GATT) defines how Bluetooth LE transfers data back and forth between peripheral and central devices. It defines profiles, which are collections of services. Each service has characteristics which contain data.

Roles change when moving from GAP to GATT. GATT defines two roles: client and server.

It may seem counterintuitive, but peripheral devices are known as GATT Servers, while the (generally more powerful) central devices are GATT Clients. Think of it this way. The server has data, and the client wants data. All connections between the devices are started by the client.

After connecting the client, we can get a list of services offered by the server. Before connecting, the central device has a potentially incomplete list of services from the advertising data.

4.1.4. Services and characteristics

Services are used to break up the data into logically associated chunks, and consists of a collection of characteristics. Characteristics are the containers that hold the data associated with a service. Both services and characteristics are identified by ua nique identifier, known as a UUID. See Section 4.1.5.

Characteristics contain at least two attributes: a characteristic declaration which contains metadata about the data, and the characteristic value which contains the data itself. The characteristic can also contain additional descriptors to expand on the metadata. Together, the declaration, value and any optional descriptors form a bundle than make up a characteristic.

Characteristics can be defined as read or write. Characteristics are read by the client using a read request, with the returned value of the request being the characteristic value. Characteristic values can be written using a write request. The server returns a confirmation after the value is written. There is an additional write property called write command. When a characteristic value is written with a write command, the server does not send any response back to the client. Write command is sometimes called write without response.

Two additional properties are notify and indicate. Both of these are server-initiated communication. A client subscribes to be notified when a characteristic's value changes. When a change occurs, the server notifies the client by sending the new value. An indication is similar to a notification, except that the client must acknowledge the receipt of the indication.

Characteristics can have multiple properties. For example one characteristic could allow read, write, write command, and notify.

4.1.5. UUIDs

Bluetooth uses Universally Unique Identifiers (UUIDs) for many things including services and characteristics. Bluetooth services that have been approved by the Bluetooth Special Interest Group are assigned 16-bit UUIDs. All other services and characteristics must use 128-bit UUIDs. 128-bit UUIDs can be generated with tools such as uuidgen, e.g.

```
$ uuidgen
437121E5-A6F0-43F9-8F8F-4AB73D6CC3EB
```

It's fine to reuse UUIDs if there are services and characteristics that meet your needs. If your making your own services, use 128-bit UUIDs. See the Bluetooth developer site for more information.

Chapter 5. Working with Bluetooth and the C SDK

Bluetooth support for Raspberry Pi Pico W has been added to the C/C++ SDK.

Abbreviated instructions for installing the SDK and examples can be found in Section 2.1. For full instructions on how to get started with the SDK, see the Getting started with Raspberry Pi Pico book.

> **ℹ NOTE**
>
> If you have not previously used an RP2040-based board you can get started by reading Getting started with Raspberry Pi Pico, while further details about the SDK, along with API-level documentation, can be found in the Raspberry Pi Pico C/C++ SDK book.

> **⊖ WARNING**
>
> If you have not initialised the `tinyusb` submodule in your `pico-sdk` checkout, then USB CDC serial, and other USB functions and example code, will not work, as the SDK will contain no USB functionality. Similarly, if you have not initialised the `cyw43-driver` and `lwip` submodules in your checkout, then network- and bluetooth-related functionality will not be enabled.

5.1. An example Bluetooth service

A standalone Bluetooth example, without all the common example build infrastructure, is available in the `pico-examples` GitHub repository.

> **ℹ NOTE**
>
> The standalone example code lives in the pico-examples GitHub repository.

Full details of supported Bluetooth protocols and profiles are Blue Kitchen BTStack Github repository.

5.1.1. Creating a temperature service peripheral

The standalone example implements a temperature service.

```
1 PRIMARY_SERVICE, GAP_SERVICE
2 CHARACTERISTIC, GAP_DEVICE_NAME, READ, "picow_temp"
3
4 PRIMARY_SERVICE, GATT_SERVICE
5 CHARACTERISTIC, GATT_DATABASE_HASH, READ,
6
7 PRIMARY_SERVICE, ORG_BLUETOOTH_SERVICE_ENVIRONMENTAL_SENSING
8 CHARACTERISTIC, ORG_BLUETOOTH_CHARACTERISTIC_TEMPERATURE, READ | NOTIFY | INDICATE | DYNAMIC,
```

To build this example you should:

```
$ git clone https://github.com/raspberrypi/pico-sdk.git --branch master
$ cd pico-sdk
$ git submodule update --init
$ cd ..
$ git clone https://github.com/raspberrypi/pico-examples.git --branch master
$ cd pico-examples
$ mkdir build
$ cd build
$ export PICO_SDK_PATH=../../pico-sdk
$ cmake -DPICO_BOARD=pico_w  ..
Using PICO_SDK_PATH from environment ('../../pico-sdk')
PICO_SDK_PATH is /home/pi/pico/pico-sdk
   .
   .
   .
-- Build files have been written to: /home/pi/pico/pico-examples/build
$ cd pico_w/bt/standalone
$ make
```

Along with other targets, we have now built two binaries called `picow_ble_temp_sensor.uf2` and `picow_ble_temp_reader.uf2`, which can be dragged onto the RP2040 USB mass-storage device.

The fastest method to load software onto an RP2040-based board for the first time is by mounting it as a USB mass-storage device. Doing this allows you to drag a file onto the board to program the flash memory. Go ahead and connect the Raspberry Pi Pico W to your Raspberry Pi using a micro-USB cable, making sure that you hold down the BOOTSEL button as you do so, to force it into USB mass-storage mode.

If you are running the Raspberry Pi Desktop, the Raspberry Pi Pico W should automatically mount as a USB mass-storage device. From here, you can drag and drop the UF2 file onto the mass-storage device. RP2040 will reboot, unmounting itself as a mass-storage device, and start to run the flashed code.

If you connect to the Bluetooth device using a scanner application on your phone (see Figure 7), you should see a service entry in the "Environmental Sensing" section. Beneath this, you'll find a temperature section. You should be able to read the temperature or subscribe for notifications when the temperature value changes.

Figure 7. The `picow_temp` peripheral in the Punch Through LightBlue app; advertising data (left), services (middle), and temperature service (right).
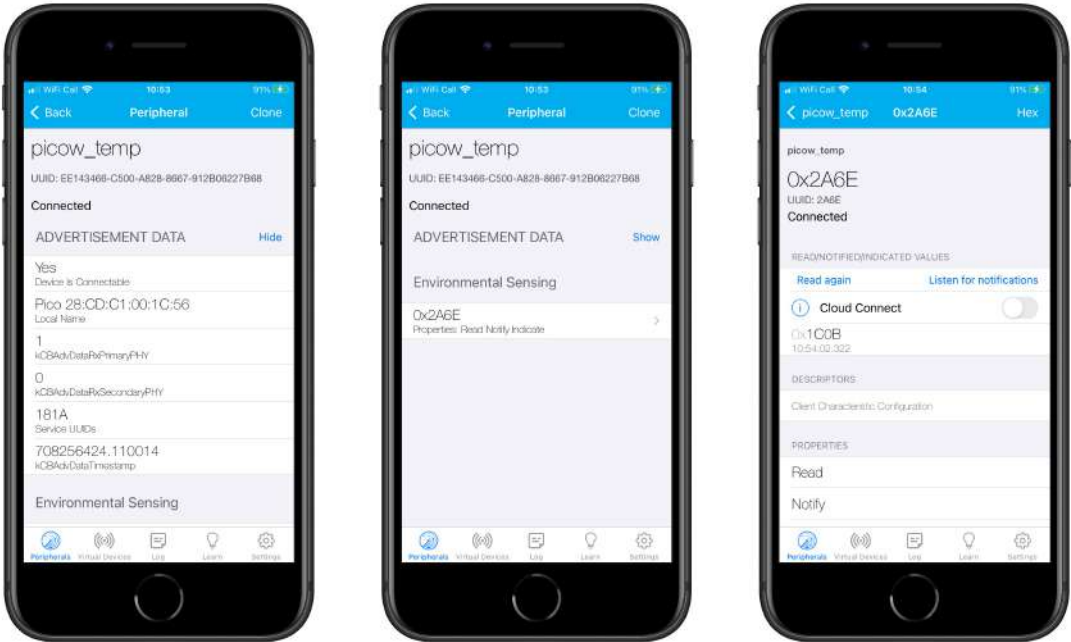
The temperature will show up as a hexadecimal number, e.g. `0x4B0B`. This number is a two-byte little-endian

representation of the temperature multiplied by 100. To get the value back in centigrade, you will need to flip this representation to big-endian (e.g. 0x4B0B becomes 0x0B4B), convert the value to decimal (e.g. 0x0B4B becomes 2891), and then divide by 100 to get the value in centigrade (here it would be 28.91°C).

5.2. Availability of other example code

More example code is available from the pico-examples GitHub repository. These examples are for the Pico W, and are only built when `PICO_BOARD=pico_w` is passed to CMake.

ⓘ NOTE

The examples in the `pico-examples` repository are taken from the Blue Kitchen Bluetooth stack examples.

By default, the Bluetooth examples are only built in one "mode" only (`background`, `poll`, or `freertos`), with the default being `background`. This can be changed by passing a mode to CMake when building on the command line, e.g. `-DBTSTACK_EXAMPLE_TYPE=poll`.

ⓘ NOTE

FreeRTOS versions can only be built if `FREERTOS_KERNEL_PATH` is defined.

Chapter 6. Working with Bluetooth in MicroPython

🛈 **IMPORTANT**

Make sure you have the latest version of MicroPython installed, with Bluetooth support enabled. Until Bluetooth functionality has been upstreamed, a pre-built binary will be availble from the MicroPython section of the documentation site.

Bluetooth support for Raspberry Pi Pico W has been added to MicroPython. A pre-built binary, which can be downloaded from the MicroPython section of this documentation website, should serve most use cases and comes with `micropython-lib` pre-integrated into the binary.

Pre-built Binary

A pre-built binary of the latest MicroPython firmware is available from the MicroPython section of the Raspberry Pi documentation site. See Section 3.2 for instruction on installation.

🛈 **NOTE**

If you have not previously used an RP2040-based board, begin by reading the Raspberry Pi Pico Python SDK book.

🛈 **NOTE**

More information on using Bluetooth from MicroPython can be found online in the MicroPython documentation.

6.1. Advertsing a Bluetooth service

We can create a custom service in MicroPython and advertise it using the following code:

```python
1  # Helpers for generating BLE advertising payloads
2
3  from micropython import const
4  import struct
5  import bluetooth
6
7  # Advertising payloads are repeated packets of the following form:
8  #   1 byte data length (N + 1)
9  #   1 byte type (see constants below)
10 #   N bytes type-specific data
11
12 _ADV_TYPE_FLAGS = const(0x01)
13 _ADV_TYPE_NAME = const(0x09)
14 _ADV_TYPE_UUID16_COMPLETE = const(0x3)
15 _ADV_TYPE_UUID32_COMPLETE = const(0x5)
16 _ADV_TYPE_UUID128_COMPLETE = const(0x7)
17 _ADV_TYPE_UUID16_MORE = const(0x2)
18 _ADV_TYPE_UUID32_MORE = const(0x4)
19 _ADV_TYPE_UUID128_MORE = const(0x6)
20 _ADV_TYPE_APPEARANCE = const(0x19)
21
```

```
22
23  # Generate a payload to be passed to gap_advertise(adv_data=...).
24  def advertising_payload(limited_disc=False, br_edr=False, name=None, services=None,
    appearance=0):
25      payload = bytearray()
26
27      def _append(adv_type, value):
28          nonlocal payload
29          payload += struct.pack("BB", len(value) + 1, adv_type) + value
30
31      _append(
32          _ADV_TYPE_FLAGS,
33          struct.pack("B", (0x01 if limited_disc else 0x02) + (0x18 if br_edr else 0x04)),
34      )
35
36      if name:
37          _append(_ADV_TYPE_NAME, name)
38
39      if services:
40          for uuid in services:
41              b = bytes(uuid)
42              if len(b) == 2:
43                  _append(_ADV_TYPE_UUID16_COMPLETE, b)
44              elif len(b) == 4:
45                  _append(_ADV_TYPE_UUID32_COMPLETE, b)
46              elif len(b) == 16:
47                  _append(_ADV_TYPE_UUID128_COMPLETE, b)
48
49      # See org.bluetooth.characteristic.gap.appearance.xml
50      if appearance:
51          _append(_ADV_TYPE_APPEARANCE, struct.pack("<h", appearance))
52
53      return payload
54
55
56  def decode_field(payload, adv_type):
57      i = 0
58      result = []
59      while i + 1 < len(payload):
60          if payload[i + 1] == adv_type:
61              result.append(payload[i + 2 : i + payload[i] + 1])
62          i += 1 + payload[i]
63      return result
64
65
66  def decode_name(payload):
67      n = decode_field(payload, _ADV_TYPE_NAME)
68      return str(n[0], "utf-8") if n else ""
69
70
71  def decode_services(payload):
72      services = []
73      for u in decode_field(payload, _ADV_TYPE_UUID16_COMPLETE):
74          services.append(bluetooth.UUID(struct.unpack("<h", u)[0]))
75      for u in decode_field(payload, _ADV_TYPE_UUID32_COMPLETE):
76          services.append(bluetooth.UUID(struct.unpack("<d", u)[0]))
77      for u in decode_field(payload, _ADV_TYPE_UUID128_COMPLETE):
78          services.append(bluetooth.UUID(u))
79      return services
80
81
82  def demo():
83      payload = advertising_payload(
84          name="micropython",
```

```
85         services=[bluetooth.UUID(0x181A), bluetooth.UUID("6E400001-B5A3-F393-E0A9-
    E50E24DCCA9E")],
86     )
87     print(payload)
88     print(decode_name(payload))
89     print(decode_services(payload))
90
91
92 if __name__ == "__main__":
93     demo()
```

Full details of supported Bluetooth protocols and profiles are Blue Kitchen BTStack Github repository.

6.2. An example Bluetooth service

We can use the generic peripheral advertising code in Section 6.1 to help implement an example service.

6.2.1. Creating a temperature service peripheral

This example demonstrates a simple temperature sensor peripheral. The sensor's local value is updated every ten seconds, and any connected central device will be notified of the change.

```
 1 import bluetooth
 2 import random
 3 import struct
 4 import time
 5 import machine
 6 import ubinascii
 7 from ble_advertising import advertising_payload
 8 from micropython import const
 9 from machine import Pin
10
11 _IRQ_CENTRAL_CONNECT = const(1)
12 _IRQ_CENTRAL_DISCONNECT = const(2)
13 _IRQ_GATTS_INDICATE_DONE = const(20)
14
15 _FLAG_READ = const(0x0002)
16 _FLAG_NOTIFY = const(0x0010)
17 _FLAG_INDICATE = const(0x0020)
18
19 # org.bluetooth.service.environmental_sensing
20 _ENV_SENSE_UUID = bluetooth.UUID(0x181A)
21
22 # org.bluetooth.characteristic.temperature
23 _TEMP_CHAR = (
24     bluetooth.UUID(0x2A6E),
25     _FLAG_READ | _FLAG_NOTIFY | _FLAG_INDICATE,
26 )
27 _ENV_SENSE_SERVICE = (
28     _ENV_SENSE_UUID,
29     (_TEMP_CHAR,),
30 )
31
32 # org.bluetooth.characteristic.gap.appearance.xml
33 _ADV_APPEARANCE_GENERIC_THERMOMETER = const(768)
34
```

```
35 class BLETemperature:
36     def __init__(self, ble, name=""):
37         self._sensor_temp = machine.ADC(4)
38         self._ble = ble
39         self._ble.active(True)
40         self._ble.irq(self._irq)
41         ((self._handle,),) = self._ble.gatts_register_services((_ENV_SENSE_SERVICE,))
42         self._connections = set()
43         if len(name) == 0:
44             name = 'Pico %s' % ubinascii.hexlify(self._ble.config('mac')[1],':').decode
   ().upper()
45         print('Sensor name %s' % name)
46         self._payload = advertising_payload(
47             name=name, services=[_ENV_SENSE_UUID]
48         )
49         self._advertise()
50
51     def _irq(self, event, data):
52         # Track connections so we can send notifications.
53         if event == _IRQ_CENTRAL_CONNECT:
54             conn_handle, _, _ = data
55             self._connections.add(conn_handle)
56         elif event == _IRQ_CENTRAL_DISCONNECT:
57             conn_handle, _, _ = data
58             self._connections.remove(conn_handle)
59             # Start advertising again to allow a new connection.
60             self._advertise()
61         elif event == _IRQ_GATTS_INDICATE_DONE:
62             conn_handle, value_handle, status = data
63
64     def update_temperature(self, notify=False, indicate=False):
65         # Write the local value, ready for a central to read.
66         temp_deg_c = self._get_temp()
67         print("write temp %.2f degc" % temp_deg_c);
68         self._ble.gatts_write(self._handle, struct.pack("<h", int(temp_deg_c * 100)))
69         if notify or indicate:
70             for conn_handle in self._connections:
71                 if notify:
72                     # Notify connected centrals.
73                     self._ble.gatts_notify(conn_handle, self._handle)
74                 if indicate:
75                     # Indicate connected centrals.
76                     self._ble.gatts_indicate(conn_handle, self._handle)
77
78     def _advertise(self, interval_us=500000):
79         self._ble.gap_advertise(interval_us, adv_data=self._payload)
80
81     def _get_temp(self):
82         conversion_factor = 3.3 / (65535)
83         reading = self._sensor_temp.read_u16() * conversion_factor
84
85         return 27 - (reading - 0.706) / 0.001721 ①
86
87 def demo():
88     ble = bluetooth.BLE()
89     temp = BLETemperature(ble)
90     counter = 0
91     led = Pin('LED', Pin.OUT)
92     while True:
93         if counter % 10 == 0:
94             temp.update_temperature(notify=True, indicate=False)
95         led.toggle()
96         time.sleep_ms(1000)
97         counter += 1
```

```
 98
 99 if __name__ == "__main__":
100     demo()
```

① The temperature sensor measures the V_{be} voltage of a biased bipolar diode, connected to the fifth ADC channel. Typically, V_{be} = 0.706V at 27°C, with a slope of -1.721mV (0.001721) per degree.

> ℹ️ **NOTE**
>
> This example service imports code from Section 6.1.

If you connect to the Bluetooth device using a scanner application on your phone (see Figure 8) you should see a service entry in the "Environmental sensing" section. Beneath this, there's a temperature section. You should be able to read the temperature or subscribe for notifications when the temperature value changes.

Figure 8. The `mpy-temp` *peripheral in the Punch Through LightBlue app: advertising data (left), services (middle), and temperature service (right).*
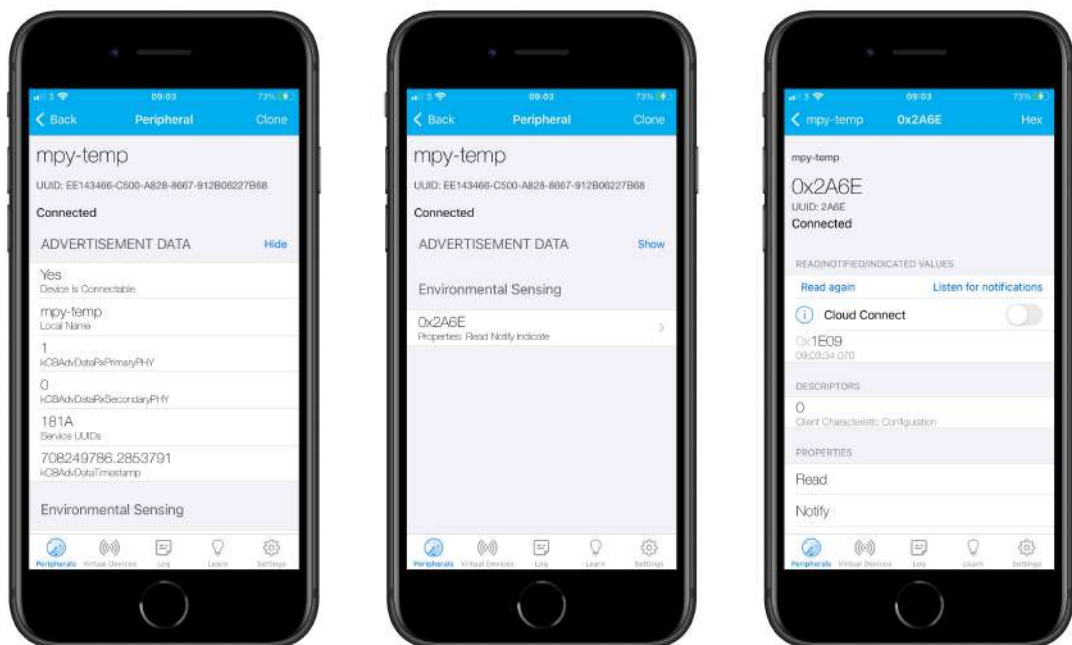
The temperature will show up as a hexadecimal number, e.g. `0x4B0B`. This number is a two-byte, little-endian representation of the temperature multiplied by 100. To get the value back in centigrade, you will need flip this representation to big-endian (e.g. 0x4B0B becomes 0x0B4B), convert the value to decimal (e.g. 0x0B4B becomes 2891), and then divide by 100 to get the value in centigrade (here it would be 28.91°C).

6.2.2. Implementing a central device

While it is likely that the main use-case for BLE on Pico W will be as a peripheral offering services, it is also possible to use your Pico W as a central device. Here we implement a central device that finds and connects to temperature peripherals (see Section 6.2.1).

```
1 # This example finds and connects to a BLE temperature sensor (e.g. the one in
  ble_temperature.py).
2
3 import bluetooth
4 import random
```

```
 5 import struct
 6 import time
 7 import micropython
 8 from ble_advertising import decode_services, decode_name
 9 from micropython import const
10 from machine import Pin
11
12 _IRQ_CENTRAL_CONNECT = const(1)
13 _IRQ_CENTRAL_DISCONNECT = const(2)
14 _IRQ_GATTS_WRITE = const(3)
15 _IRQ_GATTS_READ_REQUEST = const(4)
16 _IRQ_SCAN_RESULT = const(5)
17 _IRQ_SCAN_DONE = const(6)
18 _IRQ_PERIPHERAL_CONNECT = const(7)
19 _IRQ_PERIPHERAL_DISCONNECT = const(8)
20 _IRQ_GATTC_SERVICE_RESULT = const(9)
21 _IRQ_GATTC_SERVICE_DONE = const(10)
22 _IRQ_GATTC_CHARACTERISTIC_RESULT = const(11)
23 _IRQ_GATTC_CHARACTERISTIC_DONE = const(12)
24 _IRQ_GATTC_DESCRIPTOR_RESULT = const(13)
25 _IRQ_GATTC_DESCRIPTOR_DONE = const(14)
26 _IRQ_GATTC_READ_RESULT = const(15)
27 _IRQ_GATTC_READ_DONE = const(16)
28 _IRQ_GATTC_WRITE_DONE = const(17)
29 _IRQ_GATTC_NOTIFY = const(18)
30 _IRQ_GATTC_INDICATE = const(19)
31
32 _ADV_IND = const(0x00)
33 _ADV_DIRECT_IND = const(0x01)
34 _ADV_SCAN_IND = const(0x02)
35 _ADV_NONCONN_IND = const(0x03)
36
37 # org.bluetooth.service.environmental_sensing
38 _ENV_SENSE_UUID = bluetooth.UUID(0x181A)
39 # org.bluetooth.characteristic.temperature
40 _TEMP_UUID = bluetooth.UUID(0x2A6E)
41 _TEMP_CHAR = (
42     _TEMP_UUID,
43     bluetooth.FLAG_READ | bluetooth.FLAG_NOTIFY,
44 )
45 _ENV_SENSE_SERVICE = (
46     _ENV_SENSE_UUID,
47     (_TEMP_CHAR,),
48 )
49
50 class BLETemperatureCentral:
51     def __init__(self, ble):
52         self._ble = ble
53         self._ble.active(True)
54         self._ble.irq(self._irq)
55         self._reset()
56         self._led = Pin('LED', Pin.OUT)
57
58     def _reset(self):
59         # Cached name and address from a successful scan.
60         self._name = None
61         self._addr_type = None
62         self._addr = None
63
64         # Cached value (if we have one)
65         self._value = None
66
67         # Callbacks for completion of various operations.
68         # These reset back to None after being invoked.
```

```
69              self._scan_callback = None
70              self._conn_callback = None
71              self._read_callback = None
72
73              # Persistent callback for when new data is notified from the device.
74              self._notify_callback = None
75
76              # Connected device.
77              self._conn_handle = None
78              self._start_handle = None
79              self._end_handle = None
80              self._value_handle = None
81
82      def _irq(self, event, data):
83          if event == _IRQ_SCAN_RESULT:
84              addr_type, addr, adv_type, rssi, adv_data = data
85              if adv_type in (_ADV_IND, _ADV_DIRECT_IND):
86                  type_list = decode_services(adv_data)
87                  if _ENV_SENSE_UUID in type_list:
88                      # Found a potential device, remember it and stop scanning.
89                      self._addr_type = addr_type
90                      self._addr = bytes(addr)  # Note: addr buffer is owned by caller so need
    to copy it.
91                      self._name = decode_name(adv_data) or "?"
92                      self._ble.gap_scan(None)
93
94          elif event == _IRQ_SCAN_DONE:
95              if self._scan_callback:
96                  if self._addr:
97                      # Found a device during the scan (and the scan was explicitly stopped).
98                      self._scan_callback(self._addr_type, self._addr, self._name)
99                      self._scan_callback = None
100                 else:
101                     # Scan timed out.
102                     self._scan_callback(None, None, None)
103
104         elif event == _IRQ_PERIPHERAL_CONNECT:
105             # Connect successful.
106             conn_handle, addr_type, addr = data
107             if addr_type == self._addr_type and addr == self._addr:
108                 self._conn_handle = conn_handle
109                 self._ble.gattc_discover_services(self._conn_handle)
110
111         elif event == _IRQ_PERIPHERAL_DISCONNECT:
112             # Disconnect (either initiated by us or the remote end).
113             conn_handle, _, _ = data
114             if conn_handle == self._conn_handle:
115                 # If it was initiated by us, it'll already be reset.
116                 self._reset()
117
118         elif event == _IRQ_GATTC_SERVICE_RESULT:
119             # Connected device returned a service.
120             conn_handle, start_handle, end_handle, uuid = data
121             if conn_handle == self._conn_handle and uuid == _ENV_SENSE_UUID:
122                 self._start_handle, self._end_handle = start_handle, end_handle
123
124         elif event == _IRQ_GATTC_SERVICE_DONE:
125             # Service query complete.
126             if self._start_handle and self._end_handle:
127                 self._ble.gattc_discover_characteristics(
128                     self._conn_handle, self._start_handle, self._end_handle
129                 )
130             else:
131                 print("Failed to find environmental sensing service.")
```

```
132
133        elif event == _IRQ_GATTC_CHARACTERISTIC_RESULT:
134            # Connected device returned a characteristic.
135            conn_handle, def_handle, value_handle, properties, uuid = data
136            if conn_handle == self._conn_handle and uuid == _TEMP_UUID:
137                self._value_handle = value_handle
138
139        elif event == _IRQ_GATTC_CHARACTERISTIC_DONE:
140            # Characteristic query complete.
141            if self._value_handle:
142                # We've finished connecting and discovering device, fire the connect callback.
143                if self._conn_callback:
144                    self._conn_callback()
145            else:
146                print("Failed to find temperature characteristic.")
147
148        elif event == _IRQ_GATTC_READ_RESULT:
149            # A read completed successfully.
150            conn_handle, value_handle, char_data = data
151            if conn_handle == self._conn_handle and value_handle == self._value_handle:
152                self._update_value(char_data)
153                if self._read_callback:
154                    self._read_callback(self._value)
155                    self._read_callback = None
156
157        elif event == _IRQ_GATTC_READ_DONE:
158            # Read completed (no-op).
159            conn_handle, value_handle, status = data
160
161        elif event == _IRQ_GATTC_NOTIFY:
162            # The ble_temperature.py demo periodically notifies its value.
163            conn_handle, value_handle, notify_data = data
164            if conn_handle == self._conn_handle and value_handle == self._value_handle:
165                self._update_value(notify_data)
166                if self._notify_callback:
167                    self._notify_callback(self._value)
168
169    # Returns true if we've successfully connected and discovered characteristics.
170    def is_connected(self):
171        return self._conn_handle is not None and self._value_handle is not None
172
173    # Find a device advertising the environmental sensor service.
174    def scan(self, callback=None):
175        self._addr_type = None
176        self._addr = None
177        self._scan_callback = callback
178        self._ble.gap_scan(2000, 30000, 30000)
179
180    # Connect to the specified device (otherwise use cached address from a scan).
181    def connect(self, addr_type=None, addr=None, callback=None):
182        self._addr_type = addr_type or self._addr_type
183        self._addr = addr or self._addr
184        self._conn_callback = callback
185        if self._addr_type is None or self._addr is None:
186            return False
187        self._ble.gap_connect(self._addr_type, self._addr)
188        return True
189
190    # Disconnect from current device.
191    def disconnect(self):
192        if not self._conn_handle:
193            return
194        self._ble.gap_disconnect(self._conn_handle)
195        self._reset()
```

```
196
197     # Issues an (asynchronous) read, will invoke callback with data.
198     def read(self, callback):
199         if not self.is_connected():
200             return
201         self._read_callback = callback
202         try:
203             self._ble.gattc_read(self._conn_handle, self._value_handle)
204         except OSError as error:
205             print(error)
206
207     # Sets a callback to be invoked when the device notifies us.
208     def on_notify(self, callback):
209         self._notify_callback = callback
210
211     def _update_value(self, data):
212         # Data is sint16 in degrees Celsius with a resolution of 0.01 degrees Celsius.
213         try:
214             self._value = struct.unpack("<h", data)[0] / 100
215         except OSError as error:
216             print(error)
217
218     def value(self):
219         return self._value
220
221 def sleep_ms_flash_led(self, flash_count, delay_ms):
222     self._led.off()
223     while(delay_ms > 0):
224         for i in range(flash_count):
225             self._led.on()
226             time.sleep_ms(100)
227             self._led.off()
228             time.sleep_ms(100)
229             delay_ms -= 200
230         time.sleep_ms(1000)
231         delay_ms -= 1000
232
233 def print_temp(result):
234     print("read temp: %.2f degc" % result)
235
236 def demo(ble, central):
237     not_found = False
238
239     def on_scan(addr_type, addr, name):
240         if addr_type is not None:
241             print("Found sensor: %s" % name)
242             central.connect()
243         else:
244             nonlocal not_found
245             not_found = True
246             print("No sensor found.")
247
248     central.scan(callback=on_scan)
249
250     # Wait for connection...
251     while not central.is_connected():
252         time.sleep_ms(100)
253         if not_found:
254             return
255
256     print("Connected")
257
258     # Explicitly issue reads
259     while central.is_connected():
```

```
260          central.read(callback=print_temp)
261          sleep_ms_flash_led(central, 2, 2000)
262
263      print("Disconnected")
264
265 if __name__ == "__main__":
266      ble = bluetooth.BLE()
267      central = BLETemperatureCentral(ble)
268      while(True):
269          demo(ble, central)
270          sleep_ms_flash_led(central, 1, 10000)
```

ⓘ NOTE

This example imports code from Section 6.1.

Copy `ble_advertising.py` and `picow_ble_temp_reader.py` to a second Raspberry Pi Pico W. It should start running, and the on-board LED will flash briefly once repeatedly if it can't find a device to connect to. Once it finds another device running the `picow_ble_temp_sensor.py`, it will repeatedly flash twice, quickly, when it is connected and reading the temperature over Bluetooth.

Appendix A: Building MicroPython from source

Before you can proceed with building a MicroPython UF2 for Raspberry Pi Pico W from source, you should install the normal dependencies to build MicroPython. See Section 1.3 of the Raspberry Pi Pico Python SDK book for full details.

Afterwards you should clone the `micropython` and `micropython-lib` repositories.

```
$ mkdir pico_w
$ cd pico_w
$ git clone https://github.com/micropython/micropython.git --branch master
$ git clone https://github.com/micropython/micropython-lib.git --branch master
```

ℹ **NOTE**

> Putting `micropython-lib` side-by-side with your MicroPython checkout will mean that it is automatically pulled into your MicroPython build, and libraries in `micropython-lib` will be "pre-added" to the list of modules available by default on your Pico W device.

Then build MicroPython:

```
$ cd micropython
$ make -C ports/rp2 BOARD=PICO_W submodules
$ make -C mpy-cross
$ cd ports/rp2
$ make BOARD=PICO_W
```

If everything went well, there will be a new directory called `build-PICO_W` (that's `ports/rp2/build-PICO_W` relative to the top-level `micropython` directory), which contains the new firmware binaries. Drag and drop the `firmware.uf2` onto the RPI-RP2 drive that pops up once your Raspberry Pi Pico W is in BOOTSEL mode.

Appendix B: Documentation release history

Table 1.
Documentation
release history

Release	Date	Description
1.0	30 Jun 2022	• Initial release
Pico and Pico W databooks combined into a unified release history		
2.0	01 Dec 2022	• Minor updates and corrections • Added RP2040 availability information • Added RP2040 storage conditions and thermal characteristics • Replace SDK library documentation with links to the online version • Updated Picoprobe build and usage instructions
2.1	03 Mar 2023	• A large number of minor updates and corrections • SMT footprint of Pico W corrected • Updated for the 1.5.0 release of the Raspberry Pi Pico C SDK • Added errata E15 • Added documentation around the new Pico Windows Installer • Added documentation around the Pico-W-Go extension for Python development • Added a wireless networking example to the Python documentation • Added package marking specifications • Added RP2040 baseline power consumption figures • Added antenna keep out diagram to Pico W datasheet
2.2	14 Jun 2023	• Minor updates and corrections • Updated for the 1.5.1 release of the Raspberry Pi Pico C SDK • Documentation around Bluetooth support for Pico W

The latest release can be found at https://datasheets.raspberrypi.com/picow/connecting-to-the-internet-with-pico-w.pdf.