Wi-Fi® and Bluetooth connectivity on Pico W-series devices

Getting online with Wi-Fi® and connecting with Bluetooth with C/C++ or MicroPython

Colophon

Release	8
Build date	03/07/2026
Build version	d4e0f1799616

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 LTD (“RPL”) “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 RPL 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.

RPL 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 RPL harmless against all liabilities, costs, damages or other losses arising out of their use of the RESOURCES.

RPL 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 RPL 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”). RPL 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 RPL’s Standard Terms. RPL’s provision of the RESOURCES does not expand or otherwise modify RPL’s Standard Terms including but not limited to the disclaimers and warranties expressed in them.

Table of Contents

1. About Pico W-series devices

Pico W-series are microcontroller boards based on the Raspberry Pi RP-series microcontroller.

Figure 1.
The Raspberry Pi Pico W Rev3 board.

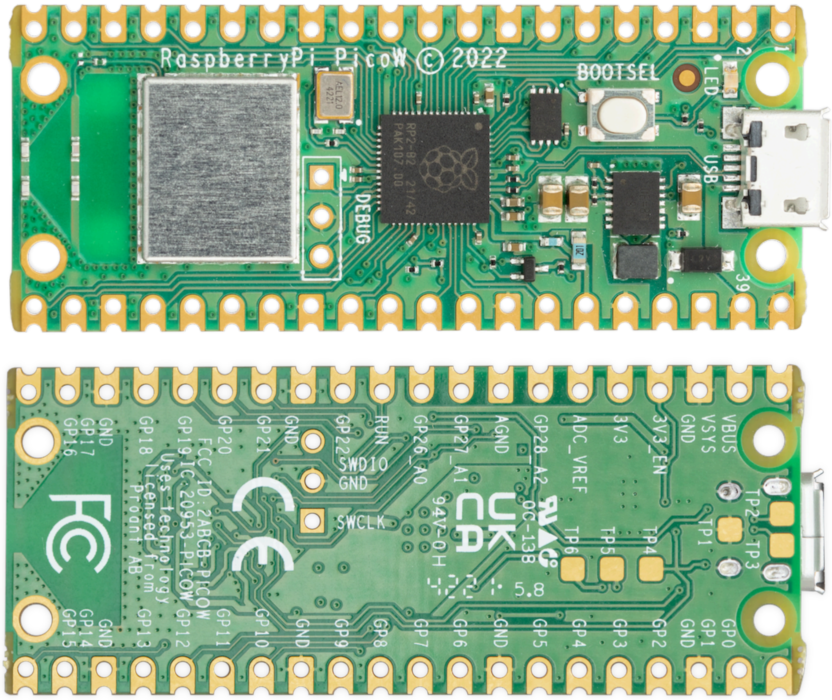
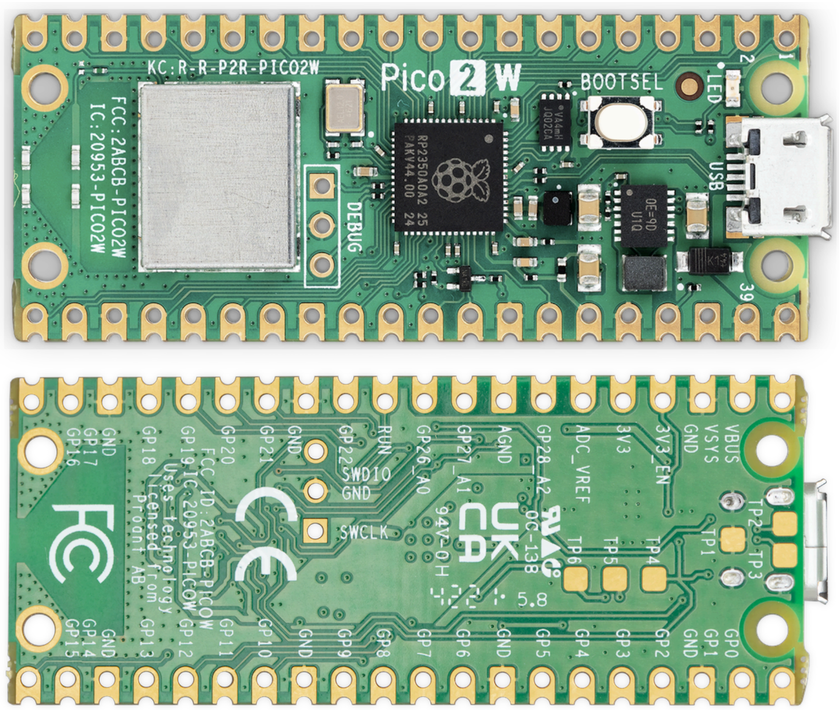


Figure 2.
The Raspberry Pi Pico 2 W board.



Pico W-series is a low cost yet flexible development platform for RP-series microcontroller, combining a 2.4 GHz wireless interface with the following key Pico-series features:

- RP-series microcontroller microcontroller with onboard flash
- On board 2.4 GHz 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 21 mm×51 mm 'DIP' style 1 mm thick PCB with 0.1 inch through-hole pins also with edge castellations
  - Exposes 26 multi-function 3.3 V general purpose I/O (GPIO)
  - 23 GPIO are digital-only, with 3 (RP2350) or 4 (RP2040) more which also support ADC
  - Can be surface-mounted as a module

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

Note

Full details of the Pico W-series can be found in Raspberry Pi Pico W datasheet and Raspberry Pi Pico 2 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.

2. Getting on the internet with the C SDK

Wireless support for Pico W-series devices has been added to the C/C++ SDK.

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

2.1. Installing the SDK and examples

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

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

> **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:

Code
```
$ 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
$
```

> **Note**
>
> 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 a Pico W-series device that will scan for nearby wireless networks, you can do:

Code
```
$ cd pico-examples/pico_w/wifi/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 RP-series microcontroller USB mass storage device.

This binary will scan for wireless networks using the Pico W-series device's wireless chip.

The fastest method to load software onto a RP-series microcontroller-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 Pico W-series device 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 Pico W-series device should automatically mount as a USB mass storage device. From here, you can drag-and-drop the UF2 file onto the mass storage device. RP-series microcontroller 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 Pico W-series UART TX pin (out) is pin GP0, and the UART RX pin (in) 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`:

Code

```
$ sudo raspi-config
```

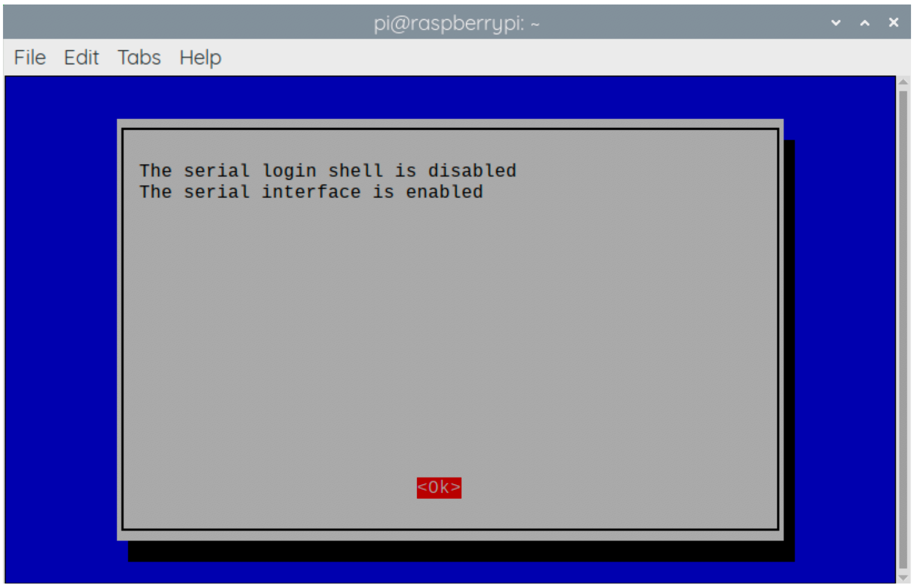
Go to `Interfacing Options` -> `Serial`.

When asked "Would you like a login shell to be accessible over serial?", select "No".

When asked "Would you like the serial port hardware to be enabled?", select "Yes".

You should see something like Figure 3:

Figure 3.
Enabling a serial UART using `raspi-config` on the Raspberry Pi.



Leaving `raspi-config`, choose "Yes" and reboot your Raspberry Pi to enable the serial port.

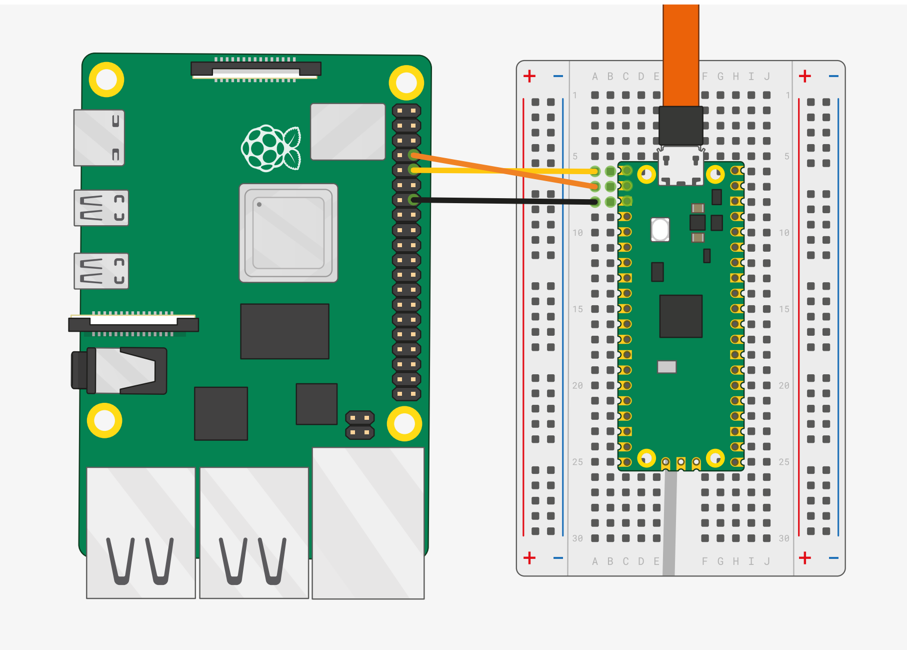
Then, wire the Raspberry Pi and the Pico W-series device together with the following mapping:

Table 1.

Raspberry Pi	Pico W-series device
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 4.

Figure 4.
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:

Code
```
$ sudo apt install minicom
```

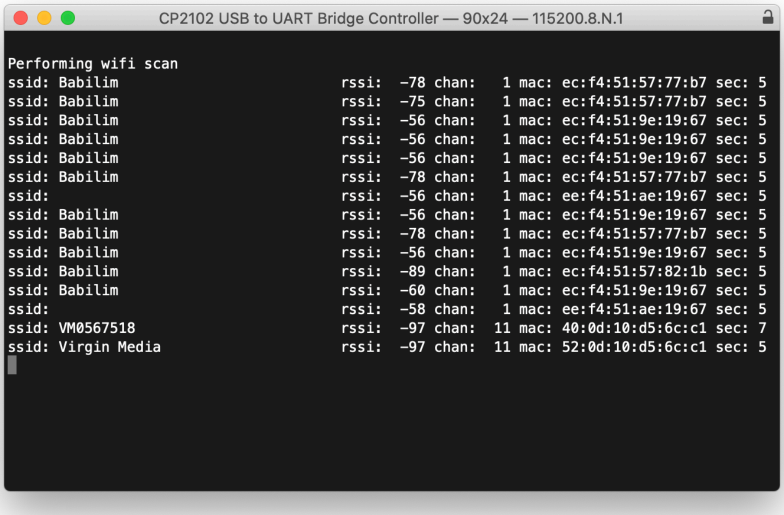
and open the serial port:

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

You should see the results of our wireless scanning being printed to the console, see Figure 5:

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

Figure 5.
Results of our wireless scanning in the console



2.3. Creating your own project

Run the following commands to create a directory to house your test project sitting alongside the `pico-sdk` directory:

Code
```
$ 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
```

Then, create a `test.c` file in that directory with the following contents:

Code
```
#include <stdio.h>

#include "pico/stdlib.h"
#include "pico/cyw43_arch.h"

char ssid[] = "A Network";<1>
char pass[] = "A Password";<2>

int main() {
    stdio_init_all();

    if (cyw43_arch_init_with_country(CYW43_COUNTRY_UK)) {
        printf("failed to initialise\n");
        return 1;
    }
    printf("initialised\n");

    cyw43_arch_enable_sta_mode();

    if (cyw43_arch_wifi_connect_timeout_ms(ssid, pass, CYW43_AUTH_WPA2_AES_PSK, 10000)) {
```

```
        printf("failed to connect\n");
        return 1;
    }
    printf("connected\n");
}
```

1. Replace `A Network` with the SSID name of your wireless network.
2. Replace `A Password` with the password for your wireless network.

And create a `CMakeLists.txt` file with the following contents:

Code

```
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)
```

Copy the `pico_sdk_import.cmake` file from the `external` folder in your `pico-sdk` installation to your test project folder:

Code

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

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

Code

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

You should now have something that looks like this:

Code

```
$ 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.

Code

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

Finally, unplug your Pico W-series device 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
```

You should see the a message indicating that the Pico W-series device 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-series 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. For additional C examples for Pico W-series devices, see the networking examples.

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 Pico-series or a Pico W-series. 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 0 V for Pico W-series and approximately 1/3 of `VSYS` for Pico-series.

Create a `test.c` file with the following contents:

```c
#include <stdio.h>

#include "pico/stdlib.h"
#include "hardware/gpio.h"
#include "hardware/adc.h"

int main() {
    stdio_init_all();

    adc_init();
    adc_gpio_init(29);
    adc_select_input(3);
    const float conversion_factor = 3.3f / (1 << 12);
    uint16_t result = adc_read();
    printf("ADC3 value: 0x%03x, voltage: %f V\n", result, result * conversion_factor);

    gpio_init(25);
    gpio_set_dir(25, GPIO_IN);
    uint value = gpio_get(25);
    printf("GP25 value: %i", value);
}
```

And a `CMakeLists.txt` file with the following contents:

```cmake
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)
```

You should see the following values for Raspberry Pi Pico W:

Code

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

You should see the following values for Raspberry Pi Pico:

Code

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

3. Getting on the internet with MicroPython

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

> **Note**
>
> If you have not previously used an RP-series microcontroller-based board you can get started by reading Raspberry Pi Pico-series Python SDK book.

3.1. Getting MicroPython for Pico W-series devices

> **Tip**
>
> A pre-built binary of the latest MicroPython firmware is available from the Pico section of the Raspberry Pi documentation .

The fastest way to get MicroPython is to download the pre-built release binary from the Pico section of the Raspberry Pi documentation . 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. Building MicroPython from source> to get the source code for MicroPython, which you can use to build your own MicroPython firmware binary.

3.2. Installing MicroPython on Pico W-series devices

Pico W-series devices have 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 Pico W-series devices 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 Pico W-series device.

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 Pico W-series GPIOs.

> **Note**
>
> The MicroPython port for RP-series microcontroller does not expose a REPL over a UART port by default, please see Raspberry Pi Pico-series 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, accessed through the micro USB connector on your Pico W-series devices. 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:

Code
```
$ sudo apt install minicom
```

and then open it as such:

Code
```
$ 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:

Code
```
>>>
```

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:

Code
```
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.

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:

Code
```
$ brew install minicom
```

and connect to the board as below.

Code
```
$ 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 Pico W-series and other RP-series microcontroller-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-series 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.

Code
```
$ 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 Pico W-series devices.

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

Note

For more information on `mpremote` see the documentation.

3.4. The on-board LED

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

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

or even:

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

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

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

You can also do the following:

Code
```
>>> 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.

Note

Full details of the Pico W-series can be found in the Raspberry Pi Pico W datasheet and Raspberry Pi Pico 2 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.

Code
```
>>> 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:

Code
```
import network
import time

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect('Wireless Network', 'The Password')

while not wlan.isconnected() and wlan.status() >= 0:
    print("Waiting to connect:")
    time.sleep(1)

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.

Code
```
import time
import network

ssid = 'Wireless Network'
password = 'The Password'

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect(ssid, password)

# Wait for connect or fail
max_wait = 10
while max_wait > 0:
    if wlan.status() < 0 or wlan.status() >= 3:
        break
    max_wait -= 1
    print('waiting for connection...')
    time.sleep(1)

# Handle connection error
if wlan.status() != 3:
    raise RuntimeError('network connection failed')
else:
    print('connected')
    status = wlan.ifconfig()
    print( 'ip = ' + status[0] )
```

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

Code
```
# Connect to another network
wlan.disconnect();
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.

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.

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

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

Code
```
>>> 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.

Code
```
import network

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.config(pm = 0xa11140)
```

3.7. The MAC address

The MAC is stored in the wireless chip OTP.

Code
```
import network
import ubinascii

wlan = network.WLAN(network.STA_IF)
```

```
wlan.active(True)
mac = ubinascii.hexlify(network.WLAN().config('mac'),':').decode()
print(mac)

# Other things you can query
print(wlan.config('channel'))
print(wlan.config('essid'))
print(wlan.config('txpower'))
```

> **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

Code

```
# Connect to network
import network

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect('Wireless Network', 'The Password')

# Should be connected and have an IP address
wlan.status() # 3 == success
wlan.ifconfig()

# Get IP address for google.com
import socket
ai = socket.getaddrinfo("google.com", 80)
addr = ai[0][-1]

# Create a socket and make a HTTP request
s = socket.socket()
s.connect(addr)
s.send(b"GET / HTTP/1.0\r\n\r\n")

# Print the response
print(s.recv(512))
```

3.8.2. HTTP with urequests

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

Code

```
# Connect to network
import network
wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect('Wireless Network', 'The Password')

# Make GET request
import urequests
r = urequests.get("http://www.google.com")
print(r.content)
r.close()
```

Support has been added for redirects.

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

Note

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

The `urequests` library comes with limited JSON support.

Code
```
>>> 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`.

Code
```
import time
import network
import urequests as requests

ssid = 'A Network'
password = 'The Password'

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect(ssid, password)

# Wait for connect or fail
max_wait = 10
while max_wait > 0:
    if wlan.status() < 0 or wlan.status() >= 3:
        break
    max_wait -= 1
    print('waiting for connection...')
    time.sleep(1)

# Handle connection error
if wlan.status() != 3:
    raise RuntimeError('network connection failed')
else:
    print('connected')
    status = wlan.ifconfig()
    print( 'ip = ' + status[0] )

while True:

    # Do things here, perhaps measure something using a sensor?
```

```
    # ...and then define the headers and payloads
    headers = ...
    payload = ...

    # Then send it in a try/except block
    try:
        print("sending...")
        response = requests.post("A REMOTE END POINT", headers=headers, data=payload)
        print("sent (" + str(response.status_code) + "), status = " + str(wlan.status()) )
        response.close()
    except:
        print("could not connect (status =" + str(wlan.status()) + ")")
        if wlan.status() < 0 or wlan.status() >= 3:
            print("trying to reconnect...")
            wlan.disconnect()
            wlan.connect(ssid, password)
            if wlan.status() == 3:
                print('connected')
            else:
                print('failed')

    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.

Code

```
import network
import socket
import time

from machine import Pin

led = Pin(15, Pin.OUT)

ssid = 'A Network'
password = 'A Password'

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect(ssid, password)

html = """<!DOCTYPE html>
<html>
    <head> <title>Pico W</title> </head>
    <body> <h1>Pico W</h1>
        <p>Hello World</p>
    </body>
</html>
"""

# Wait for connect or fail
max_wait = 10
while max_wait > 0:
    if wlan.status() < 0 or wlan.status() >= 3:
```

```
        break
    max_wait -= 1
    print('waiting for connection...')
    time.sleep(1)

# Handle connection error
if wlan.status() != 3:
    raise RuntimeError('network connection failed')
else:
    print('connected')
    status = wlan.ifconfig()
    print( 'ip = ' + status[0] )

# Open socket
addr = socket.getaddrinfo('0.0.0.0', 80)[0][-1]

s = socket.socket()
s.bind(addr)
s.listen(1)

print('listening on', addr)

# Listen for connections
while True:
    try:
        cl, addr = s.accept()
        print('client connected from', addr)
        cl_file = cl.makefile('rwb', 0)
        while True:
            line = cl_file.readline()
            if not line or line == b'\r\n':
                break
        response = html
        cl.send('HTTP/1.0 200 OK\r\nContent-type: text/html\r\n\r\n')
        cl.send(response)
        cl.close()

    except OSError as e:
        cl.close()
        print('connection closed')
```

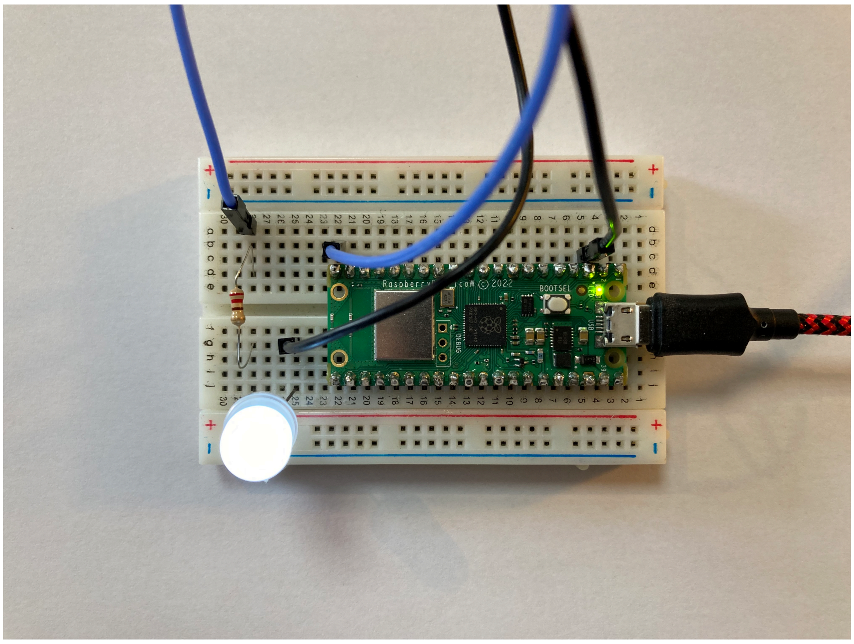
Note

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 6.
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-series device, which will be printed in the console after it connects to the network.

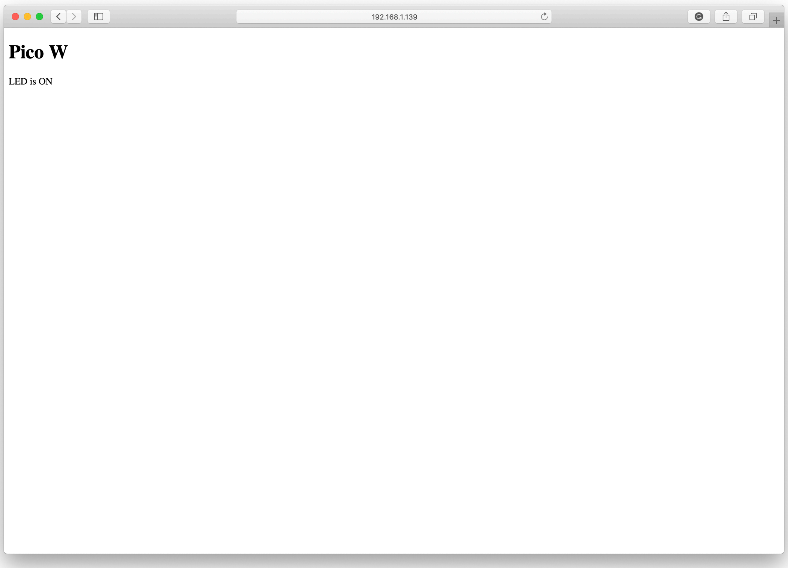
Code

```
import network
import socket
import time

from machine import Pin

led = Pin(15, Pin.OUT)

ssid = 'A Network'
password = 'A Password'

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect(ssid, password)

html = """<!DOCTYPE html>
<html>
    <head> <title>Pico W</title> </head>
    <body> <h1>Pico W</h1>
        <p>%s</p>
    </body>
</html>
"""

# Wait for connect or fail
max_wait = 10
while max_wait > 0:
    if wlan.status() < 0 or wlan.status() >= 3:
        break
    max_wait -= 1
    print('waiting for connection...')
    time.sleep(1)
```

```
# Handle connection error
if wlan.status() != 3:
    raise RuntimeError('network connection failed')
else:
    print('connected')
    status = wlan.ifconfig()
    print( 'ip = ' + status[0] )

# Open socket
addr = socket.getaddrinfo('0.0.0.0', 80)[0][-1]

s = socket.socket()
s.bind(addr)
s.listen(1)

print('listening on', addr)

# Listen for connections
while True:
    try:
        cl, addr = s.accept()
        print('client connected from', addr)
        request = cl.recv(1024)
        print(request)

        request = str(request)
        led_on = request.find('/light/on')
        led_off = request.find('/light/off')
        print( 'led on = ' + str(led_on))
        print( 'led off = ' + str(led_off))

        if led_on == 6:
            print("led on")
            led.value(1)
            stateis = "LED is ON"

        if led_off == 6:
            print("led off")
            led.value(0)
            stateis = "LED is OFF"

        response = html % stateis

        cl.send('HTTP/1.0 200 OK\r\nContent-type: text/html\r\n\r\n')
        cl.send(response)
        cl.close()

    except OSError as e:
        cl.close()
        print('connection closed')
```

Running the code, we can see the response in our browser.

Figure 7.
What we see in our web browser when connecting to our Pico-series device web server



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.

Code
```
import network
import socket
import time

from machine import Pin
import uasyncio as asyncio

led = Pin(15, Pin.OUT)
onboard = Pin("LED", Pin.OUT, value=0)

ssid = 'A Network'
password = 'A Password'

html = """<!DOCTYPE html>
<html>
    <head> <title>Pico W</title> </head>
    <body> <h1>Pico W</h1>
        <p>%s</p>
    </body>
</html>
"""

wlan = network.WLAN(network.STA_IF)

def connect_to_network():
    wlan.active(True)
    wlan.config(pm = 0xa11140)  # Disable power-save mode
    wlan.connect(ssid, password)

    max_wait = 10
    while max_wait > 0:
```

```
        if wlan.status() < 0 or wlan.status() >= 3:
            break
        max_wait -= 1
        print('waiting for connection...')
        time.sleep(1)

    if wlan.status() != 3:
        raise RuntimeError('network connection failed')
    else:
        print('connected')
        status = wlan.ifconfig()
        print('ip = ' + status[0])

async def serve_client(reader, writer):
    print("Client connected")
    request_line = await reader.readline()
    print("Request:", request_line)
    # We are not interested in HTTP request headers, skip them
    while True:
        header = await reader.readline()
        if not header or header == b"\r\n":
            break

    request = str(request_line)
    led_on = request.find('/light/on')
    led_off = request.find('/light/off')
    print( 'led on = ' + str(led_on))
    print( 'led off = ' + str(led_off))

    stateis = ""
    if led_on == 6:
        print("led on")
        led.value(1)
        stateis = "LED is ON"

    if led_off == 6:
        print("led off")
        led.value(0)
        stateis = "LED is OFF"

    response = html % stateis
    writer.write('HTTP/1.0 200 OK\r\nContent-type: text/html\r\n\r\n')
    writer.write(response)

    await writer.drain()
    await writer.wait_closed()
    print("Client disconnected")

async def main():
    print('Connecting to Network...')
    connect_to_network()

    print('Setting up webserver...')
    asyncio.create_task(asyncio.start_server(serve_client, "0.0.0.0", 80))
    while True:
        onboard.on()
        print("heartbeat")
        await asyncio.sleep(0.25)
        onboard.off()
        await asyncio.sleep(5)

try:
    asyncio.run(main())
finally:
    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.

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 Pico-series or a Pico W-series by looking at the hardware. However, you can tell indirectly by looking to see if network functionality is included in your particular MicroPython firmware:

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

Alternatively, you can inspect the MicroPython firmware version to check whether it was compiled for Pico-series or for Pico W-series 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-series.

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.

> **Note**
>
> Full details of supported Bluetooth protocols and profiles are available 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.

> **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 a unique identifier, known as a UUID. See Section 4.1.5. UUIDs.

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.

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

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

5. Working with Bluetooth and the C SDK

The C/C++ SDK contains Bluetooth support for Pico W-series devices.

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

Note

If you have not previously used an RP-series microcontroller-based board you can get started by reading Getting started with Raspberry Pi Pico-series, while further details about the Pico SDK, along with API-level documentation, can be found in the Raspberry Pi Pico-series 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.

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.

Code
```
PRIMARY_SERVICE, GAP_SERVICE
CHARACTERISTIC, GAP_DEVICE_NAME, READ, "picow_temp"

PRIMARY_SERVICE, GATT_SERVICE
CHARACTERISTIC, GATT_DATABASE_HASH, READ,

PRIMARY_SERVICE, ORG_BLUETOOTH_SERVICE_ENVIRONMENTAL_SENSING
CHARACTERISTIC, ORG_BLUETOOTH_CHARACTERISTIC_TEMPERATURE, READ | NOTIFY | INDICATE | DYNAMIC,
```

To build this example you should:

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

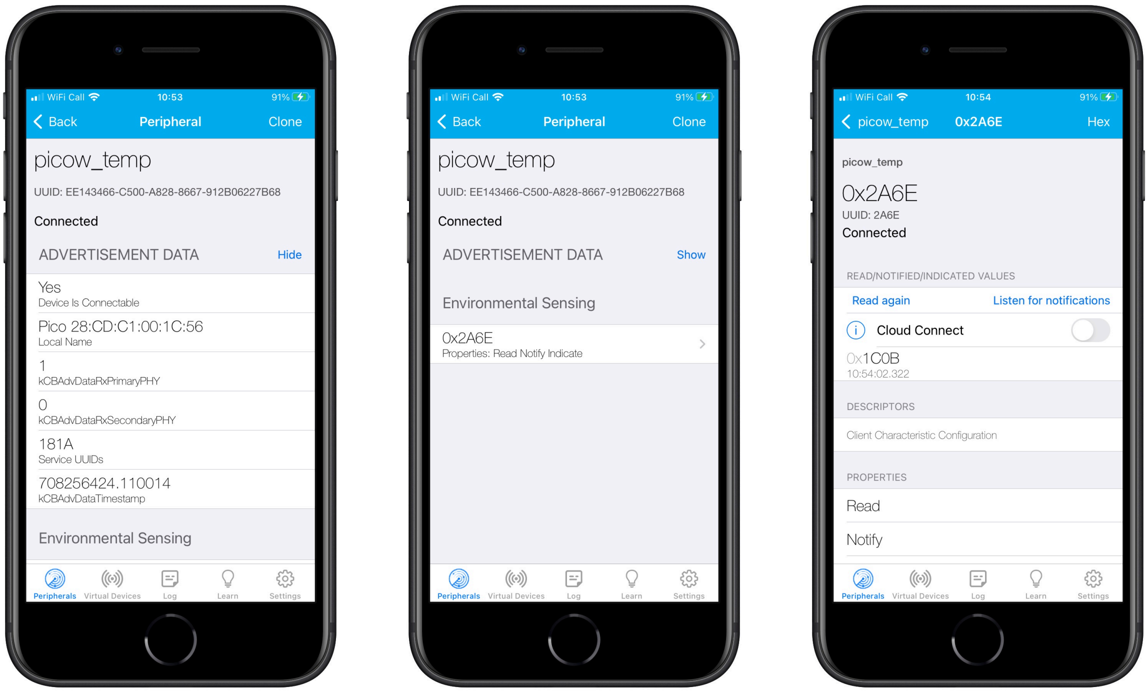
Along with other targets, we have now built two binaries called `picow_ble_temp_sensor.uf2` and `picow_ble_temp_reader.uf2`, which can be dragged onto the RP-series microcontroller USB mass-storage device.

The fastest method to load software onto an RP-series microcontroller-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 Pico W-series device 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 Pico W-series device should automatically mount as a USB mass-storage device. From here, you can drag and drop the UF2 file onto the mass-storage device. RP-series microcontroller will reboot, unmount 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 8), 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 8.
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.

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 available from the Pico section of the Raspberry Pi documentation .

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

Tip

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

Note

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

Note

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

6.1. Advertising a Bluetooth service

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

https://github.com/raspberrypi/pico-micropython-examples/blob/master/src/bluetooth/ble_advertising.py

```
# Helpers for generating BLE advertising payloads.

from micropython import const
import struct
import bluetooth

# Advertising payloads are repeated packets of the following form:
#   1 byte data length (N + 1)
#   1 byte type (see constants below)
#   N bytes type-specific data

_ADV_TYPE_FLAGS = const(0x01)
_ADV_TYPE_NAME = const(0x09)
_ADV_TYPE_UUID16_COMPLETE = const(0x3)
_ADV_TYPE_UUID32_COMPLETE = const(0x5)
_ADV_TYPE_UUID128_COMPLETE = const(0x7)
_ADV_TYPE_UUID16_MORE = const(0x2)
_ADV_TYPE_UUID32_MORE = const(0x4)
_ADV_TYPE_UUID128_MORE = const(0x6)
_ADV_TYPE_APPEARANCE = const(0x19)


# Generate a payload to be passed to gap_advertise(adv_data=...).
def advertising_payload(limited_disc=False, br_edr=False, name=None, services=None, appearance=0):
    payload = bytearray()

    def _append(adv_type, value):
        nonlocal payload
        payload += struct.pack("BB", len(value) + 1, adv_type) + value
```

```

    _append(
        _ADV_TYPE_FLAGS,
        struct.pack("B", (0x01 if limited_disc else 0x02) + (0x18 if br_edr else 0x04)),
    )

    if name:
        _append(_ADV_TYPE_NAME, name)

    if services:
        for uuid in services:
            b = bytes(uuid)
            if len(b) == 2:
                _append(_ADV_TYPE_UUID16_COMPLETE, b)
            elif len(b) == 4:
                _append(_ADV_TYPE_UUID32_COMPLETE, b)
            elif len(b) == 16:
                _append(_ADV_TYPE_UUID128_COMPLETE, b)

    # See org.bluetooth.characteristic.gap.appearance.xml
    if appearance:
        _append(_ADV_TYPE_APPEARANCE, struct.pack("<h", appearance))

    return payload


def decode_field(payload, adv_type):
    i = 0
    result = []
    while i + 1 < len(payload):
        if payload[i + 1] == adv_type:
            result.append(payload[i + 2 : i + payload[i] + 1])
        i += 1 + payload[i]
    return result


def decode_name(payload):
    n = decode_field(payload, _ADV_TYPE_NAME)
    return str(n[0], "utf-8") if n else ""


def decode_services(payload):
    services = []
    for u in decode_field(payload, _ADV_TYPE_UUID16_COMPLETE):
        services.append(bluetooth.UUID(struct.unpack("<h", u)[0]))
    for u in decode_field(payload, _ADV_TYPE_UUID32_COMPLETE):
        services.append(bluetooth.UUID(struct.unpack("<d", u)[0]))
    for u in decode_field(payload, _ADV_TYPE_UUID128_COMPLETE):
        services.append(bluetooth.UUID(u))
    return services


def demo():
    payload = advertising_payload(
        name="micropython",
        services=[bluetooth.UUID(0x181A), bluetooth.UUID("6E400001-B5A3-F393-E0A9-E50E24DCCA9E")],
    )
    print(payload)
    print(decode_name(payload))
    print(decode_services(payload))


if __name__ == "__main__":
    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. Advertising a Bluetooth service 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.

https://github.com/raspberrypi/pico-micropython-examples/blob/master/src/bluetooth/picow_ble_temp_sensor.py

```python
# This example demonstrates a simple temperature sensor peripheral.
#
# The sensor's local value is updated, and it will notify
# any connected central every 10 seconds.

import bluetooth
import random
import struct
import time
import machine
import ubinascii
from ble_advertising import advertising_payload
from micropython import const
from machine import Pin

_IRQ_CENTRAL_CONNECT = const(1)
_IRQ_CENTRAL_DISCONNECT = const(2)
_IRQ_GATTS_INDICATE_DONE = const(20)

_FLAG_READ = const(0x0002)
_FLAG_NOTIFY = const(0x0010)
_FLAG_INDICATE = const(0x0020)

# org.bluetooth.service.environmental_sensing
_ENV_SENSE_UUID = bluetooth.UUID(0x181A)
# org.bluetooth.characteristic.temperature
_TEMP_CHAR = (
    bluetooth.UUID(0x2A6E),
    _FLAG_READ | _FLAG_NOTIFY | _FLAG_INDICATE,
)
_ENV_SENSE_SERVICE = (
    _ENV_SENSE_UUID,
    (_TEMP_CHAR,),
)

# org.bluetooth.characteristic.gap.appearance.xml
_ADV_APPEARANCE_GENERIC_THERMOMETER = const(768)

class BLETemperature:
    def __init__(self, ble, name=""):
        self._sensor_temp = machine.ADC(4)
        self._ble = ble
        self._ble.active(True)
        self._ble.irq(self._irq)
        ((self._handle,),) = self._ble.gatts_register_services((_ENV_SENSE_SERVICE,))
        self._connections = set()
        if len(name) == 0:
            name = 'Pico %s' % ubinascii.hexlify(self._ble.config('mac')[1],':').decode().upper()
        print('Sensor name %s' % name)
```

```
        self._payload = advertising_payload(
            name=name, services=[_ENV_SENSE_UUID]
        )
        self._advertise()

    def _irq(self, event, data):
        # Track connections so we can send notifications.
        if event == _IRQ_CENTRAL_CONNECT:
            conn_handle, _, _ = data
            self._connections.add(conn_handle)
        elif event == _IRQ_CENTRAL_DISCONNECT:
            conn_handle, _, _ = data
            self._connections.remove(conn_handle)
            # Start advertising again to allow a new connection.
            self._advertise()
        elif event == _IRQ_GATTS_INDICATE_DONE:
            conn_handle, value_handle, status = data

    def update_temperature(self, notify=False, indicate=False):
        # Write the local value, ready for a central to read.
        temp_deg_c = self._get_temp()
        print("write temp %.2f degc" % temp_deg_c);
        self._ble.gatts_write(self._handle, struct.pack("<h", int(temp_deg_c * 100)))
        if notify or indicate:
            for conn_handle in self._connections:
                if notify:
                    # Notify connected centrals.
                    self._ble.gatts_notify(conn_handle, self._handle)
                if indicate:
                    # Indicate connected centrals.
                    self._ble.gatts_indicate(conn_handle, self._handle)

    def _advertise(self, interval_us=500000):
        self._ble.gap_advertise(interval_us, adv_data=self._payload)

    # ref https://github.com/raspberrypi/pico-micropython-examples/blob/master/adc/temperature.py
    def _get_temp(self):
        conversion_factor = 3.3 / (65535)
        reading = self._sensor_temp.read_u16() * conversion_factor

        # The temperature sensor measures the Vbe voltage of a biased bipolar diode, connected to the fifth
        ADC channel
        # Typically, Vbe = 0.706V at 27 degrees C, with a slope of -1.721mV (0.001721) per degree.
        return 27 - (reading - 0.706) / 0.001721

def demo():
    ble = bluetooth.BLE()
    temp = BLETemperature(ble)
    counter = 0
    led = Pin('LED', Pin.OUT)
    while True:
        if counter % 10 == 0:
            temp.update_temperature(notify=True, indicate=False)
        led.toggle()
        time.sleep_ms(1000)
        counter += 1

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

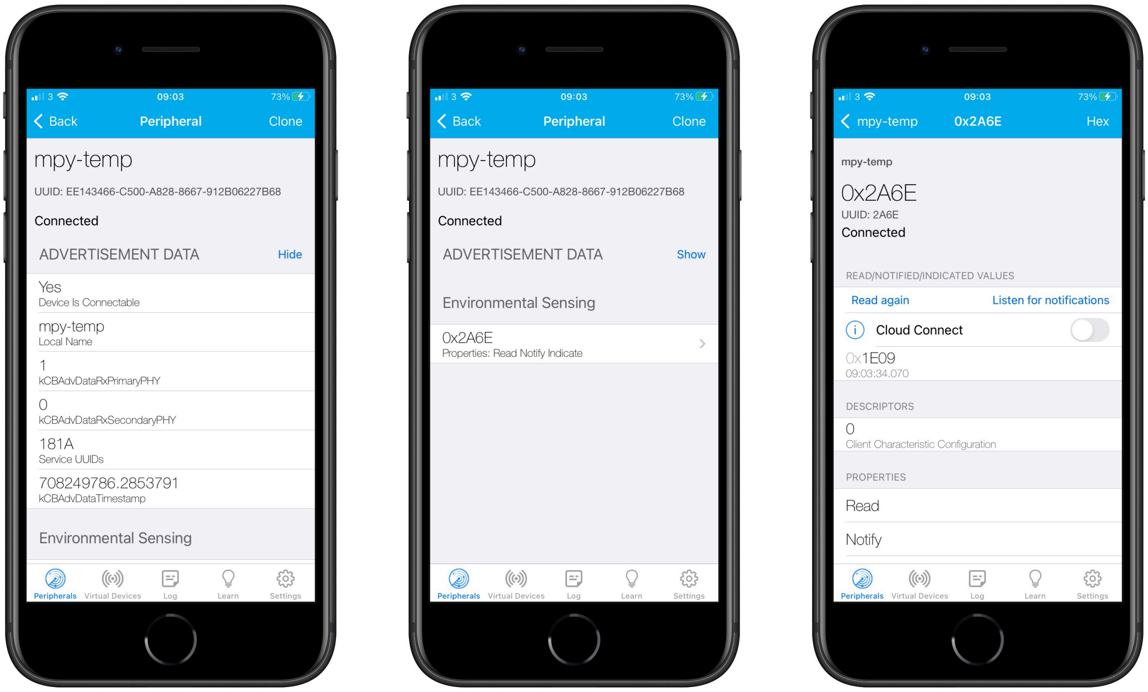
Note

The RP2040 temperature sensor measures the V_{be} voltage of a biased bipolar diode, connected to the fifth ADC channel. Typically, V_{be} = 0.706 V at 27°C, with a slope of −1.721 mV (0.001721) per degree.

> **Note**
>
> This example service imports code from Section 6.1. Advertising a Bluetooth service.

If you connect to the Bluetooth device using a scanner application on your phone (see Figure 9) 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 9.
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. Creating a temperature service peripheral).

https://github.com/raspberrypi/pico-micropython-examples/blob/master/src/bluetooth/picow_ble_temp_reader.py

```python
# This example finds and connects to a BLE temperature sensor (e.g. the one in ble_temperature.py).

import bluetooth
import random
import struct
import time
import micropython
from ble_advertising import decode_services, decode_name
from micropython import const
from machine import Pin

_IRQ_CENTRAL_CONNECT = const(1)
_IRQ_CENTRAL_DISCONNECT = const(2)
```

```
_IRQ_GATTS_WRITE = const(3)
_IRQ_GATTS_READ_REQUEST = const(4)
_IRQ_SCAN_RESULT = const(5)
_IRQ_SCAN_DONE = const(6)
_IRQ_PERIPHERAL_CONNECT = const(7)
_IRQ_PERIPHERAL_DISCONNECT = const(8)
_IRQ_GATTC_SERVICE_RESULT = const(9)
_IRQ_GATTC_SERVICE_DONE = const(10)
_IRQ_GATTC_CHARACTERISTIC_RESULT = const(11)
_IRQ_GATTC_CHARACTERISTIC_DONE = const(12)
_IRQ_GATTC_DESCRIPTOR_RESULT = const(13)
_IRQ_GATTC_DESCRIPTOR_DONE = const(14)
_IRQ_GATTC_READ_RESULT = const(15)
_IRQ_GATTC_READ_DONE = const(16)
_IRQ_GATTC_WRITE_DONE = const(17)
_IRQ_GATTC_NOTIFY = const(18)
_IRQ_GATTC_INDICATE = const(19)

_ADV_IND = const(0x00)
_ADV_DIRECT_IND = const(0x01)
_ADV_SCAN_IND = const(0x02)
_ADV_NONCONN_IND = const(0x03)

# org.bluetooth.service.environmental_sensing
_ENV_SENSE_UUID = bluetooth.UUID(0x181A)
# org.bluetooth.characteristic.temperature
_TEMP_UUID = bluetooth.UUID(0x2A6E)
_TEMP_CHAR = (
    _TEMP_UUID,
    bluetooth.FLAG_READ | bluetooth.FLAG_NOTIFY,
)
_ENV_SENSE_SERVICE = (
    _ENV_SENSE_UUID,
    (_TEMP_CHAR,),
)

class BLETemperatureCentral:
    def __init__(self, ble):
        self._ble = ble
        self._ble.active(True)
        self._ble.irq(self._irq)
        self._reset()
        self._led = Pin('LED', Pin.OUT)

    def _reset(self):
        # Cached name and address from a successful scan.
        self._name = None
        self._addr_type = None
        self._addr = None

        # Cached value (if we have one)
        self._value = None

        # Callbacks for completion of various operations.
        # These reset back to None after being invoked.
        self._scan_callback = None
        self._conn_callback = None
        self._read_callback = None

        # Persistent callback for when new data is notified from the device.
        self._notify_callback = None

        # Connected device.
        self._conn_handle = None
        self._start_handle = None
```

```
        self._end_handle = None
        self._value_handle = None

    def _irq(self, event, data):
        if event == _IRQ_SCAN_RESULT:
            addr_type, addr, adv_type, rssi, adv_data = data
            if adv_type in (_ADV_IND, _ADV_DIRECT_IND):
                type_list = decode_services(adv_data)
                if _ENV_SENSE_UUID in type_list:
                    # Found a potential device, remember it and stop scanning.
                    self._addr_type = addr_type
                    self._addr = bytes(addr)  # Note: addr buffer is owned by caller so need to copy it.
                    self._name = decode_name(adv_data) or "?"
                    self._ble.gap_scan(None)

        elif event == _IRQ_SCAN_DONE:
            if self._scan_callback:
                if self._addr:
                    # Found a device during the scan (and the scan was explicitly stopped).
                    self._scan_callback(self._addr_type, self._addr, self._name)
                    self._scan_callback = None
                else:
                    # Scan timed out.
                    self._scan_callback(None, None, None)

        elif event == _IRQ_PERIPHERAL_CONNECT:
            # Connect successful.
            conn_handle, addr_type, addr = data
            if addr_type == self._addr_type and addr == self._addr:
                self._conn_handle = conn_handle
                self._ble.gattc_discover_services(self._conn_handle)

        elif event == _IRQ_PERIPHERAL_DISCONNECT:
            # Disconnect (either initiated by us or the remote end).
            conn_handle, _, _ = data
            if conn_handle == self._conn_handle:
                # If it was initiated by us, it'll already be reset.
                self._reset()

        elif event == _IRQ_GATTC_SERVICE_RESULT:
            # Connected device returned a service.
            conn_handle, start_handle, end_handle, uuid = data
            if conn_handle == self._conn_handle and uuid == _ENV_SENSE_UUID:
                self._start_handle, self._end_handle = start_handle, end_handle

        elif event == _IRQ_GATTC_SERVICE_DONE:
            # Service query complete.
            if self._start_handle and self._end_handle:
                self._ble.gattc_discover_characteristics(
                    self._conn_handle, self._start_handle, self._end_handle
                )
            else:
                print("Failed to find environmental sensing service.")

        elif event == _IRQ_GATTC_CHARACTERISTIC_RESULT:
            # Connected device returned a characteristic.
            conn_handle, def_handle, value_handle, properties, uuid = data
            if conn_handle == self._conn_handle and uuid == _TEMP_UUID:
                self._value_handle = value_handle

        elif event == _IRQ_GATTC_CHARACTERISTIC_DONE:
            # Characteristic query complete.
            if self._value_handle:
                # We've finished connecting and discovering device, fire the connect callback.
                if self._conn_callback:
```

```
                    self._conn_callback()
            else:
                print("Failed to find temperature characteristic.")

        elif event == _IRQ_GATTC_READ_RESULT:
            # A read completed successfully.
            conn_handle, value_handle, char_data = data
            if conn_handle == self._conn_handle and value_handle == self._value_handle:
                self._update_value(char_data)
                if self._read_callback:
                    self._read_callback(self._value)
                    self._read_callback = None

        elif event == _IRQ_GATTC_READ_DONE:
            # Read completed (no-op).
            conn_handle, value_handle, status = data

        elif event == _IRQ_GATTC_NOTIFY:
            # The ble_temperature.py demo periodically notifies its value.
            conn_handle, value_handle, notify_data = data
            if conn_handle == self._conn_handle and value_handle == self._value_handle:
                self._update_value(notify_data)
                if self._notify_callback:
                    self._notify_callback(self._value)

    # Returns true if we've successfully connected and discovered characteristics.
    def is_connected(self):
        return self._conn_handle is not None and self._value_handle is not None

    # Find a device advertising the environmental sensor service.
    def scan(self, callback=None):
        self._addr_type = None
        self._addr = None
        self._scan_callback = callback
        self._ble.gap_scan(2000, 30000, 30000)

    # Connect to the specified device (otherwise use cached address from a scan).
    def connect(self, addr_type=None, addr=None, callback=None):
        self._addr_type = addr_type or self._addr_type
        self._addr = addr or self._addr
        self._conn_callback = callback
        if self._addr_type is None or self._addr is None:
            return False
        self._ble.gap_connect(self._addr_type, self._addr)
        return True

    # Disconnect from current device.
    def disconnect(self):
        if not self._conn_handle:
            return
        self._ble.gap_disconnect(self._conn_handle)
        self._reset()

    # Issues an (asynchronous) read, will invoke callback with data.
    def read(self, callback):
        if not self.is_connected():
            return
        self._read_callback = callback
        try:
            self._ble.gattc_read(self._conn_handle, self._value_handle)
        except OSError as error:
            print(error)

    # Sets a callback to be invoked when the device notifies us.
    def on_notify(self, callback):
```

```
        self._notify_callback = callback

    def _update_value(self, data):
        # Data is sint16 in degrees Celsius with a resolution of 0.01 degrees Celsius.
        try:
            self._value = struct.unpack("<h", data)[0] / 100
        except OSError as error:
            print(error)

    def value(self):
        return self._value

def sleep_ms_flash_led(self, flash_count, delay_ms):
    self._led.off()
    while(delay_ms > 0):
        for i in range(flash_count):
            self._led.on()
            time.sleep_ms(100)
            self._led.off()
            time.sleep_ms(100)
            delay_ms -= 200
        time.sleep_ms(1000)
        delay_ms -= 1000

def print_temp(result):
    print("read temp: %.2f degc" % result)

def demo(ble, central):
    not_found = False

    def on_scan(addr_type, addr, name):
        if addr_type is not None:
            print("Found sensor: %s" % name)
            central.connect()
        else:
            nonlocal not_found
            not_found = True
            print("No sensor found.")

    central.scan(callback=on_scan)

    # Wait for connection...
    while not central.is_connected():
        time.sleep_ms(100)
        if not_found:
            return

    print("Connected")

    # Explicitly issue reads
    while central.is_connected():
        central.read(callback=print_temp)
        sleep_ms_flash_led(central, 2, 2000)

    print("Disconnected")

if __name__ == "__main__":
    ble = bluetooth.BLE()
    central = BLETemperatureCentral(ble)
    while(True):
        demo(ble, central)
        sleep_ms_flash_led(central, 1, 10000)
```

Note

This example imports code from Section 6.1. Advertising a Bluetooth service.

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 Pico W-series devices from source, you should install the normal dependencies to build MicroPython. See Section 1.3 of the Raspberry Pi Pico-series Python SDK book for full details.

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

Code
```
$ 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
```

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-series device.

Then build MicroPython:

Code
```
$ 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 Pico W-series device enters `BOOTSEL` mode.

Appendix B. Documentation Release History

B.1. 03 July 2026

- Converted to the new build toolchain.
- Document renamed to better reflect its content.

B.2. 25 November 2024

- Added support for Pico 2 W.
- Switched back to separate release histories per PDF.

B.3. 02 February 2024

- Corrected minor typos and formatting issues.
- Updated ROSC register information.

B.4. 14 June 2023

- Corrected minor typos and formatting issues.
- Updated for the 1.5.1 release of the Raspberry Pi Pico C SDK.
- Added Pico W Bluetooth usage instructions.

B.5. 03 March 2023

- Corrected even more typos and formatting issues.
- Updated for the 1.5.0 release of the Raspberry Pi Pico C SDK.
- Added a wireless networking example to the Python documentation.

B.6. 01 December 2022

- Corrected various typos and formatting issues.
- Replaced SDK library documentation with links to the online

version.

B.7. 30 June 2022

- Initial release.

Raspberry Pi

Raspberry Pi Ltd