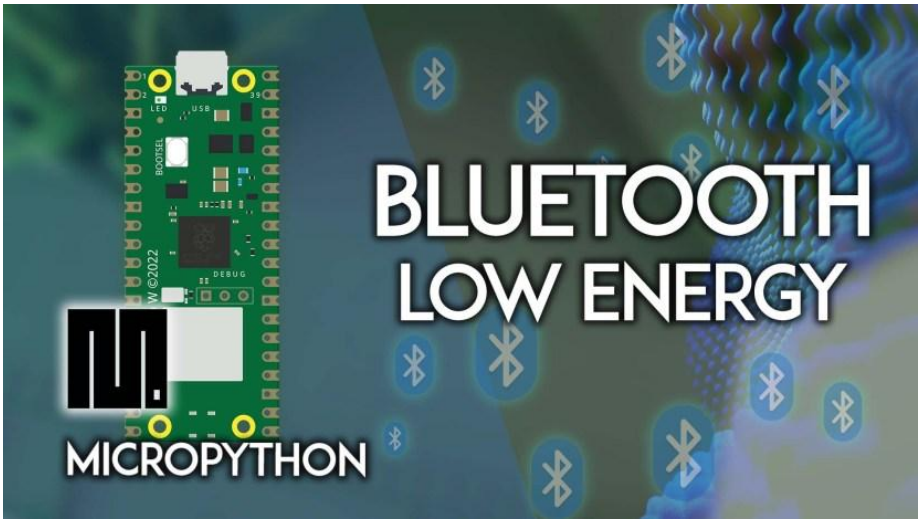
Raspberry Pi Pico W: Bluetooth Low Energy (BLE) with MicroPython

Source : https://randomnerdtutorials.com/raspberry-pi-pico-w-bluetooth-low-energy-micropython/

The Raspberry Pi Pico W (and 2 W) supports Bluetooth Low Energy, which can be useful in several IoT and automation projects. This article is a getting-started guide on how to use Bluetooth Low Energy with the Raspberry Pi Pico. We'll cover how to set the Raspberry Pi Pico as a BLE peripheral and as a BLE central device.



New to the Raspberry Pi Pico? Check out our eBook: **Learn Raspberry Pi Pico/Pico W with MicroPython**.

Prerequisites – MicroPython Firmware

To follow this tutorial, you need MicroPython firmware installed on your Raspberry Pi Pico board. You also need an IDE to write and upload the code to your board.

The recommended MicroPython IDE for the Raspberry Pi Pico is Thonny IDE. Follow the next tutorial to learn how to install Thonny IDE, flash MicroPython firmware, and upload code to the board.

- Programming Raspberry Pi Pico using MicroPython

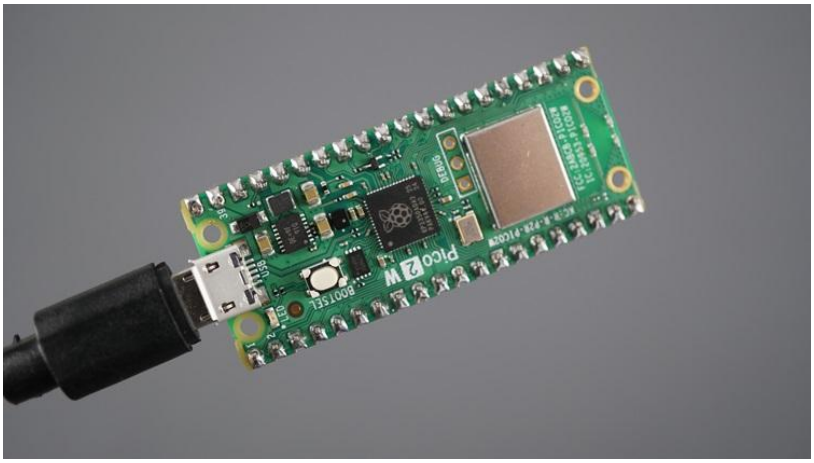
Bluetooth with the Raspberry Pi Pico W

The Raspberry Pi Pico W and Raspberry Pi Pico 2 W come with an Infineon CYW43439 chip that adds Wi-Fi and Bluetooth support.

This tutorial is only compatible with the W versions of the Raspberry Pi Pico:

- Raspberry Pi Pico W
- Raspberry Pi Pico 2 W

The documentation mentions it supports both Bluetooth Classic and Bluetooth Low Energy. However, I couldn't find examples for Bluetooth Classic with MicroPython at the moment. So, our examples will be focusing on Bluetooth Low Energy.


Raspberry Pi Pico 2 W

MicroPython support for Bluetooth with the Pico W is relatively recent, so there is still little support in terms of examples, libraries, documentation, and functionality. Additionally, some libraries are still in beta version, and either they don't have all their methods developed, or they don't provide examples or documentation, or have bugs or issues. So, at the moment, take that into account when dealing with Bluetooth on the Raspberry Pi Pico. Nonetheless, we'll provide you with some essential information and basic working examples that we could run and test that will help you get started.

What is Bluetooth Low Energy?

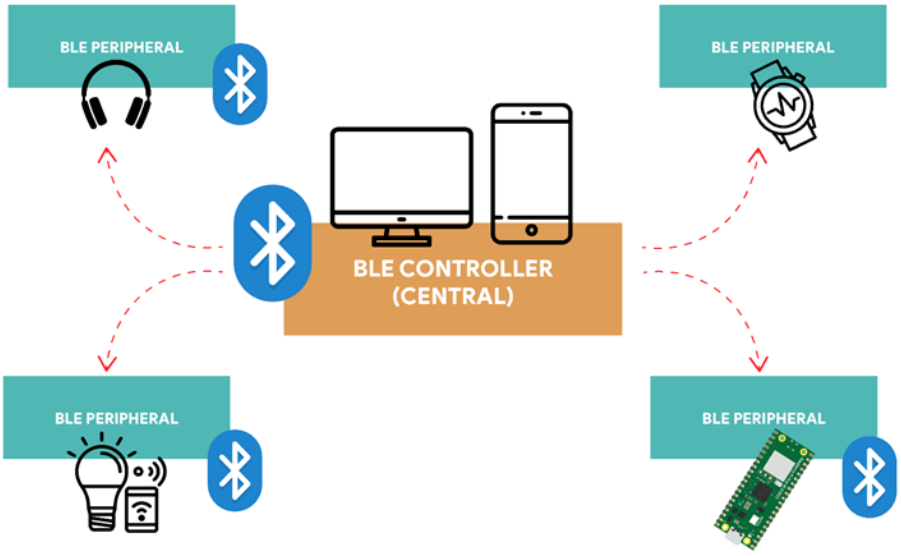
Bluetooth Low Energy, BLE for short (also called Bluetooth Smart), is a power-conserving variant of Bluetooth. BLE's primary application is short-distance transmission of small amounts of data (low bandwidth). Unlike Bluetooth, which is always on, BLE remains in sleep mode constantly except for when a connection is initiated. This makes it consume very little power. BLE consumes approximately 100x less power than Bluetooth (depending on the use case).

Bluetooth Low Energy Basic Concepts

Before proceeding, it's important to get familiar with some basic BLE concepts

BLE Peripheral and Controller (Central Device)

When using Bluetooth Low Energy (BLE), it's important to understand the roles of BLE Peripheral and BLE Controller (also referred to as the Central Device).

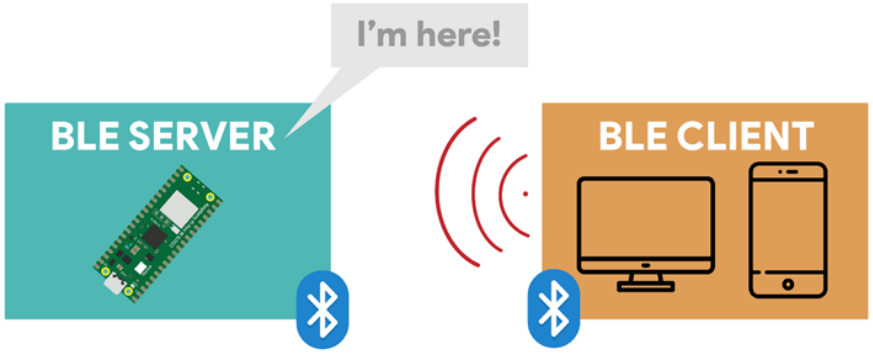


The Pico W can act either as a Peripheral or as a central device. When it acts as a peripheral it sets up a GATT profile and advertises its service with characteristics that the central devices can read. On the other hand, when it is set as a central device, it can connect to other BLE devices to read or interact with their profiles and read their characteristics.

In the above diagram, the Pico takes the role of the BLE Peripheral, serving as the device that provides data or services. Your smartphone or computer acts as the BLE Controller, managing the connection and communication with the Pico.

BLE Server and Client

With Bluetooth Low Energy, there are two types of devices: the server and the client. The Pico can act either as a client or as a server. In the picture below it acts as a server, exposing its GATT structure containing data. The BLE Server acts as a provider of data or services, while the BLE Client consumes or uses these services.

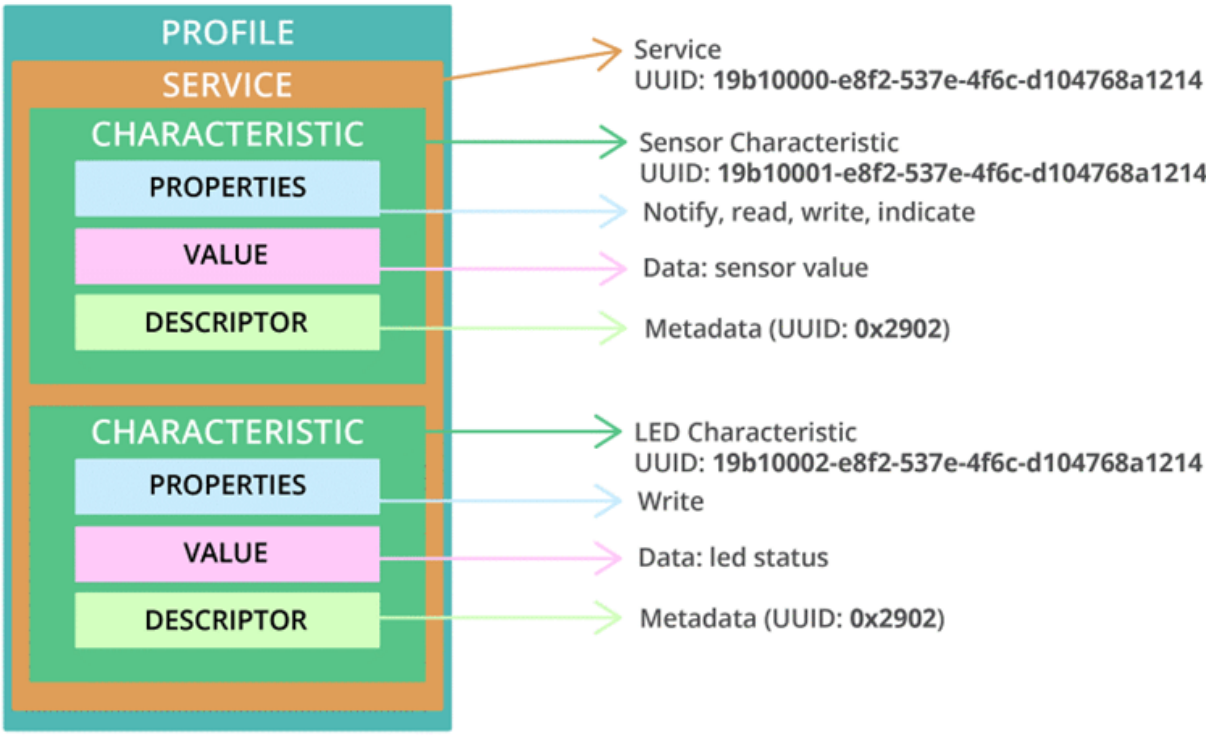


The server advertises its existence, so it can be found by other devices and contains data that the client can read or interact with. The client scans the nearby devices, and when it finds the server, it is looking for, it establishes a connection and can interact with that device by reading or writing to its characteristics.

The BLE server is basically the BLE peripheral before establishing a connection. The BLE Client is the BLE controller before establishing a connection. Many times, these terms are used interchangeably.

GATT

GATT, which stands for Generic Attribute Profile, is a fundamental concept in Bluetooth Low Energy (BLE) technology. Essentially, it serves as a blueprint for how BLE devices communicate with each other. Think of it as a structured language that two BLE devices use to exchange information seamlessly.



- **Profile:** standard collection of services for a specific use case;
- **Service:** collection of related information, like sensor readings, battery level, heart rate, etc.;
- **Characteristic:** it is where the actual data is saved on the hierarchy (value);
- **Descriptor:** metadata about the data;
- **Properties:** describe how the characteristic value can be interacted with. For example: read, write, notify, broadcast, indicate, etc.

Let's take a more in-depth look at the BLE Service and Characteristics.

BLE Service

The top level of the hierarchy is a profile, which is composed of one or more services. Usually, a BLE device contains more than one service, like the battery service and the heart rate service.

Every service contains at least one characteristic. There are predefined services for several types of data defined by the SIG (Bluetooth Special Interest Group) like: Battery Level, Blood Pressure, Heart Rate, Weight Scale, Environmental Sensing, etc. You can check the following link for predefined services:

- bluetooth.com/specifications/assigned-numbers/

UUID

A UUID is a unique digital identifier used in BLE and GATT to distinguish and locate services, characteristics, and descriptors. It's like a distinct label that ensures every component in a Bluetooth device has a unique name.

Each service, characteristic, and descriptor has a UUID (Universally Unique Identifier). A UUID is a unique 128-bit (16 bytes) number. For example:

55072829-bc9e-4c53-938a-74a6d4c78776

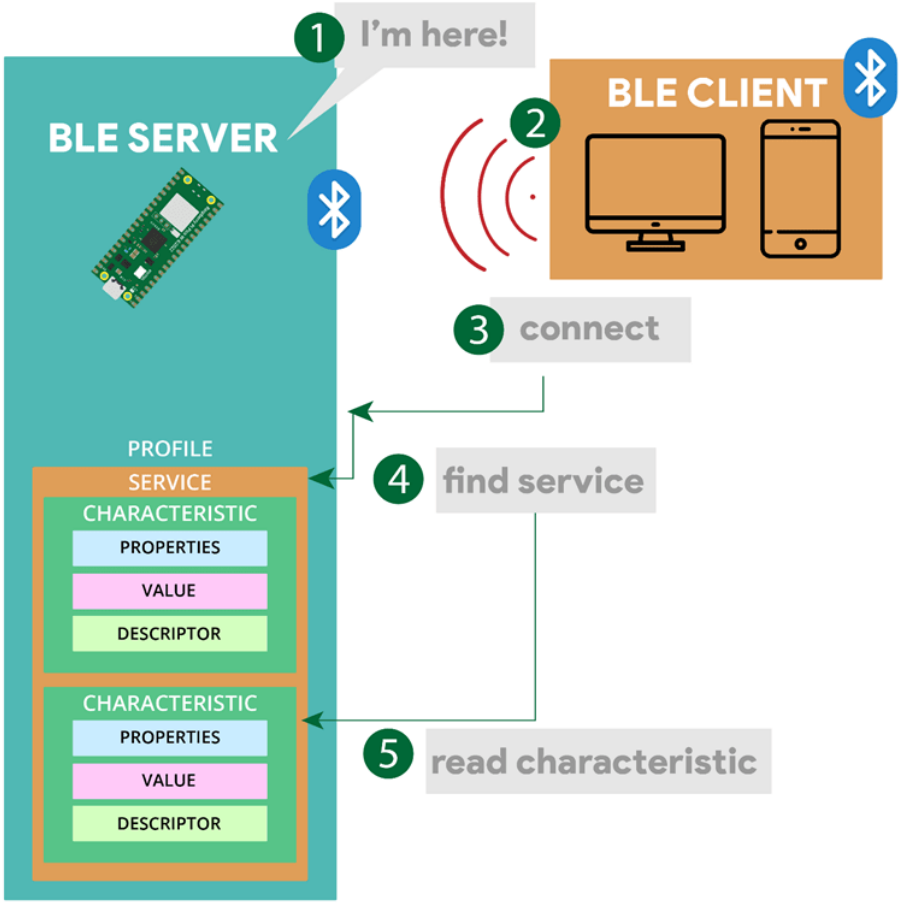
There are shortened and default UUIDs for services and characteristics specified in the SIG (Bluetooth Special Interest Group). This means that if you have a BLE device that uses the default UUIDs for its services and characteristics, you'll know exactly how to interact with that device to get or interact with the information you're looking for.

You can also generate your own custom UUIDs if you don't want to stick with predefined values or if the data you're exchanging doesn't fit in any of the categories.

You can generate custom UUIDs using this UUID generator website.

Communication between BLE Devices

Here are the usual steps that describe the communication between BLE Devices.



1. The BLE Peripheral (server) advertises its existence.

2. The BLE Central Device (client) scans for BLE devices.
3. When the central device finds the peripheral it is looking for, it connects to it.
4. After connecting, it reads the GATT profile of the peripheral and searches for the service it is looking for (for example: *environmental sensing*).
5. If it finds the service, it can now interact with the characteristics. For example, reading the temperature value.

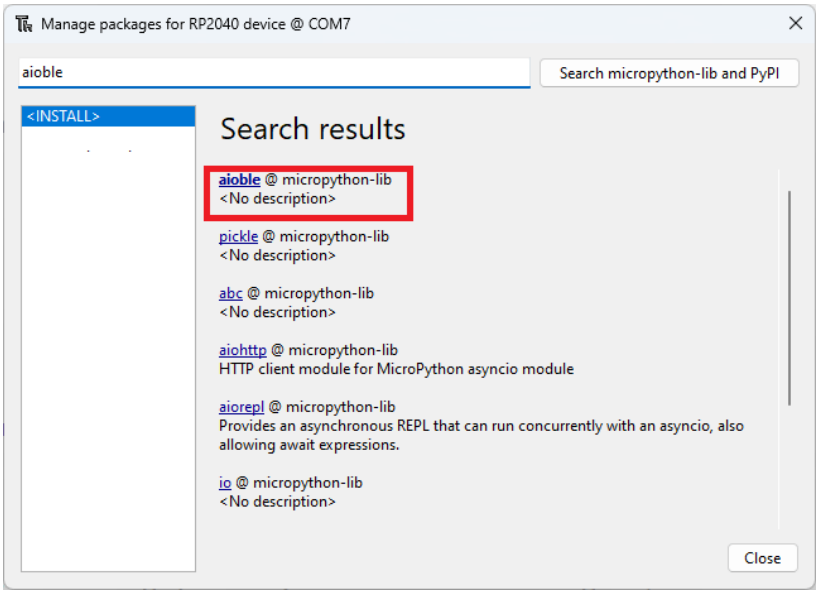
Installing the *aioble* Package

To write code to use Bluetooth with the Raspberry Pi Pico, we'll install the aioble package—that's currently the recommended library to use for BLE communication. Before proceeding to the actual examples, you need to install it on your board.

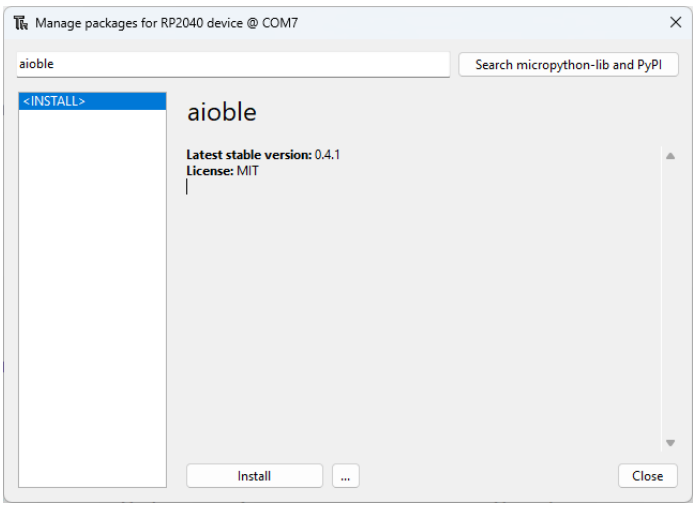
1) Connect the board to your computer and connect it to Thonny IDE.

2) On Thonny IDE, go to **Tools** > **Manage Packages…**

3) Search for *aioble* and click on the aioble option.



4) Finally, click the **Install** button.

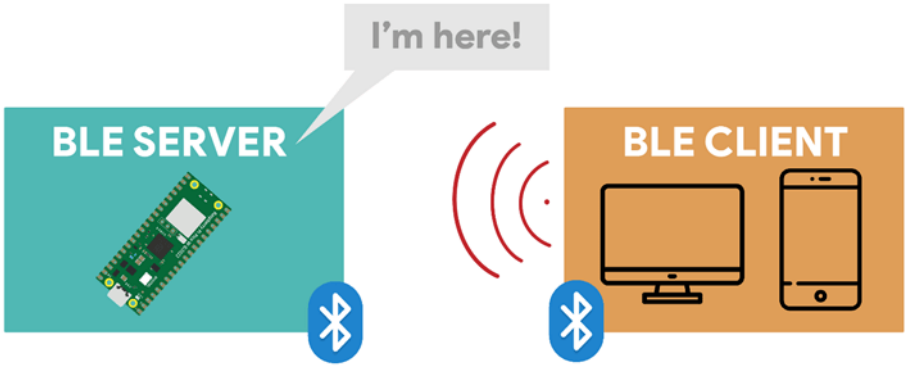


5) Wait a few seconds while it installs.

That's it. Now you can use the aioble functions to program the Raspberry Pi Pico.

Raspberry Pi Pico BLE Peripheral and Central Device – Communication Between Two Boards

For this example, you need two Raspberry Pi Pico W or 2W boards. We'll set one board as a BLE peripheral and another as a BLE Central device. You'll learn how the peripheral device advertises data and how the central device can read data from the peripheral device.



Only have one Raspberry Pi Pico W Board?

If you only have one Raspberry Pi Pico W board, you can only test and follow the peripheral device example, but you have to skip the central device.

BLE Peripheral

In this example, we'll create a BLE peripheral with the *Environmental Sensing Service* (a service defined by the SIG) with one characteristic for the *temperature*.

- **Service:***Environmental Sensing Service*
    - **Characteristic:** *Temperature*

We're going to use the default UUIDs for the Environmental Sensing Service and the Temperature characteristic.

Finding the Default UUIDs

If you go to this page and open the Assigned Numbers Document (PDF), you'll find all the default assigned UUID numbers. If you search for the *Environmental Sensing Service*, you'll find all the permitted characteristics that you can use with that service. You can see that it supports temperature.

6.1 Environmental Sensing Service

6.1.1 Permitted Characteristics

The list below specifies the characteristics that are permitted for use with the Environmental Sensing Service [7].

- Ammonia Concentration
- Apparent Wind Direction
- Apparent Wind Speed
- Barometric Pressure Trend
- Carbon Monoxide Concentration
- Dew Point
- Elevation
- Gust Factor
- Heat Index
- Humidity
- Irradiance
- Magnetic Declination
- Magnetic Flux Density - 2D
- Magnetic Flux Density - 3D
- Methane Concentration
- Nitrogen Dioxide Concentration
- Non-Methane Volatile Organic Compounds Concentration
- Ozone Concentration
- Particulate Matter - PM1 Concentration
- Particulate Matter - PM10 Concentration
- Particulate Matter - PM2.5 Concentration
- Pollen Concentration
- Pressure
- Rainfall
- Sulfur Dioxide Concentration
- Sulfur Hexafluoride Concentration
- Temperature
- True Wind Direction
- True Wind Speed
- UV Index
- Wind Chill

There's a table with the UUIDs for all services. You can see that the UUID for the Environmental Sensing service is **0x181A**.

Emergency Configuration service	0x183C
Environmental Sensing service	0x181A
Fitness Machine service	0x1826

Then, search for the temperature characteristic UUIDs. You'll find a table with the values for all characteristics. The UUID for the temperature is **0x2A6E**.

0x2A6D	Pressure
0x2A6E	Temperature
0x2A6F	Humidity
0x2A70	True Wind Speed

In summary, the UUIDs are:

- Environmental Sensing Service: **0x181A**
    - Temperature Characteristic: **0x2A6E**

Code – Raspberry Pi Pico W as a BLE Peripheral

The following code sets the Raspberry Pi Pico W or 2 W as a BLE peripheral with the *Environmental Sensing Service* and the *Temperature characteristic*. The BLE device will be writing on the *Temperature* characteristic continuously to update it with the latest temperature and will be advertising its service and characteristic. We'll get the temperature from the Raspberry Pi Pico on-board temperature sensor (**you need the picozero package installed**).

```
# Rui Santos & Sara Santos - Random Nerd Tutorials
# Complete project details at https://RandomNerdTutorials.com/raspberry-pi-pico-w-bluetooth-low-energy-micropython/

from micropython import const
import asyncio
import aioble
import bluetooth
import struct
from picozero import pico_temp_sensor

#org.bluetooth.service.environmental_sensing
_ENV_SENSE_UUID = bluetooth.UUID(0x181A)
# org.bluetooth.characteristic.temperature
_ENV_SENSE_TEMP_UUID = bluetooth.UUID(0x2A6E)
# org.bluetooth.characteristic.gap.appearance.xml
_ADV_APPEARANCE_GENERIC_THERMOMETER = const(768)
# How frequently to send advertising beacons.
_ADV_INTERVAL_MS = 250_000

# Register GATT server.
temp_service = aioble.Service(_ENV_SENSE_UUID)
temp_characteristic = aioble.Characteristic(
temp_service, _ENV_SENSE_TEMP_UUID, read=True, notify=True)
aioble.register_services(temp_service)

# Helper to encode the temperature characteristic encoding
# (sint16, hundredths of a degree).
def _encode_temperature(temp_deg_c):
    return struct.pack("<h", int(temp_deg_c * 100))
```

```
# Get temperature and update characteristic
async def sensor_task():
    while True:
        temperature = pico_temp_sensor.temp
        temp_characteristic.write(_encode_temperature(temperature), send_update=True)
        print(temperature)
        await asyncio.sleep_ms(1000)

# Serially wait for connections. Don't advertise while a central is connected.
async def peripheral_task():
    while True:
        try:
            async with await aioble.advertise(
                _ADV_INTERVAL_MS,
                name="RPi-Pico",
                services=[_ENV_SENSE_UUID],
                appearance=_ADV_APPEARANCE_GENERIC_THERMOMETER,
                ) as connection:
                    print("Connection from", connection.device)
                    await connection.disconnected()
        except asyncio.CancelledError:
            # Catch the CancelledError
            print("Peripheral task cancelled")
        except Exception as e:
            print("Error in peripheral_task:", e)
        finally:
            # Ensure the loop continues to the next iteration
            await asyncio.sleep_ms(100)

# Run both tasks
async def main():
    t1 = asyncio.create_task(sensor_task())
    t2 = asyncio.create_task(peripheral_task())
    await asyncio.gather(t1, t2)

asyncio.run(main())
```

View raw code

This example is based on the aioble example that you can find here.

How the Code Works

Let’s take a quick look at the relevant parts of the code for this example.

Include Libraries

You need to include the aioble and the bluetooth libraries to use Bluetooth with the Pico.

```
import aioble
import bluetooth
```

Our code will be asynchronous. For that, we'll use the asyncio library. We recommend following this tutorial to get familiar with asynchronous programming: Raspberry Pi Pico Asynchronous Programming – Run Multiple Tasks (MicroPython).

```
import asyncio
```

Define UUIDs and Register the GATT Service and Characteristic

We define the UUIDs for the environmental sensing service and for the temperature characteristic.

```
# org.bluetooth.service.environmental_sensing
_ENV_SENSE_UUID = bluetooth.UUID(0x181A)

# org.bluetooth.characteristic.temperature
_ENV_SENSE_TEMP_UUID = bluetooth.UUID(0x2A6E)
```

Then, register the GATT service and characteristic.

```
# Register GATT server.
temp_service = aioble.Service(_ENV_SENSE_UUID)
temp_characteristic = aioble.Characteristic(
    temp_service, _ENV_SENSE_TEMP_UUID, read=True, notify=True
)
aioble.register_services(temp_service)
```

When setting the characteristic, we set the *read* and *notify* arguments to True. This defines the way that the central device can interact with the characteristic. It can read the characteristic and be notified when it changes.

To write the actual temperature value to the temperature characteristic, it needs to be in a specific format. The _encode_temperature() function does that.

```
def _encode_temperature(temp_deg_c):
    return struct.pack("<h", int(temp_deg_c * 100))
```

Get Temperature and Write on Characteristic

We have an asynchronous function that gets temperature from the internal temperature sensor and writes to the temperature characteristic using the write() method on the temp_characteristic. This task is repeated continuously every second. You can adjust the delay time as needed.

```
# Get temperature and update characteristic
async def sensor_task():
    while True:
        temperature = pico_temp_sensor.temp
        temp_characteristic.write(_encode_temperature(temperature), send_update=True)
```

```
        print(temperature)
        await asyncio.sleep_ms(1000)
```

Advertising

Besides writing to the temperature characteristic, we also need to advertise the Raspberry Pi Pico as a BLE service. For that, we use the peripheral_task() function.

```
# Serially wait for connections. Don't advertise while a central is connected.
async def peripheral_task():
    while True:
        try:
            async with await aioble.advertise(
                _ADV_INTERVAL_MS,
                name="RPi-Pico",
                services=[_ENV_SENSE_UUID],
                appearance=_ADV_APPEARANCE_GENERIC_THERMOMETER,
            ) as connection:
                print("Connection from", connection.device)
                await connection.disconnected()
        except asyncio.CancelledError:
            # Catch the CancelledError
            print("Peripheral task cancelled")
        except Exception as e:
            print("Error in peripheral_task:", e)
        finally:
            # Ensure the loop continues to the next iteration
            await asyncio.sleep_ms(100)
```

In that function, we define the BLE device name.

```
name="RPi-Pico",
```

You can change its name if you want to. In the next example, we'll connect to this device by referring to its name, so you'll need to make sure you use the same name in both codes, as we'll see later.

Main Function

Finally, we create an asynchronous main() function, where we'll write the base for our code. We create two asynchronous tasks: one for advertising and another to write on the temperature characteristic.

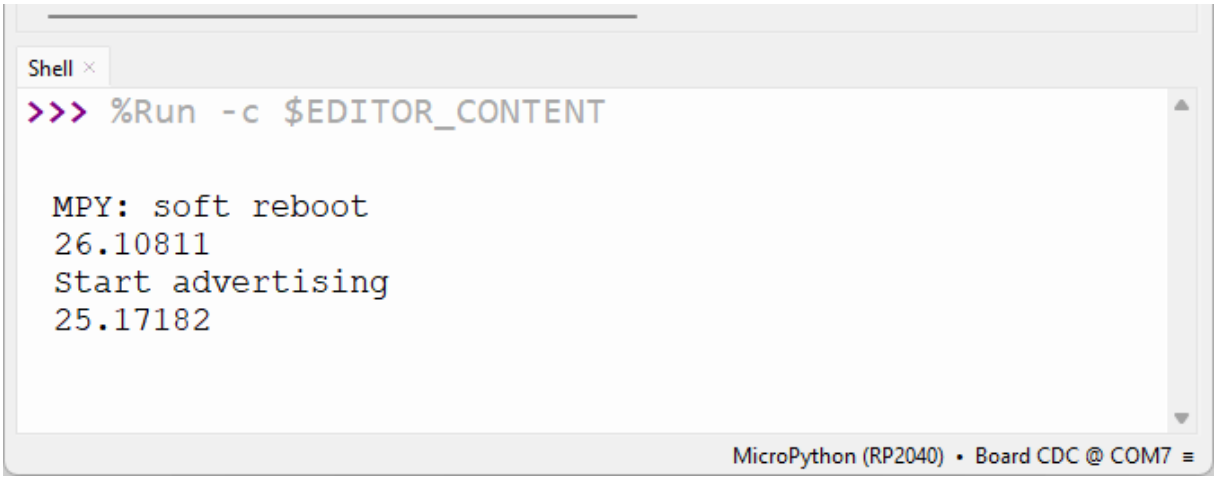
```
# Run both tasks.
async def main():
    t1 = asyncio.create_task(sensor_task())
    t2 = asyncio.create_task(peripheral_task())
```

```
    await asyncio.gather(t1, t2)

asyncio.run(main())
```

Testing the Code

Run the previous code on your Raspberry Pi Pico. It will start writing the temperature on the temperature characteristic and will advertise its service.

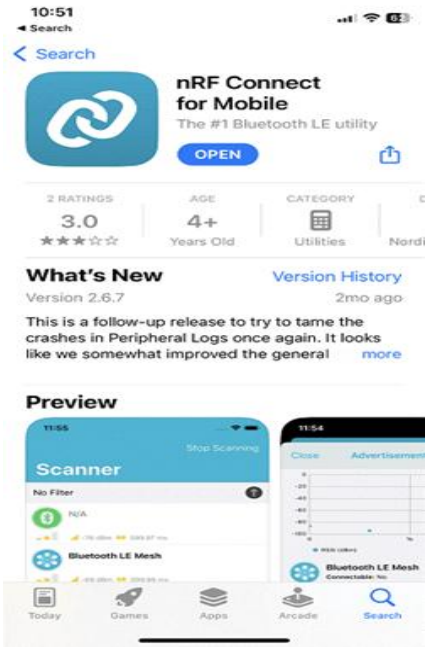


To connect to this peripheral, and read its characteristic we'll create a BLE Central device on another Raspberry Pi Pico board.

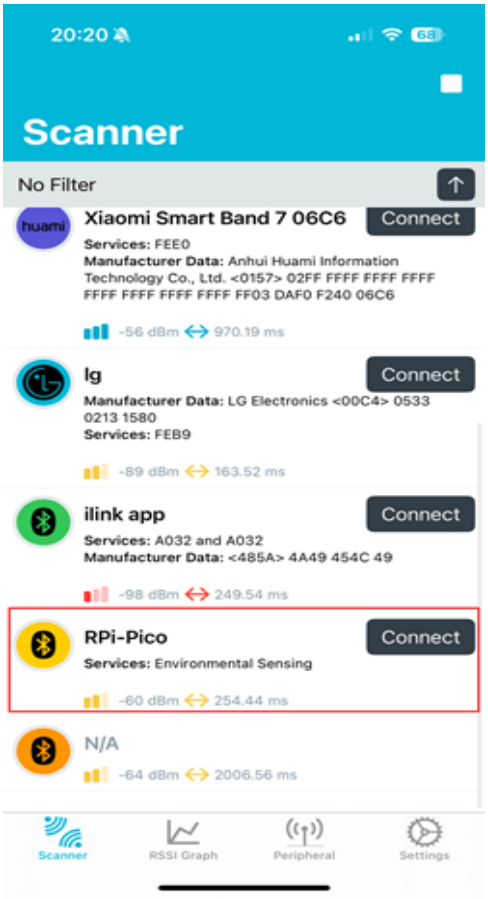
nRF Connect App

Before proceeding, you can use an app like ***nRF Connect*** to search for BLE devices and read their characteristics.

The *nRF Connect* app from Nordic works on Android (Google Play Store) and iOS (App Store). Go to Google Play Store or App Store, search for "**nRF Connect for Mobile**" and install the app on your smartphone.



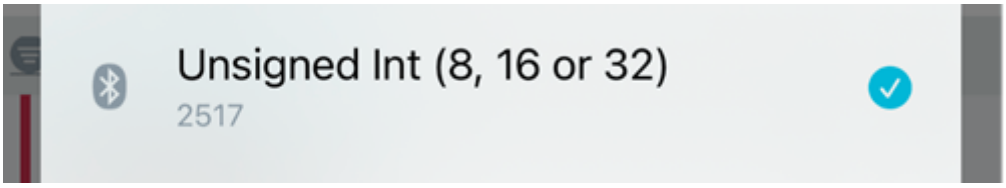
Go to your smartphone, open the *nRF Connect app from Nordic*, and start scanning for new devices. You should find a device called **RPi-Pico**—this is the BLE server name you defined earlier.



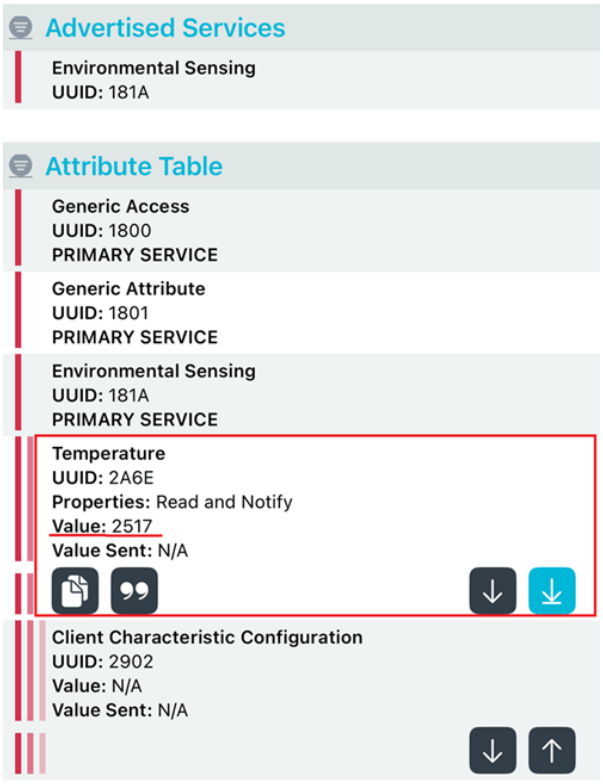
Connect to it. You'll see that it displays the ***Environmental Sensing*** service with the ***temperature*** characteristic. Click on the arrows to read the characteristic and activate the notifications.

Then, click on the second icon on the left to change the format (this option may only be available for iPhone, and it will automatically display the temperature in the correct format).

You can change the format to Unsigned Int.



You'll start seeing the temperature values being reported every 10 seconds. It will display the temperature in Celsius multiplied by 100 (on the iPhone).



Now that we know that our BLE peripheral is working as expected, we can program another Raspberry Pi Pico board as a BLE Central device to read the temperature characteristic from this board.

Code – Raspberry Pi Pico as a BLE Central Device (BLE Client)

The following code should be run on another Raspberry Pi Pico board. We'll set it as a BLE Central device that will look for the **RPi-Pico** device, search for the *environmental sensing service*, and read the *temperature* from the characteristic.

```
# Rui Santos & Sara Santos - Random Nerd Tutorials
# Complete project details at https://RandomNerdTutorials.com/raspberry-pi-pico-w-bluetooth-low-energy-micropython/

from micropython import const
import uasyncio as asyncio
import aioble
import bluetooth
import struct

# org.bluetooth.service.environmental_sensing
_ENV_SENSE_UUID = bluetooth.UUID(0x181A)
# org.bluetooth.characteristic.temperature
_ENV_SENSE_TEMP_UUID = bluetooth.UUID(0x2A6E)

# Name of the peripheral you want to connect
peripheral_name="RPi-Pico"
```

```
# Helper to decode the temperature characteristic encoding (sint16, hundredths of a
degree).
def _decode_temperature(data):
    try:
        if data is not None:
            return struct.unpack("<h", data)[0] / 100
    except Exception as e:
        print("Error decoding temperature:", e)
    return None

async def find_temp_sensor():
    # Scan for 5 seconds, in active mode, with a very low interval/window (to
    # maximize detection rate).
    async with aioble.scan(5000, interval_us=30000, window_us=30000, active=True) as
scanner:
        async for result in scanner:
            print(result.name())
            # See if it matches our name and the environmental sensing service.
            if result.name() == peripheral_name and _ENV_SENSE_UUID in result.services():
                return result.device
    return None

async def main():
    while True:
        device = await find_temp_sensor()
        if not device:
            print("Temperature sensor not found. Retrying...")
            await asyncio.sleep_ms(5000)  # Wait for 5 seconds before retrying
            continue

        try:
            print("Connecting to", device)
            connection = await device.connect()
        except asyncio.TimeoutError:
            print("Timeout during connection. Retrying...")
            await asyncio.sleep_ms(5000)  # Wait for 5 seconds before retrying
            continue

        async with connection:
            try:
                temp_service = await connection.service(_ENV_SENSE_UUID)
                temp_characteristic = await temp_service.characteristic(_ENV_SENSE_TEMP_UUID)
```

```
        except asyncio.TimeoutError:
            print("Timeout discovering services/characteristics. Retrying...")
            await asyncio.sleep_ms(5000)  # Wait for 5 seconds before retrying
            continue

        while True:
            try:
                temp_data = await temp_characteristic.read()
                if temp_data is not None:
                    temp_deg_c = _decode_temperature(temp_data)
                    if temp_deg_c is not None:
                        print("Temperature: {:.2f}".format(temp_deg_c))
                    else:
                        print("Invalid temperature data")
                else:
                    print("Error reading temperature: None")
            except Exception as e:
                print("Error in main loop:", e)
                break  # Break out of the inner loop and attempt to reconnect

            await asyncio.sleep_ms(1000)

# Create an Event Loop
loop = asyncio.get_event_loop()
# Create a task to run the main function
loop.create_task(main())

try:
    # Run the event loop indefinitely
    loop.run_forever()
except Exception as e:
    print('Error occurred: ', e)
except KeyboardInterrupt:
    print('Program Interrupted by the user')
```

View raw code

This example is based on the aioble example that you can find here.

How the Code Works

Let's take a quick look at the relevant parts of the code for this example.

Include Libraries

You need to include the aioble and the bluetooth libraries to use Bluetooth with the Pico.

```
import aioble
```

```
import bluetooth
```

Our code will be asynchronous. For that, we'll use the asyncio library. We recommend following this tutorial to get familiar with asynchronous programming: Raspberry Pi Pico Asynchronous Programming – Run Multiple Tasks (MicroPython).

```
import asyncio
```

Define UUIDs

We define the UUIDs for the environmental sensing service and for the temperature characteristic that we want to read.

```
# org.bluetooth.service.environmental_sensing
_ENV_SENSE_UUID = bluetooth.UUID(0x181A)

# org.bluetooth.characteristic.temperature
_ENV_SENSE_TEMP_UUID = bluetooth.UUID(0x2A6E)
```

BLE Device Name We Want to Connect To

Then, we define the name of the BLE device that we want to connect to. It must be the name of the BLE peripheral you set up previously, in our case **RPi-Pico**.

```
# Name of the peripheral you want to connect
peripheral_name="RPi-Pico"
```

The temperature is saved on a characteristic in a specific format. To help us transform that value into a value that we can read, we use the _decode_temperature() function.

```
# Helper to decode the temperature characteristic encoding
# (sint16, hundredths of a degree).
def _decode_temperature(data):
    try:
        if data is not None:
            return struct.unpack("<h", data)[0] / 100
    except Exception as e:
        print("Error decoding temperature:", e)
    return None
```

Scanning and Searching for BLE Devices

The find_temp_sensor() function will scan for nearby BLE devices and when it finds the BLE device with the name we've defined previously, it returns that BLE device with result.device.

```
async def find_temp_sensor():
    # Scan for 5 seconds, in active mode, with a very low interval/window
    # (to maximize detection rate)
```

```
  async with aioble.scan(5000, interval_us=30000, window_us=30000, active=True) as
scanner:
      async for result in scanner:
         print(result.name())
         # See if it matches our name and the environmental sensing service.
         if result.name() == peripheral_name and _ENV_SENSE_UUID in result.services():
            return result.device
  return None
```

Main Function

In the main() async function, we first try to find the BLE device we're looking for by calling the find_temp_sensor(). We save the BLE device on the device variable.

```
device = await find_temp_sensor()
```

Then, we try to connect to that device:

```
try:
  print("Connecting to", device)
  connection = await device.connect()
```

After getting a successful connection, we get the temperature service on the temp_service variable. After that, we use that service to get the characteristic we're looking for and save it on the temp_characteristic variable.

```
async with connection:
  try:
    temp_service = await connection.service(_ENV_SENSE_UUID)
    temp_characteristic = await temp_service.characteristic(_ENV_SENSE_TEMP_UUID)
```

Finally, we can get the temperature value by reading the characteristic temp_characteristic.read().

```
while True:
  try:
    temp_data = await temp_characteristic.read()
    if temp_data is not None:
       temp_deg_c = _decode_temperature(temp_data)
       if temp_deg_c is not None:
          print("Temperature: {:.2f}".format(temp_deg_c))
        else:
          print("Invalid temperature data")
      else:
        print("Error reading temperature: None")
   except Exception as e:
      print("Error in main loop:", e)
      break  # Break out of the inner loop and attempt to reconnect
   await asyncio.sleep_ms(1000)
```

We create an event loop and associate the main() task to that event.

```
# Create an Event Loop
loop = asyncio.get_event_loop()
# Create a task to run the main function
loop.create_task(main())
```

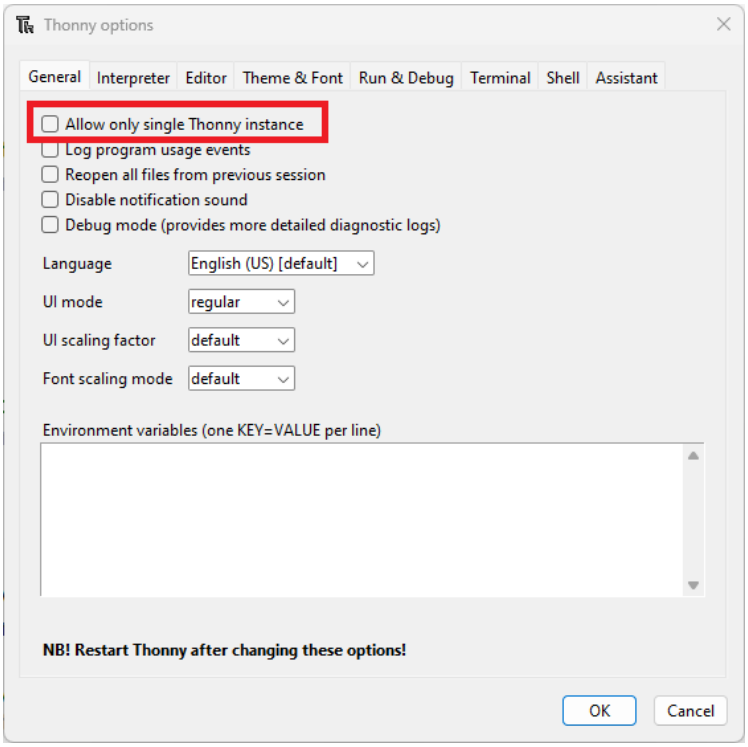
Finally, we run the event loop to keep our code running.

```
# Run the event loop indefinitely
loop.run_forever()
```

Testing the Code

For testing this code, it will be useful to have two instances of Thonny IDE running so that we can run code on both devices and see the console for both connections.

To activate multiple instances of Thonny IDE, go to **Tools** > **Options** and untick the *Allow only single Thonny instance* option.

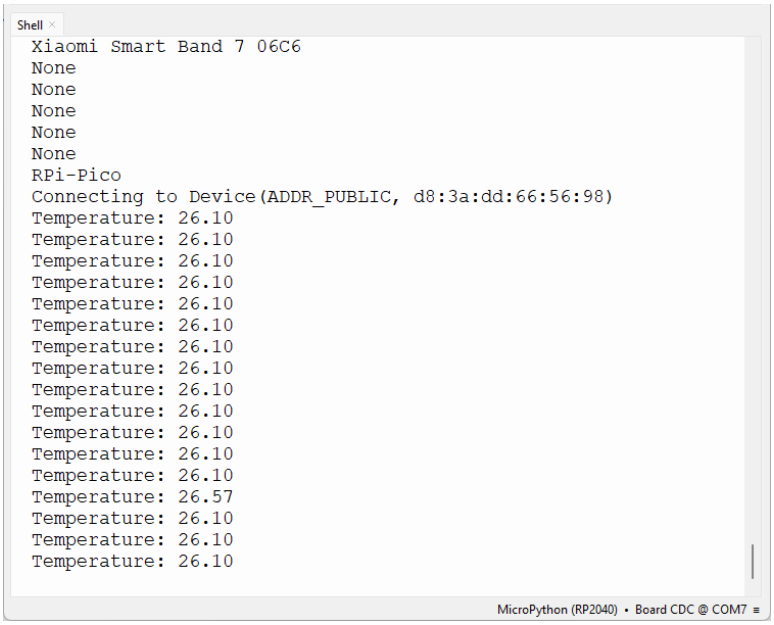


Close Thonny for the changes to take effect. Then, simply open Thonny IDE on your computer twice to open two windows.

Make sure you have each window connected to the right board. You can select the COM port at the bottom right corner.

After having the BLE peripheral device running (make sure it is advertising the temperature), copy the code provided in this section to the other Thonny IDE instance (make sure it is connected to the right COM port). Then, run it on this new board.

It will start scanning for BLE devices. When it finds the **RPi-Pico** device, it will connect to it and display the temperature values sent by the other board on the Shell.



Wrapping Up

In this tutorial, you learned the basics of Bluetooth Low Energy with the Raspberry Pi Pico when programmed with MicroPython. You learned how to set the Pico as a BLE Central Device and as a BLE Peripheral Device. You learned how to set and read characteristic values.

To further develop this project, you can add a BME280 or another sensor of your choice and display temperature, humidity, and pressure characteristics. You can also add an OLED or LCD to the BLE Central device to display the readings from the other board.

More BLE Resources

To explore more about Bluetooth, you can take a look at the following resources:

- Bluetooth Documentation for MicroPython (low-level functions):
  - docs.micropython.org/en/latest/library/bluetooth.html
- MicroPython Bluetooth examples (low-level functions):
  - github.com/micropython/micropython/tree/master/examples/bluetooth
- aioble library page with examples:
  - github.com/micropython/micropython-lib/tree/master/micropython/bluetooth/aioble

If you want to learn more about the Raspberry Pi Pico, check out our resources:

- **Learn Raspberry Pi Pico/Pico W with MicroPython (eBook)**
- Free Raspberry Pi Pico projects and tutorials

We hope you've found this guide useful. We have similar guides for the ESP32 boards:

- MicroPython: ESP32 – Getting Started with Bluetooth Low Energy (BLE)
- Getting Started with ESP32 Bluetooth Low Energy (BLE) on Arduino IDE