Python Radio 23: Bluetooth Low Energy

Simon Quellen Field

Simon Quellen Field

Follow

6 min read

·

Sep 10, 2024

Listen

Share

More

Designed by committee…

Press enter or click to view image in full size

MidJourney

Wifi is not too difficult to set up. Micropython is well supplied with classes built-in to handle Wifi. A web server may be a fair chunk of code, but it is easy to set up and configure. Bluetooth is another matter. Bluetooth was designed to handle a large number of use cases, and as a result is quite complex, and doing something simple is often quite tedious.

If we wanted to send temperature data from one computer to another using the HC-12, we would simply open up a UART and send the data.

This is what you have to do with Bluetooth:

Import bluetooth

Import random

Import struct

Import time

From ble\_advertising import advertising\_payload

From micropython import const

\_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=”mpy-temp”):

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()

Self.\_payload = advertising\_payload(

Name=name, services=[\_ENV\_SENSE\_UUID], appearance=\_ADV\_APPEARANCE\_GENERIC\_THERMOMETER

)

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 set\_temperature(self, temp\_deg\_c, notify=False, indicate=False):

# Data is sint16 in degrees Celsius with a resolution of 0.01 degrees Celsius.

# Write the local value, ready for a central to read.

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)

We have to create a service and advertise that it is a sensor and returns temperature data. We must indicate that it can be read, it can notify, and it can indicate.

We give it a name “mpy-temp” so that other devices can find its advertisements and pair with it. We have to track connections and disconnections.

In the micropython library is a package called aioble which makes much of the BLE processing a little simpler. You can find it at <https://github.com/micropython/micropython-lib/tree/master/micropython/bluetooth/aioble>.

Here is the temperature sensor example from that library:

Import sys

Sys.path.append(”)

From micropython import const

Import uasyncio as asyncio

Import aioble

Import bluetooth

Import random

Import struct

# 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))

# This would be periodically polling a hardware sensor.

Async def sensor\_task():

T = 24.5

While True:

Temp\_characteristic.write(\_encode\_temperature(t))

T += random.uniform(-0.5, 0.5)

Await asyncio.sleep\_ms(1000)

# Serially wait for connections. Don’t advertise while a central is

# connected.

Async def peripheral\_task():

While True:

Async with await aioble.advertise(

\_ADV\_INTERVAL\_MS,

Name=”mpy-temp”,

Services=[\_ENV\_SENSE\_UUID],

Appearance=\_ADV\_APPEARANCE\_GENERIC\_THERMOMETER,

) as connection:

Print(”Connection from”, connection.device)

Await connection.disconnected()

# 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())

It isn’t a lot smaller, but it is perhaps a little easier to understand and modify.

The aioble library has an example file server as well, so your devices can store and serve files to one another, or to phones, tablets, laptops, or other bluetooth clients.

To get the aioble library onto your ESP32, run this small main.py program:

From mip import install

From network import WLAN, STA\_IF

Def main():

Wlan = WLAN(STA\_IF)

Wlan.active(True)

Wlan.connect(”BirdfarmOffice2”,”12345678”) # SSID and password

Print(wlan.isconnected())

Install(”aioble”)

Main()

You would use your own Wifi SSID and password of course, instead of mine.

Or, you can run the following version:

From connect import Connect

From mip import install

Def main():

Con = Connect(”Get Library”)

Con.reconnect()

Install(”aioble”)

Main()

It uses the connect.py module from the previous project and the network.cfg file from there. It connects to the Internet using your local Wifi, so that it can then download the library from micropython.org.

There are many libraries there that you can download.

Once a library is downloaded, it resides on your flash filesystem until you erase the chip, so you only need to run that program once.

Suppose we wish to know when a particular Bluetooth-enabled device is in the vicinity. This probably won’t work to track people, or tell you when someone special has arrived home or left home, as phones and watches change their addresses often to prevent this. But two of our own devices can find each other this way and tell roughly how far away they are from one another. A program to do that would look like this:

From time import time, localtime

From ntptime import settime

From connect import Connect

From uasyncio import get\_event\_loop, sleep

TIMEZONE = -8 \* 60 \* 60

Devices = {}

Async def lookie():

From aioble import scan

Async with scan(duration\_ms=5000, interval\_us=30000, window\_us=30000, active=True) as scanner:

Async for result in scanner:

My\_time = localtime(time() + TIMEZONE)

Time\_str = my\_time

Devices[str(result.device.addr\_hex())] = [result.rssi, result.device.addr\_hex(), time(), result.name()]

Lookfor = “5a:5d:bb:ef:0c:64”

Async def printem():

For x in devices:

If x == lookfor:

Print(”\* “, end=”)

Else:

Print(” “, end=”)

Print(x, end=”: “)

Count = 0

For y in devices[x]:

If count == 1: # address already printed

Pass

Elif count == 2: # The timestamp

Diff = time() – y

If diff < 10:

Print(”just now”, end=”, “)

Elif diff < 60:

Print(str(diff) + “ seconds ago”, end=”, “)

Elif diff < 120:

Print(”a minute ago”, end=”, “)

Elif diff < 3600:

Print(str(round(diff/60)) + “ minutes ago”, end=”, “)

Elif diff < 7200:

Print(”an hour ago”, end=”, “)

Elif diff < 86400:

Print(str(round(diff/3600)) + “ hours ago”, end=”, “)

Else:

Print(”long ago”)

Elif count == 3:

Print(y)

Else:

Print(y, end=”, “)

Count += 1

Print()

Def main():

Con = Connect(”BLE\_Scanner”)

Con.reconnect()

# from mip import install

# install(”aioble”)

# import mip

Settime()

Def scan\_task():

While True:

Await lookie()

Def print\_task():

While True:

Await printem()

Await sleep(10)

Loop = get\_event\_loop()

Loop.create\_task(scan\_task())

Loop.create\_task(print\_task())

Loop.run\_forever()

Loop.close()

Main()

The main program sets up two tasks that will run in parallel. One task scans for Bluetooth devices and records them in a dictionary.

The other task reads the dictionary and prints out the information every ten seconds.

If the target address is found, it puts an asterisk in front of the line.

The output looks like this:

79:3c:0c:8d:f9:a2: -74, 4 minutes ago, None

54:e9:b0:b9:8d:37: -78, a minute ago, None

55:77:11:96:f3:4a: -69, 3 minutes ago, None

45:0d:0b:7f:77:15: -77, 5 minutes ago, None

F2:69:bb:f8:fe:ba: -64, just now, Two\_feba

F8:04:2e:86:42:ab: -50, just now, [TV] office

5a:ad:96:0f:f0:e0: -71, just now, None

50:68:34:35:88:61: -35, 3 minutes ago, None

46:3c:d9:72:c8:eb: -36, 2 minutes ago, None

54:e5:88:49:66:d1: -78, 10 minutes ago, None

* D5:ef:31:87:5b:cb: -49, just now, Andy\_5bcb

5b:58:f6:85:2c:49: -66, 3 minutes ago, None

76:79:7d:da:b5:e1: -69, 47 seconds ago, None

76:66:86:81:bb:3a: -78, 13 minutes ago, None

Ec:85:48:29:a0:1f: -56, just now, Echo\_a01f

4e:78:cb:24:0e:f0: -74, 2 minutes ago, Galaxy Watch6 (MHAZ)

78:25:33:e4:e5:c6: -67, 37 seconds ago, None

66:03:6d:61:85:22: -36, 10 seconds ago, None

50:e7:a6:ba:00:29: -67, 2 minutes ago, None

47:e6:6c:33:0a:12: -62, 13 minutes ago, None

5a:38:26:63:55:fd: -59, 4 minutes ago, None

5b:a6:45:35:eb:1d: -78, 3 minutes ago, None

49:c9:69:59:9e:2e: -76, 4 minutes ago, None

Notice that the Galaxy Watch announces itself by name, making it easy to tell if the only person in the house with a Galaxy Watch happens to be at home. The Pixel phone and Pixel Watch do not, and the program returns None for the name.

You can find out what the address of a device is by simply holding it close to our ESP32 while we are scanning. The RSSI (Received Signal Strength Indicator) value will rise (become less negative) as the device nears.

Our program needs an Internet connection to get the correct time by way of NTP (the Network Time Protocol). We import settime() from the module ntptime to set our clock properly. Being able to do this can be quite useful to synchronize two radios, so that one will transmit to the other at an exact time, making decoding easier in some protocols.

Ble

Bluetooth Low Energy

Python Programming

Esp32