

# Getting Started with ESP32 Bluetooth Low Energy (BLE) on Arduino IDE

The ESP32 comes not only with Wi-Fi but also with Bluetooth and Bluetooth Low Energy (BLE). This post is a quick introduction to BLE with the ESP32. First, we'll explore what's BLE and what it can be used for, and then we'll take a look at some examples with the ESP32 using Arduino IDE. For a simple introduction we'll create an ESP32 BLE server, and an ESP32 BLE scanner to find that server.

## Introducing Bluetooth Low Energy

For a quick introduction to BLE, you can watch the video below, or you can scroll down for a written explanation.

**Recommended reading:** learn how to use [ESP32 Bluetooth Classic with Arduino IDE](#) to exchange data between an ESP32 and an Android smartphone.

## What is Bluetooth Low Energy?

Bluetooth Low Energy, BLE for short, is a power-conserving variant of Bluetooth. BLE's primary application is short distance transmission of small amounts of data (low bandwidth). Unlike Bluetooth that is always on, BLE remains in sleep mode constantly except for when a connection is initiated.

This makes it consume very low power. BLE consumes approximately 100x less power than Bluetooth (depending on the use case).



Additionally, BLE supports not only point-to-point communication, but also broadcast mode, and mesh network.

Take a look at the table below that compares BLE and [Bluetooth Classic](#) in more detail.

---

	Bluetooth Low Energy (LE)	Bluetooth Basic Rate/ Enhanced Data Rate (BR/EDR)
Optimized For...	Short burst data transmission	Continuous data streaming
Frequency Band	2.4GHz ISM Band (2.402 – 2.480 GHz Utilized)	2.4GHz ISM Band (2.402 – 2.480 GHz Utilized)
Channels	40 channels with 2 MHz spacing (3 advertising channels/37 data channels)	79 channels with 1 MHz spacing
Channel Usage	Frequency-Hopping Spread Spectrum (FHSS)	Frequency-Hopping Spread Spectrum (FHSS)
Modulation	GFSK	GFSK, $\pi/4$ DQPSK, 8DPSK
Power Consumption	~0.01x to 0.5x of reference (depending on use case)	1 (reference value)
Data Rate	LE 2M PHY: 2 Mb/s LE 1M PHY: 1 Mb/s LE Coded PHY (S=2): 500 Kb/s LE Coded PHY (S=8): 125 Kb/s	EDR PHY (8DPSK): 3 Mb/s EDR PHY ( $\pi/4$ DQPSK): 2 Mb/s BR PHY (GFSK): 1 Mb/s
Max Tx Power*	Class 1: 100 mW (+20 dBm) Class 1.5: 10 mW (+10 dBm) Class 2: 2.5 mW (+4 dBm) Class 3: 1 mW (0 dBm)	Class 1: 100 mW (+20 dBm) Class 2: 2.5 mW (+4 dBm) Class 3: 1 mW (0 dBm)
Network Topologies	Point-to-Point (including piconet) Broadcast Mesh	Point-to-Point (including piconet)

[View Image Souce](#)

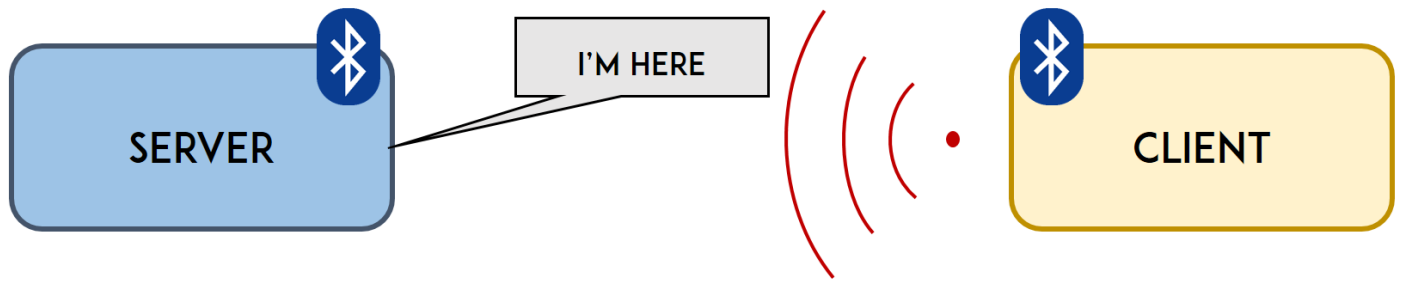
Due to its properties, BLE is suitable for applications that need to exchange small amounts of data periodically running on a coin cell. For example, BLE is of great use in healthcare, fitness, tracking, beacons, security, and home automation industries.



## BLE Server and Client

With Bluetooth Low Energy, there are two types of devices: the server and the client. The ESP32 can act either as a client or as a server.

The server advertises its existence, so it can be found by other devices, and contains the data that the client can read. The client scans the nearby devices, and when it finds the server it is looking for, it establishes a connection and listens for incoming data. This is called point-to-point communication.



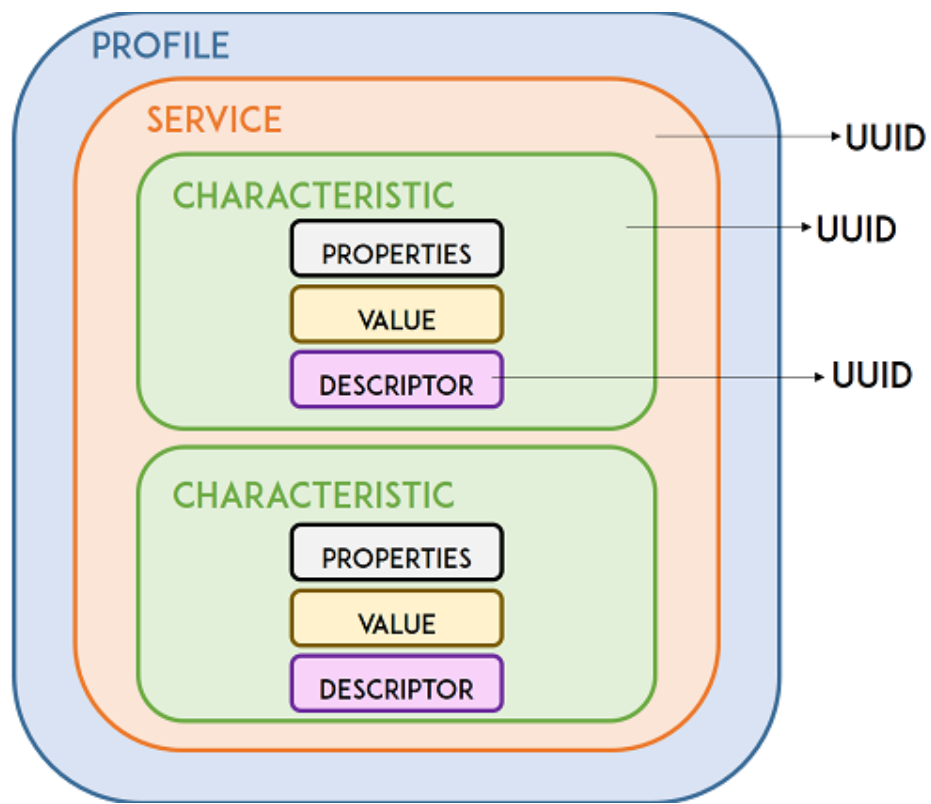
As mentioned previously, BLE also supports broadcast mode and mesh network:

- **Broadcast mode:** the server transmits data to many clients that are connected;
- **Mesh network:** all the devices are connected, this is a many to many connection.

Even though the broadcast and mesh network setups are possible to implement, they were developed very recently, so there aren't many examples implemented for the ESP32 at this moment.

## GATT

GATT stands for Generic Attributes and it defines an hierarchical data structure that is exposed to connected BLE devices. This means that GATT defines the way that two BLE devices send and receive standard messages. Understanding this hierarchy is important, because it will make it easier to understand how to use the BLE and write your applications.



## 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.

Every service contains at least one characteristic, or can also reference other services. A service is simply a collection of information, like sensor readings, for example.

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, etc. You can [check here other defined services](#).

Name	Uniform Type Identifier	Assigned Number	Specification
Generic Access	org.bluetooth.service.generic_access	0x1800	GSS
Alert Notification Service	org.bluetooth.service.alert_notification	0x1811	GSS
Automation IO	org.bluetooth.service.automation_io	0x1815	GSS
Battery Service	org.bluetooth.service.battery_service	0x180F	GSS
Blood Pressure	org.bluetooth.service.blood_pressure	0x1810	GSS
Body Composition	org.bluetooth.service.body_composition	0x181B	GSS
Bond Management Service	org.bluetooth.service.bond_management	0x181E	GSS
Continuous Glucose Monitoring	org.bluetooth.service.continuous_glucose_monitoring	0x181F	GSS
Current Time Service	org.bluetooth.service.current_time	0x1805	GSS
Cycling Power	org.bluetooth.service.cycling_power	0x1818	GSS
Cycling Speed and Cadence	org.bluetooth.service.cycling_speed_and_cadence	0x1816	GSS
Device Information	org.bluetooth.service.device_information	0x180A	GSS
Environmental Sensing	org.bluetooth.service.environmental_sensing	0x181A	GSS
Fitness Machine	org.bluetooth.service.fitness_machine	0x1826	GSS
Generic Attribute	org.bluetooth.service.generic_attribute	0x1801	GSS

[View Image Souce](#)

## BLE Characteristic

The characteristic is always owned by a service, and it is where the actual data is contained in the hierarchy (value). The characteristic always has two attributes: characteristic declaration (that provides metadata about the data) and the characteristic value.

Additionally, the characteristic value can be followed by descriptors, which further expand on the metadata contained in the characteristic declaration.

The properties describe how the characteristic value can be interacted with. Basically, it contains the operations and procedures that can be used with the characteristic:

- Broadcast
- Read
- Write without response
- Write
- Notify
- Indicate
- Authenticated Signed Writes
- Extended Properties

# UUID

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

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

There are shortened UUIDs for all types, services, and profiles specified in the [SIG \(Bluetooth Special Interest Group\)](#).

But if your application needs its own UUID, you can generate it using this [UUID generator website](#).

In summary, the UUID is used for uniquely identifying information. For example, it can identify a particular service provided by a Bluetooth device.

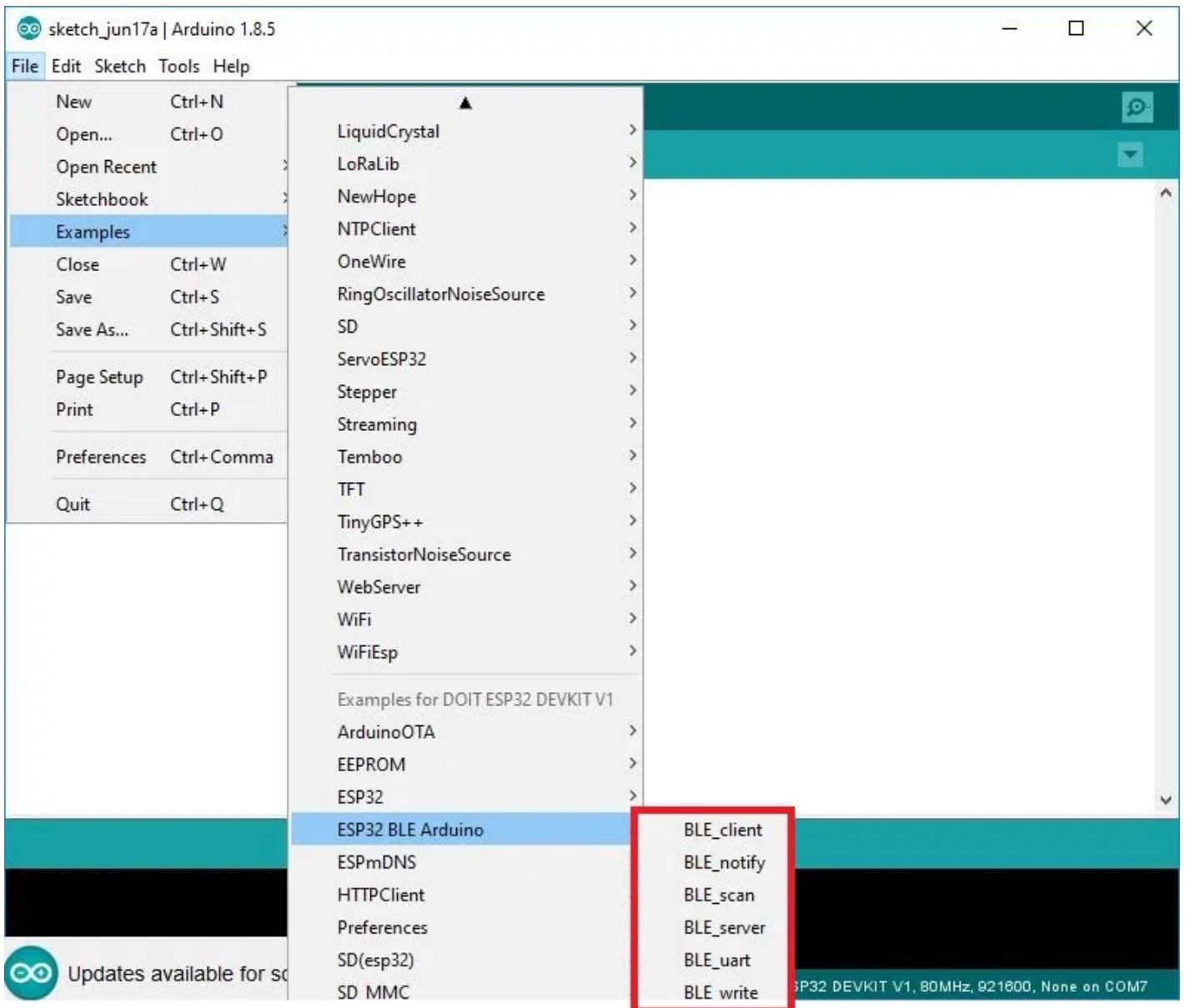
## BLE with ESP32

The ESP32 can act as a BLE server or as a BLE client. There are several BLE examples for the ESP32 in the [ESP32 BLE library for Arduino IDE](#). This library comes installed by default when you install the ESP32 on the Arduino IDE.

**Note:** You need to have the ESP32 add-on installed on the Arduino IDE. Follow one of the next tutorials to prepare your Arduino IDE to work with the ESP32, if you haven't already.

- [Windows instructions – ESP32 Board in Arduino IDE](#)
- [Mac and Linux instructions – ESP32 Board in Arduino IDE](#)

In your Arduino IDE, you can go to **File > Examples > ESP32 BLE Arduino** and explore the examples that come with the BLE library.



**Note:** to see the ESP32 examples, you must have the ESP32 board selected on **Tools > Board**.

For a brief introduction to the ESP32 with BLE on the Arduino IDE, we'll create an ESP32 BLE server, and then an ESP32 BLE scanner to find that server. We'll use and explain the examples that come with the BLE library.

To follow this example, you need two ESP32 development boards. We'll be using the [ESP32 DOIT DEVKIT V1 Board](#).

## ESP32 BLE Server

To create an ESP32 BLE Server, open your Arduino IDE and go to **File > Examples > ESP32 BLE Arduino** and select the **BLE\_server** example. The following code should load:

/\*

Based on Neil Kolban example for IDF: [https://github.com/nkolban/ESP32\\_BLE\\_Arduino](https://github.com/nkolban/ESP32_BLE_Arduino)  
Ported to Arduino ESP32 by Evandro Copercini  
updates by chegewara

\*/

```
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEServer.h>

// See the following for generating UUIDs:
// https://www.uuidgenerator.net/

#define SERVICE_UUID          "4fafc201-1fb5-459e-8fcc-c5c9c331914"
#define CHARACTERISTIC_UUID   "beb5483e-36e1-4688-b7f5-ea07361b26c"

void setup() {
  Serial.begin(115200);
  Serial.println("Starting BLE work!");

  BLEDevice::init("Long name works now");
  BLEServer *pServer = BLEDevice::createServer();
  BLEService *pService = pServer->createService(SERVICE_UUID);
  BLECharacteristic *pCharacteristic = pService->createCharacteristic(
    CHARACTERISTIC_UUID,
    BLECharacteristic::PROPERTY_READ |
    BLECharacteristic::PROPERTY_WRITE |
    BLECharacteristic::PROPERTY_NOTIFY |
    BLECharacteristic::PROPERTY_INDICATE
  );
}
```

[View raw code](#)

For creating a BLE server, the code should follow the next steps:

1. Create a BLE Server. In this case, the ESP32 acts as a BLE server.
2. Create a BLE Service.
3. Create a BLE Characteristic on the Service.
4. Create a BLE Descriptor on the Characteristic.
5. Start the Service.
6. Start advertising, so it can be found by other devices.

## How the code works

Let's take a quick look at how the BLE server example code works.



It starts by importing the necessary libraries for the BLE capabilities.

```
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEServer.h>
```

Then, you need to define a UUID for the Service and Characteristic.

```
#define SERVICE_UUID "4fafc201-1fb5-459e-8fcc-c5c9c331914b"
#define CHARACTERISTIC_UUID "beb5483e-36e1-4688-b7f5-ea07361b26c"
```

You can leave the default UUIDs, or you can go to [uuidgenerator.net](https://uuidgenerator.net) to create random UUIDs for your services and characteristics.

In the `setup()`, it starts the serial communication at a baud rate of 115200.

```
Serial.begin(115200);
```

Then, you create a BLE device called **"MyESP32"**. You can change this name to whatever you like.

```
// Create the BLE Device
BLEDevice::init("MyESP32");
```

In the following line, you set the BLE device as a server.

```
BLEServer *pServer = BLEDevice::createServer();
```

After that, you create a service for the BLE server with the UUID defined earlier.

```
BLEService *pService = pServer->createService(SERVICE_UUID);
```

Then, you set the characteristic for that service. As you can see, you also use the UUID defined earlier, and you need to pass as arguments the characteristic's properties. In

this case, it's: READ and WRITE.

```
BLECharacteristic *pCharacteristic = pService->createCharacteristic(
    CHARACTERISTIC_UUID,
    BLECharacteristic::PROPERTY_READ |
    BLECharacteristic::PROPERTY_WRITE
);
```

After creating the characteristic, you can set its value with the `setValue()` method.

```
pCharacteristic->setValue("Hello World says Neil");
```

In this case we're setting the value to the text "Hello World says Neil". You can change this text to whatever you like. In future projects, this text can be a sensor reading, or the state of a lamp, for example.

Finally, you can start the service, and the advertising, so other BLE devices can scan and find this BLE device.

```
BLEAdvertising *pAdvertising = pServer->getAdvertising();
pAdvertising->start();
```

This is just a simple example on how to create a BLE server. In this code nothing is done in the `loop()`, but you can add what happens when a new client connects (check the `BLE_notify` example for some guidance).

## ESP32 BLE Scanner

Creating an ESP32 BLE scanner is simple. Grab another ESP32 (while the other is running the BLE server sketch). In your Arduino IDE, go to **File > Examples > ESP32 BLE Arduino** and select the **BLE\_scan** example. The following code should load.

```
/*
   Based on Neil Kolban example for IDF: https://github.com/nkolban/ESP32\_BLE\_Arduino
   Ported to Arduino ESP32 by Evandro Copercini
*/

#include <BLEDevice.h>
```

```

// include <BLEUtils.h>
#include <BLEUtils.h>
#include <BLEScan.h>
#include <BLEAdvertisedDevice.h>

int scanTime = 5; //In seconds
BLEScan* pBLEScan;

class MyAdvertisedDeviceCallbacks: public BLEAdvertisedDeviceCal
    void onResult(BLEAdvertisedDevice advertisedDevice) {
        Serial.printf("Advertised Device: %s \n", advertisedDevice
    }
};

void setup() {
    Serial.begin(115200);
    Serial.println("Scanning...");

    BLEDevice::init("");
    pBLEScan = BLEDevice::getScan(); //create new scan
    pBLEScan->setAdvertisedDeviceCallbacks(new MyAdvertisedDeviceCal
    pBLEScan->setActiveScan(true); //active scan uses more power,

```

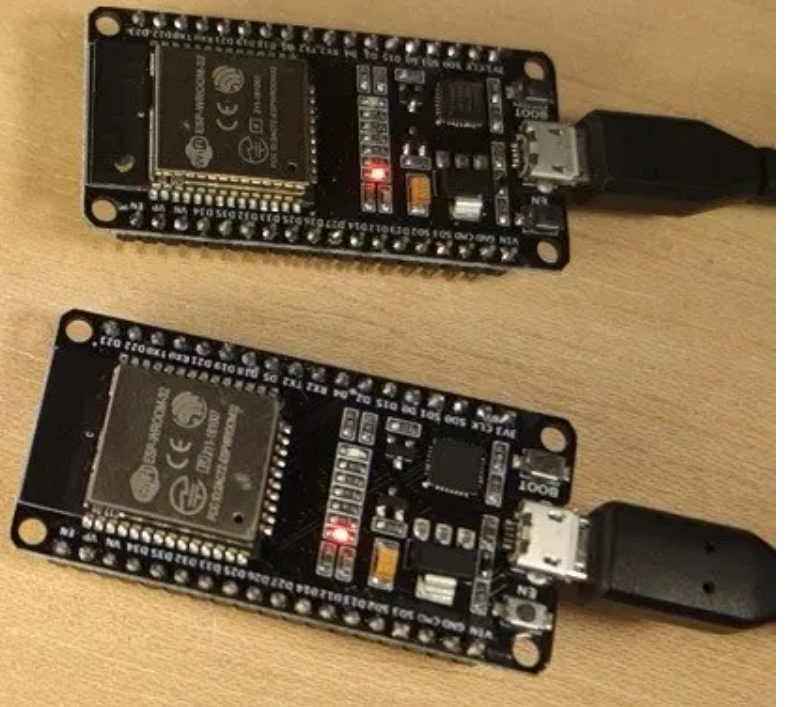
[View raw code](#)

This code initializes the ESP32 as a BLE device and scans for nearby devices. Upload this code to your ESP32. You might want to temporarily disconnect the other ESP32 from your computer, so you're sure that you're uploading the code to the right ESP32 board.

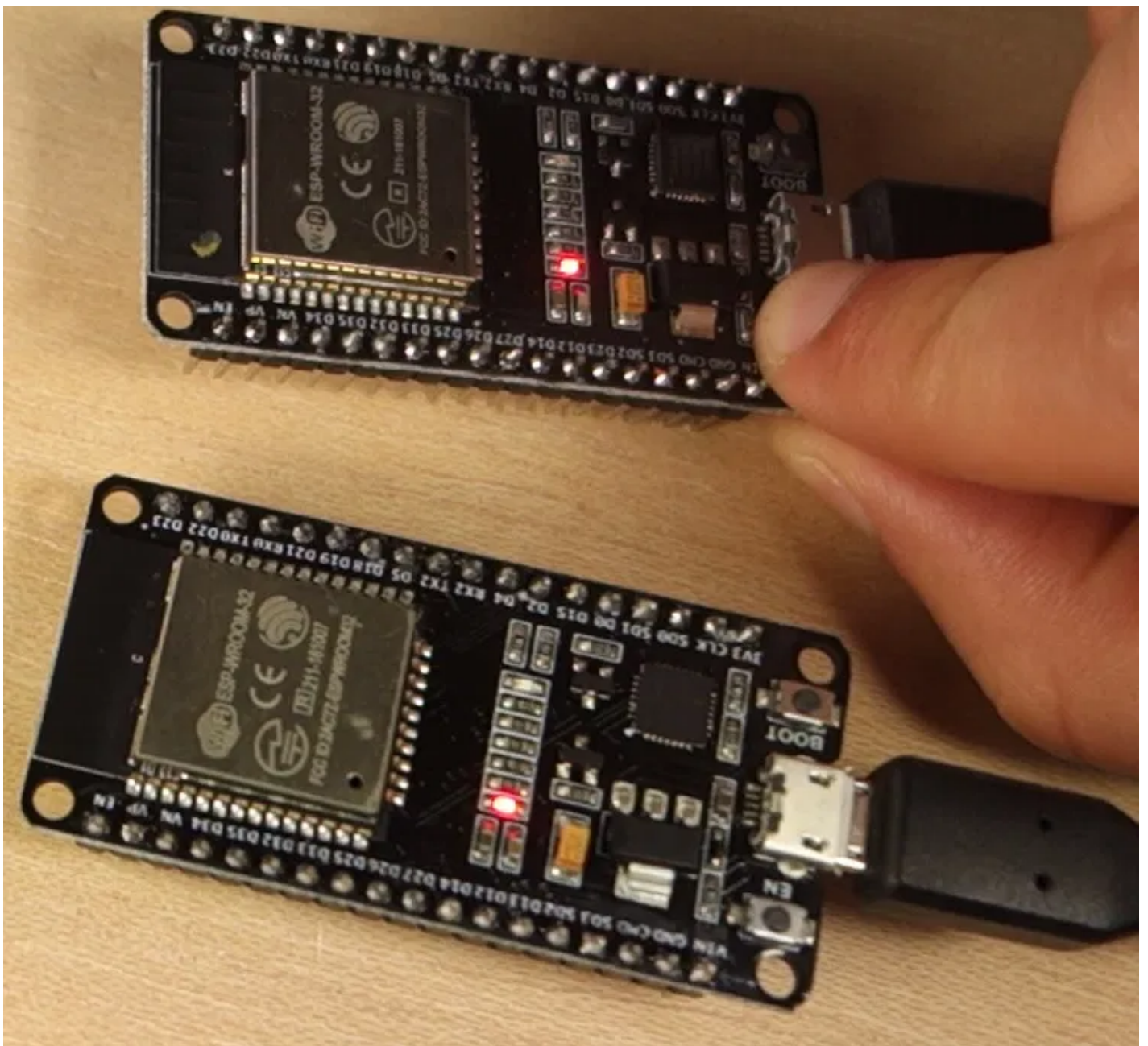
Once the code is uploaded and you should have the two ESP32 boards powered on:

- One ESP32 with the “BLE\_server” sketch;
- Other with ESP32 “BLE\_scan” sketch.

# SCAN SERVER

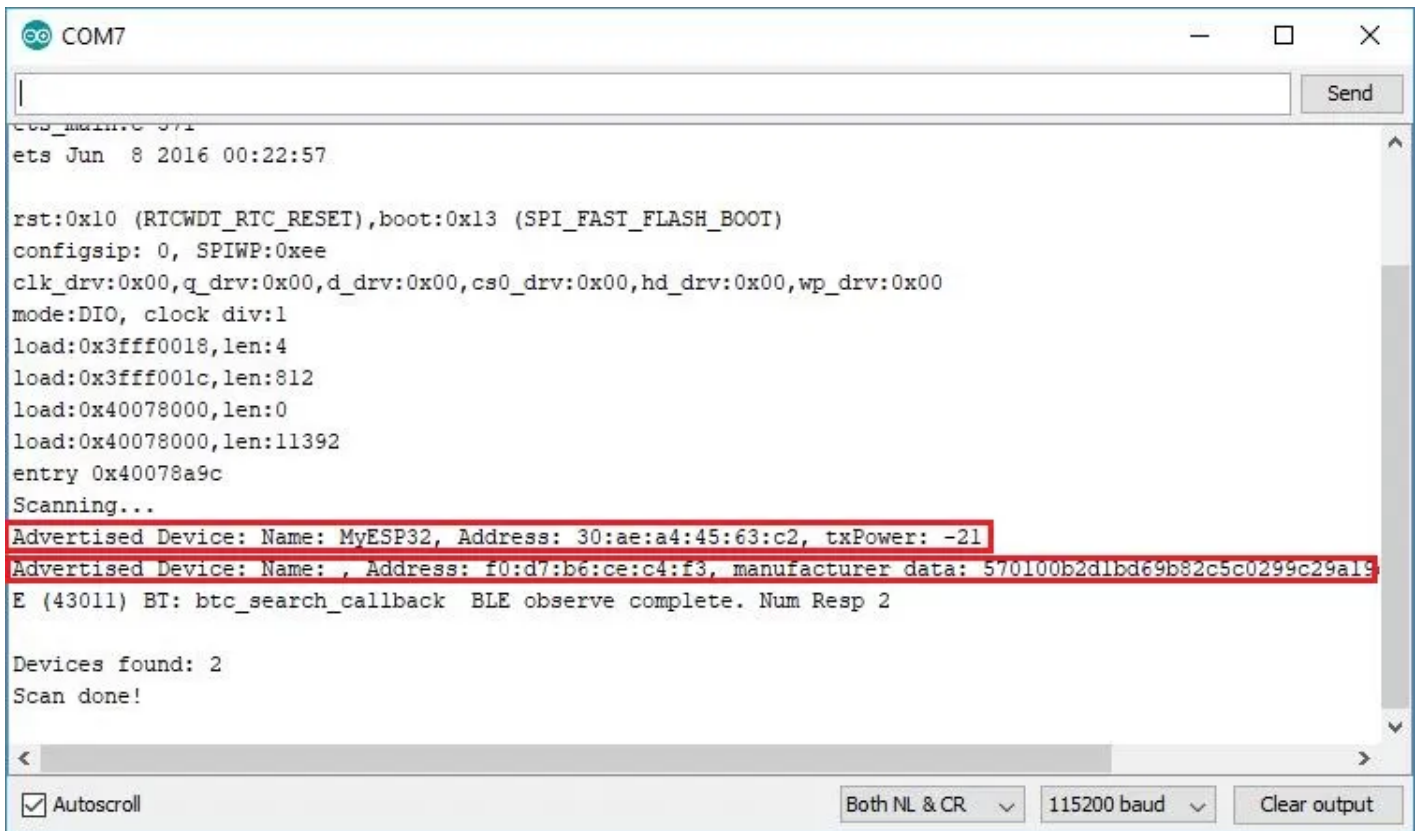


Go to the Serial Monitor with the ESP32 running the “BLE\_scan” example, press the ESP32 (with the “BLE\_scan” sketch) ENABLE button to restart and wait a few seconds while it scans.



The scanner found two devices: one is the ESP32 (it has the name “**MyESP32**”), and the other is our [MiBand2](#).





```
ets Jun  8 2016 00:22:57

rst:0x10 (RTCWDT_RTC_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:812
load:0x40078000,len:0
load:0x40078000,len:11392
entry 0x40078a9c
Scanning...
Advertised Device: Name: MyESP32, Address: 30:ae:a4:45:63:c2, txPower: -21
Advertised Device: Name: , Address: f0:d7:b6:ce:c4:f3, manufacturer data: 570100b2d1bd69b82c5c0299c29a19
E (43011) BT: btc_search_callback BLE observe complete. Num Resp 2

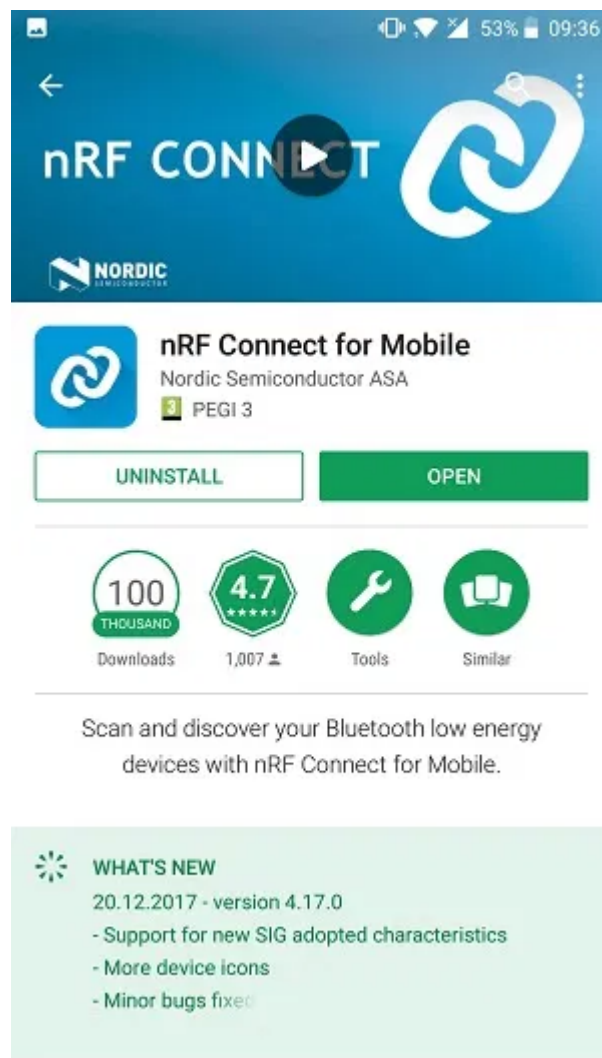
Devices found: 2
Scan done!
```

## Testing the ESP32 BLE Server with Your Smartphone

Most modern smartphones should have BLE capabilities. I'm currently using a [OnePlus 5](#), but most smartphones should also work.

You can scan your ESP32 BLE server with your smartphone and see its services and characteristics. For that, we'll be using a free app called **nRF Connect for Mobile** from Nordic, it works on [Android \(Google Play Store\)](#) and [iOS \(App Store\)](#).

Go to Google Play Store or App Store and search for "nRF Connect for Mobile". Install the app and open it.



Don't forget go to the Bluetooth settings and enable Bluetooth adapter in your smartphone. You may also want to make it visible to other devices to test other sketches later on.



## Devices



Bluetooth adapter is disabled.

ENABLE

SCANNER

BONDED

ADVERTISER

Location disabled.

ENABLE

MORE

No filter



Once everything is ready in your smartphone and the ESP32 is running the BLE server sketch, in the app, tap the scan button to scan for nearby devices. You should find an ESP32 with the name **“MyESP32”**.





# Devices

SCAN



SCANNER

BONDED

ADVERTISER

No filter



**MyESP32**

30:AE:A4:45:63:C2

NOT BONDED

-51 dBm ↔ 39 ms

CONNECT



**MI Band 2**

F0:D7:B6:CE:C4:F3

NOT BONDED

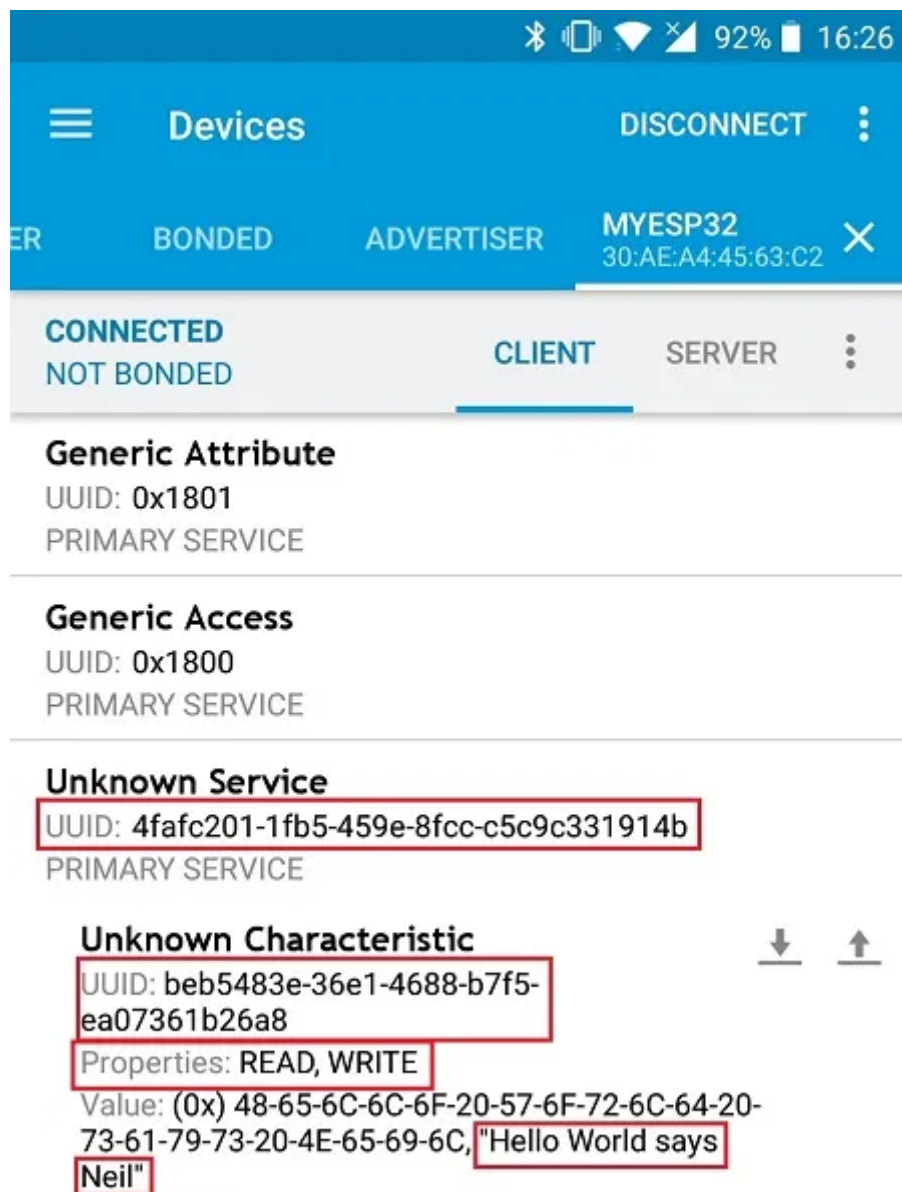
-59 dBm ↔ N/A

CONNECT



Click the “**Connect**” button.

As you can see in the figure below, the ESP32 has a service with the UUID that you’ve defined earlier. If you tap the service, it expands the menu and shows the Characteristic with the UUID that you’ve also defined.



The characteristic has the READ and WRITE properties, and the value is the one you've previously defined in the BLE server sketch. So, everything is working fine.