Table of Contents

1. ESP32 BLE in Arduino. Part2: Setting-up BLE Server 2
1.1. Prerequisites 2
1.2. Suggested Readings and Knowledge Resources 2
1.3. Hands-on Lab Scenario 2
1.3.1. Task to be implemented 2
1.3.2. Start 2
1.3.3. Steps 2
1.3.3.1. Step 1 3
1.3.3.2. Step 2 4
1.3.4. Result validation 5
1.4. FAQ 5
1.5. Project information 6

1. ESP32 BLE in Arduino. Part2: Setting-up BLE Server

This scenario presents how to create the Bluetooth Low Energy server device, which allows us to connect to and read the data. This can be the sensor device which responds to the client with the results of the measurements. This can also be the output device, which we can control writing the data to. This scenario presents the use of the concept of services and characteristics.

1.1. Prerequisites

It is necessary to understand the principles of the Bluetooth Low Energy protocol with concepts of services, characteristics and descriptors. We will use in this scenario the knowledge of the advertising process so making the IoT_7 exercise is recommended.

1.2. Suggested Readings and Knowledge Resources

None at the time.

1.3. Hands-on Lab Scenario

This scenario is intended to be implemented using actual equipment on-site. Although it is possible to make it remotely, with the use of two BLE laboratory nodes, to observe all the details you need to have your mobile phone or similar device with an application which allows you to scan and explore your Bluetooth Low Energy devices and communicate with them. One of the best applications is nRF Connect created by Nordic Semiconductors.

1.3.1. Task to be implemented

Implement a program that operates as the BLE server which advertises itself and allows us to connect to, and read or write the exemplary data.

1.3.2. Start

A similar program is available as the software example for the BLE server device in the ESP32 BLE Arduino package. You can use this example or the beacon program from IoT_7 as the starting point.

1.3.3. Steps

We will present the lab in a few steps. We begin with a simple program which advertises the device with a single service containing a single characteristic. It allows us to establish a connection and, if successfully connected, responds with simple text data. The text can be modified by writing the data back. In further steps, we will improve the program to make it more reliable, eg. by returning to the advertisement procedure after disconnecting.

1.3.3.1. Step 1

Let's begin with a simple program which advertises a single service. The code should start with including Arduino and BLE libraries.

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

We need variables to hold the class pointers for advertising, the server instance, service, and characteristics.

```
BLEAdvertising     *pAdvertising;    //class for the advertising data
BLEServer          *pServer;         //class for the BLE Server device
BLEService         *pService;        //class for the BLE Service in the
Server
BLECharacteristic  *pCharacteristic; //class for the characteristic
```

Any service or characteristic is identified with a unique UUID. If you use the standard service, like eg. Health Thermometer or Binary Sensor Service you can use a 16-bit UUID defined by Bluetooth SIG in the document [1)]. The same document defines the UUIDs for characteristics.



Example of the standard UUIDs for service and characteristic:
Health Thermometer Service - 0x1809
Temperature Measurement Characteristic - 0x2A1C

While none of the standard UUIDs fits your device you need to define your own UUID. It must be 128-bit long and can be randomly generated with the tool available on the website https://www.uuidgenerator.net/[2)]. The service UUID and characteristic UUID must differ. Although they can be completely different in many implementations the UUIDs of characteristics grouped under one service differ on one digit.

```
#define SERVICE_UUID        "6e9b7b26-ca96-4774-b056-8ec5b759fd86"
#define CHARACTERISTIC_UUID "6e9b7b27-ca96-4774-b056-8ec5b759fd86"
```

The setup() function creates and initialises the BLE device instance with the name “SUT BLE device”, and creates the server and the service within the server.

```
BLEDevice::init("SUT BLE device");      //initialise the BLE device
pServer = BLEDevice::createServer();
```

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

Data for reading or writing is organised using characteristics. In this example, we create one characteristic with reading and writing enabled, and the initial data as the “BLE onboard” text.

```
pCharacteristic = pService->createCharacteristic(
                                         CHARACTERISTIC_UUID,
                                         BLECharacteristic::PROPERTY_READ |
                                         BLECharacteristic::PROPERTY_WRITE
                                       );
pCharacteristic->setValue("BLE onboard");
```

In the advertising packet, we can add the service UUID.

```
pAdvertising = BLEDevice::getAdvertising();
pAdvertising->addServiceUUID(SERVICE_UUID);
pAdvertising->setScanResponse(true);
```



Due to the limited size of the advertisement packet, not all service UUIDs can be broadcast. Even more, service UUID does not have to appear in the advertising frame, but its presence makes it possible to scan nearby devices by their functionality, not by names only.

After preparing all the elements we can close the setup() function by starting the service and beginning advertising.

```
pService->start();
pAdvertising->start();
```

The loop function can remain empty or call the delay function periodically.

```
void loop(){
   delay(1000);
};
```

1.3.3.2. Step 2

BLE peripheral device stops advertising after establishing a connection. While the connection is closed or lost it should restart advertising to be able to accept further connections. Doing it is possible with the use of callback functions, executed in selected events occurrence. We need to add the callback functions called on the connection and disconnection events. We will use two global boolean variables, one to store the state of the device, and another to control the single advertising start.

```
bool deviceConnected = false;
bool advStarted = true;  //first advertise starts automatically

class MyServerCallbacks: public BLEServerCallbacks {
```

```
    void onConnect(BLEServer* pServer) {
      deviceConnected = true;
    };

    void onDisconnect(BLEServer* pServer) {
      deviceConnected = false;
    }
};
```

In the setup() function, just after the creation of the server we set up the callbacks.

```
pServer->setCallbacks(new MyServerCallbacks());
```

In the loop() function we have to program the starting of the advertising while the connection is ended. The startAdvertising() function should be called just once so the advStarted boolean variable helps us to control this process.

```
void loop(){
    if (!deviceConnected && !advStarted) {
        pServer->startAdvertising(); // restart advertising
        advStarted = true;
    }

    if (deviceConnected){
        advStarted = false;
    }

    delay(500);
};
```

1.3.4. Result validation

You should be able to find the device in the BLE neighbourhood. It should allow us to establish connection, read the characteristic data and write a new value. Notice that the modified value remains unchanged until the new start of the device.

1.4. FAQ

What types of data can be sent with the characteristics?: The BLE characteristics can send any value as a single byte or a sequence of bytes. BLE does not interpret the data in characteristic in any way. Interpretation lies on the side of the application.
What if I need to read or write more data than a single text?: It is possible to implement many services in one device and add many characteristics to one service. Theoretically, the maximum number of characteristics in one service is 512, but the implementation and the memory available for the BLE library limit it.
What is the maximum length of the data in the characteristics?: The maximum length of the data in the characteristics in the BLE specification is 512 bytes.

<WRAP noprint>

1.5. Project information

This Intellectual Output was implemented under the Erasmus+ KA2.
Project IOT-OPEN.EU Reloaded – Education-based strengthening of the European universities, companies and labour force in the global IoT market.
Project number: 2022-1-PL01-KA220-HED-000085090.

Erasmus+ Disclaimer

This project has been funded with support from the European Commission.
This publication reflects the views of only the author, and the Commission cannot be held responsible for any use that may be made of the information contained therein.

Copyright Notice

This content was created by the IOT-OPEN.EU Reloaded consortium, 2022,2024.
The content is Copyrighted and distributed under CC BY-NC Creative Commons Licence, free for Non-Commercial use.

</WRAP>

1)

2)

https://www.uuidgenerator.net/

From:
https://dokuwiki.zmitac.aei.polsl.pl/ - **RAU6 Dokuwiki**

Permanent link:
https://dokuwiki.zmitac.aei.polsl.pl/en:edu:zmitac:s2:iotn:esp32ble2?rev=1711898887

Last update: **2024/03/31 17:28**