Table of Contents

1. ESP32 BLE in Arduino. Part 4: BLE Client 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 7
1.4. FAQ 8
1.5. Project information 8

1. ESP32 BLE in Arduino. Part 4: BLE Client

This scenario presents the Bluetooth Low Energy client device with a notification/indication mechanism for receiving data. It works in cooperation with the server device created in Part 3 of the ESP32 Arduino BLE laboratory.

1.1. Prerequisites

It is necessary to understand the principles of the Bluetooth Low Energy protocol with concepts of services, characteristics and descriptors. Notification and indication methods of data transmission should be known. We will use in this scenario the knowledge and the server device created in part 3.

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 client which allows us to read the data from the server with a notification/indication mechanism.

1.3.2. Start

A similar program is available as the software example for the BLE client device in the ESP32 BLE Arduino package. You can use this example, or create the program from scratch based on the info from the following steps.

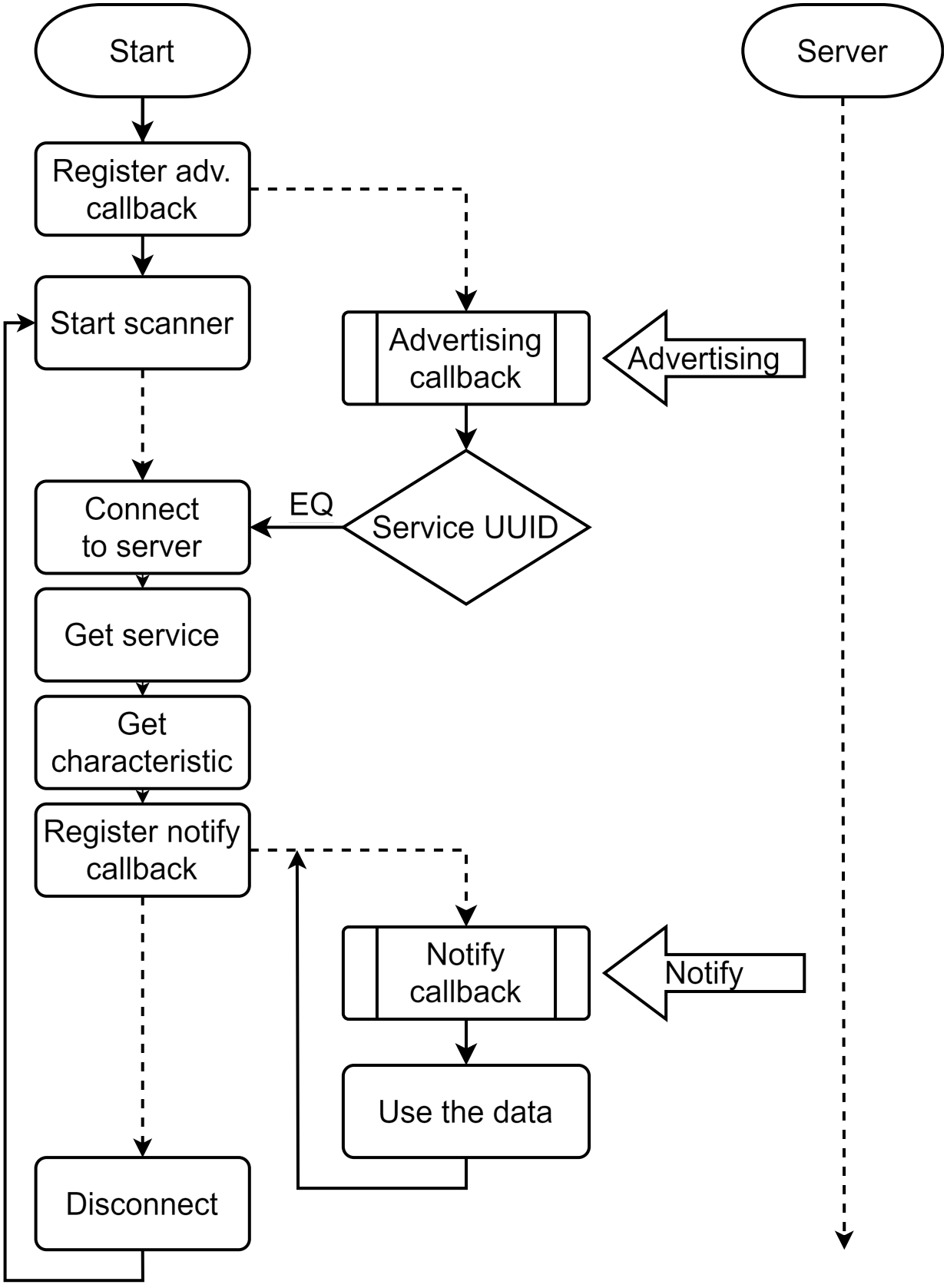
1.3.3. Steps

We will pass through the lab in a few steps. We will create the client device with the definition of remote service and a characteristic. We will subscribe to this characteristic with the notify/indicate

feature. With this feature enabled the peripheral device will periodically transmit the data to our client.

1.3.3.1. Step 1

In this step, we will analyse the behaviour of the client. The client software is much more complex than the server. It is because the central device in many circumstances is a more powerful device than the peripheral. Some parts of the software are implemented as callback functions because they handle reactions on the data coming asynchronously from the server. The diagram presents the algorithm of the client and data coming from the server.



1.3.3.2. Step 2

We start implementation with the libraries needed.

```
#include "Arduino.h"
#include "BLEDevice.h"
```

Next, we define the UUIDs for remote service and characteristic. Notice they must match the ones defined in the server.

```
// The remote service we wish to connect to.
#define SERVICE_UUID "6e9b7b26-ca96-4774-b056-8ec5b759fd86"
// The characteristic of the remote service we are interested in.
#define REMOTE_CHARACTERISTIC_UUID "6e9b7b28-ca96-4774-b056-8ec5b759fd86"
```

Some global variables will be needed for our software.

```
// Variables
static boolean doConnect = false;
static boolean connected = false;
static boolean doScan = false;
static BLERemoteCharacteristic* pRemoteCharacteristic;
static BLEAdvertisedDevice* myDevice;
```

Three callback functions will be defined. The first callback is for advertising.

```
// Scan for BLE servers and find the first one that advertises
// the service we are looking for.
class MyAdvertisedDeviceCallbacks: public BLEAdvertisedDeviceCallbacks {
  void onResult(BLEAdvertisedDevice advertisedDevice) {
    // We have found a device, let's see if it contains the service we are
looking for.
    if (advertisedDevice.haveServiceUUID() &&
advertisedDevice.isAdvertisingService(BLEUUID(SERVICE_UUID))) {

      BLEDevice::getScan()->stop();
      myDevice = new BLEAdvertisedDevice(advertisedDevice);
      doConnect = true;
      doScan = true;

    } // Found our server
  } // onResult
}; // MyAdvertisedDeviceCallbacks
```

Another callback will be executed on the notify event. In the example, we simply print incoming data to the console, but you can use the data in any other way.

```
static void notifyCallback(
  BLERemoteCharacteristic* pBLERemoteCharacteristic,
  uint8_t* pData,
  size_t length,
  bool isNotify) {
    Serial.print("Notify data: ");
    Serial.write(pData, length);
    Serial.println();
}
```

The third callback class helps to inform the other parts of the program about the connection close.

```
class MyClientCallback : public BLEClientCallbacks {
  void onConnect(BLEClient* pclient) {
  }

  void onDisconnect(BLEClient* pclient) {
    connected = false;
  }
};
```

The setup function initialises the Bluetooth, registers the advertising callback function, and starts the scan to look for nearby devices.

```
void setup() {
  Serial.begin(115200);
  Serial.println("BLE Client app");
  // Initialise the Bluetooth
  BLEDevice::init("");

  // Retrieve the pointer to the scan module
  BLEScan* pBLEScan = BLEDevice::getScan();
  // Register callback for incoming advertising
  pBLEScan->setAdvertisedDeviceCallbacks(new MyAdvertisedDeviceCallbacks());
  // Start scan parameters with active scan mode
  pBLEScan->setInterval(1349);
  pBLEScan->setWindow(449);
  pBLEScan->setActiveScan(true);
  // Start the scan for 30 seconds
  pBLEScan->start(30, false);
}
```



In the original example, the scan time is 5 seconds only. It can be too short to connect to a remote device, especially if there are other devices in the neighbourhood.

In the loop function, we wait for the information that the server with the service UUID we were interested in was found. It is signalled with the use of “doConnect” flag. If it's true we can connect to this server. Once we are connected the “connected” flag is set to be true in the connectToServer() function. We will show this function in a while.

```
void loop() {
  if (doConnect == true) {
    connectToServer();
    doConnect = false;
  }

  if (!connected) {
    if(doScan){
    BLEDevice::getScan()->start(0);  // this is just example to start scan
```

```
after disconnect, most likely there is better way to do it in arduino
  }
  delay(1000); // Delay a second between loops.
}
```

The connection to the server is executed in some steps:

1. Creation of the client object.
2. Setting the callback class for handling disconnection.
3. Connection to the remote device.
4. Setting parameters of the connection.
5. Getting the reference to the service.
6. Getting the reference to the characteristic.
7. Registering notification for the characteristic and setting the notify callback function.
8. Informing the main program with the “connected” flag.

```
bool connectToServer() {
    BLEClient*  pClient  = BLEDevice::createClient();
    pClient->setClientCallbacks(new MyClientCallback());

    // Connect to the remote BLE Server.
    pClient->connect(myDevice);
    pClient->setMTU(517);
    // Obtain a reference to the service we are after in the remote BLE
server.
    BLERemoteService* pRemoteService =
pClient->getService(BLEUUID(SERVICE_UUID));
    // Obtain a reference to the characteristic in the service of the remote
BLE server.
    pRemoteCharacteristic =
pRemoteService->getCharacteristic(BLEUUID(REMOTE_CHARACTERISTIC_UUID));
    if(pRemoteCharacteristic->canNotify())
      pRemoteCharacteristic->registerForNotify(notifyCallback);

    connected = true;
    return true;
}
```



Because this example is quite complex, it's good to have messages sent by the program. The original example contains many messages printed at almost every step of the program's operation.

1.3.4. Result validation

You should be able to find the server device in the BLE neighbourhood, connect to it, subscribe to the notification and receive the incoming data. We should be able to observe periodic increments of the value received with data packets from the server.

1.4. FAQ

What is the difference between notification and indication?: As it was mentioned in the beginning the notification is an unacknowledged message while an indication is an acknowledged message. It means that if the indication message remains unacknowledged, the peripheral device won't send another message for this characteristic.

Is the notification/indication mechanism similar to interrupts?: You are right. They work in a similar manner as interrupts in the microprocessor. They allow to sending of the data asynchronously, and the peripheral decides about the time of the message sent like the peripheral decides on the time of signalling interrupt. Data transmission when the central device sends the packet with a characteristic read command is similar to the polling method used in microprocessors.

<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>

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

Permanent link:
https://dokuwiki.zmitac.aei.polsl.pl/en:edu:zmitac:s2:iotn:esp32ble4?rev=1711994559

Last update: **2024/04/01 20:02**