Innovative wireless platforms for the internet of things

Report of Lab2

ESP32-S3 and BLE

Hossein Zahedi Nezhad s309247
Mohammad Eftekhari Pour s307774
Hamid Shabanipour s314041

Professor: Daniele Trinchero

April 2024

Introduction

In this laboratory exercise, we aim to configure the ESP32 S3 to establish a BLE connection with a smartphone using the nRF app, facilitating communication via the GATT protocol. This setup will enable us to manipulate the LED board's color through low-energy Bluetooth communication.

Code description

1. LED Configuration

In the first step we import necessary libraries to use BLE functions on the ESP32, initialing and managing it including:

```
#include <BLEUtils.h>
#include <BLEServer.h>
#include <BLEDevice.h>
#include <BLE2902.h>
#include <Adafruit_NeoPixel.h>
```

Figure1.1. Importing necessary Libraries

BLE Libraries:

BLEDevice.h: The primary library for utilizing BLE functions on the ESP32 is this library. In order to initialize the BLE device, establish BLE servers and clients, and control connections and communications, it offers classes and functions. It is used to configure the ESP32 as a BLE server, able to broadcast features and services.

BLEServer.h: It provides classes to create and manage a BLE server. In fact the ESP32 acts as a BLE server and Clients, like a smartphone app, can connect to this server to read from or write to the defined characteristics.

NeoPixel Library

Adafruit NeoPixel.h: It provides functions to specify the color of each pixel with RGB values, and then calling a function to send the updated color data to the strip.

Then we define number of LED boarded on the ESP32 Board which is just one and The GPIO pin used for sending data to the NeoPixel strip which based on our board is Pin number 48.

```
7   #define LED_COUNT 1  // Number of NeoPixel LEDs
8   #define NEOPIXEL_PIN 48 // Pin connected to NeoPixel data input
9   Adafruit_NeoPixel strip(LED_COUNT, NEOPIXEL_PIN, NEO_GRB + NEO_KHZ800);
```

Figure1.2. LED Configuration

2. UUID definition

Next, we'll define Universally Unique Identifiers (UUIDs) to act as unique tags for our Bluetooth Low Energy (BLE) service and characteristics. These UUIDs allow compatible mobile apps on devices like smartphones and tablets to connect to the server and discover our specific service among potentially many others offered by nearby BLE devices.

We'll be using four different UUIDs:

1. **SERVICE_UUID ("659f11b8-83a6-4dc8-b0c9-c540a19db995")**: This identifies our custom service designed for controlling an RGB LED. By using this UUID, BLE clients like compatible apps can recognize our service even if there are other devices with different services within Bluetooth range.

2. **RED_CHARACTERISTIC_UUID ("29f65805-fee8-457c-abba-bd93b5c176f4")**: This UUID pinpoints a characteristic dedicated solely to controlling the red light in your RGB LED. By sending a value to this characteristic, you can change the brightness of the red light.

3. **GREEN_CHARACTERISTIC_UUID ("7af1d167-7628-4d81-97cd-e01ad8b0f13c")**: This UUID identifies the characteristic that controls the green component of your RGB LED.

4. **BLUE_CHARACTERISTIC_UUID ("2d364af7-5315-4b55-850a-281ec984c076")**: This UUID is linked to the characteristic that controls the blue component of your RGB LED.

```
11  // Service UUID
12  #define SERVICE_UUID "659f11b8-83a6-4dc8-b0c9-c540a19db995"
13  // Characteristic UUIDs for each color channel
14  #define RED_CHARACTERISTIC_UUID   "29f65805-fee8-457c-abba-bd93b5c176f4"
15  #define GREEN_CHARACTERISTIC_UUID "7af1d167-7628-4d81-97cd-e01ad8b0f13c"
16  #define BLUE_CHARACTERISTIC_UUID  "2d364af7-5315-4b55-850a-281ec984c076"
```

Figure1.3. Defining UUIDs

3. Initializing Services

The ESP32 is initialized as "RGB Controller Group3" as a Bluetooth Low Energy (BLE) device by this code, which also establishes a BLE server on the device and configures **MyServerCallbacks** to provide specific connection and disconnection actions. After that, it

generates a BLE service with a distinct UUID (SERVICE_UUID) that collects the properties that clients can interact with (manipulating the color channels of the RGB LED).

```
71    BLEDevice::init("RGB Controller Group3");
72    BLEServer *pServer = BLEDevice::createServer();
73    pServer->setCallbacks(new MyServerCallbacks());
74
75    BLEService *pService = pServer->createService(SERVICE_UUID);
```

Figure1.4. Defining device name and Initializing services

4. Setting Up Color Control Characteristics for RGB LED

This part of the code defines and creates individual BLE characteristics for each color channel (red, green, and blue) within the previously defined BLE service. Each characteristic is identified by a unique UUID and is configured with the property **PROPERTY_WRITE**, allowing BLE clients (like smartphones) to write values to these characteristics. The purpose of these characteristics is to control the intensity of the red, green, and blue components of an RGB LED, enabling the customization of the LED's color through BLE communications.

```
78    pRedCharacteristic = pService->createCharacteristic(
79                           RED_CHARACTERISTIC_UUID,
80                           BLECharacteristic::PROPERTY_WRITE);
81    pGreenCharacteristic = pService->createCharacteristic(
82                           GREEN_CHARACTERISTIC_UUID,
83                           BLECharacteristic::PROPERTY_WRITE);
84    pBlueCharacteristic = pService->createCharacteristic(
85                           BLUE_CHARACTERISTIC_UUID,
86                           BLECharacteristic::PROPERTY_WRITE);
```

Figure1.5. Defining Characteristics for each UUID

5. Assigning Callbacks for RGB Color Characteristics

This piece of code uses the same set of instructions (called "ColorCallbacks") for all three characteristics (red, green, and blue). This lets the code combine the color values sent by a BLE client app. By adjusting the red, green, and blue values together.

```
88    ColorCallbacks* colorCallbacks = new ColorCallbacks();
89    pRedCharacteristic->setCallbacks(colorCallbacks);
90    pGreenCharacteristic->setCallbacks(colorCallbacks);
91    pBlueCharacteristic->setCallbacks(colorCallbacks);
```

Figure1.6. Defining Characteristics for each UUID

6. Initializing and Setting LED to White

Function **strip.Color (255, 255, 255)** creates a color value representing white by setting the red, green, and blue components to their maximum intensities (255). The **fill** method applies this color to all LEDs on the strip.

```
67      strip.begin();
68      strip.fill(strip.Color(255, 255, 255)); // Set all pixels to white
69      strip.show(); // Initialize all pixels to 'off'
70
```

Figure1.7. Setting white color

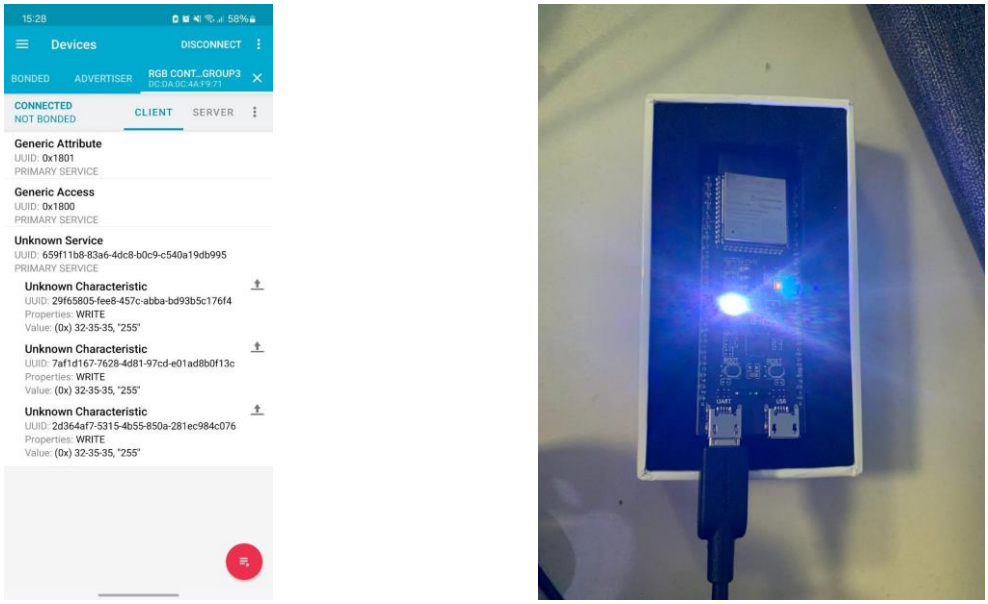

Figure1.8. Showing white color through LED by entering 255 as input of all three color characteristics

7. Combination of Colors

The **ColorCallbacks** class, specifically its **onWrite** method, is responsible for processing the new colors. This method reads the incoming data written to any of the RGB characteristics (red, green, or blue) and adjusts the intensity of the LED's color components accordingly. It directly supports the creation of new colors in two ways. In fact, The first byte of the incoming data (**value[0]**) represents the intensity of the primary color of LED which can be Green, Blue and Red. If a second byte is present in the incoming data (**value[1]**), The pure colors are changed to more complex color creation by mixing different intensities of two colors based on this mechanism:

- The second byte adds green to the red characteristic. Thus, an LED will have a reddish-green color combination.

- For the green characteristic, the second byte adds blue. Then the color of LED will be combination of Blue and Green.

- If the Red is added to the blue characteristic by entering the second byte. As a result, the LED will be a mix of blue and red.

```
class ColorCallbacks : public BLECharacteristicCallbacks {
    void onWrite(BLECharacteristic *pCharacteristic) override {
        std::string value = pCharacteristic->getValue();

        if (value.length() >= 1) {

            uint8_t primaryColorIntensity = static_cast<uint8_t>(value[0]);


            uint8_t red = 0, green = 0, blue = 0;


            if (pCharacteristic == pRedCharacteristic) {
                red = primaryColorIntensity;

                if (value.length() > 1) green = static_cast<uint8_t>(value[1]);
            } else if (pCharacteristic == pGreenCharacteristic) {
                green = primaryColorIntensity;

                if (value.length() > 1) blue = static_cast<uint8_t>(value[1]);
            } else if (pCharacteristic == pBlueCharacteristic) {
                blue = primaryColorIntensity;

                if (value.length() > 1) red = static_cast<uint8_t>(value[1]);
            }

            // Set the pixel color and show.
            strip.setPixelColor(0, strip.Color(red, green, blue));
            strip.show();
        }
```

Figure1.9. Combining colors and create new color and new value

Results

Once the code is executed, the designated name for the server (RGB Controller Group3), corresponding to the ESP32 board linked to the computer, appears in the list of advertised devices, allowing for connections to be established.

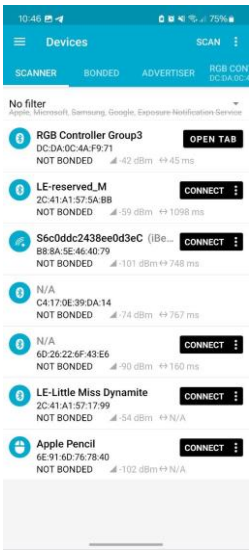

Figure2.1. List of Observable devices

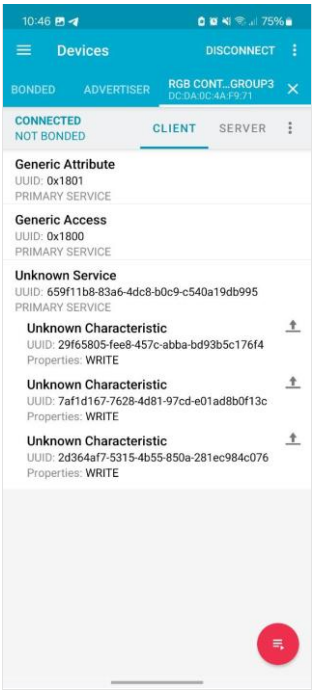

Figure2.2. List of defined UUIDs and characteristics

As it shown in figures below, after connecting the mobile phone (nRF application) to the ESP32 board through Bluetooth low energy:

When you input the `FF` command as a byte array into the characteristic associated with any color, the LED board will display that color at its brightest. It's important to recognize that `FF` in

hexadecimal corresponds to 255 in decimal. Thus, this allows for the display of any color at its maximum intensity.

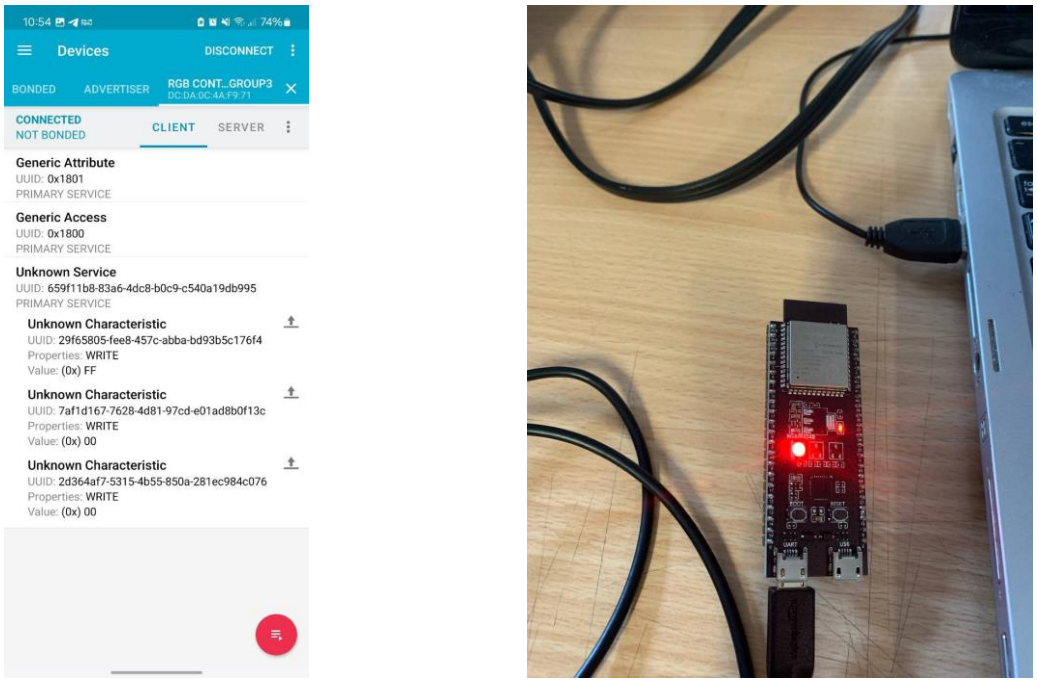

Figure2.3. Turn the LED red by entering **FF** as the input for its characteristic

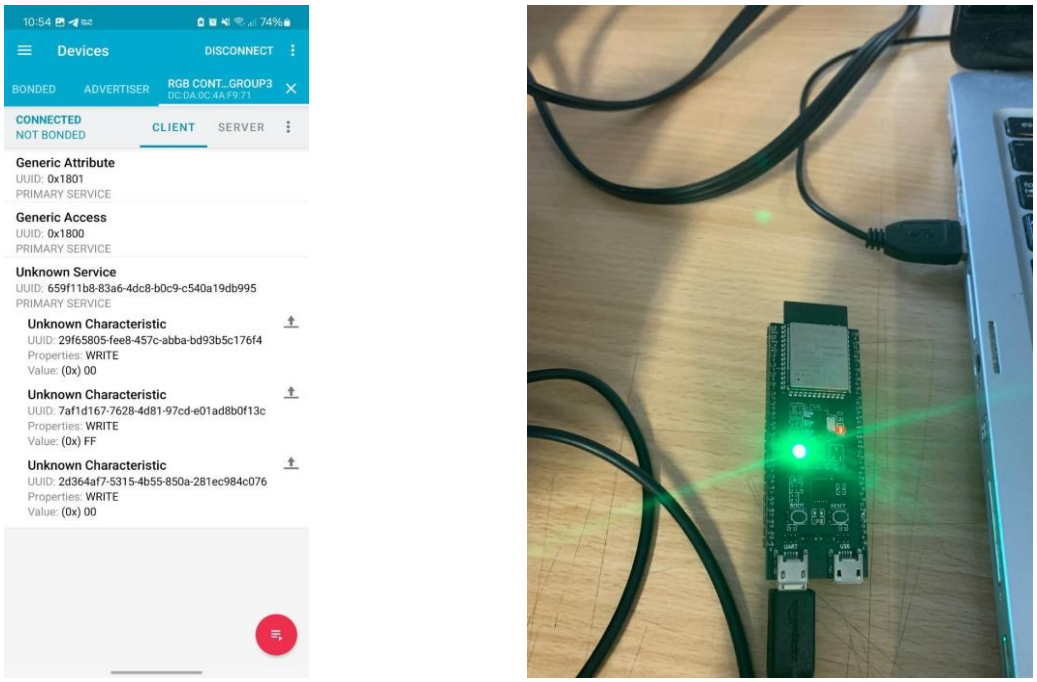

Figure2.4. Turn the LED green by entering **FF** as the input for its characteristic

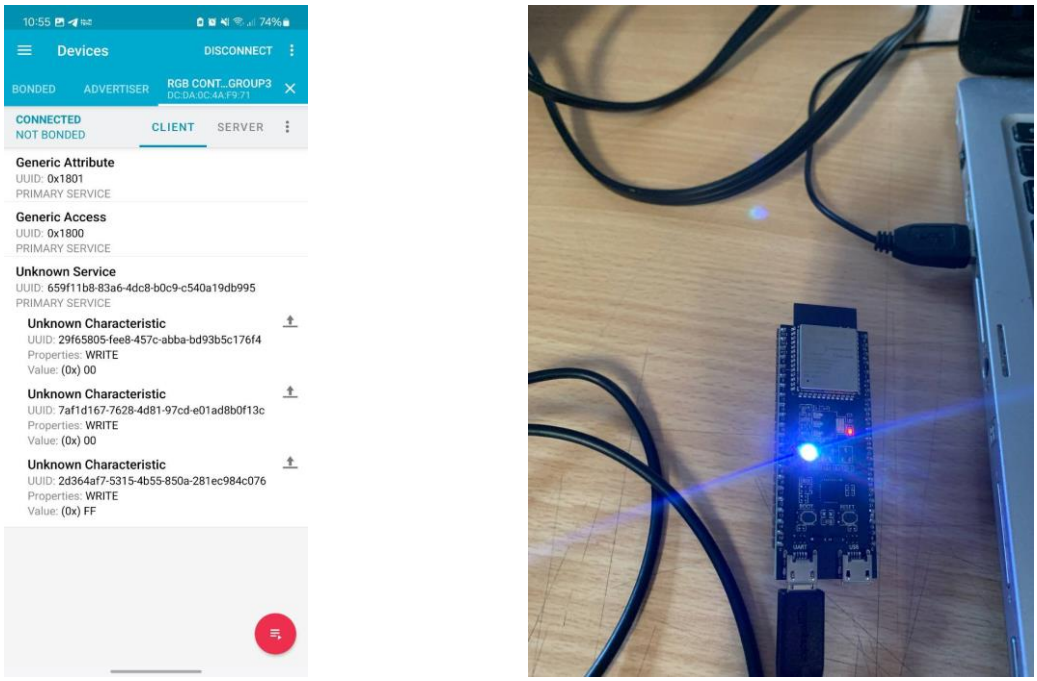

Figure2.5. Turn the LED blue by entering **FF** as the input for its characteristic

Combined Colors

As highlighted in section 1222, to display a new color, we mix the three primary colors (Red, Green, Blue). Indeed, the inclusion of a second byte in the input for each color's characteristic enables the blending of colors, resulting in the production of a new value and color. The mechanism of combining colors is two-by-two and each color is combined with a corresponding color as follows:

- For the red characteristic, the second byte adds green.
- For the green characteristic, the second byte adds blue.
- For the blue characteristic, the second byte adds red.

As shown in the image below, by giving the FF36 AS input to the UUID Characteristics of red color, the LED color will be a combination of red and green colors. Therefore, FF36 is a new value that causes the LED to turn into a specific color (combination of green and red).

It is worth mentioning that the value **FF36** represents two bytes in hexadecimal notation. In hexadecimal, each digit represents 4 bits, so two digits represent a byte (8 bits). Therefore, **FF** is first byte, and **63** is the second byte.

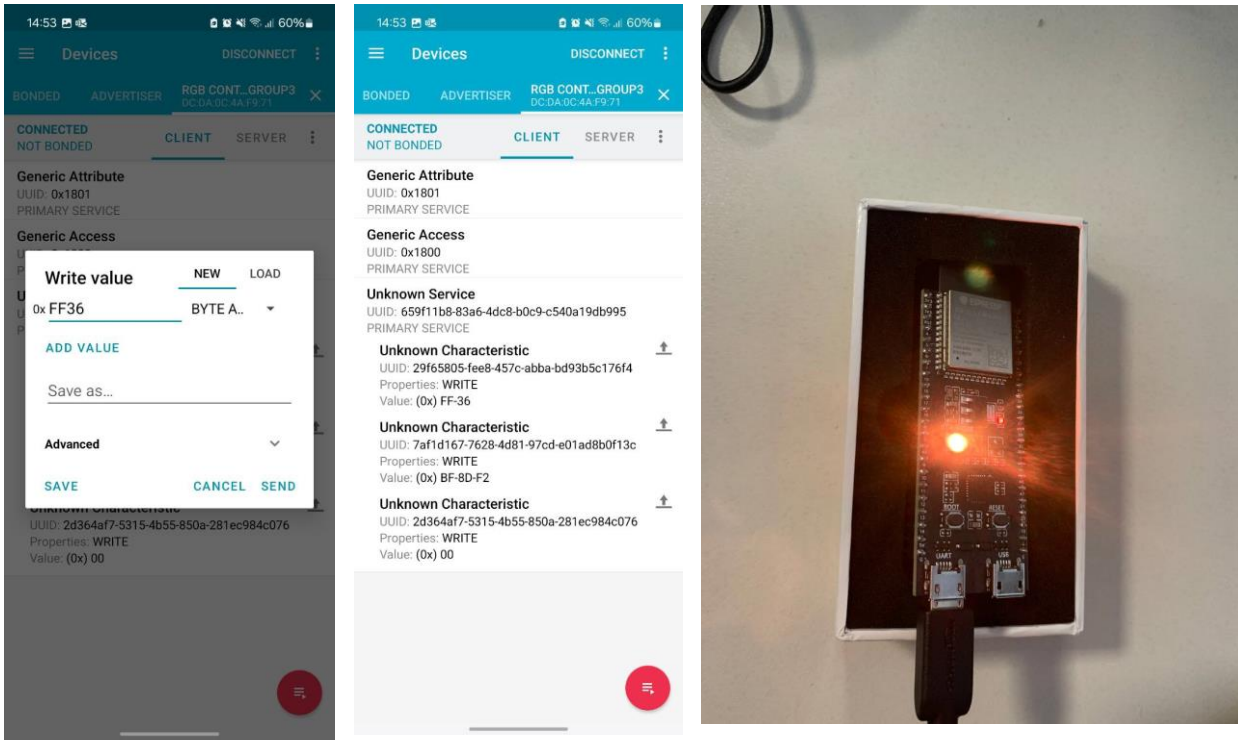

Figure2.6. Combination of Red and Green by entering **FF36** as the input of its characteristic

In the same way, inputting **FF12** into the characteristic UUID for the green color results in the LED displaying a mix of green and blue colors.

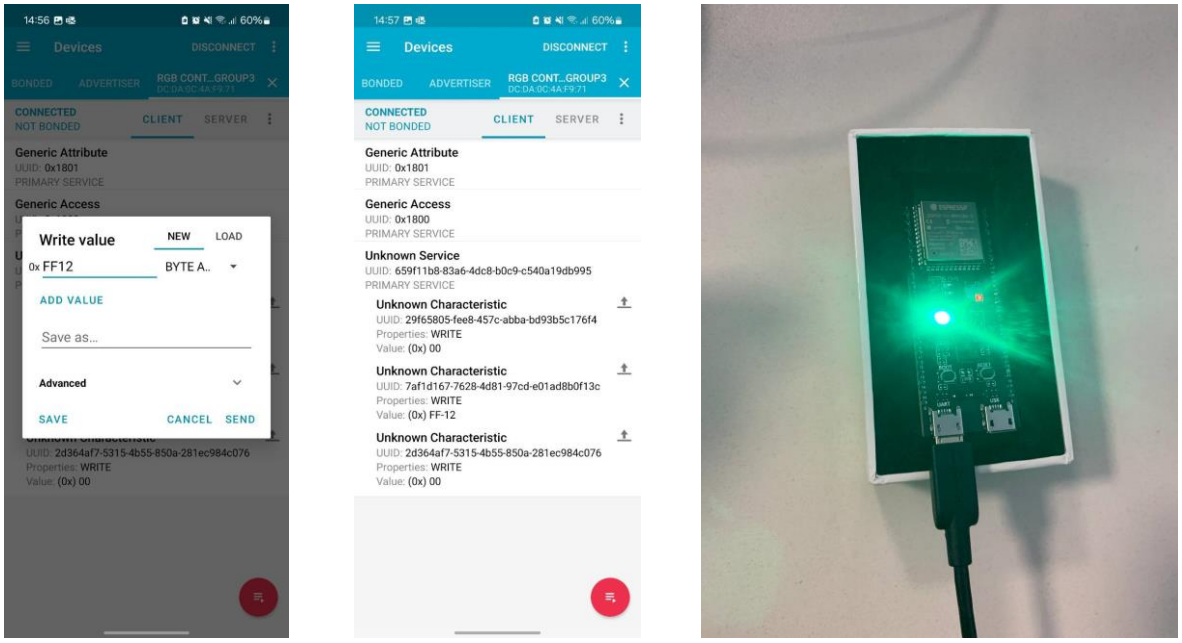

Figure2.7. Combination of Green and Blue by entering **FF12** as the input of its characteristic

And Finally, a combination of blue and red can be seen as the color of the LED in the image, which is the result of giving the FF98 as input to the UUID Characteristics of the blue color.

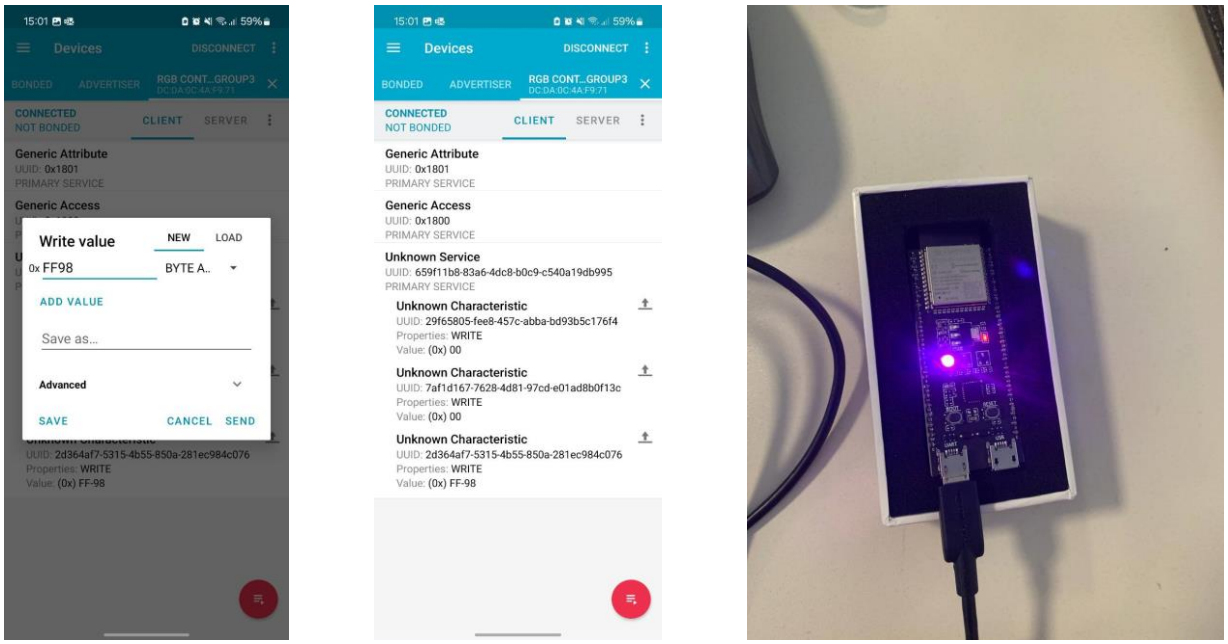

Figure2.8. Combination of Blue and Red by entering **FF98** as the input of its characteristic