Mastering Bluetooth Low Energy (BLE) 5.0 on the ESP32-C6

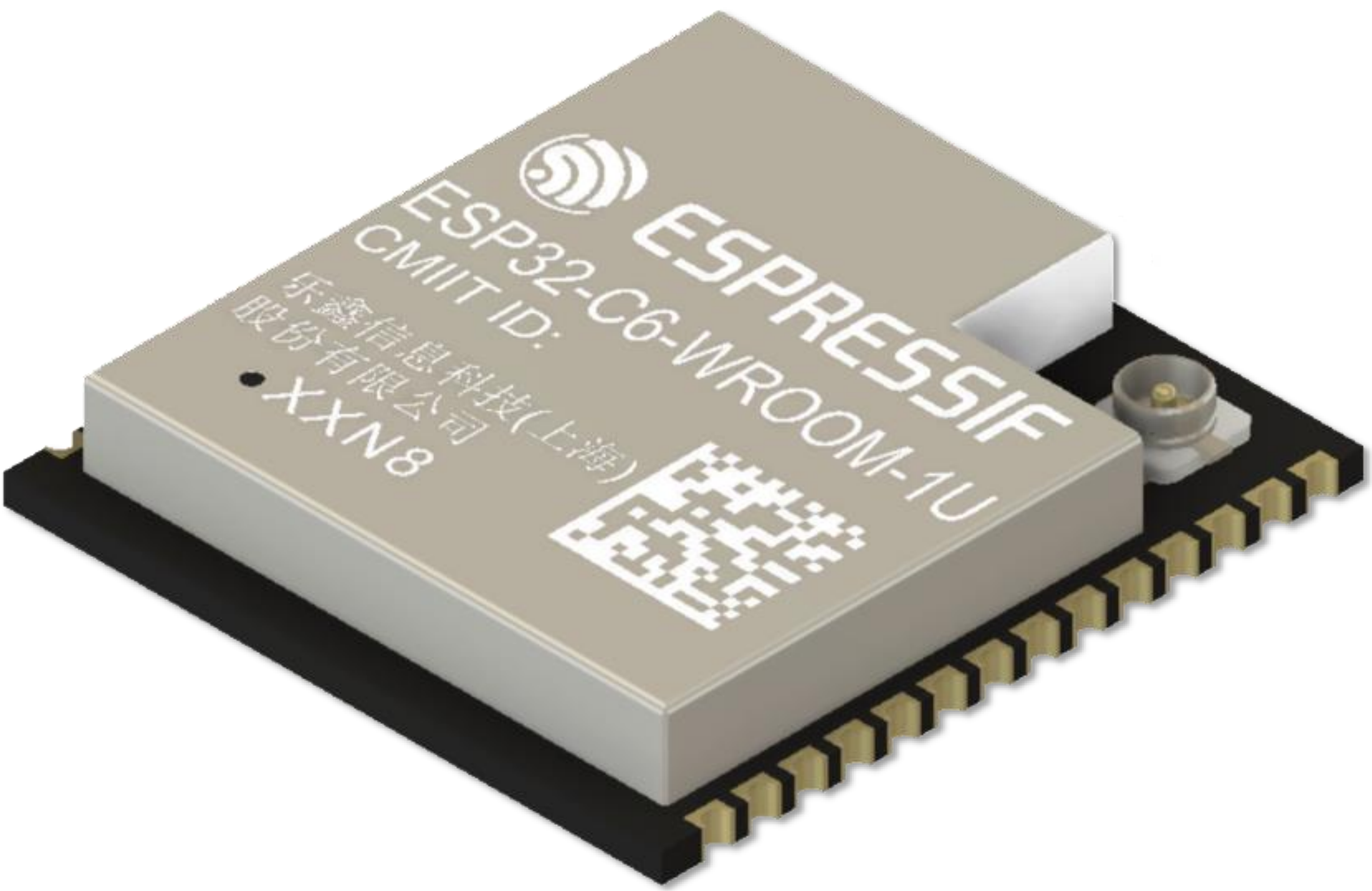


Table of Contents

1. Introduction

Bluetooth Low Energy (BLE) 5.0 has become a key enabler in the IoT space due to its low power consumption, increased range, and support for modern data transmission strategies. The **ESP32-C6**, part of Espressif's latest generation of SoCs, offers integrated BLE 5.0 support alongside Wi-Fi 6 and robust FreeRTOS compatibility.

1. Introduction

This article targets embedded engineers looking to master BLE 5.0 on the ESP32-C6 using the ESP-IDF framework. We'll explore the fundamentals, demonstrate a real-world application, and provide fully documented and commented code to get you confidently building BLE-enabled devices.

2. Overview of BLE 5.0 on the ESP32-C6

What Is Bluetooth Low Energy (BLE)?

Bluetooth Low Energy is a wireless communication protocol designed for **short-range**, **low-power**, and **low-bandwidth** data exchange. Unlike classic Bluetooth (used for streaming audio), BLE is optimized for **intermittent communication between devices**.

It's perfect for **IoT devices**, such as:

- Smart home sensors,
- Wearables,
- Health monitors,
- BLE tags and beacons.

2. Overview of BLE 5.0 on the ESP32-C6

Overview

BLE 5.0 offers several enhancements over its predecessors:

- *2x the data rate (2 Mbps PHY)*
- *4x the range (coded PHY)*
- *Increased advertising packet size (255 bytes)*
- *Improved channel selection and coexistence*

The ESP32-C6 leverages these enhancements through its integrated BLE 5.0 radio, supported natively by ESP-IDF's Bluetooth stack (based on NimBLE). It is ideal for low-power applications like sensor nodes, wearables, and home automation peripherals.

2. Overview of BLE 5.0 on the ESP32-C6

BLE Connection Lifecycle

1. Advertising

The ESP32 sends small packets periodically announcing its name or services.

[ESP32 BLE Peripheral] --> "ESP32-BLE Available!"

2. Scanning & Connection

A BLE client (like a phone app) scans for devices and connects to one.

[Phone App] --> "Connect to ESP32-BLE"

2. Overview of BLE 5.0 on the ESP32-C6

3. Service Discovery

The client asks: “What kind of data/services do you offer?”The ESP32 replies with a GATT table.

4. GATT Profile

GATT (Generic Attribute Profile) defines how data is structured:

- **Service**: A logical group (e.g., Battery Service)
- **Characteristic**: A piece of data (e.g., Battery Level)

```
Service: Battery
  └──── Characteristic: Battery Level = 90%
```

3. Setting Up the Development Environment

To get started with BLE on ESP32-C6, you need:

- **ESP-IDF v5.1 or later**
- **An ESP32-C6 development board (e.g., ESP32-C6-DevKitC-1)**
- **A USB-to-UART cable**
- **A mobile BLE scanner app (e.g., nRF Connect)**

Install the ESP-IDF by following Espressif's official documentation: 👉 https://docs.espressif.com/projects/esp-idf/en/latest/esp32c6/

4. Understanding GATT Profiles and Services

BLE communication revolves around the **GATT (Generic Attribute) Profile**, which defines how data is structured and exchanged over a BLE connection. A GATT Server exposes **services,** each containing one or more **characteristics**.

In our case, we'll create a **GATT Ser**ver that exposes the **Environmental Sensing Service (UUID: 0x181A)**, which includes a **Temperature Characteristic (UUID: 0x2A6E)**.

4. Understanding GATT Profiles and Services

GATT Services and Characteristics Definition

Example:

- A Primary Service (0x181A) is defined for environmental sensing.
- A Temperature Characteristic (0x2A6E) is included.
- It is both readable and notifiable.

```c
static const struct ble_gatt_svc_def gatt_svcs[] = {
    {
        .type = BLE_GATT_SVC_TYPE_PRIMARY,
        .uuid = BLE_UUID16_DECLARE(0x181A), // Environmental Sensing
        .characteristics = (struct ble_gatt_chr_def[]) {
            {
                .uuid = BLE_UUID16_DECLARE(0x2A6E), // Temperature
                .access_cb = temp_chr_access_cb,
                .val_handle = &temp_char_handle,
                .flags = BLE_GATT_CHR_F_READ | BLE_GATT_CHR_F_NOTIFY,
            },
            { 0 },
        }
```

4. Understanding GATT Profiles and Services

environmental sensing.

- A Temperature Characteristic (0x2A6E) is included.
- It is both readable and notifiable.

```c
static const struct ble_gatt_svc_def gatt_svcs[] = {
    {
        .type = BLE_GATT_SVC_TYPE_PRIMARY,
        .uuid = BLE_UUID16_DECLARE(0x181A), // Environmental Sensing
        .characteristics = (struct ble_gatt_chr_def[]) {
            {
                .uuid = BLE_UUID16_DECLARE(0x2A6E), // Temperature
                .access_cb = temp_chr_access_cb,
                .val_handle = &temp_char_handle,
                .flags = BLE_GATT_CHR_F_READ | BLE_GATT_CHR_F_NOTIFY,
            },
            { 0 },
        },
    },
    { 0 },
};
```

5. Real-World Application: Minimal BLE GATT Server Using NimBLE

```c
#include <stdio.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "nvs_flash.h"

#include "nimble/nimble_port.h"
#include "nimble/nimble_port_freertos.h"
#include "host/ble_hs.h"
#include "host/ble_gap.h"
#include "host/ble_gatt.h"
#include "services/gap/ble_svc_gap.h"
#include "services/gatt/ble_svc_gatt.h"

#define TAG "BLE_GATT"

#define GATTS_SERVICE_UUID_ENVIRONMENTAL   0x181A
#define GATTS_CHAR_UUID_TEMPERATURE        0x2A6E

static uint8_t ble_addr_type;

/**
 * @brief GATT characteristic read callback for temperature value.
 *
 * This callback is triggered when a client requests to read the
 * temperature characteristic.
 * It encodes a static temperature value (25.50°C) into a 16-bit format
 * in 0.01°C resolution.
 *
 * @param conn_handle The connection handle of the client making the request.
 * @param attr_handle The handle of the attribute being accessed.
 * @param ctxt Pointer to the access context, used to write the response.
```

5. Real-World Application: Minimal BLE GATT Server Using NimBLE

```c
/**
 * @brief GATT characteristic read callback for temperature value.
 *
 * This callback is triggered when a client requests to read the
 * temperature characteristic.
 * It encodes a static temperature value (25.50°C) into a 16-bit format
 * in 0.01°C resolution.
 *
 * @param conn_handle The connection handle of the client making the request.
 * @param attr_handle The handle of the attribute being accessed.
 * @param ctxt Pointer to the access context, used to write the response.
 * @return 0 on success, error code otherwise.
 */
static int temp_read_cb(uint16_t conn_handle,
                        uint16_t attr_handle,
                        struct ble_gatt_access_ctxt *ctxt,
                        void *arg)
                        {
    int16_t temp_c = (int16_t)(25.50 * 100); // Encode 25.50°C
    uint8_t temp_encoded[2] = { temp_c & 0xFF, (temp_c >> 8) & 0xFF };

    os_mbuf_append(ctxt->om, temp_encoded, sizeof(temp_encoded));
    ESP_LOGI(TAG, "Temperature read: %.2f", temp_c / 100.0f);
    return 0;
}

/**
 * @brief BLE GATT server service definitions.
 *
 * This service exposes the Environmental Sensing service (UUID: 0x181A)
 * with a read-only temperature characteristic (UUID: 0x2A6E).
 */
static const struct ble_gatt_svc_def gatt_svcs[] = {
    {
        .type = BLE_GATT_SVC_TYPE_PRIMARY,
        .uuid = BLE_UUID16_DECLARE(GATTS_SERVICE_UUID_ENVIRONMENTAL),
```

5. Real-World Application: Minimal BLE GATT Server Using NimBLE

```c
/**
 * @brief BLE GATT server service definitions.
 *
 * This service exposes the Environmental Sensing service (UUID: 0x181A)
 * with a read-only temperature characteristic (UUID: 0x2A6E).
 */
static const struct ble_gatt_svc_def gatt_svcs[] = {
    {
        .type = BLE_GATT_SVC_TYPE_PRIMARY,
        .uuid = BLE_UUID16_DECLARE(GATTS_SERVICE_UUID_ENVIRONMENTAL),
        .characteristics = (struct ble_gatt_chr_def[]) {
            {
                .uuid = BLE_UUID16_DECLARE(GATTS_CHAR_UUID_TEMPERATURE),
                .access_cb = temp_read_cb,
                .flags = BLE_GATT_CHR_F_READ,
            },
            { 0 } // End of characteristics
        }
    },
    { 0 } // End of services
};

/**
 * @brief BLE GAP event handler.
 *
 * Handles connection and disconnection events. Restarts
 * advertising when a client disconnects.
 *
 * @param event Pointer to the BLE GAP event structure.
 * @param arg Optional argument passed to the handler (unused).
 *
 * @return 0 on success, error code otherwise.
 */
static int gap_event_handler(struct ble_gap_event *event, void *arg) {
    switch (event->type) {
        case BLE_GAP_EVENT_CONNECT:
```

5. Real-World Application: Minimal BLE GATT Server Using NimBLE

```c
/**
 * @brief BLE GAP event handler.
 *
 * Handles connection and disconnection events. Restarts
 * advertising when a client disconnects.
 *
 * @param event Pointer to the BLE GAP event structure.
 * @param arg Optional argument passed to the handler (unused).
 *
 * @return 0 on success, error code otherwise.
 */
static int gap_event_handler(struct ble_gap_event *event, void *arg) {
    switch (event->type) {
        case BLE_GAP_EVENT_CONNECT:
            ESP_LOGI(TAG, "Client connected");
            break;

        case BLE_GAP_EVENT_DISCONNECT:
            ESP_LOGI(TAG, "Client disconnected. Restarting advertising...");
            ble_gap_adv_start(ble_addr_type, NULL, BLE_HS_FOREVER,
                              &(struct ble_gap_adv_params){
                                  .conn_mode = BLE_GAP_CONN_MODE_UND,
                                  .disc_mode = BLE_GAP_DISC_MODE_GEN
                              },
                              gap_event_handler, NULL);
            break;

        default:
            break;
    }
    return 0;
}

/**
 * @brief Configure and start BLE advertising.
 *
```

5. Real-World Application: Minimal BLE GATT Server Using NimBLE

```c
/**
 * @brief Configure and start BLE advertising.
 *
 * Sets the advertising fields (name, flags) and starts
 * advertising with general discovery mode.
 */
static void ble_advertise(void) {
    struct ble_hs_adv_fields fields = {0};

    fields.flags = BLE_HS_ADV_F_DISC_GEN | BLE_HS_ADV_F_BREDR_UNSUP;
    fields.name = (uint8_t *)"ESP32C6_BLE";
    fields.name_len = strlen((char *)fields.name);
    fields.name_is_complete = 1;

    ble_gap_adv_set_fields(&fields);

    ble_gap_adv_start(ble_addr_type, NULL, BLE_HS_FOREVER,
                      &(struct ble_gap_adv_params){
                          .conn_mode = BLE_GAP_CONN_MODE_UND,
                          .disc_mode = BLE_GAP_DISC_MODE_GEN
                      },
                      gap_event_handler, NULL);
}

/**
 * @brief BLE synchronization callback.
 *
 * Called when the BLE host and controller are in sync.
 * Infers BLE address and starts advertising.
 */
static void ble_on_sync(void) {
    ble_hs_id_infer_auto(0, &ble_addr_type);
    ble_advertise();
}

/**
```

5. Real-World Application: Minimal BLE GATT Server Using NimBLE

```c
/**
 * @brief BLE synchronization callback.
 *
 * Called when the BLE host and controller are in sync.
 * Infers BLE address and starts advertising.
 */
static void ble_on_sync(void) {
    ble_hs_id_infer_auto(0, &ble_addr_type);
    ble_advertise();
}

/**
 * @brief NimBLE host task entry point.
 *
 * This task runs the NimBLE host stack on a FreeRTOS task.
 *
 * @param param Pointer to user data (unused).
 */
static void ble_host_task(void *param) {
    nimble_port_run();
    nimble_port_freertos_deinit();
}

/**
 * @brief Application entry point.
 *
 * Initializes NVS, configures the BLE stack, registers services,
 * and starts the BLE host task.
 */
void app_main(void) {
    ESP_ERROR_CHECK(nvs_flash_init());

    // Initialize NimBLE stack (no controller init needed on ESP32-C6)
    nimble_port_init();

    // Register callback for BLE sync
```

5. Real-World Application: Minimal BLE GATT Server Using NimBLE

```c
/**
 * @brief Application entry point.
 *
 * Initializes NVS, configures the BLE stack, registers services,
 * and starts the BLE host task.
 */
void app_main(void) {
    ESP_ERROR_CHECK(nvs_flash_init());

    // Initialize NimBLE stack (no controller init needed on ESP32-C6)
    nimble_port_init();

    // Register callback for BLE sync
    ble_hs_cfg.sync_cb = ble_on_sync;

    // Initialize GAP and GATT services
    ble_svc_gap_init();
    ble_svc_gatt_init();

    // Register our custom GATT services
    ESP_ERROR_CHECK(ble_gatts_count_cfg(gatt_svcs));
    ESP_ERROR_CHECK(ble_gatts_add_svcs(gatt_svcs));

    // Start the BLE host task
    nimble_port_freertos_init(ble_host_task);
}
```

5. Real-World Application: Minimal BLE GATT Server Using NimBLE

What This Example Does?

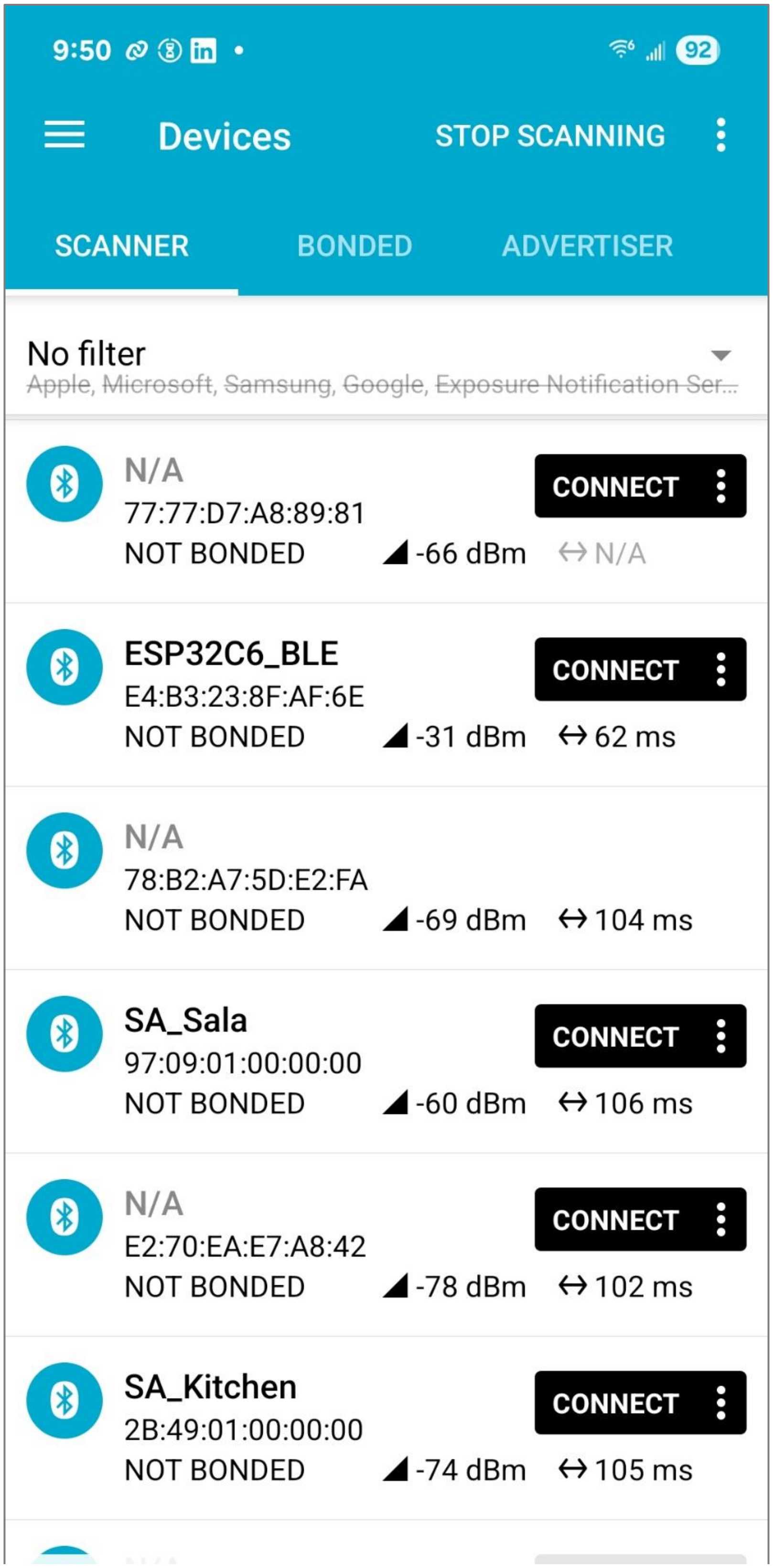
- Creates a BLE GATT server on ESP32-C6
- Advertises itself as "ESP32C6_BLE“
- Exposes a Temperature characteristic (UUID 0x2A6E) in the Environmental Sensing service
- Returns a static value 25.50°C encoded in 0.01°C format
- Uses NimBLE, fully compatible with ESP32-C6

6. How to Test

1. **Flash the code** to your ESP32-C6.
2. **Install** a BLE scanning app like **nRF Connect** or **LightBlue** on your smartphone.
3. **Scan** for BLE devices – you should see the advertised GATT server (**"ESP32C6_BLE"**).
4. **Connect** and navigate to to the Temperature Characteristic.
5. **Watch** temperature values update in real-time.

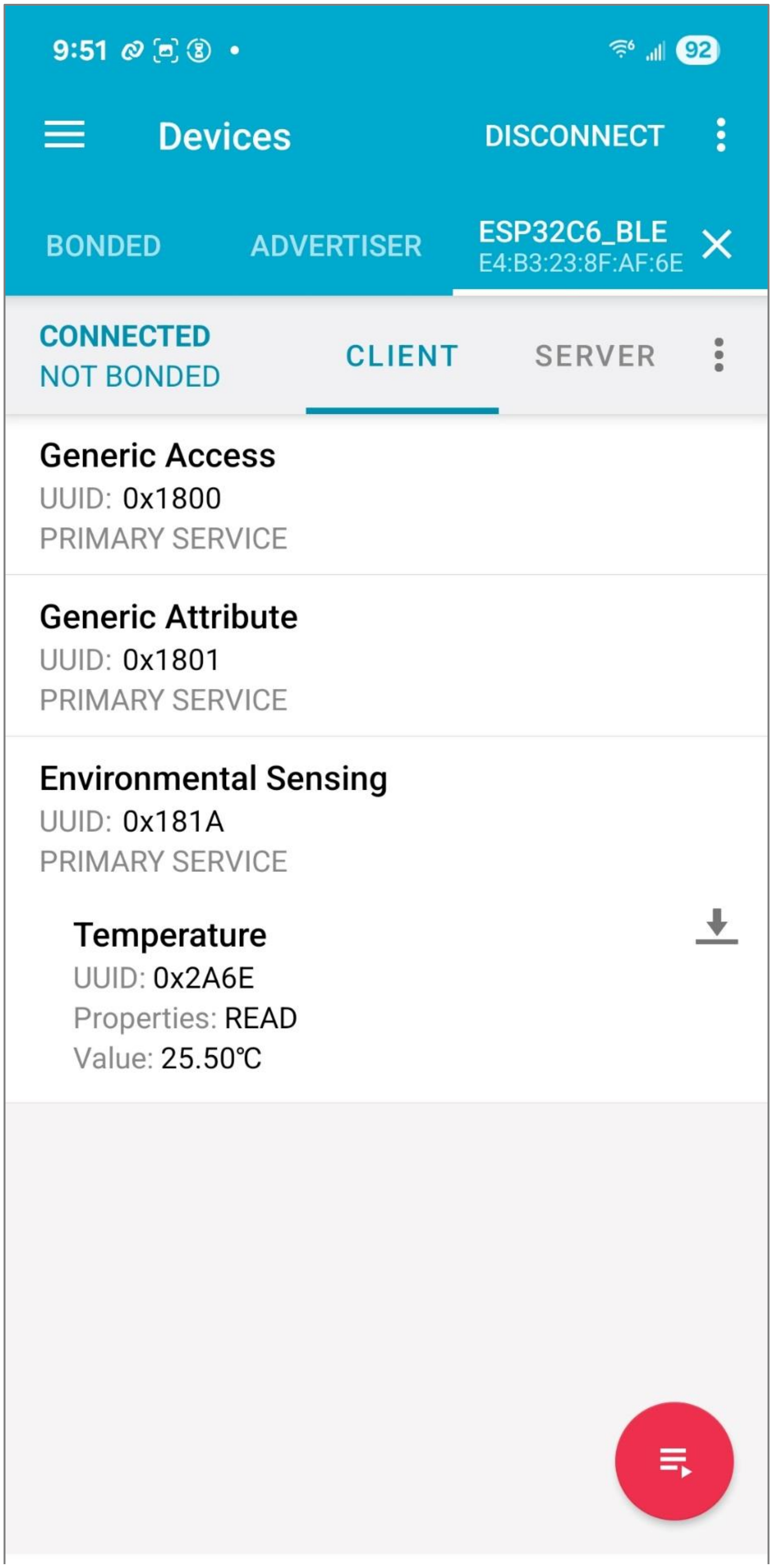
6. How to Test

LightBlue scanning for devices



6. How to Test

LightBlue connected to ESP32C6_BLE



7. Conclusion

The ESP32-C6 simplifies the process of integrating BLE 5.0 into modern IoT devices, delivering power efficiency and robust wireless capabilities. With ESP-IDF and FreeRTOS, developers gain full control of BLE interactions—from advertisement to characteristic notifications. The real-world GATT server example shown here serves as a solid foundation to build BLE-enabled sensors, beacons, or control nodes.