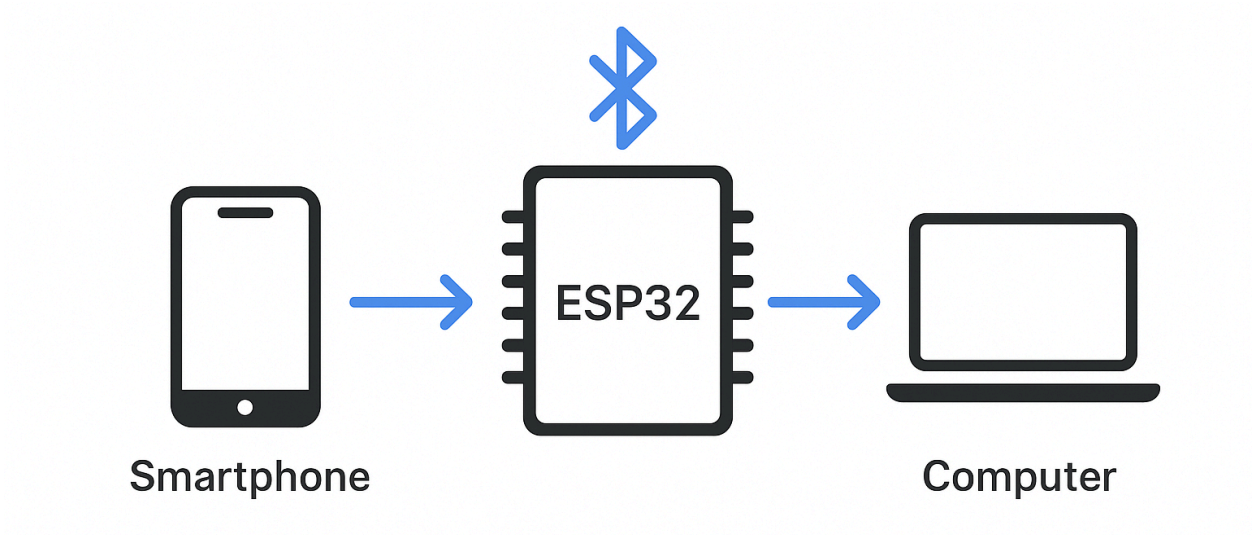
BLE Communication with ESP32



Author: S.A.C.A. Senanayaka

Affiliation: Unaffiliated

Email: achamath1@gmail.com

INTRODUCTION

Bluetooth Low Energy (BLE) is a low-power wireless protocol widely used for short-range device communication. In BLE, communication is based on the **GATT (Generic Attribute Profile)** hierarchy, where a **GATT server** (usually the *peripheral*) advertises **services** composed of **characteristics** (with values and optional descriptors)[1][2]. A **GATT client** (usually the *central*, e.g. a smartphone or PC) scans for advertising peripherals, connects, discovers services/characteristics, and then reads from or writes to characteristics. Only the GATT server holds the attribute database; the client accesses it over a BLE connection. Devices can also broadcast data without connection (beacons) or form multi-hop **BLE Mesh** networks, though typical ESP32-to-smartphone interactions use the GATT server/client model[3][4].

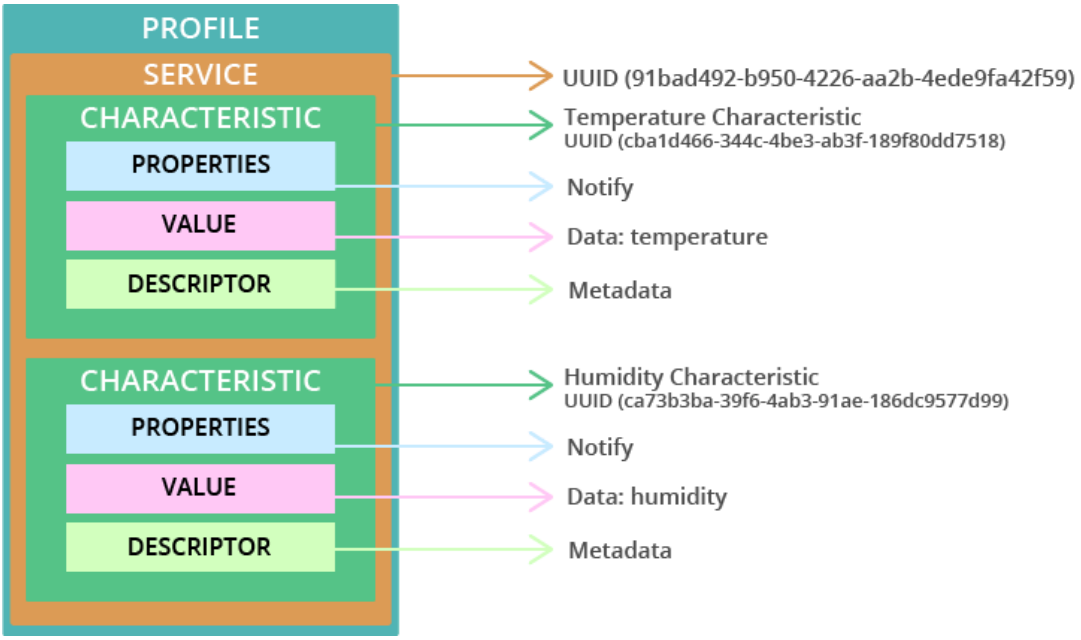


Figure 1: GATT hierarchy on ESP32 BLE server – a profile contains services, each service has characteristics (with properties like Notify) and descriptors[1][2].

Key concepts:

- **Roles:** A BLE *peripheral* (advertiser) acts as GATT server; a *central* (scanner) acts as GATT client. The peripheral advertises packets ("I'm here!") and waits for a central to connect【40†】. For example, an ESP32 can be a peripheral/server exposing sensor data, while a phone app acts as central/client to read it. Conversely, the ESP32 can also be a central to connect to other BLE peripherals (e.g. sensors)[5][4]. Typically a central can manage multiple peripherals simultaneously, but a peripheral usually connects to only one central[5].

- **GATT Structure:** A **service** groups related **characteristics** (e.g. a "Temperature Service" might have a Temperature Characteristic and a Humidity Characteristic). Each characteristic has a *value* and may support properties like Read, Write, Notify, or

Indicate. Descriptors (e.g. the Client Characteristic Configuration Descriptor) provide metadata and enable notifications. The BLE SIG defines many standard services/characteristics (Heart Rate, Battery, etc.); custom applications use custom UUIDs.

- **Data Exchange:** After connecting and discovering services, the client can **read** or **write** characteristic values, or **subscribe** to notifications/indications from the server. Reads fetch the current value; writes update the server's value. If a characteristic has the Notify property, the server can *push* new values to the client when changed, without polling. All of these operations follow the GATT protocol over the BLE Link Layer.

Communication Workflow: A typical BLE data exchange goes:

1. **Advertising:** ESP32 (as peripheral) starts advertising a defined service UUID so clients can find it (e.g. using `BLEAdvertising` APIs in ESP-IDF or Arduino).
2. **Scanning & Connect:** A smartphone/PC scans for the ESP32's advertisement. Upon selection, the client connects. (If pairing/bonding or security is needed, that is negotiated here.)
3. **Discovery:** The central discovers the GATT server's services and characteristics (by UUID). It reads the service/characteristic UUIDs and properties.
4. **Data Transfer:** The central reads or writes characteristic values. If enabled, the peripheral sends Notifications/Indications on characteristic changes. For example, the ESP32 can repeatedly update sensor readings and use BLE notifications to push them to the connected client. The central's app receives these and acts on them.

5. **Disconnection:** The devices can disconnect (e.g. on command or timeout). Either side can also initiate a new connection or continue scanning for others.

ESP32 as BLE Peripheral (GATT Server)

In this mode the ESP32 *advertises* and hosts a GATT server. It exposes one or more services with characteristics for data exchange. Developers typically define a service UUID and characteristic UUIDs (e.g. a custom service for a sensor, or use standard ones). The ESP32 BLE stack (in ESP-IDF or Arduino) handles the server role. A central device (like a phone) will scan for the ESP32, connect, then read/write/subscribe to those characteristics. All attribute data resides on the ESP32, so reads/writes go to its GATT server implementation. For example, the ESP32 might have a Temperature Service (with UUID 0x1809) containing a Temperature Measurement characteristic (with Notify enabled) that it updates and notifies.

On ESP32 (Arduino or ESP-IDF), setting up a peripheral/server involves:

- **Initializing BLE:** Create a BLE device name and start a BLE service (GATT server) with specified UUID[6][7].

- **Defining Services/Characteristics:** Add one or more services (using 128-bit or 16-bit UUIDs) and characteristics within them, each with properties (e.g. read, write, notify)[1][2]. Optionally attach descriptors (e.g. user descriptions, CCCD for notifications).

- **Advertising:** Configure and start advertising the service's UUID. The advertising packet may include the device name and service UUID【40†】. - **Handling Client Interaction:**

Implement callbacks/handlers for read, write requests, and notification status changes. For example, in ESP-IDF the GATTS callbacks (`esp_ble_gatts_cb_param_t`) indicate when a client reads or writes a characteristic. In Arduino-ESP32's `BLECharacteristic` callbacks, you can react to `onRead()` or `onWrite()`.

- **Sending Data:** To push data, the ESP32 updates the characteristic value and sends a notification (`notifyCharacteristicChanged` or similar API). The central receives it if it has subscribed.

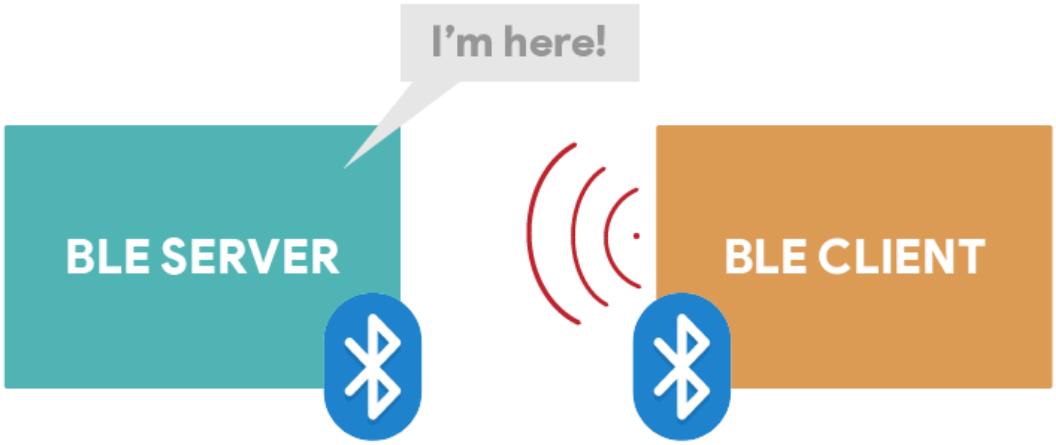


Figure 2: BLE Server (Peripheral) advertises "I'm here!" to BLE Clients. The client scans, connects, and then interacts via GATT.

Use Cases: Peripheral/server mode is used when the ESP32 should host data or services for other devices. Examples include sensor nodes reporting readings, BLE remote controllers, BLE-based HID (keyboard/mouse), or wireless data logging. Advantages: any central (phone/PC) can connect using standard BLE without extra hardware. Drawbacks: the ESP32 must constantly advertise, and only one client can

connect at a time (per default peripheral role) unless using BLE 5 extended connections.

ESP32 as BLE Central (GATT Client)

Alternatively, the ESP32 can act as a BLE *central/client* and initiate connections to BLE peripherals. In this role, the ESP32 scans for nearby advertising devices, connects to one or more, and accesses their GATT servers. This is useful if the ESP32 needs to read sensor data from a BLE sensor (e.g. a BLE heart-rate strap) or communicate with a phone app acting as a BLE peripheral. The ESP32 uses the GATT client APIs (e.g. ESP-IDF's GATTC functions or Arduino's `BLEScan`/`BLEClient`) to manage connections.

Typical steps on ESP32 client side:

- **Scanning:** Start BLE scan to find devices advertising desired service UUID(s). Filter results by name or service.
- **Connect:** Once a target peripheral is found, call the connect function to establish a BLE link.
- **Discover Services:** After connecting, perform service discovery on the remote device to retrieve its services and characteristics.
- **Subscribe or Read/Write:** If the peripheral supports notifications, you can subscribe to characteristics and receive asynchronous updates. Or you can periodically read/write characteristics.
- **Disconnect or Manage Multiple:** You can manage multiple simultaneous connections (limited by hardware) by repeating the above for each peripheral.

Use Cases: Central mode is used when the ESP32 should collect data from other BLE devices (e.g. wearable sensors, beacons) or act as a multi-protocol bridge (e.g. read BLE temperature sensor and send via Wi-Fi). It can also connect to a smartphone if the phone runs a BLE peripheral service (e.g. a custom app). Pros: ESP32 can aggregate data from many peripherals. Cons: more complex, as the code must manage scanning and connection logic, and performance is limited by the number of connections supported by its controller.

Standard Communication Workflows

The BLE GATT communication follows standard patterns on all platforms:

- **Client-Server Interaction:** After connection, the central reads the peripheral's advertised *primary services*. It then reads each service's characteristics and subscribes to needed notifications. For example, the central may do: `discoverServices()`, then for each service `discoverCharacteristics()`, then for each characteristic call `readCharacteristic()` or `setNotify(true)`. This pattern is common across Android, iOS, ESP32, etc.

- **Notifications & Indications:** To receive updates without polling, the central enables notifications on a characteristic (by writing to its CCCD descriptor). The peripheral then pushes new values as `OnCharacteristicChanged` events. In code, this appears as `gatt.setCharacteristicNotification()` (Android) or `characteristic.on('valueUpdate',…)` (Node.js) or similar. Indications are like notifications but require acknowledgement.

- **MTU Negotiation:** Many stacks allow adjusting the MTU (max data size) after connection to improve throughput. ESP-IDF has `esp_ble_gatt_set_local_mtu` and `esp_ble_gattc_send_mtu_req`, while Android/iOS handle this automatically.
- **Security:** If encrypted communication is needed, BLE pairing/bonding can be performed. This usually involves just setting the appropriate security level on the characteristic (e.g. requiring authentication). The devices exchange a key or use passkeys before data flows. Bonding stores the keys for future connections.
- **Disconnect and Reconnect:** The client can disconnect, or the peripheral can drop the connection. Clients can then scan again or reconnect by known address. Some apps auto-reconnect upon disconnection.

Data Exchange Protocols & Patterns

While BLE GATT itself defines the transport, higher-level *protocols* can be implemented. Common patterns include:

- **Custom GATT Services:** For example, defining a “UART Service” (like Nordic’s NUS) to simulate serial data over BLE. The ESP32 side uses a custom service with TX/RX characteristics, and the app treats writes to RX as “serial in” and sends back notifications on TX. This mimics a UART link.
- **Standard Profiles:** Alternatively, use existing SIG-adopted profiles (e.g. Heart Rate, Health Thermometer, Battery Level, etc.) so that generic apps (like nRF Connect) or OS components can understand the data without custom apps[8][9].

For instance, an ESP32 could present itself as a BLE Heart Rate Monitor by implementing the standard HR service and characteristic. The official GATT specifications require including the **GAP service** (for device name, appearance) and **GATT service**, though these are often auto-created by the BLE stack[9].

- **Mesh and Broadcast:** Beyond GATT, ESP32 supports **BLE Mesh** (Espressif's BLE Mesh stack) for many-to-many networks, and can broadcast periodic advertisement packets (like iBeacon or Eddystone) for one-way data (e.g. sensor beacons). Mesh allows multi-hop routing between ESP32 nodes[3].

- **Wi-Fi Provisioning via BLE:** A notable implementation is **Espressif's Blufi**. Blufi is an Espressif protocol for sending Wi-Fi credentials to an ESP32 over BLE[10][7]. In this use case, the ESP32 acts as a BLE peripheral with a Blufi service; a smartphone app connects and writes the SSID/password. This allows IoT setups without hard-coding network credentials. Espressif provides Android/iOS "Wi-Fi Provisioning" apps and a library (e.g. Arduino's `WiFiProv` example) for this functionality[10][7].

ESP32 BLE Software Stacks & Libraries

On the ESP32 itself, several BLE libraries are available:

- **Espressif ESP-IDF (C API):** The official framework has a full BLE (bluedroid or NimBLE) GATT server/client implementation. The ESP-IDF Programming Guide documents `esp_ble_gatts_register_callback`, `esp_ble_gatts_add_service`, `esp_ble_gatts_send_indicate`, etc. This is the most feature-complete and is kept

up-to-date by Espressif. It supports all BLE 4.x/5 features (advertising extensions, multiple connections, security modes). ESP-IDF examples include GATT server and client demos. **Pros:** Full control, high performance, BLE 5 features, official support. **Cons:** C API can be complex; steep learning curve.

- **Arduino-ESP32 Core (C++ wrappers):** Builds on ESP-IDF but with Arduino-style APIs. It includes either the Bluedroid or NimBLE stack (NimBLE is lightweight and often recommended). The `BLEDevice`, `BLEServer`, `BLEClient` classes (from libraries like NimBLE-Arduino) allow easy setup of services/characteristics. There is also `BLESerial` (deprecated) and example sketches (e.g. “BLE_Server”, “BLE_Client”). **Pros:** Easy to use, integrates with Arduino ecosystem, plenty of tutorials (RandomNerd). **Cons:** May use more RAM with Bluedroid; NimBLE has some limitations (only one active connection in older versions).

- **MicroPython/CircuitPython:** Both support BLE on ESP32. MicroPython (ESP32 port) has a `bluetooth` module allowing you to define services/characteristics and handle events in Python[11]. For instance, `bluetooth.BLE` can advertise and have callbacks on reads/writes. CircuitPython on supported boards (like Adafruit ESP32S2) offers a similar `adafruit_ble` library. **Pros:** High-level, quick prototyping; interactive. **Cons:** Limited by MicroPython’s BLE stack; not all features.

- **Espressif BLE Libraries (AT, etc):** If using the AT command firmware, commands like `AT+BLUFI=1` enable Blufi provisioning[10]. Developers often use Blufi via AT or the `WiFiProv` Arduino library as shown by RandomNerd[7].

- **NodeMCU/Other OS:** Less common for BLE; not a major platform here.

In all cases, the ESP32 libraries implement the GAP (advertising, connecting) and GATT (services, characteristics) layers per BLE specs. They also manage BLE roles (peripheral, central). Developers typically choose between **peripheral/server mode** and **central/client mode** depending on the application.

Mobile & Desktop BLE Libraries and Tools

Android: Uses the built-in `BluetoothAdapter/BluetoothGatt` APIs in Java/Kotlin. A typical app: call `BluetoothAdapter.startLeScan()`, then `gatt.connect()`, `gatt.discoverServices()`, and `readCharacteristic/setCharacteristicNotification` on `BluetoothGattCharacteristic`. Libraries/frameworks:

- **Android BluetoothGatt (native):** Full BLE functionality. Requires runtime permissions (BLUETOOTH_SCAN, etc) on newer Android[12].

- **RxAndroidBle:** A popular RxJava wrapper for BLE on Android, simplifying async flows.

- **Nordic BLE SDK for Android:** Bluetooth libraries and examples provided by Nordic Semiconductor, including UART, DFU, etc.

- **SmartGattLib, FastBle:** Community libraries for faster or simpler BLE code.

- **Pros:** Deep control, can handle both central/peripheral (limited). Android is common client. **Cons:** Verbose APIs, many permission/battery caveats.

iOS/macOS: Uses **CoreBluetooth** framework (Swift/ObjC). The central role uses `CBCentralManager` and `CBPeripheral` classes: scan with `centralManager.scanForPeripherals`, connect, and read/write with `CBPeripheral` calls. For peripheral role, `CBPeripheralManager` can advertise services (though less commonly used on iOS). Developer guides (e.g. Apple CoreBluetooth docs) detail methods. Libraries:

- **RxBluetoothKit:** RxSwift wrappers around CoreBluetooth.

- **CombineCoreBluetooth:** Uses Combine framework for modern Swift code.

- **AsyncBluetooth:** Concurrency helpers for CoreBluetooth.

- **Pros:** Deep integration, very stable; allows both central and (with iOS 13+) peripheral. **Cons:** Strict background restrictions, must declare usage descriptions in Info.plist[13].

Flutter: Cross-platform SDK that can target Android, iOS, desktop, and even Web. BLE libraries include:

- **FlutterBlue / FlutterBluePlus:** The original Flutter BLE plugin (now FlutterBluePlus) works on Android, iOS, macOS, Linux, and Web[6]. It supports BLE central only (no peripheral) and provides APIs for scanning, connecting, and GATT operations. FlutterBluePlus has extensive features and community support[6].

- **flutter_reactive_ble (Philips Hue):** A well-maintained plugin supporting multi-device BLE (central)[12]. It offers asynchronous scanning, connection management, and GATT operations across Android and iOS. It lacks built-in peripheral mode but emphasizes reliability.

- **FlutterBleLib:** Another plugin that even supports simulating peripheral mode[14].

Useful for testing and certain use cases.

- **Others:** react-native-ble-plx (if using React Native); Kotlin KMM libraries (BlueFalcon, Kable) for multiplatform.

Python (Desktop/Embedded):

- **Bleak:** A popular cross-platform BLE client library using asyncio[15][16]. It supports Windows 10+, Linux (BlueZ 5.55+), macOS, and Android (via Python-for-Android). It provides easy APIs to scan, connect, read/write GATT characteristics[15][16].

- **BluePy:** Python library for Linux (Raspberry Pi) using BlueZ.

- **PyGATT:** Multi-platform (uses BlueZ on Linux or a BLED112 USB dongle).

- **python-Bluetooth (PyBluez):** Generally for classic Bluetooth, less BLE-focused.

- **Pros:** Easy for quick scripts or PC apps; Bleak's async design is modern. **Cons:** Depends on OS BLE support; Windows requires certain updates; Linux uses BlueZ with sometimes complex setup.

Node.js (Desktop/Embedded):

- **Noble:** Node.js BLE central library (supports Windows, macOS, Linux)[17]. It provides `noble.startScanning()`, `peripheral.connect()`, and characteristic `read/write/on('data')` events. On Linux, Node must have HCI privileges. Noble is mature with many examples.

- **Bleno:** Companion library for Node to act as a BLE peripheral/server[17][18]. It allows advertising and exposing GATT services from Node.

- **Pros:** Good for IoT gateways or desktop apps in JS; works on Raspberry Pi, etc. **Cons:**

Less common than Python; requires Node native addons and Linux capabilities (e.g. `setcap` for BLE)[19].

.NET (C#/UWP):

- **Windows.Devices.Bluetooth (UWP):** Built into Windows 10+, allows BLE central (and now peripheral since Win10 v1703)[20][21]. You can enumerate with `BluetoothLEDevice.FromIdAsync`, discover GATT services/characteristics, and read/write them. Microsoft provides samples for BLE GATT clients/servers[21].
- **32feet.NET / InTheHand:** Third-party .NET libraries that support classic and BLE on Windows/Mac.

Linux (General):

- **BlueZ stack:** The official Linux Bluetooth protocol suite[22]. Tools include `bluetoothctl` (CLI for pairing/connect) and `gatttool` (old, now deprecated) for GATT. The BlueZ D-Bus API can be used from C, Python (via dbus), or other languages. Many IoT devices use BlueZ under the hood.
- **BLE Sniffing:** Tools like **Wireshark** with BLE adapter or *nRF Sniffer* hardware can inspect BLE packets for debugging.

MacOS:

- Uses **CoreBluetooth** like iOS. Also has tools like **PacketLogger** (Apple's macOS app) for capturing iOS BLE logs[23].

Web (Browser):

- **Web Bluetooth API:** Supported in Chrome/Edge, allows JavaScript in webpages to

scan and connect to BLE peripherals (must be served over HTTPS). Useful for web-based apps interacting with BLE devices. Example samples exist[24].

Mobile/Dev Tools:

- **nRF Connect (iOS/Android/desktop):** A widely-used app to scan, connect, and explore any BLE device's services[25]. (Nordic also offers Toolboxes.)

- **Adafruit Bluefruit (iOS/Android):** Apps for interacting with Adafruit BLE custom services.

- **Bettercap/LightBlue:** Tools for hacking or testing BLE (Bettercap for security research)[25].

- **Arduino IDE Examples:** For quick prototyping using ESP32.

- **Visual Studio/UWP samples:** For Windows developers (see Microsoft sample[20]).

Library Comparison (Pros/Cons & Use Cases)

- **Espressif IDF vs Arduino:** IDF is robust and best for production; Arduino is easier for hobbyists. Arduino's NimBLE uses less memory than Bluedroid, but may limit connections.
- **FlutterBluePlus vs ReactiveBle vs BLELib:** FlutterBluePlus is easy and cross-platform[6] but only supports central. FlutterReactiveBLE is well-maintained (by Philips Hue) and reliable[12]. FlutterBLELib can simulate peripheral. Choose based on need for peripheral mode and community support[26][27].

- **Android (BluetoothGatt) vs RxAndroidBle/Nordic SDK:** Native API gives full control but is verbose. RxAndroidBle offers a nicer API with observable streams. Nordic's SDK provides ready-to-use components for common tasks (e.g. UART service).
- **iOS (CoreBluetooth) vs RxBluetoothKit:** CoreBluetooth is stable; RxBluetoothKit (or CombineCoreBluetooth) is nicer in Swift for reactive coding. Peripheral mode is available on iOS but limited in background.
- **Bleak vs BluePy vs PyGATT:** Bleak is cross-platform and actively maintained[15], whereas BluePy is Linux-only (Pi), PyGATT depends on backend (e.g. BLED112). For multi-OS, Bleak is recommended.
- **Node.js (Noble/Bleno):** Good for JavaScript projects (e.g. Electron apps). Noble is full-featured but requires some OS setup. Bleno makes it possible to emulate a BLE device on a PC/RPi.

Use Cases:

- **IoT Sensor Networks:** Use ESP32 as peripheral nodes broadcasting sensor data via GATT, a smartphone or gateway central collects data. Or ESP32 as central reading BLE-only sensors.

- **Wearables & Health:** ESP32 can emulate standard health devices (HRM, thermometer) to connect with phones.

- **Smart Home:** BLE allows smartphone control of ESP32-based lights, locks, etc.

- **Industrial:** BLE Mesh on ESP32 for building automation (lights, AC sensors, etc.).

- **Beaconing:** Use BLE advertisements (as iBeacon/Eddystone) for proximity or location

services (no connection needed).

- **Configuration & Provisioning:** As noted, use BLE for initial setup (Wi-Fi credentials) or sending commands to headless devices.

- **Audio/Video:** (ESP32 also supports Bluetooth Classic A2DP, but not BLE; out of scope for BLE discussion.)

Summary

Communicating with an ESP32 over BLE involves understanding BLE's GATT architecture (services/characteristics) and roles (peripheral vs. central). The ESP32 can serve as either a BLE server (peripheral) or client (central), exposing or consuming GATT attributes. Standard workflows involve advertising, scanning, connecting, service discovery, and then GATT read/write/notification exchanges. Tools and libraries abound for every platform: on ESP32 (ESP-IDF, Arduino, MicroPython) and on clients (Android/iOS native APIs, cross-platform Flutter/React Native, desktop libs like Bleak or Noble). Each library and framework has trade-offs in ease, performance, and support. Current trends favor cross-platform solutions (e.g. Flutter, Bleak) and efficient stacks (NimBLE). References: Espressif documentation and examples, RandomNerdTutorials guides[1][7], Microsoft BLE samples[21], and community libraries (FlutterBluePlus[6], ReactiveBle[12], Noble/Bleno[17][18], Bleak[15][16], etc.).

References: Official docs and libraries cited above. Each platform's SDK documentation should be consulted for API details. The BLE SIG specifications (GATT, GAP) define the

core protocols[1][8]. The “Awesome BLE” list[28][25] and various tutorials offer additional resources on libraries and patterns.

---

[1] [2] [3] [4] ESP32 BLE Server and Client (Bluetooth Low Energy) | Random Nerd Tutorials

https://randomnerdtutorials.com/esp32-ble-server-client/

[5] [26] [27] Flutter Bluetooth Low Energy (BLE): Developer's Guide - LeanCode

https://leancode.co/blog/bluetooth-low-energy-in-flutter

[6] GitHub - chipweinberger/flutter_blue_plus: Flutter plugin for connecting and communicationg with Bluetooth Low Energy devices, on Android, iOS, macOS, Web, Linux, Windows.

https://github.com/chipweinberger/flutter_blue_plus

[7] ESP32 Wi-Fi Provisioning via BLE (Arduino IDE) | Random Nerd Tutorials

https://randomnerdtutorials.com/esp32-wi-fi-provisioning-ble-arduino/

[8] [9] Bluetooth GATT: How to Design Custom Services & Characteristics

https://novelbits.io/bluetooth-gatt-services-characteristics/

[10] AT Network Provisioning Examples - ESP32 - — ESP-AT User Guide latest documentation

https://docs.espressif.com/projects/esp-at/en/latest/esp32/AT_Command_Examples/network_provisioning_examples.html

[11] MicroPython: ESP32 - Getting Started with Bluetooth Low Energy ...

https://randomnerdtutorials.com/micropython-esp32-bluetooth-low-energy-ble/

[12] GitHub - PhilipsHue/flutter_reactive_ble: Flutter library that handles BLE operations for multiple devices.

https://github.com/PhilipsHue/flutter_reactive_ble

[13] Apple's IOS Core Bluetooth: The Ultimate Guide – Punch Through

https://punchthrough.com/core-bluetooth-guide/

[14] GitHub - dotintent/FlutterBleLib: Bluetooth Low Energy library for Flutter with support for simulating peripherals

https://github.com/dotintent/FlutterBleLib

[15] [16] GitHub - hbldh/bleak: A cross platform Bluetooth Low Energy Client for Python using asyncio

https://github.com/hbldh/bleak

[17] [19] GitHub - noble/noble: A Node.js BLE (Bluetooth Low Energy) central module

https://github.com/noble/noble

[18] GitHub - noble/bleno: A Node.js module for implementing BLE (Bluetooth Low Energy) peripherals

https://github.com/sandeepmistry/bleno

[20] [21] Bluetooth Low Energy sample - Code Samples | Microsoft Learn

https://learn.microsoft.com/en-us/samples/microsoft/windows-universal-samples/bluetoothle/

[22] [23] [24] [25] [28] GitHub - dotintent/awesome-ble: A collaborative list of Awesome Bluetooth Low Energy (BLE) resources. Feel free to contribute!

https://github.com/dotintent/awesome-ble