# **Chapter 4** Making Bluetooth Wireless Sensor Energy Efficient

In this exciting chapter, you will explore Bluetooth Low Energy (BLE) with the ESP32, opening a world of possibilities for developing energy-efficient wireless sensor nodes. As we explore the fundamentals of BLE, our focus is on building a wireless sensor with ESP32 by understanding the creation of the Bluetooth Profile, a BLE key framework for organizing data exchange between devices to create a data structure with multiple fields. You will learn the fundamentals of BLE through practical experience and how to read and write data using the intuitive BLE scanner app as the user interface for the ESP32. This chapter lays the groundwork for an extensive exploration of the revolutionary potential of BLE technology with the ESP32, providing a strong foundation for creative and effective IoT solutions through practical insights and detailed instructions.

In this chapter, you will be able to

- Understand the basic concepts of Bluetooth Low Energy, such as its architecture, key components, and communication fundamentals.
- Discover how BLE organizes data and enables smooth device communication by delving into the Generic Attribute Profile (GATT).
- Gain hands-on experience interacting with the ESP32 using a BLE scanner application as a user interface.
- Discover the best practices for handling possible issues, reducing risks, and guaranteeing the security and integrity of data transferred over BLE.
- Apply acquired knowledge and skills to create a powerful Bluetooth sensor capable of advertising sensor data to nearby devices with wireless location services and interaction using a BLE scanner app.

## 4.1 Introduction to Bluetooth Low Energy

In the previous chapter, we discussed that BLE, also known as *Bluetooth Smart*, is a different Bluetooth technology introduced in 2010 and is intended for markets where ultra-low power consumption is preferred over high throughput [1], [2]. Compared to Bluetooth Classic devices, BLE uses a transmission power between 1/10 and 1/100 (10mW to 100mW). This is accomplished using a BLE radio for short-term data communication, which does not require a high operating frequency. A typical BLE use case is to turn the radio on occasionally, send and receive a few bytes or kilobytes of data, and then switch it off or return to sleep, unlike Bluetooth Classic use cases, where a connection is kept open for an extended period to ensure the lowest latency, and the radio is not turned off.


**Figure 1** *Bluetooth Smart vs. Bluetooth Classic*.

This technology was developed by Bluetooth SIG with a focus on various industries, including home entertainment, security, and healthcare as depicted in **Figure 1**. *Bluetooth 4.0*, or BLE, is designed to enable low-speed data transfers for IoT applications such as sensors and tags so they can communicate even with limited battery capacity [3]. *Bluetooth 5.0*, introduced in 2016, is an advancement of BLE that improves speed, range, and data capacity [4]. BLE employed in smartphones provided the opportunity for you to massively scale their operations.

For smart devices operating on local networks, BLE is a critical technology that ensures coordination and data exchange without relying on external connectivity. The idea of a BLE gateway appears to bridge the gap between local BLE networks and the growing online ecosystem [5]. For example, many wearable devices use BLE to communicate with Internet-connected smartphones, tablets, or computers. This device, in turn, forwards data to the cloud, where they are stored and referenced for tools such as dashboards. In these scenarios, an Internet-connected device essentially acts as a gateway between the wearable device and the cloud services on which it relies. As a result, wearable devices use minimal power for data transfer and remain up to date every time they sync with smart devices.

### 4.1.1 Advantages of Bluetooth Low Energy

As an IoT developer, BLE offers several benefits, making it a popular option for various applications [1], [6]. The following are some significant benefits:

- **Low Power Consumption**: BLE's incredibly low power consumption is one of its main benefits. For this reason, it works perfectly with battery-powered devices and increases the lifespan of connected devices, which is important for IoT applications where energy efficiency is crucial.
- **Compatibility with Smartphones**: Most modern mobile phones have built-in BLE support, which practically benefits communication between mobile apps and IoT devices across different operating systems. Because IoT devices can readily connect to and communicate with smartphones, increasing the possibilities for user interaction, this compatibility streamlines the development process.
- **Open Documentation Access**: Open documentation and standardized protocols are key features of BLE and provide developers with a lot of material and support, which can be downloaded for free from the Bluetooth SIG official website. This accessibility promotes interoperability and enables communication between devices from different manufacturers.
- **Fast Connection and Data Transfer**: BLE connections are established quickly, allowing for rapid data transfer between devices. This is particularly important for applications that require real-time data updates, such as health monitoring and industrial-control systems.
- **Beacon Technology**: Developers can build proximity-enabled applications by leveraging BLE beacons to enable location-based services. This is particularly useful in indoor navigation, retail, and other situations where accurate location data are important.

BLE is an appealing option for IoT developers looking for effective, scalable, and standardized wireless connectivity solutions because of its low-power consumption, open documentation access, smartphone preference, and other benefits. As BLE technology continues to develop, IoT developers like you can leverage these benefits to create innovative and energy-efficient solutions across various industries.

## 4.1.2 Limitations of Bluetooth Low Energy

Although BLE offers a lot of benefits to IoT developers, certain drawbacks need to be considered when creating IoT applications. For developers to select the appropriate technology for their particular use cases, they must be aware of these constraints to make well-informed choices [1], [7]. The following are the main BLE limitations that developers need to be aware of:

- **Limited Data Throughput**: Compared to other wireless technologies, BLE has limited data throughput because it is designed for low-power communications over a short distance. It is therefore suitable for applications with relatively short and intermittent data flows, but not for applications that require high bandwidth and continuous data transmission, such as media streaming.
- **Limited Range**: The BLE is primarily designed for communication within typically tens of meters or less. Although this range works well for many IoT scenarios, the limitations of BLE may be an obstacle for applications requiring long-distance communication. If a longer range is required, you should carefully consider the required distance of communication for their application and consider alternative technologies in this respect.
- **Interference in Congested Areas**: The BLE uses the frequency band 2.4 GHz, which is also used by Wi-Fi and other wireless technologies. Interference may occur in situations where there are a large number of wireless devices and people living in the area, which may affect the reliability of BLE communications. You should consider the operating environment and potential sources of interference.
- **Internet Connectivity Requires a Gateway**: BLE service does not provide direct Internet access. Often, you need a gateway device such as a smartphone or a special BLE-to-Wi-Fi bridge to connect your BLE device to the Internet. This new requirement will make the architecture of the IoT more expensive and complex.
- **Complexity of Services and Profiles**: BLE organizes data using a hierarchical structure of service and characteristic data. Although this structure is flexible, it may add complexity, especially in multi-service and multi-function applications. It may be difficult to understand and configure the BLE profile.

Understanding these limitations will allow you to make informed decisions, choose BLE when its benefits match the needs of the application and explore alternative technologies in situations where BLE's limitations may be a barrier. In the ever-changing world of the development of IoT, understanding the advantages and disadvantages of BLE is essential to develop reliable and effective solutions.

## 4.1.3 Applications for Bluetooth Low Energy

First released in 2010, the BLE protocol was primarily intended for IoT applications that enable low-power data transmissions. BLE has been used in a variety of applications over the past decade, from industrial manufacturing to consumer devices [1]. Here are some of the most common uses we encounter in today's world:

- **Home Automation**: Straightforward communication between devices is facilitated by BLE, which is a key component of home automation. BLE allows energy efficient and connected household appliances, from door locks and security systems to smart lighting and thermostats.

- **Fitness Tracking**: BLE is used by wearables such as fitness trackers and smart watches to facilitate fast transfer of data between the device and the smart phone. By providing real-time monitoring and analysis of health metrics, the technology provides users with personalized insights into their health and fitness.
- **Audio Devices**: The Bluetooth protocol is widely used by audio devices such as portable speakers, wireless headphones and earphones, in order to enable energy-efficient pair-pairs. This maximizes battery life and provides reliable and excellent wireless audio experience.
- **Item Finding Tags**: BLE tracks and locates lost items by means of small, battery-powered tags. Apps such as search for keys, wallets and even bags use low-power BLE for longer battery life.
- **Targeted Ads**: The retail and marketing sectors use BLE for proximity advertising. As consumers approach specific points, they can receive targeted advertising or special offers, which improves the shopping experience.
- **Inventory Management**: In the industrial setting, BLE is used to manage inventories efficiently. Tracking of the movement and location of the goods in the storage area is simplified, leading to better logistics and lower running costs.
- **Industrial Automation**: In industrial automation, BLE is increasingly used for monitoring and controlling machines. Its low power consumption and reliability make it suitable for applications in which communication in real time is essential.
- **Healthcare Monitoring**: The BLE is useful for health applications and allows the development of wireless health monitoring devices. From continuous glucose monitoring to remote patient monitoring systems, BLE helps to transfer vital health information to health care professionals.
- **Smart Agriculture**: In precision agriculture, the BLE is used to monitor crop health, equipment condition and soil conditions. This will allow farmers to optimize crop yields and resource use by making data-driven decisions.

Many of the most popular applications on the IoT would not be possible without BLE. By reducing energy consumption, IoT devices can be significantly smaller and have a longer lifespan. Thus, BLE is responsible for many of the wearables, tags and other smart devices that we use today.

## 4.2 The Basic of BLE Protocol Architecture

The basic BLE protocol architecture is based on client-server model, with devices playing a peripheral or central role. Typically, a central device establishes a connection and retrieves the necessary data, while a peripheral device hosts the data or service. The protocol for exchanging data, known as the GATT (General Attribute Profile), classifies information into attributes and services [8]. Characteristics are the actual data points contained in the services, which are collections of related data points. This hierarchical structure, combined with the ease of use and efficiency of BLE, provides the basis for wireless communication in the various IoT applications, ranging from smart devices to sensor networks. The three basic layers of the basic protocol architecture, Application, Host, and Controller are represented in **Figure 2** as the BLE core operations [1], [9], [10]. Understanding this layered framework is essential if you want to make effective use of the BLE capabilities in your IoT projects.

### 4.2.1 Application Stack

The application is at the highest level of the system and serves as an interface to interact directly with end-users and facilitate the implementation of the various applications of the IoT. Here developers design and implement custom services and features that define the behavior of BLE devices and adapt them to specific use-cases. This layer encapsulates the application logic and ensures that Bluetooth-enabled devices deliver the intended functionality.

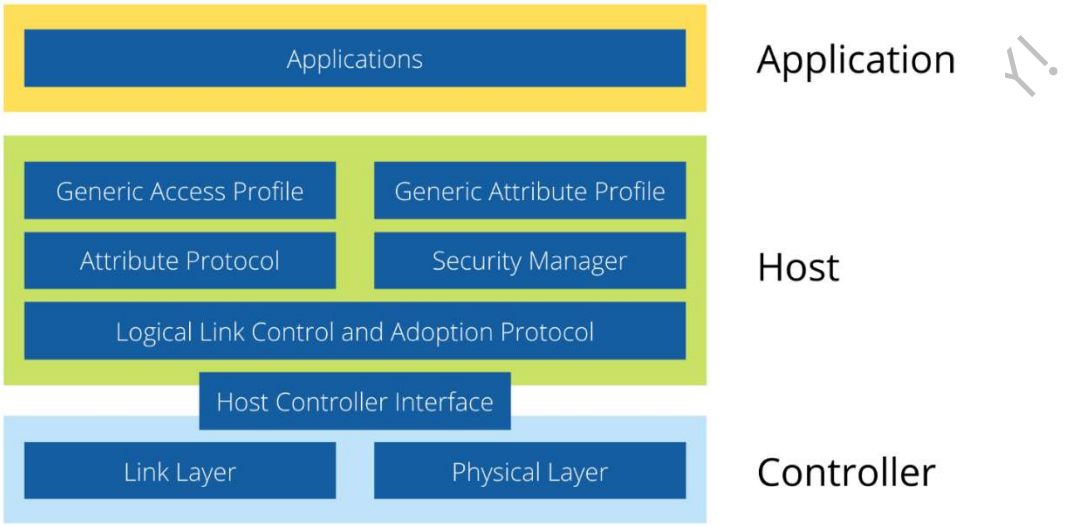



**Figure 2** *Bluetooth Low Energy protocol stack architecture*.

### 4.2.2 Host Stack

The host stack acts as a gate between the application stack and the controller stack. One of its key components is the general access profile (GAP), the basic framework for describing how BLE devices access and communicate with one another [11], [12], as shown in **Figure 3**. GAP defines the roles of BLE devices in the interaction and breaks them down into four different roles as follows:

- **Broadcaster**: The device sends out advertising packets regularly but does not establish a connection, which makes it suitable for scenarios such as beacons. In retail, BLE receivers could be strategically placed throughout the shop and continuously broadcast advertisement or promotion content to the users' smart phones. Customers with BLE enabled devices will receive these messages as they move within the broadcasters' range.
- **Observer**: Receiving equipment which scans for advertising and collects data without connecting to a broadcaster. A BLE-enabled smartphone can act as a museum-like observer. As visitors move around, their phones continuously scan the BLE signals of the surrounding exhibits, providing an interactive and personalized experience by providing them with relevant information about the works of art or artifacts.
- **Central**: Devices such as smartphones and tablets that initiate a connection and essentially enable the device to gain access to the network. It does scans to detect nearby edges. Once the peripheral device has been identified, the control center can establish a connection to exchange the data. The central device takes over the connection and controls the process of communication. It can collect data from one or more peripheral devices, request information from them, or control them from the peripheral device itself. Imagine a fitness tracker hooked up to your smartphone. The

fitness tracker collects step and heart rate data, and the smartphone is the hub for gathering this information for a comprehensive application for health tracking.

- **Peripheral**: Devices such as sensors or other low-power devices respond to requests for connectivity from central servers. It displays packets containing information about its services to allow the central device to locate and connect to it. Peripherals are typically designed to consume less energy, which makes them suitable for energy-intensive scenarios. In an industrial environment, sensor node monitoring machines could act as a peripheral device, periodically sending data to a central device such as a control station, which could be used to monitor and analyze the data in real time.

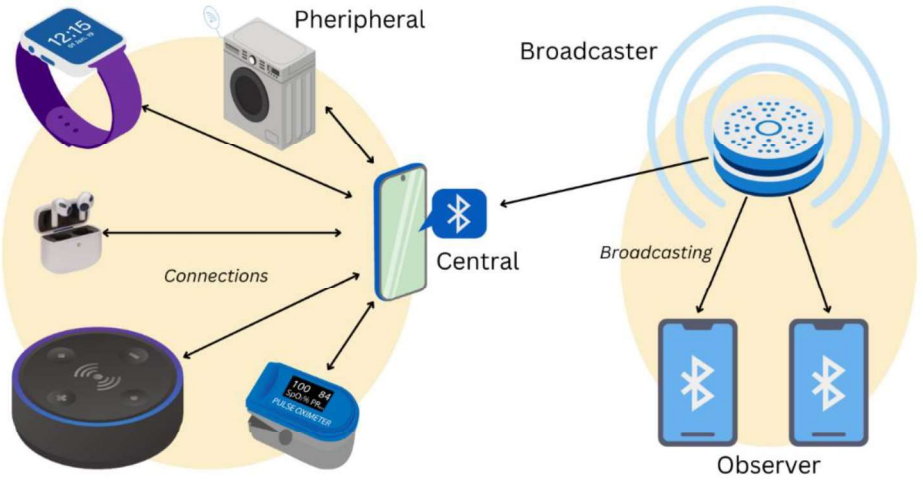



**Figure 3** *Bluetooth Low Energy has four different types of connections based on the network topology.*

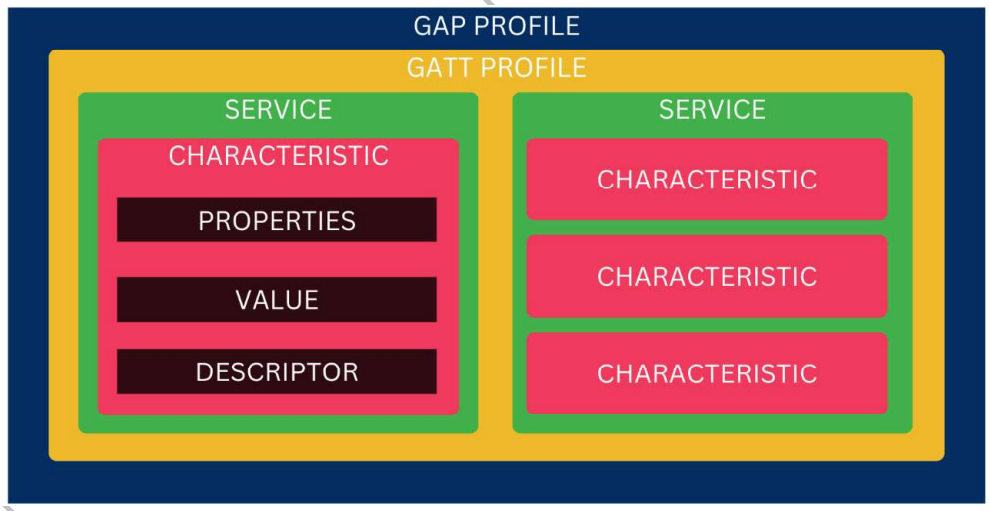



**Figure 4** *GAP and GATT profile hierarchy for BLE applications.*

The BLE GATT provides the basic framework for defining how data or attributes are structured and exchanged between connected devices [8]. **Figure 4** shows how data is arranged in a hierarchy of parts called services, which act as containers and logically group related pieces of user data known as attributes. Like the GAP, GATT also has various functions to facilitate the exchange of data between installations, such as follows.

- **Client**: Initiates a read or write request to or from the server for attributes. This is the entity that typically consumes or uses the data supplied by a server. In the fitness tracking application, the client is the smartphone. Send requests to the fitness tracker as a server to get data such as the number of steps or the heart rate.

- **Server**: Stores and provide access to attributes. It responds to requests from clients by either sending data or by confirming that the written transactions have been completed. If we continue with the scenario with the fitness tracker, the fitness tracker is the server. It stores attributes like steps and heart rate and responds to smartphone requests.

Understanding the roles of server and client is essential for BLE communications. Client initiates data read or write requests, and the server replies to these requests [13], [14]. When the client wishes to communicate with the server, he uses the GATT procedures to discover services and properties, to read or write attribute values, and to receive notifications that specify how the data is structured and transmitted. In the BLE connection phase, data is transmitted, received and processed based on the Attribute Layer (ATT) [8]. The system operates on a client-server model, where data is stored on a server and can be accessed directly by a client.

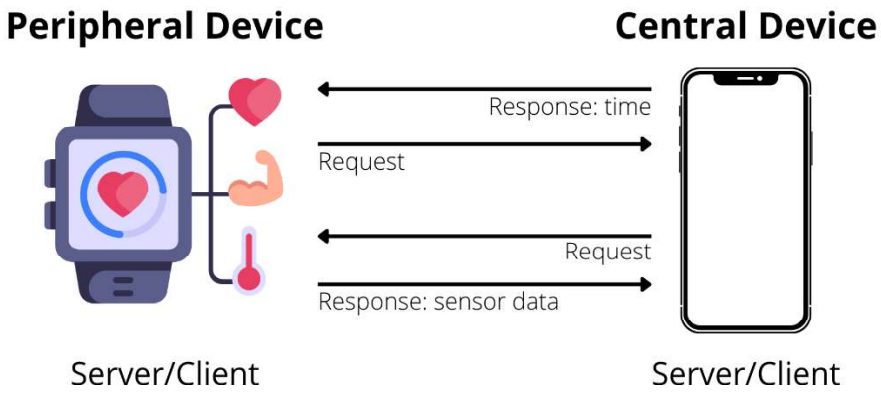



**Figure 5** *A fitness tracker and smartphone interaction show Bluetooth device roles as client and server.*

It should be noted that the GATT tasks are entirely independent of the tasks of the GAP. This means that GAP Central and the GAP peripheral may act as GATT clients or servers, or both. Consider the example shown in **Figure 5**, a Bluetooth fitness-tracking device that communicates with a smart phone. The GAP role of the fitness tracker is peripheral and acts as a GATT server when the phone requests data from the sensors of the tracker. It may also sometimes act as a GATT client, requesting precise time data from the smart phone to update its internal clock to record data. The GATT client and server roles depend only on the direction of the data request and response transactions, while the GAP roles, which are peripheral to the fitness tracker and central to the smart phone, are immutable.

The GATT profile, services and characteristics, as shown in **Figure 4**, constitute the basic structure of BLE communication and allow standardized interaction between devices. Profiles are predefined collections of services, compiled by Bluetooth SIG or peripheral developers. Services are used to break data down into logical entities and contain data that are called attributes. A service may have one or more properties, and each of these properties distinguishes it from other services by using a unique identifier (UUID), which may be 16-bit (official BLE services) or 128-bit (custom BLE services) [15]. The characteristic is the container of user data. Contains at least two attributes such as read, write, or notify characteristic property declaration and a characteristic value field. Like services, they have a UUID, which may be based on 16- or 128-bit data, depending on whether the characteristic has a standard or a custom definition. The descriptor shall provide additional information on the property and its value. Descriptor is associated with 16-bit UUID.

Taking the example of the use of wireless blood pressure measuring devices illustrated in **Figure 6**, GATT defines the structure by services, each of which is identified by a unique

identifier. In this case, UUID 0x1810 represents the service of blood pressure and 0x180A the service of device information. These services contain specific blood pressure data and device information and provide attribute values in a standardized and organized format.

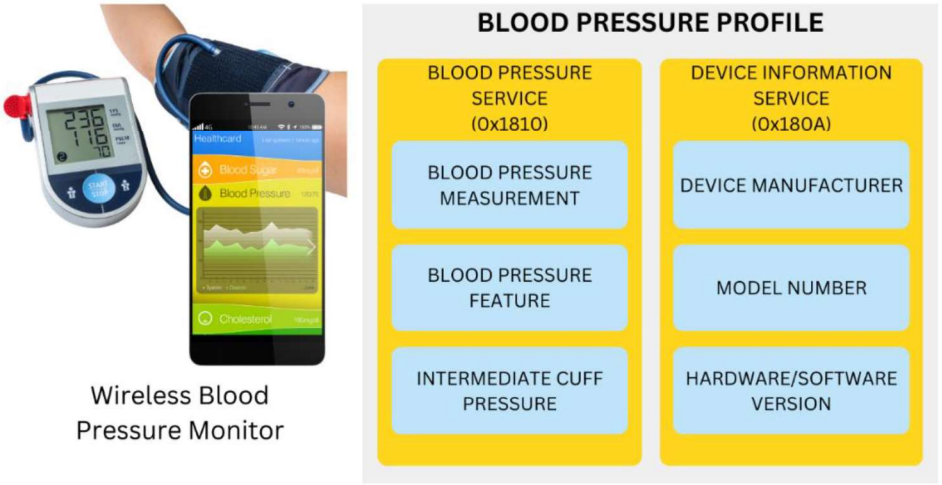



**Figure 6** *A wireless blood pressure device example profile includes two services, blood pressure and device information.*

Using this architecture, you can quickly retrieve and understand important device health metrics and information and navigate the device service accurately by focusing on the UUIDs involved. The Bluetooth website contains the official 16-bit UUIDs of the GATT services, attributes and descriptors specified by the Bluetooth SIG [15]. The official Bluetooth documentation contains an up-to-date list of assigned Bluetooth numbers, codes, and identifiers that are used to identify Bluetooth devices.

## 4.3 Bluetooth Low Energy Network Topology

The BLE device can communicate with the outside world in two ways, by broadcasting or by connecting to it. Each network topology has its own advantages and constraints, both of which are covered by the GAP guidelines [10], [16].

### 4.3.1 Broadcast Topology

As shown in **Figure 3**, a broadcast topology is a multi or single-point wireless communications network that can send data to any scanning device or receiver in the listening range. In a broadcast topology, devices act as a broadcaster or an observer. Broadcasters regularly broadcast packets into the environment in which potential observers can capture and interpret the data transmitted. It is useful in scenarios where devices need to disseminate information to potential consumers without connecting to the network.

An example of this topology is iBeacon, introduced in 2013, as shown in **Figure 7** [17], Apple's implementation of BLE wireless technology enables iOS devices to listen for and understand the location of a beacon in a confined area, and to deliver location-based information to users. The iBeacon application running on the receiving device uses the received signal strength indicator (RSSI) of the iBeacon nearby to estimate the distance between the receiving device and the beacon [18]. RSSI is monitored for any beacon within range of the device that is running the iBeacon application. The measured RSSI of the signal varies as the iPhone travels through space. RSSI decreases generally as the distance between the iPhone and the iBeacon increases.


**Figure 7** *Apple iBeacon technology uses BLE enabling location awareness for apps. Adopted from TopSoftwareCompanies* [19].

You can use BLE technology to design applications that will intelligently respond to the presence or absence of a beacon device. For example, a retail application may push promotions to the user's smartphone if it detects the proximity of a particular sign in the store [20].

## 4.3.2 Connections Topology

In a connection-oriented approach, the device establishes a connection prior to exchanging data. Here the devices act as central or peripheral appliances. The central server initiates the connection and re-examines the advertising packets. It manages the timing and starts a regular exchange of data. The peripheral receives incoming connections and sends out packets of advertisements periodically. It follows the timing of the data exchange at the central facility. Once a connection is established, devices only communicate with each other in a two-way way, so the connection topology is a private network.

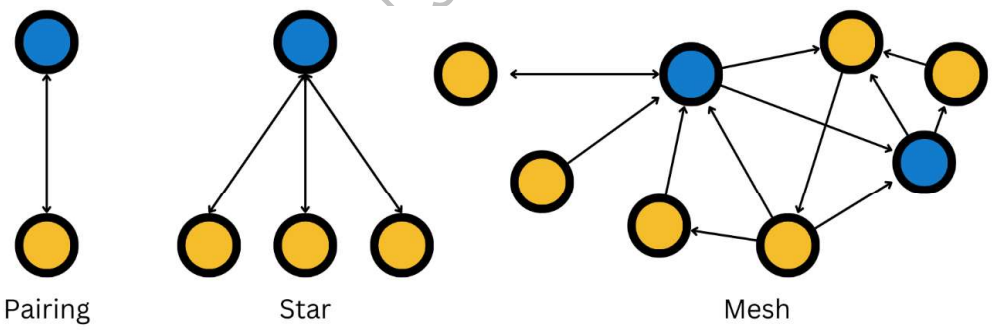



**Figure 8** *Bluetooth Smart connections network topology.*

A BLE connection network is a regular exchange of data at a specified time between two peers. Thus, a connection can be a point-to-point network (i.e., pairing), a point-to-multipoint network (i.e., star) or a multipoint-to-multipoint network (i.e., mesh), as shown in **Figure 8**. BLE topologies can be freely mixed within the larger BLE network. These BLE links are topology mesh networks [16], [21]. The mesh topology enables BLE to be used for the creation of large networks of devices. It is ideal for control, monitoring and automation systems in which it is necessary for dozens, hundreds or thousands of devices to communicate reliably and securely with one another.

As shown in **Figure 9**, a notable use of BLE is the setting of home automation by mesh networks. BLE mesh networks allow wireless communication and coordination between a multitude of devices and promote a reliable and scalable infrastructure. In a typical smart home scenario [22], BLE sensors, lights, thermostats and other appliances form a mesh network

that enables them to effectively transfer data and instructions. This mesh topology improves reliability and expands the range of communication, enabling devices in remote areas of the home to communicate efficiently with each other. With the introduction of standardized home automation profiles, such as Bluetooth SIG's Smart Home Subgroup [23], you can create interoperable solutions that improve the overall efficiency, convenience, and security of modern homes.


Figure 9: *Smart home devices through Bluetooth mesh network.*

### 4.3.3 Advantages and Limitations of Network Topology

Each topology has its advantages and limitations [12], [16]. Broadcasting excels in scenarios that require simplicity, scalability, and low power consumption, making it ideal for applications such as beaconing or simple data dissemination. However, in scenarios that require two-way communication or data security, this is not the case.

On the other hand, mesh networks provide a more sophisticated channel of communication, allowing for two-way data exchange and synchronous communications. This makes links suitable for scenarios requiring interactions in real time or for more complex exchanges of data. However, connections imply higher energy consumption and a more complex setup process.

When navigating the BLE network topology, you must carefully consider the benefits and limitations of each network topology to determine the most appropriate approach for the specific application. Whether it's broadcasting digital messages over the air or creating complicated connections, the BLE network topology offers versatile solutions for wireless interactions.

## 4.4 Get Started with Bluetooth Smart Using ESP32

This section will focus on the practical implementation of BLE on an ESP32 board using the Arduino IDE. ESP32, with its BLE capabilities on board, offers a versatile platform for developing BLE applications with the familiar Arduino IDE. The core support for Arduino-ESP32 simplifies integration of BLE features and makes them available to developers of all skill levels.

### 4.4.1 ESP32 as BLE Server

To get started on your BLE journey, the Arduino IDE provides examples for ESP32, navigate to "**File**" > "**Examples**" > "**BLE**" > "**Server**" to access an introductory example. **Listing 1** introduces fundamental concepts such as setting up a BLE server, defining characteristics, and handling BLE events. By exploring this example, you can gain insights into the structure of

a BLE server application and comprehend the interplay between the server and connected devices.

**Listing 1** *ESP32 BLE Server*

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

These statements include the libraries necessary for working with BLE on the ESP32. The BLEDevice library provides functions for initializing and creating BLE devices, while BLEUtils and BLEServer offer utilities and server-related functionalities, respectively.

```
// See the following for generating UUIDs:
// https://www.uuidgenerator.net/

#define SERVICE_UUID        "4fafc201-1fb5-459e-8fcc-c5c9c331914b"
#define CHARACTERISTIC_UUID "beb5483e-36e1-4688-b7f5-ea07361b26a8"
```

This section defines the custom UUID for the GATT service SERVICE_UUID and characteristic CHARACTERISTIC_UUID. The 128-bit custom UUID can be generated online through this link https://www.uuidgenerator.net/. UUIDs are essential for uniquely identifying services and characteristics in a BLE application.

```
void setup() {
  Serial.begin(115200);
  Serial.println("Starting BLE work!");

  BLEDevice::init("Long name works now");
  BLEServer *pServer = BLEDevice::createServer();
  BLEService *pService = pServer->createService(SERVICE_UUID);
```

The setup routine function initiates serial communications and prints a message indicating that BLE operations have started. The BLEDevice::init() function initializes the BLE device name and sets up the GATT server and service using BLEDevice::createServer(), and pServer->createService() respectively. When starting a BLE profile, you can specify your own BLE device name. If no value is assigned, the default name of the device is ES32. Service is associated with a custom service UUID that was defined previously.

```
  BLECharacteristic *pCharacteristic =
    pService->createCharacteristic(CHARACTERISTIC_UUID,
BLECharacteristic::PROPERTY_READ | BLECharacteristic::PROPERTY_WRITE);
```

A GATT characteristic is created using createCharacteristic() function within the service pService, and its properties define set permission to allow both reading and writing.

```
  pCharacteristic->setValue("Hello World says Neil");
```

The initial value of the characteristic is set to a string. In this example, we have "Hello World says Neil", and we can assign any string value. The function setValue() only passes a constant string or byte data type.

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

After defining the GATT characteristic, you can now start the GATT service which allows the client device to discover the created custom GATT service and its characteristic content.

```
  // BLEAdvertising *pAdvertising = pServer->getAdvertising();  // this still is
working for backward compatibility
  BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
  pAdvertising->addServiceUUID(SERVICE_UUID);
  pAdvertising->setScanResponse(true);
  pAdvertising->setMinPreferred(0x06);  // functions that help with iPhone
connections issue
  pAdvertising->setMinPreferred(0x12);
  BLEDevice::startAdvertising();
```

Advertising configuration is essential for the device to be discoverable. The service UUID is added, and parameters are set to enhance compatibility. This enables the client device to scan the ESP32 BLE server and establish a connection between them.

```
  Serial.println("Characteristic defined! Now you can read it in your phone!");
}
```

A message is printed to the serial monitor indicating that the BLE characteristic has been defined and is ready to be accessed.

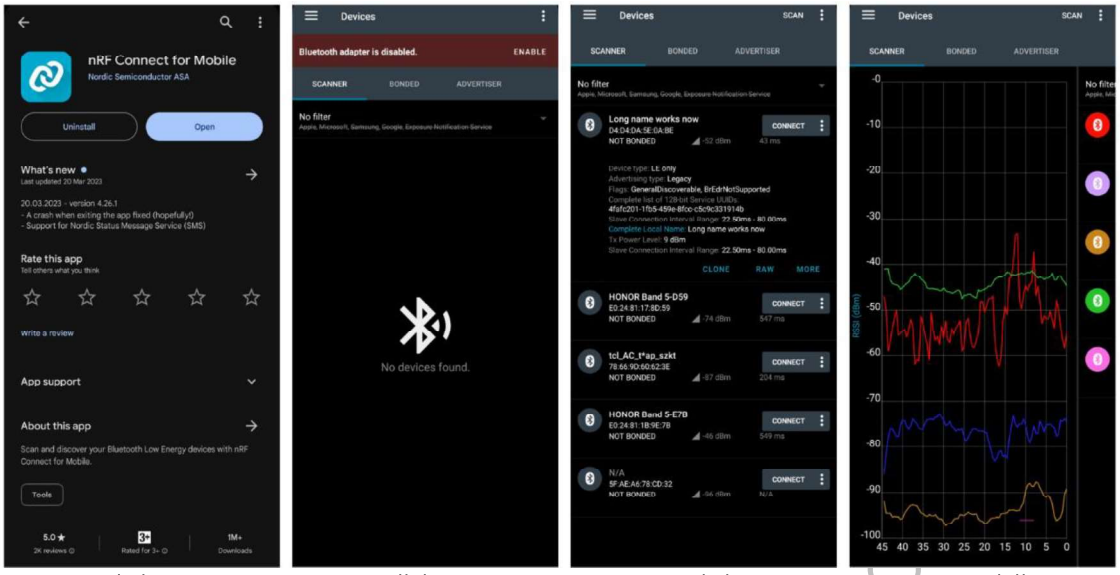
```
void loop() {
  // put your main code here, to run repeatedly:
  delay(2000);
}
```

The loop function contains a delay, but in this example, the main functionality is implemented during the setup phase, and there is no continuous operation in the loop. This example demonstrates the basic steps to set up a BLE server on the ESP32, defining a service with a readable and writable characteristic. It advertises the service, making it accessible to BLE clients, such as smartphone applications to serve as a foundation for more complex BLE applications on the ESP32 platform.

## 4.4.2 Getting Started with nRF Connect for Mobile

To get started with BLE on ESP32, we need a mobile application that can scan BLE devices and allow us to communicate with them as shown in **Figure 10**. Nordic Semiconductor's nRF Connect for Mobile is a powerful and versatile mobile application designed to interact with BLE devices. This app provides comprehensive features for BLE development, including device discovery, connection establishment, and communication with GATT characteristics. Here are instructions for getting started with nRF Connect for Mobile:

1. **Download and Installation**: Start by downloading the NRF Connect for Mobile app from the Google Play store as shown in **Figure 10 (a)**. The application is free of charge and compatible with Android devices. After installation, open the application to access its functionalities.
2. **Launch nRF Connect**: After the installation is finished, locate, and open the nRF Connect application on your device. The app might ask for specific permissions the first time it runs. **Figure 10 (b)** shows the permit Bluetooth access and any other features that the app needs to run properly.




(a) (b) (c) (d)

**Figure 10** *nRF Connect for Mobile (a) Google Play Store download page, (b) launch nRF Connect, (c) scan for BLE devices, and (d) RSSI graph of scanned BLE devices.*

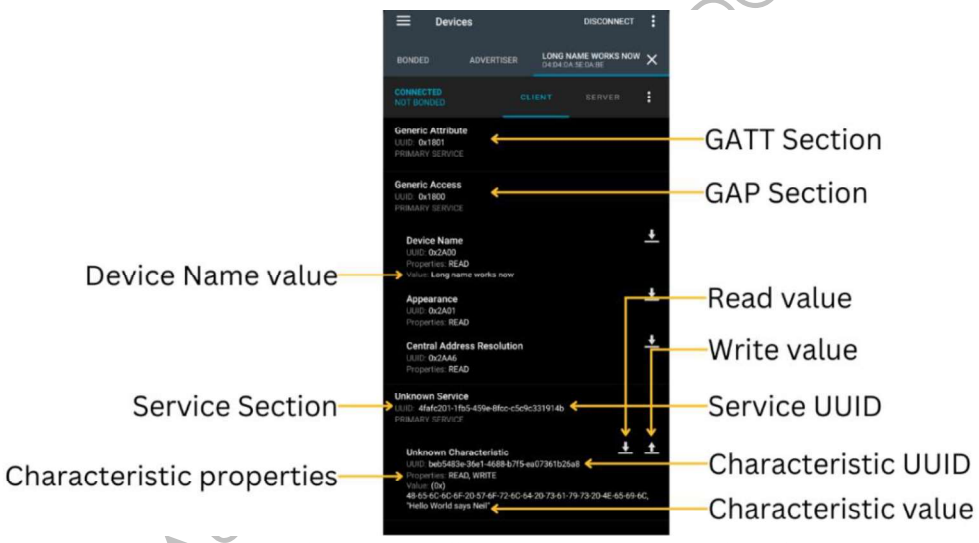



**Figure 11** *Device BLE profile nRF Connect user interface.*

3. **Device Discovery and Connection**: When starting nRF Connect, navigate to the Scanner tab to start device discovery. You should see a "SCAN" button on the app's main screen. Tap it to start a scan for nearby BLE devices. Make sure your ESP32 BLE server is turned on and actively advertising. As the scan progresses, a list of available BLE devices should appear as shown in **Figure 10 (c)**. Search for the device name or identifier associated with your ESP32. It may appear as the name you assigned in your Arduino code or as a default ESP32 BLE device name. From the scan results, select the associated ESP32 BLE device name to connect. After a successful connection, nRF Connect provides access to the GATT properties of the ESP32 server. These characteristics may include information about the device and all readable and writable data.

4. **Explore GATT Characteristics**: After tapping on the ESP32 device, the app will display a list of services and characteristics associated with the device's BLE profile as shown in **Figure 11**. Browse through the services and characteristics to find the one defined in your ESP32 Arduino code. This tab displays a hierarchical view of services and characteristics, allowing users to read and write data as defined by the server's

implementation. Depending on the properties set in your ESP32 code, you may be able to read, write, or enable notifications for specific characteristics.

### 4.4.3 BLE Profile Mimicking Smart Blood Pressure Monitor

The ESP32 microcontroller can imitate the commercial Smart Blood Pressure Monitor over BLE, as shown in **Figure 6**, when programmed with the Arduino IDE. The aim is to implement the standard blood pressure profile defined by the Bluetooth SIG, a set of unique assigned numbers to define various profiles within the Bluetooth specification. By using the Arduino BLE library on ESP32 as demonstrated in **Listing 2**, you can create a GATT server, specify the required services and properties, and advertise the device as a certified blood pressure cuff to a central server.

**Listing 2** *ESP32 Imitate commercial smart blood pressure monitor*

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

// Define BLE device name
#define BLE_DEVICE_NAME "SMART BLOOD PRESSURE MONITOR"

// Assigned official 16-bit GATT Profile UUID
#define BLOOD_PRESSURE_SERVICE_UUID BLEUUID((uint16_t)0x1810)
#define BLOOD_PRESSURE_MEAS_CHARACTERISTIC_UUID BLEUUID((uint16_t)0x2A35)
#define CUFF_PRESSURE_MEAS_CHARACTERISTIC_UUID BLEUUID((uint16_t)0x2A36)

// Declare BLE GATT profile object as global variable
BLEServer *pServer;
BLEService *pBloodPressureService;
BLECharacteristic *pBloodPressureMeasCharacteristic;
BLECharacteristic *pCuffPressureMeasCharacteristic;
```

In this example, the SMART BLOOD PRESSURE MONITOR was explicitly defined as a device name. Global variables GATT profile is declared using the official GATT UUID Bluetooth SIG for blood pressure service (i.e. 0x1810), blood pressure measure (i.e. 0x2A35) and intermediate pressure service (i.e. 0x2A36).

```
// Connection status flag
bool deviceConnected = false;
bool oldDeviceConnected = false;

class MyServerCallbacks : public BLEServerCallbacks {
  void onConnect(BLEServer *pServer) {
    deviceConnected = true;
  };

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

The connection status flag deviceConnected tracks the connection to the server and ensures that data is sent only when the device is paired. The MyServerCallbacks() function

returns the value of the deviceConnected flag depending on the server connection event. If connected, the DeviceConnected value is set to true, otherwise the value is false.

```
void setup() {
  // Initiate BLE device and creat user custom name
  BLEDevice::init(BLE_DEVICE_NAME);

  // Create BLE server object
  pServer = BLEDevice::createServer();
  pServer->setCallbacks(new MyServerCallbacks());

  // Add blood pressure service to server
  pBloodPressureService = pServer->createService(BLOOD_PRESSURE_SERVICE_UUID);

  // Add characteristics to blood pressure service
  pBloodPressureMeasCharacteristic = pBloodPressureService->createCharacteristic(
    BLOOD_PRESSURE_MEAS_CHARACTERISTIC_UUID,
    BLECharacteristic::PROPERTY_READ | BLECharacteristic::PROPERTY_NOTIFY |
BLECharacteristic::PROPERTY_INDICATE);

  pCuffPressureMeasCharacteristic = pBloodPressureService->createCharacteristic(
    CUFF_PRESSURE_MEAS_CHARACTERISTIC_UUID,
    BLECharacteristic::PROPERTY_READ | BLECharacteristic::PROPERTY_NOTIFY |
BLECharacteristic::PROPERTY_INDICATE);

  // Add BLE descriptor
  pBloodPressureMeasCharacteristic->addDescriptor(new BLE2902());
  pCuffPressureMeasCharacteristic->addDescriptor(new BLE2902());

  // Start GATT service to discover by Client device
  pBloodPressureService->start();

  // Start Advertising to scan by Client device
  BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
  pAdvertising->addServiceUUID(BLOOD_PRESSURE_SERVICE_UUID);
  pAdvertising->setScanResponse(false);
  pAdvertising->setMinPreferred(0x0);
  BLEDevice::startAdvertising();
}
```

Setup routine initializes the BLE environment and sets up ESP32 as the server. MyServerCallbacks() is connected to the BLE server profile and handles callbacks events using the setCallbacks() function. The blood pressure service pBloodPressureService and its two characteristics, blood pressure pBloodPressureMeasure and cuff pressure pCuffPressureMeasure, are defined with READ, NOTIFY and INDICATE properties to allow clients to read the data and get the updates. Descriptor BLE2902 is added to allow notifications as required by the GATT notify functionality. The service runs with pBloodPressureService->start(), and the advertisement runs with startAdvertising() and broadcasts the service UUID so that the client devices can find and connect to it.

```
void loop() {
```

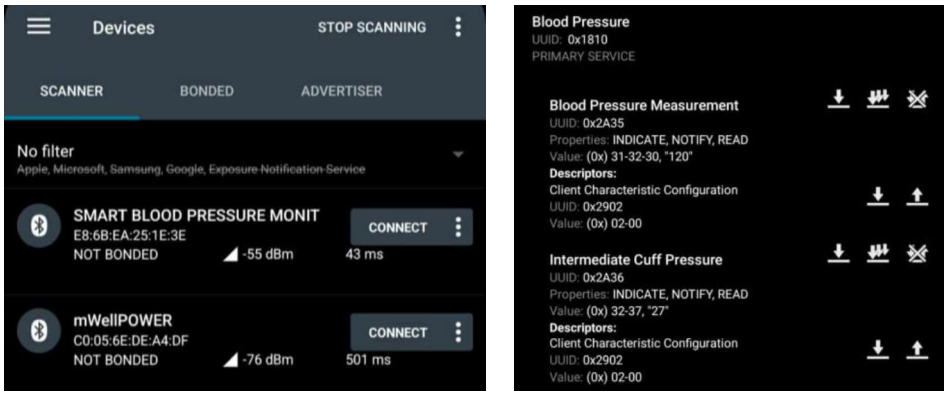
```
  // notify changed value
  if (deviceConnected) {
    // Assigned random values for systolic blood pressure measurement
    int SBP = random(120, 130);

    // Assigned random values for endotracheal tube cuff pressure measurement
    int ETCP = random(20, 30);

    // Update characteristic value
    pBloodPressureMeasCharacteristic->setValue(String(SBP).c_str());
    pBloodPressureMeasCharacteristic->notify();

    pCuffPressureMeasCharacteristic->setValue(String(ETCP).c_str());
    pCuffPressureMeasCharacteristic->notify();
    delay(500);
  }
  // disconnecting
  if (!deviceConnected && oldDeviceConnected) {
    delay(500);                  // give the bluetooth stack the chance to get
things ready
    pServer->startAdvertising();  // restart advertising
    Serial.println("start advertising");
    oldDeviceConnected = deviceConnected;
  }
  // connecting
  if (deviceConnected && !oldDeviceConnected) {
    // do stuff here on connecting
    oldDeviceConnected = deviceConnected;
  }
}
```

The loop routine function handles the regular data transfer when the client is connected, if the deviceConnected flag value is true. When disconnecting the client, the server automatically resumes advertisement, ensuring that new connections are always available. When connected to the client device, it generates random values for the emulation of systolic blood pressure (SBP) in the range of 120-130 mmHg and endotracheal pressure (ETCP) in the range of 20-30 mmHg, which are representative of the actual measurements for demonstration purposes. These values are converted to strings by casting the generated integer value to the string data type and converting the string to a c-style string as a character field. This value is set as a characteristic value by the setValue() function, followed by the notify() function, which pushes updates to the client on the connected network every 500 milli-second. **Figure 12** shows that the use of official GATT profiles guarantees interoperability with different applications for smart devices and the notify function supports the provision of real-time data. Here you can see the descriptive name of the GATT profile in the Bluetooth SIG in the nRF Connect application, so that you can easily identify the device data it represents.




(a) (b)

**Figure 12** *Blood pressure monitor GATT profile (a) device name, and (b) official Bluetooth service and characteristic with notify function.*

## 4.5 Summary

This chapter provides a comprehensive and practical exploration of BLE technology using the ESP32 microcontroller, equipping you with the knowledge and skills to develop energy-efficient wireless sensor nodes for IoT applications. You gained a deep understanding of the BLE architecture, including its low power advantages, limitations such as limited data rates, and the GATT profile, which organizes data into services and features for wireless communication between devices. Through hands-on tutorials, you have learned how to configure ESP32 as a BLE server, implement a Bluetooth profile modeled after a smart blood pressure monitor, and interact with it using the nRF Connect for Mobile application as the interface. This practical experience guided you how to read, write and advertise sensor data.

This chapter also covers BLE network topologies such as broadcast and connection-based networks which allow for the assessment of their scalability and reliability in specific use cases. In real-world scenarios, these skills are translated into the development of energy efficient IoT solutions such as environmental monitoring systems, wearables and smart home automation, where the low energy consumption of the BLE and the versatility of the ESP32 allow reliable battery-powered devices. In conclusion, this chapter gives you the power to use BLE for innovative IoT applications and provides a solid basis for designing secure, efficient and scalable wireless sensor networks addressing the practical challenges of health care, industrial monitoring and the smart environment.

## 4.6 Mini Project 3: An Energy-Efficient Bluetooth Wireless Sensor and Data Logger

### 4.6.1 Introduction

Several advanced devices demonstrate how BLE can be integrated into applications for environmental sensing. Some notable examples are the Ela Innovation industrial temperature sensor, the ExoSense Py indoor environment sensor and the Efento moisture-temperature sensor as shown in **Figure 13**. These devices use BLE technology to effectively transfer important environmental data with a minimum of energy consumption.


Efento Exo Sense Py Ela Innovation

**Figure 13** *Commercial Bluetooth environmental sensors. Obtained from Efento[24], Exo Sense Py[25], and Ela Innovation[26].*

Efento is known for its moisture and temperature wireless data recorders, which are often used for environmental monitoring [24]. Applications for Efento products are found in a wide range of sectors such as logistics, health care and pharmaceuticals where accurate and reliable environmental monitoring is of utmost importance. Advanced sensors in the indoor environment such as temperature, humidity and air quality can all be monitored with EXO Sense PY [25]. It utilizes an ESP32 chipset for Bluetooth connectivity making it easy for users to connect to the sensor and access real-time data. Applications for the EXO Sense Py can be found in smart homes, offices, and industrial settings where it is crucial to maintain a comfortable and healthy indoor environment. ELA Innovation industrial temperature sensor is one of their many industrial sensors of high quality [26]. Due to its robust design, the industrial temperature sensor can be used in a variety of industrial environments where reliable and long-term temperature monitoring is essential.

These Bluetooth sensors are an ideal example of how sensing technologies and wireless communication can coexist to enable remote monitoring and control of the state of the environment. They are valuable tools in several industries due to their Bluetooth integration, which improves usability and accessibility.

Upon completing this project, you will be able to:

- Configure ESP32 as a BLE server and define your own GATT profile with multiple services and properties to represent different sensor data and status of your actuators.
- Set up a BLE bidirectional data exchange that supports read and write operations on the BLE properties, enabling the client device to retrieve sensor data and send control commands.
- Use the ESP32 to interface with an SD card module via serial peripheral interface (SPI) and record sensor data with time stamps to a file in a structured format such as CSV.
- Use a BLE scanner app such as nRF Connect for Mobile to communicate with ESP32, read sensor data in real time and write commands to the controller.

## 4.6.2 Components Needed

Hardware Components:

1. LILYGO TTGO LoRa32 OLED ESP32 Development Board
2. Digital Temp & Humidity Sensor (AM2302/DHT22)
3. Light Intensity Sensor (Photoresistor (LDR) GL5528)
4. OLED Display (SSD1306 128x64 pixels)
5. MicroSD Card / TF Card
6. Jumper Wires

7. Power Source (Battery or USB power)

Software Tools:

1. Arduino IDE
2. nRF Connect for mobile or similar BLE scanner mobile applications

## 4.6.3 System Architecture

In this project of building a BLE wireless sensor and controller for environment monitoring, the focus shifts to an enhanced and more energy-efficient wireless communication protocol compared to Bluetooth Classic, making it ideal for sensor nodes that require prolonged battery life. This activity equips you to create sensors capable of monitoring environmental parameters like temperature, humidity, and light while consuming minimal power with a wireless controller. This advancement opens possibilities for deploying sensors in remote locations or areas where frequent battery replacement is impractical.

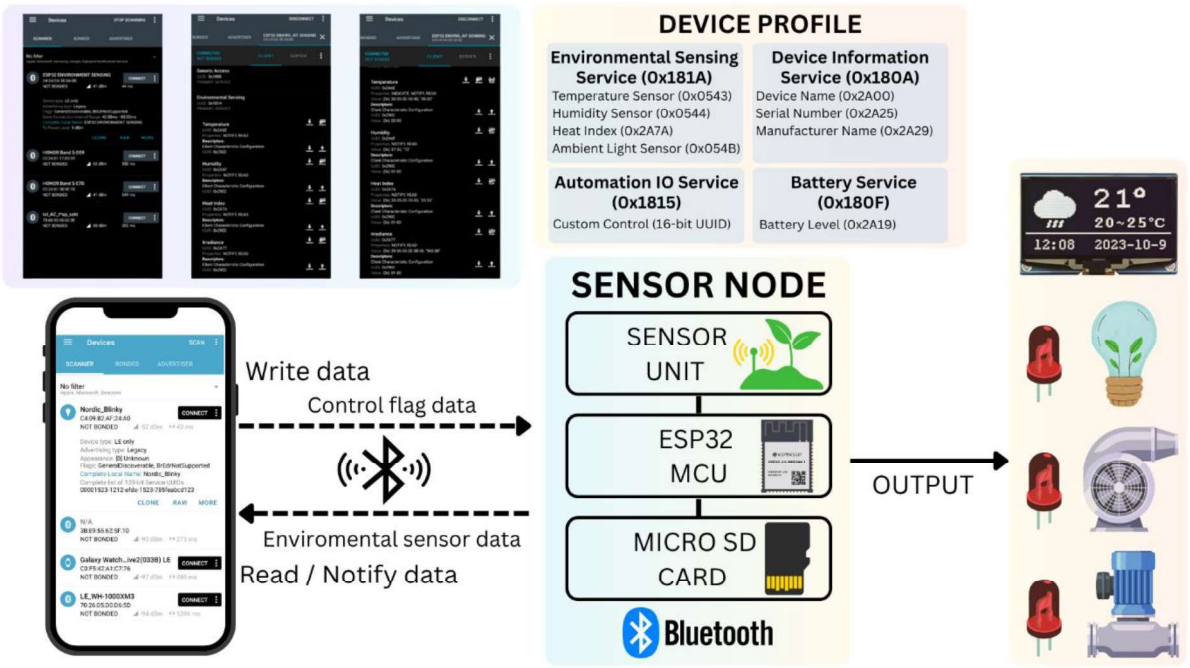



**Figure 14** *Mini Project 3: System Architecture of the wireless sensor node and data logger with controls using Bluetooth Smart.*

This architecture, as shown in **Figure 14**, builds on the basic sensors and actuators and increases energy efficiency. For real-time environmental data, environmental sensors such as digital temperature and humidity sensors (i.e., DHT22) and ambient light sensors (i.e., LDR) are directly linked to ESP32. The actuation is provided by a few LEDs serving as controllable outputs. The OLED screen provides real-time visual feedback on sensor data of the system. The core advancement in this mini-project lies in the transition to BLE for wireless communication, enabling energy-efficient, bidirectional data exchange with a mobile application, typically a BLE scanner app such as nRF connect. This mobile application acts as a remote user interface, enabling you to view sensor data and send commands to the ESP32 for control. The overall design emphasizes the creation of a comprehensive GATT profile to structure and organize the various sensor data and control status of the actuators for efficient and standardized BLE communication.

In addition, the project incorporates an SD card module, which turns the sensor node into a data logger that can store sensor data locally. In a real-world scenario of IoT, sensor data stored locally is used for long-term analysis and historical monitoring.

**Listing 3** *Storing sensor data on a microSD card*

```
// Libraries for SD card
#include “FS.h”
#include “SD.h”
#include “SPI.h”
```

The code initiates by including essential libraries such as file system FS.h, SD card functionality SD.h, and serial peripheral interface (SPI) SPI.h communication from the Arduino core. Together these libraries provide the necessary abstractions to interact with SD card modules over the SPI bus.

```
// Defined SD card SPI pins
#define SD_CS 13
#define SD_MOSI 15
#define SD_MISO 2
#define SD_CLK 14

// Define LoRa SPI chip select pin
#define LoRa_CS 18
```

The following is the definition of specific ESP32 GPIO pins to the SPI communication lines for the SD card, pins SD_CS, SD_MOSI, SD_MISO, and SD_CLK. A separate chip select pin LoRa_CS is also defined, indicating another SPI-based peripheral such as a LoRa module on the same bus. The SD card and LoRa modules are integrated on board the TTGO LILYGO LoRa32. The specified GPIO pins are therefore used internally by these interfaces.

```
// Timer variables
unsigned long lastTime = 0;
unsigned long timerDelay = 5000;

// Variables to hold sensor readings
unsigned long epochTime;
float tempC;
float hum;
float hi;
float light;
String sensorData;

// Initialize SD card
void initSDCard() {
  // Initialize SPI chip select pins
  pinMode(SD_CS, OUTPUT);
  pinMode(LoRa_CS, OUTPUT);

  // Select (LOW) SD card module peripheral
  // Deselect (HIGH) LoRa module
  digitalWrite(LoRa_CS, HIGH);
```

```
  digitalWrite(SD_CS, LOW);

  // Initialize SD card module
  SPI.begin(SD_CLK, SD_MISO, SD_MOSI, SD_CS);
  if (!SD.begin()) {
    Serial.println("Card Mount Failed");
    return;
  }

  // Detect card type
  uint8_t cardType = SD.cardType();
  if (cardType == CARD_NONE) {
    Serial.println("No SD card attached");
    return;
  }

  Serial.print("SD Card Type: ");
  if (cardType == CARD_MMC) {
    Serial.println("MMC");
  } else if (cardType == CARD_SD) {
    Serial.println("SDSC");
  } else if (cardType == CARD_SDHC) {
    Serial.println("SDHC");
  } else {
    Serial.println("UNKNOWN");
  }

  // Print card memory status
  uint64_t cardSize = SD.cardSize() / (1024 * 1024);
  Serial.printf("SD Card Size: %lluMB\n", cardSize);
  Serial.printf("Total space: %lluMB\n", SD.totalBytes() / (1024 * 1024));
  Serial.printf("Used space: %lluMB\n", SD.usedBytes() / (1024 * 1024));
}
```

The initSDCard() function encapsulates the entire SD card initialization process. It starts by configuring the selection pins on the SD card with the LoRa module. The critical step is to set the LoRa chip select pin HIGH and SD card chip select pin LOW. This will ensure that the LoRa module is deselected, and the SD card module is selected on the shared SPI bus, thus avoiding communications conflicts. The SPI.begin() function then initiates the SPI bus using the specified pins, SD_CS, SD_MOSI, SD_MISO, SD_CLK pins of the chip. Then, SD.begin() tries to mount the SD card file system. An error handling is implemented to control whether a card is not connected, or an SD card is not detected, and to print informative messages to the Arduino serial monitor. It also detects and prints the type of SD card using SD.cardType() function and its total and used storage capacity via SD.cardSize(), SD.totalBytes(), and SD.usedBytes() functions and provides comprehensive state of the storage.

```
// Write to the SD card
void writeFile(fs::FS &fs, const char *path, const char *message) {
  Serial.printf("Writing file: %s\n", path);
```

```
  File file = fs.open(path, FILE_WRITE);
  if (!file) {
    Serial.println("Failed to open file for writing");
    return;
  }
  if (file.print(message)) {
    Serial.println("File written");
  } else {
    Serial.println("Write failed");
  }
  file.close();
}

// Append data to the SD card
void appendFile(fs::FS &fs, const char *path, const char *message) {
  Serial.printf("Appending to file: %s\n", path);

  File file = fs.open(path, FILE_APPEND);
  if (!file) {
    Serial.println("Failed to open file for appending");
    return;
  }
  if (file.print(message)) {
    Serial.println("Message appended");
  } else {
    Serial.println("Append failed");
  }
  file.close();
}
```

The writeFile() and appendFile() functions manage the storage of data on an SD card and provide different operations for initial file creation and for adding new data. The writeFile() function creates or overwrites a file suitable for initializing data tags, while appendFile() function adds new data without changing the existing content, which is ideal for logging sequential data.

```
void setup() {
  Serial.begin(115200);

  // Initialized SD card
  initSDCard();

  // If the data.txt file doesn't exist
  // Create a file on the SD card and write the data labels
  File file = SD.open("/data.txt");
  if (!file) {
    Serial.println("File doesn't exist");
    Serial.println("Creating file...");
    writeFile(SD, "/data.txt", "Epoch Time, Temperature, Humidity, Heat Index,
Light \r\n");
```

```
  } else {
    Serial.println("File already exists");
  }
  file.close();
}
```

The setup routine function initialize the SD card for data logging by calling InitSDCard() Then, using SD.open(), it checks for the presence of the file \"data.txt\", creates the file with writeFile() if missing, and writes the initial headers such as epoch time, temperature, humidity, heat index, and light to establish a CSV file format. If the file already exists, it prints that the file already exists on the Arduino serial monitor.

```
void loop() {
  if ((millis() - lastTime) > timerDelay) {
    lastTime = millis();

    epochTime = millis() / 1000;
    tempC = random(2000.00, 5000.00) / 100.00;
    hum = random(8000.00, 10000.00) / 100.00;
    hi = random(2700, 3200) / 100.00;
    light = random(32000, 50000) / 100.00;

    sensorData = String(epochTime) + "," + String(tempC) + "," + String(hum) + ","
+ String(hi) + "," + String(light) + "\r\n";
    Serial.print("Saving data: ");
    Serial.println(sensorData);

    // Append the data to file
    appendFile(SD, "/data.txt", sensorData.c_str());
  }
}
```

The loop routine function implements a regular data logging operation with a non-blocking timer, which checks whether the last update has elapsed for 5000 ms by calling millis(). Upon triggering, the sensorData is updated by random values, for example, tempC between 20 and 50 degrees Celsius, and a CSV-formatted sensorData string is created with epoch time, temperature, humidity, heat index, and luminous values. The data is printed on the Arduino serial monitor and then attached to the file data.txt by using the appendFile() function. After reading the stored data from the microSD card as described below.

```
Epoch Time, Temperature, Humidity, Heat Index, Light
5,45.32,98.36,28.04,428.30
10,42.79,87.60,31.29,358.27
15,24.51,99.23,29.67,442.17
20,45.57,98.14,28.72,404.96
25,40.70,98.64,30.63,447.53
30,25.25,94.98,31.25,361.92
```

This Arduino code configures ESP32 to record simulated sensor data to the SD card every 5 seconds, initializes the card, creates CSV file, and appends sensor data within a text file.

### 4.6.4 Project Plan

You must complete the following tasks to meet the project objectives.

- Enhance the environmental sensing service to extend the GATT profile to include device information, battery service, and automation IO. Use official GATT UUIDs to ensure compatibility and compliance with the Bluetooth standard.
- Implement SD card data logging system from real sensor data.
- Explore the Automation IO and create your own attributes in this service. Define the characteristics of the control signals for managing the status of the output devices, as represented by the LED. This includes developing your own descriptor profiles to express your unique attributes and descriptions.
- Implement the logic for handling control signals in the automation IO custom properties. Ensure that the control signals the activation and deactivation of the output devices represented by the LED.
- Use the BLE scanner mobile application or similar to connect to the ESP32 BLE server, read sensor data in real time from the defined GATT characteristics and send control commands to the ESP32 actuators.

### 4.6.5 Guidelines and Tips

- GATT is the backbone of BLE communication. Each BLE device relies on GATT profiles to define the structure of services and characteristics. Understanding the GATT hierarchy is crucial for effective BLE development.
- The use of official 16-bit GATT UUIDs is a standard practice in BLE development. When creating services and characteristics, refer to the official UUIDs to maintain consistency. If official services and characteristics are not applicable, a custom UUID generated randomly can be used to create custom services and characteristics. A descriptor profile can be used as metadata for custom services and characteristics. It provides human readable information, such as a field description, user-friendly name, valid range of values, or specify the unit of measurement making it easier for developers to comprehend the purpose of specific services and characteristics.
- Device Information and Battery Services are essential components of a comprehensive BLE device. These services provide valuable insights into the device's identity and status such as device name, model number, firmware version, hardware version, manufacturer name, and battery level.
- Custom descriptors add depth to BLE characteristics by providing additional information about their properties. Create custom descriptor profiles to articulate the unique attributes of the characteristics within service.

## References


[1] M. Afaneh, "Bluetooth Low Energy (BLE): A Complete Guide," Novel Bits. Accessed: Dec. 22, 2023. [Online]. Available: https://novelbits.io/bluetooth-low-energy-ble-complete-guide/

[2] Bluetooth® Technology Website, "Bluetooth Technology Overview," Bluetooth SIG, Inc. Accessed: Nov. 28, 2023. [Online]. Available: https://www.bluetooth.com/learn-about-bluetooth/tech-overview/

[3] WeMakeIoT, "Bluetooth Low Energy (BLE)," SA WeMakeIoT. Accessed: Dec. 22, 2023. [Online]. Available: https://www.wemakeiot.com/iot-technology-solutions/bluetooth-low-energy-ble/

[4] M. Afaneh, "Bluetooth 5 Speed: How to Achieve Maximum Throughput," Novel Bits. Accessed: Dec. 22, 2023. [Online]. Available: https://novelbits.io/bluetooth-5-speed-maximum-throughput/

[5] M. Li, "What is A BLE Gateway and How Does BLE Gateway Work?," Moko Blue. Accessed: Dec. 22, 2023. [Online]. Available: https://www.mokoblue.com/what-is-a-ble-gateway/

[6] NiceRF, "Advantages and Typical Applications of Bluetooth Low Energy," NiceRF Wireless Technology Co., Ltd. Accessed: Dec. 26, 2023. [Online]. Available: https://www.nicerf.com/item/bluetooth-low-energy

[7] RIOH, "The Advantages and Disadvantages of Using Bluetooth in the Geolocation of Medical Equipment," RIOH Inc. Accessed: Dec. 27, 2023. [Online]. Available: https://rioh.io/en/the-advantages-and-disadvantages-of-using-bluetooth-in-the-geolocation-of-medical-equipment/

[8] Nordic Developer Academy, "ATT & GATT: Data Representation and Exchange," Nordic Semiconductor. Accessed: Dec. 28, 2023. [Online]. Available: https://academy.nordicsemi.com/courses/bluetooth-low-energy-fundamentals/lessons/lesson-1-bluetooth-low-energy-introduction/topic/att-gatt-data-representation-and-exchange/

[9] M. Bhargava, *IoT Projects with Bluetooth Low Energy*. 2017. Accessed: Dec. 22, 2023. [Online]. Available: https://www.oreilly.com/library/view/iot-projects-with/9781788399449/?_gl=1*1qm5yzg*_ga*MTgwODMzNjIwMi4xNzAzMjU3Njg3*_ga_092EL089CH*MTcwMzI1NzY4Ni4xLjEuMTcwMzI1ODA0MC4yMC4wLjA.

[10] K. Townsend, C. Cufí, Akiba, and R. Davidson, "Getting Started with Bluetooth Low Energy," O'Reilly Media, Inc. Accessed: Dec. 22, 2023. [Online]. Available: https://www.oreilly.com/library/view/getting-started-with/9781491900550/

[11] F. Waterlot, "What is Bluetooth Low Energy?," Ela Innovation. Accessed: Dec. 22, 2023. [Online]. Available: https://elainnovation.com/what-is-bluetooth-low-energy/

[12] Nordic Developer Academy, "GAP: Device Roles and Topologies," Nordic Semiconductor. Accessed: Dec. 28, 2023. [Online]. Available: https://academy.nordicsemi.com/courses/bluetooth-low-energy-fundamentals/lessons/lesson-1-bluetooth-low-energy-introduction/topic/gap-device-roles-and-topologies/

[13] Nordic Developer Academy, "Connection Process," Nordic Semiconductor. Accessed: Dec. 28, 2023. [Online]. Available: https://academy.nordicsemi.com/courses/bluetooth-low-energy-fundamentals/lessons/lesson-3-bluetooth-le-connections/topic/connection-process/

[14] M. Afaneh, "Mastering BLE: Centrals vs. Peripherals," Novel Bits. Accessed: Jan. 03, 2024. [Online]. Available: https://novelbits.io/ble-peripherals-centrals-guide/

[15] Bluetooth® Technology Website, "BLE Specifications Assigned Numbers," Bluetooth SIG, Inc. Accessed: Dec. 23, 2023. [Online]. Available: https://www.bluetooth.com/specifications/assigned-numbers/

[16] Bluetooth® Technology Website, "Topology Options," Bluetooth SIG, Inc. Accessed: Dec. 24, 2023. [Online]. Available: https://www.bluetooth.com/learn-about-bluetooth/topology-options/

[17] N. Newman, "Apple iBeacon Technology Briefing," *Journal of Direct, Data and Digital Marketing Practice*, vol. 15, no. 3, pp. 222–225, Jan. 2014, doi: 10.1057/DDDMP.2014.7/METRICS.

[18] Apple Developer, "iBeacon," Apple Inc. Accessed: Dec. 24, 2023. [Online]. Available: https://developer.apple.com/ibeacon/

[19] S. Zoria, "The Ultimate Guide to iBeacon Technology," TopSoftwareCompanies. Accessed: Dec. 28, 2023. [Online]. Available: https://topsoftwarecompanies.co/technology/the-ultimate-guide-to-ibeacon-technology

[20] M. Patel, "Bluetooth Low Energy (BLE) - The Future of Retail Technologies," e Infochips Arrow Electronics. Accessed: Dec. 26, 2023. [Online]. Available: https://www.einfochips.com/blog/bluetooth-low-energy-ble-the-future-of-retail-technologies/

[21] S. Gupta and S. Vojjala, "Designing with Bluetooth Mesh: Device Requirements," Embedded by AspenCore. Accessed: Dec. 24, 2023. [Online]. Available: https://www.embedded.com/designing-with-bluetooth-mesh-device-requirements/

[22] R. Huntley, "Bluetooth® Mesh & Home Automation," Mouser Electronics, Inc. Accessed: Dec. 26, 2023. [Online]. Available: https://www.mouser.com/blog/bluetooth-mesh-home-automation

[23] J. Marcel, "The Importance of Interoperability in the Smart Home," Bluetooth SIG, Inc. Accessed: Dec. 26, 2023. [Online]. Available: https://www.bluetooth.com/blog/the-importance-of-interoperability-in-the-smart-home/

[24] Efento, "Wide Range of Wireless Sensors and an IoT Platform," GetEfento. Accessed: Jan. 03, 2024. [Online]. Available: https://getefento.com/

[25] Sfera Labs, "EXO Edge Computing Environmental Sensors," Sfera Labs S.r.l. Accessed: Jan. 03, 2024. [Online]. Available: https://sferalabs.cc/exo/

[26] ELA Innovation, "Industrial Bluetooth Beacons and Sensors." Accessed: Jan. 03, 2024. [Online]. Available: https://elainnovation.com/en/