Chapter 2 Building a Wireless Sensor Using Bluetooth Classic with ESP32

Wireless connectivity is the cornerstone of innovation in the rapidly changing IoT. As we begin this new chapter, we will focus on integrating Bluetooth technology with sensor nodes, a ubiquitous device in wireless communications. Using the ESP32 microcontroller as a guide, this chapter explores the integration of Bluetooth wireless technology and its role in the context of IoT applications. This represents a significant step towards creating an agricultural ecosystem that is more responsive and connected. Get ready to see how Bluetooth improves our sensor node by providing the ability to remotely control and monitor it through wireless data transmission, opening the door to more advanced and adaptable IoT applications in agriculture. Let's dive in and unlock the potential of Bluetooth connectivity to shape the future of smart agriculture.

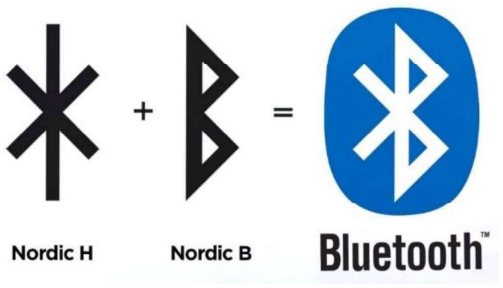
In this chapter, you will be able to

- Gain a comprehensive understanding of the distinctions between Bluetooth Classic and Bluetooth Low Energy (BLE) and recognize their respective applications in IoT.
- Dive into the basic architecture of the ESP32 Bluetooth capabilities and explore its functionalities.
- Gain the skills to establish wireless communication between an ESP32 device and a mobile application using the Bluetooth Classic protocol, unlocking the potential for remote control and monitoring.
- Learn the process of reading serial data from a device communicating over Bluetooth, a crucial skill for extracting and interpreting wirelessly transmitted data.
- Apply the gained knowledge to build a practical Bluetooth-enabled wireless controller and monitoring device using the ESP32 microcontroller, combining theory with practical implementation for a comprehensive learning experience.

2.1 What is Bluetooth Wireless Technology?

Bluetooth is a WPAN (Wireless Personal Area Network) protocol that connects two devices without the need for additional network infrastructure, such as a Wi-Fi router or access point [1], [2]. It was developed to free electronic devices from the data transmission cables, such as the RS-232 serial communication interface and UART (Universal Asynchronous Receiver/Transmitter). This enables wireless connections between headsets, microphones, stereos, computer peripherals, watches, and other devices over short distances ranging from 10 to 100 meters, depending on Bluetooth technology. It uses radio waves on frequencies between 2.4 GHz to 2.483 GHz industrial, scientific, and medical (ISM) spectrum band [3], and the technology development is overseen by a standard organization Bluetooth Special Interest Group (SIG), to meet the market needs.

(a)



(b)

Figure 1 *The origin of the name Bluetooth technology from (a) King Harald being baptized and (b) the Bluetooth logo derives from Harald's initials. Adopted from World History Encyclopedia and Bluetooth SIG [4], [5].*

Three leading companies in the market, Intel, Ericsson, and Nokia, got together in 1996 to design the standardization of this short-range radio technology, which would facilitate communication and cooperation between various products and industries. Bluetooth was proposed during this meeting by Jim Kardach of Intel. Kardach was subsequently quoted as stating, "With a short-range wireless link, we hope to unite the personal computer and cellular industries, just as King Harald Bluetooth was famous for uniting Denmark and Norway [1], [5]." According to the Bluetooth SIG [5], **Figure 1** depicts the Bluetooth logo emblem mixed with the Younger Futhark runic initials of King Harald Bluetooth, Hagall (ᚼ) and Bjarkan (ᛒ).

2.1.1 Bluetooth Standards

The two Bluetooth standards currently in use are Bluetooth Classic and Bluetooth Low Energy [3].

- **Bluetooth Classic**: It is a wireless technology for sending and receiving data such as text, images, videos, audio files, and more between smartphones and other electronic devices. Bluetooth Classic is designed to replace short-range peripheral cables and is not scalable for IoT applications such as sensor networks with hundreds of units. Bluetooth Classic is mainly used for data transfer between devices such as wireless speakers, headphones, and telephone connections.
- **Bluetooth Low Energy (BLE)**: It is an ultra-low-power variant of Bluetooth Classic designed for applications that do not require big data streams, such as low-power sensors and accessories with long-lasting batteries. It considerably extends the autonomy of connected objects by reducing the activity time. In this way, BLE is highly suitable for industrial uses and the best solution for low-bandwidth battery-operated applications such as smartwatches, fitness monitors, lighting remote controls, temperature and humidity monitors, Bluetooth medical devices, beacons for indoor location devices, and more.

Bluetooth Classic and BLE cannot communicate with one another because they employ different physical modulation/demodulation techniques. However, modern smartphones, on the other hand, have simultaneous communication capabilities with both Bluetooth Classic and BLE devices.

In conclusion, Bluetooth Classic is used for its ability to deliver massive amounts of streaming data, such as music or videos. Rather, BLE is used in industry to send a small amount of non-continuous data over time. In addition, low consumption ensures low acquisition costs and enables companies to implement IoT devices quickly and effectively [6].

2.1.2 Bluetooth Classic vs Bluetooth Low Energy

Although they share a common foundation, the Bluetooth Classic and Bluetooth Low-Energy communication protocols are different and do not serve the same audiences or meet the same needs. According to the Bluetooth SIG [3], Bluetooth Classic and Bluetooth Low Energy radios are designed to meet the specific needs of developers around the world, as shown in **Figure 2**, whether a product transfers data between a smartphone and speakers or medical devices or transmits messages between thousands of nodes in the target solution.

Figure 2 *Bluetooth Classic and Low Energy solution areas, device protocols, and data rate distinction. Adopted from Bluetooth SIG [3].*

Bluetooth technology allows developers to be creative. This adaptability is emphasized by the inclusion of two radio options: Bluetooth Classic and BLE. Bluetooth technology is not a one-for-all solution; rather, it provides developers with a comprehensive toolbox that includes complete, purpose-built solutions to meet the different and evolving needs of wireless connectivity across numerous applications.

The Bluetooth SIG [3] describes the application and solution areas in which BLE differs from one another. The main applications of Bluetooth Classic are data transfer and audio streaming, with an emphasis on point-to-point or one-to-one communication. It is perfect for applications such as efficient data sharing and high-quality audio streaming because it performs well in situations where higher data rates are essential.

On the other hand, BLE offers more applications such as device networking, data transfer, location services, and audio streaming. Point-to-point, broadcast, and mesh network configurations are among the various forms of device communication that BLE supports. Because of its adaptability, BLE is suitable for a variety of applications, including positioning services that facilitate the identification of neighboring devices and the calculation of distance and direction. While BLE is known for its adaptability and potential applications, BLE typically operates at lower data rates than Bluetooth Classic. Bluetooth SIG [3] summarizes the specifications explaining the differences between Bluetooth Classic and BLE can be found in **Table 1**.

TABLE 1
Bluetooth Standards Specifications [3]

Specifications	Bluetooth Classic	Bluetooth Low Energy
Frequency band	2.4GHz ISM Band	2.4GHz ISM Band
Channels	79 channels with 1 MHz spacing	40 channels with 2 MHz spacing
Channel Usage	Frequency-Hopping Spread Spectrum (FHSS)	Frequency-Hopping Spread Spectrum (FHSS)

Modulation	GFSK, π/4 DQPSK, 8DPSK	GFSK
Data rate	1 – 3 Mb/s	125 Kb/s – 2 Mb/s
Tx Power	≤ 100 mW (+20 dBm)	≤ 100 mW (+20 dBm)
Rx Sensitivity	≤-70 dBm	≤-70 dBm – ≤-82 dBm
Communication Topologies	Point-to-Point,	Point-to-Point, *Broadcast, and Mesh*
Positioning Features	None	Presence: Advertising Direction: Direction Finding (AoA/AoD) Distance: RSSI, Channel Sounding
Power consumption	High (~1 Watt)	Low (~0.001 – 0.5 Watt)

Both Bluetooth Classic and BLE operate within the 2.4GHz ISM Band, sharing a common frequency domain. Bluetooth performance may be affected by 2.4 GHz spectrum technologies, which include microwaves, Wi-Fi networking devices, and other similar devices that use the same ISM band [1]. The Bluetooth standard employs the Frequency Hopping Spread Spectrum (FHSS) technique, which divides the frequency band into multiple channels of 1-2 MHz wide [3], [6]. The data is subsequently transmitted through a variety of channels. The Bluetooth standard prevents interference with signals from other radio modules by changing channels up to 1600 times per second, which might be useful in an industrial environment.

Bluetooth performance depends on the physical location of the device and its environment. Bluetooth technology enables the pairing of devices that are approximately 10 to 100 meters apart [1]. According to Sony Electronics [7], maximum Bluetooth communication range is roughly 10 meters. However, this depends on some significant factors affecting the effective range of a reliable Bluetooth connection, as outlined by Bluetooth SIG [8], [9], including the following:

- **Radio Spectrum**: The lower the frequency, the greater the range. However, the lower the frequency, the lower the data rate that can be supported. Therefore, selecting a radio spectrum requires compromises between range and data rate. Bluetooth technology operates in the 2.4 GHz ISM spectrum band (2400 to 2483.5 MHz), enabling a good balance between range and throughput.
- **Physical Layer (PHY)**: The modulation scheme and other techniques used to send data over a specific radio frequency (RF) band are defined by wireless technology. Bluetooth technology offers a variety of PHY options, each with unique characteristics that affect effective range and data rates.
- **Receiver Sensitivity**: It is the lowest power level at which the receiver can continue to demodulate data, detect radio signals, and establish a connection. According to Bluetooth specifications, depending on the PHY being used, devices must be able to achieve a minimum receiver sensitivity of -70 dBm to -82 dBm. Nevertheless, receiver sensitivity levels achieved by Bluetooth implementations are usually much higher. For example, the scanning device can still detect advertising packets with a signal strength of about -100 dBm in average implementations of the BLE Coded PHY at 125 Kbps.
- **Transmit Power**: Range and power consumption are traded off in the design. The effective range of the signal being audible at greater distances increases with transmitting power. On the other hand, the device will use more power if the transmit

power is increased. Transmission powers supported by Bluetooth technology range from -20 dBm (0.01 mW) to +20 dBm (100 mW).

- **Antenna Gain**: Electric energy from the transmitter is transformed into electromagnetic energy, or radio waves, by the antenna, and the receiver receives the opposite conversion. In both cases, the transmitting and receiving antennas have an effective antenna gain. There are numerous antenna options available to designers of Bluetooth technologies. There are numerous antenna options available to designers of Bluetooth technologies. The typical antenna gain of Bluetooth devices is between -10 dBi and +10 dBi.
- **Path Loss**: What happens when a radio wave travels through the atmosphere is a decrease in signal strength. The path environment influences path attenuation, also known as path loss, which happens naturally over distance. Obstructions between the transmitter and the receiver might weaken the signal. A variety of obstacles composed of glass, wood, metal, or concrete, such as windows, doors, and towers of metal that reflect and disperse radio waves, can act as attenuators. Although objects can block radio waves, the type and density of the obstacle affect attenuation and effective path loss.

These variables interact to determine Bluetooth's effective range, which has been thoughtfully planned and standardized to provide dependable and consistent connectivity in a variety of situations. The design principles of Bluetooth technology, whether it is Classic or BLE, depend on the physical layer, radio spectrum, receiver sensitivity, transmit power, antenna gain considerations, and the challenges presented by path loss.

TABLE 2
Bluetooth Applications Based on Standards and Network Topology [10]

Bluetooth	**Classic**		**Low Energy**	
Communication topology	Point-to-point	Point-to-point	Broadcast	Mesh
Applications	Audio streaming Wireless headphones Wireless speakers Data transmission	Data transmission Sports & fitness Medical & health Peripherals & accessories	Location service Beacon services Indoor navigation Tracking	Device network Control system Monitoring system Automation system

The main applications of Bluetooth technology are wireless peripheral connections to laptops, desktop computers, and mobile phones. Bluetooth technology is also commonly used in gaming controllers to provide wireless connectivity. Bluetooth connectivity is essential for medical devices such as blood pressure monitors, oximeters, and glucose meters [1]. Bluetooth technology is also used by modern car infotainment systems to stream music or directions from the driver's phone. **Table 2** provides an overview of Bluetooth applications categorized by technology and network connectivity.

To sum up, Bluetooth Classic is excellent for point-to-point communication; it is primarily used for audio streaming and various types of high-speed data transmission. Conversely, BLE

provides a more adaptable communication topology that supports mesh, broadcast, and point-to-point setups, allowing for a wider range of applications, such as automation systems, device networking, and location services.

2.2 Get Started with Bluetooth Classic Using ESP32

Bluetooth technology has long been a driving force behind groundbreaking initiatives in wireless communications and IoT innovation. Figure 3 shows how developers created innovative wireless control and monitoring projects by integrating an Arduino board with a Bluetooth module before the release of ESP32. This was the era that brought creativity and connectivity with a separate Arduino board for processing and a peripheral Bluetooth module for wireless communication. But the game-changer ESP32 offers developers a unified and effective solution, seamlessly combining Bluetooth Classic and Low Energy with a powerful microcontroller on a single chip.

(a) (b)

Figure 3 *Arduino project (a) voice-controlled LEDs using Android app and (b) HC-05 Bluetooth Module. Adopted from Circuit Digest and eBay [11], [12].*

Figure 3 shows an Arduino Uno connected to an HC-05 Bluetooth module to wirelessly turn the LEDs on and off [11]. The Bluetooth module communicates with Arduino using a serial communication protocol known as Universal Asynchronous Receiver-Transmitter (UART) [13]. The main purpose of a UART is to send and receive serial data and only uses two wires for communication. The Bluetooth module uses a Serial Port Profile (SPP) to simulate serial cable connections between two peer devices over radio frequency communication and provide services to the applications. Since the original purpose of Bluetooth was to replace serial communication interfaces such as RS-232 cables, any connected device can use SSP to send and receive data as a UART.

2.2.1 ESP32 Bluetooth Stack Architecture

The Bluetooth protocol stack consists of two components: the *controller stack* and the *host stack* [14]. As illustrated in **Figure 4**, the Bluetooth stack is a multi-layered architecture that enables Bluetooth functionality on the ESP32 chip. The controller stack used for hardware interface management and link management consists of PHY, Baseband, Link Controller, Link Manager, Device Manager, host controller interface (HCI), and other modules [15]. It can be accessed using APIs, and functions are provided as libraries. This stack communicates directly with Bluetooth low-level protocols and hardware.

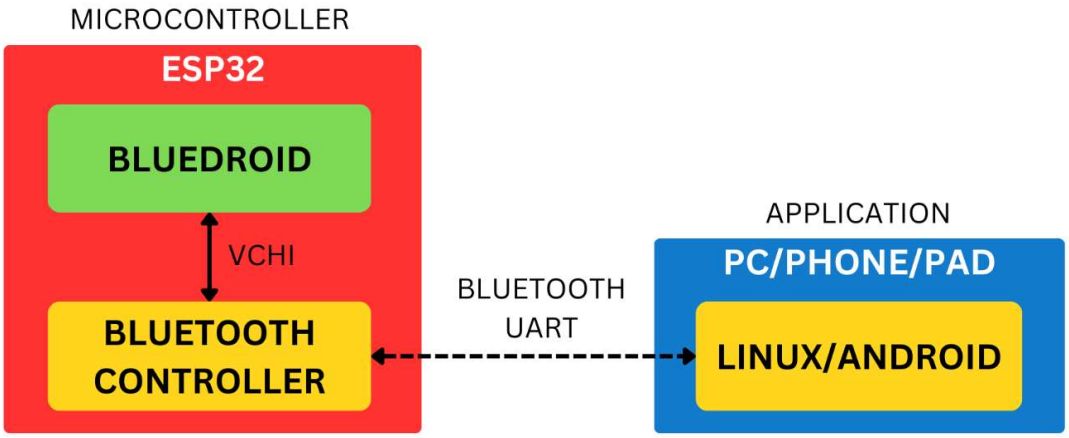


Figure 4 Basic architecture of the ESP32 Bluetooth host and controller.

The host stack serves as an interface to the application layer, allowing the latter to access the Bluetooth system. It is composed of L2CAP, SMP, SDP, ATT, GATT, GAP, and various profiles [14]. One Bluetooth host that is a modified variant of the native Android Bluetooth stack is called ESP-Bluedroid [15]. The Controller and Bluetooth Host may be implemented on the same device or a separate one. ESP32 provides support for both methods. Two scenarios are depicted in the above figure as follows:

1. The Bluetooth controller and the Bluedroid Bluetooth host exchange data via a virtual HCI interface (VHCI) implemented by software [14]. Since the Bluedroid and the controller are both implemented on the same ESP32 microcontroller, no additional devices are required to act as Bluetooth hosts in this scenario. With ESP-Bluedroid, the ESP32 can function as a standalone Bluetooth device. This is particularly useful when the ESP32 needs to communicate with other Bluetooth devices without being connected to a mobile device. It allows autonomous operation without the need for constant connection to a mobile host.

2. A separate device running the Bluetooth Host, such as tablets or Android phones running Bluedroid, is needed because the ESP32 microcontroller is only used as a Bluetooth controller [14]. This scenario facilitates the generation of creative mobile applications that interface with the ESP32 by using mobile devices as Bluetooth hosts. Creating a custom user interface that meets the needs of the application is especially beneficial. Configuration settings, control options, and real-time data visualization are all possible with mobile applications. Bluetooth UART allows wireless serial communication between the ESP32 and Bluetooth external devices [16]. Its service emulates the behavior of a physical UART system and allows a limited data packet to be exchanged in both directions at a time.

The needs and type of application will determine whether the Bluetooth host is on the ESP32 chip or an external device. Using external devices, like smartphones or tablets, as Bluetooth hosts is beneficial for projects that require a specialized, user-friendly interface, graphical data presentation, and unique mobile applications. This method enables the creation of unique mobile applications, improved user interactions, and cross-platform compatibility. Conversely, using the ESP32 as a stand-alone Bluetooth host could be more advantageous in situations where minimal dependencies, standalone functionality, and a small form factor are essential. The selection should be in line with the specific objectives and general system architecture of the relevant application. For every approach, there is a distinct and suitable solution.

2.2.2 Guide to ESP32 Bluetooth Classic with Arduino IDE

This section provides an overview of using Bluetooth Classic on the ESP32 platform with the well-known Arduino IDE. We start by introducing *BluetoothSerial*, a library that makes it easier to implement classic Bluetooth SPP communication over ESP32. Let's look at a practical example to demonstrate ESP32's Bluetooth Classic features. In this demonstration, the *Serial Bluetooth Terminal* application developed by Kai Morich is used to pair an ESP32-based Bluetooth device with a smartphone. **Figure 5** shows that the application can be downloaded from the Google Play Store.

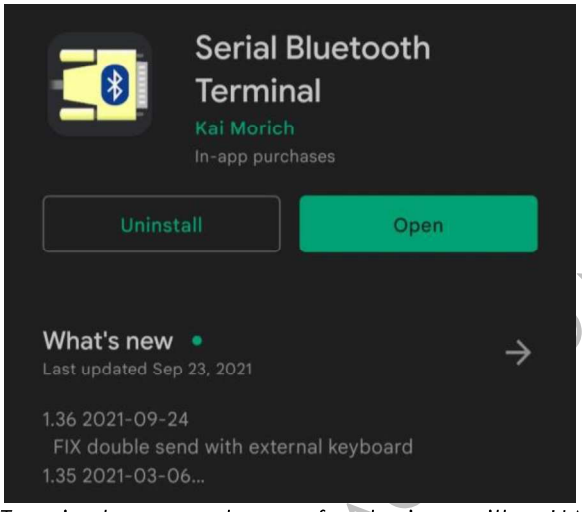
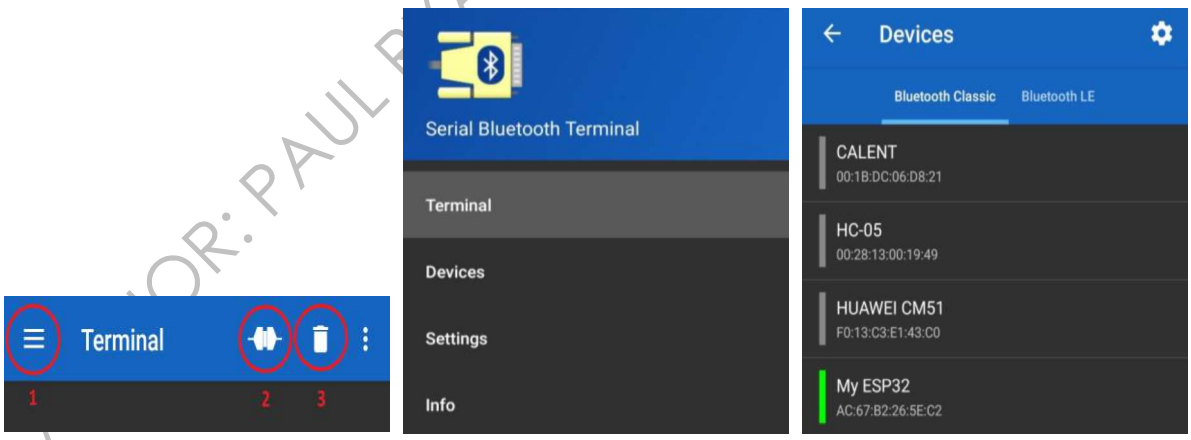


Figure 5 *Serial Bluetooth Terminal, a console app for devices with a UART interface connected with a Bluetooth serial converter to your Android device.*

The Bluetooth Serial Terminal app on a smartphone is used to walk you through the Arduino example of Bluetooth implementation. After installation, launch the Serial Bluetooth Terminal app. There are three primary function keys on the top bar, as shown in **Figure 6 (a)** 1. Menu tab, 2. Connect/disconnect Bluetooth device, and 3. Clear terminal display.



(a) (b) (c)

Figure 6 *The user interface of the Serial Bluetooth Terminal app and its configuration settings* (a) Tab bar options, (b) Settings options, and (c) Bluetooth devices.

Open the Arduino sketch example from "**File**" > "**Examples**" > "**BluetoothSerial**" > "**SerialToSerialBT**" to begin our exploration with a real-world example using the Arduino IDE, demonstrating the BluetoothSerial library for the first time. **Listing 1** shows an Arduino sketch highlighting the features of the ESP32's Bluetooth Classic integration. This example shows how

to configure an ESP32 to act as a Bluetooth slave to communicate with a mobile app through Bluetooth serial connection.

Listing 1 *Demonstration of Bluetooth Serial has the same functionalities as UART.*

```
#include "BluetoothSerial.h"
```

The code begins with the inclusion of the "BluetoothSerial.h" library, which is part of the Arduino core for ESP32, provided by Espressif Systems. This library provides the necessary functionalities for Bluetooth Classic communication, using the SPP to emulate a serial connection over a wireless channel.

```
String device_name = "ESP32-BT-Slave";
```

The device_name variable contains the name of the Bluetooth device ESP32-BT-Slave, which is defined as a Bluetooth-enabled device identifier. The assignment of a descriptive name for the device is important, as it facilitates easy identification and selection during the pairing process. This name is broadcast during Bluetooth scanning, which makes the ESP32 visible to the devices connected to it. In typical mobile-to-ESP32 communication scenarios, ESP32 operates as a slave device, passively waiting and accepting requests for connectivity from the mobile phone, which in general acts as the master.

```
// Check if Bluetooth is available
#if !defined(CONFIG_BT_ENABLED) || !defined(CONFIG_BLUEDROID_ENABLED)
#error Bluetooth is not enabled! Please run `make menuconfig` to and enable it
#endif
```

These lines check whether Bluetooth and Bluedroid are enabled. If they are not enabled, this line will throw a compile-time error and prompt you to run make menuconfig to enable them. The make menuconfig file is usually handled by the Arduino IDE by means of the board configuration. This check is essential to ensure that the hardware configuration supports Bluetooth Classic and distinguishes it from BLE, which uses a different library and profile.

```
// Check Serial Port Profile
#if !defined(CONFIG_BT_SPP_ENABLED)
#error Serial Port Profile for Bluetooth is not available or not enabled. It is
only available for the ESP32 chip.
#endif
```

Similarly, the lines above verify the enablement of the SPP within the Bluetooth configuration, which is specific to ESP32 chips. If SPP is not enabled, a compilation error is generated, explicitly stating that Bluetooth serial is not available.

```
BluetoothSerial SerialBT;
```

This line initializes an instance of an object BluetoothSerial class named SerialBT that is used to call the BluetoothSerial functions. It is responsible for managing the ESP32's Bluetooth module to facilitate wireless serial data exchange.

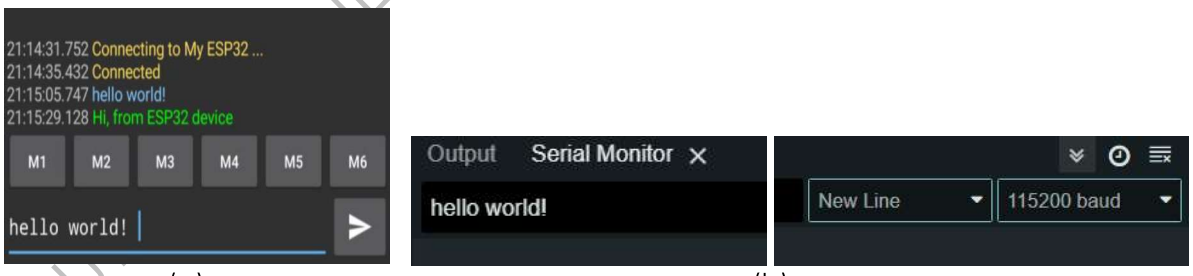
```
void setup() {
  Serial.begin(115200);
  SerialBT.begin(device_name);  //Bluetooth device name
  //SerialBT.deleteAllBondedDevices(); // Uncomment this to delete paired devices;
Must be called after begin
  Serial.printf("The device with name \"%s\" is started.\nNow you can pair it with
Bluetooth!\n", device_name.c_str());
}
```

SerialBT.begin() starts the ESP32 Bluetooth Classic stack and activates the integrated Bluetooth module. It has the input argument ESP32-BT-Slave, which is an identifier that other Bluetooth devices detect and display when searching for Bluetooth devices. A commented line //SerialBT.deleteAllBondedDevices(), if uncommented, would delete all previously paired devices. This is useful for resolving connection problems, as it removes all stored bonding information and forces a new connection process. The Serial.printf() function then prints a message to the serial monitor, informing you that the ESP32 Bluetooth device is started and ready for pairing, after setup.

```
void loop() {
  if (Serial.available()) {
    SerialBT.write(Serial.read());
  }
  if (SerialBT.available()) {
    Serial.write(SerialBT.read());
  }
  delay(20);
}
```

The loop routine manages the flow of data originating from the host computer via Arduino serial monitor and directs it to the Bluetooth-connected device, and vice versa. It uses conditional statements to check for data availability. The statement if (Serial.available()) reads data from the serial port and writes it to Bluetooth using SerialBT.write(Serial.read()), while the statement if (SerialBT.available()) reads data from Bluetooth and writes it to the serial port using Serial.write(SerialBT.read()). This two-way communication allows data entered in the serial monitor to be sent to and from a paired mobile application, with the output being visible on both interfaces.

First, turn on your phone's Bluetooth and pair it with the ESP32-BT-Slave Bluetooth client. Next, launch the Bluetooth terminal application and select "Devices" from the menu tab, as shown in **Figure 6 (b)**. Next, select the correct device name to connect your ESP32 Bluetooth device, in the scanned Bluetooth device list, as shown in **Figure 6 (c)**. You will know that your ESP32 board is now connected when the Bluetooth terminal app immediately prompts a message "Connected". The mobile application and the ESP32 can now communicate via Bluetooth.



(a) (b)

Figure 7 *Demonstrated Bluetooth Serial communication between a smartphone and ESP32 via Arduino IDE (a)Serial Bluetooth Terminal display and text field, (b) Arduino serial monitor text input and settings.*

As shown in **Figure 7**, open the Arduino Serial Monitor and check that the line ending and Baud rate settings are set to *New Line* (NL) and *115200 Baud*, respectively. In the Bluetooth terminal app, type something like "*hello, world!*" and press the send button. The Arduino serial monitor displays the data received from the Bluetooth terminal app. Conversely, data visible in the Bluetooth terminal app can be obtained by sending a message from the Arduino serial monitor. By creating a communication bridge between Bluetooth Serial and USB Serial via

Arduino serial monitor, this sketch essentially enables bi-directional data transfer. It provides a foundation for ESP32 microcontroller-based Bluetooth-enabled projects.

2.2.3 Reading Characters from Bluetooth Serial

One device transmits serial data to another one byte at a time. The processors use numbers to represent alphanumeric characters in bytes, including letters, numbers, and punctuation marks. Each alphanumeric character is assigned a decimal value between 0 and 255 using the American Standard Code for Information Interchange (ASCII) code [17]. Standard ASCII codes and Extended ASCII codes are the two categories of ASCII codes. Most characters, including the letters "*a*" through "z" and the numbers "*0*" through "9," are represented by standard ASCII codes ranging from 0 to 127. The first 32 characters are *control characters*, and the rest are alphanumeric characters and symbols called *printable characters*. The range of extended ASCII codes, which runs from 128 to 255, meets the need for additional characters and symbols used in numerous languages [18], [19]. An example that outputs the standard ASCII codes in various numeric formats can be found in the Arduino IDE. In the Arduino IDE, open the example sketch under "**File**" > "**Examples**" > "**04.Communication**" > "**ASCIITable**."

With the help of this Arduino sketch in **Listing 2**, you can print ASCII characters in binary, hexadecimal, octal, and decimal data formats to create an ASCII table. The program loops through ASCII characters starting with '!' or 33 in decimal and outputs their representations in various formats. Let's break down the main building blocks of the code.

Listing 2 *Print out ASCII characters in all possible data formats*

```
void setup() {
  //Initialize serial and wait for port to open:
  Serial.begin(9600);
  while (!Serial) {
    ;  // wait for serial port to connect. Needed for native USB port only
  }

  // prints title with ending line break
  Serial.println("ASCII Table ~ Character Map");
}
```

In the setup routine, a title, "ASCII Table ~ Character Map," is printed to the serial monitor using Serial.println() function.

```
// first visible ASCIIcharacter '!' is number 33:
int thisByte = 33;
// you can also write ASCII characters in single quotes.
// for example, '!' is the same as 33, so you could also use this:
// int thisByte = '!';
```

This line declares a variable thisByte and initializes it with the ASCII code from 33 ('!') to exclude the space character (32) and ASCII control characters (0 – 31). The ASCII code ends at 126 ('~'), covering 94 printable characters, aligning with the standard ASCII table.

```
void loop() {
  // prints value unaltered, i.e. the raw binary version of the byte.
  // The Serial Monitor interprets all bytes as ASCII, so 33, the first number,
  // will show up as '!'
  Serial.write(thisByte);

  Serial.print(", dec: ");
  // prints value as string as an ASCII-encoded decimal (base 10).
```

```
  // Decimal is the default format for Serial.print() and Serial.println(),
  // so no modifier is needed:
  Serial.print(thisByte);
  // But you can declare the modifier for decimal if you want to.
  // this also works if you uncomment it:

  // Serial.print(thisByte, DEC);


  Serial.print(", hex: ");
  // prints value as string in hexadecimal (base 16):
  Serial.print(thisByte, HEX);

  Serial.print(", oct: ");
  // prints value as string in octal (base 8);
  Serial.print(thisByte, OCT);

  Serial.print(", bin: ");
  // prints value as string in binary (base 2) also prints ending line break:
  Serial.println(thisByte, BIN);

  // if printed last visible character '~' or 126, stop:
  if (thisByte == 126) {  // you could also use if (thisByte == '~') {
    // This loop loops forever and does nothing
    while (true) {
      continue;
    }
  }
  // go on to the next character
  thisByte++;
}
```

The loop routine iterates over ASCII characters beginning with the variable thisByte initialized to 33, which corresponds to the ASCII character '!', the first visible ASCII character after space. The loop incrementally prints out the ASCII characters and their representations. The resulting output in the Serial Monitor, as shown in **Figure 8**, creates a visual representation of the ASCII table with different data formats for each character.

Using Serial.write() function, a raw byte is sent to the serial monitor, which interprets it as an ASCII character [20]. This differs from Serial.print(). It transmits bytes without conversion, relying on interpretation from a serial monitor. Conversely, the Serial.print() function is called with different format specifiers such as DEC, HEX, OCT, and BIN so that the byte can be displayed in different base codes [21]. These specifiers are part of the Arduino print class, which provides methods for formatting output of the serial monitor. Thus, the value of a byte is printed separately as an ASCII-encoded string, along with each ASCII character representing a different value, including decimal (base 10), hexadecimal (base 16), octal (base 8), and binary (base 2). This format specifier is useful when debugging a value in hexadecimal or understanding binary bit patterns in special cases.

The line at the end of the loop routine increments thisByte until it reaches 126, which corresponds to the ASCII character '~', at which point it enters an infinite loop via a while(true) loop. This is the simple way to halt the program after completing its task, preventing ASCII codes and characters from being printed again.

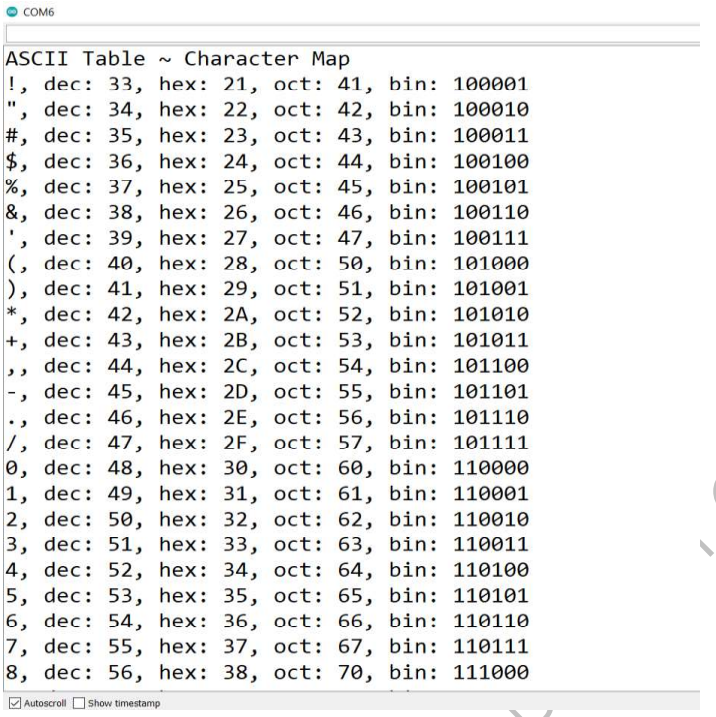


Figure 8 *ASCII table character map printed in Arduino Serial Monitor.*

2.2.4 Controlling LED from Bluetooth Terminal App

Using Bluetooth serial communication in this new Arduino sketch example, as presented in **Listing 3**, the ESP32 becomes a responsive device that can receive character data from the Bluetooth terminal app. The main idea of this example is to demonstrate how to control the built-in LED by converting the characters that are received into concrete actions. Characters are sent wirelessly and act as commands, controlling the lighting of the LED. This example perfectly captures the essence of real-time interaction by showing how Bluetooth and the ESP32 microcontroller work together seamlessly to give users the ability to remotely control physical outputs.

Listing 3 Using the Bluetooth Serial Terminal app to control the ESP32's built-in LED.

```
#include "BluetoothSerial.h"

#if !defined(CONFIG_BT_ENABLED) || !defined(CONFIG_BLUEDROID_ENABLED)
#error Bluetooth is not enabled! Please run `make menuconfig` to and enable it
#endif

#if !defined(CONFIG_BT_SPP_ENABLED)
#error Serial Bluetooth not available or not enabled. It is only available for the
ESP32 chip.
#endif

// Create object for Bluetooth classic
BluetoothSerial SerialBT;

// Create string variables
String device_name = "MyBT-ESP32";
String rcvData;
```

Two string variables are defined, device_name is set to "MyBT-ESP32", which specifies the name of the Bluetooth device broadcast during scanning, and rcvData, which is the

empty string used to aggregate the incoming Bluetooth data for processing. Using strings makes it easier to handle input of varying lengths in mobile application, such as the commands "ON" and "OFF" or '1' and '0' and supports conversion to numeric values for LED control.

```
void setup() {
  Serial.begin(115200);
  // Create Bluetooth device name
  // If not set; default is ESP32
  SerialBT.begin(device_name);

  // Initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);

  Serial.println("The device started, you can now pair it with Bluetooth.");
}
```

The setup routine configures the built-in LED, referenced as LED_BUILTIN as an output using pinMode() function. This is GPIO 25 on TTGO LILYGO LoRa32 board, preparing it for digital control. Before the setup ends, it prints a message to the Arduino serial monitor, informing you that the ESP32 is ready for Bluetooth pairing.

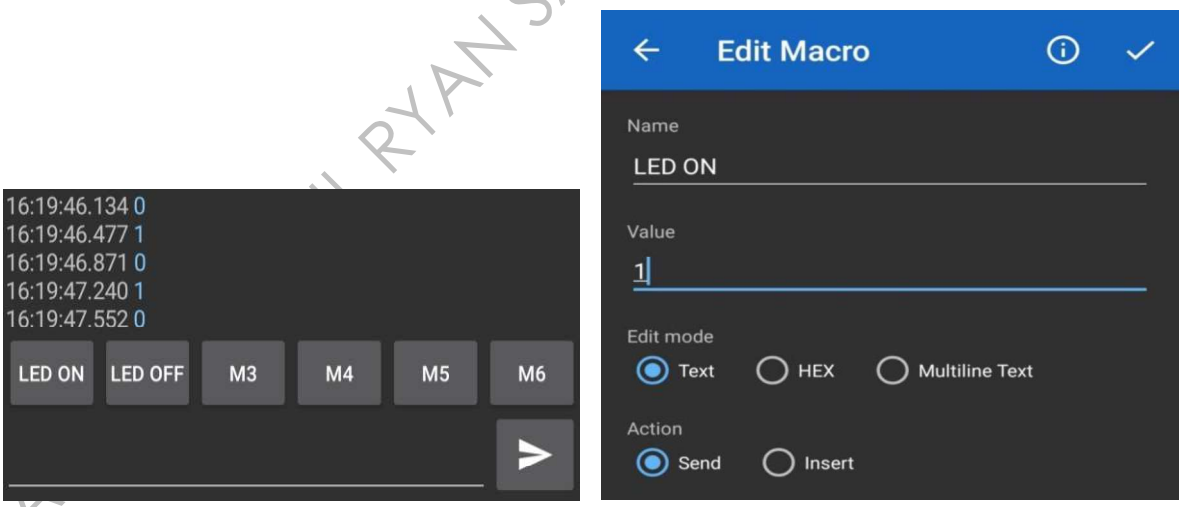
```
void loop() {
  // Check if Bluetooth serial data is available
  while (SerialBT.available()) {
    // Receive character data from Bluetooth Serial
    char rcvChar = SerialBT.read();

    // Print ASCII code character map
    Serial.write(rcvChar);
    Serial.print(", DEC: ");
    Serial.print(rcvChar, DEC);
    Serial.print(", HEX: ");
    Serial.print(rcvChar, HEX);
    Serial.print(", OCT: ");
    Serial.print(rcvChar, OCT);
    Serial.print(", BIN: ");
    Serial.print(rcvChar, BIN);
    Serial.print('\n');
    Serial.println("-------------------------------------------------");

    // Convert character array data into String format
    rcvData += rcvChar;
  }
```

The loop function continuously searches for available Bluetooth serial data. First it checks for incoming Bluetooth bytes, reading each character with SerialBT.read() function and appending it to the rcvData string. It prints the ASCII code for each received character in binary, hexadecimal, octal, and decimal format.

```
  if (rcvData.length() > 0) {
    // Print data on Serial Monitor
    Serial.print("Received BT data: ");
    Serial.print(rcvData);
    Serial.print(", number of characters: ");
    Serial.println(rcvData.length());

    // Convert valid string value to an integer
    int ledState = rcvData.toInt();
```

```
    // Control the built-in LED using '1' or '0' values
    if (ledState == 1) {
      digitalWrite(LED_BUILTIN, HIGH);
    }
    if (ledState == 0) {
      digitalWrite(LED_BUILTIN, LOW);
    }

    // Control the built-in LED using "ON" or "OFF" values
    if (rcvData.equals("ON\r\n")) {
      digitalWrite(LED_BUILTIN, HIGH);
    }
    if (rcvData.equals("OFF\r\n")) {
      digitalWrite(LED_BUILTIN, LOW);
    }

    // Clear the previous string data for new readings
    rcvData = "";
  }
  delay(10);
}
```

Next is to check that a complete message has been received before processing. If rcvData variable contains data, the code prints the received string and its length to the serial monitor, as debugging reference. It then attempts to interpret the received strings in two ways. Using rcvData.toInt() function to convert the received string into an integer [22]. This allows the LED to be controlled by numerical commands, where 1 turns the LED on and 0 turns it off. The other way is to checks for exact string matches with the commands like "ON" and "OFF" using rcvData.equals() function [23]. Finally, the rcvData string is cleared for the new readings.



(a) (b)

Figure 9 *Bluetooth Serial Terminal app (a) macro configurable buttons and (b) edit macro button settings to automate command functions.*

It is possible to create some automated button functions in the Bluetooth terminal app. The apps above the text box, as shown in **Figure 9(a)**, are identified as six (6) macro buttons named M1, M2, M3 through M6. By automating the input sequence using the macro keys, many repetitive tasks are eliminated. **Figure 7(b)** shows how to configure the macro buttons by first holding down the M1 key and then entering "LED ON" as the label and 1 as the predefined value of the macro button. Follow these steps again for the M2 macro button labeled "LED OFF," and the predefined value is 0. The M1 and M2 macro buttons have the

labels "LED ON" and "LED OFF," and the text character type values of 1 and 0 are sent to the ESP32. Try using custom macro keys to send text commands to control the ESP32's built-in LED.

2.3 Summary

This chapter focuses on integrating Bluetooth Classic into the ESP32 microcontroller sensor node, which is a major development for the development of IoT. The main objective of this chapter is to move from passive data gathering to two-way communication. The importance of this integration lies in its ability to allow wireless control and remote monitoring of sensor data in real time. By the end of this chapter, you will have gained knowledge and practical experience in wireless communications and will have demonstrated the potential to create advanced IoT applications by merging sensor devices, Bluetooth Classic technology, and mobile applications.

2.4 Mini Project 2: Integration of Wireless Control and Monitoring to a Sensor Node

2.4.1 Introduction

Bluetooth Classic, a protocol for wireless personal area network (WPAN), operates in the 2.4 GHz ISM band and offers short-range communication without the need for additional infrastructure. Building on the basic understanding gained from previous hands-on demonstrations, you will design to explore integrating Bluetooth Classic into a sensor node using the ESP32 microcontroller as a major step towards creating a responsive, real-time IoT application. This approach uses Bluetooth technology to develop an interactive wireless sensor, thus extending the functionality of the device by allowing for two-way interaction between the sensor node and external devices such as smart phones. Bluetooth Classic is designed for applications with high data rates, such as systems that require continuous data transmission. This technology is ideal for real-time applications such as smart agriculture, where the need for immediate updates and control. In this project, you will enhance the sensor node to monitor environmental parameters and control actuators wirelessly, with a particular emphasis on real-time applications in smart agriculture.

Upon completing this project, you will be able to:

- Apply the knowledge gained to build a practical Bluetooth-enabled wireless controller and monitoring device using the ESP32 microcontroller.
- Transmit sensor data such as temperature, humidity, soil moisture, light intensity from the ESP32 to a smartphone via Bluetooth Classic.
- Configure the Bluetooth terminal macro buttons as a user interface.
- Interface a physical button to actuate devices and synch with virtual macro buttons.
- Parse comma-separated values data from macro buttons.
- Employ non-blocking delay for real-time and multi-tasking operations.

2.4.2 Components Needed

Hardware Components:

1. LILYGO TTGO LoRa32 OLED ESP32 Development Board
2. Digital Temp & Humidity Sensor (AM2302/DHT22)
3. Capacitive Soil Moisture Sensor
4. Light Intensity Sensor (Photoresistor (LDR) GL5528)
5. OLED Display (SSD1306 128x64 pixels)
6. LED Indicator
7. Push Buttons

8. Jumper Wires
9. Power Source (Battery or USB power)

Software Tools:

1. Arduino IDE
2. Digikey Scheme-it Free Flow Chart Creation or similar applications
3. Bluetooth Serial Terminal Application

2.4.3 System Architecture

The system is based on a previous project, the sensor node, which includes digital temperature and humidity sensors, light sensors, and soil moisture sensors. It remains at the core of the system, providing essential information about the environment. The LEDs serve as actuators, such as an exhaust fan, water pump, and supplementary light. This project introduces the incorporation of push buttons that provide a physical interface for controlling the LEDs. Furthermore, the sensor values are not only displayed on an OLED screen for on-board data representation but also printed in the Serial Bluetooth Terminal app for remote data monitoring.

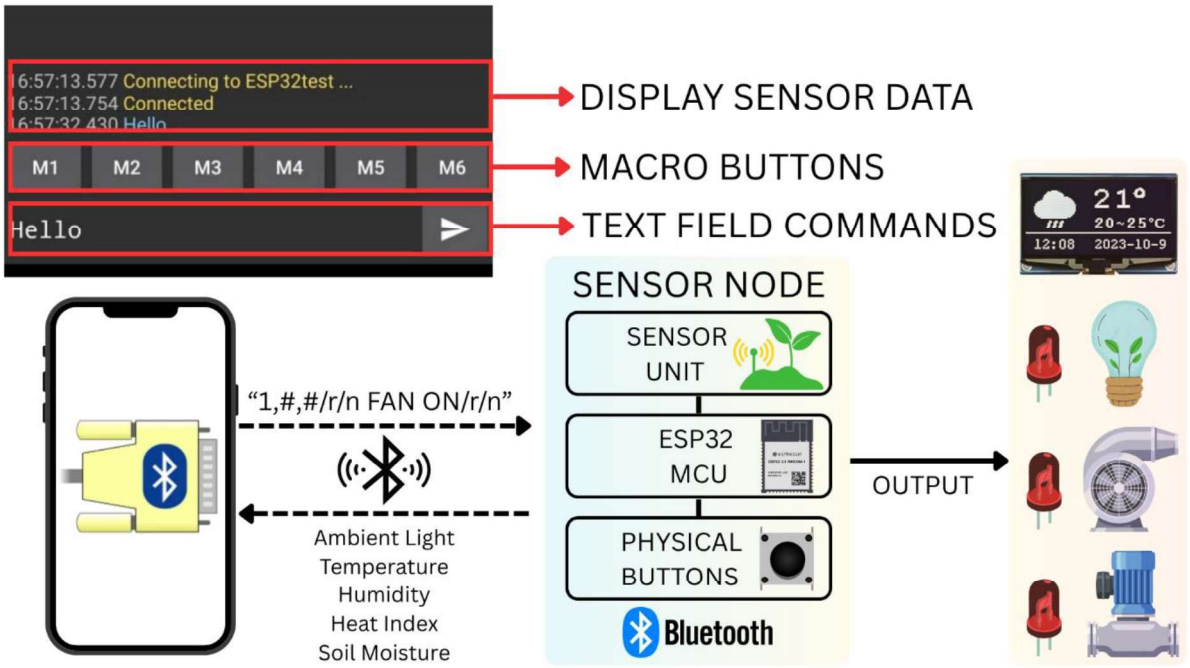


Figure 10 *Mini Project 2: System Architecture of the wireless sensor node with controls using Bluetooth Classic.*

The Bluetooth Classic two-way communication feature is a real game-changer, as it not only allows the application to receive sensor data but also remotely controls the sensor node's actuators, as shown in **Figure 10**. This allows the smartphone application to wirelessly interface with physical buttons and provides an interactive experience, as demonstrated in **Listing 4**.

Listing 4 Real-time control and sensor data transmission with Bluetooth Classic.

```
#include "BluetoothSerial.h"    // ESP32 Bluetooth classic library
#include <DHT.h>                // DHT sensor library
#include <Wire.h>               // I²C communication
#include <Adafruit_GFX.h>       // Core graphics library
#include <Adafruit_SSD1306.h>   // OLED display driver
```

```
String device_name = "ESP32-Wireless-Sensor";

// Check if Bluetooth is available
#if !defined(CONFIG_BT_ENABLED) || !defined(CONFIG_BLUEDROID_ENABLED)
#error Bluetooth is not enabled! Please run `make menuconfig` to and enable it
#endif

// Check Serial Port Profile
#if !defined(CONFIG_BT_SPP_ENABLED)
#error Serial Port Profile for Bluetooth is not available or not enabled. It is
only available for the ESP32 chip.
#endif

// Hardware configuration
#define OLED_SDA 21    // OLED SDA pin (ESP32 default)
#define OLED_SCL 22    // OLED SCL pin (ESP32 default)
#define OLED_RST -1    // Reset pin (-1 = unused)
#define DHTPIN 2       // DHT sensor data pin
#define DHTTYPE DHT22  // Sensor type (DHT22/AM2302)
#define LED_FAN 12     // LED pin as ventilator fan
#define BUTTON_FAN 36  // Push button pin to control LED

// OLED display parameters
#define SCREEN_WIDTH 128  // OLED width in pixels
#define SCREEN_HEIGHT 64  // OLED height in pixels

// Initialize hardware objects
Adafruit_SSD1306 oled(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RST);
DHT dht(DHTPIN, DHTTYPE);
BluetoothSerial SerialBT;

// Timing parameters
const uint32_t READ_INTERVAL = 3000;  // Measurement interval (ms)
uint32_t lastReadTime = 0;            // Last measurement timestamp

// Declare global variables
volatile bool fanState = false;

// Interrupt Service Routines
portMUX_TYPE mux = portMUX_INITIALIZER_UNLOCKED;

void IRAM_ATTR SwitchFan() {
  static unsigned long last_interrupt_time = 0;
  unsigned long interrupt_time = millis();

  // timer to debounce pushbutton
  if (interrupt_time - last_interrupt_time > 200) {
    // toggle light state
    portENTER_CRITICAL(&mux);
    fanState = !(fanState);
    portEXIT_CRITICAL(&mux);
  }
  last_interrupt_time = interrupt_time;
}
```

In this example, interrupt routine service (ISR) is introduced. It is a function that is executed when an interrupt event occurs. This function cannot take any parameters, cannot

return anything, and should be as fast and short as possible [24]. The ISR SwitchFan() function is defined to handle the button clicks to switch the state of the LED and simulate the control of the fan. The function implements a debounce by checking the time between interrupt events, which requires a gap of at least 200ms to filter out the button's mechanical noise. A global variable fanState, is used to transfer data between an ISR and the main program. It is declared as a volatile data type to ensure that it is shared between an ISR and the main program and is correctly updated.

```
void setup() {
  Serial.begin(115200);  // Start serial communication at 115200 baud
  dht.begin();           // Initialize DHT sensor

  // Configure I²C communication for OLED
  Wire.begin(OLED_SDA, OLED_SCL);

  // Initialize OLED display
  if (!oled.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {  // 0x3C = I²C address
    Serial.println(F("SSD1306 allocation failed"));
    while (true)
      ;  // Halt execution if display fails
  }

  // Display boot-up configuration
  oled.setTextSize(1);
  oled.setTextColor(WHITE);
  oled.display();

  // Initialize Bluetooth classic
  SerialBT.begin(device_name);
  Serial.printf("The device with name \"%s\" is started.\nNow you can pair it with
Bluetooth!\n", device_name.c_str());

  // Set GPIO configuration mode
  pinMode(BUTTON_FAN, INPUT);
  pinMode(LED_FAN, OUTPUT);

  // Attached interrupt routines to respective pushbuttons
  attachInterrupt(digitalPinToInterrupt(BUTTON_FAN), SwitchFan, FALLING);
}
```

In the setup routine, the ISR SwitchFan() function is linked to the interrupt pin BUTTON_FAN using the attachInterrupt() function [24]. The LED status can be changed by pressing the physical button, BUTTON_FAN, only on the falling edge signal (i.e., from HIGH to LOW). It is therefore important to configure the button's resistor as a pull-up resistor. The digitalPinToInterrupt() function specifies that the GPIO pin is assigned as an interrupt pin [24], [25].


```
void loop() {

  uint32_t currentTime = millis();  // Get current time

  // Execute at fixed intervals without blocking
  if (currentTime - lastReadTime >= READ_INTERVAL) {
    lastReadTime = currentTime;  // Update timing reference

    // Read sensor data (250ms sampling time required)
    float humidity = dht.readHumidity();  // Relative humidity (%)
    float tempC = dht.readTemperature();  // Temperature in °C
```

```
    // Validate sensor readings
    if (isnan(humidity) || isnan(tempC)) {
      Serial.println(F("DHT read error!"));
      return;  // Skip failed cycle
    }

    // Calculate heat index (perceived temperature)
    float heatIndexC = dht.computeHeatIndex(tempC, humidity, false);

    // Serial Monitor Output
    Serial.print(F("Temp: "));
    Serial.print(tempC);
    Serial.print(F("°C "));
    Serial.print(F("HeatIdx: "));
    Serial.print(heatIndexC);
    Serial.print(F("°C "));
    Serial.print(F("Humidity: "));
    Serial.print(humidity);
    Serial.println(F("%"));

    // OLED Display Routine
    oled.clearDisplay();
    oled.setCursor(0, 0);

    // Display formatted data
    oled.print(F("Temp: "));
    oled.print(tempC);
    oled.write(247);
    oled.println(F("C"));
    oled.print(F("HeatIdx: "));
    oled.print(heatIndexC);
    oled.write(247);
    oled.println(F("C"));
    oled.print(F("Humidity: "));
    oled.print(humidity);
    oled.println(F("%"));
    oled.display();

    // Print data to Bluetooth terminal app
    SerialBT.println("Temparature:" + String(tempC) + "°C");
    SerialBT.println("Humidity:" + String(humidity) + "%");
    SerialBT.println("Heat Index:" + String(heatIndexC) + "°C");
  }

  // Drive the LED indicators
  digitalWrite(LED_FAN, fanState);
}
```

The loop routine manages the regular acquisition, display, and transmission of sensor data, while continuously updating the status of the LED. It uses non-blocking timing by means of the millis() function and checks whether 3,000ms have passed since the last reading of the sensor. Within this sampling period, the temperature, humidity, and heat index values are read off by the DHT sensor. All collected sensor data is simultaneously printed to an Arduino serial monitor for debugging, an OLED screen for real-time visualization, and a Bluetooth Serial for sending the serial data to a connected mobile application using SerialBT.println() function.

At the end of the loop, the line digitalWrite(LED_FAN, fanState) directly controls the LED_FAN based on the current value of the fanState boolean variable. Since the fanState variable is shifted by the ISR SwitchFan() function, it allows immediate control of the LED in response to physical button presses.

Furthermore, the LED status can be changed by using the macro buttons in the application. You can assign data representing the status of the actuator to each macro button, as shown in **Figure 9 (b)**. You will use the CSV data structure for this project. CSV is a set of records separated by a line feed (LF), also known as a newline character (\n), and a pair of carriage return, and line feed (CR/LF) represents characters (\r\n) can be found in a CSV file. Every record has a set of values with commas between them [26]. For example, humidity, temperature, light, and soil moisture) could be expressed as "88,23.74,358.41,93.64\r\n." CSV data can be interpreted using a character array or a string data type.

The ability to parse CSV data from string and character arrays is essential in serial communications applications. When working with data streams, parsing allows us to extract and manipulate individual data fields, especially in situations where the information is organized in CSV format. By breaking the received characters into their respective components using separators such as commas, we can process and use the data efficiently. Whether working with strings or character arrays, the parsing process involves identifying delimiters, extracting substrings, and converting them to appropriate data types. This capability is valuable in applications ranging from sensor data transmission to command interpretation, providing a foundation for seamless and structured communications in the serial interfaces. **Listing 5** demonstrates parsing sensor data in CSV format within a given string. Let's break down the code and understand its main components.

Listing 5 Parsing CSV values from a string.

```
String rcvString = "88,23.74,358.41,93.64\r\n";
int idx_1, idx_2, idx_3;
```

A string rcvString is declared with example CSV formatted sensor data: "88,23.74,358.41,93.64\r\n". Three integer variables, idx_1, idx_2, and idx_3 are declared to store the indices of the commas in the string.

```
void setup() {
  Serial.begin(115200);
  idx_1 = rcvString.indexOf(',');
  idx_2 = rcvString.indexOf(',', idx_1 + 1);
  idx_3 = rcvString.indexOf(',', idx_2 + 1);
}
```

The indices of the commas in the rcvString are calculated and saved in the respective variables. We pass a single input for the first index (idx_1), where the counts start at 0 for the index value to search in each string value. To count the index value of the next comma delimiter to be searched for, we pass an argument idx_nth + 1 to the remaining function as the starting point.

```
void loop() {
  String firstValue = rcvString.substring(0, idx_1);
  String secondValue = rcvString.substring(idx_1 + 1, idx_2);
  String thirdValue = rcvString.substring(idx_2 + 1, idx_3);
  String fourthValue = rcvString.substring(idx_3 + 1);

  Serial.print("Humidity: ");
  Serial.println(firstValue);
  Serial.print("Temperature: ");
```

```
  Serial.println(secondValue);
  Serial.print("Light: ");
  Serial.println(thirdValue);
  Serial.print("Soil Moisture: ");
  Serial.println(fourthValue);
  delay(1000);
}
```

In the loop routine, the code uses the substring function to extract individual values from the CSV string based on the comma positions. Four String variables (i.e., "humidity", "temperature", "light", and "soil moisture") are assigned substrings representing different data fields. In the CSV data, the Arduino sketch above returns the substring of interest, also called fields. The extracted values are then printed on the Arduino serial monitor. Another method of parsing CSV from a character array variable is shown in **Listing 6**.

Listing 6 Parsing CSV values from a character array.

```
const byte charLength = 32;
char csvChar[charLength] = "88,23.74,358.41,93.64\r\n";
char tempChar[charLength];
char *value;
```

The code begins with defining constants and initializing variables charLength specifies the maximum length of the character array, and csvChar contains the sensor data in CSV format. To preserve the original data during parsing, a temporary character array tempChar is created.

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

void loop() {
  // Copy original data to temporary variable to protect the original data
  strcpy(tempChar, csvChar);

  // Parse CSV character array using strtok() function
  value = strtok(tempChar, ",");
  char *firstValue = value;

  value = strtok(NULL, ",");
  char *secondValue = value;

  value = strtok(NULL, ",");
  char *thirdValue = value;

  value = strtok(NULL, ",");
  char *fourthValue = value;

  Serial.print("Humidity: ");
  Serial.println(firstValue);
  Serial.print("Temperature: ");
  Serial.println(secondValue);
  Serial.print("Light: ");
  Serial.println(thirdValue);
  Serial.print("Soil Moisture: ");
  Serial.println(fourthValue);
  delay(1000);
}
```

The original CSV data is copied to a temporary variable, tempChar, to avoid modifying the original data using the strcpy() function [27]. The strtok() function is used to tokenize the CSV data based on the comma delimiter [28]. It returns a pointer to the next token in the string. The parsed values are assigned to individual variables (i.e., "humidity", "temperature", "light", and "soil moisture") corresponding to different sensor readings.

2.4.4 Project Plan

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

1. You are free to assemble the circuit by connecting sensors, the OLED display, buttons, and the LED to the ESP32 from the trainer board.
2. Utilize the provided Arduino sketch as a reference for synchronizing physical buttons with virtual buttons. Modify the code to incorporate logic that interprets CSV-formatted data sent from virtual buttons. Implement a mechanism to parse incoming CSV data and use it to control the LEDs connected to the ESP32.
3. Integrate virtual buttons into your Serial Bluetooth Terminal App using macro keys. Configure these virtual buttons to send CSV-formatted logic values representing the button's on and off states. Designate each virtual button to control a specific LED on your sensor node.
4. Ensure your sensor node continues to read environmental data (i.e., temperature, humidity, light, soil moisture) using the interfaced sensors while controlling the LEDs with physical and virtual buttons in real-time. Display this sensor data in real-time on the Serial Bluetooth Terminal App and OLED display, providing a comprehensive view of your environment.

2.4.5 Guidelines and Tips

- Create six (6) macro keys as virtual keys via the Serial Bluetooth Terminal App. Assign three (3) buttons to turn on the LEDs and three (3) to turn them off. Label them differently.
- The CSV data structure is designed accordingly as "fanState, pumpState, lightState". A Boolean value (1 or 0) is inserted into each field, indicating the state of the corresponding LED. For example, to turn on the fan, the CSV data for the corresponding macro button is "1, #, #". To turn off the fan, the CSV data in the other macro button is "0, #, #". The "#" character is a dummy value, while 1 and 0 are meaningful Boolean data.
- Choose between string or character array methods for parsing CSV data. Both approaches are acceptable, and the choice can be based on individual programming preferences.

References

[1] Intel, "What Is Bluetooth® Technology?," Intel Corporation. Accessed: Nov. 28, 2023. [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/wireless/what-is-bluetooth.html

[2] The Editors of Britannica Encyclopedia, "Bluetooth | Definition, Uses, History, & Facts," *Encyclopedia Britannica*. Nov. 26, 2023. Accessed: Nov. 28, 2023. [Online]. Available: https://www.britannica.com/technology/Bluetooth

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

[4] "Baptism of Harald Bluetooth - World History Encyclopedia." Accessed: Jul. 31, 2025. [Online]. Available: https://www.worldhistory.org/image/13878/baptism-of-harald-bluetooth/

[5] Bluetooth® Technology Website, "Origin of the Name | Bluetooth® Technology Website," Bluetooth SIG, Inc. Accessed: Nov. 29, 2023. [Online]. Available: https://www.bluetooth.com/about-us/bluetooth-origin/

[6] M. Abet, "Bluetooth vs Bluetooth Low Energy, what's the difference?," ELA Innovation. Accessed: Nov. 29, 2023. [Online]. Available: https://elainnovation.com/en/bluetooth-vs-bluetooth-low-energy-whats-the-difference/

[7] Sony Support, "What is the Maximum Communication Range of the Bluetooth connection?," Sony Electronics Inc. Accessed: Dec. 05, 2023. [Online]. Available: https://www.sony.com/electronics/support/articles/00014085

[8] Bluetooth® Technology Website, "Understanding Bluetooth Range," Bluetooth SIG, Inc. Accessed: Dec. 05, 2023. [Online]. Available: https://www.bluetooth.com/learn-about-bluetooth/key-attributes/range/#estimator

[9] J. Marcel, "3 Key Factors That Determine the Range of Bluetooth," Bluetooth SIG, Inc. Accessed: Dec. 05, 2023. [Online]. Available: https://www.bluetooth.com/blog/3-key-factors-that-determinethe-range-of-bluetooth/

[10] "Topology Options | Bluetooth® Technology Website." Accessed: Jul. 31, 2025. [Online]. Available: https://www.bluetooth.com/learn-about-bluetooth/topology-options/

[11] P. Khatri, "Arduino Based Voice Controlled LEDs using Bluetooth," Circuit Digest. Accessed: Dec. 06, 2023. [Online]. Available: https://circuitdigest.com/microcontroller-projects/arduino-based-voice-controlled-leds

[12] survy2014, "HC-05 Wireless Bluetooth RF Transceiver Module Serial RS232 TTL for Arduino," eBay. Accessed: Dec. 06, 2023. [Online]. Available: https://www.ebay.ph/itm/400985350948

[13] Feasycom, "UART Communication Bluetooth Module," Feasycom 2023. Accessed: Dec. 06, 2023. [Online]. Available: https://www.feasycom.com/uart-communication-bluetooth-module.html

[14] Espressif Systems, "ESP32 Bluetooth Architecture," 2019. Accessed: Dec. 07, 2023. [Online]. Available: https://espressif.com/sites/default/files/documentation/esp32_bluetooth_architecture_en.pdf

[15] Espressif Systems, "Bluetooth® Overview," ESP-IDF Programming Guide. Accessed: Dec. 07, 2023. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/bluetooth.html

[16] Microsoft Make Code, "Bluetooth UART Service," Microsoft. Accessed: Dec. 07, 2023. [Online]. Available: https://makecode.microbit.org/reference/bluetooth/start-uart-service

[17] "Interpreting Serial Data," Code, Circuits, & Construction. Accessed: Dec. 11, 2023. [Online]. Available: https://www.tigoe.com/pcomp/code/communication/interpreting-serial-data-bytes/

[18] "Table of ASCII Codes, Characters and Symbols," Injosoft AB. Accessed: Dec. 12, 2023. [Online]. Available: https://www.ascii-code.com/

[19] "ASCII Chart," CommFront. Accessed: Dec. 11, 2023. [Online]. Available: https://www.commfront.com/pages/ascii-chart

[20] Arduino Docs, "Serial.write() | Arduino Documentation," Arduino. Accessed: Aug. 01, 2025. [Online]. Available: https://docs.arduino.cc/language-reference/en/functions/communication/serial/write/

[21] Arduino Docs, "Serial.print() | Arduino Documentation," Arduino. Accessed: Aug. 01, 2025. [Online]. Available: https://docs.arduino.cc/language-reference/en/functions/communication/serial/print/

[22] Arduino Docs, "toInt() | Arduino Documentation," Arduino. Accessed: Aug. 01, 2025. [Online]. Available: https://docs.arduino.cc/language-reference/en/variables/data-types/stringObject/Functions/toInt/

[23] Arduino Docs, "equals() | Arduino Documentation," Arduino. Accessed: Aug. 01, 2025. [Online]. Available: https://docs.arduino.cc/language-reference/en/variables/data-types/stringObject/Functions/equals/

[24] Arduino Docs, "attachInterrupt() | Arduino Documentation," Arduino. Accessed: Aug. 02, 2025. [Online]. Available: https://docs.arduino.cc/language-reference/en/functions/external-interrupts/attachInterrupt/

[25] Espressif, "GPIO: Arduino ESP32 Documentation," Espressif Systems (Shanghai) Co., Ltd. Accessed: Jul. 26, 2025. [Online]. Available: https://docs.espressif.com/projects/arduino-esp32/en/latest/api/gpio.html

[26] IBM Documentation, "Example Comma-Separated Value (CSV) File," IBM Corporation. Accessed: Dec. 12, 2023. [Online]. Available: https://www.ibm.com/docs/en/sim/6.0.2?topic=schedules-example-comma-separated-value-csv-file

[27] cplusplus, "strcpy," cplusplus.com. Accessed: Aug. 02, 2025. [Online]. Available: https://cplusplus.com/reference/cstring/strcpy/

[28] cplusplus, "strtok," cplusplus.com. Accessed: Aug. 02, 2025. [Online]. Available: https://cplusplus.com/reference/cstring/strtok/