Assignment - IOT Intern

TASK: Low Energy Bluetooth Service Broadcasting

Submitted to

Nineti GmbH

Submitted by

Aditya Singh

(Final year IOT student)

Introduction

This document details the implementation of Low Energy Bluetooth (BLE) service broadcasting using an IoT development board such as the ESP32 or Arduino with built-in Bluetooth capabilities. The task involves broadcasting temperature and humidity data as BLE services, which can be read using the nRF-Connect app.

In the era of interconnected devices, the Internet of Things (IoT) plays a pivotal role in enhancing automation and data exchange across various domains. One of the core technologies enabling IoT is Bluetooth Low Energy (BLE), a wireless communication protocol designed for short-range and low-power communication. BLE is particularly advantageous for applications that require infrequent data transmissions, such as sensor readings, due to its energy efficiency and robust connectivity.

This document outlines the implementation of BLE service broadcasting using an IoT development board, specifically the ESP32 or Arduino. The task involves creating a BLE service that broadcasts temperature and humidity data, which can be accessed and monitored using the nRF-Connect app. By simulating or interfacing actual sensors, the development board will periodically update and broadcast these readings, allowing BLE-enabled devices to receive the data in real-time.

### **Objectives**

The primary objectives of this task are:

1. **Implement BLE Service Broadcasting**: Develop a BLE service on an ESP32 or Arduino board that broadcasts temperature and humidity data.
2. **Use Standard BLE Characteristics**: Utilize standard BLE characteristics for temperature and humidity to ensure compatibility with BLE clients.
3. **Enable Read and Notify Operations**: Ensure that the BLE characteristics support both read and notify operations, allowing for dynamic data updates.
4. **Demonstrate Functionality**: Validate the implementation by reading the broadcasted values using the nRF-Connect app.

### **Significance**

Implementing BLE service broadcasting has significant implications for various applications, including:

* **Environmental Monitoring**: Real-time monitoring of environmental conditions in smart homes, greenhouses, or industrial settings.
* **Health and Fitness**: Wearable devices that track and broadcast vital signs and environmental conditions to smartphones or other monitoring systems.
* **Automation Systems**: Integration with home automation systems to adjust heating, ventilation, and air conditioning (HVAC) systems based on real-time data.

This project serves as a foundation for understanding and implementing BLE in IoT devices, providing a stepping stone for more complex and integrated IoT solutions.

## **About ESP32**

The ESP32 is a powerful and versatile microcontroller module developed by Espressif Systems. It features built-in WiFi and Bluetooth connectivity, making it ideal for IoT applications requiring wireless communication capabilities. Here are key features and benefits of using the ESP32 for your project:

### **Key Features:**

* **Dual-core Processor**: ESP32 integrates a dual-core Tensilica Xtensa LX6 processor, which allows for efficient multitasking and processing of data.
* **Wireless Connectivity**: Besides Bluetooth Low Energy (BLE), ESP32 supports WiFi connectivity (802.11 b/g/n), providing flexibility in networking solutions.
* **Low Power Consumption**: ESP32 is designed for low power consumption, making it suitable for battery-operated devices and energy-efficient applications.
* **Rich Peripherals**: It offers a wide range of peripherals including GPIOs, SPI, I2C, UART, ADC, DAC, and more, enabling versatile interfacing with sensors and actuators.
* **Security Features**: ESP32 includes hardware-accelerated encryption algorithms (AES, SHA-2, RSA), ensuring secure data transmission over wireless networks.
* **Development Support**: Supported by the Arduino IDE, ESP-IDF (Espressif IoT Development Framework), and a vibrant community, ESP32 facilitates rapid prototyping and development.

### **Applications:**

* **IoT Devices**: ESP32 is widely used in IoT applications such as smart home devices, environmental monitoring systems, and industrial automation.
* **Wearable Technology**: Its low power consumption and compact size make ESP32 suitable for wearable devices that require Bluetooth connectivity.
* **Consumer Electronics**: ESP32 is used in consumer electronics products like smartwatches, fitness trackers, and remote-controlled gadgets.

### **Development Environment:**

* **Arduino IDE**: Programming the ESP32 can be done using the Arduino IDE, leveraging its simplicity and extensive library support.
* **ESP-IDF**: For advanced users, ESP-IDF provides deeper control over ESP32’s functionalities and allows for custom firmware development.

### **Community and Support:**

* Espressif Systems provides comprehensive documentation, tutorials, and forums to support developers working with ESP32. The active community ensures quick resolution of issues and continuous improvement of development resources.

## **About DHT11**

The DHT11 is a basic, low-cost digital temperature and humidity sensor. It operates using a digital signal protocol, making it easy to interface with microcontrollers like Arduino or ESP32. Here are some key points about the DHT11:

**Measurement Range**: It can measure temperatures from 0°C to 50°C with an accuracy of ±2°C, and humidity from 20% to 80% with an accuracy of ±5%.

**Communication**: It communicates over a single-wire digital interface, which makes it straightforward to integrate into projects.

**Power Requirements**: It operates at 3.3V to 5V DC and consumes very low power, suitable for battery-operated applications.

**Output**: The sensor outputs data in a proprietary format that needs to be decoded by the microcontroller.

**Usage**: Commonly used in weather stations, HVAC systems, and other projects requiring environmental monitoring.

**Jumper wire:**

A jumper wire is a electrical wire used to connect two terminals or pins on a breadboard, development board, or other electronic components. It's a short, flexible wire that allows you to "jump" a connection from one point to another, hence the name.

In the context of my task, I use jumper wires to connect your IoT development board (e.g. ESP32 or Arduino) to a temperature sensor, humidity sensor, or other components.

Here are some details about jumper wires:

- Typically 20-25 AWG (American Wire Gauge) wire

- Insulated with a flexible plastic or rubber coating

- Ends are usually terminated with a pin or socket that can be inserted into a breadboard or development board

- Available in various colors and lengths

**Breadboard**

A breadboard is a reusable platform used to build and prototype electronic circuits. It's a plastic board with a grid of holes and copper connections that allow you to easily connect and disconnect electronic components, wires, and other devices.

Here are some key features of a breadboard:

- Grid of holes: Typically 0.1 inches (2.54 mm) spaced, allowing for easy insertion of component leads

- Copper connections: Underneath the board, connecting holes in a specific pattern (usually rows and columns)

- Reusable: Components can be easily removed and replaced without damaging the board

- Versatile: Suitable for building a wide range of electronic projects, from simple circuits to complex prototypes

Breadboards are commonly used in electronics prototyping, circuit design, and development, making them a fundamental tool for makers, students, and professionals alike.

**Micro-USB**

A Micro-USB is a type of connector used for charging and data transfer between devices. It's a smaller version of the standard USB (Universal Serial Bus) connector, typically used on mobile devices, cameras, and other portable electronics.

Here are some key features of Micro-USB:

- Smaller size: Compared to standard USB connectors, Micro-USB is more compact and suitable for smaller devices

- Reversible: Can be inserted either way into a port, unlike standard USB connectors

- Data transfer: Supports data transfer between devices, such as transferring files between a phone and computer

- Charging: Supports charging of devices, such as charging a phone or tablet

- Common use: Found on many devices, including Android phones, tablets, cameras, and e-readers

Micro-USB has been widely adopted as a standard connector for many devices, but it's being gradually replaced by USB-C (USB Type-C) in newer devices, which offers faster data transfer and charging speeds.

**Arduino IDE**

The Arduino Integrated Development Environment (IDE) is a software application that allows users to write, compile, and upload code to Arduino boards. It's a free, open-source platform that provides a comprehensive development environment for creating interactive projects, prototypes, and devices.

Here are some key features of the Arduino IDE:

- Code editor: Write and edit code in a syntax-highlighted environment

- Compiler: Compile code into a format that can be uploaded to Arduino boards

- Uploader: Upload compiled code to Arduino boards

- Library manager: Manage libraries and dependencies for projects

- Serial monitor: Communicate with Arduino boards via serial communication

- Debugging tools: Debug code with tools like the serial monitor and error messages

The Arduino IDE supports a wide range of Arduino boards, including popular ones like the Uno, Mega, and Nano. It's available for Windows, macOS, and Linux operating systems.

The IDE is designed to be user-friendly and accessible, making it a great tool for beginners and experienced developers alike. It's widely used in various fields, including education, prototyping, art, and engineering.

## **Prerequisites**

* **Development Board**: ESP32 with Wifi & Bluetooth capability
* **Sensors**: Temperature and humidity sensors (DHT11)
* **Software**: Arduino IDE or ESP-IDF, nRF-Connect app
* **Account**: GitHub for code submission
* **Jumper wires**
* **Micro-USB**
* **Breadboard**

**Objective**

The objective is to write code that enables the development board to broadcast two BLE services:

1. Temperature Measurement
2. Humidity

These services should be accessible via the nRF-Connect app.

**Service Details**

### **UUID and Characteristics**

* **Service UUID**: 00000002-0000-0000-FDFD-FDFDFDFDFDFD
* **Characteristics**:
  + **Temperature Measurement**: Standard BLE characteristic
  + **Humidity**: Standard BLE characteristic

Both characteristics support read and notify operations.

**Implementation**

**Code**

**For connecting esp32 with n-RF connect app & mimic temperature & humidity**

#include <BLEDevice.h>

#include <BLEServer.h>

#include <BLEUtils.h>

#include <BLE2902.h>

// UUIDs for the BLE service and characteristics

#define SERVICE\_UUID "00000002-0000-0000-FDFD-FDFDFDFDFDFD"

#define CHARACTERISTIC\_TEMPERATURE\_UUID "00002A6E-0000-1000-8000-00805F9B34FB"

#define CHARACTERISTIC\_HUMIDITY\_UUID "00002A6F-0000-1000-8000-00805F9B34FB"

// Placeholder function for temperature and humidity sensor data

float getTemperature() {

return 25.0; // Simulated temperature value

}

float getHumidity() {

return 60.0; // Simulated humidity value

}

BLECharacteristic \*pTemperatureCharacteristic;

BLECharacteristic \*pHumidityCharacteristic;

void setup() {

Serial.begin(115200);

BLEDevice::init("ESP32\_BLE");

BLEServer \*pServer = BLEDevice::createServer();

BLEService \*pService = pServer->createService(SERVICE\_UUID);

// Create Temperature Characteristic

pTemperatureCharacteristic = pService->createCharacteristic(

CHARACTERISTIC\_TEMPERATURE\_UUID,

BLECharacteristic::PROPERTY\_READ | BLECharacteristic::PROPERTY\_NOTIFY

);

// Create Humidity Characteristic

pHumidityCharacteristic = pService->createCharacteristic(

CHARACTERISTIC\_HUMIDITY\_UUID,

BLECharacteristic::PROPERTY\_READ | BLECharacteristic::PROPERTY\_NOTIFY

);

pService->start();

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();

Serial.println("BLE service is broadcasting");

}

void loop() {

// Update sensor values

float temperature = getTemperature();

float humidity = getHumidity();

// Set the new values

pTemperatureCharacteristic->setValue(temperature);

pHumidityCharacteristic->setValue(humidity);

// Notify clients

pTemperatureCharacteristic->notify();

pHumidityCharacteristic->notify();

delay(1000); // Update every second

}

### **Detailed Explanation**

1. **BLE Initialization**: The code initializes BLE with a device name "ESP32\_BLE".
2. **Service and Characteristics Creation**: A BLE service and two characteristics (Temperature and Humidity) are created with the specified UUIDs.
3. **Simulated Data**: Functions getTemperature() and getHumidity() simulate sensor data.
4. **Advertising**: The BLE service is advertised so it can be discovered by BLE clients like the nRF-Connect app.
5. **Data Broadcasting**: In the loop, sensor data is updated and broadcasted via the notify property.

**Code for extra credit exp32 connect with wifi**

#include <WiFi.h>

BLECharacteristic \*pSSIDCharacteristic;

BLECharacteristic \*pPasswordCharacteristic;

void connectToWiFi(const char\* ssid, const char\* password) {

WiFi.begin(ssid, password);

while (WiFi.status() != WL\_CONNECTED) {

delay(1000);

Serial.println("Connecting to WiFi...");

}

Serial.println("Connected to WiFi");

}

void setup() {

// Existing setup code...

// Additional BLE Characteristics for WiFi credentials

pSSIDCharacteristic = pService->createCharacteristic(

BLEUUID((uint16\_t)0xFFFF),

BLECharacteristic::PROPERTY\_WRITE

);

pPasswordCharacteristic = pService->createCharacteristic(

BLEUUID((uint16\_t)0xFFFE),

BLECharacteristic::PROPERTY\_WRITE

);

// Add callback for when WiFi credentials are received

pSSIDCharacteristic->setCallbacks(new WiFiCallback());

pPasswordCharacteristic->setCallbacks(new WiFiCallback());

}

class WiFiCallback: public BLECharacteristicCallbacks {

void onWrite(BLECharacteristic \*pCharacteristic) {

std::string value = pCharacteristic->getValue();

if (pCharacteristic == pSSIDCharacteristic) {

ssid = value.c\_str();

} else if (pCharacteristic == pPasswordCharacteristic) {

password = value.c\_str();

}

if (ssid.length() > 0 && password.length() > 0) {

connectToWiFi(ssid.c\_str(), password.c\_str());

}

}

};

void loop() {

// Existing loop code...

}

**Code for esp32 simple wifi server**

#include <WiFi.h>

const char \*ssid = "My 3V";

const char \*password = "12345678";

int ledPin = 2;

NetworkServer server(80);

void setup() {

Serial.begin(115200);

pinMode(ledPin, OUTPUT); // set the LED pin mode

delay(10);

// We start by connecting to a WiFi network

Serial.println();

Serial.println();

Serial.print("Connecting to ");

Serial.println(ssid);

WiFi.begin(ssid, password);

while (WiFi.status() != WL\_CONNECTED) {

delay(500);

Serial.print(".");

}

Serial.println("");

Serial.println("WiFi connected.");

Serial.println("IP address: ");

Serial.println(WiFi.localIP());

server.begin();

}

void loop() {

NetworkClient client = server.accept(); // listen for incoming clients

if (client) { // if you get a client,

Serial.println("New Client."); // print a message out the serial port

String currentLine = ""; // make a String to hold incoming data from the client

while (client.connected()) { // loop while the client's connected

if (client.available()) { // if there's bytes to read from the client,

char c = client.read(); // read a byte, then

Serial.write(c); // print it out the serial monitor

if (c == '\n') { // if the byte is a newline character

// if the current line is blank, you got two newline characters in a row.

// that's the end of the client HTTP request, so send a response:

if (currentLine.length() == 0) {

// HTTP headers always start with a response code (e.g. HTTP/1.1 200 OK)

// and a content-type so the client knows what's coming, then a blank line:

client.println("HTTP/1.1 200 OK");

client.println("Content-type:text/html");

client.println();

// the content of the HTTP response follows the header:

client.print("Click <a href=\"/H\">here</a> to turn the LED on pin 5 on.<br>");

client.print("Click <a href=\"/L\">here</a> to turn the LED on pin 5 off.<br>");

// The HTTP response ends with another blank line:

client.println();

// break out of the while loop:

break;

} else { // if you got a newline, then clear currentLine:

currentLine = "";

}

} else if (c != '\r') { // if you got anything else but a carriage return character,

currentLine += c; // add it to the end of the currentLine

}

// Check to see if the client request was "GET /H" or "GET /L":

if (currentLine.endsWith("GET /H")) {

digitalWrite(ledPin, HIGH); // GET /H turns the LED on

}

if (currentLine.endsWith("GET /L")) {

digitalWrite(ledPin, LOW); // GET /L turns the LED off

}

}

}

// close the connection:

client.stop();

Serial.println("Client Disconnected.");

}

}

**Code for esp32 wifiaccesspoint**

#include <WiFi.h>

#include <NetworkClient.h>

#include <WiFiAP.h>

#ifndef LED\_BUILTIN

#define LED\_BUILTIN 2 // Set the GPIO pin where you connected your test LED or comment this line out if your dev board has a built-in LED

#endif

// Set these to your desired credentials.

const char \*ssid = "ESP-32";

const char \*password = "ESP32@832";

NetworkServer server(80);

void setup() {

pinMode(LED\_BUILTIN, OUTPUT);

Serial.begin(115200);

Serial.println();

Serial.println("Configuring access point...");

// You can remove the password parameter if you want the AP to be open.

// a valid password must have more than 7 characters

if (!WiFi.softAP(ssid, password)) {

log\_e("Soft AP creation failed.");

while (1);

}

IPAddress myIP = WiFi.softAPIP();

Serial.print("AP IP address: ");

Serial.println(myIP);

server.begin();

Serial.println("Server started");

}

void loop() {

NetworkClient client = server.accept(); // listen for incoming clients

if (client) { // if you get a client,

Serial.println("New Client."); // print a message out the serial port

String currentLine = ""; // make a String to hold incoming data from the client

while (client.connected()) { // loop while the client's connected

if (client.available()) { // if there's bytes to read from the client,

char c = client.read(); // read a byte, then

Serial.write(c); // print it out the serial monitor

if (c == '\n') { // if the byte is a newline character

// if the current line is blank, you got two newline characters in a row.

// that's the end of the client HTTP request, so send a response:

if (currentLine.length() == 0) {

// HTTP headers always start with a response code (e.g. HTTP/1.1 200 OK)

// and a content-type so the client knows what's coming, then a blank line:

client.println("HTTP/1.1 200 OK");

client.println("Content-type:text/html");

client.println();

// the content of the HTTP response follows the header:

client.print("Click <a href=\"/H\">here</a> to turn ON the LED.<br>");

client.print("Click <a href=\"/L\">here</a> to turn OFF the LED.<br>");

// The HTTP response ends with another blank line:

client.println();

// break out of the while loop:

break;

} else { // if you got a newline, then clear currentLine:

currentLine = "";

}

} else if (c != '\r') { // if you got anything else but a carriage return character,

currentLine += c; // add it to the end of the currentLine

}

// Check to see if the client request was "GET /H" or "GET /L":

if (currentLine.endsWith("GET /H")) {

digitalWrite(LED\_BUILTIN, HIGH); // GET /H turns the LED on

}

if (currentLine.endsWith("GET /L")) {

digitalWrite(LED\_BUILTIN, LOW); // GET /L turns the LED off

}

}

}

// close the connection:

client.stop();

Serial.println("Client Disconnected.");

}

}

**Code esp32 connecting with bluetooth:**

#include "BluetoothSerial.h"

BluetoothSerial serialBT;

char cmd;

void setup() {

serialBT.begin("Esp32-BT");

pinMode(2, OUTPUT);

}

void loop() {

if(serialBT.available()){

cmd = serialBT.read();

}

if(cmd == '1'){

digitalWrite(2, HIGH);

}

if(cmd == '0'){

digitalWrite(2, LOW);

}

delay(20);

}

**Code for connecting esp32 with n-RF app**

#include <BLEDevice.h>

#include <BLEServer.h>

#include <BLEUtils.h>

#include <BLE2902.h>

BLECharacteristic \*pCharacteristic;

bool deviceConnected = false;

int txValue = 0;

#define SERVICE\_UUID "00000002-0000-0000-FDFD-FDFDFDFDFDFD"

#define CHARACTERISTIC\_UUID\_TX "00000002-0000-0000-FDFD-FDFDFDFDFDFD"

class MyServerCallbacks : public BLEServerCallbacks {

void onConnect(BLEServer \*pServer) {

deviceConnected = true;

};

void onDisconnect(BLEServer \*pServer) {

deviceConnected = false;

}

};

void setup() {

Serial.begin(9600);

// Create the BLE Device

BLEDevice::init("ESP32");

// Create the BLE Server

BLEServer \*pServer = BLEDevice::createServer();

pServer->setCallbacks(new MyServerCallbacks());

// Create the BLE Service

BLEService \*pService = pServer->createService(SERVICE\_UUID);

// Create a BLE Characteristic

pCharacteristic = pService->createCharacteristic(

CHARACTERISTIC\_UUID\_TX,

BLECharacteristic::PROPERTY\_NOTIFY);

//BLE2902 needed to notify

pCharacteristic->addDescriptor(new BLE2902());

// Start the service

pService->start();

// Start advertising

pServer->getAdvertising()->start();

Serial.println("Waiting for a client connection to notify...");

}

void loop() {

if (deviceConnected) {

txValue = random(10, 50);

// Conversion of txValue

char txString[8];

dtostrf(txValue, 1, 2, txString);

//Setting the value to the characteristic

pCharacteristic->setValue(txString);

//Notifying the connected client

pCharacteristic->notify();

Serial.println("Sent value: " + String(txString));

delay(500);

}

}

**Code to measure temprature & humidity using DHT11**

#include "DHT.h"

#define DPIN 4 //Pin to connect DHT sensor (GPIO number)

#define DTYPE DHT11 // Define DHT 11 or DHT22 sensor type

DHT dht(DPIN,DTYPE);

void setup() {

Serial.begin(9600);

dht.begin();

}

void loop() {

delay(2000);

float tc = dht.readTemperature(false); //Read temperature in C

float tf = dht.readTemperature(true); //Read Temperature in F

float hu = dht.readHumidity(); //Read Humidity

Serial.print("Temp: ");

Serial.print(tc);

Serial.print(" C, ");

Serial.print(tf);

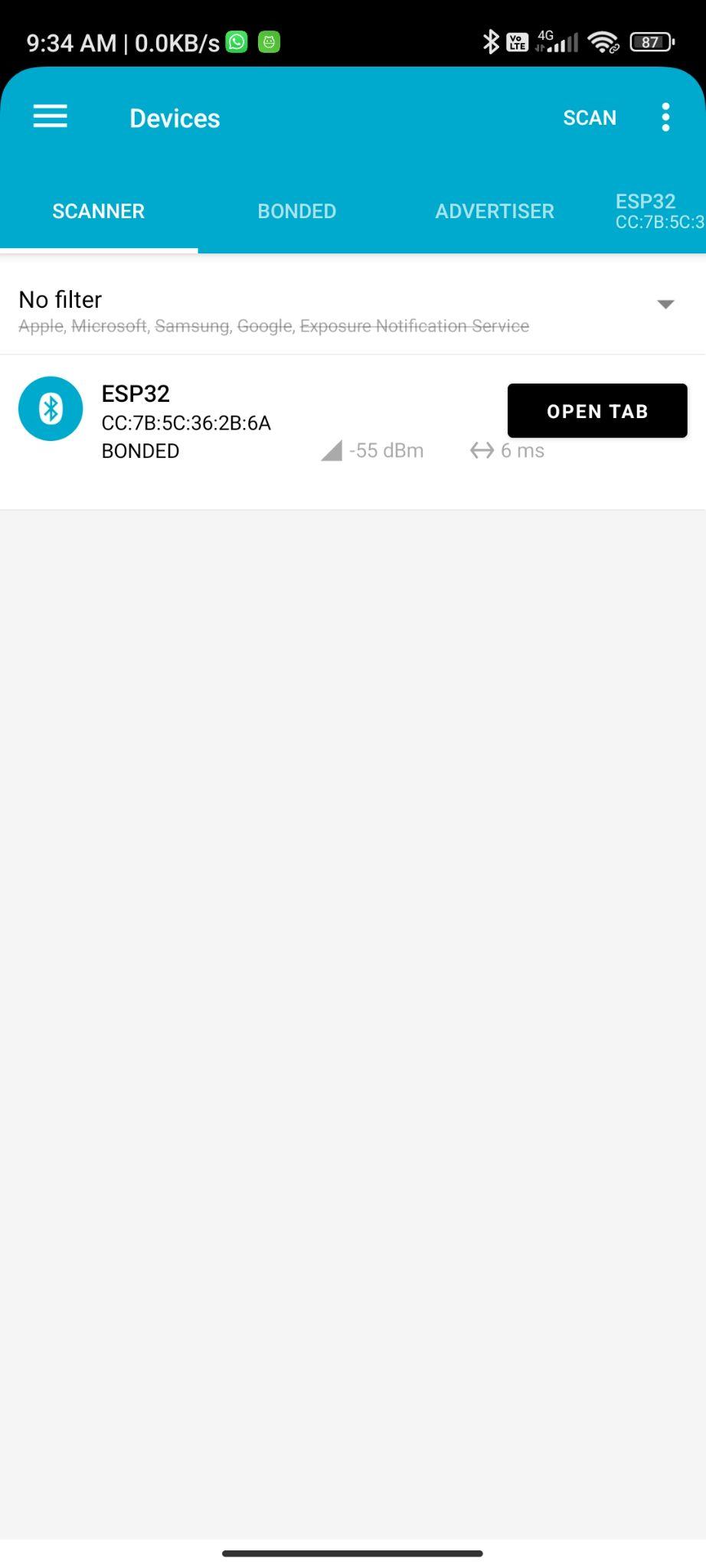
Serial.print(" F, Hum: ");

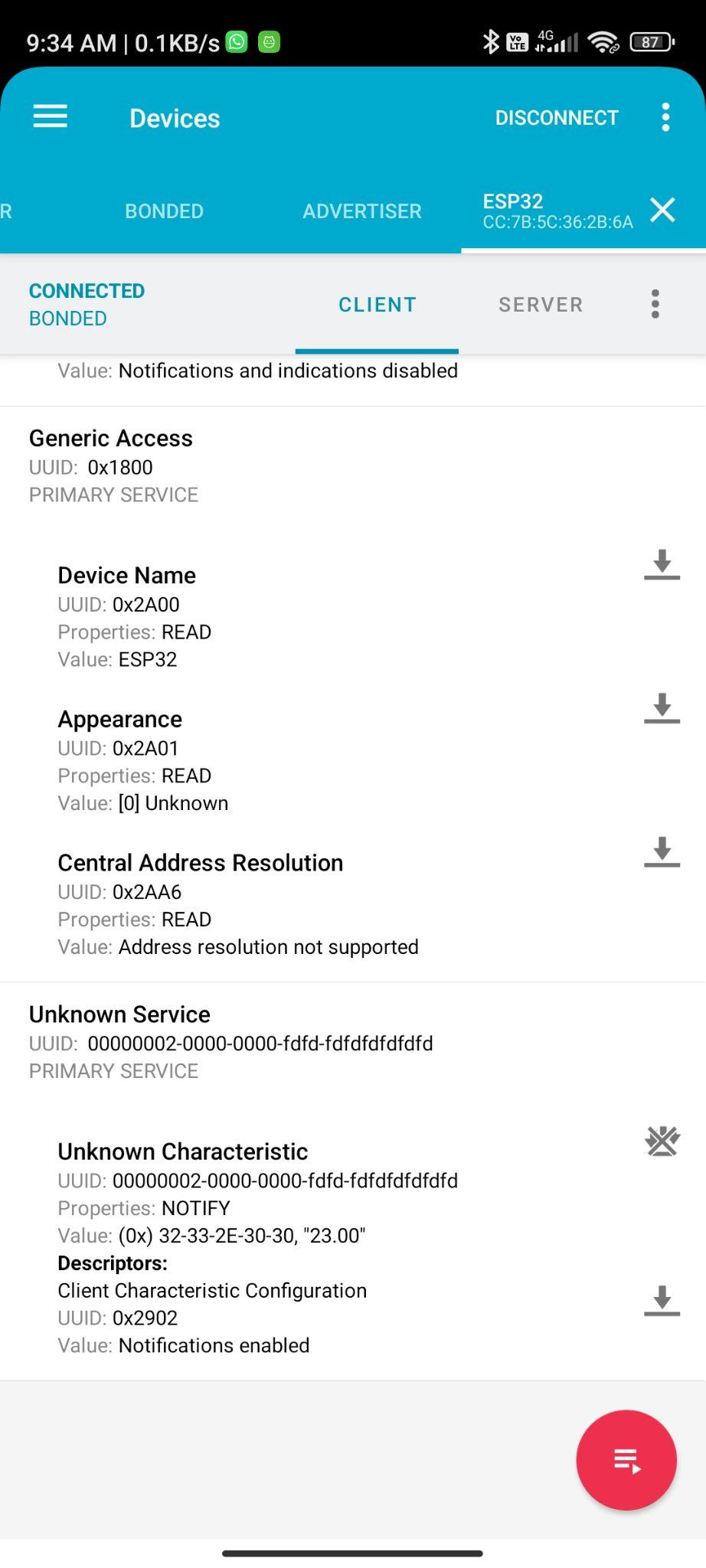
Serial.print(hu);

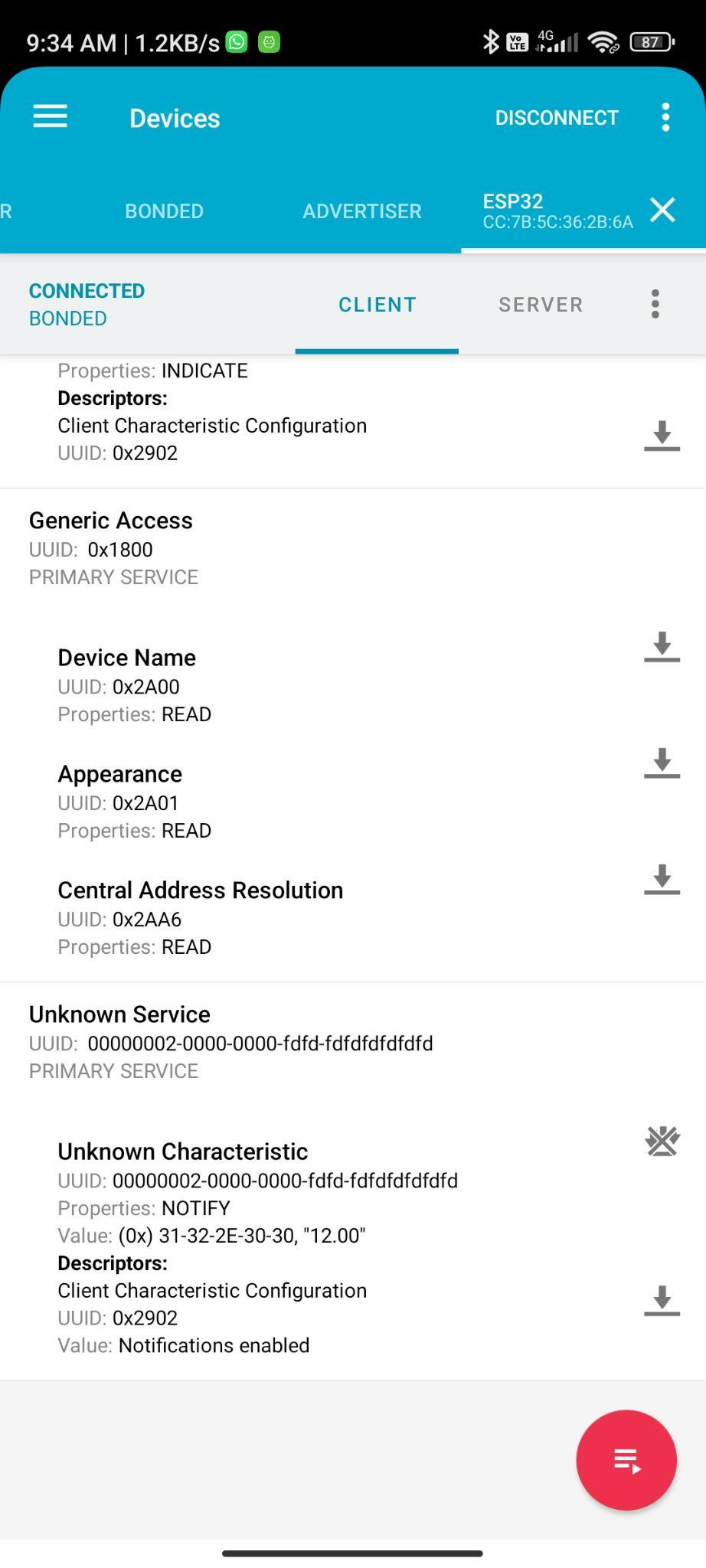
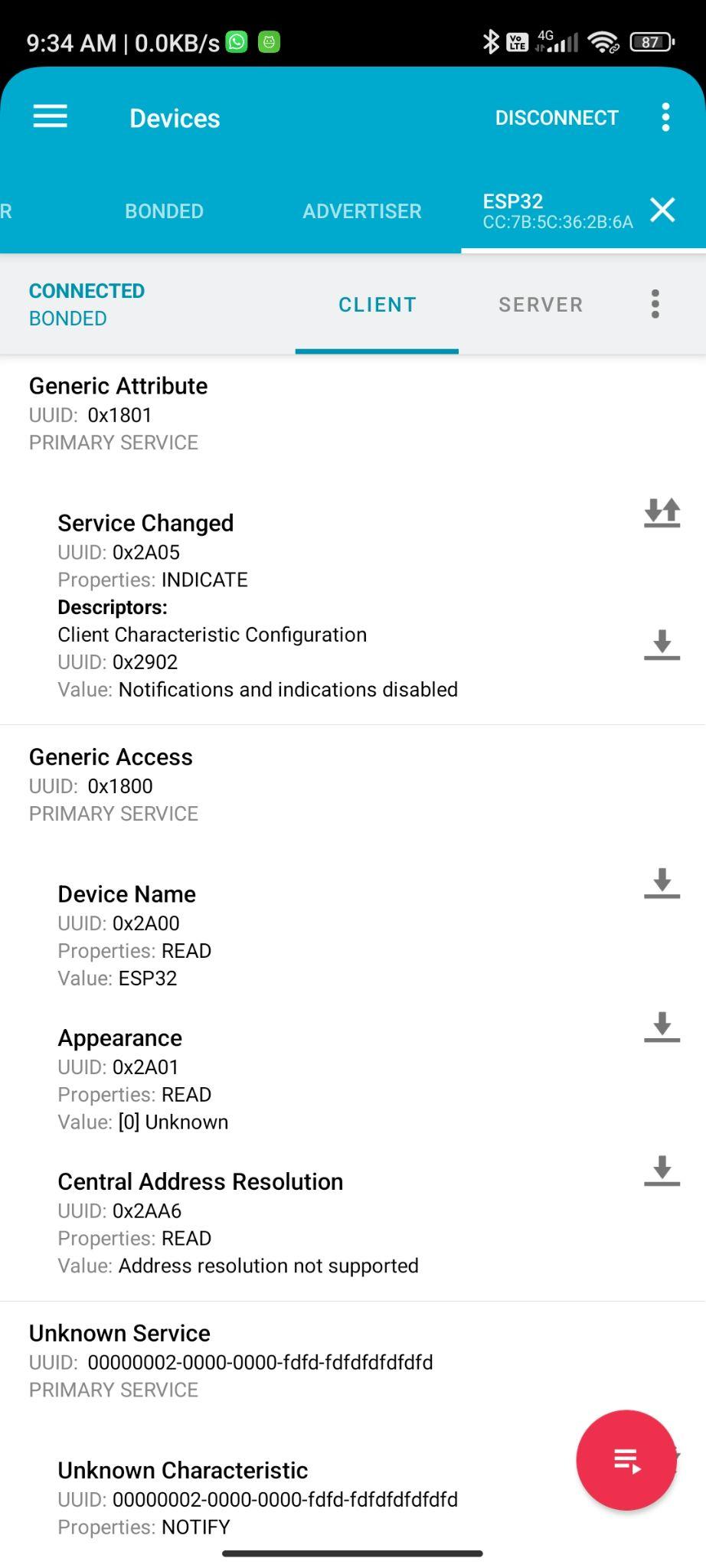
Serial.println("%");

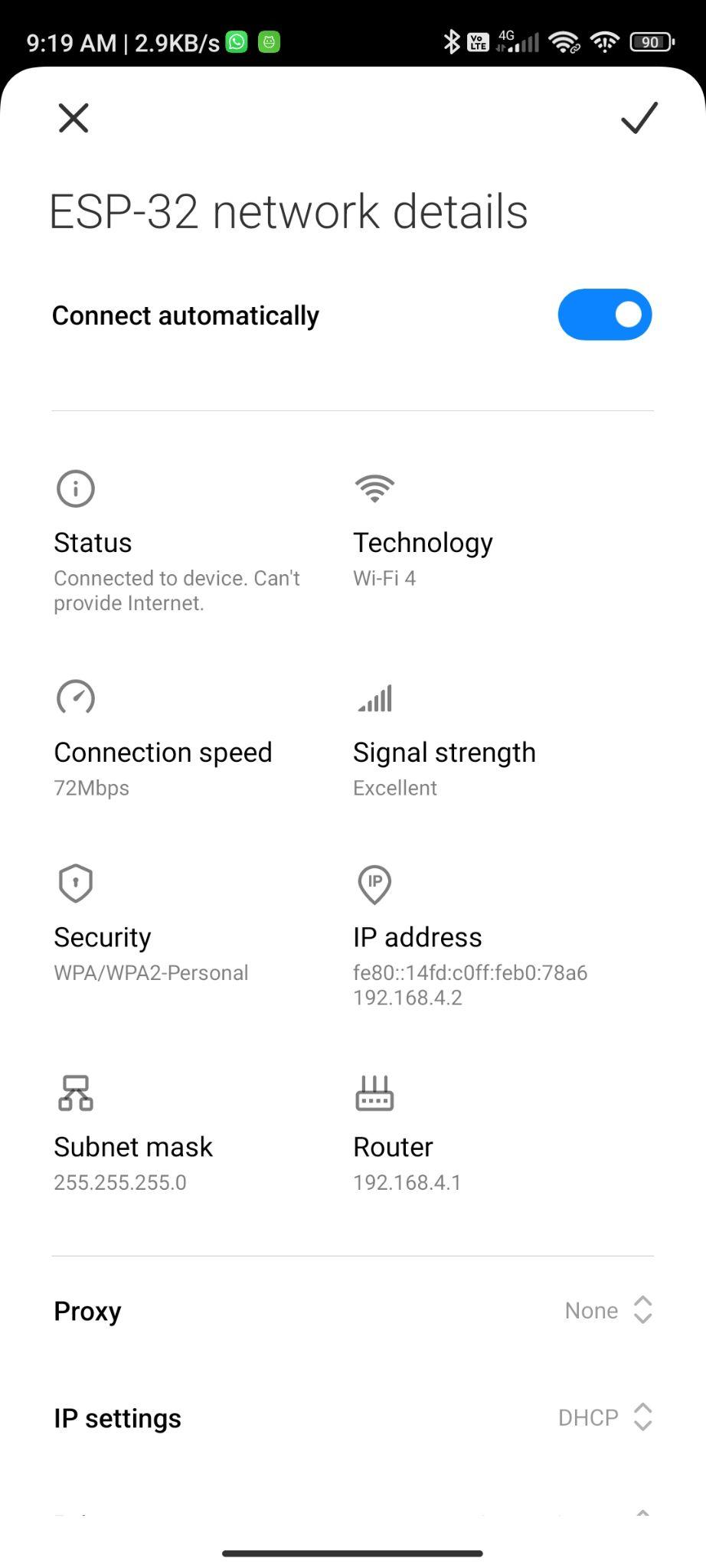
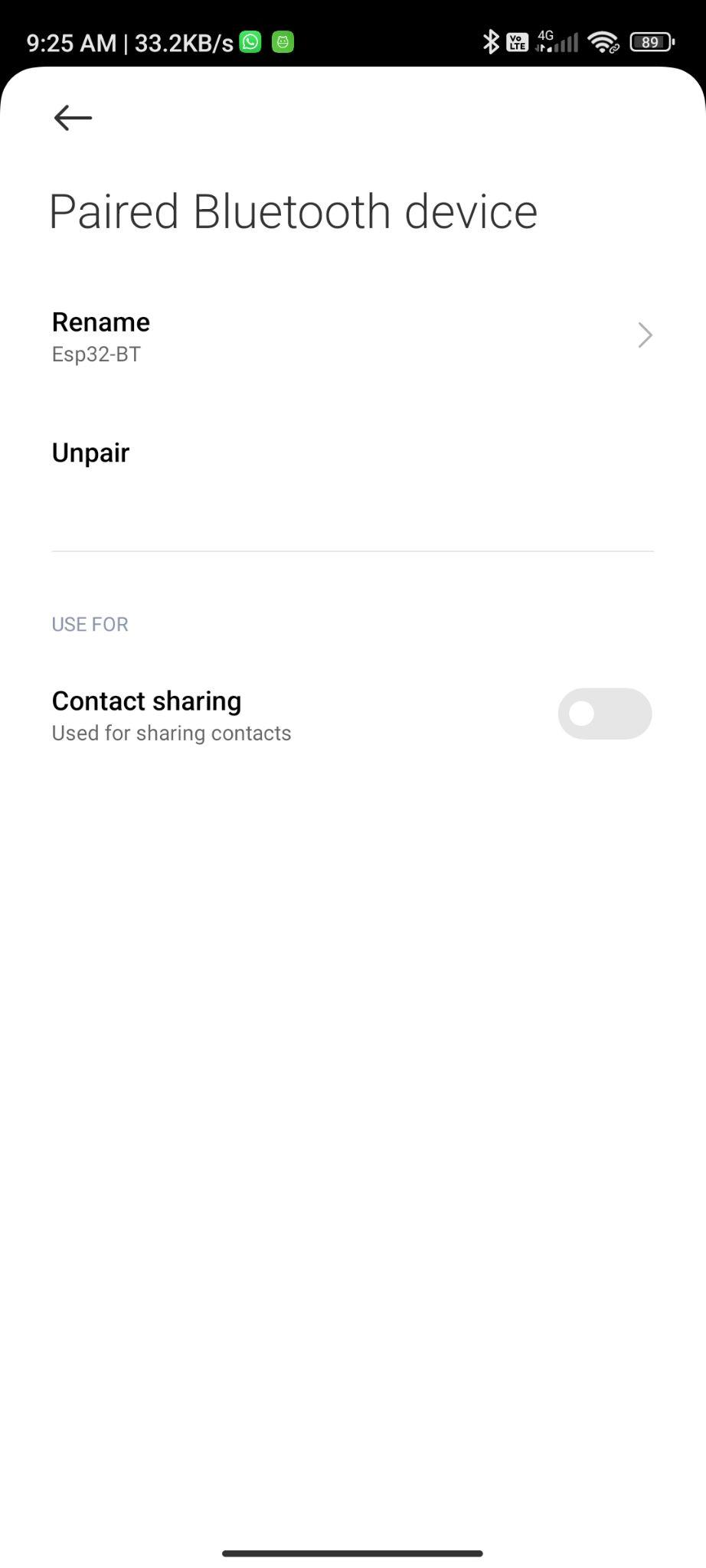
}

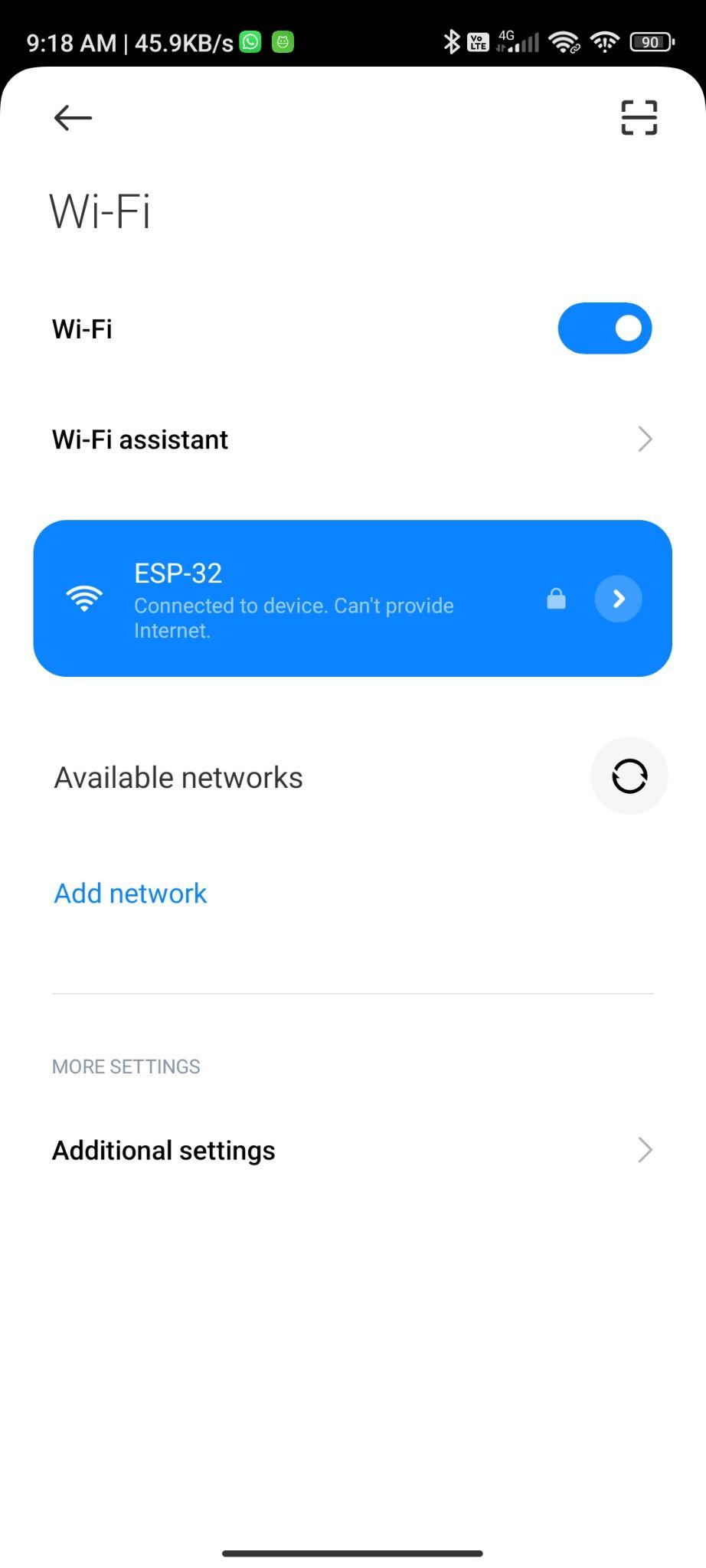
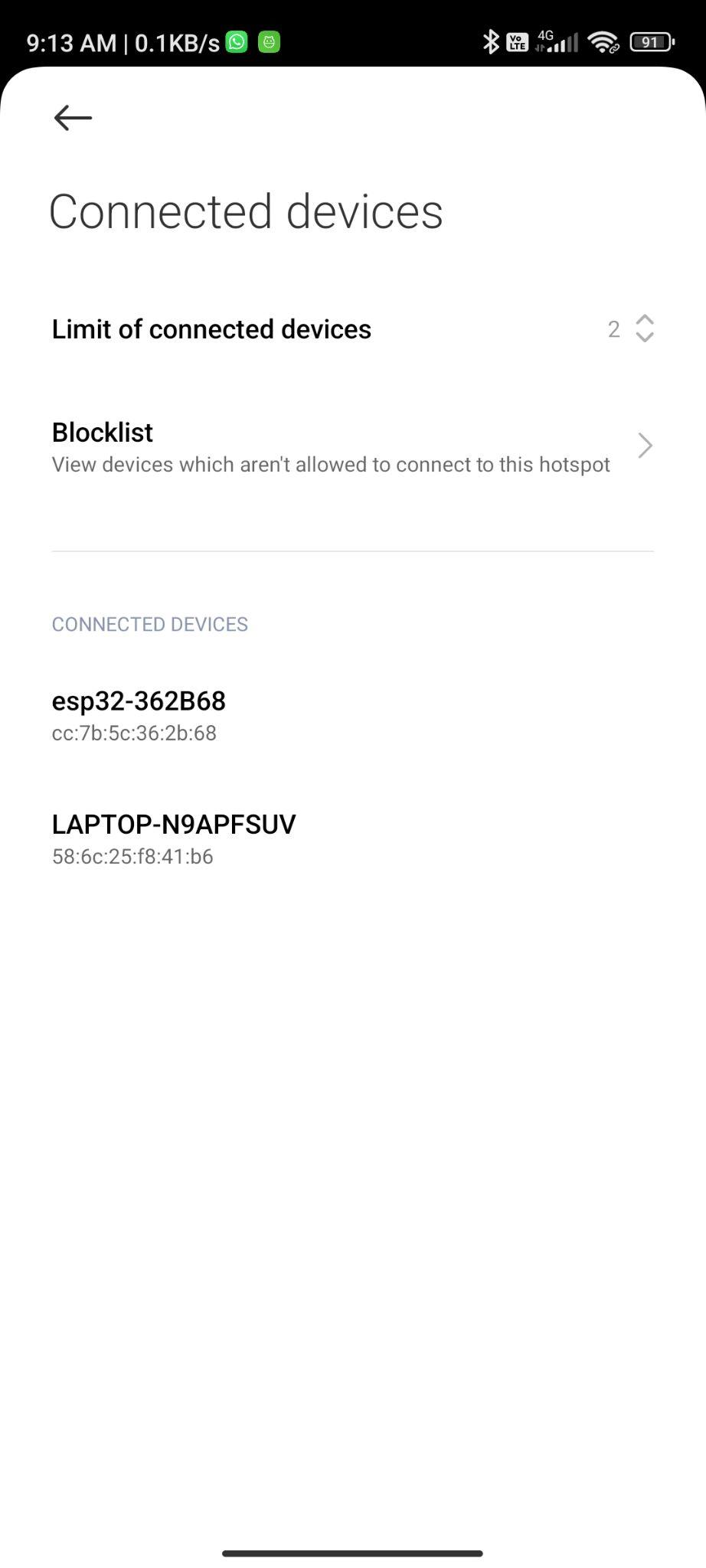
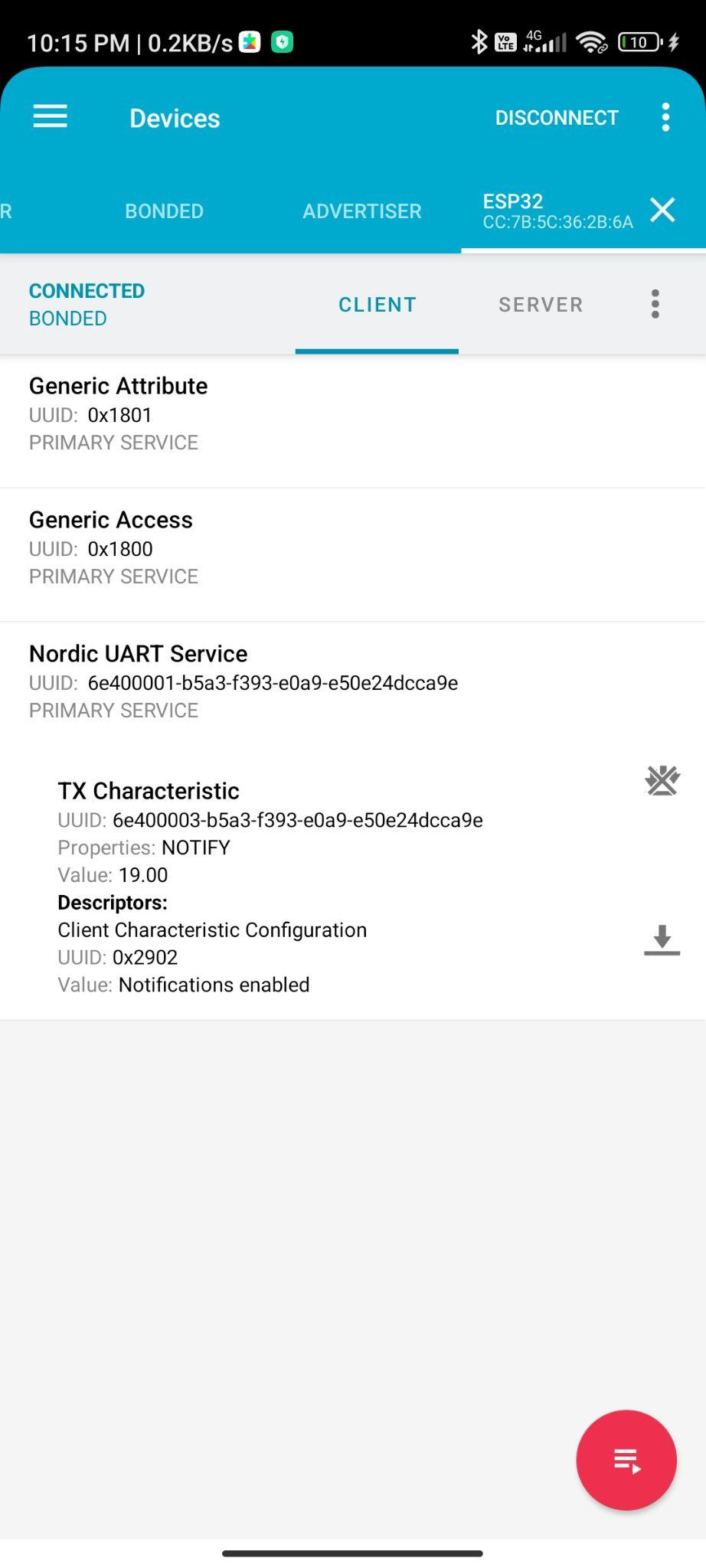
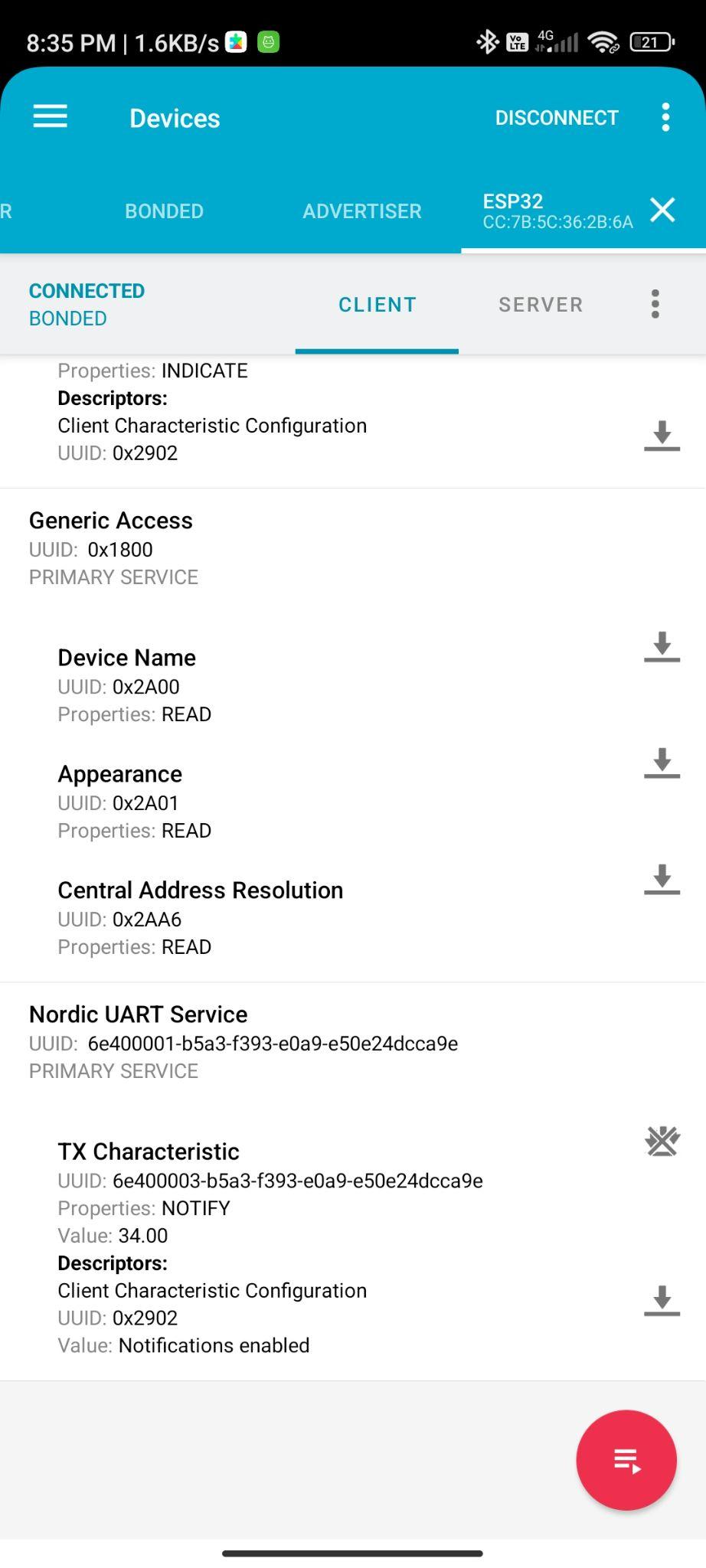
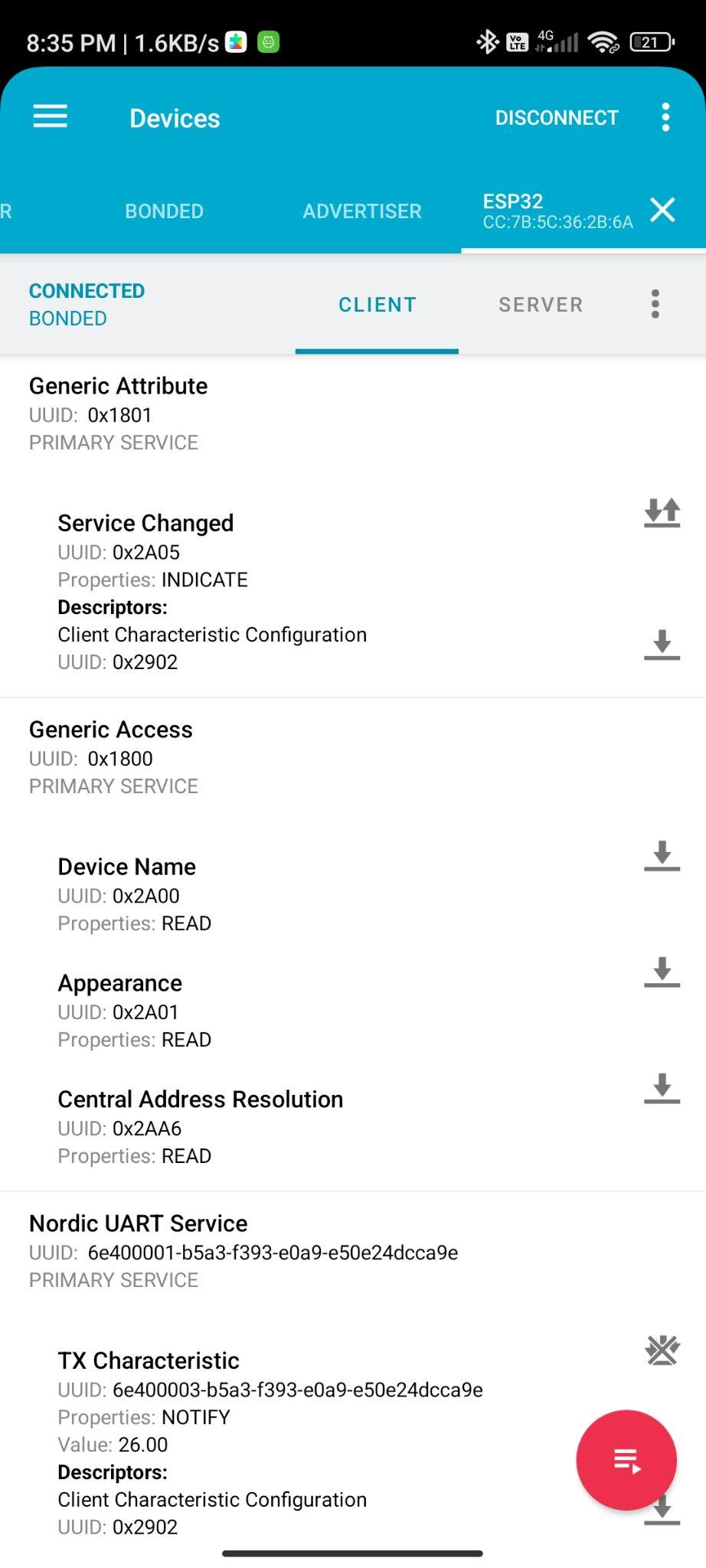
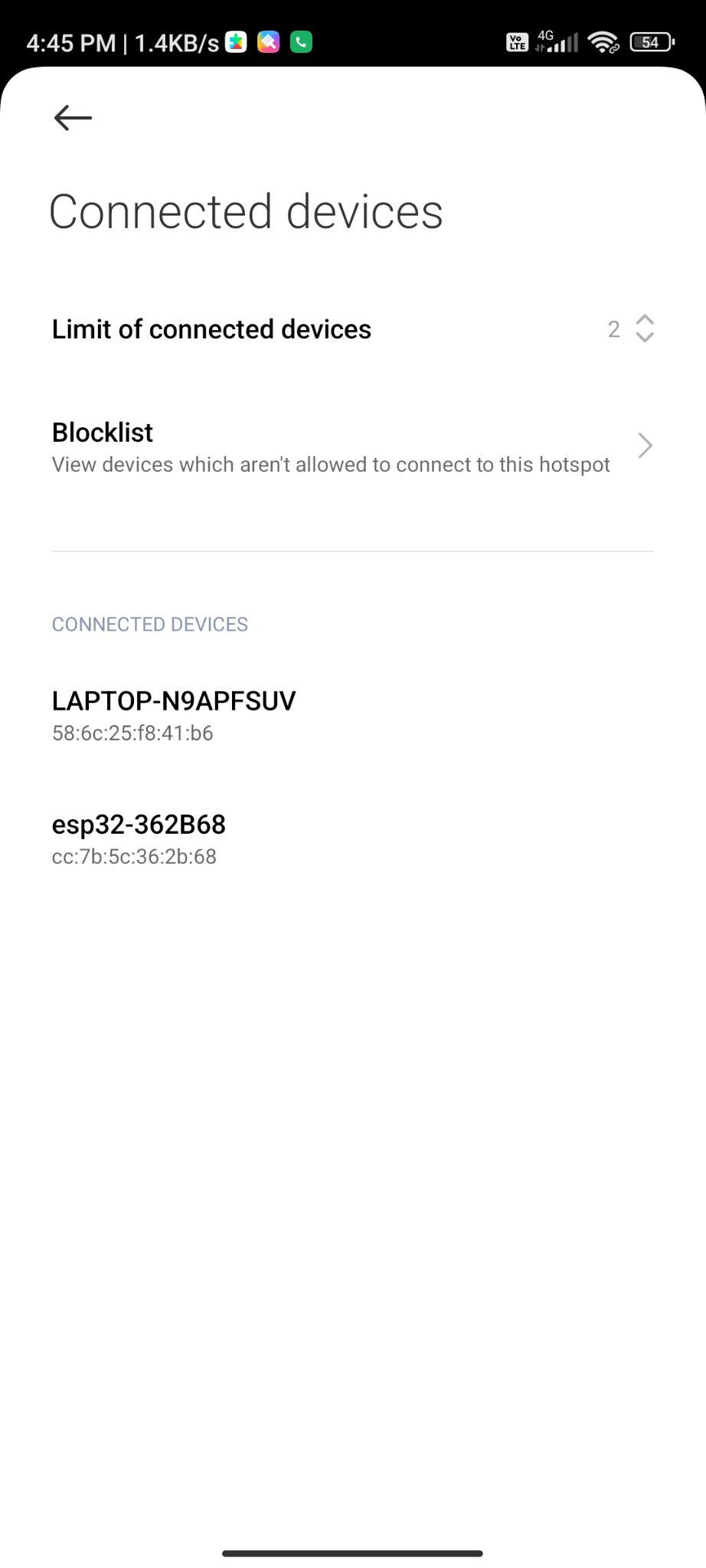
**Output, Screenshot & Videos**

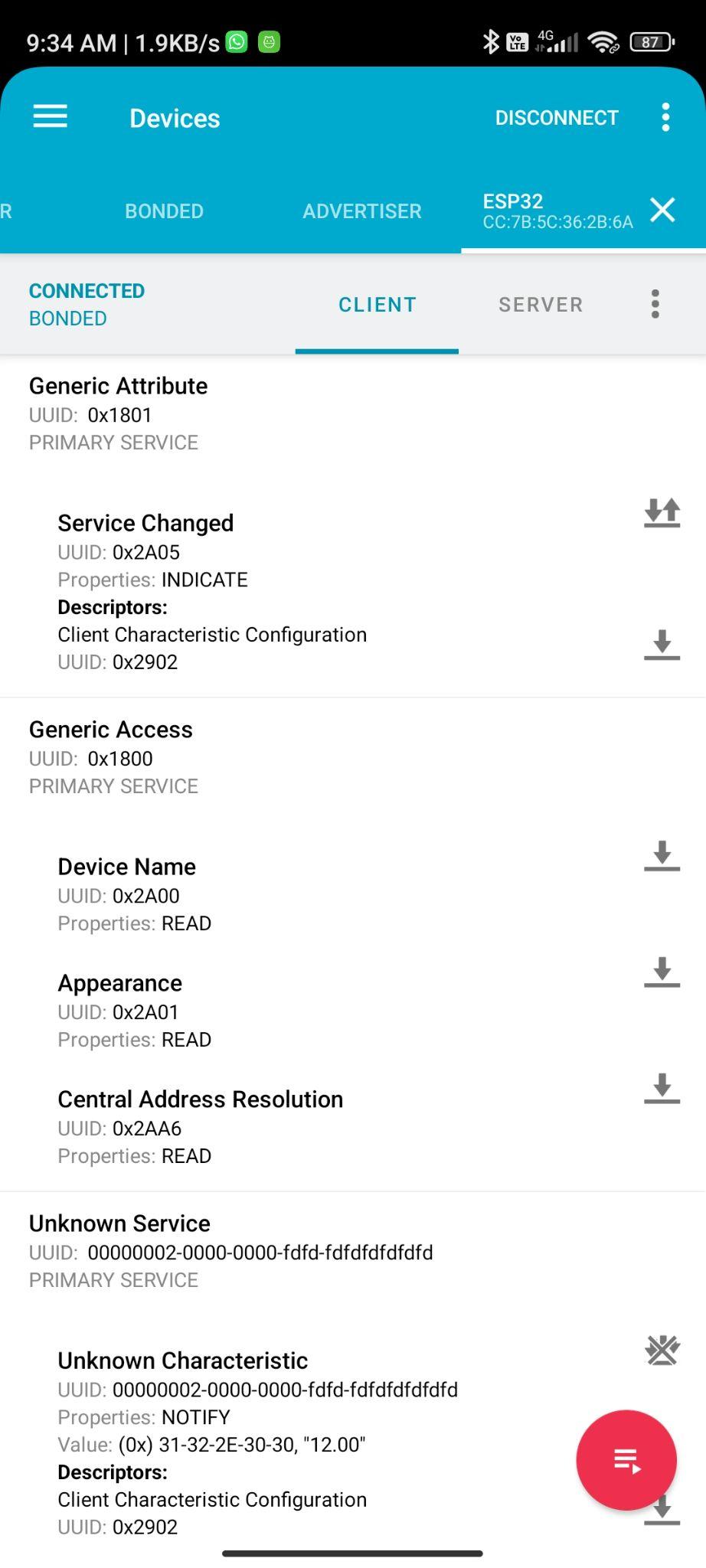


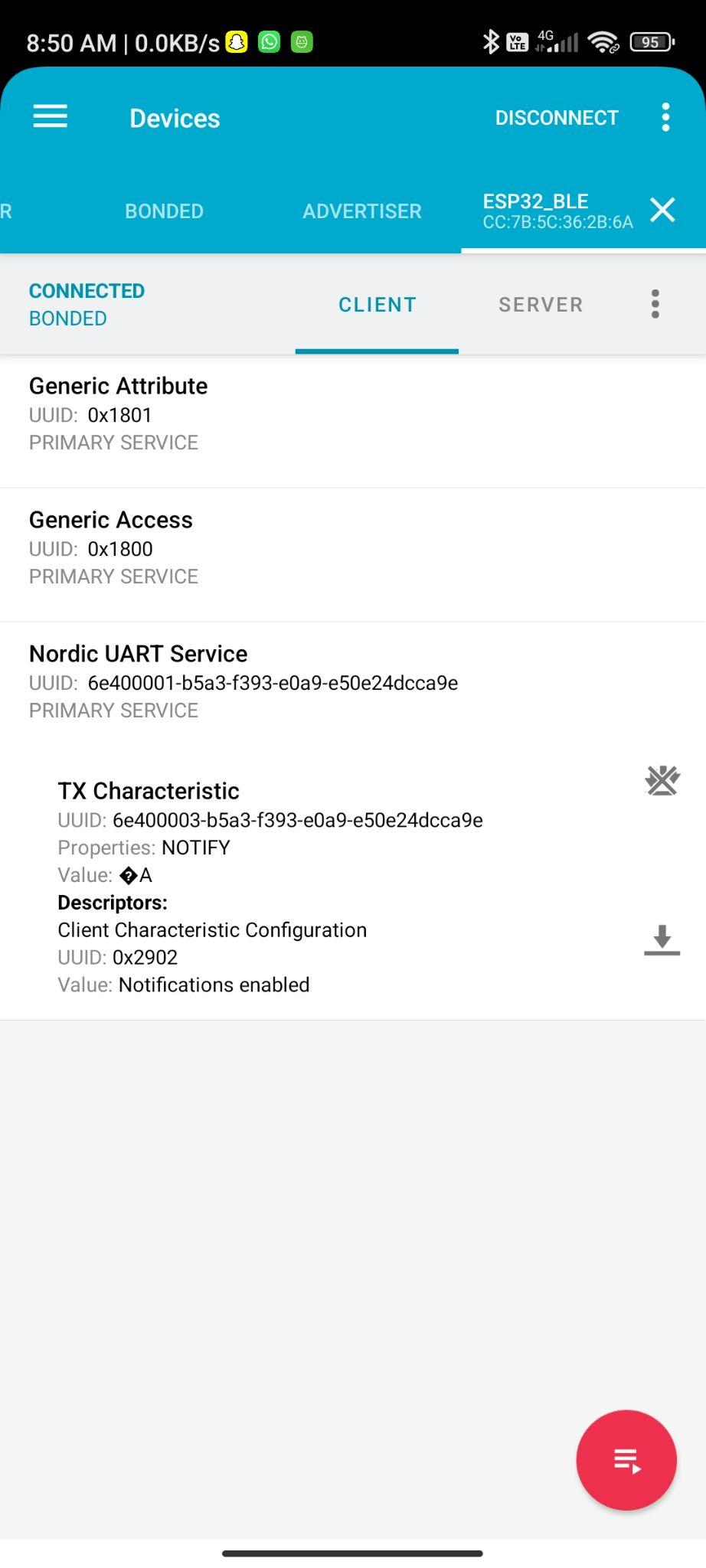


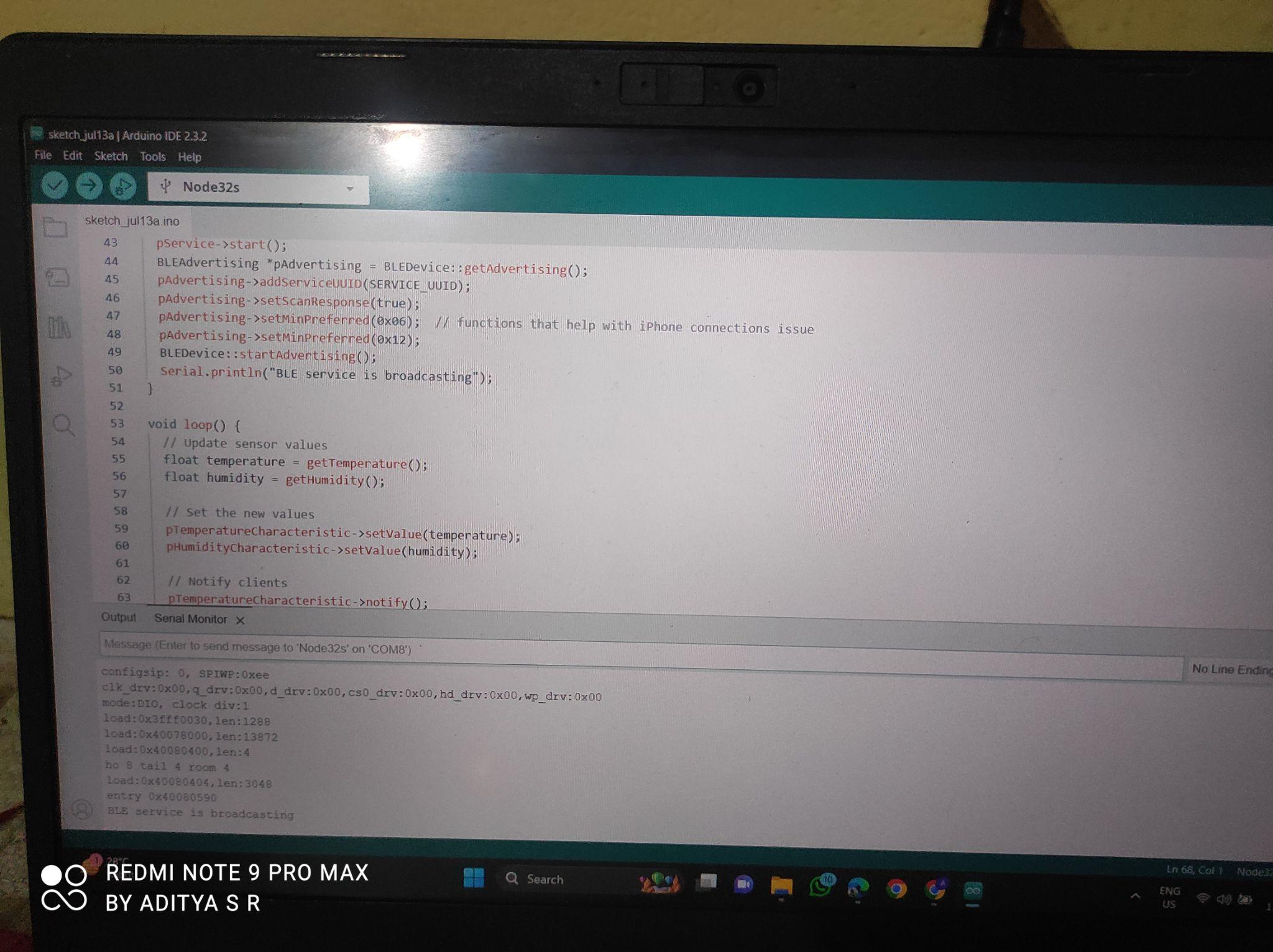
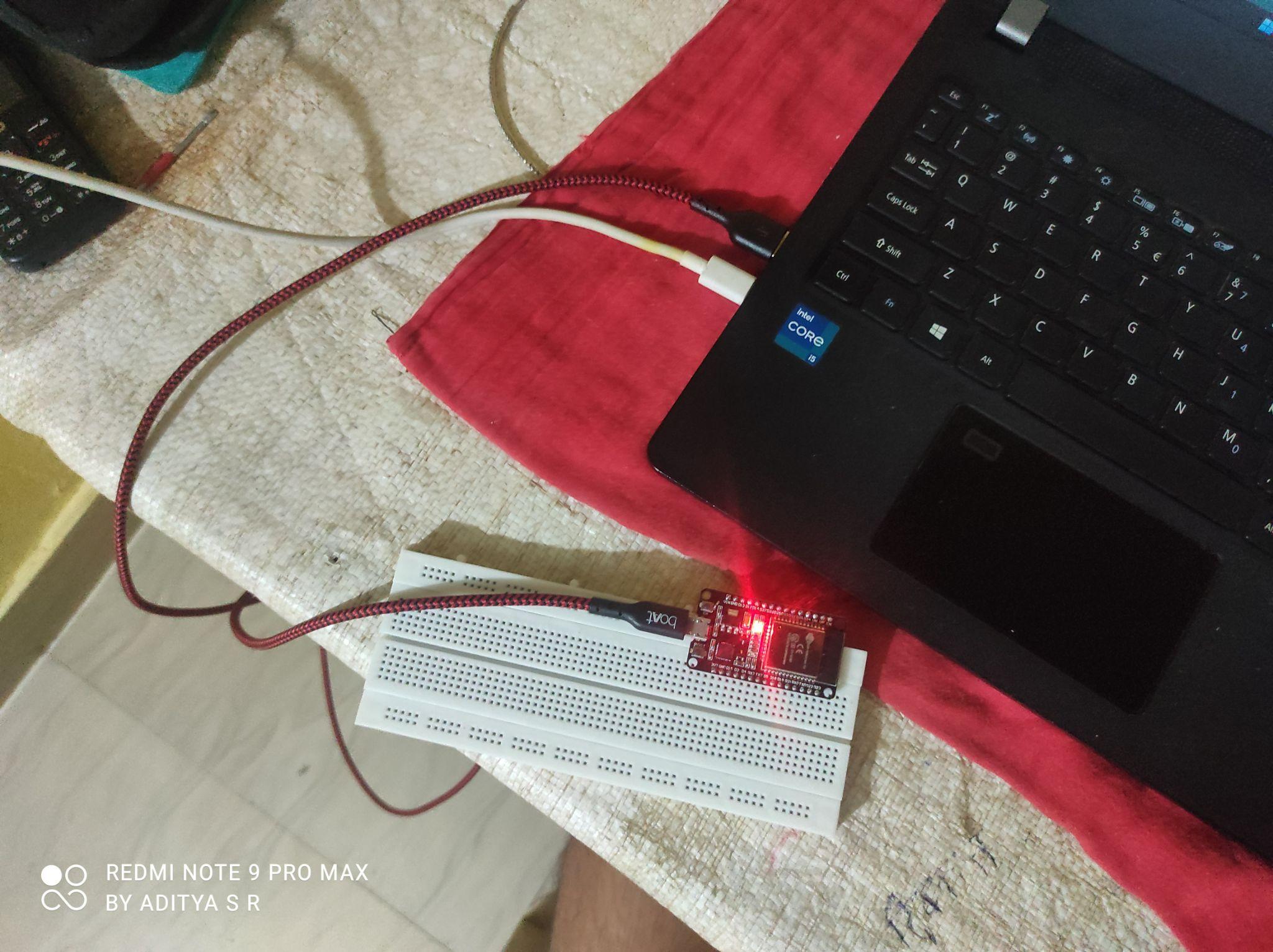
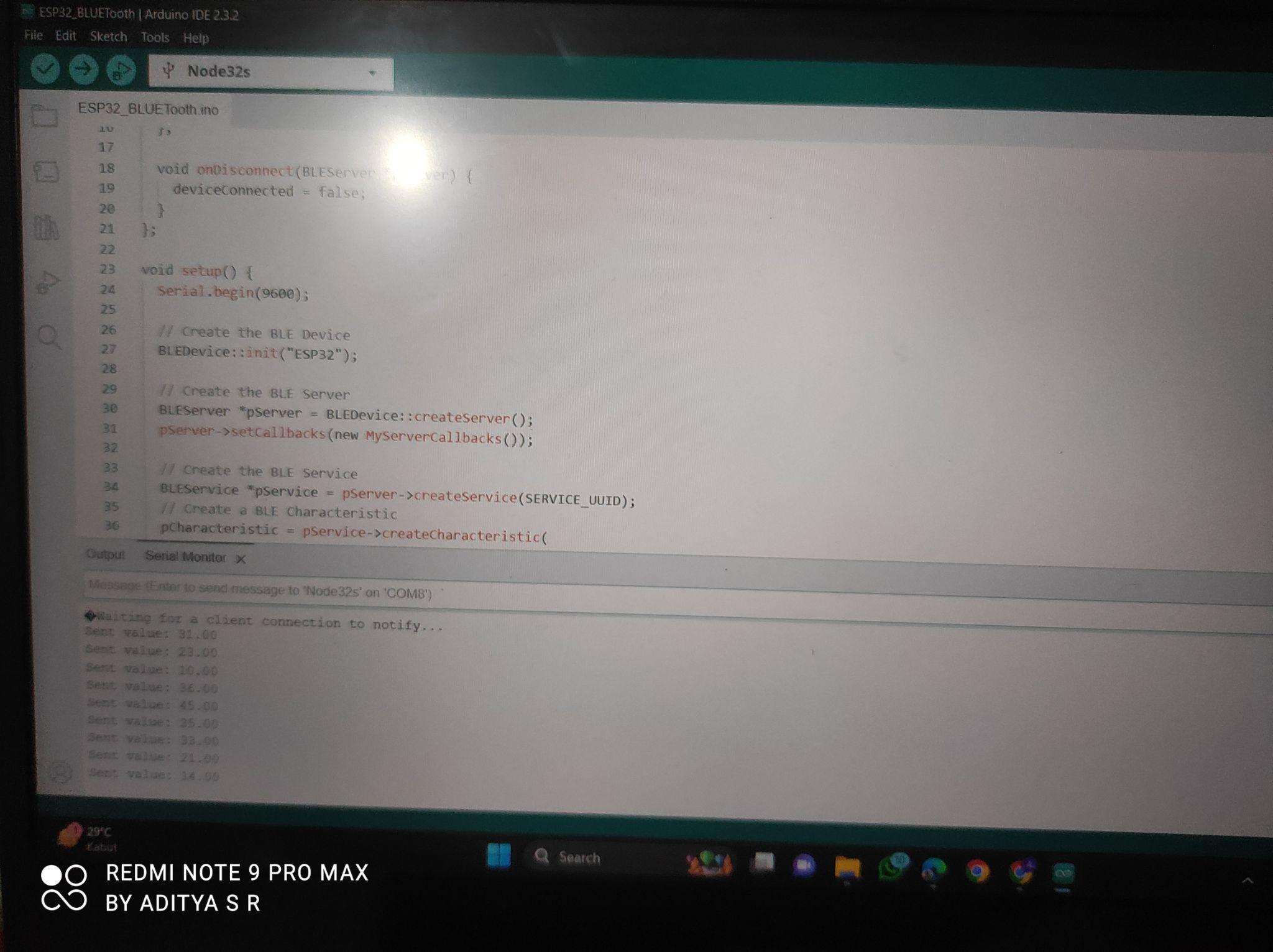
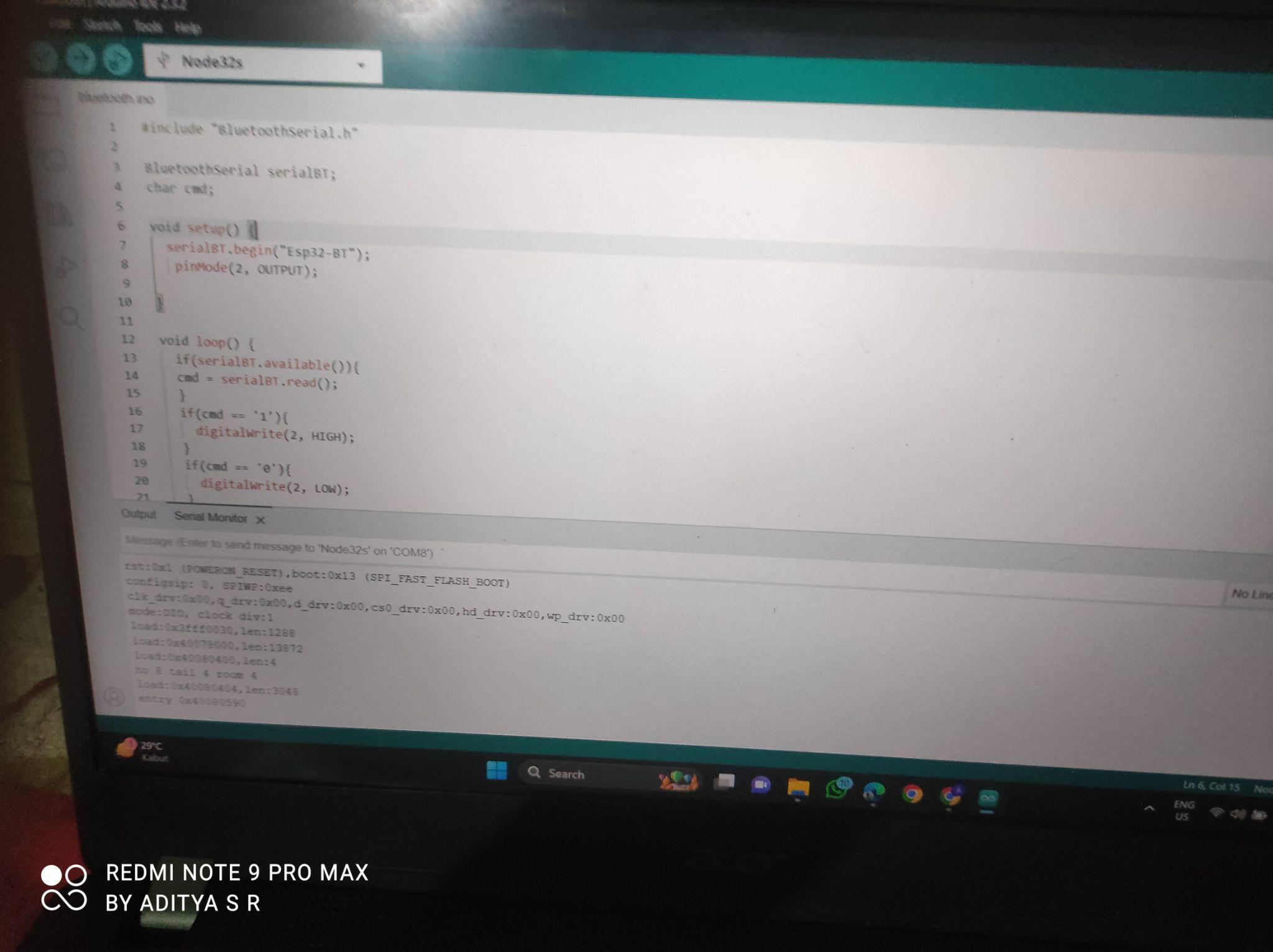
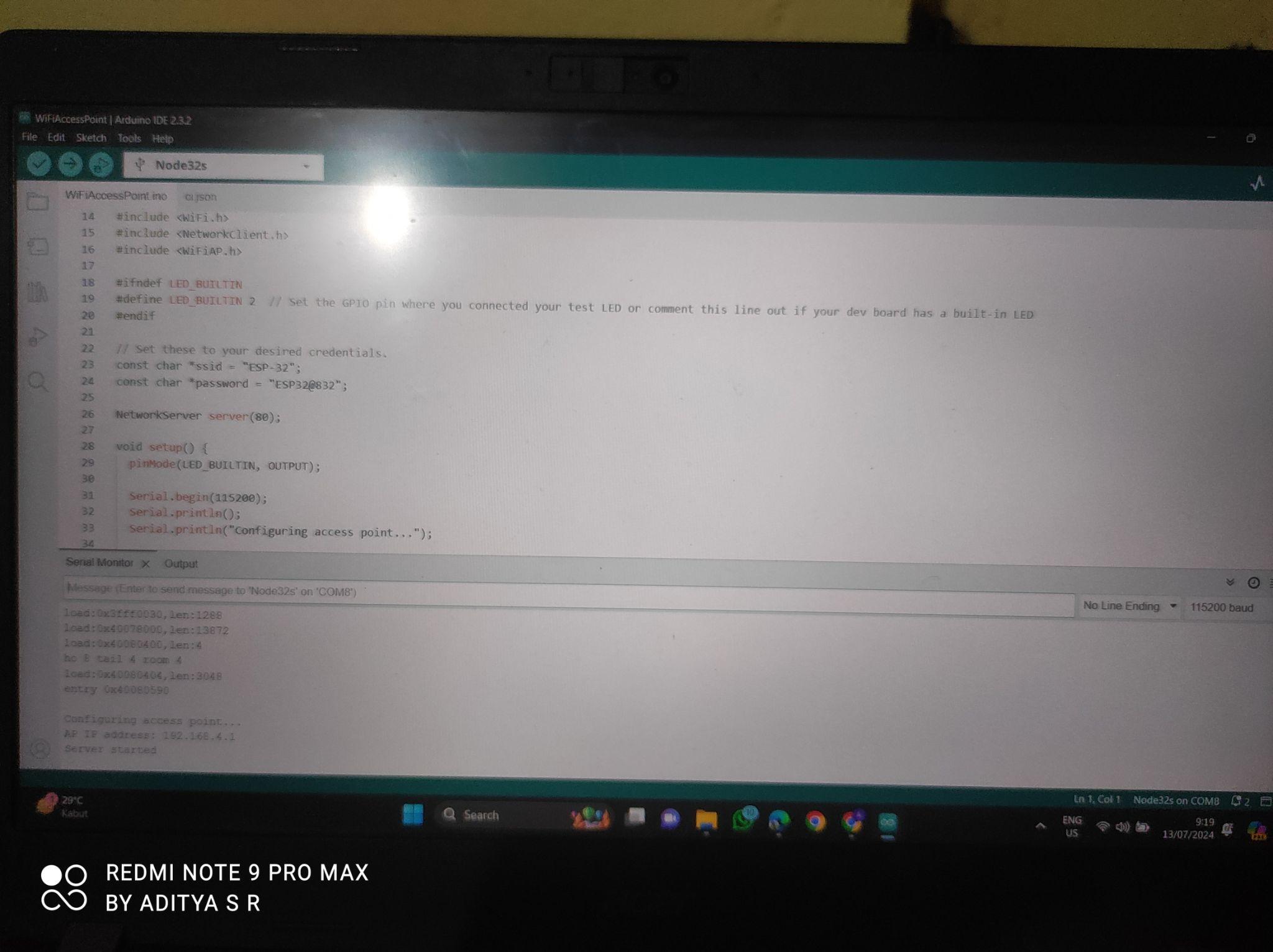
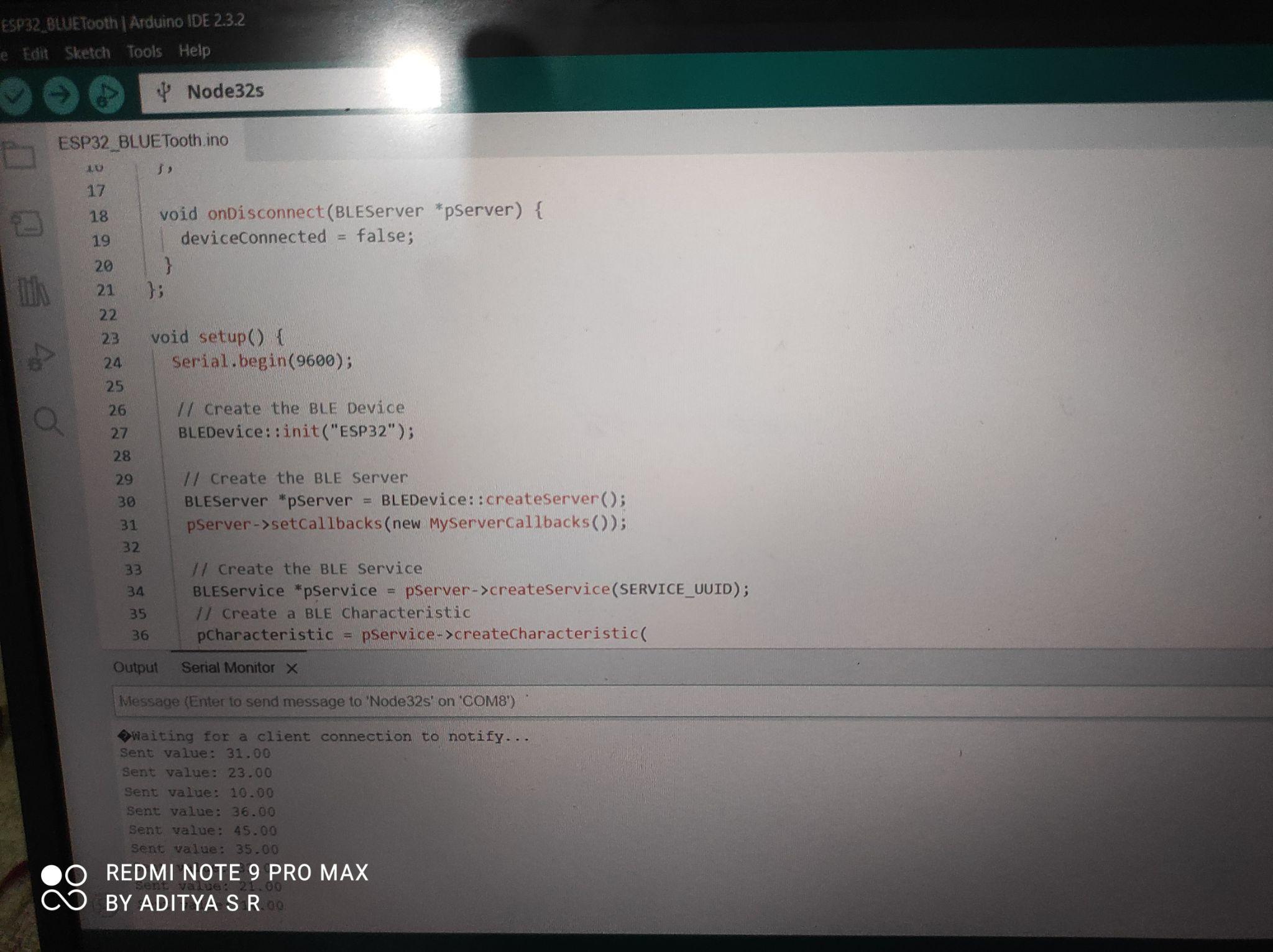


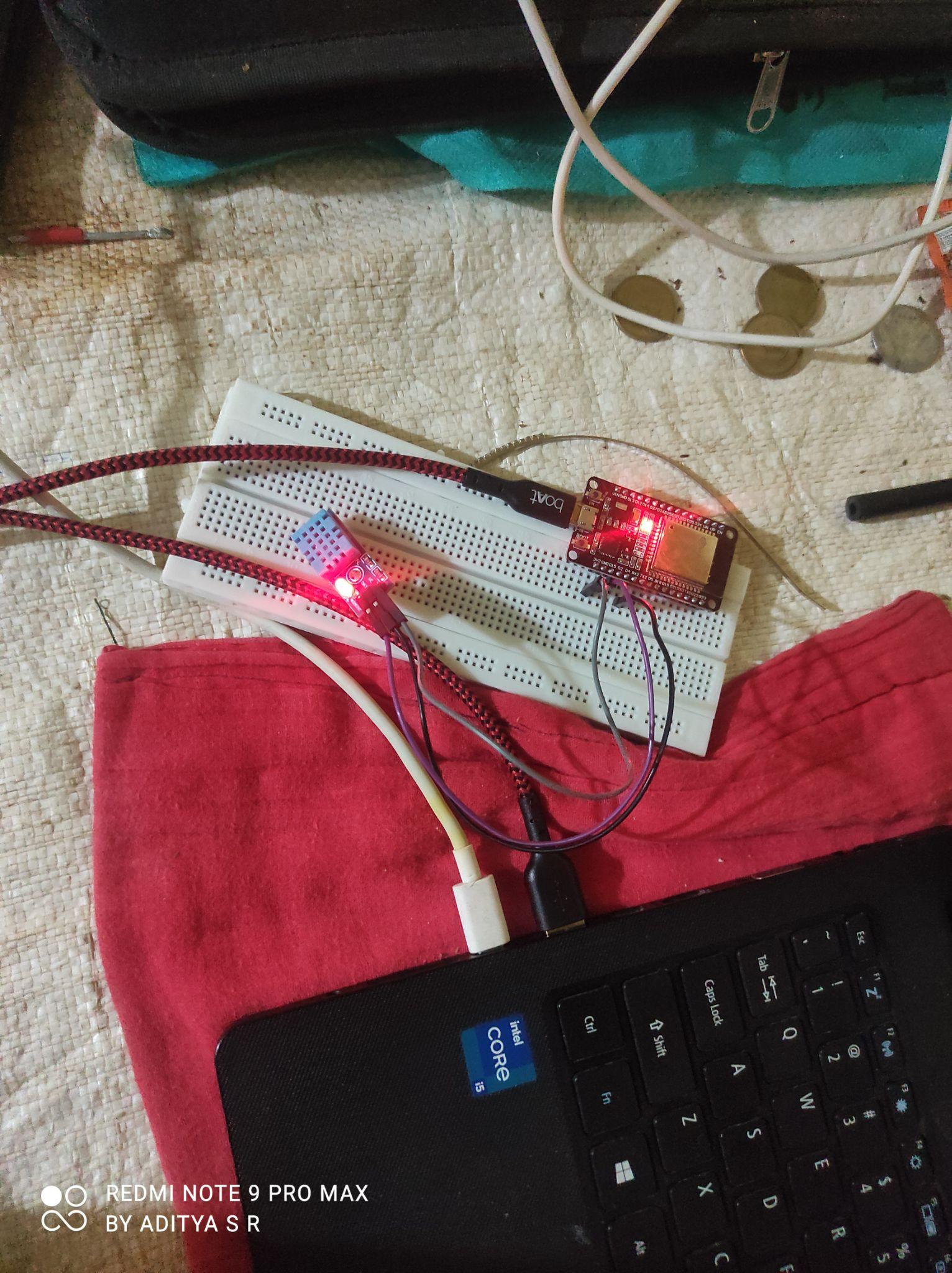


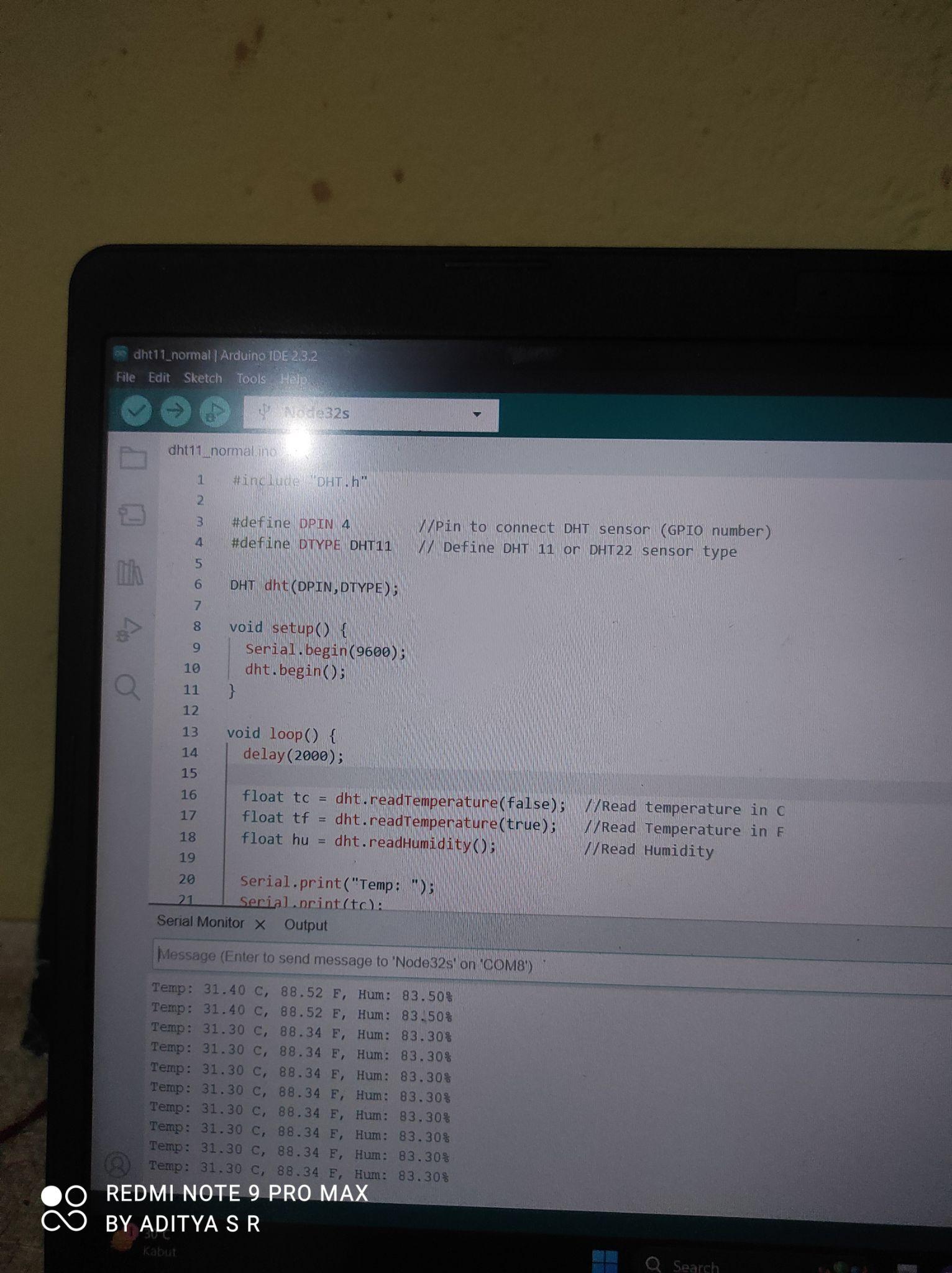












**Conclusion**

This project demonstrates the effective use of the ESP32 development board to broadcast temperature and humidity data over Bluetooth Low Energy (BLE). By leveraging the ESP32's built-in BLE capabilities and interfacing with the DHT11 sensor, we created a simple yet powerful system for real-time environmental monitoring. The project not only highlights the practicality and versatility of the ESP32 but also showcases the ease of integrating standard sensors to create functional IoT solutions.

### **Key Takeaways:**

1. **BLE Implementation**: Successfully implemented BLE service broadcasting using the ESP32, making the temperature and humidity data available to BLE clients.
2. **Sensor Integration**: Demonstrated the integration of the DHT11 sensor to provide real-time environmental data.
3. **Data Accessibility**: Enabled read and notify operations for BLE characteristics, ensuring that the data is dynamically updated and accessible via the nRF-Connect app.
4. **Practical Applications**: Showcased how such a setup can be used in various real-world applications, including smart homes, wearable technology, and environmental monitoring systems.

### **Future Work:**

There are several ways to expand and enhance this project:

* **Additional Sensors**: Integrate more sensors to broadcast additional environmental data (e.g., air quality, light intensity).
* **WiFi Integration**: Use the ESP32's WiFi capabilities to send the collected data to a cloud server for remote monitoring and analysis.
* **Power Optimization**: Implement power-saving techniques to extend the battery life of the device, making it more suitable for long-term deployment.
* **User Interface**: Develop a mobile or web-based interface for more intuitive interaction with the data and control over the device.

### **Extra Credits:**

For those seeking extra credits, implementing a feature to transfer WiFi credentials over Bluetooth to connect the ESP32 to a WiFi network can further enhance the utility of the device. This enables seamless integration with IoT ecosystems and expands the range of applications.

In conclusion, this project provides a solid foundation for understanding and implementing BLE service broadcasting in IoT devices. The skills and knowledge gained through this project can be applied to a wide array of IoT solutions, driving innovation and enhancing connectivity in everyday applications.