



VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Hoang Hieu

DESIGN AND IMPLEMENTATION OF A BLE GATEWAY USING ESP32 CHIPSET

Technology and Communication
2023

ABSTRACT

Author	Hoang Hieu
Title	Design and Implemetation of a BLE Gateway Using ESP32 Chipset.
Year	2023
Language	English
Pages	65
Name of Supervisor	Jukka Matila

This thesis presents the design and implementation of a custom ESP32-based gateway that integrates Bluetooth Low Energy (BLE) and Long Range (LoRa) technologies for IoT applications.

The background for this study includes the in-creasing demand for low-power, wide-area networks (LPWAN) to facilitate seamless data collection and transmission for connected devices in various sectors, such as smart cities, agriculture, and industrial automation.

The aim of this thesis was to develop a cost-effective and versatile gateway that combines the advantages of both BLE and LoRa communication, leveraging the capabilities of the ESP32 microcontroller. The material and methods used in this project involve hardware selection, schematic design, PCB layout, firmware development.

The results demonstrate the development of the custom ESP32 gateway, enabling efficient data collection from BLE devices, and transmitting the aggregated data to an MQTT server or a LoRaWAN cloud. In conclusion, the custom ESP32 gateway provides a promising solution for IoT deployments, offering a tailored and flexible approach to data communication and management.

Keywords	Bluetooth Low Energy, Internet of Things, ESP32 Microcontroller, MQTT Protocol, and PCB design.
----------	---

CONTENTS

ABSTRACT.....	2
LIST OF FIGURES AND TABLES.....	6
LIST OF LISTINGS.....	8
1 INTRODUCTION	10
1.1 The objectives of the Thesis.....	10
1.2 Gateway Topology	11
2 THEORETICAL BACKGROUND	13
2.1 Bluetooth IOT Gateway.....	13
2.2 Bluetooth Low Energy (BLE).....	14
2.3 LoRa Technology	16
2.4 MQTT Protocol.....	18
3 PROOF-OF-CONCEPT (POC) PROTOTYPES	20
3.1 Prototyping Process	20
3.2 Prototyping Hardware Components.....	21
3.2.1 ESP32 Development Kit.....	21
3.2.2 Ethernet Module	23
3.2.3 LoRa Module	24
3.3 Connections Table.....	26
4 PCB DEVELOPMENT	28
4.1 PCB Block Diagram	28
4.2 Schematic Design	29
4.2.1 Ethernet LAN8720A Chip	30
4.2.2 USB-C to UART CP2104 Chip	33
4.2.3 Power Management.....	34
4.2.4 ESP32 MCU and Lora Module	35
4.2.5 Peripherals	36
4.3 PCB Layout	37
4.4 PCB Assembled and Testing.....	39

5	FIRMWARE DEVELOPMENT	43
5.1	BLE Beacon Gateway Setup	43
5.2	Firmware	45
5.2.1	Main Code (main.ino).....	45
5.2.2	MQTT settings (mqtt.h).....	49
5.2.3	LoRa Settings (lora.h)	52
5.2.4	Other Settings (setting.h).....	55
5.3	Results	56
5.3.1	MQTT Data and Vizualization.....	56
5.3.2	Lora Server Data.....	59
6	CONCLUSIONS.....	61
	REFERENCES	63

LIST OF FIGURES AND TABLES

Figure 1. Gateway design topology.....	11
Figure 2. BLE Gateway topology /1/	13
Figure 3. Configurations Bluetooth Smart /3/	15
Figure 4. MQTT Architecture /5/.....	18
Figure 5. Proof-of-concept setup	20
Figure 6. ESP32 DevKitC-32U	21
Figure 7. ESP32-WROOM-32U module	23
Figure 8. LAN8720 ETH module	23
Figure 9. Wio-E5 mini development board.....	24
Figure 10. LoRa-E5 module /7/	26
Figure 11. PCB Block Diagram	29
Figure 12. LAN8720A system block /9/	30
Figure 13. LAN8720A power	31
Figure 14. LAN8720A Transceiver	31
Figure 15. REF CLK for LAN8720A	32
Figure 16. RJ45 to LAN8720A.....	32
Figure 17. CP2104 chip.....	33
Figure 18. Power switch	34
Figure 19. Circuit power	34
Figure 20. ESP32 Pin IN/OUT.....	35
Figure 21. LoRa circuit.....	35
Figure 22. UART switch	36
Figure 23. Peripherals	36
Figure 24. Design rules.....	37
Figure 25. PCB layers.....	37
Figure 26. PCB layout	38
Figure 27. PCB 3D viewer	38
Figure 28. PCB unassembled	39

Figure 29. PCB assembled top.....	39
Figure 30. PCB assembled front	40
Figure 31. PCB problem 1.....	40
Figure 32. PCB problem 2.....	41
Figure 33. Fixed PCB problem 2 layout	41
Figure 34. Fixed PCB problem 2 back.....	42
Figure 35. Fully functional PCB.....	42
Figure 36. BLE beacons Gateway setup	44
Figure 37. MQTT data collected	57
Figure 38. Node-Red Dashboard	58
Figure 39. LoRa server data.....	59
Table 1 LoRa Specifications /4/	17
Table 2. ESP32-WROOM-32U Specifications /6/	22
Table 3. Wio-E5 specifications /7/	25
Table 4. ESP32 to LAN8720A Module.....	26
Table 5. ESP32 to Wio-E5	27

LIST OF LISTINGS

Listing 1. Libraries for main.ino	45
Listing 2. Scan times	45
Listing 3. Decode beacons	48
Listing 4. BLE setups	48
Listing 5. Main loop	49
Listing 6. MQTT libraries.....	49
Listing 7: ETH and MQTT connect	50
Listing 8. MQTT publish.....	50
Listing 9. Send to MQTT	51
Listing 10. Lora settings	52
Listing 11. String to HEX	52
Listing 12. Lora iBeacon.....	53
Listing 13. Lora Eddystone beacon.....	54
Listing 14. Source code settings	55
Listing 15. Status LEDS settings	56
Listing 16. Decode Python Script.....	60

LIST OF ABBREVIATIONS AND ACRONYMS

BLE	Bluetooth Low Energy
LoRa	Long Range
IoT	Internet of Things
ESP32	Espressif Systems 32-bit Microcontroller
MQTT	Message Queuing Telemetry Transport
PCB	Printed Circuit Board
LPWAN	Low-Power Wide-Area Networks
UART	Universal Asynchronous Receiver/Transmitter
HW	Hardware
GPIO	General Purpose Input/Output
IDE	Integrated Development Environment
MCU	Microcontroller Unit
SMT	Surface Mount Technology
SWD	Serial Wire Debug
SoC	System on Chip
SW	Software
RSSI	Received Signal Strength Indicator

1 INTRODUCTION

1.1 The objectives of the Thesis

BLE (Bluetooth Low Energy) and LoRa (Long Range) are two popular wireless technologies employed to address the connectivity requirements of IoT devices due to their low power consumption and extended range. The integration of both BLE and LoRa technologies into a single gateway can significantly enhance the versatility of IoT networks, allowing for seamless data collection and transmission.

BLE technology is designed for short-range, low-power communication, typically suited for IoT devices, such as wearables, smart home appliances, and health monitoring equipment. On the other hand, LoRa technology is intended for low-power, wide-area networks (LPWAN), offering long-range communication for IoT devices in remote or challenging environments.

This thesis aims to design and implement a custom ESP32 gateway using the ESP32 microcontroller, which offers a low-cost, flexible, and powerful platform for IoT applications. The custom gateway will combine the advantages of both BLE and LoRa communication, enabling efficient data collection from BLE devices and LoRa nodes, and transmitting the aggregated data to an MQTT server. This work includes hardware selection, schematic design, PCB layout, firmware development, and enclosure design, ultimately providing a tailored and flexible approach to data communication and management for IoT deployments.

The main objectives of this thesis are as follows:

- Design and prototype a gateway using the ESP32 microcontroller.
- Develop the schematic and PCB layout for the gateway, integrating the required hardware components.
- Implement firmware for data collection, processing, and transmission to an MQTT server.

1.2 Gateway Topology

The primary use case for the custom ESP32 gateway is to collect data from nearby BLE devices and transmit it to cloud platforms using MQTT. The gateway serves as a central point of communication between the BLE devices and the cloud platforms, enabling efficient data collection and transmission within the IoT ecosystem.

In case the primary Internet connection is lost or disrupted, the custom gateway leverages LoRa technology to transmit the data to The Things Network as a backup communication channel. This ensures that the data collected from BLE devices is not lost and can still be forwarded to the appropriate cloud platform or application server.

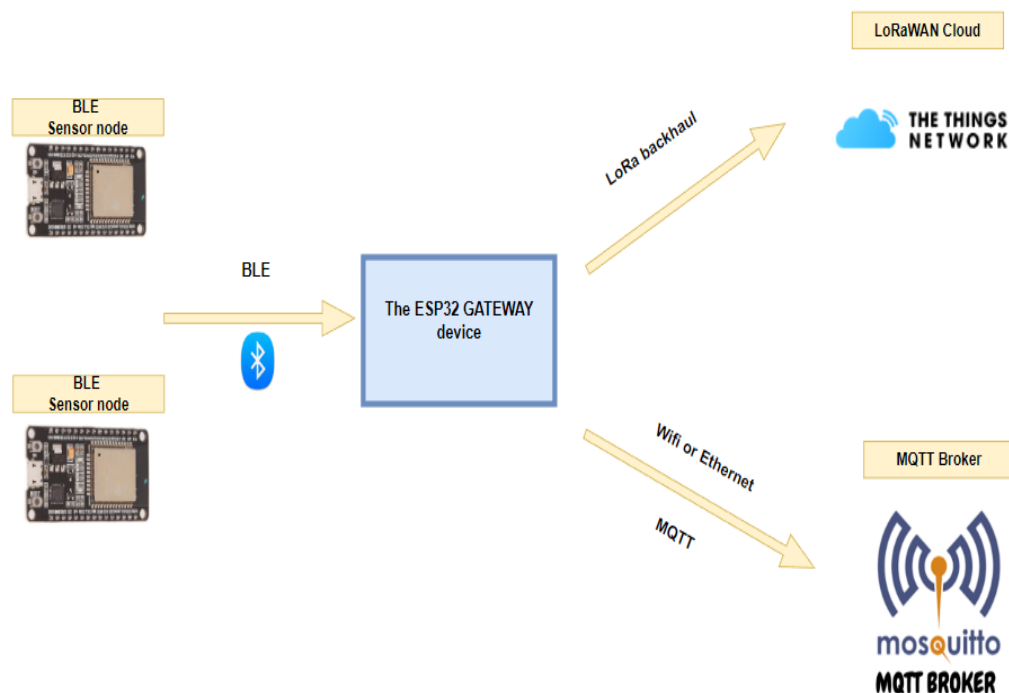


Figure 1. Gateway design topology

The Gateway topology includes:

- BLE sensors devices include various IoT devices equipped with BLE capabilities, such as sensors, wearables, or smart home appliances. They transmit data wirelessly to the custom gateway using the BLE protocol.
- The LoRa module refers to IoT devices that communicate with the custom gateway using the LoRa protocol. They can be used for long-range communication in remote or challenging environments and serve as a backup communication channel in case the primary Internet connection is disrupted.
- The ESP32 Gateway is responsible for collecting data from BLE devices and LoRa nodes, aggregating the data, and transmitting it to an MQTT broker using the primary Internet connection or the LoRa-based backup communication channel.
- The MQTT Broker receives the data from the custom gateway and forwards it to the appropriate cloud platform or application server, where it can be processed, analyzed, or integrated with other IoT systems.
- The Cloud Platforms receive and store the data collected by the custom gateway. They can also provide various services, such as data analytics, visualization, and integration with other IoT systems or third-party applications.

2 THEORETICAL BACKGROUND

This chapter provides an overview of the theoretical background relevant to the custom ESP32 gateway, including a brief introduction to IOT Gateway, BLE, LoRa, and MQTT technologies. Understanding these fundamental concepts is crucial for the design and development of the gateway, as it allows for informed decisions regarding component selection, system architecture, and communication protocols.

2.1 Bluetooth IOT Gateway

A Bluetooth gateway is a key component in a Bluetooth IoT solution. A Bluetooth gateway features a distribution network that supports the Bluetooth protocol. After receiving information from BLE end devices, Bluetooth Gateways forward information to the network server via Wi-Fi/Ethernet/LTE. /1/

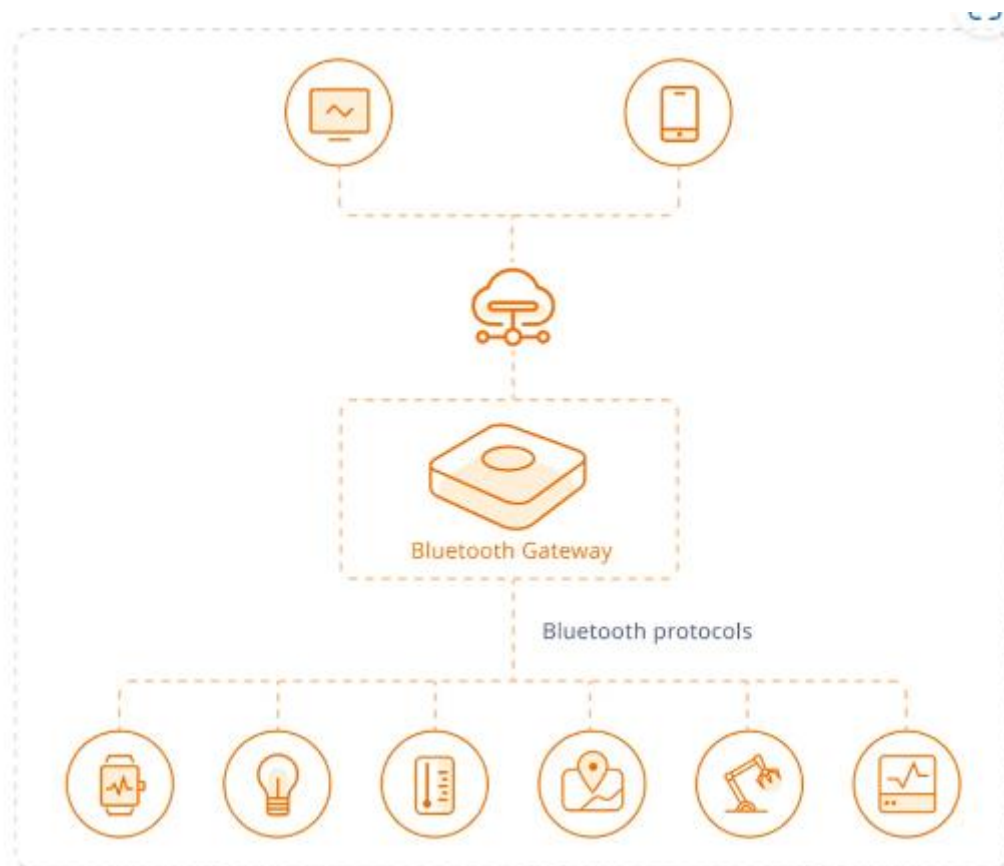


Figure 2. BLE Gateway topology /1/

The Bluetooth IoT gateway locates a device, it recognizes the features and data structures of surrounding BLE devices, which will transmit signals at regular intervals. After that, the Bluetooth IoT gateway handles a request using MQTT or another communication protocol to establish a connection. The MQTT server is called a broker and the clients are simply the connected devices. When a device (a client) wants to send data to the broker, we call this operation a “publish”. When a device (a client) wants to receive data from the broker, we call this operation a “subscribe”. Through the MQTT protocol, data from BLE devices can be quickly retrieved by the cloud, and commands from the cloud will be sent back to BLE devices.

2.2 Bluetooth Low Energy (BLE)

Bluetooth Low Energy (BLE) is a wireless communication technology designed specifically for short-range, low-power connectivity in IoT devices. This section provides a comprehensive overview of the theory behind BLE technology and its application in the custom ESP32 gateway.

Bluetooth Low Energy (BLE, also marketed as Bluetooth Smart) started as part of the Bluetooth 4.0 Core Specification. It's tempting to present BLE as a smaller, highly optimized version of its bigger brother, classic Bluetooth, but, BLE has an entirely different lineage and design goals. Originally designed by Nokia as Wibree before being adopted by the Bluetooth Special Interest Group (SIG), the developers were not trying to propose another overly broad wireless solution that attempts to solve every possible problem. From the beginning, the focus was to design a radio standard with the lowest possible power consumption, specifically optimized for low cost, low bandwidth, low power, and low complexity. /2/

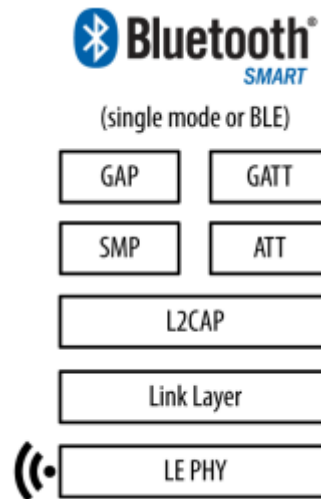


Figure 3. Configurations Bluetooth Smart /3/

In the ESP32 gateway, BLE technology is used to collect data from nearby BLE-enabled devices. The gateway acts as a central device that scans for available BLE peripherals, such as sensors, wearables, or smart home devices, and establishes a connection with them to retrieve data.

The ESP32 microcontroller, which serves as the core of the custom gateway, has integrated support for BLE communication, simplifying the implementation process. The firmware for the gateway is developed to include BLE functionality, enabling it to scan available peripherals, connect to them, and collect data.

When the gateway receives data from a BLE device, it aggregates the information and combines it with data collected from LoRa nodes. The aggregated data is then transmitted to an MQTT server, which facilitates communication with cloud platforms, analytics tools, or other IoT systems.

2.3 LoRa Technology

Long Range (LoRa) technology is a low-power, wide-area network (LPWAN) technology that enables long-range communication between IoT devices. This section provides an in-depth overview of the theory behind LoRa technology and its application in the custom ESP32 gateway, where it serves as a backhaul to cloud platforms, such as The Things Network.

LoRa is a wireless communication technology that focuses on providing long-range connectivity, low power consumption, and support for many devices. It uses a proprietary modulation scheme called Chirp Spread Spectrum (CSS) to provide robust and scalable communication in challenging environments, such as urban areas with high interference levels or rural areas with limited network coverage.

LoRa operates in the sub-GHz frequency bands, which are globally available and generally unlicensed, eliminating the need for regulatory approval in most cases. It can achieve a range of up to 15 kilometers in rural areas and several kilometers in urban environments, depending on factors such as antenna height, output power, and line-of-sight conditions. (4)

LoRa networks typically consist of end devices (for example, sensors, actuators), gateways, and a network server. End devices transmit data to gateways using the LoRa modulation scheme, while gateways forward the received data to the network server over a backhaul connection, such as Ethernet, Wi-Fi, or cellular networks. The network server then processes the data and sends it to the appropriate application server or cloud platform. (4)

Table 1 LoRa Specifications /4/

Specification	LoRa Feature
Range	2-5Km Urban (1.24-3.1 mi), 15Km suburban (9.3 mi)
Frequency	ISM 868/915 MHz
Standard	IEEE 802.15.4g
Modulation	Spread spectrum modulation type based on FM pulses which vary.
Capacity	One LoRa gateway takes thousands of nodes
Battery	Long battery life
LoRa Physical layer	Frequency, power, modulation and signalling between nodes and gateways

In the custom ESP32 gateway, the primary focus is to collect data from BLE devices and upload it to an MQTT server. LoRa technology is integrated into the gateway to serve as a backup communication channel in case the main Internet connection is disrupted.

To enable this backup communication channel, the gateway is equipped with a LoRa module, which communicates with LoRaWAN gateways connected to TTN. When the primary Internet connection is unavailable, the ESP32 microcontroller interfaces with the LoRa module to transmit the aggregated data from BLE devices to TTN. /10/

2.4 MQTT Protocol

MQTT (Message Queuing Telemetry Transport) is a lightweight, publish-subscribe messaging protocol designed for efficient communication in IoT applications. This section provides an overview of the MQTT protocol and its application in the custom ESP32 gateway, where it is used for transmitting data to cloud platforms or other IoT systems. /5/

MQTT was developed by IBM in the late 1990s as a protocol for telemetry systems and has since evolved into a widely used communication protocol for IoT applications. It operates over TCP/IP, providing a reliable and ordered delivery of messages between devices. MQTT's publish-subscribe model allows devices to send (publish) messages to "topics" and receive (subscribe) messages from those topics, enabling efficient and scalable communication. /5/

The MQTT protocol is designed to be lightweight, with a small code footprint and minimal bandwidth usage, making it suitable for resource-constrained IoT devices and low-bandwidth networks. It also provides quality of service (QoS) levels that enable devices to choose the appropriate level of message delivery assurance based on their requirements. /5/

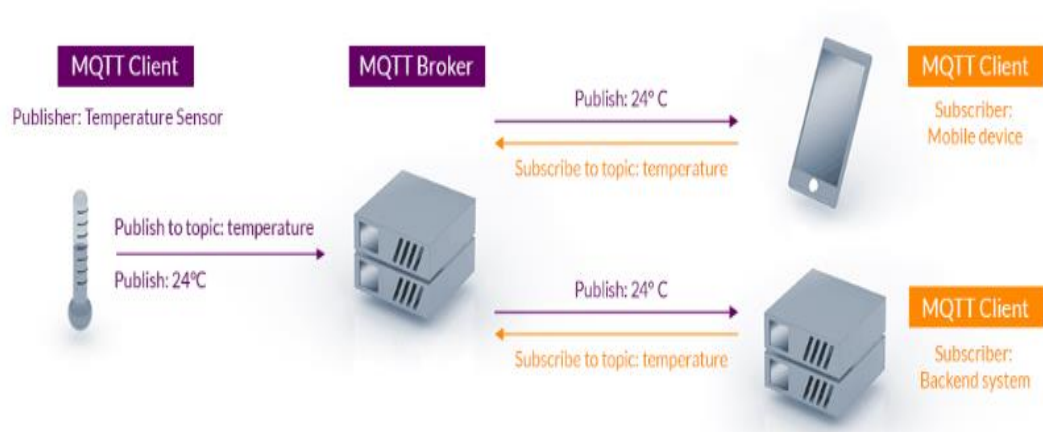


Figure 4. MQTT Architecture /5/

In the ESP32 gateway, the MQTT protocol is used to transmit data collected from BLE devices and LoRa nodes to cloud platforms or other IoT systems. The gateway firmware is developed to include MQTT functionality, allowing it to connect to an MQTT broker and publish messages containing aggregated data from the connected devices.

By incorporating the MQTT protocol into the custom gateway, efficient and scalable communication with cloud platforms and IoT systems can be achieved. MQTT's lightweight design and quality of service levels make it an ideal choice for the gateway, ensuring reliable transmission of data while minimizing resource usage and bandwidth consumption.

3 PROOF-OF-CONCEPT (POC) PROTOTYPES

3.1 Prototyping Process

The prototyping process is an essential step in the development of the custom ESP32 gateway, as it allows for the testing and validation of design concepts, hardware components, and software functionality. Initially, off-the-shelf modules, such as an Ethernet module, ESP32 Development Board, and a LoRa module, were used to create a prototype of the gateway. This prototype serves as a platform for evaluating the performance and compatibility of the selected components and assessing their suitability for the final product.

During the prototyping process, various iterations may be developed and tested to refine the design, optimize the performance, and address any issues that arise. This iterative process helps to identify potential problems early in the development cycle and allows for modifications to be made before moving on to the next stages, such as PCB design.

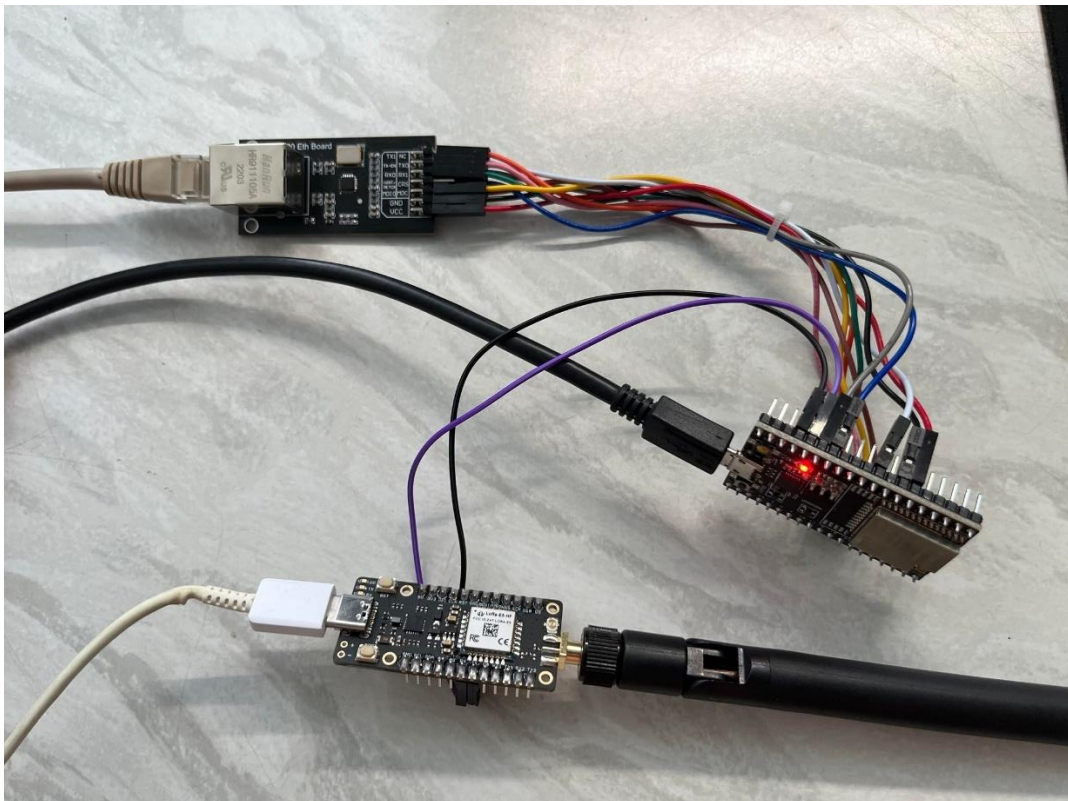


Figure 5. Proof-of-concept setup

3.2 Prototyping Hardware Components

Choosing the appropriate hardware components for the custom ESP32 gateway is crucial for achieving the desired performance, functionality, and reliability. The components used in the prototype serve as the basis for the final hardware selection and integration into the PCB design. The following sections describe the rationale behind the selection of the main hardware components and their integration into the PCB design.

3.2.1 ESP32 Development Kit

The ESP32-DevKitC-32U is the core component of the custom gateway due to its powerful processing capabilities, integrated BLE and Wi-Fi functionalities, and extensive support for various communication protocols. The ESP32-DevKitC used in the prototype is incorporated into the PCB design, enabling efficient multitasking and concurrent processing of multiple tasks while maintaining its integrated communication capabilities and support for IoT applications. ESP32-WROOM-32U integrates a connector to connect an external antenna.

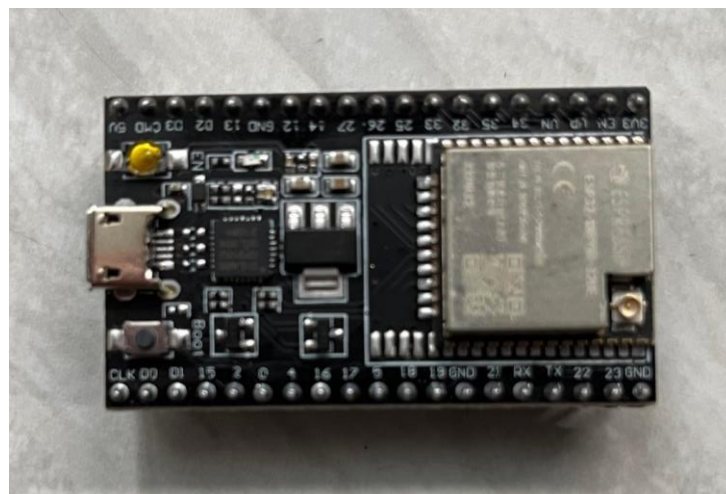


Figure 6. ESP32 DevKitC-32U

An overview of the specifications of the ESP32-DevKitC-32U used in the prototype. This helped in understanding the capabilities and features of the ESP32 microcontroller that contribute to the performance of the ESP32 gateway. A detailed picture of the ESP32-DevKitC-32U specifications is attached below for reference.

Table 2. ESP32-WROOM-32U Specifications /6/

Wi-Fi	Protocols	802.11 b/g/n (802.11n up to 150 Mbps) A-MPDU and A-MSDU aggregation and 0.4 μ s guard interval support
	Center frequency range of operating channel	2412 ~ 2484 MHz
Bluetooth	Protocols	Bluetooth v4.2 BR/EDR and Bluetooth LE specification
	Radio	NZIF receiver with -97 dBm sensitivity
		Class-1, class-2 and class-3 transmitter
		AFH
	Audio	CVSD and SBC
Hardware	Module interfaces	SD card, UART, SPI, SDIO, I2C, LED PWM, Motor PWM, I2S, IR, pulse counter, GPIO, capacitive touch sensor, ADC, DAC, Two-Wire Automotive Interface (TWAI [®]), compatible with ISO11898-1 (CAN Specification 2.0)
	Integrated crystal	40 MHz crystal
	Integrated SPI flash ¹	4 MB
	Operating voltage/Power supply	3.0 V ~ 3.6 V
	Operating current	Average: 80 mA
	Minimum current delivered by power supply	500 mA
	Recommended operating ambient temperature range ²	-40 °C ~ +85 °C
	Moisture sensitivity level (MSL)	Level 3

The ESP32-WROOM-32U included in the development kit above was used later in the PCB design.

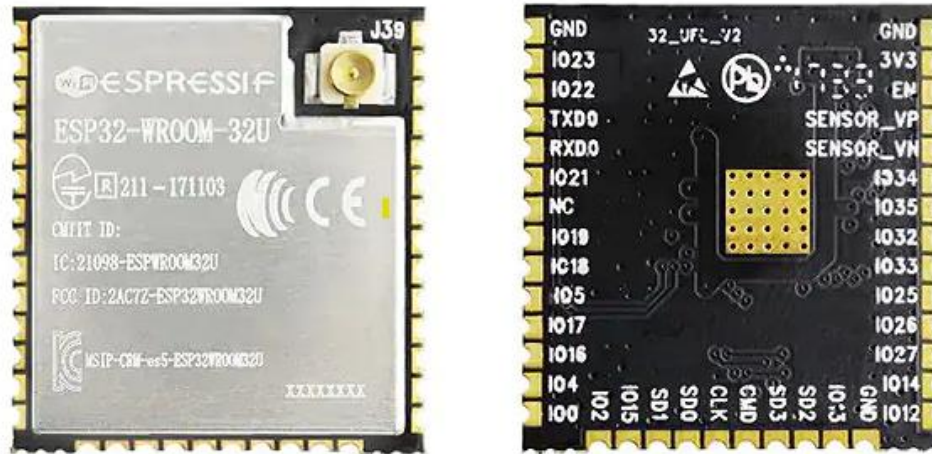


Figure 7. ESP32-WROOM-32U module

3.2.2 Ethernet Module

The Ethernet module was selected for its ability to provide a stable and reliable Internet connection to the custom gateway, ensuring efficient data transmission to cloud platforms or other IoT systems. The Ethernet module used in the prototype is integrated into the PCB design, allowing for a compact and efficient layout while maintaining the benefits of a wired connection, such as higher reliability and lower latency compared to Wi-Fi. In this case, the LAN8720 ETH Board was chosen.

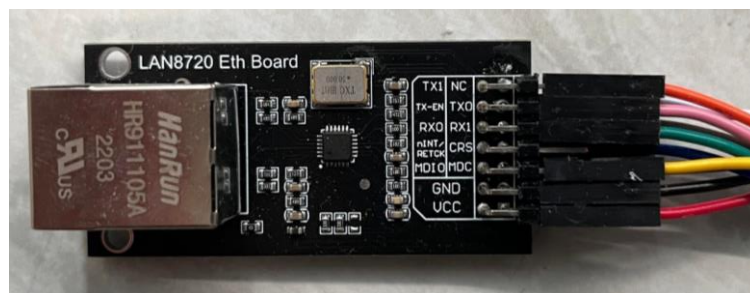


Figure 8. LAN8720 ETH module

The advantages of the LAN8720 ETH module are:

- High-Performance 10/100 Ethernet Physical Layer Transceiver (PHY)
- Supports single 3.3V supply.
- Supports the reduced pin count RMI interface.
- Supports HP Auto-MDIX
- Onboard chip package: 24-pin QFN (4x4 mm) Lead-Free RoHS Compliant package
- Flexible Power Management Architecture
- Integrated 1.2V regulator.
- I/O voltage range: +1.6V to +3.6V

3.2.3 LoRa Module

The LoRa module is integrated into the custom gateway to provide long-range communication capabilities and serve as a backup communication channel in case the primary Internet connection is disrupted. The LoRa module used in the prototype is integrated into the PCB design, ensuring a compact and efficient layout while maintaining the benefits of long-range connectivity. In this case, I chose the Wio-E5 from SEEED Studio.

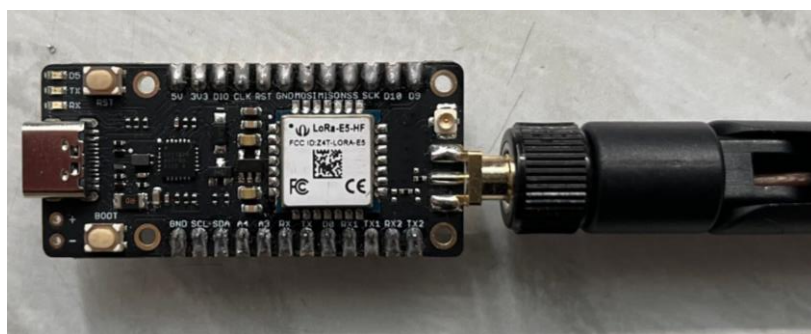


Figure 9. Wio-E5 mini development board

Wio-E5 mini is a compact-sized development board suitable for the rapid testing and building of small-size prototyping. Wio-E5 mini is embedded with Wio-E5 STM32WLE5JC Module, which delivers the world-first combo of LoRa® RF and MCU chip into one single tiny chip and is FCC and CE certified. It is powered by ARM Cortex-M4 core and Semtech SX126X LoRa® chip and supports LoRaWAN® protocol on the worldwide frequency and (G)FSK, BPSK, (G)MSK, and LoRa® modulations. The built-in AT command firmware makes it easy to interact for our ESP32 gateway application. /7/

The Lora-E5 STM32WLE5JC Module included in the above development kit was used later in the PCB design (See Figure 10).

Table 3. Wio-E5 specifications /7/

Parameters	Specifications
size	50*23mm
voltage - supply	3.7V - 5V
power - output	up to +20.8 dBm at 3.3V
working frequency	868/915MHz
protocol	Long Range
sensitivity	-116.5 dBm ~ -136 dBm
interfaces	USB Type C / 2P-2.54mm Hole / 1*12P-2.54mm Header*2 / SMA-K / IPEX
modulation	Long Range , (G)FSK, (G)MSK, BPSK
working temperature	-40°C ~ 85°C
current	Wio-E5 module sleep current as low as 2.1uA (WOR mode)



LoRa-E5 (STM32WLE5JC)

Core	32-bit Arm Cortex-M4 CPU, up to 48MHz
LoRaWAN stack	Built-in with AT Command Firmware; Program with STM32Cube MCU Package
Package	12*12mm, 28 pins SMD
Interfaces	UART*3, I2C*1, ADC(12-bit)*1, SPI*1, GPIO*6
Sensitivity	-116.5dBm(SF5), -121.5dBm(SF7), -136dBm(SF12)
Modulation	LoRa, (G)FSK, (G)MSK and BPSK
Certificate	FCC and CE (EU868/US915)
Power Supply	1.8V ~ 3.6V
RF Output Power	up to +20.8 dBm at 3.3V

Figure 10. LoRa-E5 module /7/

3.3 Connections Table

Table 4. ESP32 to LAN8720A Module

ESP32 DEVKIT	LAN8720A ETH module
GPIO5 - PHY_POWER	NC - Osc. Enable
GPIO22 - EMAC_TXD1	TX1
GPIO19 - EMAC_TXD0	TX0
GPIO21 - EMAC_TX_EN	TX_EN
GPIO26 - EMAC_RXD1	RX1
GPIO25 - EMAC_RXD0	RX0
GPIO27 - EMAC_RX_DV	CRS
GPIO00 - EMAC_TX_CLK	nINT/REFCLK
GPIO23 - SMI_MDC	MDC
GPIO18 - SMI_MDIO	MDIO
GND	GND
3V3	VCC

PHY_POWER, **SMI_MDC** and **SMI_MDIO** can freely be moved to other GPIOs.

EMAC_TXD0, **EMAC_TXD1**, **EMAC_TX_EN**, **EMAC_RXD0**, **EMAC_RXD1**,
EMAC_RX_DV and **EMAC_TX_CLK** are fixed and can't be rerouted to other GPIOs.

Table 5. ESP32 to Wio-E5

ESP32 DEVKIT	WIO-E5 mini
GPIO12	TX
GPIO13	RX

No connection for 3.3 V and GND since the Wio-E5 mini has its own power supply using USB Type-C.

4 PCB DEVELOPMENT

In the development of the custom BLE-LoRa gateway, designing a robust and efficient Printed Circuit Board (PCB) is crucial. The PCB serves as the backbone of the electronic device, providing a stable platform for mounting and interconnecting various components. This chapter will discuss the process of PCB development, from selecting appropriate design software to creating the schematic design and routing the connections between components. The PCB development process aims to create a compact and efficient layout that meets the requirements of the custom ESP32 gateway while adhering to best practices and design constraints.

4.1 PCB Block Diagram

Before going into the detailed schematic design and PCB layout, it is essential to develop a block diagram that represents the placement of components on the PCB. This block diagram serves as a visual guide for understanding the overall structure of the custom ESP32 gateway and aids in the organization of components during the design process. The diagram considers the functional relationships between the various components, ensuring that their placement on the PCB is efficient and logical.

A block diagram of the PCB for the custom ESP32 gateway is provided below in Figure 11, illustrating the arrangement of the ESP32 module, LAN8720A Ethernet module, Wio E5 LoRa module, and other essential components.

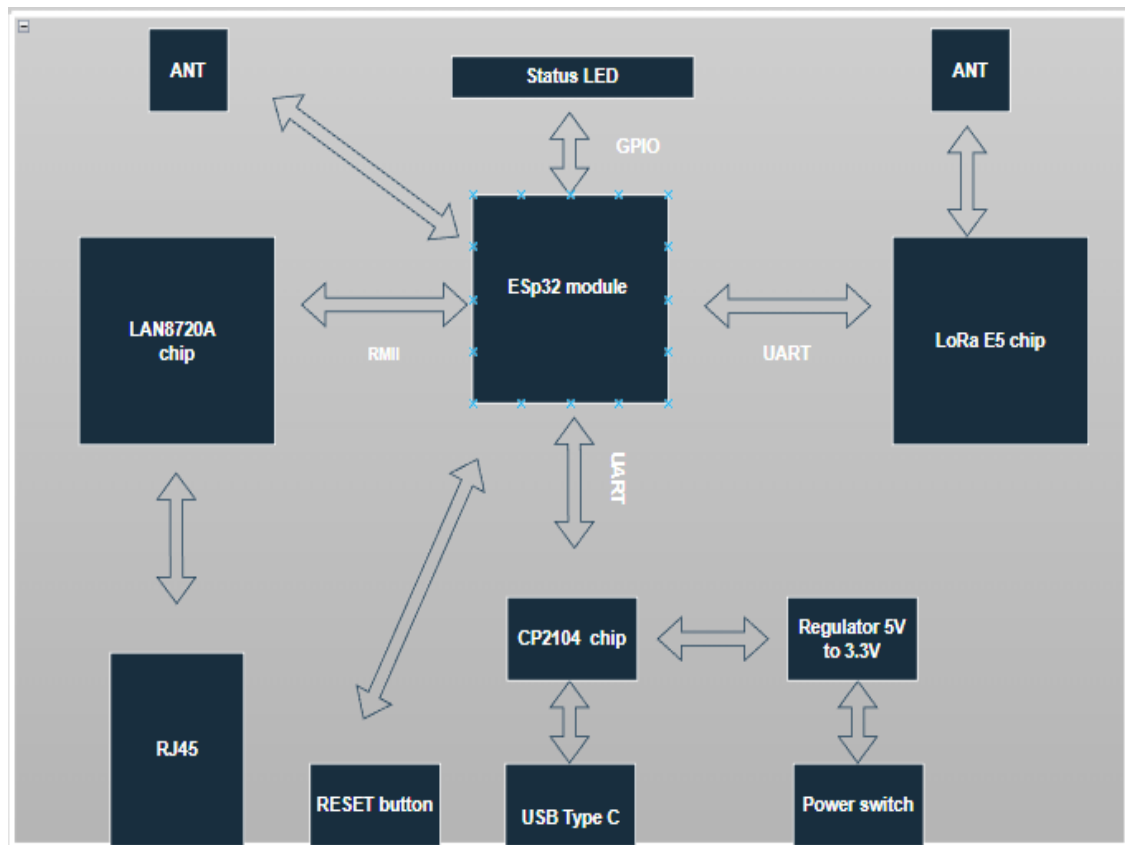


Figure 11. PCB Block Diagram

By creating a clear and organized block diagram, the PCB design process is streamlined, allowing for more effective component placement and routing in the subsequent stages of development.

4.2 Schematic Design

Creating a thorough schematic diagram that illustrates the connections between components in the customized ESP32 gateway is part of the schematic design process. This is an important step in PCB development since it offers a visual representation of the overall design and ensures that all components are properly connected before moving on to the PCB layout stage.

4.2.1 Ethernet LAN8720A Chip

The LAN8720A is a low-power 10BASE-T/100BASE-TX physical layer (PHY) transceiver with variable I/O voltage that is compliant with the IEEE 802.3-2005 standards. The LAN8720A/LAN8720Ai supports communication with an Ethernet MAC via a standard RMII interface. It contains a full-duplex 10-BASE-T/100BASE-TX transceiver and supports 10Mbps (10BASE-T) and 100Mbps (100BASE-TX) operation. /9/

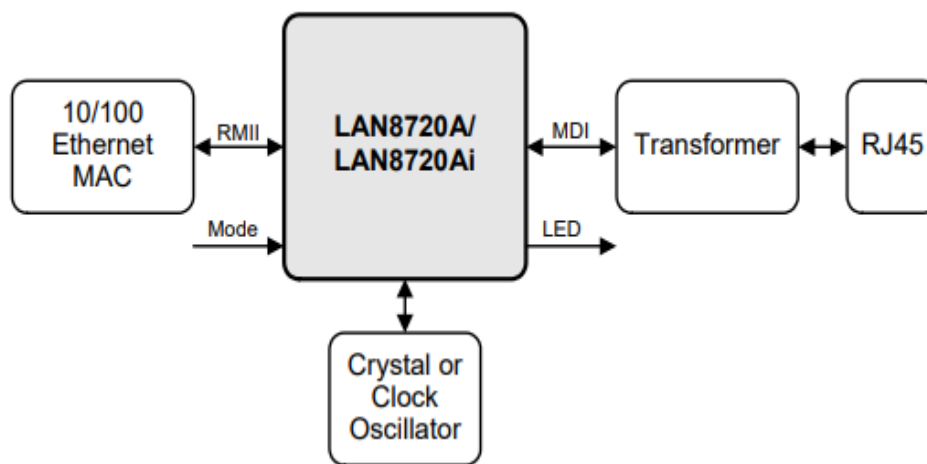


Figure 12. LAN8720A system block /9/

The power for the LAN8720A is based on Twisted-Pair Interface Diagram or Dual power supplies. In this case are name **+3.3VLAN** and **VDDA** connecting with a 600-ohm Ferrite Bead (L1).

The power for the LAN8720A is triggered by the **PHY_PWR** pin wich connected to GPIO pin 5 in ESP32 MCU. When **PHY_PWR** is HIGH, in the Q1 Mosfet the GND (Emitter) pin will connect with the OUT (Collector) pin. When Pin 1 in Mosfet Q4 is connected to GND, the dual power supplies are triggered by connecting to the +3.3V (See Figure 13).

4.2.3 Power Management

The VUSB_UNSWITCHED, VBUS is connected to a power switch to control the power of the board.

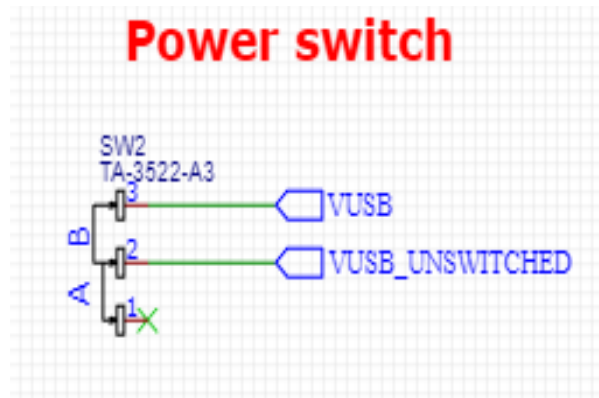


Figure 18. Power switch

A linear regulator was added to lower the output voltage from 5V to 3.3V. The 3.3V pin out of the regulator will be the power supply for the +3.3V layer of the board since this is the 4 layers PCB.

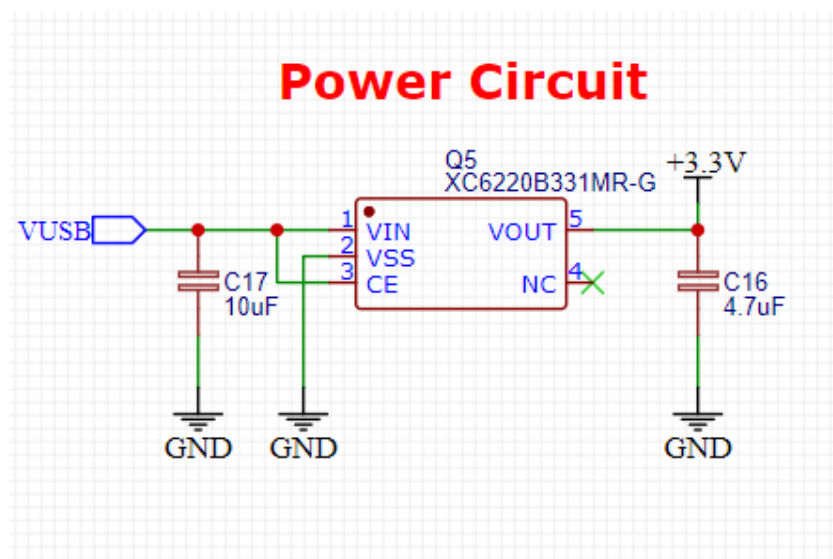


Figure 19. Circuit power

4.2.5 Peripherals

The Uart Switch is for debugging purposes, if several ESP32 connect to the computer, sometimes choosing the COM port for uploading firmware can be confusing. An Uart Switch is designed so that when the ESP32 gateway is not in uploading mode, it can be turned OFF. LORA mode is for testing with the LoRa module first before uploading the setting to ESP32, with Serial Terminal software like RealTerm or Putty.

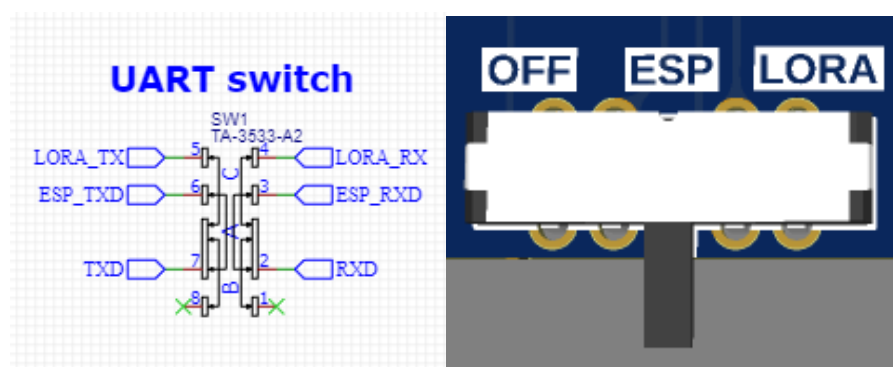


Figure 22. UART switch

The Status LED using an addressable LED, its only need 1 GPIO pin to control the 3 LEDs, the power use is 5V which is VUSB. Other connections for External Antennas and SWD pin out for LoRa firmware.

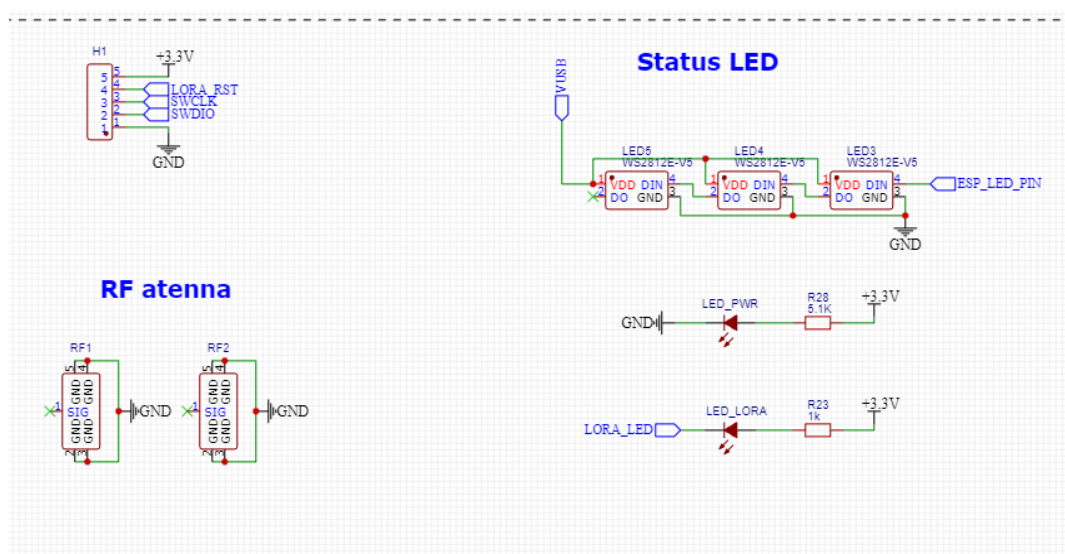


Figure 23. Peripherals

4.3 PCB Layout

Before starting with the PCB layout and routing, we must set the design rules for the PCB that fits the PCB manufacture capabilities which in this case is JLC PCB.

Design Rule					
Rule	Track Width	Clearance	Via Diameter	Via Drill Diameter	Track Length
Default	0.254	0.152	0.6	0.305	

Figure 24. Design rules

The next step is the Layer Manager, 4 layers PCB were used. PCB with layer 1 and 4 are signal layer, layer 2 is +3.3V and layer 3 for GND.

Layer Manager					
Copper Layer: 4					
No.	<input type="checkbox"/> Display	Name	Type	Color	Transparency(%)
1	<input checked="" type="checkbox"/>	TopLayer	Signal	#FF0000	0
2	<input checked="" type="checkbox"/>	+3.3V	Plane	#999966	0
3	<input checked="" type="checkbox"/>	GND	Plane	#008000	0
4	<input checked="" type="checkbox"/>	BottomLayer	Signal	#0000FF	0
5	<input checked="" type="checkbox"/>	TopSilkLayer	Non-Signal	#FFCC00	0
6	<input checked="" type="checkbox"/>	BottomSilkLayer	Non-Signal	#66CC33	0
7	<input checked="" type="checkbox"/>	TopPasteMaskLayer	Non-Signal	#808080	0
8	<input checked="" type="checkbox"/>	BottomPasteMaskLayer	Non-Signal	#800000	0
9	<input checked="" type="checkbox"/>	TopSolderMaskLayer	Non-Signal	#800080	30
10	<input checked="" type="checkbox"/>	BottomSolderMaskLayer	Non-Signal	#AA00FF	30

Figure 25. PCB layers

Figures 26 and 27 show the finished layout, routing and 3D view of the PCB.

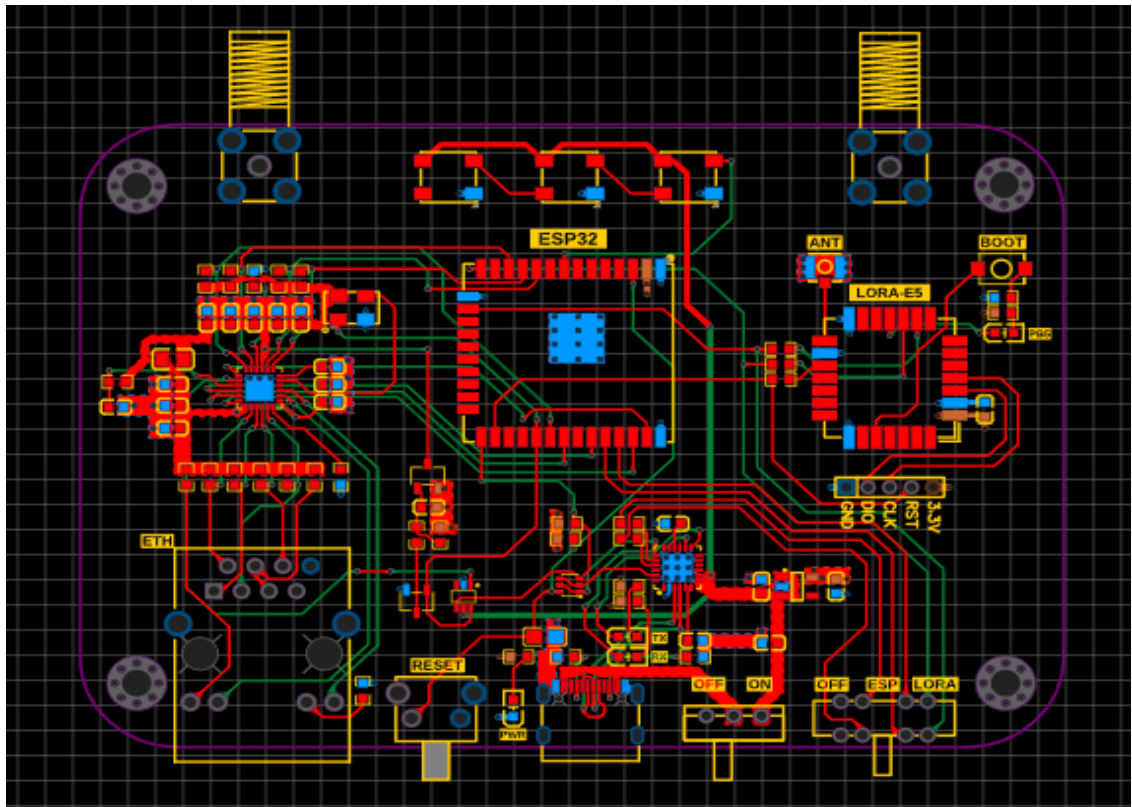


Figure 26. PCB layout

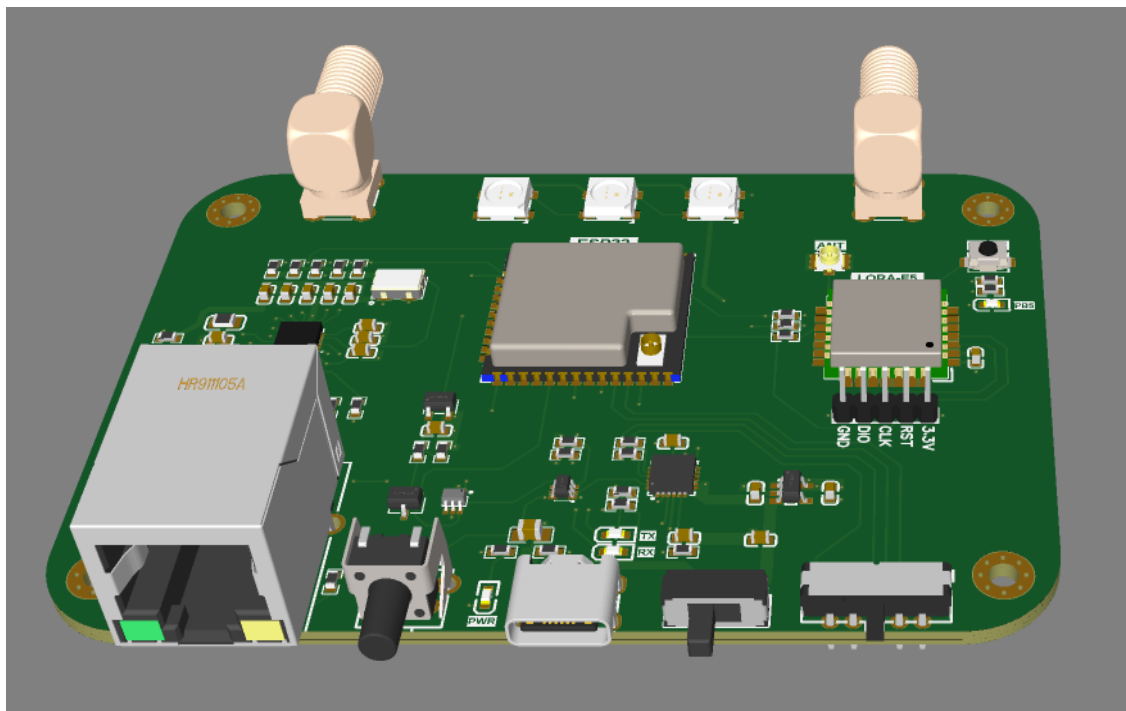


Figure 27. PCB 3D viewer

4.4 PCB Assembled and Testing

Components were soldered to the PCB using a hot air gun for SMT parts and solder iron for through-hole parts.

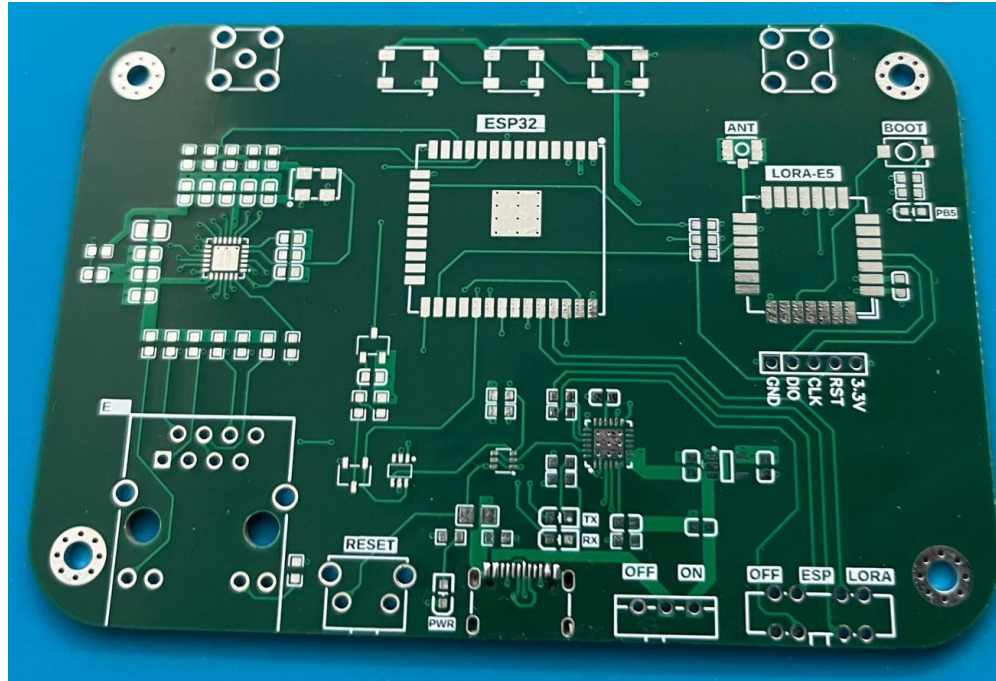


Figure 28. PCB unassembled

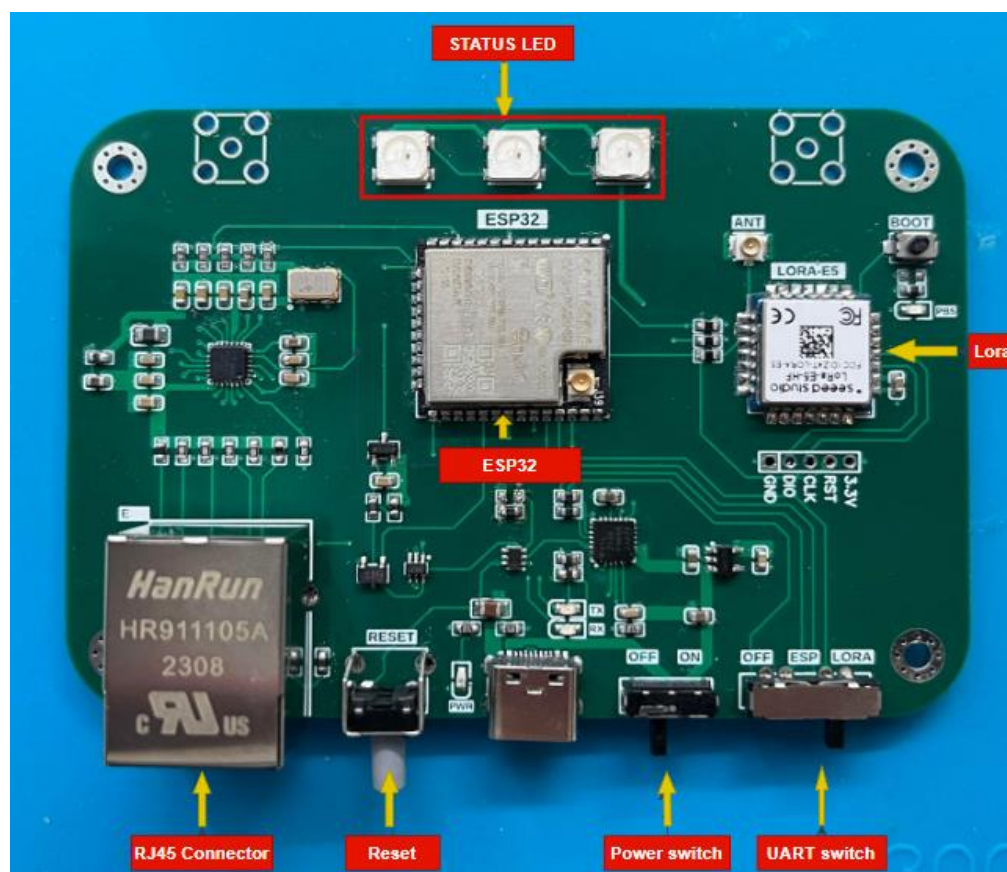


Figure 29. PCB assembled top



Figure 30. PCB assembled front

The next step was testing the PCB with our firmware, to see all the parts and modules working as expected and fixing the problems.

After testing, a multimeter was used for checking values. Two problems were found, and they are fixed as shown below.

Problem 1: USB to UART CP2104 was working but the firmware could not be uploaded to the ESP32. The reason was the ESP32 only enters the uploading mode when GPIO 0 is LOW, but there was a mistake in the schematic that made this GPIO 0 pin pull-up to +3.3V.

The problems were fixed by removing the R6 resistor and soldering an external wire between GPIO 0 and the “right” pad of R6 footprint, following shown in Figure 31 below.

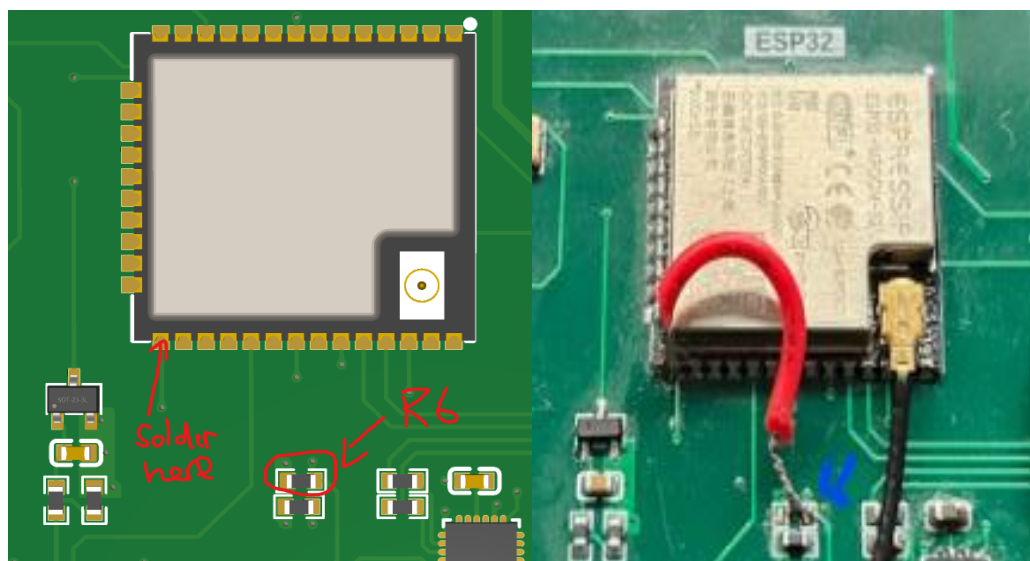


Figure 31. PCB problem 1

Problem 2: UART switch was not working for LORA mode. The reason for this was there was a wrong pin for LORA_RX and LORA_TX in the schematic, it is supposed to be opposite, as in Figure 32.

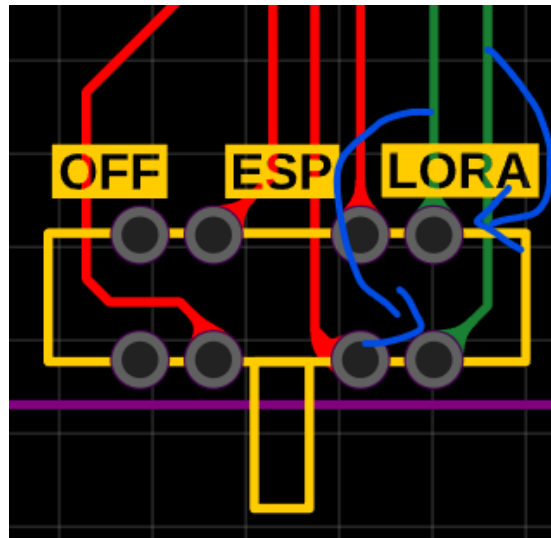


Figure 32. PCB problem 2

This LORA mode is optional, and we can still use UART2 of the ESP32 to do the testing with LORA module, it was decided to leave it that way.

However, there is one way to fix this by soldering some external wires. Since we have the OFF mode which are blank pins with no connection, we can wire it as in Figure 33 below, and now the OFF mode turns into LORA mode and LORA mode can be left unused.

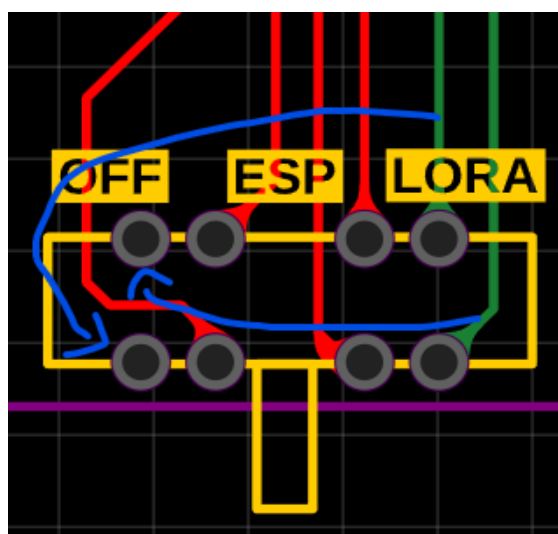


Figure 33. Fixed PCB problem 2 layout

We can wire it in the back of the board to keep the “aesthetic” look of the front PCB.

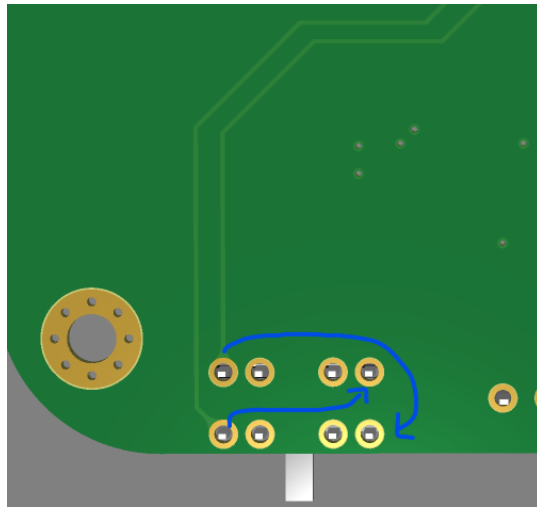


Figure 34. Fixed PCB problem 2 back

After fixing all the problems, the board is fully functional and ready.

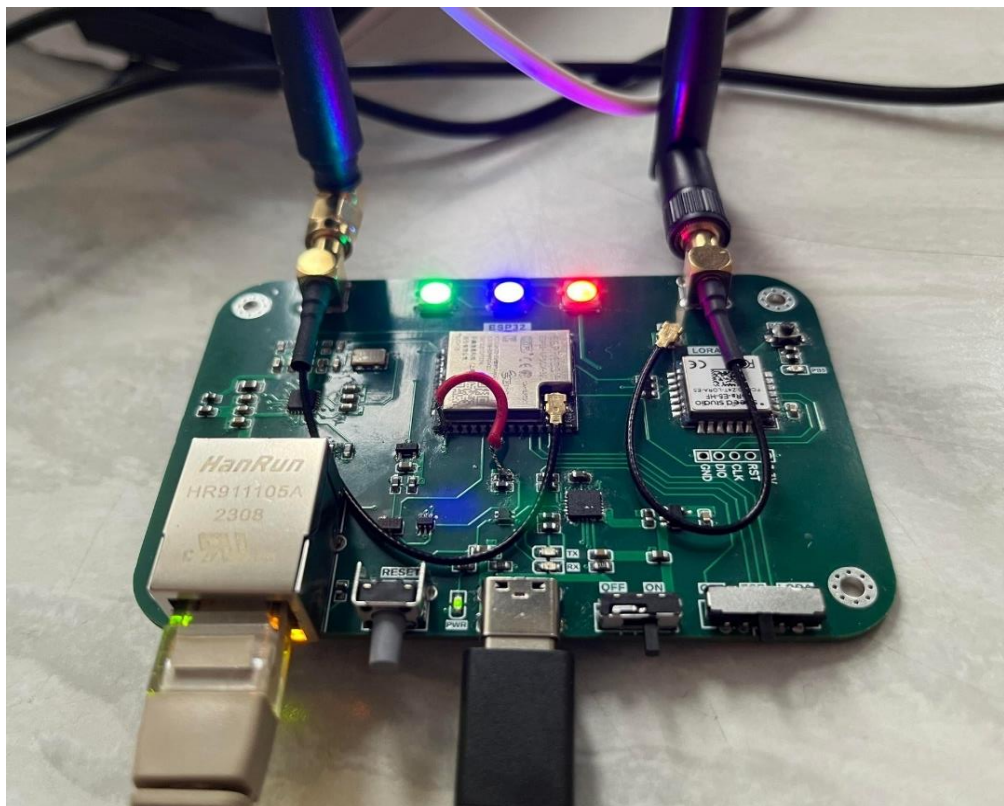


Figure 35. Fully functional PCB

5 FIRMWARE DEVELOPMENT

The firmware of the custom ESP32 gateway serves as a software foundation that controls and manages the functionality of the device. It enables communication between the hardware components and implements the use case topology designed during the hardware development phase (Figure 1). In this chapter, an overview of the firmware development process, along with a visual representation, will be provided.

5.1 BLE Beacon Gateway Setup

The BLE beacon gateway can automatically detect any BLE Beacons nearby, but this firmware focuses on **iBeacon** and **Eddystone** beacon frametype. BLE beacon is a broadcaster type, which means that it does not require making connections to get the data from them (as BLE GATT devices do), it will advertise data packet to the surrounding at regular intervals. The ESP32 gateway scans for the advertise packets and decodes them according to the beacon frametype, collects data from them, and then passes data using Ethernet/ LoRa module to the server by MQTT protocols and LoRa server.

The gateway can scan for every iBeacon or Eddystone beacon in range but only send their data to MQTT topics or LoRa server when it matches the beacon MAC address (each Beacon has a unique MAC address).

The BLE Beacon Gateway setup is shown in Figure 36.



Figure 36. BLE beacons Gateway setup

In the testing setup, there were 3 BLE beacons, **node 1** and **node 2** (mark in Figure 41) are Eddystone beacons frametype, the advertise packet included a temperature and battery level whose the values been generated by random function. **Node 3** (mark in Figure 41) is an iBeacon frametype that include manufacture ID and an RSSI value to estimate the power signal.

The ESP32 with Ethernet connected, the 3 status LEDs were: **GREEN** for MQTT server connection, **BLUE** for BLE connection and **RED** for LORA connection. Status LEDs will blink **WHITE** color every time the connections get data in or out.

The Wio-E5 LoRa module with the long antenna connected to PC, was used as a LoRa Gateway to replace the use case of The Thing Networks since there are no nearby TTN Gateway in the region of Vaasa. The Lora module will receive data sent from the ESP32 gateway and decode the data and print it to Serial Monitor.

5.2 Firmware

The source code includes four main files: the main code (main.ino) for scanning BLE beacons and decode them, the MQTT settings (mqtt.h) and topic, the lora settings (lora.h) and other settings (settings.h).

5.2.1 Main Code (main.ino)

The library includes: the necessary libraries for handling HTTP requests, BLE devices, MQTT communication, LED control, and LoRa communication are included.

```
#include <HTTPClient.h>
#include <Arduino.h>
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEScan.h>
#include <BLEAdvertisedDevice.h>
#include <BLEEddystoneURL.h>
#include <BLEEddystoneTLM.h>
#include <BLEBeacon.h>
#include "mqtt.h"
#include "led.h"
#include "settings.h"
#include "lora.h"
```

Listing 1. Libraries for main.ino

Global variables and objects were **scanTime** and **pBLEScan**, defined to control the BLE scanning process. **MyAdvertisedDeviceCallbacks** class was derived from **BLEAdvertisedDeviceCallbacks** to handle the discovered BLE devices.

```
int scanTime = 5; //In seconds
BLEScan *pBLEScan;

class MyAdvertisedDeviceCallbacks : public BLEAdvertisedDeviceCallbacks
```

Listing 2. Scan times

MyAdvertisedDeviceCallbacks::onResult: This method is called when a new BLE device is discovered. It handles two types of BLE devices: iBeacons and Eddystone beacons. When a device is found, it processes its data, checks the device's MAC address, and calls the appropriate functions to send data to the MQTT server and over LoRa.

```
if (advertisedDevice.haveManufacturerData() == true)
{
    std::string strManufacturerData = advertisedDevice.getManufacturerData();

    uint8_t cManufacturerData[100];
    strManufacturerData.copy((char *)cManufacturerData, strManufacturerData.length(),
0);

    if (strManufacturerData.length() == 25 && cManufacturerData[0] == 0x4C &&
cManufacturerData[1] == 0x00)
    {
        int rssi = advertisedDevice.getRSSI();
        Serial.println("Found an iBeacon!");
        BLEBeacon oBeacon = BLEBeacon();
        oBeacon.setData(strManufacturerData);
        Serial.printf("iBeacon Frame\n");
        Serial.printf("ID: %04X Major: %d Minor: %d UUID: %s Power: %d RSSI: %d\n",
oBeacon.getManufacturerId(), ENDIAN_CHANGE_U16(oBeacon.getMajor()),
ENDIAN_CHANGE_U16(oBeacon.getMinor()),
oBeacon.getProximityUUID().toString().c_str(), oBeacon.getSignalPower(), rssi);
        Serial.println("\n");

        std::string deviceAddress = advertisedDevice.getAddress().toString();
        if (deviceAddress == iBeaconMacAddress) {
            sendMqttBeacon(deviceAddress.c_str(), oBeacon.getManufacturerId(),
ENDIAN_CHANGE_U16(oBeacon.getMajor()), ENDIAN_CHANGE_U16(oBeacon.getMinor()),
oBeacon.getProximityUUID().toString().c_str(), oBeacon.getSignalPower(), rssi);
            bleBlink();
            sendLoRaBeacon(deviceAddress.c_str(), oBeacon.getManufacturerId(),
ENDIAN_CHANGE_U16(oBeacon.getMajor()), ENDIAN_CHANGE_U16(oBeacon.getMinor()),
oBeacon.getProximityUUID().toString().c_str(), oBeacon.getSignalPower(), rssi);
        }
    }
}

uint8_t *payload = advertisedDevice.getPayload();
const uint8_t serviceDataEddystone[3] = {0x16, 0xAA, 0xFE};
const size_t payloadLen = advertisedDevice.getPayloadLength();
uint8_t *payloadEnd = payload + payloadLen - 1;
while (payload < payloadEnd) {
    if (payload[1] == serviceDataEddystone[0] && payload[2] == serviceDataEddystone[1]
&& payload[3] == serviceDataEddystone[2]) {
```

```

    // found!
    payload += 4;
    break;
}
payload += *payload + 1;
}

if (payload < payloadEnd)
{
    if (*payload == 0x10)
    {
        Serial.println("Found an EddystoneURL beacon!");
        BLEddystoneURL foundEddyURL = BLEddystoneURL();
        uint8_t URLLen = *(payload - 4) - 3;
        foundEddyURL.setData(std::string((char*)payload, URLLen));
        std::string bareURL = foundEddyURL.getURL();
        if (bareURL[0] == 0x00)
        {
            Serial.println("DATA-->");
            uint8_t *payload = advertisedDevice.getPayload();
            for (int idx = 0; idx < payloadLen; idx++)
            {
                Serial.printf("0x%02X ", payload[idx]);
            }
            Serial.println("\nInvalid Data");
            return;
        }

        Serial.printf("Found URL: %s\n", foundEddyURL.getURL().c_str());
        Serial.printf("Decoded URL: %s\n", foundEddyURL.getDecodedURL().c_str());
        Serial.printf("TX power %d\n", foundEddyURL.getPower());
        Serial.println("\n");
    }
    else if (*payload == 0x20)
    {
        Serial.println("Found an EddystoneTLM beacon!");
        BLEddystoneTLM eddystoneTLM;
        eddystoneTLM.setData(std::string((char*)payload, 14));
        float roundedTemp = round(eddystoneTLM.getTemp() * 100.0) / 100.0;
        Serial.printf("Reported battery voltage: %dmV\n", eddystoneTLM.getVolt());
        Serial.printf("Reported temperature: %.2f°C (raw data=0x%04X)\n",
eddystoneTLM.getTemp(), eddystoneTLM.getRawTemp());
        Serial.printf("Reported advertise count: %d\n", eddystoneTLM.getCount());
        Serial.printf("Reported time since last reboot: %ds\n", eddystoneTLM.getTime());
        Serial.println("\n");

        String scannedUUID = advertisedDevice.getServiceUUID().toString().c_str();
        std::string deviceAddress = advertisedDevice.getAddress().toString();
        if (deviceAddress == EddyMacAddress1) {
            sendEddystoneTlmMqttMessage1(deviceAddress.c_str(), scannedUUID,
eddystoneTLM.getVolt(), roundedTemp, eddystoneTLM.getCount(),
eddystoneTLM.getTime());

```

```

        sendLoRaEddystoneTlm1(deviceAddress.c_str(), scannedUUID,
eddstoneTLM.getVolt(), roundedTemp, eddstoneTLM.getCount(),
eddstoneTLM.getTime());
        bleBlink();

    }
    else if (deviceAddress == EddyMacAddress2) {
        sendEddystoneTlmMqttMessage2(deviceAddress.c_str(), scannedUUID,
eddstoneTLM.getVolt(), roundedTemp, eddstoneTLM.getCount(),
eddstoneTLM.getTime());
        sendLoRaEddystoneTlm2(deviceAddress.c_str(), scannedUUID,
eddstoneTLM.getVolt(), roundedTemp, eddstoneTLM.getCount(),
eddstoneTLM.getTime());
        bleBlink();
    }
}

```

Listing 3. Decode beacons

setup(): initializes the status LEDs, serial communication, BLE scanning with custom callbacks, MQTT, and LoRa.

```

void setup()
{
    initStatusLeds();
    updateLed(0, CRGB::Red);
    updateLed(1, CRGB::Blue);
    Serial.begin(115200);
    Serial.println("Scanning...");

    BLEDevice::init("");
    pBLEScan = BLEDevice::getScan();
    pBLEScan->setAdvertisedDeviceCallbacks(new MyAdvertisedDeviceCallbacks());
    pBLEScan->setActiveScan(true);
    pBLEScan->setInterval(100);
    pBLEScan->setWindow(99);

    mqttInit();
    loraInit();
}

```

Listing 4. BLE setups

loop(): The main loop starts the BLE scanning process, waits for the specified scan time, clears the results, and repeats the process every 2 seconds.

```
void loop()
{
  BLEScanResults foundDevices = pBLEScan->start(scanTime, false);
  Serial.println("Scan done!\n");
  pBLEScan->clearResults();
  delay(2000);
}
```

Listing 5. Main loop

5.2.2 MQTT settings (mqtt.h)

MQTT settings.h include the following files and declarations: The required header files are included, and the AsyncMqttClient and TimerHandle_t instances are declared.

```
#include <WiFi.h>
extern "C" {
  #include "freertos/FreeRTOS.h"
  #include "freertos/timers.h"
}
#include <AsyncMqttClient.h>
#include <ArduinoJson.h>
#include <BLEBeacon.h>
#include "settings.h"
#include "led.h"
```

Listing 6. MQTT libraries

The MQTT and Ethernet connection functions are shown in Listing 7:

```
void connectToMqtt() {
  Serial.println("Connecting to MQTT...");
  mqttClient.connect();
}

void connectEthernet() {
  bool result = ETH.begin(ETH_ADDR, ETH_POWER_PIN, ETH_MDC_PIN, ETH_MDIO_PIN,
    ETH_TYPE, ETH_CLK_MODE, true);
  if (!result) {
    Serial.println("ETH Init failed");
  }
}
```

Listing 7: ETH and MQTT connect

Message publishing functions:

- **publishMessageiBeacon()**: publishes an iBeacon message to the specified MQTT topic.
- **publishMessageEddy1()**: publishes an EddystoneTLM message to the specified MQTT topic.
- **publishMessageEddy2()**: publishes an EddystoneTLM message to another specified MQTT topic.

```
void publishMessageiBeacon(String message) {
  mqttClient.publish(MQTT_TOPIC_SEND, 0, true, message.c_str());
}

void publishMessageEddy1(String message) {
  mqttClient.publish(MQTT_TOPIC_SEND_2, 0, true, message.c_str());
}

void publishMessageEddy2(String message) {
  mqttClient.publish(MQTT_TOPIC_SEND_3, 0, true, message.c_str());
}
```

Listing 8. MQTT publish

The Message sending functions are:

- **sendMqttiBeacon()**: sends an iBeacon message to the MQTT server.
- **sendEddystoneTlmMqttMessage1()**: sends an EddystoneTLM message to the MQTT server.

```
void sendMqttiBeacon(const char *mac, uint16_t ID, uint16_t major, uint16_t minor, const
char* uuid, int8_t signalPower, int rssi) {
    if (mqttClient.connected()) {
        DynamicJsonDocument doc(200);

        doc["MAC"] = mac;
        doc["ID"] = ID;
        doc["Major"] = major;
        doc["Minor"] = minor;
        doc["UUID"] = uuid;
        doc["SignalPower"] = signalPower;
        doc["RSSI"] = rssi;

        String message;
        serializeJson(doc, message);
        publishMessageiBeacon(message);
        mqttBlink();
    } else {
        Serial.println("MQTT is not connected. Discarding message");
    }
}

void sendEddystoneTlmMqttMessage1(const char *mac, String uuid, uint16_t
batteryVoltage, float temperature, uint32_t advertiseCount, uint32_t timeSinceReboot) {
    if (mqttClient.connected()) {
        DynamicJsonDocument doc(200);

        doc["MAC"] = mac;
        doc["UUID"] = uuid;
        doc["BatteryVoltage"] = batteryVoltage;
        doc["Temperature"] = temperature;
        doc["AdvertiseCount"] = advertiseCount;
        doc["TimeSinceReboot"] = timeSinceReboot;

        String message;
        serializeJson(doc, message);
        publishMessageEddy1(message);
        mqttBlink();
    } else {
        Serial.println("MQTT is not connected. Discarding message");
    }
}
```

Listing 9. Send to MQTT

5.2.3 LoRa Settings (lora.h)

loralnit(): Initializes the LoRa module by setting up Serial communication and configuring the LoRa module with the required settings.

```
Serial1.begin(9600, SERIAL_8N1, LORA_RXD, LORA_TXD, false);
Serial.print("LORA start\r\n");
at_send_check_response("+AT: OK", 1000, "AT\r\n");
at_send_check_response("+MODE: TEST", 1000, "AT+MODE=TEST\r\n");
at_send_check_response("+TEST: RFCFG", 1000, "AT+TEST=RFCFG,868,SF7,125,8,8,
14,ON,OFF,OFF\r\n");
delay(200);
```

Listing 10. Lora settings

toHexString(const String &input): Converts a given string into a hex string, which is used for sending data over LoRa.

```
String toHexString(const String &input)
{
    String hexString = "";
    for (size_t i = 0; i < input.length(); ++i)
    {
        hexString += String(input.charAt(i), HEX);
    }
    return hexString;
}
```

Listing 11. String to HEX

sendLoRaIBeacon(): Sends an iBeacon message over LoRa with the given parameters, such as MAC address, ID, major and minor values, UUID, signal power, and RSSI.

```
void sendLoRaIBeacon(const char *mac, uint16_t ID, uint16_t major, uint16_t minor, const
char *uuid, int8_t signalPower, int rssi)
{
    // Check if it's time to send a LoRa message (every 2 minutes)
    if (millis() - lastLoRaSent >= 0) {
        char cmd[512];

        DynamicJsonDocument doc(256);
        doc["MAC"] = mac;
        doc["ID"] = ID;
        doc["Major"] = major;
        doc["Minor"] = minor;
        doc["UUID"] = uuid;
        doc["SignalPower"] = signalPower;
        doc["RSSI"] = rssi;

        String jsonString;
        serializeJson(doc, jsonString);

        String hexString = toHexString(jsonString);

        sprintf(cmd, "AT+TEST=TXLRPKT,\"%s\"\\r\\n", hexString.c_str());
        int ret = at_send_check_response("+TEST: TXLRPKT", 5000, cmd);

        if (ret)
            Serial.println("Sent lora iBeacon.\\n");
        else
            Serial.println("Send failed!\\r\\n\\r\\n");

        loraBlink();

        // Update the last time a LoRa message was sent
        lastLoRaSent = millis();
    }
}
```

Listing 12. Lora iBeacon

sendLoRaEddystoneTlm1(): Sends an Eddystone TLM (Telemetry) message (type 1) over LoRa with the given parameters, such as MAC address, UUID, battery voltage, temperature, advertise count, and time since reboot.

```
void sendLoRaEddystoneTlm1(const char *mac, String uuid, uint16_t batteryVoltage, float
temperature, uint32_t advertiseCount, uint32_t timeSinceReboot)
{
    // Check if it's time to send a LoRa message (every 2 minutes)
    if (millis() - lastLoRaSent >= 0) {
        char cmd[512];

        DynamicJsonDocument doc(256);
        doc["MAC"] = mac;
        doc["UUID"] = uuid;
        doc["BatteryVoltage"] = batteryVoltage;
        doc["Temperature"] = temperature;
        doc["AdvertiseCount"] = advertiseCount;
        doc["TimeSinceReboot"] = timeSinceReboot;

        String jsonString;
        serializeJson(doc, jsonString);

        String hexString = toHexString(jsonString);

        sprintf(cmd, "AT+TEST=TXLRPKT, \"%s\\r\\n\"", hexString.c_str());
        int ret = at_send_check_response("+TEST: TXLRPKT", 5000, cmd);

        if (ret)
            Serial.println("Sent lora Eddystone 1.\\n");
        else
            Serial.println("Send failed!\\r\\n\\r\\n");

        loraBlink();

        // Update the last time a LoRa message was sent
        lastLoRaSent = millis();
    }
}
```

Listing 13. Lora Eddystone beacon

5.2.4 Other Settings (setting.h)

Other settings for Ethernet, MQTT server, the MAC addresses of the BLE Beacons and LoRa connections.

```
#include <ETH.h>

// Ethernet settings
#define ETH_CLK_MODE ETH_CLOCK_GPIO16_OUT
#define ETH_POWER_PIN 5
#define ETH_TYPE ETH_PHY_LAN8720
#define ETH_ADDR 0
#define ETH_MDC_PIN 23
#define ETH_MDIO_PIN 18

// MQTT Settings
#define MQTT_HOST IPAddress(192, 168, 0, 103)
#define MQTT_PORT 1883
#define MQTT_TOPIC_SEND "BLEtoMQTT/iBeacon"
#define MQTT_TOPIC_SEND_2 "BLEtoMQTT/Eddystone1"
#define MQTT_TOPIC_SEND_3 "BLEtoMQTT/Eddystone2"

// BLE beacons MAC addresses
const std::string iBeaconMacAddress = "b8:d6:1a:5c:1e:c6";
const std::string EddyMacAddress1 = "30:ae:a4:19:78:56";
const std::string EddyMacAddress2 = "30:ae:a4:1e:98:62";

// LoRa Settings
#define LORA_RXD 12
#define LORA_TXD 13

// LED Settings
#define ESP_LED_PIN 32
#define LED_COUNT 3
#define CHANNEL 0
```

Listing 14. Source code settings

The LED status settings are shown in Listing 15.:

```
// Lora status
void loraBlink(){
  updateLed(0, CRGB::White);
  delay(150);
  updateLed(0, CRGB::Red);
}

// BLE status
void bleBlink(){
  updateLed(1, CRGB::White);
  delay(150);
  updateLed(1, CRGB::Blue);
}

// MQTT status
void mqttBlink(){
  updateLed(2, CRGB::White);
  delay(150);
  updateLed(2, CRGB::Green);
}
```

Listing 15. Status LEDS settings

5.3 Results

5.3.1 MQTT Data and Visualization

In Figure 37, the result of data collected is shown on MQTT server, three topics had been created, each for one beacon. A simple dashboard was made using Node-RED to demonstrate the use of data collected.

```

03/05/2023, 16.44.27  node: d94f4e90630ad7ec
BLEtoMQTT/Eddystone1 : msg.payload : Object
  ▼ object
    MAC: "30:ae:a4:19:78:56"
    UUID: "0000feaa-0000-1000-8000-00805f9b34fb"
    BatteryVoltage: 2900
    Temperature: 16
    AdvertiseCount: 3
    TimeSinceReboot: 27
03/05/2023, 16.44.27  node: db31e2cdf4beacad
BLEtoMQTT/iBeacon : msg.payload : Object
  ▼ object
    MAC: "b8:d6:1a:5c:1e:c6"
    ID: 76
    Major: 5
    Minor: 88
    UUID: "2d7a9f0c-e0e8-4cc9-a71b-a21db2d034a1"
    SignalPower: -59
    RSSI: -41
03/05/2023, 16.44.27  node: d55af2e2cfd5ffca
BLEtoMQTT/Eddystone2 : msg.payload : Object
  ▼ object
    MAC: "30:ae:a4:1e:98:62"
    UUID: "0000feaa-0000-1000-8000-00805f9b34fb"
    BatteryVoltage: 3300
    Temperature: 12.82999992
    AdvertiseCount: 386
    TimeSinceReboot: 5653

```

Figure 37. MQTT data collected

The Node-Red Dashboard is shown in Figure 38.

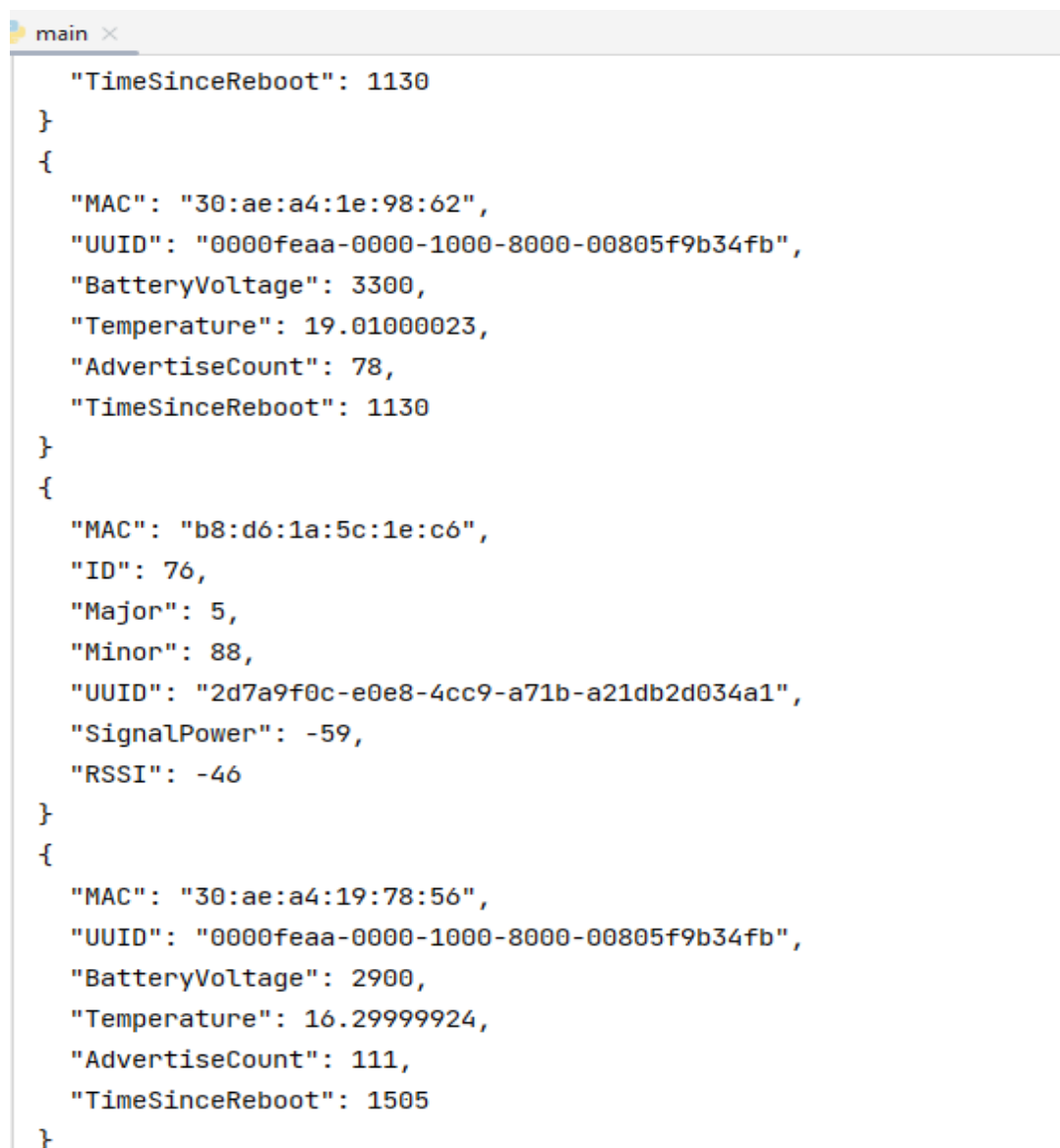


Figure 38. Node-Red Dashboard

The data collected from Eddystone beacons can be used to indicate temperature, chart, battery level. While the iBeacon is usually used for proximity applications, for example, the iBeacon can be used to trigger a LED when it is near the ESP32 Gateway.

5.3.2 Lora Server Data

There is no available Gateway from The Thing Networks to connect to in the region of Vaasa. To demonstrate the use case of LoRa module that acts as a backhaul channel to The Thing Networks LoRaWAN server, Wio-E5 lora devkit as a gateway was used, connected to a PC. The Devkit receives the data from the ESP32 Gateway, decodes them by a short Python script and prints them to the console log. An important note is this just “pure” Lora or just PHY layer, which means that every listener on the same channel can receive our message and no security was added.



```
main x
{
  "TimeSinceReboot": 1130
}
{
  "MAC": "30:ae:a4:1e:98:62",
  "UUID": "0000feaa-0000-1000-8000-00805f9b34fb",
  "BatteryVoltage": 3300,
  "Temperature": 19.01000023,
  "AdvertiseCount": 78,
  "TimeSinceReboot": 1130
}
{
  "MAC": "b8:d6:1a:5c:1e:c6",
  "ID": 76,
  "Major": 5,
  "Minor": 88,
  "UUID": "2d7a9f0c-e0e8-4cc9-a71b-a21db2d034a1",
  "SignalPower": -59,
  "RSSI": -46
}
{
  "MAC": "30:ae:a4:19:78:56",
  "UUID": "0000feaa-0000-1000-8000-00805f9b34fb",
  "BatteryVoltage": 2900,
  "Temperature": 16.29999924,
  "AdvertiseCount": 111,
  "TimeSinceReboot": 1505
}
```

Figure 39. LoRa server data

The Python script was used to decode the data received from the ESP32 Gateway to JSON string since the Lora module only sends HEX data.

```
import serial
import json
import codecs

def hex_to_string(hex_string):
    return codecs.decode(hex_string, 'hex').decode('utf-8')

def parse_received_data(data):
    # Remove +TEST: RX and quotes from the received string
    cleaned_hex_string = data[11:-1]
    json_string = hex_to_string(cleaned_hex_string)

    try:
        json_data = json.loads(json_string)
        return json_data
    except json.JSONDecodeError:
        print("Error decoding JSON")
        return None

def main():
    port = "COM12"
    baudrate = 9600

    ser = serial.Serial(port, baudrate, timeout=1)

    while True:
        data = ser.readline().decode('utf-8').strip()

        if data.startswith("+TEST: RX"):
            json_data = parse_received_data(data)

            if json_data is not None:
                print(json.dumps(json_data, indent=2))

if __name__ == "__main__":
    main()
```

Listing 16. Decode Python Script

6 CONCLUSIONS

The primary objective of this project was to create a low-cost and effective BLE gateway using the ESP32 chipset. Despite some challenges in the hardware design, all the goals have been achieved with satisfactory performance in terms of topology design, range, and data throughput. The project can be considered successful in delivering a functional BLE gateway that meets the intended criteria.

From an electronic point of view, there were some mistakes in the schematic that could be addressed in future iterations of the project. By refining the hardware design and correcting these errors, the performance and reliability of the BLE gateway could be further improved. Additionally, optimizing the selection of components and their placement on the PCB could contribute to a reduction in the bill of materials (BOM) cost.

In terms of software and firmware optimization, future work may involve enhancing the efficiency of the algorithms, improving power consumption, and exploring the possibility of incorporating additional features, such as support for multiple communication protocols or advanced security measures. These improvements could provide users with a more versatile and robust solution for their wireless communication needs.

Moreover, it would be valuable to conduct comprehensive testing in various real-world scenarios, which would allow for the identification of potential areas for improvement and the assessment of the performance of the gateway under different conditions. This would ultimately lead to the development of a more robust and reliable BLE gateway that meets the diverse requirements of users in various contexts.

In conclusion, the project has laid a solid solution for the development of affordable and high-performing BLE gateways. By addressing the identified limitations and incorporating the suggested improvements, there is significant potential for continued innovation and growth in the field of wireless communication technology. The project had highlighted the viability of ESP32-based solutions for creating cost-effective and efficient BLE gateways, opening new possibilities for the future of wireless connectivity.

REFERENCES

- /1/ What is a Bluetooth Gateway? Complete Guide 2023. Accessed 12.04.2023.
<https://www.dusuniot.com/blog/what-is-a-bluetooth-gateway/>

- /2/ Bluetooth SIG, Inc. (2021). Bluetooth Core Specification. Accessed 18.04.2023.
<https://www.bluetooth.com/specifications/bluetooth-core-specification/>

- /3/ Nordic Semiconductor. (2021). Bluetooth Low Energy - Introduction. Accessed 12.04.2023. <https://www.nordicsemi.com/Products/Low-power-short-range-wireless/Bluetooth-low-energy-LE>

- /4/ LoRa Alliance. (2021). LoRaWAN Specification. Accessed 04.05.2023.
<https://lora-alliance.org/resource-hub/lorawanr-specification-v102>

- /5/ MQTT.org. (2021). MQTT Protocol. Accessed 13.04.2023. <https://mqtt.org/>

- /6/ NXP Semiconductors. (2021). ESP32 Series Datasheet. Accessed 13.04.2023.
<https://www.nxp.com/products/wireless-connectivity/2-4-ghz-solutions/esp32-series-datasheet:P-DS-ESP32-SERIES>

- /7/ Wio-E5 module. Accessed 13.04.2023.
[Wio-E5 STM32WLE5JC lora module, embedded SX126X and MCU for LoRaWAN Wireless Sensor Network & IoT devices - EU868 & US915 - Seeed Studio](#)

- /8/ Semtech Corporation. (2021). LoRa Modulation Basics. Accessed 04.05.2023.
<https://www.semtech.com/products/wireless-rf/lora-transceivers/applications/LoRa-Modulation-Basics>

- /9/ Microchip's LAN8720A/LAN8720Ai datasheet. Accessed 13.04.2023.
[LAN8720A | Microchip Technology](#)

- /10/ The Things Network. (2021). LoRaWAN Gateway. Accessed 04.05.2023.
<https://www.thethingsnetwork.org/docs/gateways/>

APPENDIX I

Schematic Design

