Acknowledgements

We extend our heartfelt gratitude to all the individuals and organizations who supported us throughout this project. Their guidance, encouragement, and expertise were invaluable, and we could not have completed this without their help.

Table of Contents

I. Project Presentation

1. Context and Motivation

As final-year students in embedded systems, we wanted to expand our knowledge beyond the concepts we had already explored in our practical coursework. While we had worked with certain embedded technologies in previous lab sessions, we aimed to take a different approach by incorporating industry-demanded tools and technologies that are widely used in professional environments and integrating them into a fully functional system.

This project provided an opportunity to explore Real-Time Operating Systems (RTOS) for advanced task scheduling, customized Linux kernel optimization for embedded performance tuning, custom electronic board and wireless communication for efficient data transmission.

2. Project Objectives

The system is designed to acquire environmental parameters such as temperature, humidity, pressure, light intensity, and gas concentration, which are particularly useful for monitoring air quality and

enabling responsive actions. Additionally, this system is particularly well-suited for data acquisition in remote or hard-to-access areas, where traditional connectivity is limited. By leveraging LoRa technology, sensor data can be transmitted over long distances with minimal power consumption, ensuring continuous environmental monitoring in areas where conventional networks are impractical or unavailable. This capability makes the system highly effective for applications such as environmental monitoring in rural or industrial zones, smart agriculture, and air quality assessment in isolated locations, ultimately providing a reliable and efficient solution for real-time data collection and decision-making.

3. Overall Vision

Key Focus Areas of the Project

- ❖ Real-time data acquisition.
- ❖ Data transmission via a LoRa module.
- ❖ Customization of a Linux kernel before integration into a Raspberry Pi.
- ❖ Data visualization using Blynk
- ❖ Feedback using a full custom electronic board.

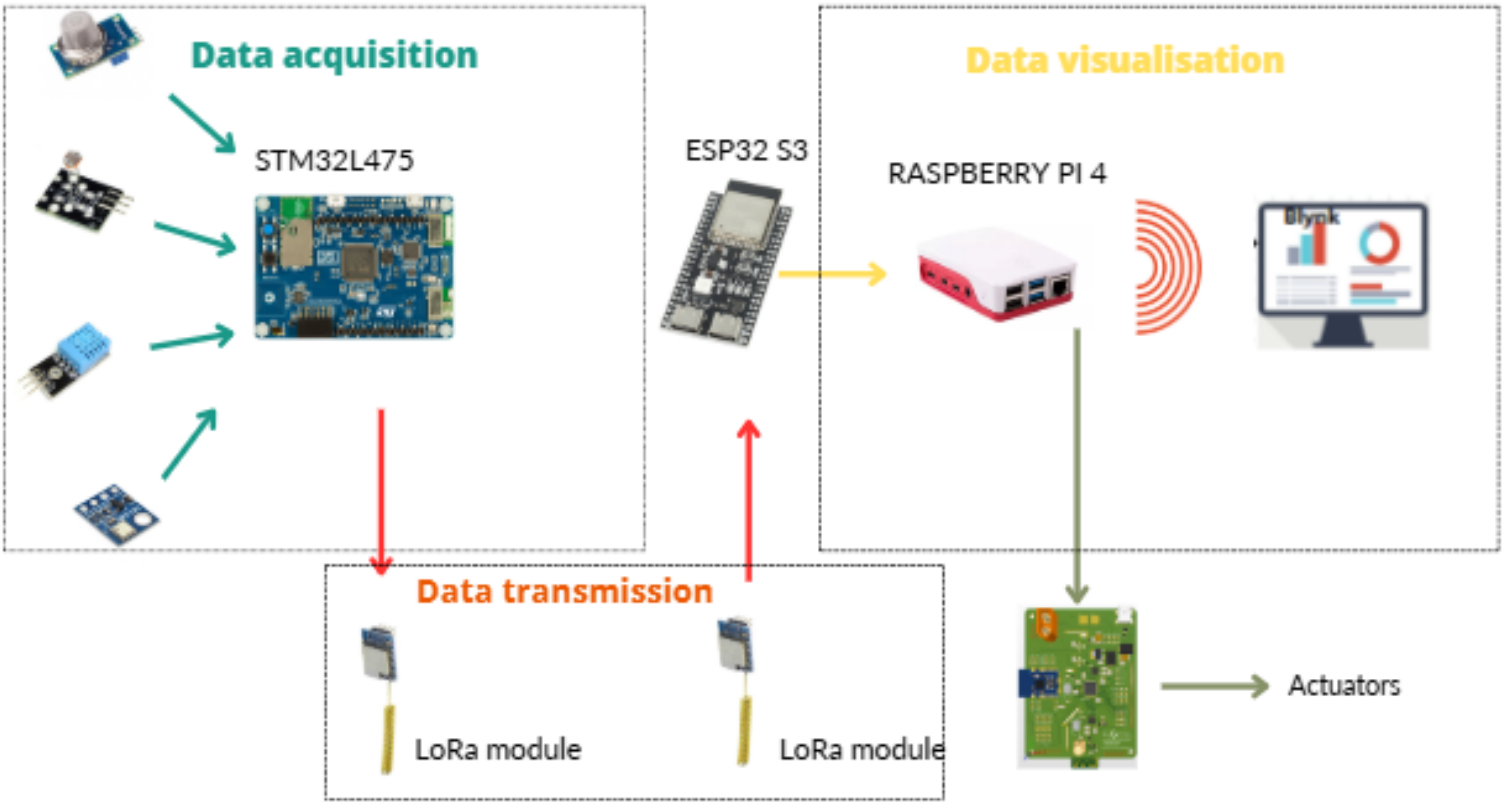


Figure 1.1.1 : Overall system overview

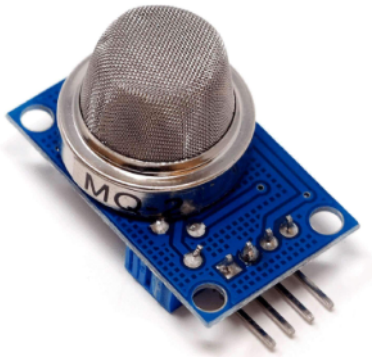
II. Hardware and Software Utilized

Part	Hardware	Software
Data Acquisition	• STM32L475 • DHT22 • MQ-2 • BMP180 • Photoresistor	• Arduino IDE
Data Transmission	• LoRa module SX1728 433MHz • ESP32-S3-WROOM-1	• Arduino IDE
Data Visualization	• Raspberry Pi 4	• Blynk • Node-RED
Feedback	• Custom STMF070	• Blynk • Node-RED

Figure 3.1.1 : Table of HW/SW Utilized

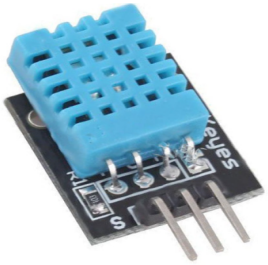
Technical Specifications of sensors

1. Gas sensor : MQ-2



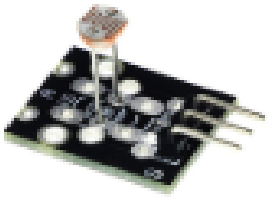
- **Operating Voltage:** 5V
- **Detects** : LPG, CH_4, CO, H_2, Alcohol, Propane, Butane, Smoke
- **Current Consumption:** ~150mA
- **Output:** Analog (0-5V) & Digital (adjustable threshold)
- **Preheating Time:** 24-48h for stable readings
- **Response Time:** <10s
- **Adjustable Sensitivity:** Via onboard potentiometer
- **Operating Temperature:** -10°C to +50°C
- **Humidity Range:** Up to 95% RH

2. Temperature & Humidity Sensor : DHT11



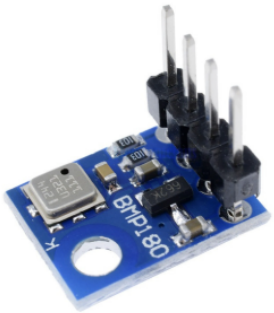
- **Operating Voltage:** 3.3V - 5V
- **Temperature Range:** 0°C to 50°C (±2°C accuracy)
- **Humidity Range:** 20% to 90% RH (±5% accuracy)
- **Output:** Digital (One-Wire communication protocol)
- **Update Rate:** 1 reading every 2 seconds
- **Current Consumption:** 2.5mA (active mode)

3. Light sensor



- **Operating Voltage:** 3.3V - 5V
- **Resistance Variation:**
  - **Bright Light:** 1kΩ - 10kΩ
  - **Darkness:** ~1MΩ
- **Output: Analog (Voltage divider with a 10kΩ fixed resistor)**
- **Response Time:** Few milliseconds

4. Pressure sensor : BMP180

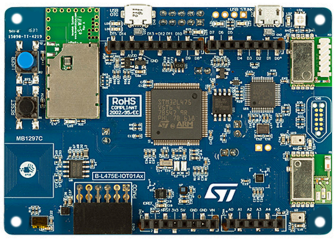


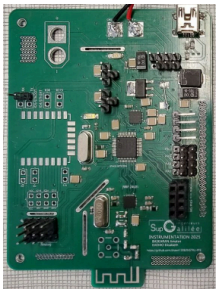
- **Operating Voltage:** 3.3V - 5V
- **Pressure Range:** 300 - 1100 hPa
- **Measurement Accuracy:** ±0.02 hPa (~0.17m altitude resolution)
- **Communication: I2C Communication**
- **Output : Digital**
- **Current Consumption:** 5µA (sleep mode), 12mA (active mode)
- **Operating Temperature:** -40°C to +85°C
- **Response Time:** <7.5 ms

Summary Table of Sensors

Sensor	Output	Measurement Range	Operating Voltage	Communication	Response Time	Applications
MQ-2 (Gas)	Analog/ Digital	300-10000 ppm (Combustible gas)	5V	ADC / Threshold	<10s	Air quality, gas leak detection
LDR (Light)	Analog	Light intensity	3.3V - 5V	ADC / Comparator	Few ms	Automatic lighting, solar tracking
DHT11 (Temp & Humidity)	Digital	0-50°C, 20-90% RH	3.3V - 5V	One-Wire	6s - 15s	Weather monitoring, HVAC
BMP180 (Pressure)	Digital	300-1100 hPa	3.3V - 5V	I2C	<7.5ms	Weather stations, drones, altimeters

Technical Specifications of Development Boards





NAME	B-L475E-IOT01A	ESP32-S3 DevKitC	Custom Board
µProcessor	STM32 L4	ESP32-S3	STM32 F0
IDE	Arduino IDE	Arduino IDE	Arduino IDE
Specs	BLE, Sup-Ghz, Wifi, NFC.	Wi-Fi & Bluetooth	OTA + RF (Planned)
Sensors	Microphones, IMU, Barometer, Temperature Time-of-Flight	None	IMU

III. Data acquisition

1. FreeRTOS

FreeRTOS is an open-source real-time operating system (RTOS) designed for microcontrollers and embedded processors. It features a lightweight kernel with a task scheduler that enables efficient multitasking while optimizing hardware resource usage.

Compatible with a wide range of platforms, FreeRTOS is adaptable to various embedded environments. Its modular architecture and low memory footprint make it a suitable solution for systems requiring precise control over execution time.

a. FreeRTOS and other RTOS

The table below provides a comparative analysis of four RTOS commonly used in embedded systems: FreeRTOS, Zephyr OS, ChibiOS, and RTEMS. These RTOS options are evaluated based on key criteria such as license type, memory footprint, microcontroller support, ease of use, security, community support, and certification availability.

Criteria	FreeRTOS	Zephyr OS	ChibiOS	RTEMS
License	MIT	Apache 2.0	GPL3	BSD
Memory Footprint	Low	Medium	Very Low	Medium
Microcontroller Support	Large	Large	Medium	Medium
Ease of Use	High	Medium	Medium	Low
Security	Good	Excellent	Low	Excellent
Community & Support	High	High	Medium	Low
Certification	Available	Available	Not Available	Available

Figure 3.1.1 : Table of comparison between FreeRTOS and other popular RTOS

Based on this comparison, FreeRTOS emerges as a good choice for our project. It is open-source and offers essential features such as ease of use, broad microcontroller support, and efficient memory management, making it well-suited for our requirements.

b. Key features of FreeRTOS

FreeRTOS provides several essential features that enable efficient real-time multitasking and resource management in embedded systems. Below are some of the key features and their uses:

1. **Preemptive and Cooperative Scheduling**
    a. Supports preemptive scheduling, where higher-priority tasks interrupt lower-priority ones.
    b. Also allows cooperative scheduling, where tasks voluntarily yield control.
2. **Task Management**
    a. Enables the creation of multiple tasks, each with its own priority and execution flow.
    b. Allows dynamic task creation and deletion.
3. **Inter-Task Communication (Queues, Semaphores, and Mutexes)**
    a. Queues: Enable safe data exchange between tasks.
    b. Semaphores: Manage synchronization and prevent task conflicts.
    c. Mutexes: Prevent simultaneous access to shared resources, avoiding race conditions.
4. **Software Timers**
    a. Allows scheduling of periodic or one-time task execution without blocking other processes.
    b. Useful for implementing timeouts, delays, and periodic updates.

2. RTOS implementation in our project

Before implementing our system in RTOS, we explored the possibility of doing our system in a non-RTOS in order to evaluate the effect of this and see if a RTOS was really necessary. Results of these experiments can be seen in the Test and result section.
Based on the results obtained from these experiments, this project implements a RTOS using FreeRTOS on an STM32L475 microcontroller within the Arduino IDE to efficiently manage and acquire data from multiple environmental sensors. The system is designed to ensure reliable, time-sensitive data processing while optimizing resource usage through multitasking and inter-task communication mechanisms.

a. System Overview

The system integrates four key environmental sensors:

➔ Temperature and Humidity Sensor: Captures ambient temperature and humidity levels.

➔ Pressure Sensor: Measures atmospheric pressure variations.

➔ Photoresistor (LDR): Detects ambient light intensity.

➔ Gas Sensor: Monitors air quality by measuring gas concentration levels.

To ensure a deterministic and responsive execution, the system employs preemptive scheduling, allowing higher-priority tasks to interrupt lower-priority ones when necessary.
Additionally, queues are implemented to manage inter-task communication, preventing data corruption and ensuring synchronized sensor readings.

b. Task Management

The system operates through five dedicated tasks, each responsible for handling specific sensor data and communication:

➔ Task 1: Acquires temperature and humidity data.

➔ Task 2: Reads atmospheric pressure measurements.

➔ Task 3: Measures ambient light intensity using a photoresistor.

➔ Task 4: Reads data from the gas sensor to measure specific gas concentration.

➔ Task 5: Transmits collected data via a LoRa module for remote monitoring.

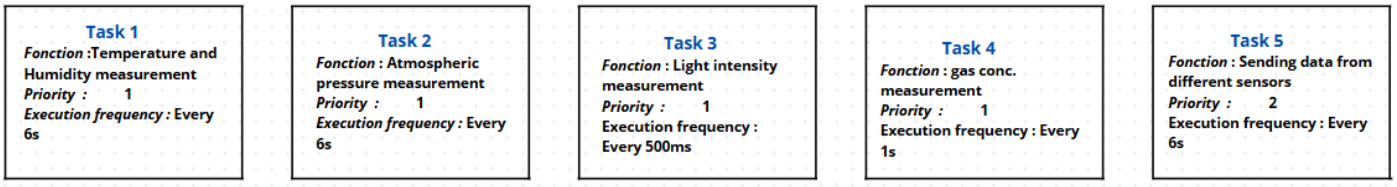


Figure 3.2.1 : Tasks used under freeRTOS

In order to prevent data corruption or overlapping outputs which result from the fact that multiple tasks attempt to write simultaneously on the serial monitor , only Task 5 has access to the serial monitor. By centralizing serial communication within a single task, we ensure a structured and conflict-free data transmission, improving the readability and reliability of the system's output. Additionally, this approach helps reduce CPU overhead, as only one task is responsible for handling the final data output, while other tasks focus on acquiring and processing sensor readings efficiently.

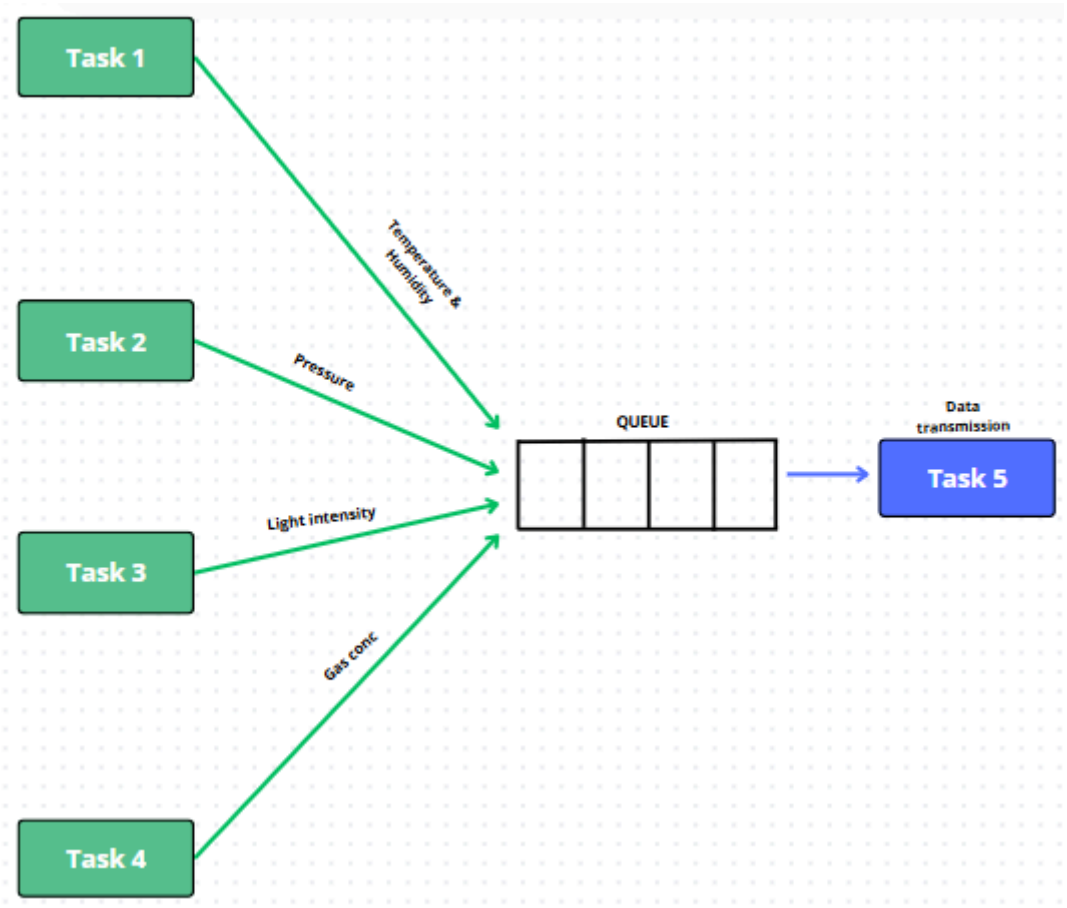


Figure 3.2.2 : Queue implementation for data acquisition

One of the main challenges was determining the optimal frequency for each sensor to read data and send it to the queue. Several factors needed to be considered, including the duty cycle of our LoRa module detailed in the Data Transmission and Communication section, the intended application of our system, and a sampling rate that ensures the data displayed on Node-RED appears dynamic rather than static, providing the impression of continuous updates.

For environmental monitoring applications, parameters such as atmospheric pressure change gradually. Given this, we opted to read the pressure sensor data at a longer interval than its refresh rate. The sensor provides new data every 25ms, but since atmospheric pressure changes gradually, such a high frequency is unnecessary.

The technique we implemented for sensors measuring rapidly changing environmental parameters, such as ambient light intensity and gas concentration, involved reading data as frequently as possible to minimize data loss. The collected readings were then stored and averaged before sending the processed data to the queue every 6 seconds. Therefore, we decided to read the sensor every 6 seconds. Similarly, to minimize data loss from our light intensity sensor, we chose to read its value every 500ms, storing these readings over a 6-second period and averaging the 12 collected values. As

a result, the graph displayed in Node-RED effectively represents light intensity variations over a 6-second interval.

This 6-second interval was chosen as it aligns with the response time of the slowest sensor in our system, ensuring that all sensors provide accurate and meaningful data while minimizing information loss.

Additionally, this approach helped us comply with the duty cycle regulations imposed on LoRa transmissions, as further detailed in the *Implementing LoRa communication in our system* section.

IV. Data Transmission and Communication

Choosing the correct communication system is crucial for the efficiency and reliability of environmental monitoring systems. In wireless communication, three key aspects must be considered: energy consumption, data transmission rate, and range.

An optimal system ensures a balance between these aspects, but achieving this balance is not always straightforward. As a result, a trade-off is often necessary. For example, Wi-Fi can transmit data at high speeds, reaching up to GHz frequencies, but it consumes a significant amount of energy and has a limited range of only a few meters. The selection of the appropriate data transmission protocol and the aspect that can be compromised depends on the specific requirements of the application in which the protocol is used.

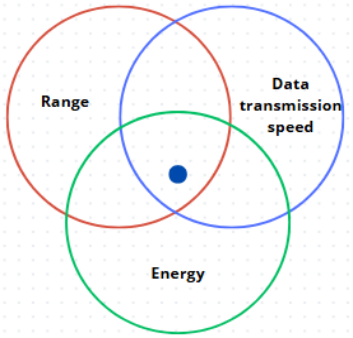
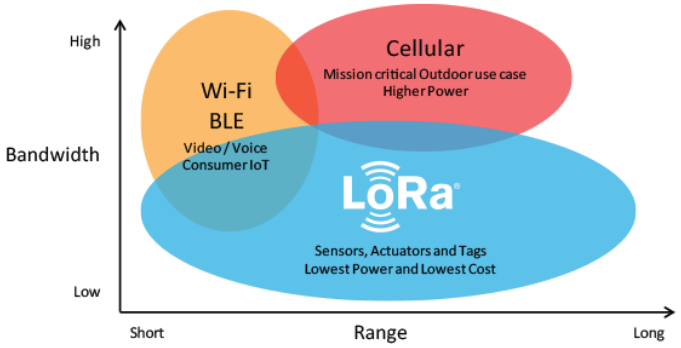


Figure 4.0.1 : Key features in wireless transmission

For our environmental monitoring application, range is the key factor. Additionally, the communication system must be highly energy-efficient, as this system is particularly used in remote or hard-to-reach locations where power availability is limited. Considering these requirements, we chose to use a LoRa module.

1. LoRa and Its Advantages

LoRa technology is a low-power, wide-area network designed for long-distance communication with minimal energy consumption (***Figure 4.1.1***). It's commonly used in IoT applications, such as environmental monitoring, due to its ability to cover large areas, handle



numerous devices. LoRa is cost-effective and secure, making it an ideal choice for large-scale IoT deployments.

Figure 4.1.1 : LoRa Overview

a. LoRa Protocol & Chirp spread spectrum (CSS)

The LoRa protocol uses Chirp Spread Spectrum (CSS) modulation techniques for data transmission. This modulation technique plays a crucial role in enabling the long-range, low-power communication that LoRa is renowned for.

This method involves using chirp signals, which are sinusoidal signals whose frequency increases or decreases over time (***Figure 4.1.2***). Imagine sending a message over a long distance with a unique "chirp" pattern that can be easily recognized by the receiver, even if there's a lot of noise around. This "chirp" pattern ensures that the message is clear and identifiable despite the interference.

CSS modulation is known for its robustness and reliability. It is resistant to Doppler effects, which occur when there is relative motion between the transmitter and receiver, causing frequency shifts. This resistance makes CSS modulation highly suitable for mobile radio or IoT applications

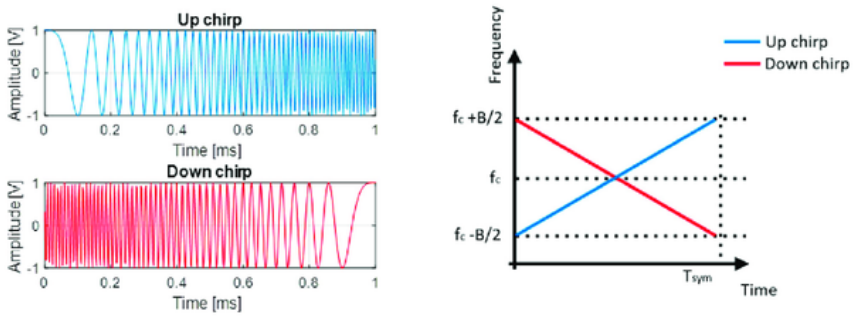


Figure 4.1.2 : Chirp / Frequency over time

During data transmission, data is firstly encoded into symbols, typically binary values, which are then mapped to specific chirp signals. The spreading factor (SF) determines the number of chirps used to encode each symbol, with higher SFs providing better range but lower data rates. Frequency hopping can be employed to avoid interference by changing the carrier frequency according to a predefined pattern. Error correction techniques are applied to add redundancy to the data, allowing the receiver to detect and correct errors without needing retransmission.

Let's take an example to simplify the concept of SF. A SF of 2 means we can hold 2^SF values, which corresponds to four possible values: {00, 01, 10, 11}. With the following graph, we can see how data is encoded using CSS modulation. However, in practice, SF typically ranges between 6 and 12.

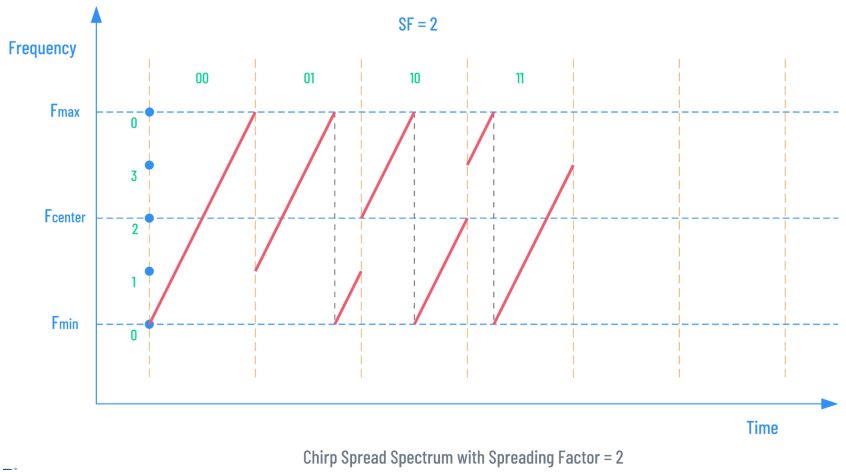


Figure 4.1.3 : Example SF = 2

In Europe, LoRa and sub-GHz LPWAN technologies are regulated to ensure fair spectrum access and prevent interference. Devices can typically transmit for only 1% of the time within the 863-870 MHz band. Each sub-band has specific duty cycle limits, and devices must have a "time off sub-band" period after transmitting to allow other devices to use the spectrum. Additionally, there are limits on transmission power to minimize interference.

However, LoRa alone does not define a network protocol; this is where LoRaWAN (LoRa Wide Area Network) comes in.

b. LoRaWAN

LoRaWAN builds upon the LoRa protocol by adding a network layer that manages communication between devices and gateways, and ultimately to the application server (***Figure 4.1.4***). While LoRa focuses on the physical layer of long-range, low-power wireless data transmission, LoRaWAN defines the network architecture, including addressing, routing, and security. This allows for scalable and interoperable IoT networks, making it easier to deploy and manage large numbers of devices across various applications.

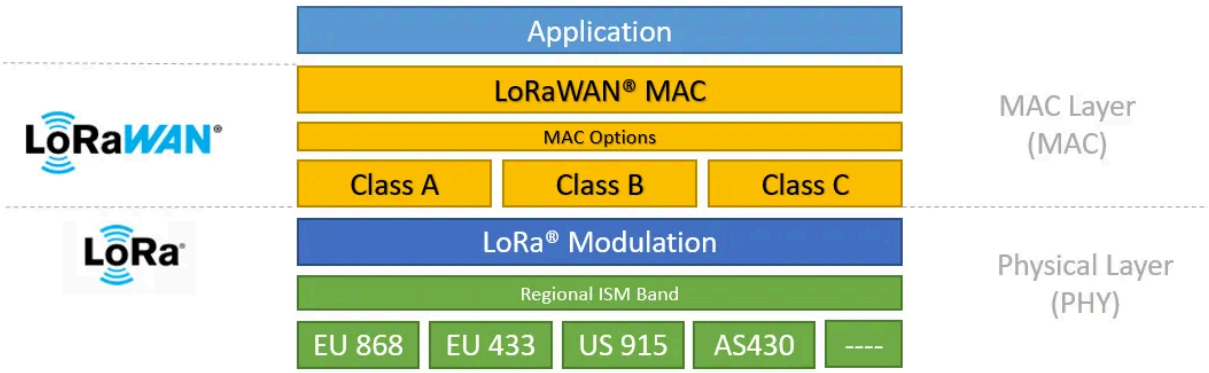


Figure 4.1.4 : LoRa Protocol Stack

c. Details of the RA-01 SX1278 Lora module

For our project, we decided to use the RA-01 SX1278 module from Semtech. The RA-01 SX1278 operates in the 433 MHz frequency band and is designed for IoT projects, sensor networks, and other applications requiring long-range communication. This module features a maximum output power of +20 dBm, a sensitivity of -148 dBm, and can achieve a communication range of up to 10 km under optimal conditions. It supports various modulation methods, including LoRa, which we will use in this project. With its low power consumption, it is ideal for battery-powered devices. Additionally, the RA-01 includes an SPI interface, making it easy to integrate with microcontrollers.

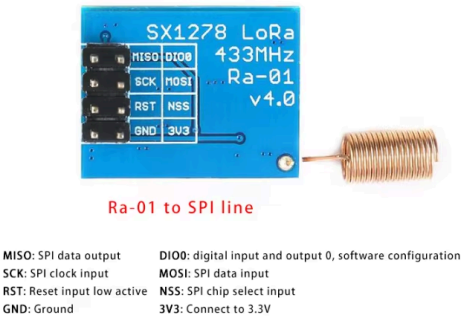


Figure 4.1.5 : RA-01 433MHz

LoRa operates within the ISM (Industrial, Scientific, and Medical) band, which is a range of radio frequencies designated for non-commercial applications and widely used for license-free wireless communication. The ISM band at 433 MHz, specifically in the range of 433.05 MHz to 434.79 MHz, is authorized in Europe for low-power communications, making it suitable for LoRa devices. This frequency band allows LoRa modules, such as the SX1278, to transmit data over long distances without requiring a license, as long as they adhere to power and duty cycle regulations.

However, while the ISM band is license-free, its use is subject to strict regulations to avoid interference with other communication systems. In Europe, the European Telecommunications

Standards Institute (ETSI) mandates compliance with EN 300 220, which limits the maximum transmission power to 10 mW (+10 dBm). Additionally, devices operating in this band must comply with duty cycle restrictions, meaning they cannot transmit continuously. The duty cycle for the 433 MHz band is limited to 10%, meaning a device can only be active for a total of 360 seconds per hour. Below is an extract of the document that contains this information.

Bande de fréquences	Puissance max.	Paramètres additionnels	Références / observations
433,05 à 434,79 MHz	1 mW p.a.r. -13 dBm/10 kHz pour une largeur de bande de modulation supérieure à 250 kHz	Applications vocales : mise en œuvre de techniques d'accès au spectre et d'atténuation des interférences.	Décision 2006/771/CE modifiée (bande n° 44a). Les applications audio autres que vocales et les applications vidéo sont exclues.
	10 mW p.a.r.	Coefficient d'utilisation limite : 10%	Décision 2006/771/CE modifiée (bande n° 44b)

Figure 4.1.6 : Extract of the AFNR document

The full document can be accessed via the link provided in the Appendices section of this document.

d. Implementing LoRa communication in our system (TX)

i) Lora transmitter.

The LoRa module connected to the STM32 microcontroller was used to transmit sensor data via a Serial task. The microcontroller and the LoRa module communicate via SPI communication protocol.

With this module and using the appropriate library, we can configure LoRa module characteristics such as the frequency band, spreading factor, bandwidth, coding rate, preamble length, sync word, and transmission power using available functions.

```
LoRa.setSpreadingFactor(SPREADING_FACTOR);
LoRa.setSignalBandwidth(BANDWIDTH);
LoRa.setCodingRate4(CODING_RATE);
LoRa.setPreambleLength(PREAMBLE_LENGTH);
LoRa.setSyncWord(SYNC_WORD);
LoRa.setTxPower(TX_POWER);
```

In Task 5 (Task Management), we implemented the transfer data using the LoRa module. When determining the transmission frequency, it was essential to consider regulatory constraints, particularly the duty cycle limitations. To comply with these regulations, we carefully optimized our

data transmission intervals, balancing the need for timely environmental monitoring with adherence to legal transmission constraints.

Determining the optimal time interval for transmitting data from the LoRa module's transmitter to the receiver was a challenging task. It was crucial to ensure that sensor data was read at regular intervals to prevent the loss of important information. Additionally, we had to consider a fault-tolerant mechanism, ensuring that a failure in one sensor would not disrupt the entire data transmission process.

To select the appropriate interval, we analyzed the typical application of our system. LoRa-based communication is particularly suited for long-distance data transmission, making it ideal for monitoring environmental parameters in remote areas where real-time data collection is essential.

We decided to fix :
SF : 8
BW : 250kHz
CRC : 0
Preampule : 12

Duty cycle(%) = $\left(\frac{time\ On\ Air(ms)\ \ x\ Number\ of\ transmissions\ per\ hour}{3600000}\right) x\ 100$

Time On Air (ToA) (ms): Time required to send a LoRa message.
Number of transmissions per hour: Number of times the LoRa module transmits data in one hour.
3,600s:
Total number of milliseconds in an hour (1h = 3600 seconds = 3,600,000 ms).
10% max in France on 433 MHz → The module must not exceed 360s of transmission per hour.
The Time On Air of our project was determined to be 54ms. The formula used to calculate this time is found in the datasheet of the device.
Concerning our application, we decided to send 1 message every 6 seconds (60 messages per hour).
=> Duty cycle(%) = 0.9 << 10. As demanded by the regulation.

ii) Lora receiver

The LoRa module connected to the ESP32 functioned as the receiver in our setup. For successful communication, various parameters configured on the transmitter side, such as the Spreading Factor (SF), Bandwidth (BW), and Preamble Length, had to be identical on the receiver side. Ensuring these settings matched on both modules was essential for achieving efficient and reliable data transmission between the two LoRa modules.

On the transmitter side and receiver side, a LED was used for status indication. When a message is transmitted or received, the LED toggles, confirming successful communication. This approach provides a simple and efficient way to monitor and evaluate the communication reliability between the two LoRa modules.

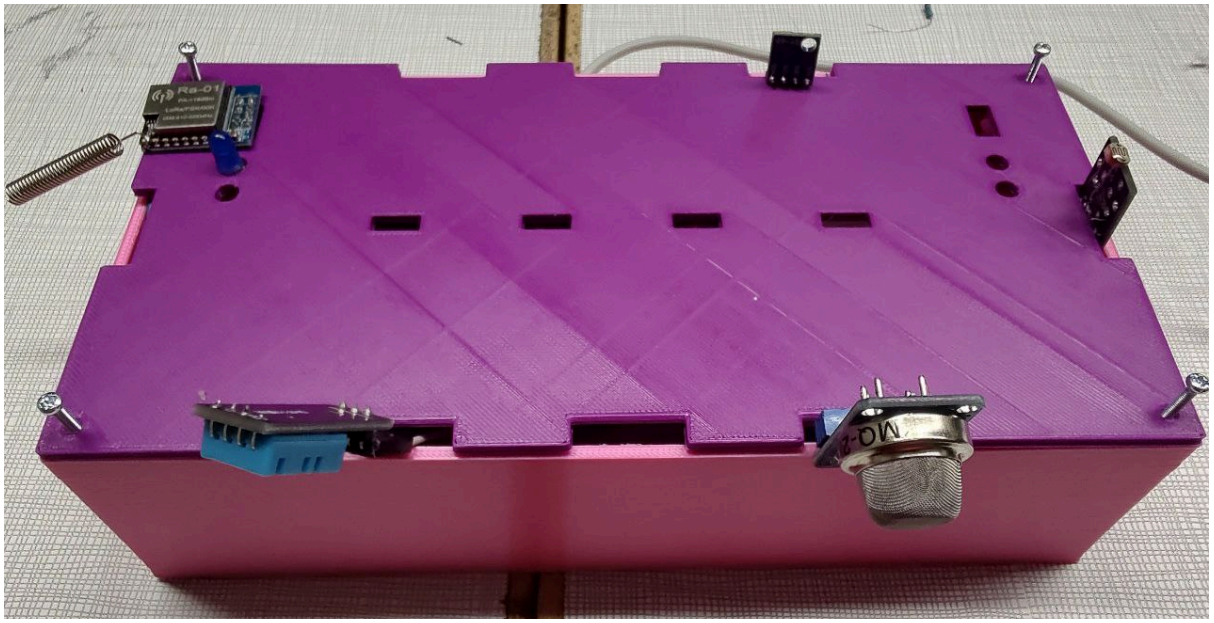


Figure 4.1.7 : Data Acquisition system transmitter design

Connected to our ESP32 which is responsible for receiving data from the LoRa module, is a Raspberry Pi 4, serving as the platform for the development and implementation of an IoT solution.

This system is structured around four core components:

1. Embedded Linux with Yocto Configuration – Providing a customized and lightweight operating system tailored for IoT applications.
2. Data Monitoring with Node-RED – Enabling real-time visualization and data flow management.
3. Application Development and Integration with Blynk Cloud – Facilitating remote control and monitoring via a cloud-based interface.
4. PCB Design – Used to respond to user action via Blynk interface.

V. Implementation of a custom linux kernel

We decided to generate a custom Linux kernel for our project, allowing us to include only the necessary modules required for our application. This approach is commonly used in embedded systems to minimize latency and optimize memory management, ensuring efficient resource utilization.

To generate our Linux kernel, we chose Yocto, which is a flexible and widely adopted framework for creating custom Linux distributions tailored to specific hardware and application requirements.

a. Presentation and configuration of The Yocto project

The Yocto Project is a collaborative project of the Linux Foundation launched in 2010 to facilitate the creation of Linux distributions for embedded systems. It aims to provide tools and processes to quickly and repeatedly develop customized embedded Linux systems.

The Yocto Project works closely with OpenEmbedded (***Figure 5.1.1***), an open-source community project. Together, they share the maintenance of several key components, such as the BitBake build engine and the OpenEmbedded-Core metadata layer.

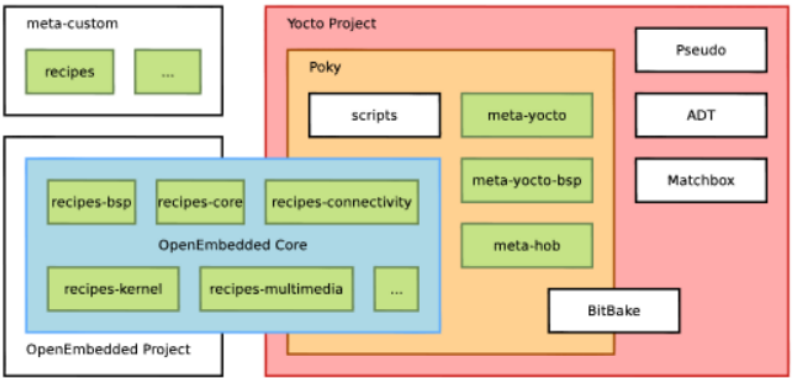


Figure 5.1.1 : Yocto and OpenEmbedded

The Yocto Project comprises multiple layers (***Figure 5.1.2***), each serving a unique purpose. The foundational layer, OE core (meta), provides the essential metadata necessary for building distributions. The Yocto layer (meta-yocto-bsp / meta-poky) includes the Board Support Package (BSP) and Poky reference distribution. BSP layers, such as meta-raspberrypi, cater to specific hardware platforms, ensuring compatibility and optimization. User interface layers, offer various UI components, while commercial layers include offerings from open-source software vendors. Finally, the user-specific layer allows for customizations unique to individual needs. This structured approach facilitates the creation of flexible, optimized, and highly customizable embedded Linux distributions.

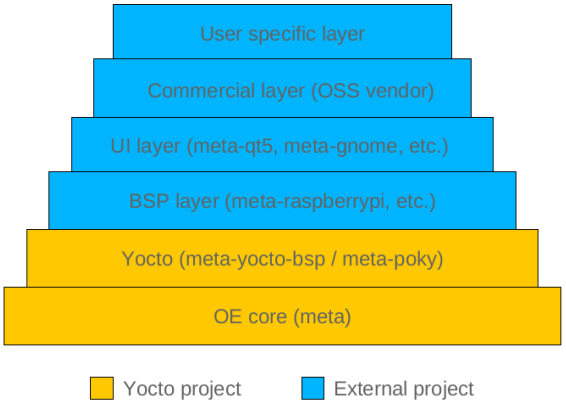


Figure 5.1.2 : Yocto Layers

The Yocto Project is regularly updated, with recent enhancements focusing on performance optimization, security improvements, and broader hardware support. The latest versions incorporate updated Linux kernels and advanced debugging tools (***See Appendix Figure A.5.1***), aiming to improve system stability and development efficiency.

As you can see, the current version is Styhead, and the upcoming version is Welnascar. Both versions offer long-term support (LTS), with Kristone 4.0 and ScarthGap 5.0 being the LTS releases.

b. Prerequisites

To begin working with the Yocto Project, a Linux-based system is required. For our project, the host machine used to configure Yocto runs Fedora 39, an open-source Linux distribution sponsored by Red Hat. Ensuring that the Linux system is properly configured with the necessary packages and settings is essential for a successful Yocto Project setup.

Below are the essential packages required for downloading and installing Yocto:

```
$ sudo yum install gawk make wget tar bzip2 gzip python unzip perl patch diffutils diffstat git cpp gcc gcc-c++ glibc-devel texinfo chrpath ccache perl-Data-Dumper perl-Text-ParseWords perl-Thread-Queue socat findutils which SDL-devel xterm
```

More info about installation : https://docs.yoctoproject.org/5.0.6/singleindex.html

c. Directory Structure

Experts from the Yocto Project recommend maintaining a well-structured working directory to ensure a smooth development environment and facilitate efficient navigation between directories. Proper

directory organization helps streamline the build process, manage dependencies, and maintain clarity in system configurations.

When Yocto is downloaded, it includes several directories, each serving a specific purpose, such as source files, build configurations, metadata, and output artifacts. These directories play an important role in the customization and compilation of the embedded Linux system.

Before starting the customization process, we created a main directory "YOCTOS" which contained our entire project:

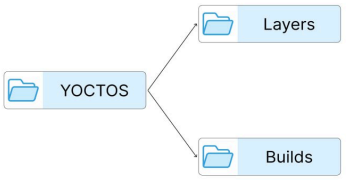


Figure 5.1.3 : Main Directory

In the “Layers” Directory, we clone all Yocto core :

```
Fedora:~/<Dir>/YOCTOS/Layers $
git clone git://git.yoctoproject.org/poky -b scarthgap
git clone git://git.openembedded.org/meta-openembedded -b
scarthgap
git clone git://git.yoctoproject.org/meta-raspberrypi -b
scarthgap
```

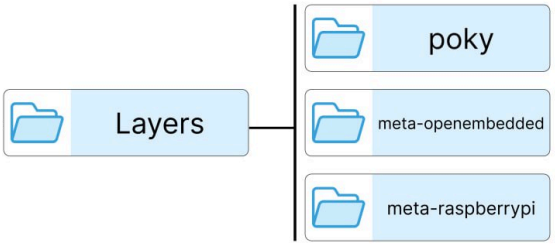


Figure 5.1.4 : Layer Directory

After cloning the three essential Yocto core components, the next step is to initialize the build environment for the project in the "Builds" directory.

```
Fedora:~/<Dir>/YOCTOS/Builds $
source /../Layers/poky/oe-init-build-env Build_pi64_pfe
```

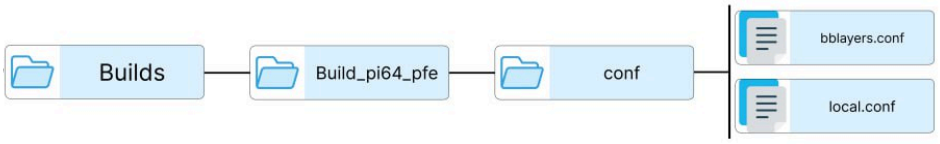


Figure 5.1.5 : Builds Directory

The two files we modified in our Yocto Project were “bblayers.conf” and “local.conf”. These are the only files that require changes for our project setup.

d. Functionality

- **bblayers.conf** : This file defines the layers that are part of the build environment. Layers are collections of recipes, classes, and configurations that can be used to build images. The file includes a list of directories where these layers are located. Each layer contributes additional functionality or support for different hardware platforms (***Figure 5.1.6***).

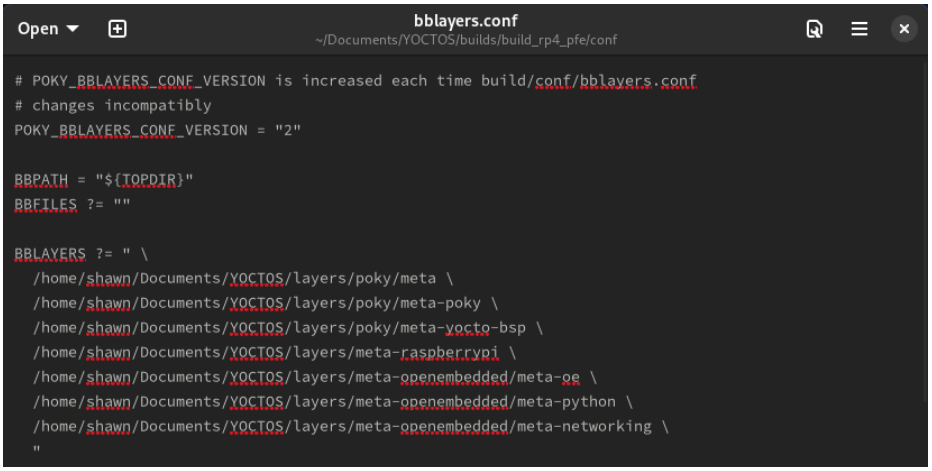


Figure 5.1.6 : bblayers.conf

- **local.conf** : This file contains local configuration settings specific to the build environment and project. It allows you to customize the build process by setting various parameters. The file includes settings such as the target machine, parallel build settings, package management configuration, and any additional customizations needed for your project (***Figure 5.1.7***).

We set the target machine by defining `MACHINE = "raspberrypi4-64"` and adding the following lines to enable UART and include default packages like the Nano text editor and WiFi configuration files.

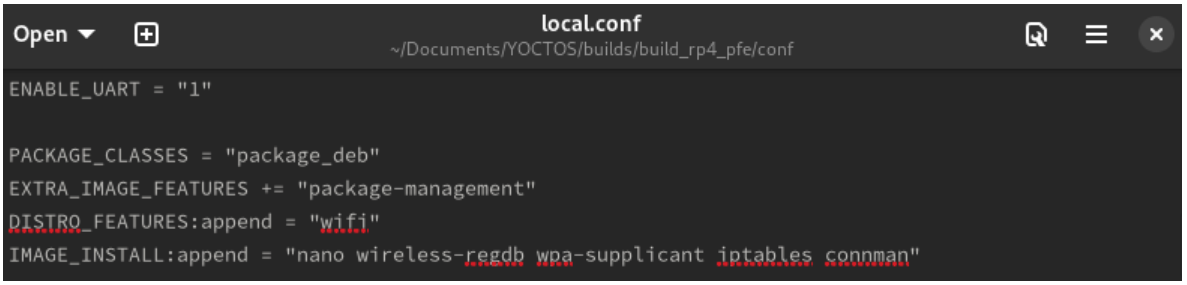


Figure 5.1.7 : local.conf

Once both files are configured, we can proceed to bake the Linux images. There are several options available for baking images using the bitbake command:

- ➔ **core-image-minimal**: A small, minimal image that is just capable of allowing a device to boot. It includes only the essential components needed to get the device running.
- ➔ **core-image-full-cmdline**: Similar to core-image-base, this console-only image provides additional full-featured Linux system functionality. It offers a more comprehensive environment for command-line operations.

There are other options available, but these should be sufficient for our project.

We chose core-image-full-cmdline because it provides an efficient command-line environment, offers the necessary features for embedded development, and remains lightweight, ensuring optimal performance for our specific application without unnecessary overhead.

Let's bake our image with core-image-full-cmdline :

```
Fedora:~/<Dir>/YOCTOS/Builds/Build_pi64_pfe          $          bitbake
core-image-full-cmdline
```

This command will take approximately 2 hours on the first attempt (depending on the machine). Once completed, We can find the generated image at:

Figure 5.1.8 : image path

Once the image was generated, we prepared the SD card by formatting it and identifying its device name in the appropriate system directory. Proper formatting ensures compatibility with the bootloader and the Yocto-generated image, while identifying the correct device name prevents accidental overwriting of other storage devices during the flashing process. (*See Appendix A.5.2*)

We can flash the image onto the SD card using bmptools with the following command:

```
Fedora:~/<Image Directory> $ sudo bmaptool copy <IMAGE.wic.bz2>
/dev/<MEDIA>
```

By executing this command, we achieve the following:

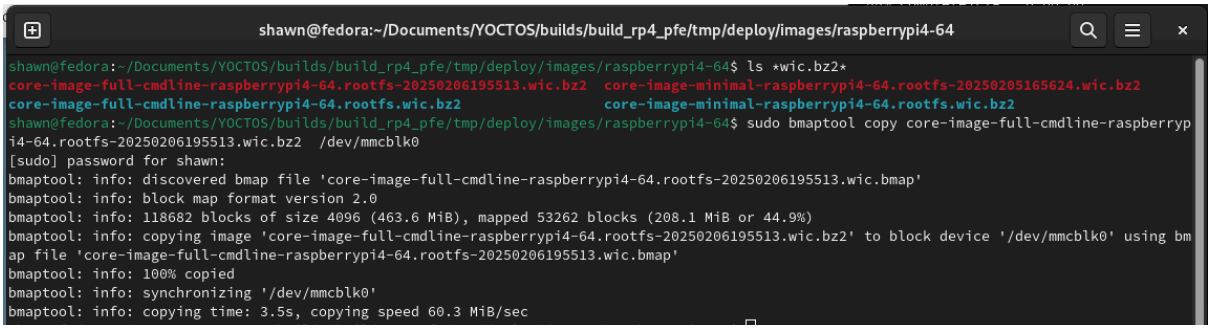


Figure 5.1.9 : Image Location + Flash

Once we have completed this step, we need to test the image on our Raspberry Pi.

e. Verification & Configuration

Since we configured UART in the local.conf file, we will use UART to debug our image. To achieve this, we will utilize a USB-TTL UART module (connected to GPIO pins 14 and 15 on the Raspberry Pi 4 (***Figure 5.1.10***).

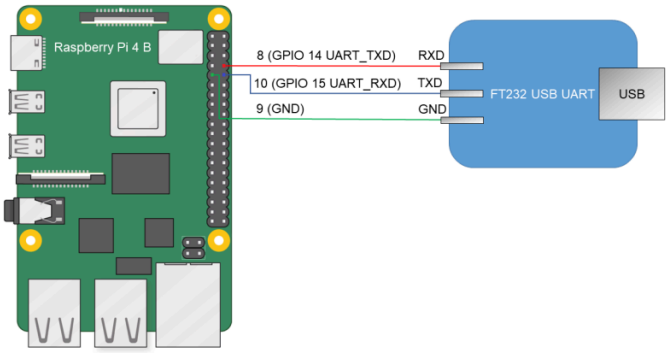


Figure 5.1.10 : UART Raspberry Pi 4

Next, configure PuTTY(a free and open-source terminal emulator) for serial communication on this module, setting the baud rate to 115200 (***See Appendix A.5.3***). Once we insert the flashed SD card into the Raspberry Pi and power it on. After a few seconds, it will prompt us to enter the login credentials. By default, the username is `root` and no password is required. After logging in, we can use basic commands to view hardware information (***See Appendix A.5.4***).

Finally, we configured the Wi-Fi by modifying the `/etc/wpa_supplicant.conf` file with our Wi-Fi credentials (*See Appendix A.5.5*). To connect to the Wi-Fi, we simply reboot the system.

VI. Data Monitoring

The data transmitted from the ESP32 to the Raspberry Pi 4 via UART communication is intended for real-time visualization on Blynk through Node-RED. Consequently, the subsequent step involved configuring the data monitoring environment, ensuring seamless data acquisition, efficient processing, and integration between the hardware components and the cloud-based visualization platform.

1. Node-RED application development

Node-RED is an open-source visual programming tool designed to facilitate the integration of various technologies. It provides a flow-based development environment, allowing users to create automated workflows by linking different functional nodes. These nodes can perform tasks such as data collection from sensors, information processing, communication with cloud services, and system notifications.

In the context of IoT (Internet of Things), Node-RED is commonly used to manage data flow between devices, services, and cloud platforms. Its graphical interface enables users to develop and deploy automation solutions without extensive programming knowledge. Additionally, its library of prebuilt nodes allows for integration with protocols and services such as MQTT, HTTP, WebSockets, databases, and cloud providers, making it a versatile tool for a wide range of applications.

a. Installation

To install Node-RED on our Linux system, there are several methods available, such as using a Docker image or installing required packages individually. However, we found a convenient bash command on GitHub that simplifies the installation process of Node-RED.

```
$ bash <(curl -sL
https://raw.githubusercontent.com/node-red/linux-installers/master/deb/u
pdate-nodejs-and-nodered)
```

After completing the installation, we set Node-RED to start automatically every time we power on the Raspberry Pi by executing the following command.

```
$ sudo systemctl enable nodered.service
```

After rebooting, the Node-RED interface can be accessed on the local network via the Raspberry Pi's IP address at port 1880.

```
http://192.168.0.39:1880/
```

This will provide a web page similar to the one below :

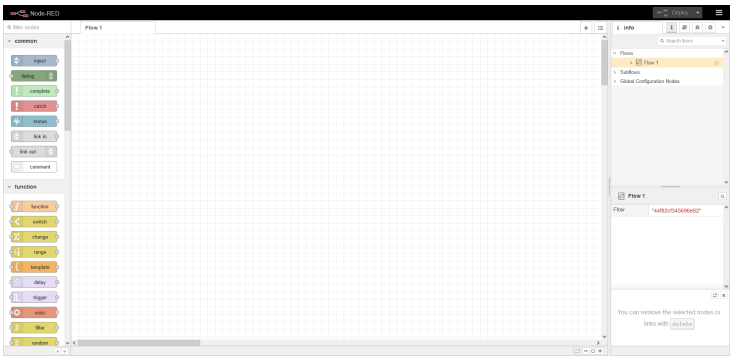
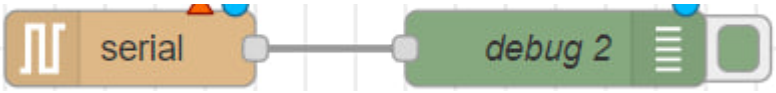


Figure 5.2.1 : Node-Red Interface

b. Serial Communication

For this project, we connect an ESP32 S3 to a Raspberry Pi using a wired connection to gather data. Node-RED simplifies our task; with just two blocks, we can read the serial input



messages from the ESP32 to the Raspberry Pi.

Figure 5.2.2 : Basic Serial Read

However, we need to configure the Serial blocks by setting parameters like baud rate or port.

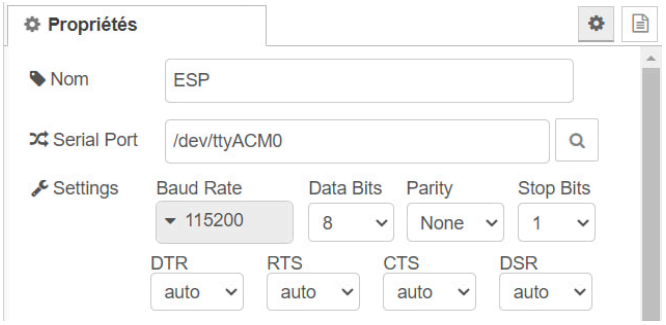


Figure 5.2.3 : Serial Parameters

As we send data from the ESP32 in the format `CAPTEUR_X: int data`, we can observe the following raw data on the debug console:

Figure 5.2.4 : Serial Raw Data

To obtain only the integer values from each sensor, we need to create a function block utilizing JavaScript for this purpose. (***Github*** for Javascript code).

Figure 5.2.5 : Function Block

As we have several sensors, we need to extract the value of each one contained in the message sent from the LoRa transmitter. To achieve this, we simply need to add a function block for each sensor. For example, here we have three sensors:

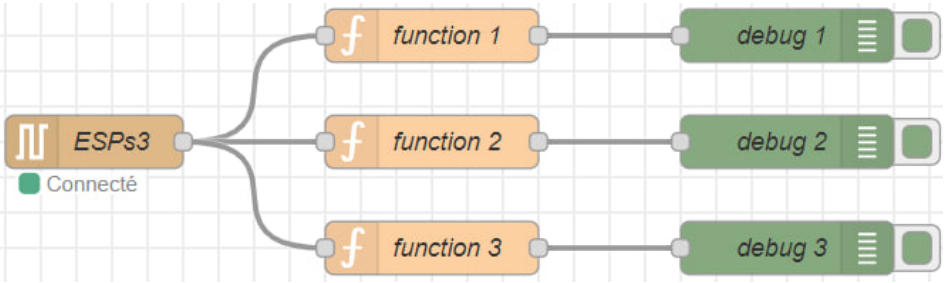


Figure 5.2.6 : Function Block for multiple sensor

On the debug console, we can see that the values are integers for each entry.

Figure 5.2.6 : Multiple sensor debug Console

2. Sending Data to Blynk

Blynk is a comprehensive low-code Internet of Things (IoT) software platform designed for both businesses and developers. It simplifies the process of building, deploying, and managing IoT applications. It's important to mention a few words about the Blynk server. This server allows us to view our data from anywhere in the world. Once we have our data on a Wi-Fi enabled device, we use specific libraries to send this data to the Blynk server. In our case, we use Node-RED-Blynk libraries to send those data.

There are several ways to establish a communication between Node-RED and Blynk. For our application, we decided to use the WebSocket communication which is a real-time and bidirectional communication. It enables a permanent communication between Node-RED and Blynk thus reducing system latency.

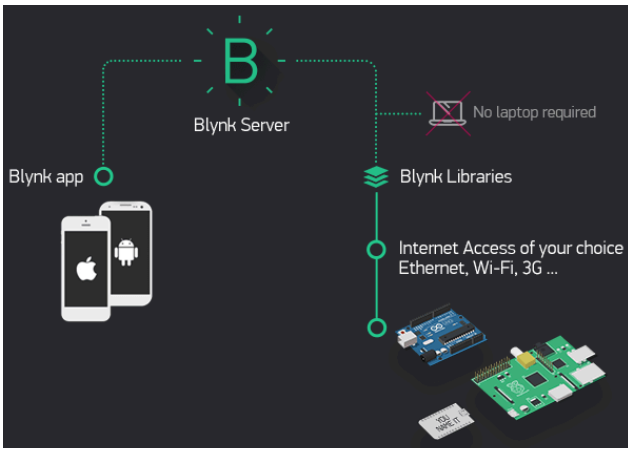


Figure 4.3.1 : Blynk Server

c. Blynk Connection

We need to send each of these integer values to the Blynk platform so we can visualize them on the dashboard from anywhere. To do this, we have to install the Blynk-iot package on our Node-RED application and then use the write block.

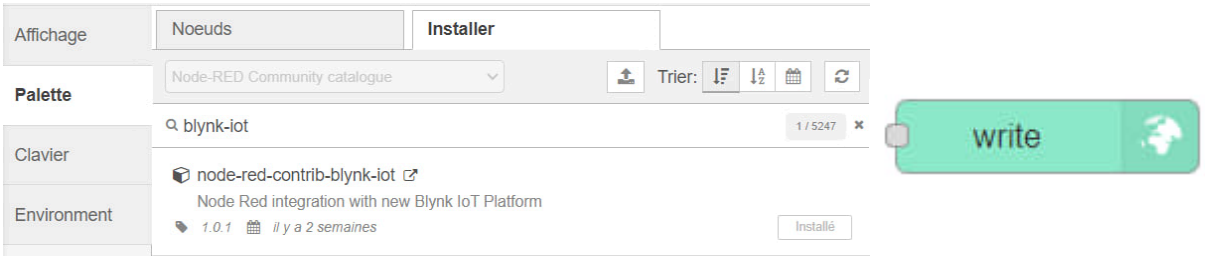


Figure 5.2.7 : blynk-iot and write block

Combining the Blynk-iot block with our sensor data will look like this:

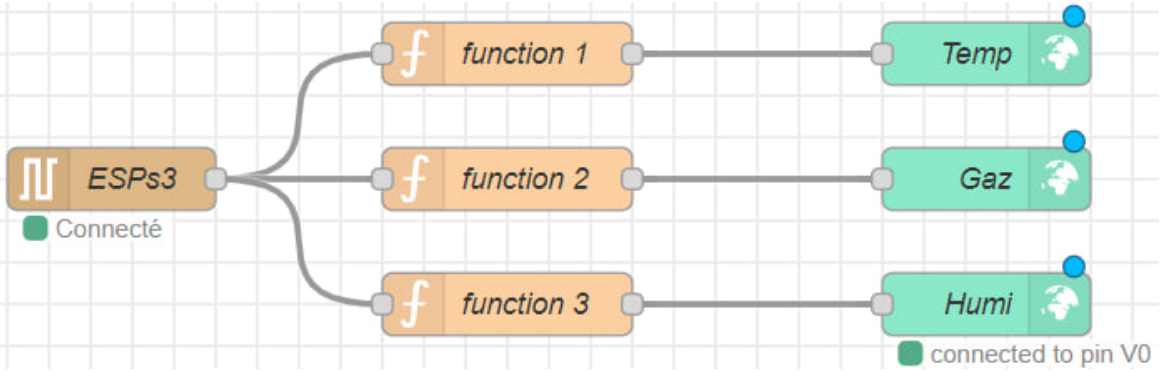


Figure 5.2.8 : Example connection node Node-Red - Blynk

To modify the write block, a double-click is required to select a virtual pin. Next, the necessary authentication credentials, including the key, template ID, and Blynk server URL, must be provided. These credentials are generated by the Blynk platform upon creating a template for a specific project.

Data transmission occurs through virtual pins, which can be mapped to various graphical elements within the Blynk interface, such as charts, gauges, sliders, or buttons, serving as datastreams. This configuration enables efficient visualization and interaction with real-time data, enhancing the functionality and flexibility of IoT applications. The final project design is as follows :

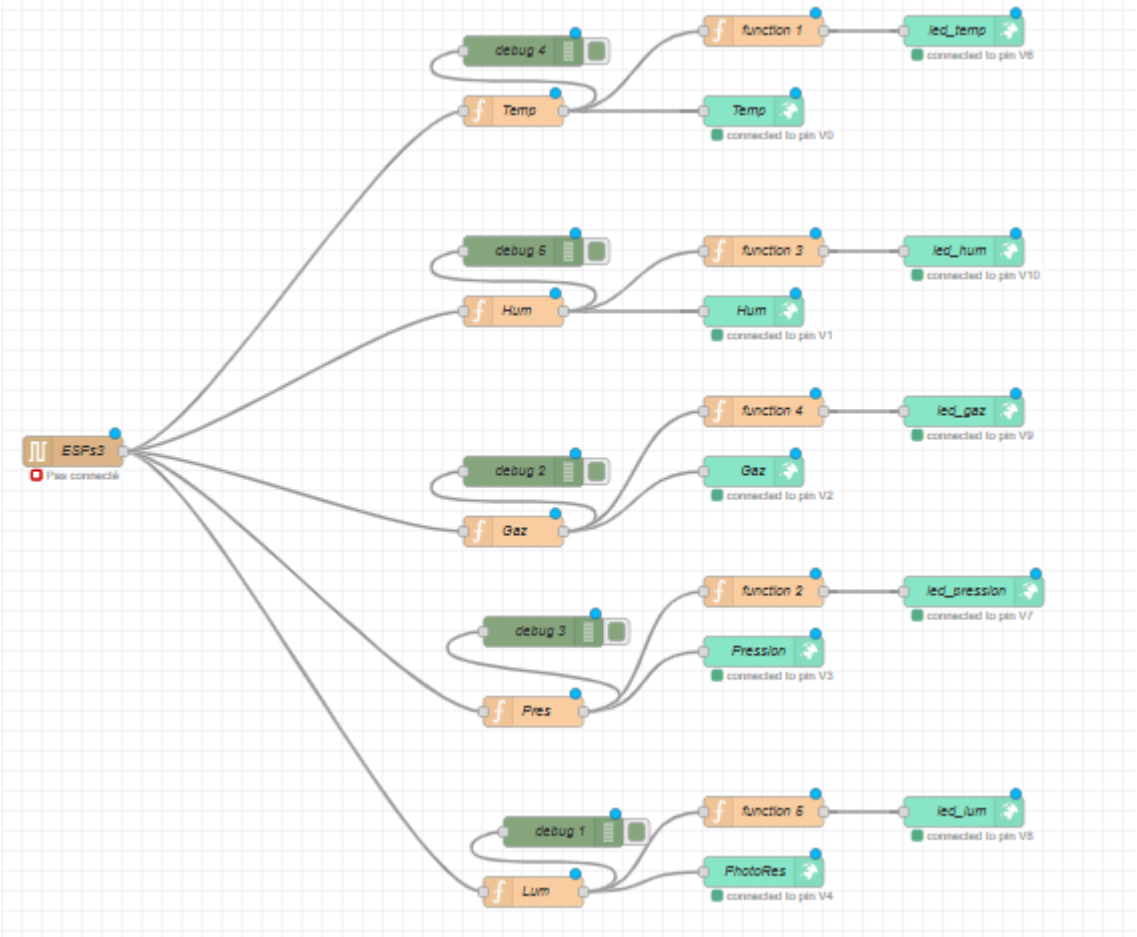


Figure 5.2.8 : Final Application Type Node-Red & Blynk

PS: the debug node is not obligatory but enables to display in real time what we receive before sending to Blynk

2. Integration with Blynk Cloud

In the next section, we will explore how to configure the Blynk platform and obtain the necessary authentication information.

a. Template Creation

To begin a project, we need to create a Blynk account. Once we have created or logged into our account, we can access the "Create New Template" option in the Developer Zone.

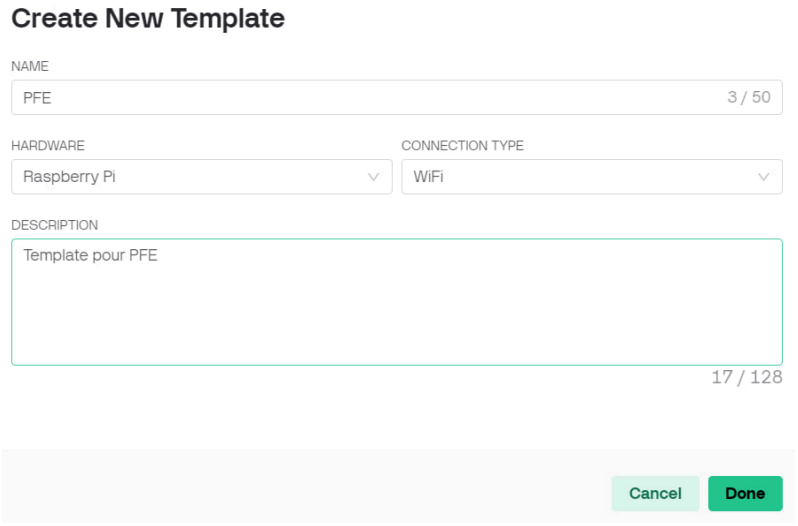


Figure 5.3.1 : Template Creation

In the created template, we can see a menu called "Datastreams." Within this menu, we need to create virtual pins for our temperature data for example.

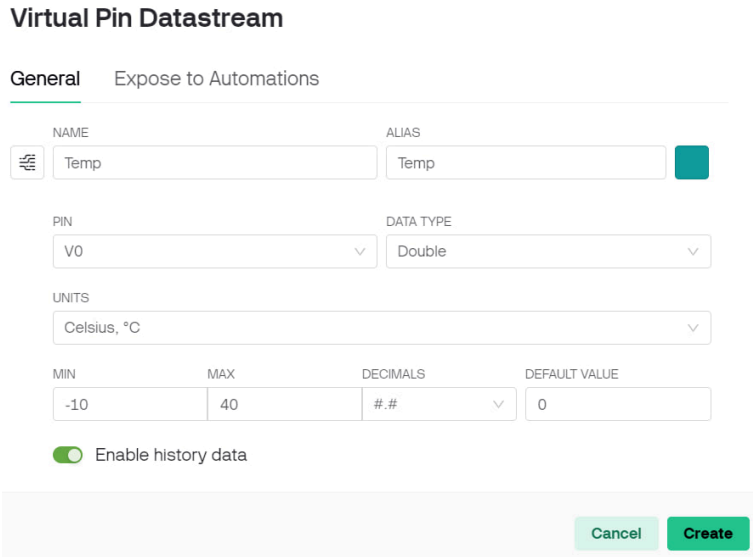


Figure 5.3.2 : Virtual Pin Creation

We need to create virtual pins for each sensor we have, and then configure these datastreams to graphical dashboard objects.

b. Dashboard Configuration

On the web dashboard menu, we can configure a graphical data visualizer for each of the virtual pins. First, we need to drag and drop the desired visualizers. For example, I choose one chart and two gauges.

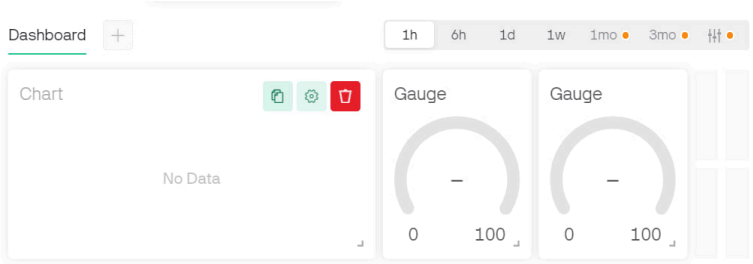


Figure 5.3.3 : Graphical objects

Next, we should click on the settings icon of each graphical object to assign them a data stream (virtual pin). (***See Appendix A.5.8*** for the data stream assignments for the chart view).

After assigning a virtual pin (data stream) that we created earlier, the dashboard should appear as follows:

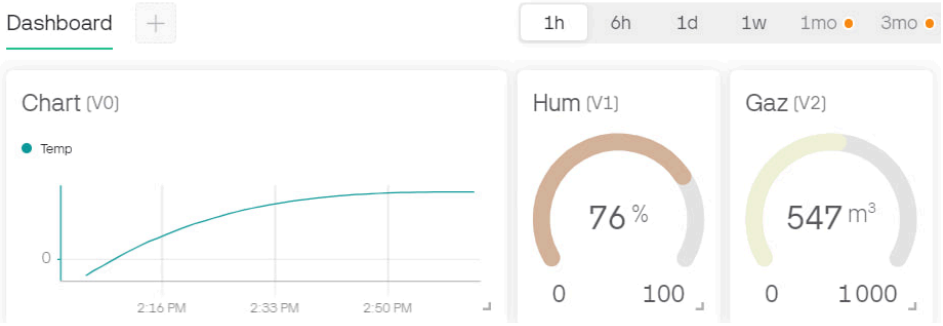


Figure 5.3.4 : Graphical objects after Virtual Pin assigning

After completing this step, we need to create a device on this template to control the virtual pins (data stream) via our Node-RED interface. To do this, navigate to the device menu and create a new device from a template. Select the template created earlier and click 'OK.' The Blynk platform will then provide the authentication credentials as shown below:

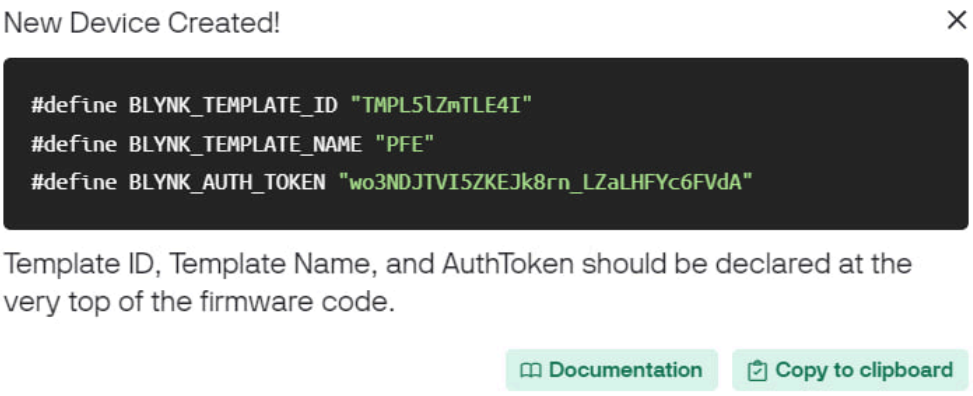


Figure 5.3.5 : authentication credentials keys

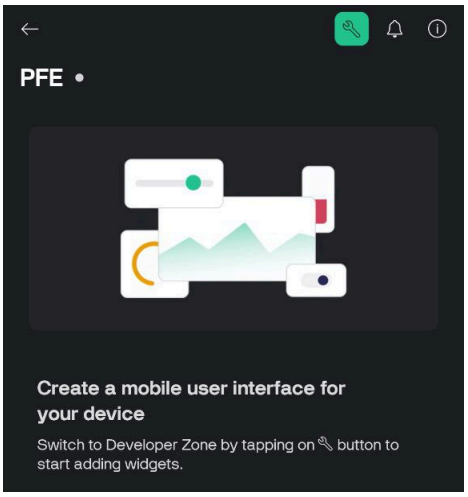
We simply need to enter those credentials and select the virtual pins on the Node-RED application. After that, the monitoring platform will function perfectly.

c. Mobile Configuration

One of the reasons we chose the Blynk platform is its availability as a mobile application, allowing us to monitor our system via our mobile phones. This convenience makes the task much easier. Since we've completed most of the setup on the desktop, we simply need to create a mobile dashboard and assign the virtual pins.

Once we download the Blynk app on our phone, we need to log in with the same account. After logging in, we will see the name of the template we created earlier. When you click on that template, you will see a blank page as shown below:

Figure 5.3.6 : PFE Template Blank page



We need to create the mobile dashboard by clicking on the green icon at the top right. This will open the edit page where we can add graphical objects by clicking the green **Add** button on the bottom menu.

Figure 5.3.6 : ADD button (Green)

We can choose as many graphical objects as we want and assign a data stream (virtual pin) to them by simply clicking on each object. Finally, we just need to go back to see the full dashboard. If we update the values on those virtual pins from Node-RED, the values on both the web dashboard and mobile dashboard will change accordingly.

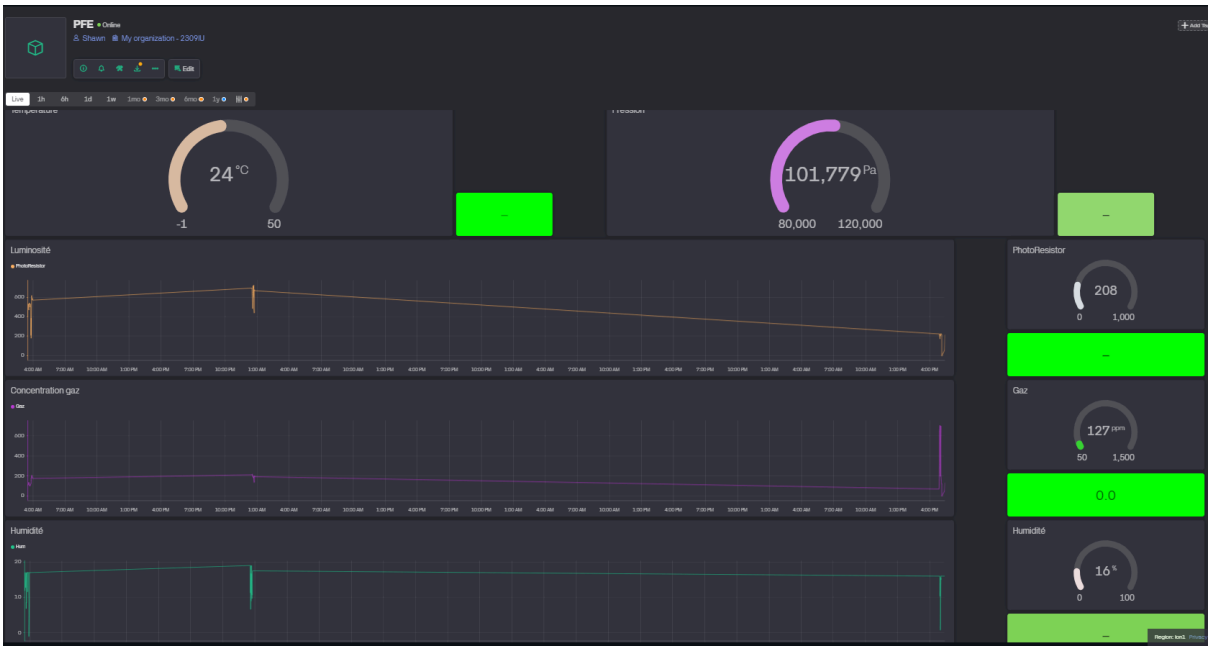


Figure 5.3.6 : Final user interface

VII. PCB for Actuator Control

The need for a deeper understanding and optimization of our system led to the development of a custom electronic board. Initially, this board was designed to replace the STM32 Discovery Board used for data acquisition. However, due to certain technical challenges—which are detailed later—we repurposed the electronic board to handle action generation triggered by the Blynk user interface.

Below is a comprehensive overview of the design and implementation of this electronic board.

3. PCB Design

A custom PCB is crucial for a monitoring system because it allows for the integration of various sensors and components into a compact and reliable design. This ensures efficient data collection, processing, and communication while meeting specific environmental and operational requirements. Custom PCBs can also reduce production costs and offer design flexibility, making them an essential element in developing an effective monitoring system.

PS: This PCB was designed not only for the end of year project but also for future use in personal projects, so some of the components/parts may not be used here.

a. Characteristics

After extensive research on PCB layers, including thorough exploration of Altium tutorials, We opted for a 4-layer design for this PCB. We selected the following configuration:

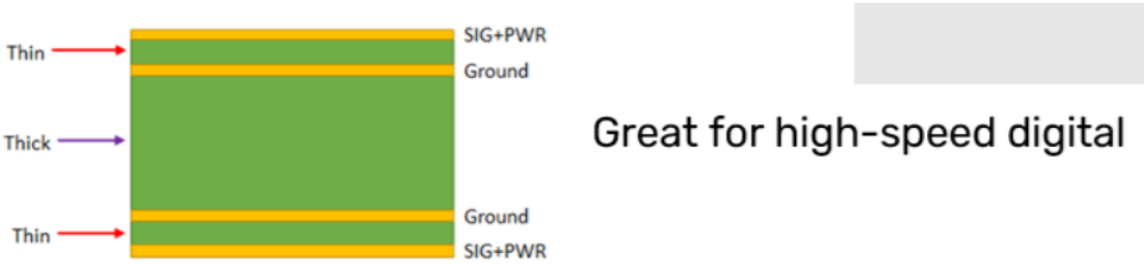


Figure 5.4.1 : stackup configuration

This stackup provides ground planes adjacent to both signal layers, which helps in maintaining signal integrity by offering a consistent return path for the signals. This configuration also helps in reducing

electromagnetic interference (EMI), as the ground planes act as a shield between the signal layers. Additionally, having separate power and ground layers allows for better power distribution and reduces noise, ensuring that sensitive components receive clean power. Overall, this stackup is ideal for high-speed PCBs and mixed-signal applications where signal integrity and EMI control are critical.

b. Power System

Selecting the right power system is crucial for an environmental monitoring PCB. To accommodate different needs, we provided three options for powering the PCB:

- An XT-60 connector, widely used with LiPo batteries.
- An in-pad connector, allowing power through two wires.
- A Mini USB connector.

The user can choose the desired option using jumpers. We then use an AMS1117 3.3V regulator to convert any input voltage(1.5-12V) to 3.3V, which subsequently powers the entire system.

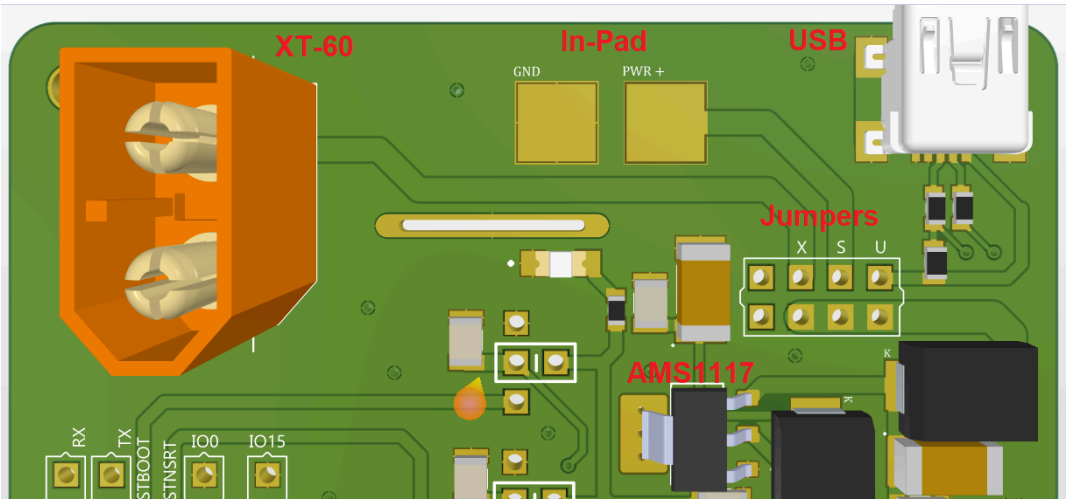


Figure 5.4.2 : Power

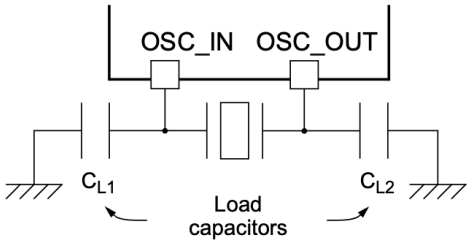
c. STM32

For the central microprocessor of the board, we had two options: the ESP32 and the STM32. While the ESP32 is often pre-configured, requiring only power configuration, we aimed to design a board from scratch. This meant handling not just the microcontroller's power and external clock, but also managing boot configurations and integrating various sensors and peripherals. By choosing the

STM32, we embraced the challenge of creating a comprehensive, custom-designed board that addresses all these elements.

We chose the STM32F070CBT6 for its affordability and impressive characteristics. This microprocessor offers up to 48 MHz of clock frequency using PLL and includes a 12-bit ADC with 12 channels. However, due to most pins being allocated for other purposes, we were able to use only three of these channels.

For the external clock source, we used a crystal ceramic resonator in the traditional configuration with



two capacitors. These capacitors ensure stable oscillation.

Figure 5.4.3 : External Clock

This MCU offers three boot modes. We will use the boot-from-flash mode, which is defined by the BOOT0 and NRST pins. The STM32 microcontrollers have three boot modes: Boot from Main Flash Memory, Boot from System Memory, and Boot from Embedded SRAM. For our application, we will be using the Boot from Main Flash Memory mode. This mode is used for normal operation, where the MCU runs the application code stored in the main flash memory. It is configured by setting the BOOT0 pin to 0 (LOW), ensuring that the microcontroller boots from the main flash memory. The NRST pin, which serves as the reset pin, plays a crucial role in this process. When pulled low, it resets the microcontroller and triggers the boot process. By correctly setting these pins, we ensure that the MCU boots from the main flash memory, running our application code as intended.

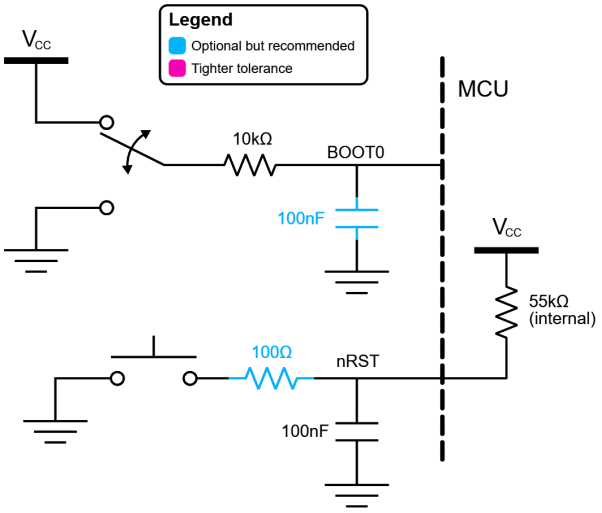


Figure 5.4.4 : BOOT Mode Choose

On the PCB, we decided to place jumpers on these pins, allowing the user to choose between connecting them to VCC, GND, or even controlling them via the ESP8266.

To program the STM32, we provided the option to use the SWD (Serial Wire Debug) interface. This allows programming the microcontroller via an external ST-Link programmer.

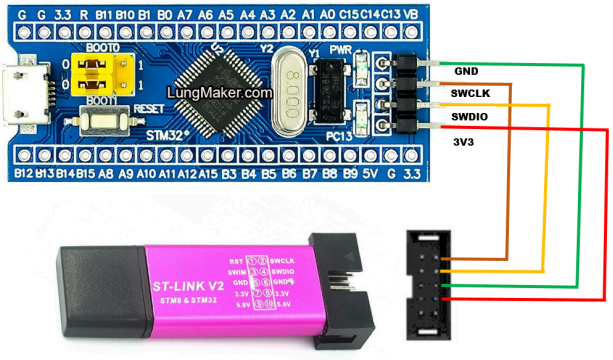


Figure 5.4.5 : SWD with ST-LINK V2 on BluePill

We also provided the option to program the STM32 via the UART interface using jumpers. This enables the user to employ a USB-TTL UART adapter or connect to the ESP8266, allowing the STM32 to be programmed wirelessly using the ESP8266's Wi-Fi capabilities. This flexibility enhances the overall user experience by offering multiple programming methods.

d. ESP8266

As you can see in the previous part, the ESP8266 supports the main microprocessor, STM32, by enabling wireless programming via Wi-Fi and selecting the STM32's boot mode. Additionally, the ESP8266 can serve various other purposes. We provided several free pins to encourage user creativity and allow for custom implementations. This flexibility ensures that users can leverage the ESP8266's capabilities in multiple ways, enhancing the overall functionality of the board.

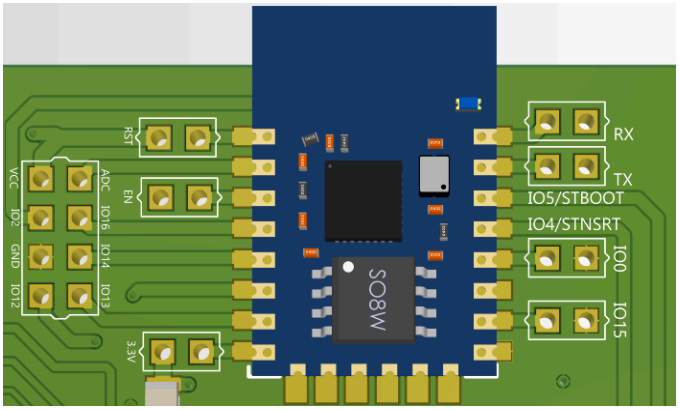


Figure 5.4.6 : ESP8266

e. NRF24L01 & MPU6050

We decided to integrate two sensors/devices directly on this PCB. One of them is the MPU6050, a well-known accelerometer and gyroscope. We followed the schematic example provided in the IC's datasheet to ensure proper integration into our PCB design. This ensures the MPU6050 functions effectively within our system.

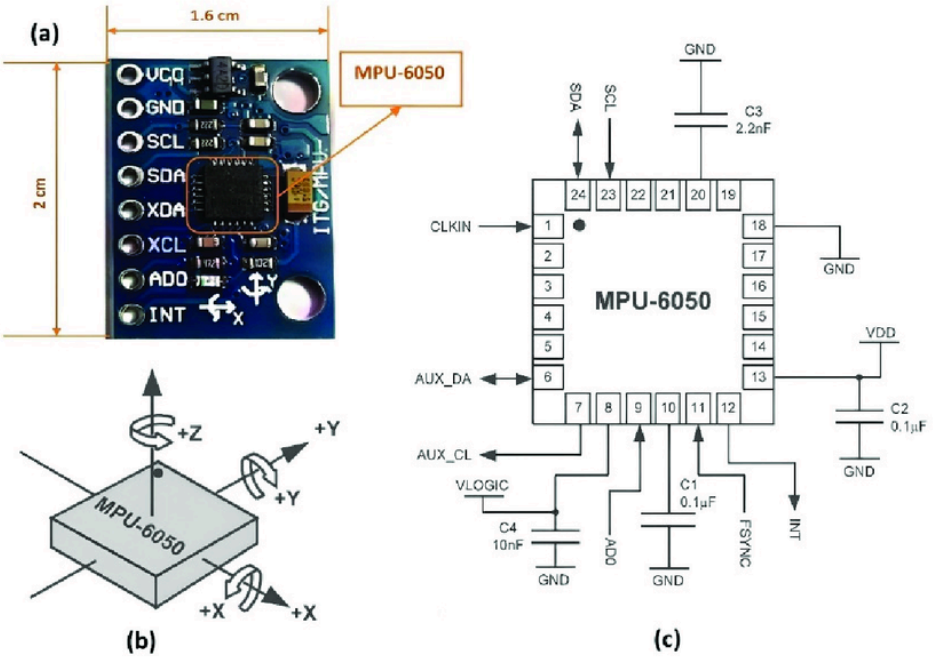


Figure 5.4.7 : MPU 6050

The second device is the NRF24L01, a popular wireless transceiver module used for short-range communication, operating in the 2.4 GHz ISM band. The module supports bi-directional communication. One crucial aspect to ensure when using an antenna with this device is impedance matching, as most antennas have a 50-ohm impedance. Nordic Semiconductor provides a datasheet for the IC, which includes a schematic to follow.

To gain some knowledge about antennas, we decided to experiment with a 50-ohm PCB antenna. Texas Instruments provided a design for a small-size 2.4 GHz PCB antenna, which we replicated on our PCB. However, in our case, the material and the number of layers differed from the original TI design. Therefore, we conducted tests to ensure the bandwidth, gain, and other characteristics of our antenna and compared these results with those of the TI antenna.

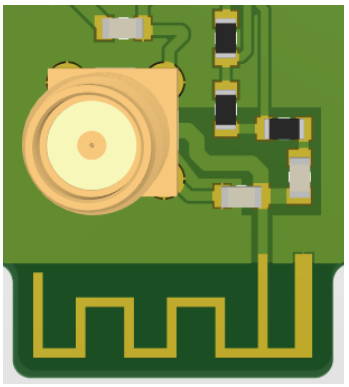
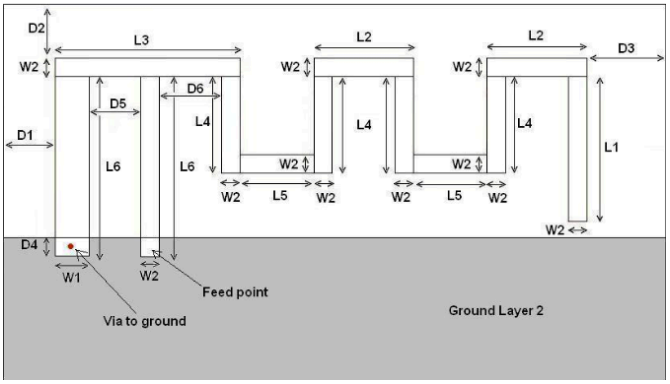


Figure 5.4.8 : Our PCB Antenna

However, we also included an SMA connector as a backup. In case the PCB antenna doesn't perform as expected, this connector allows us to attach a 50-ohm external antenna, ensuring reliable communication.



L1	3.94 mm
L2	2.70 mm
L3	5.00 mm
L4	2.64 mm
L5	2.00 mm
L6	4.90 mm
W1	0.90 mm
W2	0.50 mm
D1	0.50 mm
D2	0.30 mm
D3	0.30 mm
D4	0.50 mm
D5	1.40mm
D6	1.70 mm

Figure 5.4.9 : TI’s Antenna

You can find all the PCB details, including the project and schematics, in the GitHub repository.

Having completed these developments, we proceeded to analyze the results of the PCB design and conduct a series of tests to validate its functionality. These tests, detailed in the next chapter, were essential to ensuring the board met the design specifications.

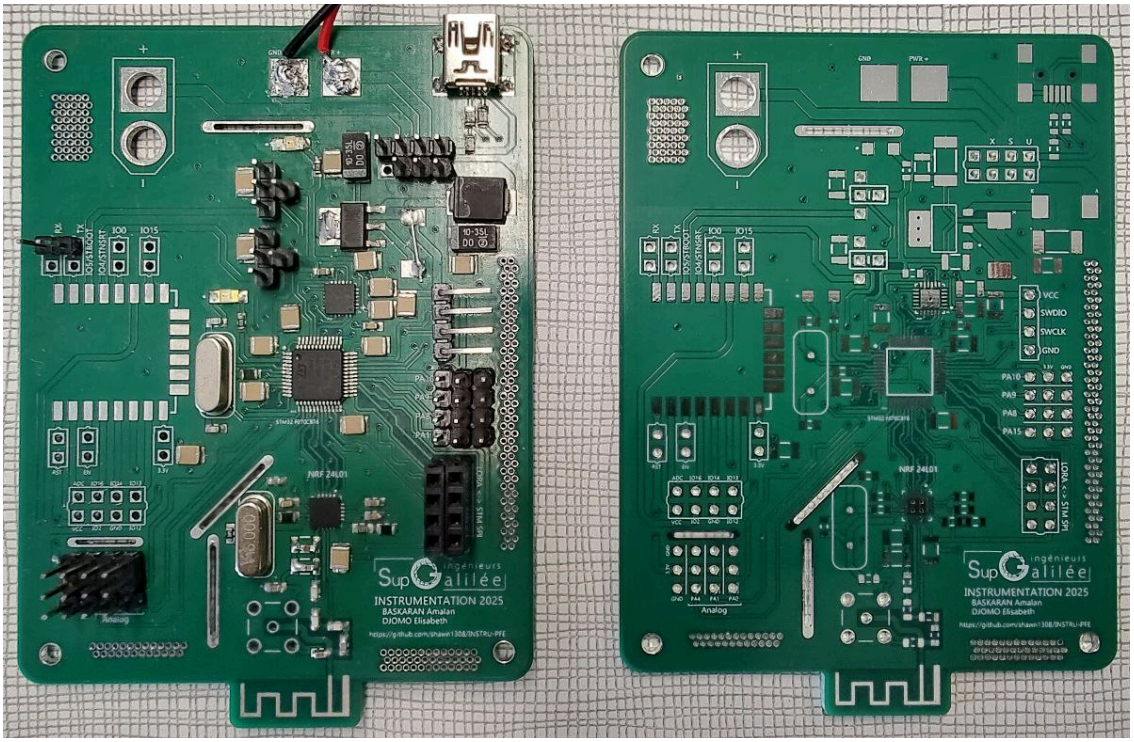


Figure 5.4.10 : PCB After & Before Soldering

f. Blynk to Actuators

The PCB is designed to control actuators via the Blynk interface, which communicates with the PCB through Node-RED and wired serial communication. The PCB connects to actuators, including a buzzer, LEDs, and the UART TTL communication module, allowing the Raspberry Pi to communicate with the PCB via Node-RED, enabling or disabling the actuators.

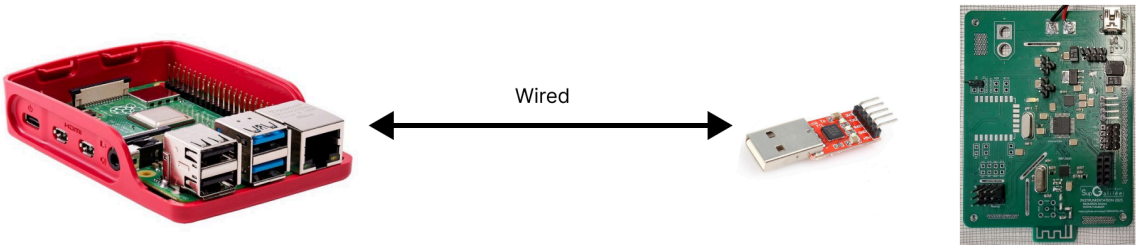


Figure 5.4.11 : RP PI to Custom STM32

The custom STM32 was encased within a 3D-printed box, featuring an LED and a buzzer.

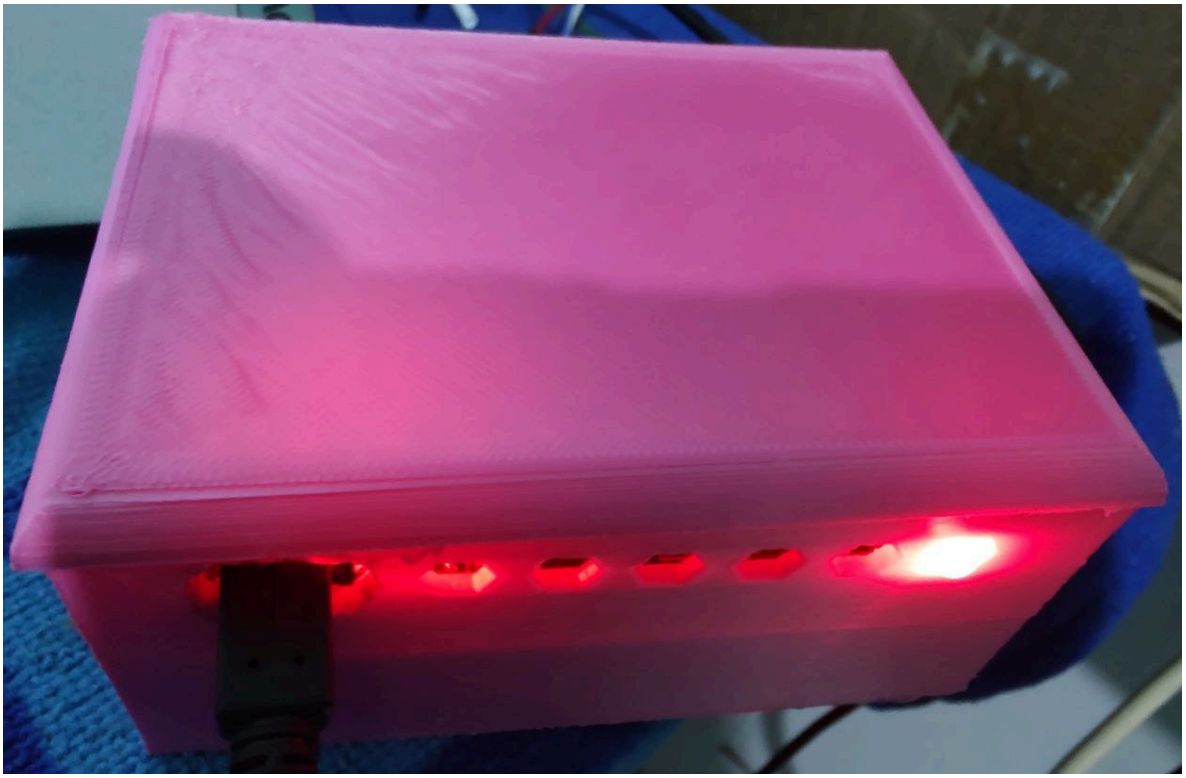


Figure 5.4.12 : Custom STM32 with Led turn via Blynk

VIII. Tests and Results

1. RTOS Tests

a. Experiment 1: Observing Sensor Response Time

Objective: Compare the behavior of an RTOS-based system versus a non-RTOS system by measuring task execution time. In other to verify this, we did the following :

- Add timestamps using millis() to measure the execution time of each sensor.
- Compare the results between the two systems.

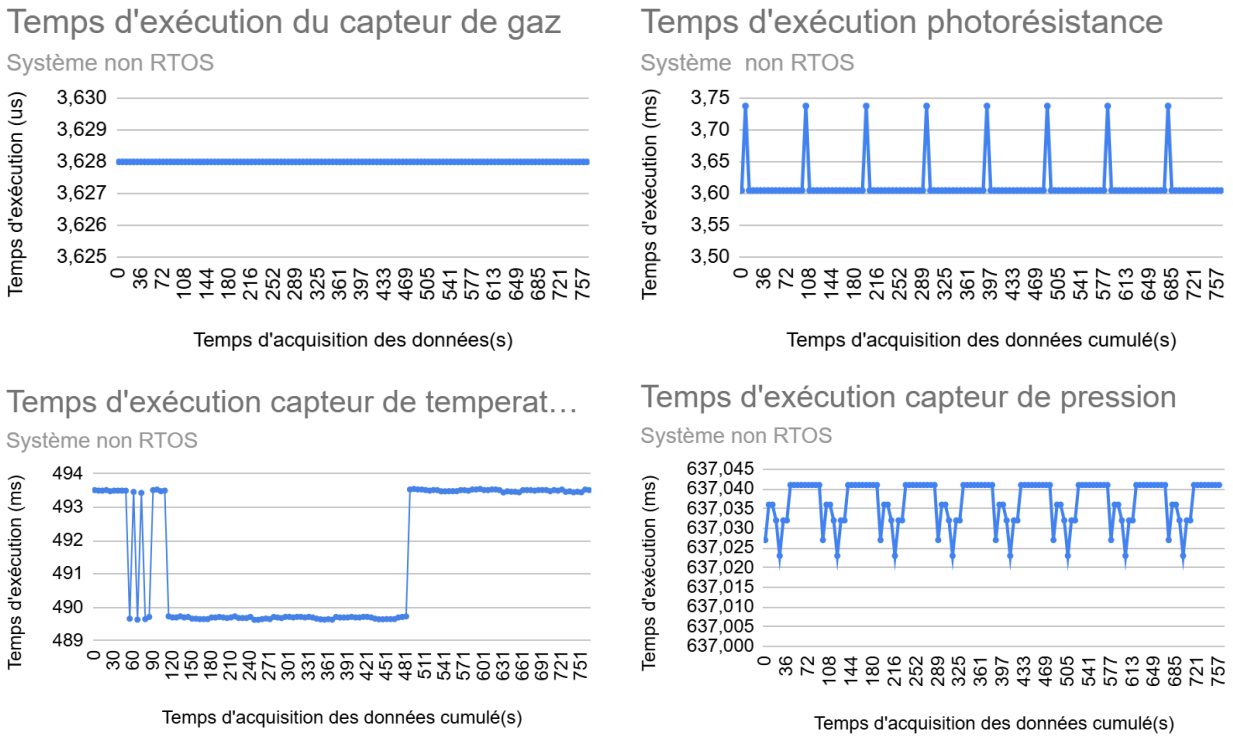


Figure 6.1.1 : Result showing the execution time of different task non RTOS

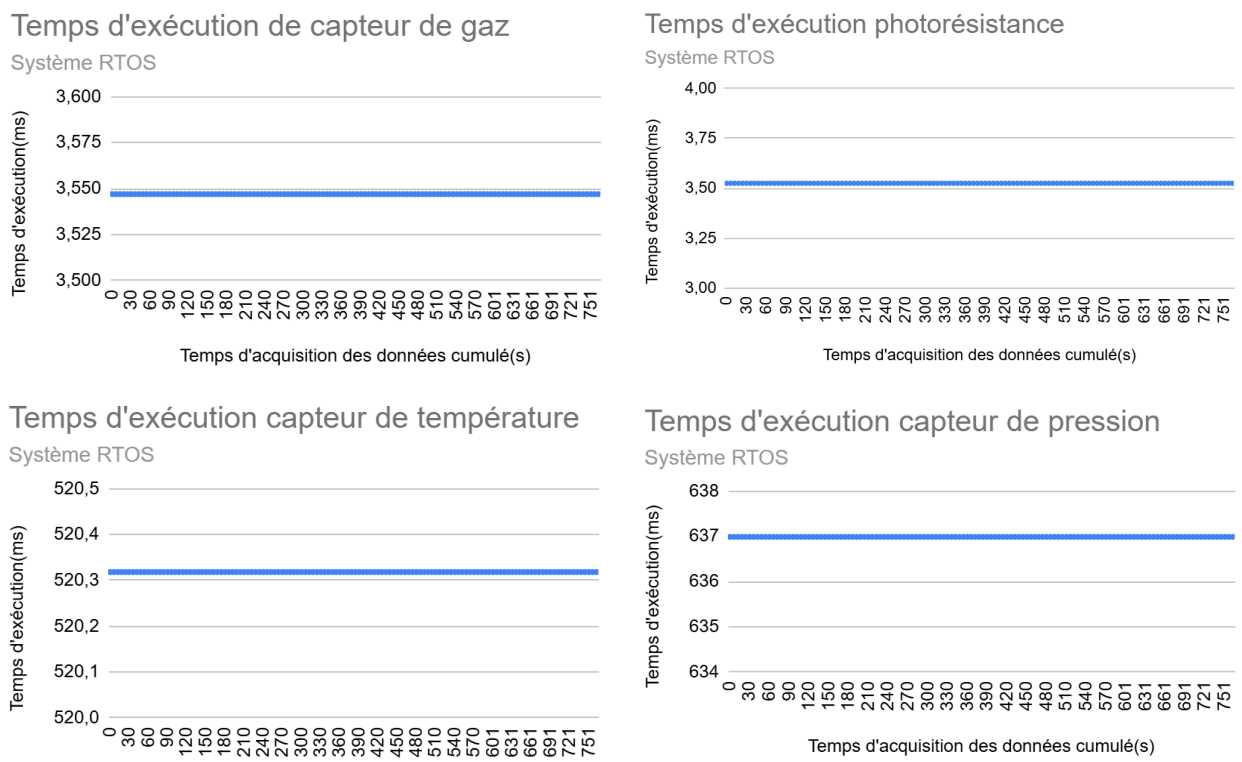


Figure 6.1.2 : Result showing the execution time of different task RTOS

The results show the execution time of different tasks in both non-RTOS and RTOS-based systems. Even though the non-RTOS program was designed to execute tasks without any blocking mechanism, the recorded execution times in the non-RTOS system are not stable and lack predictability.

In contrast, the RTOS system provides more consistent and predictable execution times, ensuring that each task runs at regular intervals, independent of other processes. This confirms that RTOS improves timing determinism, making it better suited for real-time applications requiring precise task scheduling and execution stability.

b. Experiment 2: Testing with a Blocking Task

Objective: Verify that RTOS can handle task blocking effectively. In other to verify this, we did the following :

- Add a delay that corresponds to the time needed for the task to give us relevant information.
- Observe whether the other sensors continue to operate without being affected.

In order to carry out this experiment, each task was given an internal counter that could increment each time the task is being executed.

Execution frequency:

Pressure task : 50ms

Brightness task : 50ms

Gaz task : 1s

Temperature and humidity task : 6s

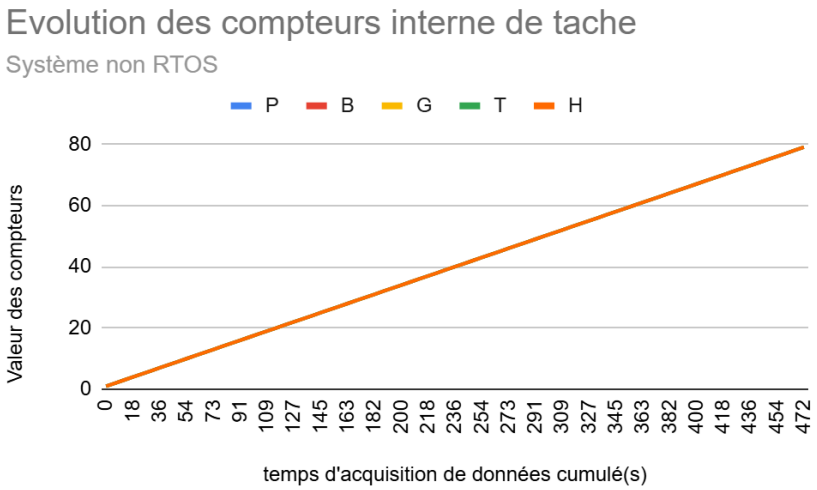


Figure 6.1.3 : Result of the experiment 2 Non-RTOS

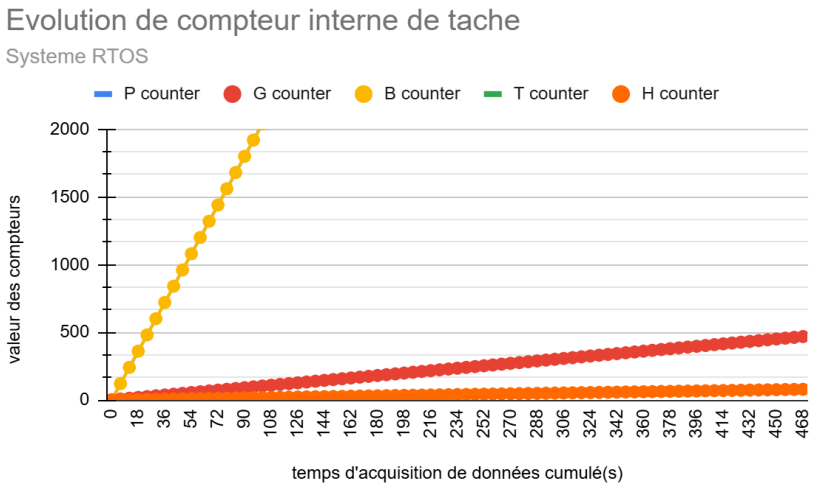


Figure 6.1.4 : Result of the experiment 2 RTOS

In the non- rtos system the graph displays a single overlapping line, indicating that the counters of different tasks increment at the same rate. This suggests that the temperature and humidity sensor task, which has the longest response time, is blocking the execution of other tasks. As a result, all tasks progress at nearly the same rate.

The execution pattern appears linear and slow, reflecting the sequential nature of a non-RTOS system, where tasks are executed one after another rather than concurrently.

On the other hand, in the RTOS system multiple task counters (B, P, G, T, H) are incrementing at different rates, showing that tasks continue executing independently despite the long response time of the temperature and humidity sensor.

Execution time obtained on Node-RED.

Below is an extract of the various data received in Node-RED. We can observe that data from different sensors reaches Node-RED at precise 6-second intervals. It is important to note that the data is transmitted from our LoRa module (transmitter) to the receiver, and Node-RED accesses this data through the serial port of our Raspberry Pi. Our system is still capable of respecting the time constraint imposed using freeRTOS.

In other to know which data is displayed by the debug node, please refer to Blynk connection section

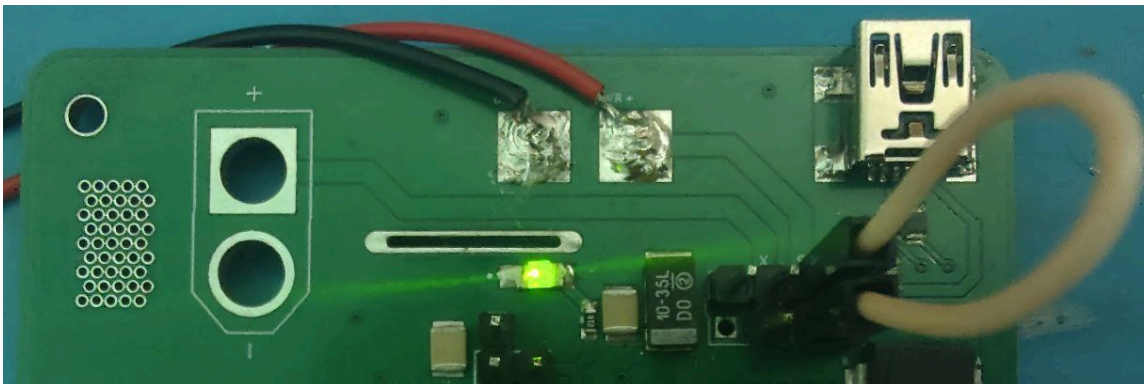
Figure 6.1.4 : Real-time data from Node-RED with a timeline showing a 6-second difference.

2. PCB Functionality Tests

a. Setting Up

After soldering all the essential components required for basic functionality and integrating a power indication LED, we proceeded to power up the device.

To evaluate the PCB design, we conducted a series of basic experiments. The first test performed was the power supply test. This involved checking whether the LED indicator turned on, signaling that the device was receiving power correctly. If the LED failed to illuminate, it would indicate a potential



issue with soldering or circuit design. In our case, the LED turned on successfully, confirming that the power supply was functioning as expected.

Figure 6.3.0 : Powered Board using 3xAA batteries

The second question that arises is how to program the device. As we discussed in part V.4.c, the primary programming system is SWD using an ST-Link. We need to connect five pins from the board to the ST-Link, which are:

VCC	GND	SWD_CLK	SWD_IO	NSRT

Figure 6.3.1 : ST-Link Connection

We can then check our STM32 chip information using the STM32CubeProgrammer software. This software allows us to upload firmware or programs to STM32 chips in bin or hex formats. It also enables us to modify various parameters of the chip, such as erasing the flash memory, putting the chip in protected mode, and more. Here is our chip information :

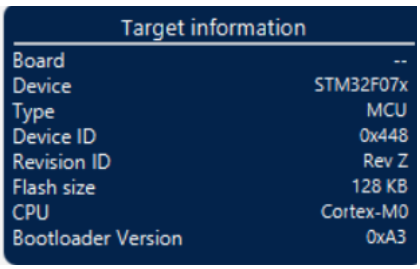


Figure 6.3.2 : STM32CubeProgrammer info

We chose Arduino as our programming IDE. Each compatible chip has its own default pin configuration. For example, our STM32F070 has two SPI interfaces, but in Arduino, we can only use one by default. Furthermore, the default SPI pin configuration may not be compatible with our design. To solve this problem, we need to modify the pin configuration file, which can be found at the following location on Windows machines:

```
C:\Users\<USER>\AppData\Local\Arduino15\packages\STMicroelectronics\hardware\stm32\2.9.0\variants\STM32F0xx\F070CBT\PinNamesVar.h
```

In this file, we can modify all the pin configurations. For example, we can assign different TX and RX pins for serial communication or change the LED_BUILTIN pin to PC13, where our user LED is connected.

b. Test program

The Blink program is a simple test that verifies whether we can successfully communicate with the board through a basic GPIO pin. On our board, Pin PC13, where the LED is connected, is defined as LED_BUILTIN. Using the following Arduino code, the onboard LED blinks to confirm proper functionality:

```
// Setup
pinMode(LED_BUILTIN, OUTPUT); // OUTPUT
// Loop
digitalWrite(LED_BUILTIN, HIGH); // ON
delay(1000); // Wait for 1 second
digitalWrite(LED_BUILTIN, LOW);  // OFF
delay(1000); // Wait for 1 second
```

In our case, the LED blinked correctly, confirming that the board is capable of processing basic GPIO updates and executing simple commands.

IX. Issues Encountered and Solutions

1. LoRa Communication Issues

One of the challenges encountered during the project was the lack of an adequate library to directly interface the LoRa module with the Raspberry Pi. Recent updates in the SPI library of the Raspberry Pi 4 introduced compatibility issues, making it difficult to integrate the module, which relies on SPI communication. To overcome this limitation, we opted to use an ESP32 microcontroller as an intermediary, connecting it to the LoRa module for data transmission before later transferring the collected data to the Raspberry Pi via UART. This approach ensured a stable communication link while maintaining the flexibility needed for real-time data acquisition.

Additionally, regulatory constraints on the duty cycle of the SX1278 LoRa module, combined with the varying response times of the environmental sensors used in this project, necessitated careful selection of the data transfer intervals. The transmission frequency had to be optimized to minimize data loss while ensuring compliance with duty cycle limitations. Furthermore, the LoRa module's parameters were fine-tuned to balance transmission speed and range, ensuring that the system met both performance and regulatory requirements while maintaining reliable long-range communication.

2. PCB Problems

We encountered multiple problems with the PCB. For example, the first version of the PCB didn't work because we made a design mistake by forgetting to connect the current return path. This forced us to correct the error and reorder the PCB. Additionally, the IMU and NRF parts didn't work as expected.

X. Future Perspectives

This project primarily focused on the development of a soft RTOS, which provided an efficient way to manage tasks and optimize system responsiveness. However, as a future improvement, it would be valuable to explore the development of a hard RTOS to achieve even stricter real-time constraints, ensuring deterministic task execution with minimal jitter. This would be particularly beneficial for applications requiring ultra-low latency and high reliability.

Another potential enhancement would be the integration of LoRaWAN instead of a traditional LoRa point-to-point communication. LoRaWAN would enable scalability, allowing multiple devices to communicate over a centralized network with reduced power consumption and improved data

management. This would make the system more adaptable for large-scale IoT applications where multiple nodes need to transmit data efficiently over long distances.

XI. Conclusion

Through this project, we gained a deeper understanding of embedded system development by designing and implementing a functional solution for environmental monitoring. Our system achieved a response time of 6 seconds, demonstrating its ability to monitor environmental parameters efficiently. To optimize performance and minimize latency, we explored various approaches, including the use of an RTOS for task management, a custom Linux kernel to reduce system overhead, and WebSocket communication between Node-RED and Blynk to improve responsiveness.

By integrating LoRa for long-range communication, Blynk for real-time visualization and control, and Node-RED for data processing, we moved beyond traditional academic projects to gain hands-on experience with industry-relevant technologies in IoT and automation. This enabled us to develop a fully operational system capable of sensor data acquisition, wireless transmission, processing, and remote monitoring and control.

Beyond the technical aspects, this project helped us enhance our adaptability, problem-solving skills, and understanding of real-world constraints in embedded system design. It reinforced the importance of making trade-offs to balance performance, efficiency, and system constraints. For instance, while a 6-second response time is well-suited for our application, it would be unacceptable in safety-critical systems such as airbags, where near-instantaneous response is required. This highlights that system efficiency is always relative to its intended application, and the best design choices depend on the specific use case and constraints.

XII. Appendices

1. Resource

Presentation link	: Presentation
Github link	: Github

1. Sensor datasheet
    a. MQ-2 Semiconductor Sensor for Combustible Gas
    b. DHT 11 Humidity & Temperature Sensor
    c. BMP180 – Digital Pressure Sensor

2. LoRa
    a. Wiki
    b. Tableau national de répartition des bandes de fréquences

3. YOCTO
    a. Documentation
    b. Presentation
    c. French Yocto Class
4. PCB Design
    a. Layer Stack
    b. JLC PCB Capabilities
    c. 4 Layer PCB by Zachariah Peterson
    d. EMI/EMC by Zachariah Peterson

2. Image

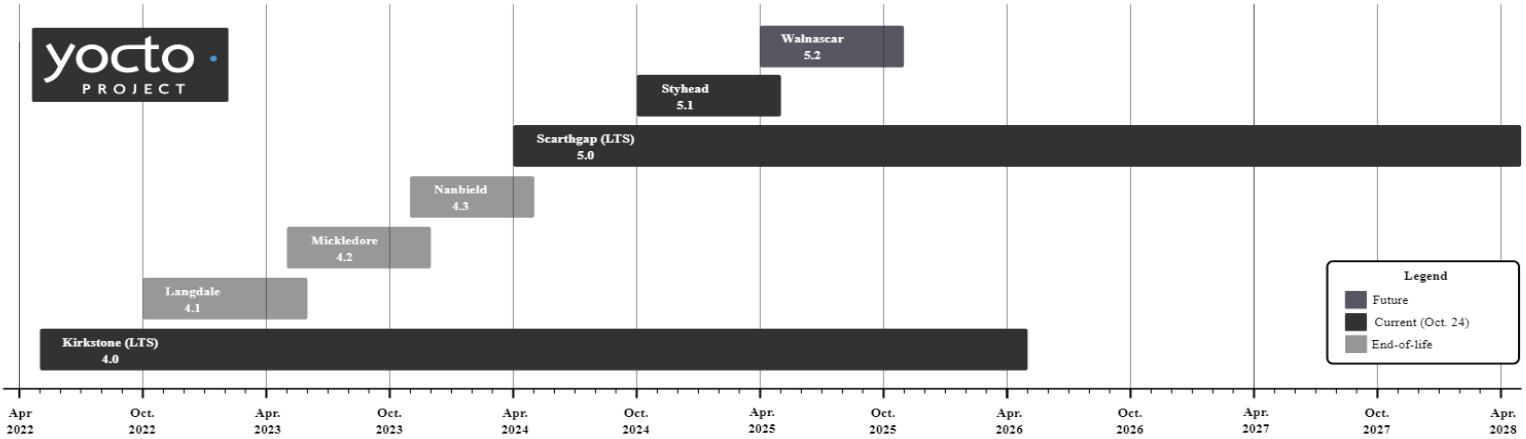


Figure A.5.1 : Linux versions (Page X)

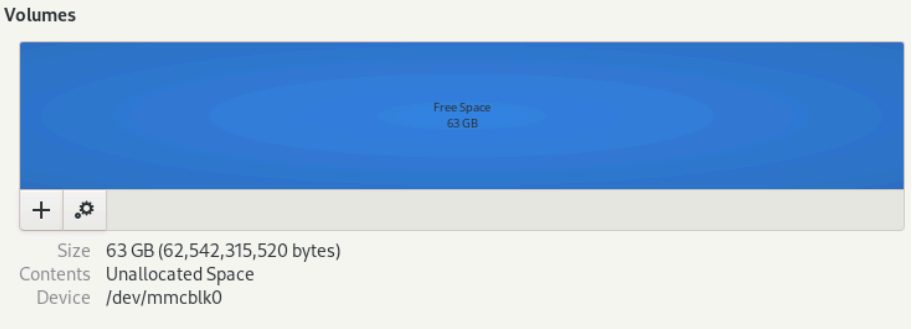


Figure A.5.2 : SD card /dev/mmcblk0 (Page X)

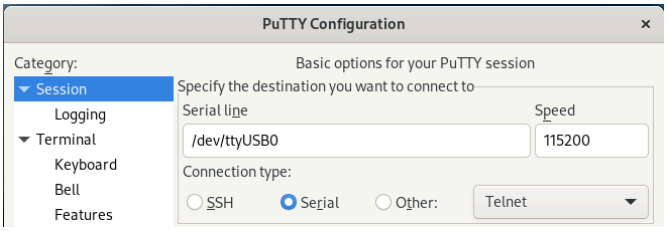


Figure A.5.3 : UART info (Page X)

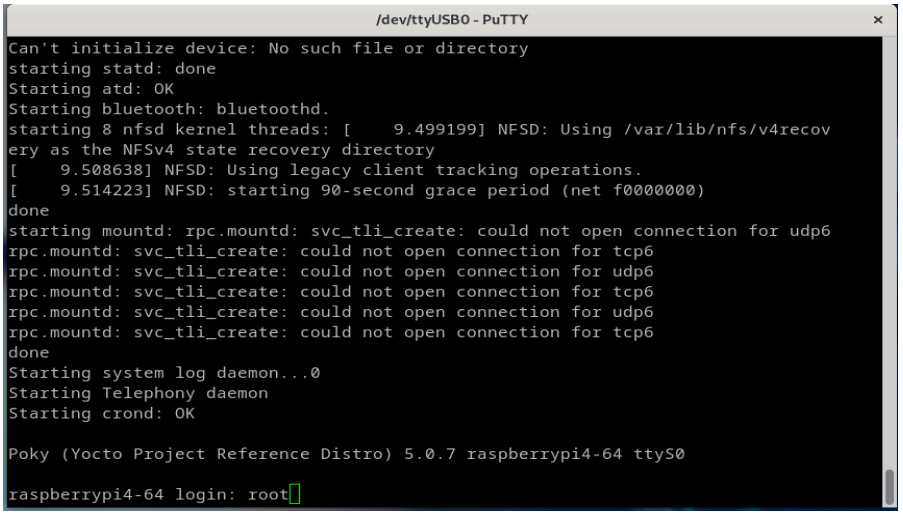


Figure A.5.4 : Login (Page X)

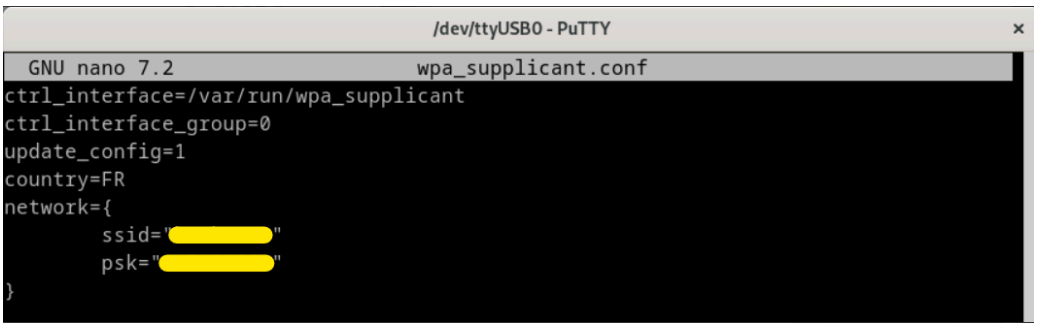


Figure A.5.5 : Wifi (Page X)