LoRaWAN communications for a plant monitoring IoT system using the B-L072Z-LRWAN1 ARM mbed-based platform

Sensor Networks

Estela Mora Barba

MIoT 2025-2026

Contents

List of Figures

List of Tables

List of Listings

Acronyms

ADC Analog-to-Digital Converter

AES Advanced Encryption Standard

CPU Central Processing Unit

EUI Extended Unique Identifier

GPIO General Purpose Input-Output

GPS Global Positioning System

I2C Inter-Integrated Circuit

IoT Internet of Things

LED Light Emitting Diode

LoRaWAN Long Range Wide Area Network

LPWAN Low-Power Wide-Area Network

MCU Microcontroller Unit

NVS Non-Volatile Storage

OTAA Over-The-Air Activation

RAM Random-Access Memory

RGB Red, Green and Blue

RTOS Real-Time Operating System

UART Universal Asynchronous Receiver-Transmitter

USART Universal Synchronous Asynchronous Receiver-Transmitter

1 Overview and Introduction

1.1 Document Overview

This document defines the technical specifications, development requirements, software components, and ResIoT configurations of the IoT-based Plant Monitoring System implemented during the course *Sensor Networks*. The objective of this specification is to establish a clear and comprehensive reference for the design, implementation, and verification of a system that integrates long-range connectivity via Long Range Wide Area Network (LoRaWAN).

1.2 Project Introduction

The main goal of Project 1 is to enable LoRaWAN connectivity on an STM32-based IoT node running the Zephyr Real-Time Operating System (RTOS). The system is designed to monitor the environmental conditions and physiological state of a plant.

1.3 Summary of the Work Done

This section provides a consolidated overview of the work completed, which builds upon the sensor management hardware and software developed in the previous subject. The current work focuses on the integration of the LoRaWAN stack and ResIoT data visualization.

1.3.1 Phase 1: LoRaWAN Stack Integration and Analysis

The initial phase involved the deployment and analysis of the `zephyr-os-example-LoRaWAN` project. Key tasks included:

- Configuration of device identification parameters, including Device Extended Unique Identifier (EUI), Join EUI, and Application Key for Over-The-Air Activation (OTAA) activation.
- Integration of the LoRaWAN stack within the Zephyr RTOS environment.
- Verification of network join procedures, uplink and downlink transmission through a gateway located at the University Campus.

1.3.2 Phase 2: Application Development and Remote Monitoring

Building on the connectivity baseline, the plant monitoring application was integrated into the LoRaWAN communication loop:

- Implementation of data acquisition for sensors and send it to the ResIoT platform.
- Design of a custom LUA payload decoder for the ResIoT network server to process incoming data.
- Creation of a remote dashboard featuring map widgets for real-time tracking and historical data.
- Development of a downlink command handler to remotely change the Red, Green and Blue (RGB) Light Emitting Diode (LED) (OFF, Green or Red) from the ResIoT platform.

1.3.3 Phase 3: Payload Optimization and Efficiency

To maximize data throughput and adhere to the 30-byte message limit, significant optimizations were implemented:

- Transition from string-based messaging to a binary-packed structure using an `__attribute__((packed))` C struct to reduce payload size.
- Conversion of geographical coordinates and sensor values to appropriate data types to minimize byte usage.
- Refinement of the LUA decoder to handle the optimized binary format and update the ResIoT variables accordingly.

2 Project Specifications

2.1 System Infrastructure

The infrastructure relies on a star-of-stars topology consisting of the following elements:

- **LoRaWAN Nodes**: Hardware based on the STM32WL55JC microcontroller[1] equipped with sensors.
- **Gateways**: A LoRaWAN Gateway located on the roof of Building 8 at the university campus. It relays messages to the server address `eu72udp.resiot.io`.
- **Network and Application Server**: The ResIoT.io platform[2] is used for device management, data decoding via LUA scripts, and dashboard visualization.

2.2 Hardware Specifications

The hardware components utilized in this project are detailed in Table 1.

Table 1: Summary of Hardware for the LoRaWAN IoT System

Parameter	**Sensor / Module**	**Interface**
Microcontroller Unit (MCU) Board	STM32WL55JC[1]	LoRaWAN
Ambient Light	HW5P-1 Phototransistor[3]	ADC
Soil Moisture	SEN-13322[4]	ADC
Temperature / Humidity	Si7021[5]	I2C
Leaf Colour	TCS34725[6]	I2C
Acceleration	MMA8451Q[7]	I2C
Global Location	Adafruit Ultimate GPS[8]	UART
Downlink Commands	RGB LED	GPIO

2.3 Software and Connectivity Requirements

A summary of the connectivity parameters and payload configuration is presented in Table 2.

Table 2: Connectivity and Payload Specifications

Feature	**Requirement**
Activation Mode	OTAA
Join EUI	`0x70B3D57ED000FC4D`
Transmission Interval	60 seconds
Max Message Length	30 bytes
Implemented Payload	30 bytes (`packed` structure)
Downlink Control	Remote LED change via “OFF”, “Green”, or “Red”

2.4 Additional Specifications Implemented

The technical measures for resiliency and data handling are listed in Table 3.

Table 3: Additional Specifications Implemented

Requirement	**Description**
Join Resiliency	30 join retries with a delay between attempts.
Atomic Storage	Measurements use Zephyr atomic variables to prevent race conditions between sensor threads and the main LoRaWAN loop.
LUA Decoding	A LUA script on the ResIoT server parses the hexadecimal payload.
GPS Fail-safe	Predefined coordinates are used if the Global Positioning System (GPS) signal is not fixed.

3 Node Software Organization

3.1 Description of the implementation

3.1.1 LoRaWAN Network Fundamentals

LoRaWAN is a Low-Power Wide-Area Network (LPWAN) protocol designed for battery-operated devices. In this project, the **STM32WL55JC** node uses its integrated radio transceiver to communicate with a **Gateway** located at the university campus (Building 8, roof). The data flow follows the LoRaWAN star topology, enabling bi-directional communication through Uplink and Downlink messages:

- **End-Node**: The STM32 node acquires environmental parameters (temperature, humidity, light, moisture, leaf color, and acceleration) and transmits binary-encoded messages every 60 seconds. The implementation leverages LoRaWAN Class A downlink windows to receive control strings.
- **Gateway**: Receives the packets and forwards them to the server address `eu72udp.resiot.io`.
- **Network Server (ResIoT)**: Manages device authentication, security keys, and data routing to the application dashboard. The system allows sending remote commands from the ResIoT platform back to the node.

3.1.2 Project Structure

The software architecture is designed as a modular, multi-threaded application running on the Zephyr RTOS[9]. To ensure efficient resource management and prevent blocking the LoRaWAN communication stack, the system is partitioned into several functional modules and specialized threads.

The interaction between sensors and hardware abstraction logic is organized into specific libraries for each sensor and actuator. The application utilizes three main threads to handle concurrent Atomic Variables for shared measurements:

- **Main Thread**: Acts as the central orchestrator. It manages the LoRaWAN join process (OTAA), coordinates the sampling triggers, and handles the transmission of the binary-packed payload to the ResIoT server every 30 seconds.
- **Sensors Thread**: A dedicated thread that, upon receiving a signal from the Main thread, sequentially samples all sensors (except GPS).
- **GPS Thread**: Same as Sensors Thread but exclusively for the GPS module.

3.1.3 Network Activation (OTAA)

The system implements **Over-the-Air Activation (OTAA)** for joining the network. This process requires three specific identifiers configured in the `main.c` file to perform the cryptographic handshake with the ResIoT server.

- **LORAWAN_DEV_EUI (Device EUI):** A 64-bit globally unique identifier for the end-device. Every device must have a different EUI to be distinguished by the server.
- **LORAWAN_JOIN_EUI (Application EUI):** A 64-bit global application identifier. It identifies the specific application.
- **LORAWAN_APP_KEY (Application Key):** A 128-bit Advanced Encryption Standard (AES)-128 secret key unique to the device. This key is used to sign the "Join Request" and verify the "Join Accept" message. It is a root security key that never travels through the air. Instead, it is used to derive the session keys (**AppSKey** and **NwkSKey**) that encrypt the actual plant data during transmission.

Listing 1: LoRaWAN OTAA Configuration in main.c

```
/* Unique identifiers for the ResIoT network */
#define LORAWAN_DEV_EUI  { 0x7a, 0x39, 0x32, 0x35, 0x59, 0x37, 0x91, 0x94 }
#define LORAWAN_JOIN_EUI { 0x70, 0xB3, 0xD5, 0x7E, 0xD0, 0x00, 0xFC, 0x4D }
#define LORAWAN_APP_KEY  { 0xf3, 0x1c, 0x2e, 0x8b, 0xc6, 0x71, 0x28, 0x1d,
                           0x51, 0x16, 0xf0, 0x8f, 0xf0, 0xb7, 0x92, 0x8f }
```

3.1.4 Payload Encoding

The application uses **Atomic Variables** (4 bytes each) for shared storage between the **sensors_thread**, **gps_thread**, and the **main** loop. This ensures that the data sent via LoRaWAN is never corrupted by a thread context switch during a read operation, maintaining system stability and data integrity.

Listing 2: Atomic Variables Structure

```
struct system_measurement {
    atomic_t brightness; /**< Latest ambient brightness (0-100%). */
    atomic_t moisture;  /**< Latest soil moisture (0-100%). */

    atomic_t accel_x; /**< Latest X-axis acceleration. */
    atomic_t accel_y; /**< Latest Y-axis acceleration. */
    atomic_t accel_z; /**< Latest Z-axis acceleration. */

    atomic_t temp;      /**< Latest temperature (C). */
    atomic_t hum;       /**< Latest relative humidity (%RH). */

    atomic_t red;       /**< Latest red color value (raw). */
    atomic_t green;     /**< Latest green color value (raw). */
    atomic_t blue;      /**< Latest blue color value (raw). */
    atomic_t clear;     /**< Latest clear color channel value (raw). */

    atomic_t gps_lat;   /**< Latest GPS latitude (degrees). */
    atomic_t gps_lon;   /**< Latest GPS longitude (degrees). */
    atomic_t gps_alt;   /**< Latest GPS altitude (meters). */
    atomic_t gps_sats;  /**< Latest number of satellites in view. */
    atomic_t gps_time;  /**< Latest GPS timestamp. */
};
```

To comply with the project's strict **30-byte limit**, Phase 3 focused on converting the data to a **packed structure**.

Listing 3: Optimized 30-byte Binary Structure

```
struct __attribute__((packed)) main_measurement {
    // GPS Data (17 bytes)
    int32_t lat;      // 4 bytes (Scaled by 1e6)
    int32_t lon;      // 4 bytes (Scaled by 1e6)
    int32_t alt;      // 4 bytes (Value * 100 in meters)
    uint8_t time[3];  // 3 bytes (HH, MM, SS)
    uint8_t sats;     // 1 byte (Satellites in view)

    // Temperature and Humidity (4 bytes)
    int16_t temp;     // 2 bytes (Celsius * 100)
    uint16_t hum;     // 2 bytes (Relative humidity * 10)

    // Light and Soil (4 bytes)
    uint16_t light;   // 2 bytes (Percentage * 10)
    uint16_t moisture; // 2 bytes (Percentage * 10)

    // Color (3 bytes)
    uint8_t r_norm;   // 1 byte
    uint8_t g_norm;   // 1 byte
    uint8_t b_norm;   // 1 byte

    // Accelerometer (3 bytes)
    int8_t  x_axis;   // 1 byte
    int8_t  y_axis;   // 1 byte
    int8_t  z_axis;   // 1 byte
};
```

The selection of data types is based on the precision required and the expected range $[min, max]$ for each sensor to optimize the LoRaWAN payload.

Latitude (lat):

- **Range:** $[-90 : 90]$ decimal degrees with 6 decimal places.
- **Scaled Range:** $[-90,000,000 : 90,000,000]$ (Factor 10^6).
- **Ideal Type:** `int32_t` (4 bytes). Although the value fits within a smaller range than longitude, `int32_t` is required to maintain 6-decimal precision, because of that, a 16-bit integer is insufficient.

Longitude (lon):

- **Range:** $[-180 : 180]$ decimal degrees with 6 decimal places.
- **Scaled Range:** $[-180,000,000 : 180,000,000]$ (Factor 10^6).
- **Ideal Type:** `int32_t` (4 bytes). This provides the necessary range for global coverage with decametric precision, fitting well within the $\pm 2.1 \times 10^9$ limit of the data type.

Altitude (alt):

- **Range:** $[0 : 10,000]$ meters with centimeter precision.
- **Scaled Range:** $[0 : 1,000,000]$ (Factor 10^2).
- **Ideal Type:** `int32_t` (4 bytes). While a 16-bit integer is capped at 65,535, `int32_t` comfortably handles altitude in centimeters without risk of overflow.

Time and Satellites:

- **Ideal Type:** `uint8_t` (1 byte each). Hours (0-23), minutes (0-59), seconds (0-59), and satellite count are all stored in 8-bit integers as they never exceed the byte limit (0-255).

Temperature (temp):

- **Range:** $[-10 : 50]°C$ with two decimal places.
- **Scaled Range:** $[-1000 : 5000]$ (Factor 10^2).
- **Ideal Type:** `int16_t` (2 bytes). This type provides a 50% size reduction compared to a `float`, while maintaining the required $0.01°C$ resolution.

Relative Humidity(hum):

- **Range:** $[0 : 100]\%$ with two decimal places.
- **Scaled Range:** $[0 : 10000]$ (Factor 10^2).
- **Ideal Type:** `uint16_t` (2 bytes). Since a single byte (`uint8_t`) is limited to 255, a 16-bit container is necessary to store the 0.1% resolution.

Light, and Moisture (light, moisture):

- **Range:** $[0 : 100]\%$ with one decimal place.
- **Scaled Range:** $[0 : 1000]$ (Factor 10^1).
- **Ideal Type:** `uint16_t` (2 bytes). Same as Relative Humidity.

Color Normalization (r_norm, g_norm, b_norm):

- **Range:** $[0 : 100]\%$.
- **Ideal Type:** `uint8_t` (1 byte). The percentage fits perfectly within the 8-bit range (0-255), minimizing the footprint for color representation.

Accelerometer Axes (x, y, z):

- **Range:** $\pm 12.7\, m/s^2$ (covers gravity and moderate tilt).
- **Scaled Range:** $[-127 : 127]$ (Factor 10^1).

- **Ideal Type:** `int8_t` (1 byte per axis). This maximizes payload efficiency while providing $0.1\,m/s^2$ resolution, sufficient for detecting orientation and structural failure.

The system architecture prioritizes high-resolution environmental readings over the full dynamic range of the accelerometer. Given that the accelerometer's primary function is to detect orientation or determine if the plant has tipped over, the full $\pm 19.6\,m/s^2$ range is unnecessary. By capping this range, the system can compress each axis into a single byte. This optimization saves significant bandwidth, which is instead allocated to maintaining decimal precision for Temperature, Relative Humidity, Light, and Moisture.

Figure 1 and Table 4 illustrate the final distribution of the binary frame. The `__attribute__((packed))` in C prevents the compiler from adding padding, ensuring the frame is exactly as described.

Table 4: LoRaWAN Binary Payload Structure

Byte Offset	Field	Data Type	Scaling	Range
0 - 3	Latitude	int32_t	10^6	$\pm$90.000000°
4 - 7	Longitude	int32_t	10^6	$\pm$180.000000°
8 - 11	Altitude	int32_t	10^2	0 - 10,000.00 m
12 - 14	Time	uint8_t[3]	-	HH:MM:SS
15	GPS Sats	uint8_t	-	0 - 255
16 - 17	Temp	int16_t	10^2	-10.00 - 50.00°C
18 - 19	Humidity	uint16_t	10^2	0 - 100.00%
20 - 21	Light	uint16_t	10^1	0 - 100.0%
22 - 23	Moisture	uint16_t	10^1	0 - 100.0%
24 - 26	RGB Norm	uint8_t[3]	%	0 - 100%
27 - 29	Accel XYZ	int8_t[3]	10^1	-12.8 to 12.7 m/s^2

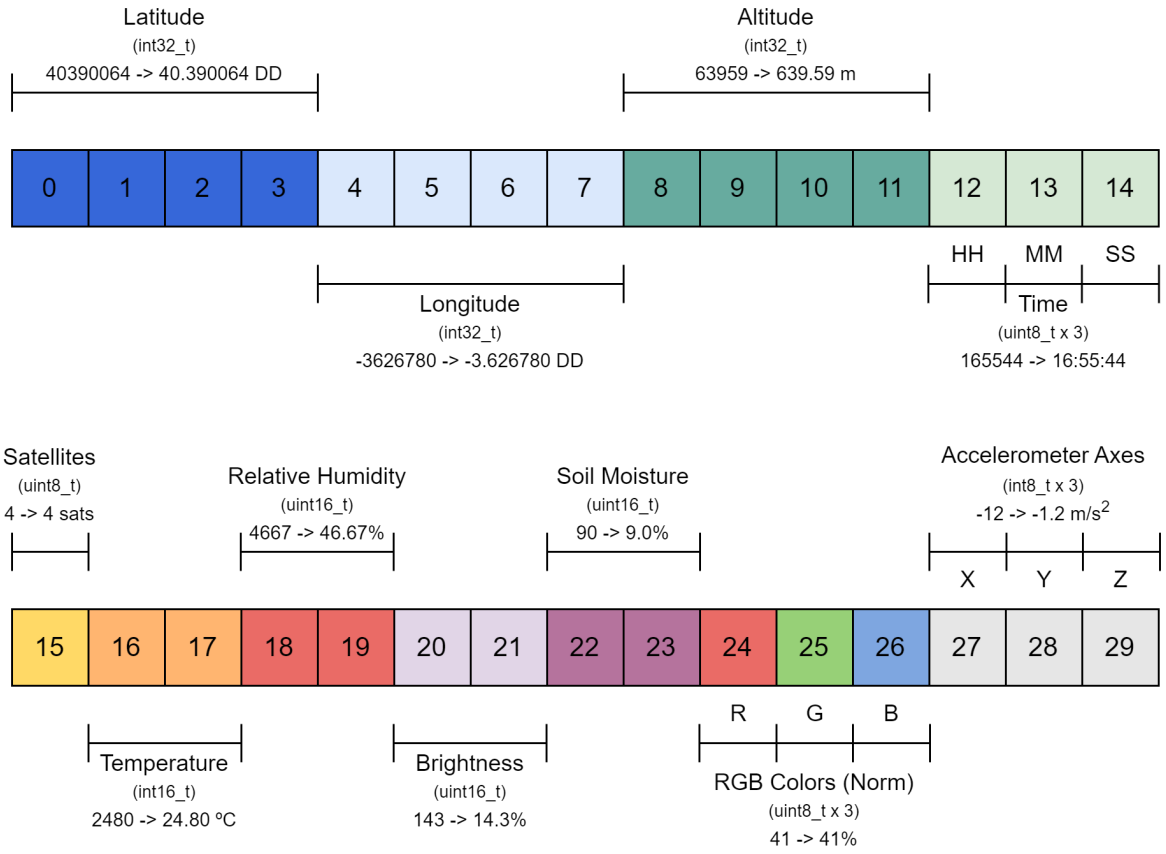

Figure 1: LoRaWAN Binary Payload Structure

3.1.5 Main Application Control Flow

The **main** function acts as the system coordinator, ensuring timing compliance and thread synchronization through Zephyr semaphores. The loop follows a 60-second duty cycle to monitor the plant health and transmit the report via the LoRaWAN stack. The transmission interval was increased from 30 to 60 seconds to ensure long-term system stability. During initial testing, a shorter duty cycle led to buffer saturation issues.

Listing 4: Core Application Logic in main()

```
int main(void)
{
    printk("==== Plant Monitoring System (ResIoT/LoRaWAN) ====\n");

    /* 1. Hardware Initialization */
    if (gps_init(&gps) || adc_init(&pt) || adc_init(&sm) ||
        accel_init(&accel, ACCEL_RANGE) || temp_hum_init(&th, TEMP_HUM_RESOLUTION) ||
        color_init(&color, COLOR_GAIN, COLOR_INTEGRATION_TIME) ||
        rgb_led_init(&rgb_leds) || rgb_led_off(&rgb_leds)) {
        LOG_ERR("Hardware initialization failed. Aborting.");
        return -1;
    }

    /* 2. LoRaWAN Stack Initialization */
    if (init_lorawan() < 0) {
        LOG_ERR("LoRaWAN stack initialization failed.");
        return -1;
    }

    /* 3. Thread Launch */
    start_sensors_thread(&ctx, &measure);
    start_gps_thread(&ctx, &measure);

    /* 4. Join Network */
    if (join_lorawan() < 0) {
        return -1;
    }

    /* 5. Main Loop: Sensor Sampling & LoRaWAN Transmission */
    while (1) {
        /* Request new readings from threads */
        k_sem_give(ctx.sensors_sem);
        k_sem_give(ctx.gps_sem);

        /* Wait for thread completion */
        k_sem_take(ctx.main_sensors_sem, K_FOREVER);
        k_sem_take(ctx.main_gps_sem, K_FOREVER);

        get_measurements();

        /* Send uplink message */
        int ret = lorawan_send(1, (uint8_t *)&main_data, sizeof(main_data),
            LORAWAN_MSG_UNCONFIRMED);
        if (ret < 0) {
            LOG_ERR("LoRaWAN transmission failed: %d", ret);
        } else {
            LOG_INF("Data packet sent successfully (%d bytes)", sizeof(main_data));
        }

        display_measurements();
        k_sleep(DELAY);
    }
}
```

3.1.6 get_measurements() and display_measurements() Functions

The data processing logic is divided into two primary functions that handle the translation between raw sensor data, human-readable units, and the optimized binary format required for LoRaWAN transmission.

The `get_measurements` function retrieves the latest values from the shared atomic measurement structure.

Listing 5: Get measurements and binary scaling

```
static void get_measurements(void)
{
    // GPS Data
    main_data.lat = (int32_t)atomic_get(&measure.gps_lat);
    main_data.lon = (int32_t)atomic_get(&measure.gps_lon);
    main_data.alt = (int32_t)atomic_get(&measure.gps_alt);
    main_data.sats = (uint8_t)atomic_get(&measure.gps_sats);

    // Time Decompression: HHMMSS -> [HH, MM, SS]
    uint32_t full_time = (uint32_t)atomic_get(&measure.gps_time);
    main_data.time[0] = (uint8_t)(full_time / 10000);
    main_data.time[1] = (uint8_t)((full_time / 100) % 100);
    main_data.time[2] = (uint8_t)(full_time % 100);

    // Temperature and Humidity
    main_data.temp = (int16_t)atomic_get(&measure.temp);
    main_data.hum = (uint16_t)atomic_get(&measure.hum);

    // Soil and Light
    main_data.light = (uint16_t)atomic_get(&measure.brightness);
    main_data.moisture = (uint16_t)atomic_get(&measure.moisture);

    // Color Normalization (0-100%)
    uint32_t clear = atomic_get(&measure.clear);
    if (clear > 0) {
        main_data.r_norm = (uint8_t)((atomic_get(&measure.red) * 100) / clear);
        main_data.g_norm = (uint8_t)((atomic_get(&measure.green) * 100) / clear);
        main_data.b_norm = (uint8_t)((atomic_get(&measure.blue) * 100) / clear);
    }

    // Accelerometer data
    main_data.x_axis = (int8_t)(atomic_get(&measure.accel_x) / 10);
    main_data.y_axis = (int8_t)(atomic_get(&measure.accel_y) / 10);
    main_data.z_axis = (int8_t)(atomic_get(&measure.accel_z) / 10);
}
```

The **display_measurements** function provides local feedback via the serial console.

Listing 6: Local sensor report via Serial

```
static void display_measurements(void)
{
    printk("-------------- SENSOR REPORT --------------\n");

    // 1. Soil Moisture
    printk("MOISTURE: Raw: %ld | LoRa: %u | Value: %.1f%%\n",
           atomic_get(&measure.moisture), main_data.moisture, (double)main_data.moisture /
               10.0);

    // 2. Light
    printk("LIGHT:    Raw: %ld | LoRa: %u | Value: %.1f%%\n",
           atomic_get(&measure.brightness), main_data.light, (double)main_data.light / 10.0);
```

```
    // 3. Temperature & Humidity
    printk("TEMP:     Raw: %ld | LoRa: %d | Value: %.2f C\n",
           atomic_get(&measure.temp), main_data.temp, (double)main_data.temp / 100.0);
    printk("HUMIDITY: Raw: %ld | LoRa: %u | Value: %.2f%%\n",
           atomic_get(&measure.hum), main_data.hum, (double)main_data.hum / 100.0);

    // 4. GPS Location
    printk("LATITUDE: Raw: %ld | LoRa: %d | Value: %.6f\n",
           atomic_get(&measure.gps_lat), main_data.lat, (double)main_data.lat / 1e6);
    printk("LONGITUDE: Raw: %ld | LoRa: %d | Value: %.6f\n",
           atomic_get(&measure.gps_lon), main_data.lon, (double)main_data.lon / 1e6);
    printk("ALTITUDE: Raw: %ld | LoRa: %d | Value: %.2f m\n",
           atomic_get(&measure.gps_alt), main_data.alt, (double)main_data.alt / 100.0);

    // 5. GPS Sats & Time
    printk("GPS SATS: Raw: %ld | LoRa: %u | Value: %u satellites\n",
           atomic_get(&measure.gps_sats), main_data.sats, main_data.sats);

    printk("GPS TIME: Raw: %ld | LoRa: [%02d,%02d,%02d] | Value: %02d:%02d:%02d\n",
           atomic_get(&measure.gps_time),
           main_data.time[0], main_data.time[1], main_data.time[2],
           main_data.time[0], main_data.time[1], main_data.time[2]);

    // 6. Color (Normalizado en LoRa y Value)
    printk("COLOR:    Raw R:%ld G:%ld B:%ld | LoRa R:%u%% G:%u%% B:%u%%\n",
           atomic_get(&measure.red), atomic_get(&measure.green), atomic_get(&measure.blue),
           main_data.r_norm, main_data.g_norm, main_data.b_norm);

    // 7. Accelerometer
    printk("ACCEL:    Raw X:%ld Y:%ld Z:%ld | LoRa: X: %d Y: %d Z: %d | Value X:%.1f Y:%.1f
        Z:%.1f m/s2\n",
           atomic_get(&measure.accel_x), atomic_get(&measure.accel_y),
               atomic_get(&measure.accel_z),
           main_data.x_axis, main_data.y_axis, main_data.z_axis,
           (double)main_data.x_axis / 10.0, (double)main_data.y_axis / 10.0,
               (double)main_data.z_axis / 10.0);

    printk("----------------------------------------\n\n");
}
```

```
[01:08:09.002,000] <inf> plant_monitor_main: Data packet sent successfully (30 bytes)
------------- SENSOR REPORT -------------
MOISTURE:  Raw: 87 | LoRa: 87 | Value: 8.7%
LIGHT:     Raw: 133 | LoRa: 133 | Value: 13.3%
TEMP:      Raw: 2530 | LoRa: 2530 | Value: 25.30 C
HUMIDITY:  Raw: 4533 | LoRa: 4533 | Value: 45.33%
LATITUDE:  Raw: 40390192 | LoRa: 40390192 | Value: 40.390192
LONGITUDE: Raw: -3628918 | LoRa: -3628918 | Value: -3.628918
ALTITUDE:  Raw: 60959 | LoRa: 60959 | Value: 609.59 m
GPS SATS:  Raw: 6 | LoRa: 6 | Value: 6 satellites
GPS TIME:  Raw: 174119 | LoRa: [17,41,19] | Value: 17:41:19
COLOR:     Raw R:665 G:543 B:421 | LoRa R:41% G:33% B:26%
ACCEL:     Raw X:0 Y:33 Z:979 | LoRa: X: 0 Y: 3 Z: 97 | Value X:0.0 Y:0.3 Z:9.7 m/s2
----------------------------------------
```

Figure 2: LoRaWAN Local Feedback

3.1.7 Downlink Control Logic

The system is capable of receiving remote commands (Downlink) from the ResIoT platform. This is managed via a callback function that listens for specific strings to control the plant's status indicator (RGB LED).

Listing 7: Downlink Callback for RGB Control

```
static void dl_callback(uint8_t port, uint8_t flags, int16_t rssi, int8_t snr,
                        uint8_t len, const uint8_t *hex_data) {
    if (hex_data) {
        /* Control RGB LED based on server commands */
        if (strncmp((const char *)hex_data, "OFF", len) == 0) rgb_led_off(&rgb_leds);
        else if (strncmp((const char *)hex_data, "Green", len) == 0) rgb_green(&rgb_leds);
        else if (strncmp((const char *)hex_data, "Red", len) == 0) rgb_red(&rgb_leds);
    }
}
```

3.2 Zephyr RTOS

3.2.1 prj_nucleo_wl55jc.conf

The project configuration file (`prj_nucleo_wl55jc.conf`) defines the kernel services and device drivers required for the LoRaWAN-enabled plant monitoring system. Beyond enabling basic General Purpose Input-Output (GPIO), Analog-to-Digital Converter (ADC), and Inter-Integrated Circuit (I2C) peripherals, this file is critical for activating the LoRaWAN stack and the EU868 regional settings.

Additionally, it configures Non-Volatile Storage (NVS) and the Zephyr Settings subsystem to persist network parameters such as the DevNonce, which is mandatory for secure OTAA joins.

Listing 8: prj_nucleo_wl55jc.conf

```
# General system configuration
CONFIG_MAIN_STACK_SIZE=2048
CONFIG_SYSTEM_WORKQUEUE_STACK_SIZE=2048
CONFIG_ASSERT=y

# Logging and debugging
CONFIG_LOG=y
CONFIG_LOG_DEFAULT_LEVEL=3
CONFIG_STDOUT_CONSOLE=y
CONFIG_CBPRINTF_FP_SUPPORT=y

# Console and UART
CONFIG_UART_CONSOLE=y
CONFIG_CONSOLE=y
CONFIG_PRINTK=y
CONFIG_SERIAL=y

# Device drivers
CONFIG_GPIO=y
CONFIG_ADC=y
CONFIG_I2C=y
CONFIG_SENSOR=y
CONFIG_FLASH=y
CONFIG_FLASH_MAP=y
CONFIG_FLASH_PAGE_LAYOUT=y
CONFIG_STREAM_FLASH=y
CONFIG_IMG_MANAGER=y

# UART configuration
CONFIG_UART_ASYNC_API=y
CONFIG_UART_INTERRUPT_DRIVEN=y
```

```
# LoRa and LoRaWAN configuration
CONFIG_LORA=y
CONFIG_LORAWAN=y
CONFIG_LORAMAC_REGION_EU868=y
CONFIG_LORAWAN_NVM_SETTINGS=y # Use NVM to store OTAA parameters
CONFIG_ENTROPY_GENERATOR=y # Entropy source

# Non-volatile storage (NVS)
CONFIG_NVS=y
CONFIG_SETTINGS=y # Required to store LoRaWAN DevNonce and other settings

# Zephyr Bus configuration
CONFIG_ZBUS=y
CONFIG_ZBUS_CHANNEL_NAME=y

# Thread and stack analysis
# CONFIG_INIT_STACKS=y
# CONFIG_THREAD_STACK_INFO=y
# CONFIG_THREAD_ANALYZER=y
# CONFIG_THREAD_ANALYZER_AUTO=y
# CONFIG_THREAD_NAME=y
```

3.2.2 nucleo_wl55jc.overlay

The DeviceTree overlay file (`nucleo_wl55jc.overlay`) customizes the hardware description for the STM32WL55JC-based node to support plant monitoring. It defines a RGB LED structure using GPIO-controlled nodes, with aliases for "red", "green", and "blue" to simplify actuator control via the downlink commands received from the network. The overlay also configures ADC1 for analog sensing (light and moisture), I2C2 for environmental and color sensors, and Universal Synchronous Asynchronous Receiver-Transmitter (USART)1 for the external GPS module.

Listing 9: nucleo_wl55jc.overlay

```
#include <zephyr/dt-bindings/pinctrl/stm32-pinctrl.h>

/ {
    zephyr,user {
     io-channels = <&adc1 5>;
  };

  gpio_keys {
     compatible = "gpio-keys";

     accel_interrup: adc_in1_pb4 {
        label = "FF";
        gpios = <&gpiob 4 (GPIO_ACTIVE_LOW | GPIO_PULL_UP)>;
        zephyr,code = <INPUT_KEY_3>;
     };
  };

    rgb_leds {
        compatible = "gpio-leds";

        rgb_red: rgb_0 {
            gpios = <&gpioa 6 GPIO_ACTIVE_LOW>;
            label = "Red RGB LED";
        };
        rgb_green: rgb_1 {
            gpios = <&gpioa 7 GPIO_ACTIVE_LOW>;
            label = "Green RGB LED";
        };
        rgb_blue: rgb_2 {
```

```
            gpios = <&gpioa 9 GPIO_ACTIVE_LOW>;
            label = "Blue RGB LED";
        };
    };

    aliases {
        red = &rgb_red;
        green = &rgb_green;
        blue = &rgb_blue;
        led0 = &blue_led_1;  // This is LED1 as labeled STM32WL55JC board's
     led1 = &green_led_2;  // This is LED2 as labeled STM32WL55JC board's
     led2 = &red_led_3;    // This is LED3 as labeled STM32WL55JC board's
        ff0 = &accel_interrup;
    };


};

&adc1 {
  #address-cells = <1>;
  #size-cells = <0>;

  channel@5 {
     reg = <5>;
     zephyr,gain = "ADC_GAIN_1";
     zephyr,reference = "ADC_REF_INTERNAL";
     zephyr,acquisition-time = <ADC_ACQ_TIME_DEFAULT>;
     zephyr,resolution = <12>;
  };
};

&i2c2 {
    status = "okay";
    clock-frequency = <I2C_BITRATE_FAST>; // 400kHz
};

&usart1 {
  dmas = <&dmamux1 11 18 (STM32_DMA_PERIPH_TX | STM32_DMA_PRIORITY_HIGH)
     &dmamux1 1 17 (STM32_DMA_PERIPH_RX | STM32_DMA_PRIORITY_HIGH)>;
  dma-names = "tx", "rx";
  pinctrl-0 = <&usart1_tx_pb6 &usart1_rx_pb7>;
  pinctrl-names = "default";
  current-speed = <9600>;
  status = "okay";
};

&dma1 {
  status = "okay";
};

&dma2 {
  status = "okay";
};

&dmamux1 {
  status = "okay";
};

&gpiob {
    status = "okay";
};
```

3.3 Thread Stack and CPU Usage Analysis

Zephyr provides runtime diagnostics that allow monitoring of the stack usage and CPU load of each thread in the system. The output shown in the image presents detailed information for all active threads, including their stack consumption, remaining free stack space, and the total number of CPU cycles executed since startup.

```
Thread analyze:
 gps_thread          : STACK: unused 752 usage 272 / 1024 (26 %); CPU: 0 %
                     : Total CPU cycles used: 4612
 sensors_thread      : STACK: unused 576 usage 448 / 1024 (43 %); CPU: 0 %
                     : Total CPU cycles used: 35239
 thread_analyzer     : STACK: unused 552 usage 472 / 1024 (46 %); CPU: 0 %
                     : Total CPU cycles used: 181018
 sysworkq            : STACK: unused 944 usage 1104 / 2048 (53 %); CPU: 0 %
                     : Total CPU cycles used: 4833785
 logging             : STACK: unused 1444 usage 604 / 2048 (29 %); CPU: 0 %
                     : Total CPU cycles used: 12029906
 idle                : STACK: unused 192 usage 64 / 256 (25 %); CPU: 99 %
                     : Total CPU cycles used: 2863406289
 main                : STACK: unused 880 usage 1168 / 2048 (57 %); CPU: 0 %
                     : Total CPU cycles used: 2124172
 ISR0                : STACK: unused 1556 usage 492 / 2048 (24 %)
```

Figure 3: Thread stack and CPU usage

Overall, the reported values confirm that:

- All thread stacks remain within safe usage ranges, below the recommended 60% threshold.
- CPU usage distribution behaves as expected for a sensor-driven, event-based embedded application.
- The idle thread dominates CPU cycles, indicating efficient low-power execution and minimal background processing overhead.

3.4 Compilation and Flashing Output Analysis

During the compilation process, Zephyr generates a memory usage summary that indicates how much Flash and Random-Access Memory (RAM) the final application occupies. As shown in Figure 4, after linking the executable `zephyr.elf`, the memory report provides the following information:

- **FLASH:** 135.432 KB used out of 256 KB (approximately 51.66%).
- **RAM:** 20.608 KB used out of 64 KB (approximately 31.45%).

```
-- west build: building application
[242/242] Linking C executable zephyr\zephyr.elf
Memory region         Used Size  Region Size  %age Used
           FLASH:      135432 B       256 KB     51.66%
             RAM:       20608 B        64 KB     31.45%
        IDT_LIST:          0 GB        32 KB      0.00%
```

Figure 4: Compilation memory usage report

3.5 Flashing the Firmware onto the STM32WL55

The Figure 5 corresponds to the flashing process performed using `STM32CubeProgrammer`, which communicates with the NUCLEO-WL55JC board via the onboard ST-LINK debugger. The tool successfully identifies the target device, displaying key details. After loading the generated `zephyr.hex` file (**132.26KB**), the programmer performs the necessary steps to start the application.

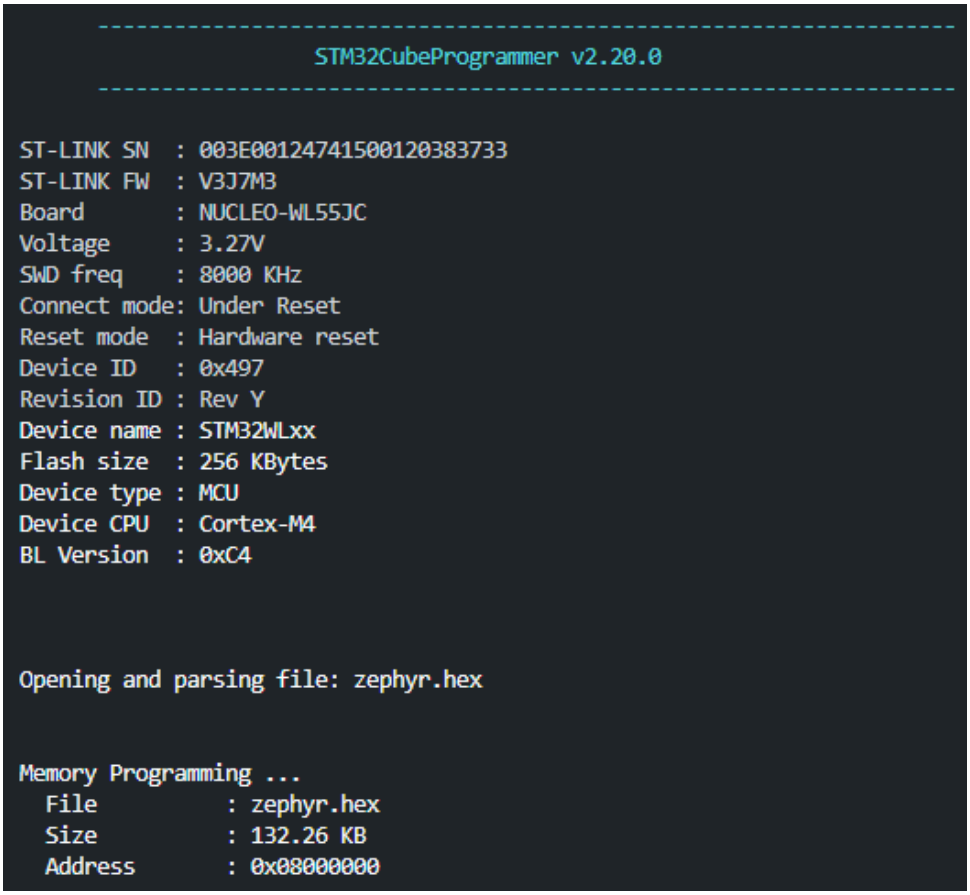

Figure 5: Flashing process using STM32CubeProgrammer

3.6 Code Documentation

The project documentation is generated automatically through a continuous integration workflow implemented using a GitHub Action (subsection A.3). This workflow executes the Doxygen engine, which extracts structured information directly from the annotated comments within the source code. By following Doxygen's documentation conventions, each module, function, and data structure is described where it is implemented, ensuring that the documentation remains consistent with the evolving codebase.

Whenever new commits are pushed to the repository, the GitHub Action is triggered, automatically regenerating the documentation and preventing discrepancies between the implementation and its technical description. As part of the same workflow, the generated documentation is automatically deployed to a GitHub Pages site, making it accessible online without requiring manual intervention.

The documentation can be accessed directly through the following link: `https://estelamb.github.io/Sensor_Networks/`.

4 Network Server Configuration

4.1 Node definition

The LoRaWAN end-node is registered within the ResIoT platform under the name **SN_NODE_J**. To ensure secure and unique connectivity within the network, the device is configured using the OTAA method. Its primary identifiers include:

- **Device EUI:** 7a:39:32:35:59:37:91:94.
- **Application EUI:** SEN_NET_2526.
- **Application KEY:** f3:1c:2e:8b:c6:71:28:1d:51:16:f0:8f:f0:b7:92:8f.

The node is set as a LoRaWAN Class A device, operating on the EU868 frequency band. The internal data architecture of the node is organized through a series of **Node Fields** that act as the destination for the decoded payload values, defined in Table 5.

Table 5: Node Field Definitions for SN_NODE_J

Field Tag	Content Type	Field Tag	Content Type
Light	Numeric	Latitude	Numeric
Moisture	Numeric	Longitude	Numeric
Temperature	Numeric	Altitude	Numeric
Humidity	Numeric	Time	String
Red / Green / Blue	Numeric	Sats	Numeric
X / Y / Z	Numeric		

4.2 LUA Scripting

The data transmitted by the end-node is received by the ResIoT platform as a hexadecimal payload. To make this information actionable, a LUA script is implemented to decode the raw byte array into human-readable sensor values. The script is structured into three main functional blocks:

- **Binary Conversion Logic:** In LoRaWAN payloads, data is sent in binary format. The function `bytesToInt` handles Little Endian conversion. It reconstructs integers by iterating through the byte array and applying the formula:

$$Value = \sum_{i=0}^{size-1} bytes[start + i] \cdot 256^i \tag{1}$$

  It also includes logic to handle signed integers using Two's Complement verification.

Listing 10: LUA SN_TEST_J - Binary Conversion Logic

```
-- Helper function to convert bytes to Integer (Little Endian)
function bytesToInt(bytes, start, size, signed)
    local val = 0
    for i = 0, size - 1 do
        val = val + (bytes[start + i] * (256 ^ i))
    end
    if signed and val >= (256 ^ size) / 2 then
        val = val - (256 ^ size)
    end
    return val
end
```

- **Payload Decapsulation:** The `parsePayload` function slices the buffer to extract specific sensor variables based on predefined offsets matching the Payload Encoding subsection. Each variable is scaled according to its resolution and unit requirements.

Listing 11: LUA SN_TEST_J - Payload Parsing

```
function parsePayload(appeui, deveui, payload)
    -- Convert hex payload to byte array
    local bytes = resiot_hexdecode(payload)

    -- 1. GPS Data (1 to 17)
    local lat = bytesToInt(bytes, 1, 4, true) / 1000000.0
    local lon = bytesToInt(bytes, 5, 4, true) / 1000000.0
    local alt = bytesToInt(bytes, 9, 4, true) / 100.0

    -- GPS Time (13-15) HHMMSS
    local hh = bytes[13]
    local mm = bytes[14]
    local ss = bytes[15]
    local time = string.format("%02d:%02d:%02d", hh, mm, ss)

    local sats = bytes[16]

    -- 2. Temperature and Humidity Data (17 to 20)
    local temp = bytesToInt(bytes, 17, 2, true) / 100.0
    local hum = bytesToInt(bytes, 19, 2, false) / 100.0

    -- 3. Brightness and Moisture (21 to 24)
    local light   = bytesToInt(bytes, 21, 2, false) / 10.0
    local moisture = bytesToInt(bytes, 23, 2, false) / 10.0

    -- 4. Color RGB (Offsets 25 to 27)
    local r = bytes[25]
    local g = bytes[26]
    local b = bytes[27]

    -- 5. Accelerometer (28 to 30)
    -- These are int8 (signed). If value > 127, it's negative.
    local function toInt8(b) return b > 127 and b - 256 or b end
    local x = toInt8(bytes[28]) / 10.0
    local y = toInt8(bytes[29]) / 10.0
    local z = toInt8(bytes[30]) / 10.0

    ...
end
```

- **Platform Integration:** After decoding, the script utilizes the `resiot_setnodevalue` function. This updates the device within the platform, mapping the decoded variables to specific node fields. Also, debug logs are generated to facilitate monitoring and troubleshooting.

Listing 12: LUA SN_TEST_J - Debug and Node Update

```
function parsePayload(appeui, deveui, payload)
    ...

    -- Debug Logs
    resiot_debug(string.format("GPS: Lat: %.6f, Long: %.6f, Alt: %.2f, Time: %s, Sats:
        %d", lat, lon, alt, time, sats))
    resiot_debug(string.format("Sensors: Temp: %.2f, Hum: %.2f, Light: %.1f, Moisture:
        %.1f", temp, hum, light, moisture))
    resiot_debug(string.format("Color: R:%d, G:%d, B:%d", r, g, b))
    resiot_debug(string.format("Accel: X:%.1f, Y:%.1f, Z:%.1f", x, y, z))

    -- Update Nodes in ResIoT
    resiot_setnodevalue(appeui, deveui, "Latitude", lat)
    resiot_setnodevalue(appeui, deveui, "Longitude", lon)
    resiot_setnodevalue(appeui, deveui, "Altitude", alt)
    resiot_setnodevalue(appeui, deveui, "Time", time)
```

```
    resiot_setnodevalue(appeui, deveui, "Sats", sats)
    resiot_setnodevalue(appeui, deveui, "Temperature", temp)
    resiot_setnodevalue(appeui, deveui, "Humidity", hum)
    resiot_setnodevalue(appeui, deveui, "Light", light)
    resiot_setnodevalue(appeui, deveui, "Moisture", moisture)
    resiot_setnodevalue(appeui, deveui, "Red", r)
    resiot_setnodevalue(appeui, deveui, "Green", g)
    resiot_setnodevalue(appeui, deveui, "Blue", b)
    resiot_setnodevalue(appeui, deveui, "X", x)
    resiot_setnodevalue(appeui, deveui, "Y", y)
    resiot_setnodevalue(appeui, deveui, "Z", z)
end
```

- **Manual Testing:** The script also includes an `Origin` check, allowing for manual testing with a static hex string or automatic processing when a live uplink is detected from the LoRaWAN gateway.

Listing 13: LUA SN_TEST_J - Manual Testing Block

```
-- --- Main Process ---
Origin = resiot_startfrom()

if Origin == "Manual" then
    -- Test payload (30 bytes hex)
    payload = "0102030405060708091011121314151617181920212223242526272829 30"
    appeui = "70b3d57ed000fc4d"
    deveui = "7a39323559379194"
else
    appeui = resiot_comm_getparam("appeui")
    deveui = resiot_comm_getparam("deveui")
    payload, err = resiot_getlastpayload(appeui, deveui)
end

parsePayload(appeui, deveui, payload)
```

The LUA script is integrated into the ResIoT ecosystem through the LoRaWAN node's automation logic. Rather than running as a static process, the code is embedded within the **On RX** (On Receive) event. This configuration establishes an event-driven architecture where the script remains idle until a new radio frame is successfully demodulated by the gateway and forwarded to the network server.

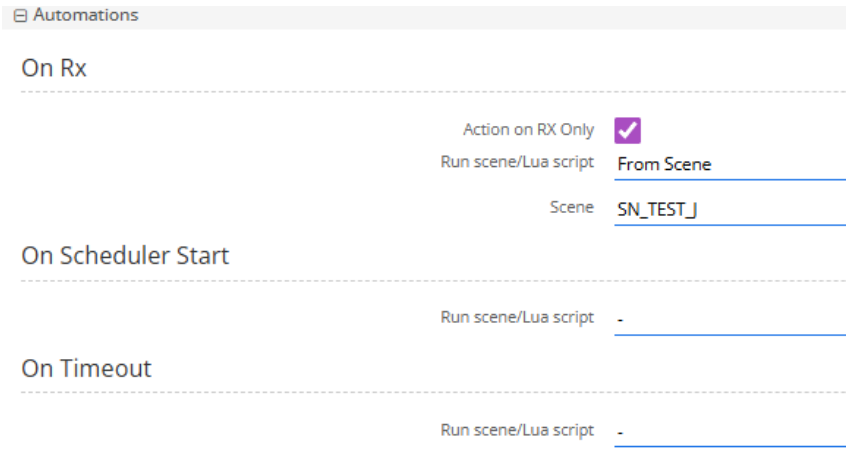

Figure 6: ResIoT Node Automation - On RX Event

Once an uplink packet is received, the platform automatically triggers the execution of the script, passing the device identifiers and the raw hexadecimal payload as input parameters.

4.3 Dashboard

The monitoring interface is organized within a dedicated group named **SN_GROUP_J**. The dashboard consists of a variety of widgets. The following list details the specific widgets and their respective visualization types:

- **Node J - Temperature / Humidity / Brightness / Soil Moisture:** These environmental parameters are displayed using **Gauge** widgets.
- **Node J - Historical Values:** A **Line Chart** is dedicated to tracking the temporal evolution of sensor data over time.
- **Node J - Alarms:** A **Table Values** widget is configured to show the status of the boolean alarm flags, allowing for quick identification of threshold breaches (Temperature / Humidity / Brightness / Soil Moisture).
- **Node J - Leaf Color:** The RGB values are represented through a **Pie chart** widget.
- **Node J - Acceleration (m/s^2):** Acceleration data is visualized using a **Line Chart**.
- **Node J - Location:** A **Maps** widget is used to track the GPS data.

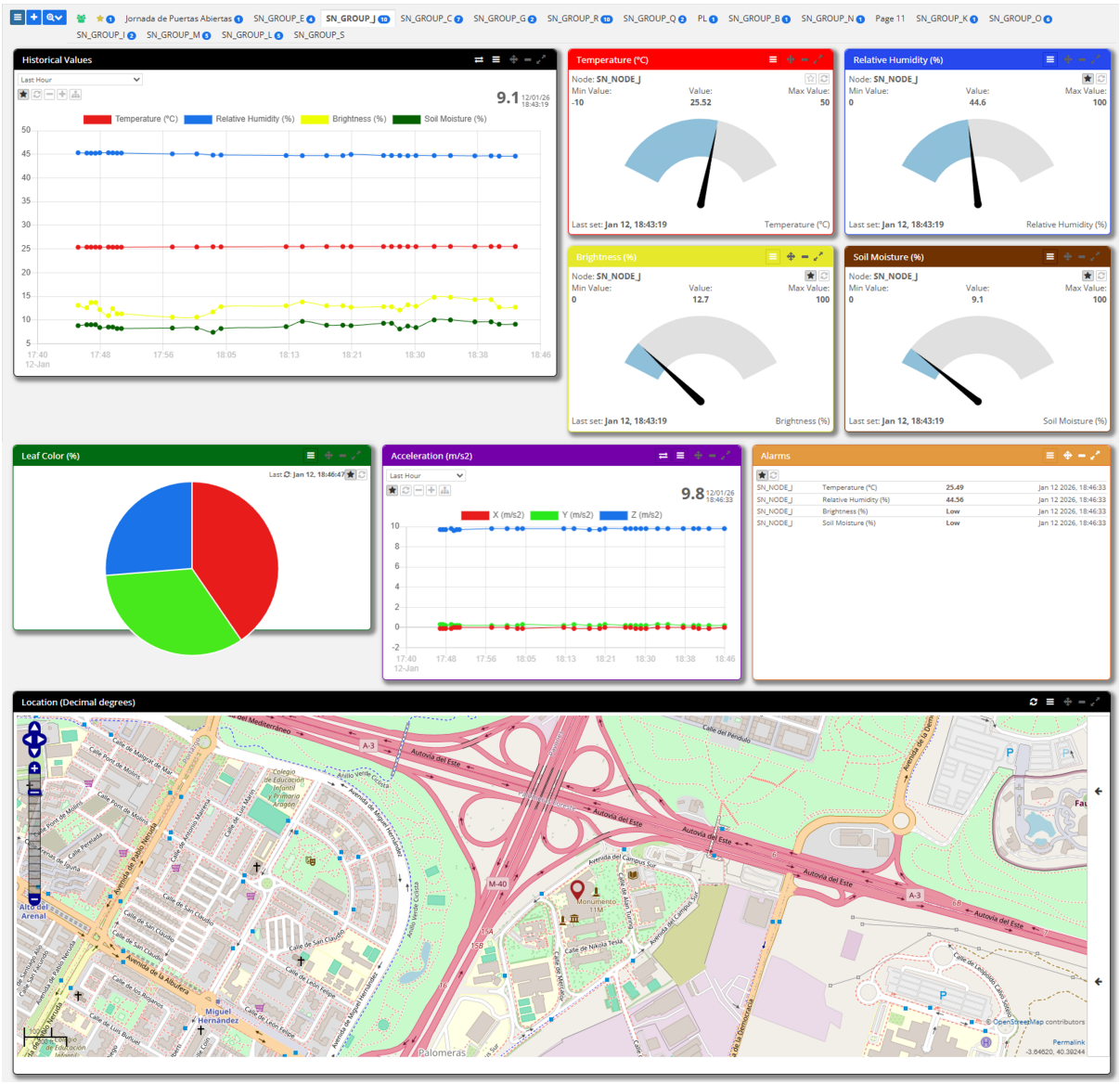

Figure 7: ResIoT Dashboard - SN_GROUP_J

4.3.1 Temperature / Humidity / Brightness / Soil Moisture Gauges

The primary environmental metrics are visualized through four specialized **Gauge** widgets. The configuration for these gauges (as exemplified by the Temperature widget in Figure 8) is defined by the following parameters:

- **Data Mapping:** Each widget is linked to the specific numeric `Node Field` updated by the LUA script.
- **Range Calibration:** To provide context to the raw data, specific minimum and maximum thresholds are established:
  - **Temperature:** Set from -10°C to 50°C.
  - **Humidity / Brightness / Soil Moisture:** Calibrated on a percentage scale.
- **Visual Styling:** Each gauge utilizes a distinct header color.
- **Unit Management:** The widgets are configured with their respective Units of Measurement.
- **Refresh Logic:** The chart includes an **Autorefresh** feature, ensuring the visualization remains synchronized with the latest environmental data processed by the platform.

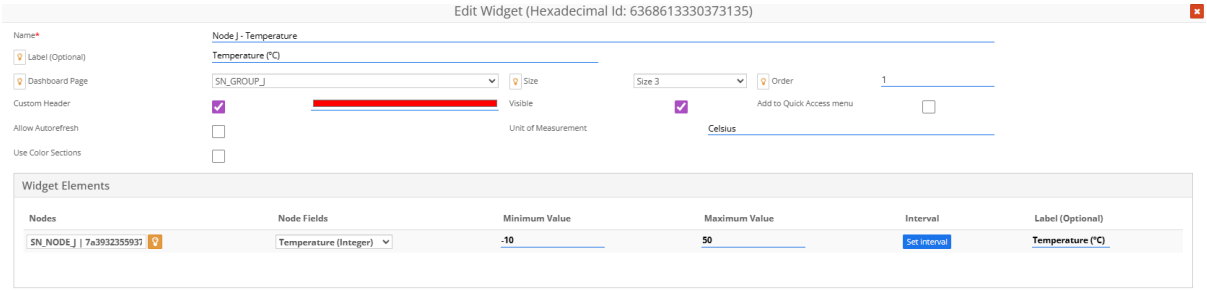

Figure 8: Gauge Configuration - Temperature Example

As shown in Figure 9, the gauges also display the timestamp of the last received data packet (**"Last set"**), confirming the real-time nature of the **On RX** automation process.

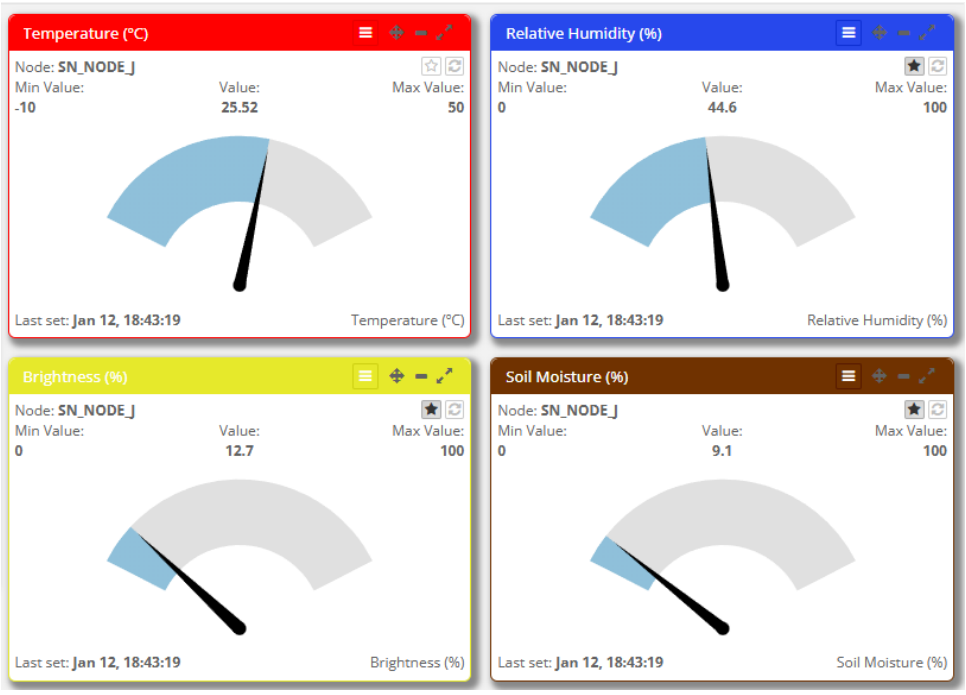

Figure 9: ResIoT Dashboard - Temperature / Humidity / Brightness / Soil Moisture Gauges

4.3.2 Historical Values

The **Node J - Historical Values** widget is a **Line Chart** designed to monitor the temporal evolution of the environment.

Figure 10: ResIoT Dashboard - Historical Values Line Chart

The configuration of this chart, as illustrated in Figure 11, is based on the following technical parameters:

- **Data Aggregation:** The chart simultaneously plots four distinct data series from the `SN_NODE_J` node fields.
- **Color Coding:** Specific colors are assigned to each variable.
- **Time Window:** The widget is configured with a **Last Hour** default interval.
- **Dynamic Scaling:** The Y-axis automatically adjusts to the numeric range of the incoming data, while the X-axis represents the chronological timeline updated by the `On RX` automation.
- **Refresh Logic**.

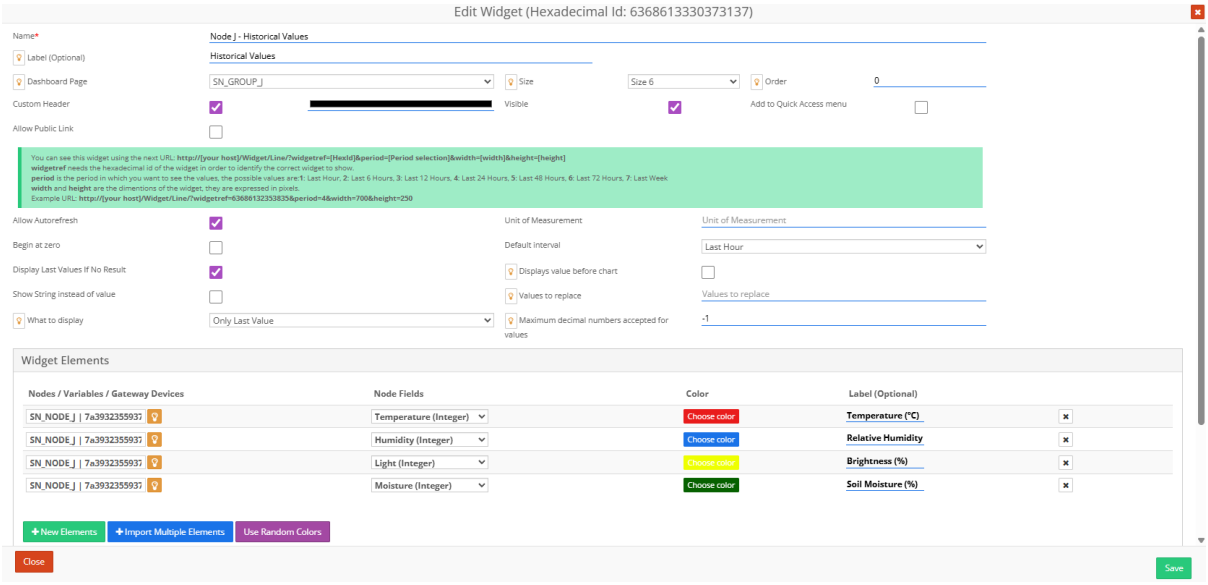

Figure 11: Line Chart Configuration - Historical Values

4.3.3 Alarms

The **Node J - Alarms** widget is implemented as a **Table Values** display. Unlike the gauges, which show raw numerical data, this table is specifically configured to interpret threshold breaches and translate them into human-readable alerts.

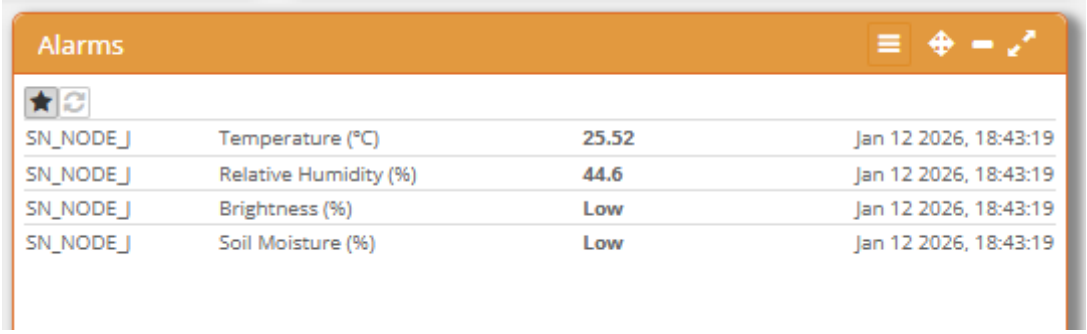

Figure 12: ResIoT Dashboard - Alarms Table

As illustrated in the configuration details in Figure 13, the widget utilizes a conditional string replacement logic to define the status of each variable:

- **Temperature Logic:** The system monitors for extreme conditions, displaying **"High"** if the value exceeds 50 and **"Low"** if it falls below −10.
- **Percentage Thresholds:** For Relative Humidity, Brightness, and Soil Moisture, a unified logic is applied where values above 75 trigger a **"High"** status and values below 25 trigger a **"Low"** status.
- **Real-time Monitoring:** If the current values are within the safe operational range, the table displays the actual numerical value.
- **Data Synchronization:** Every entry in the table is accompanied by a timestamp (`''Date Last Setting''`), ensuring that the alerts correspond to the most recent LoRaWAN uplink processed by the `On RX` automation.
- **Refresh Logic**.

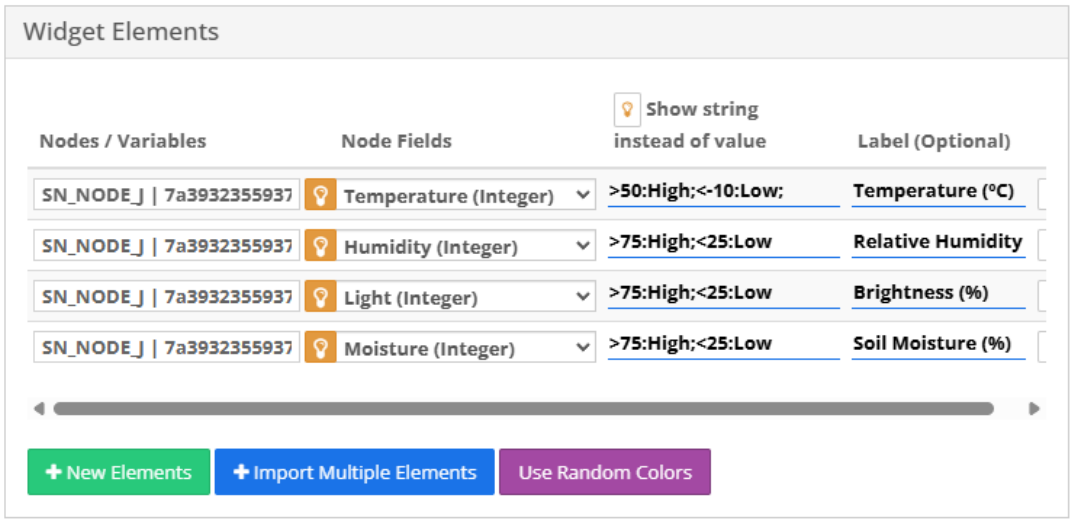

Figure 13: Table Configuration

4.3.4 Leaf Color

The **Node J - Leaf Color** widget provides a qualitative representation of RGB color data using a **Pie Chart** format.

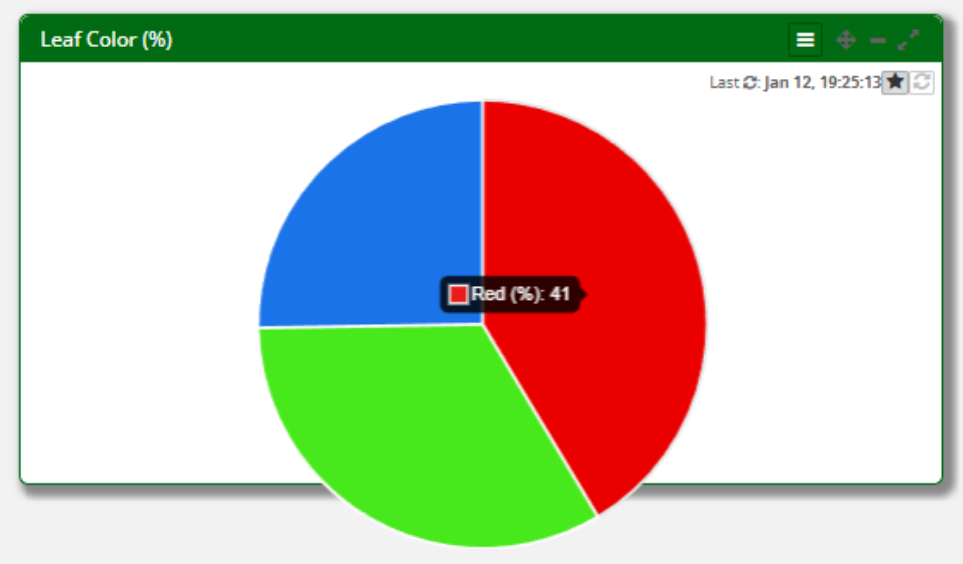

Figure 14: ResIoT Dashboard - Leaf Color Pie Chart

The configuration of this widget, as detailed in Figure 15, includes the following parameters:

- **Data Integration:** The chart aggregates the three colors.
- **Visual Mapping:** Red for the Red (%) field, Green for the Green (%) field, and Blue for the Blue (%) field.
- **Interactive Display:** The widget is configured to show specific values upon user interaction.
- **Refresh Logic**.

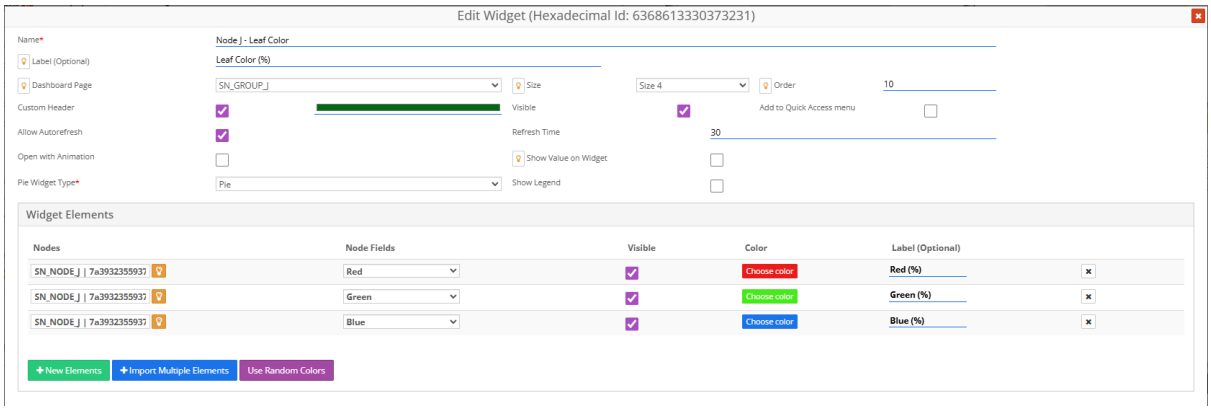

Figure 15: Pie Chart Configuration

4.3.5 Acceleration

Acceleration monitoring is performed through the **Node J - Acceleration** (m/s^2) widget, which utilizes a **Line Chart**.

Figure 16: ResIoT Dashboard - Acceleration Line Chart

The implementation for this chart is defined by the following configuration parameters:

- **Axis Mapping:** The X, Y and Z fields are updated as signed integers by the LUA script to reflect directional acceleration.
- **Visual Representation:** The chart uses a standardized color coding scheme: Red for the **X-axis**, Green for the **Y-axis**, and Blue for the **Z-axis**.
- **Data Interval:** The widget is configured with a **Last Hour** default interval.
- **Scaling and Units:** The Y-axis is scaled in meters per second squared (m/s^2).
- **Refresh Logic**.

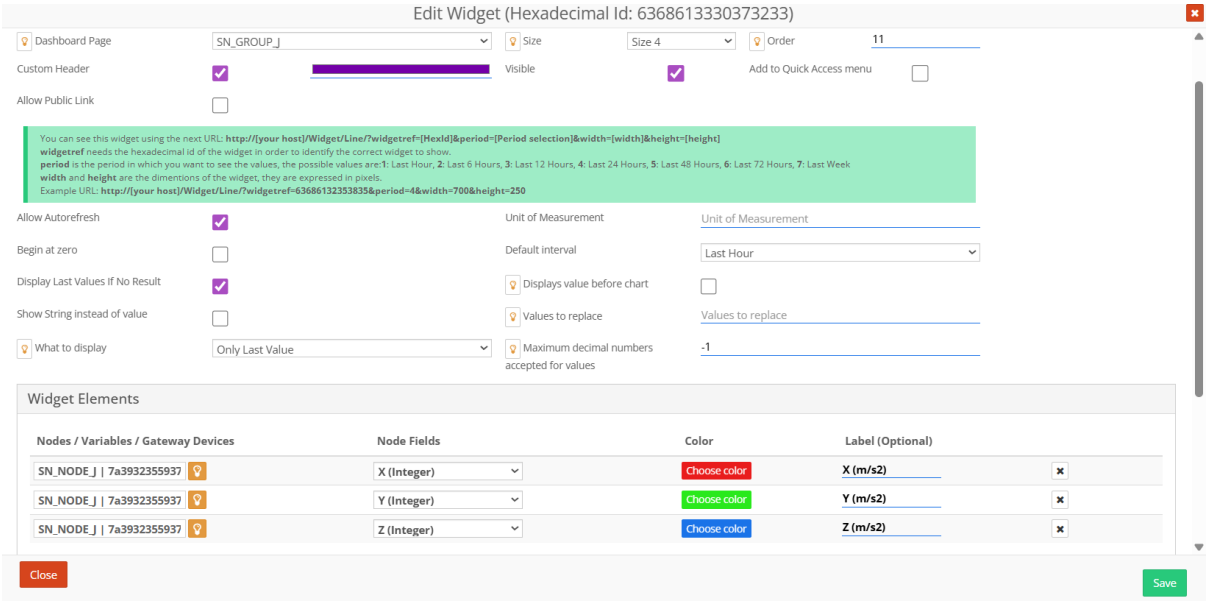

Figure 17: Line Chart Configuration - Acceleration

4.3.6 Location

Geospatial monitoring is provided through the **Node J - Location** widget, which integrates GPS data into an interactive mapping interface.

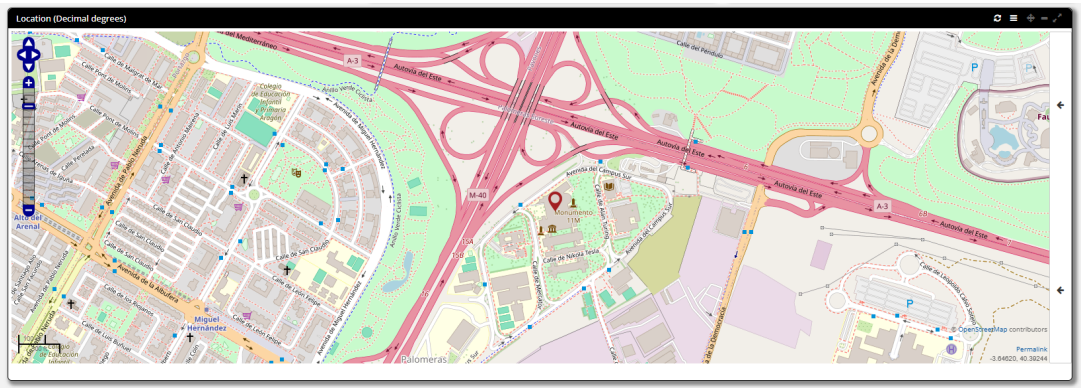

Figure 18: ResIoT Dashboard - Location Map

The technical implementation of the location system is based on the following configurations:

- **Spatial Data Mapping:** As shown in Figure 19, the widget is linked to five critical node fields: `Latitude`, `Longitude`, `Altitude`, `Time`, and `Sats` (Satellite count).

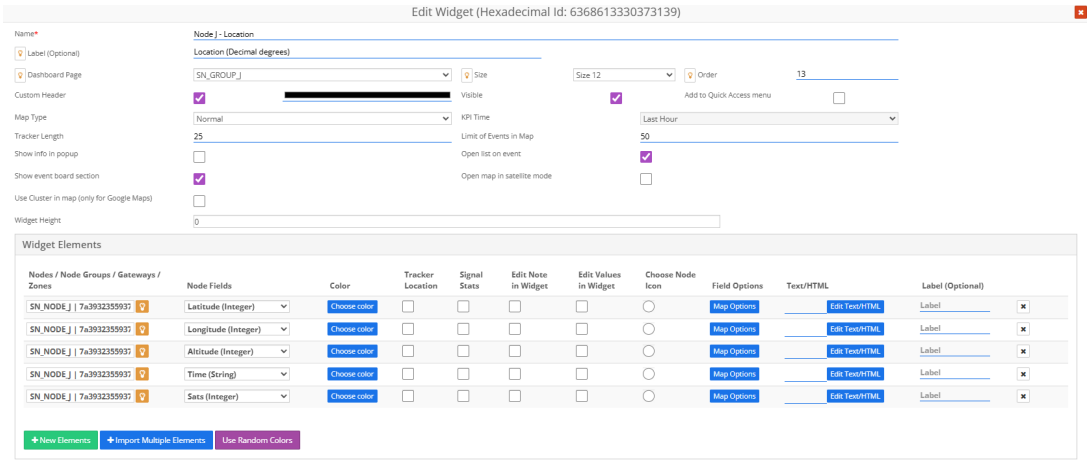

Figure 19: Map Configuration - Location

Clicking on the specific red location pin (as shown in Figure 20) triggers a pop-up window, that provides the latitude, longitude, altitude, the exact time of the record, and the satellite count.

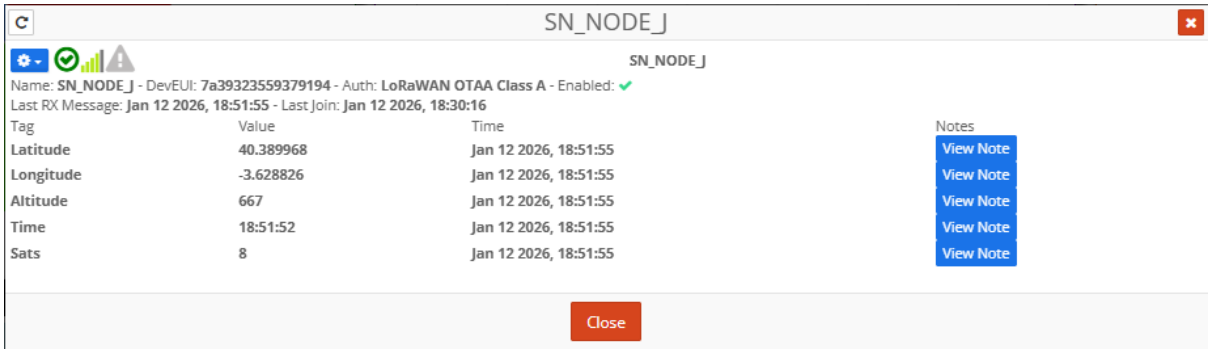

Figure 20: ResIoT Dashboard - Location Info

5 Results

5.1 Unitary Tests

The individual software components and hardware drivers used in this project underwent rigorous **Unitary Testing** during the previous subject. These tests were essential to establish a verified baseline for the current system integration.

5.2 Local Test

A **Local Test** was performed to verify the integrity of the data acquisition and the preliminary encoding of the payload. As shown in Figure 21, the system provides a comprehensive **Sensor Report** via the serial console, allowing for a side-by-side comparison between the raw sensor readings and the formatted LoRaWAN values.

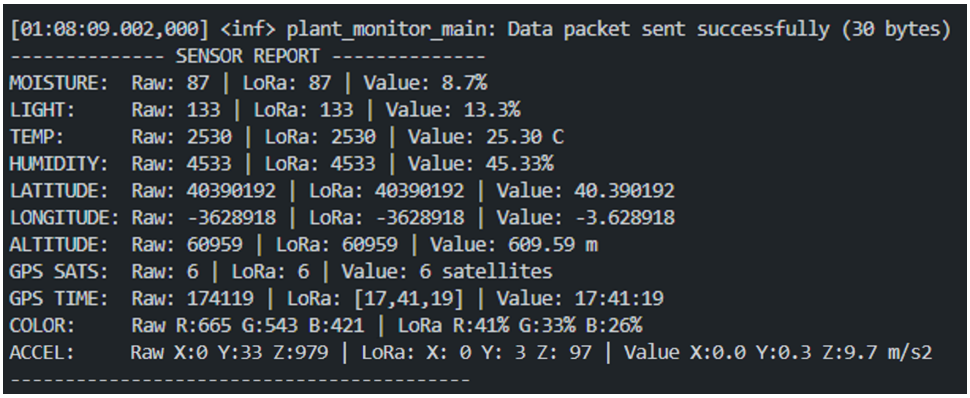

Figure 21: LoRaWAN Local Feedback Test

The confirmation message "*Data packet sent successfully (30 bytes)*" indicates that the internal buffer has been correctly assembled and handed over to the LoRaWAN stack for transmission to the ResIoT gateway.

5.3 ResIoT Test

The **ResIoT Test** focused on validating the LUA script responsible for decoding the incoming LoRaWAN payloads. As shown in Figure 22, the test involved simulating an uplink message with a predefined hexadecimal payload and passing it through the LUA decoder function.

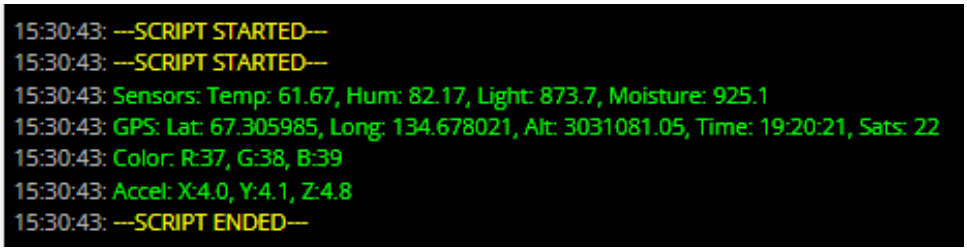

Figure 22: LUA Test

5.4 System Integration Test

The final validation of the project was conducted through a comprehensive **System Integration Test**, confirming the end-to-end functionality of the data pipeline, from physical sensing to cloud visualization. As shown in Figure 23, the **ResIoT Dashboard** successfully aggregates all telemetry streams into a cohesive monitoring station.

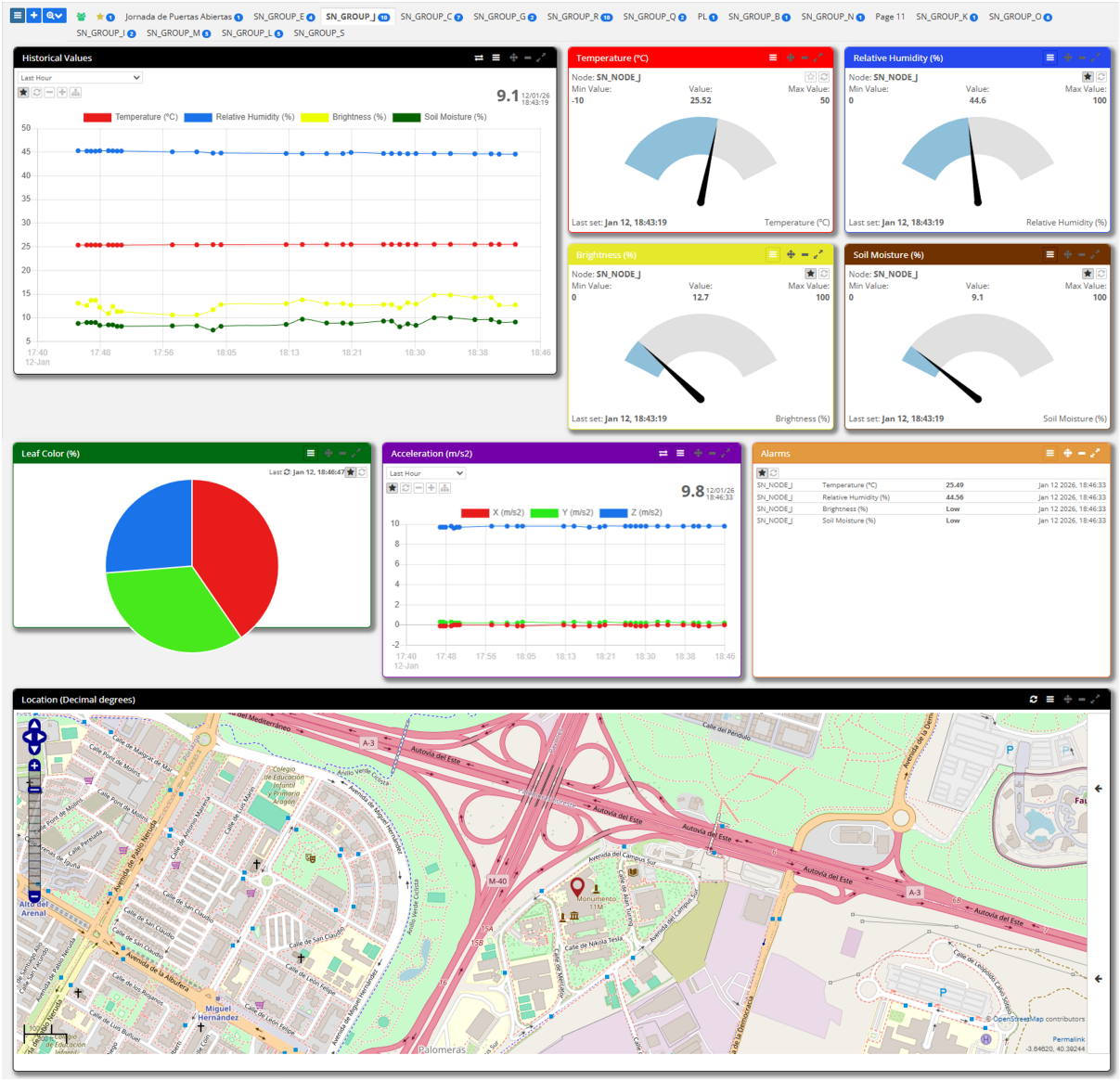

Figure 23: ResIoT Dashboard Result

5.5 Mobile Application

The ResIoT platform features a responsive web-based architecture that allows users to monitor the **SN_NODE_J** system through a mobile interface. As demonstrated in Figure 24, the **SN_GROUP_J** dashboard automatically adapts its layout to smaller screen sizes, ensuring that all telemetry data remains legible and interactive for remote monitoring.

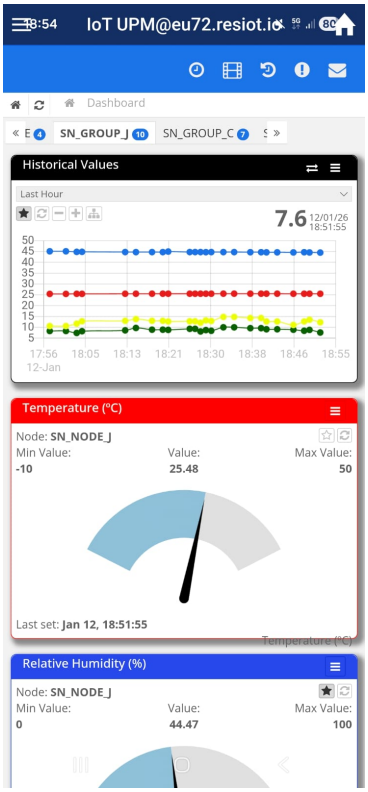

Figure 24: Mobile Application View

5.6 RGB Commands

The ResIoT platform facilitates bidirectional communication, allowing the user to send **downlink** commands to the **SN_NODE_J** device to control its integrated RGB LED. This process is initiated through the "Node Downlink" interface, where the specific **DevEUI** and the LoRaWAN gateway (`Building 8 - Roof`) are selected to route the message. As illustrated in Figure 25, the user inputs the desired color command (in this case, "Green") as the data payload to be transmitted over the **EU868** network.

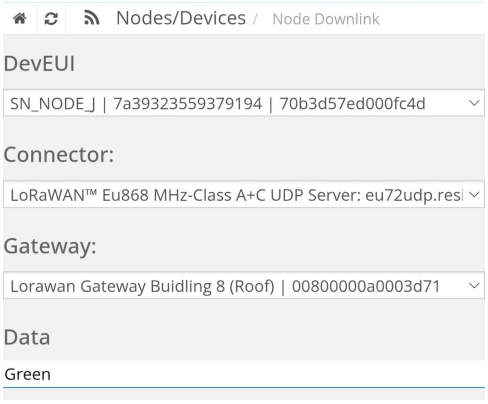

Figure 25: Command Sent

Once the command is issued, the platform provides a **Command Result** log (Figure 26) to confirm the execution of the request. This log details critical transmission metadata, including the **AppEUI**, the timestamp of the event, and the specific communication connector utilized.

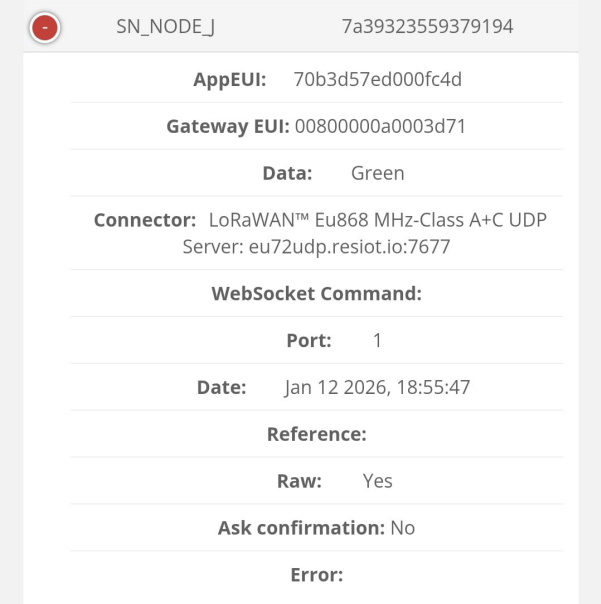

Figure 26: Command Result

The physical result of this automation is shown in Figure 27, where the LoRaWAN end-node receives the downlink packet and updates the state of its hardware. The integrated RGB LED on the breadboard illuminates in **green**, matching the issued command.

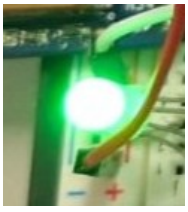

Figure 27: LED Result

5.7 CPP Check

Static code analysis was performed on the project's modules using CPPcheck with a GitHub Action. As illustrated in Figure 28, the analysis yielded a limited number of warnings. The majority of these notifications arise from missing Zephyr-specific libraries. This is a direct result of the GitHub Actions environment lacking a comprehensive Zephyr build context and the necessary toolchain or board support packages.

Furthermore, while CPPcheck identifies several unused functions, this is expected behavior given that the modules are designed as general libraries rather than exclusive components of this specific system. Ultimately, no critical defects were identified, and the existing warnings do not compromise the functional integrity of the codebase.

```
^
src/sensors_thread.c:17:0: information: Include file: <zephyr/sys/printk.h> not found. Please note: Cppcheck does not need standard library headers to
get proper results. [missingIncludeSystem]
#include <zephyr/sys/printk.h>
^
10/10 files checked 100% done
src/sensors/adc/adc.c:96:0: style: The function 'adc_read_normalized' is never used. [unusedFunction]
float adc_read_normalized(const struct adc_config *cfg)
^
src/sensors/i2c/accel.c:132:0: style: The function 'accel_convert_to_g' is never used. [unusedFunction]
void accel_convert_to_g(int16_t raw, uint8_t range, float *g_value) {
^
```

Figure 28: Cppcheck Result

6 Conclusions and Future Works

6.1 Conclusions

This project has successfully achieved the design and implementation of a complete IoT-based Plant Monitoring System using the STM32WL55JC microcontroller and the Zephyr RTOS. The developed system integrates multiple analog and digital sensors to acquire environmental, physical, and positional data, demonstrating robust multitasking through a multi-threaded architecture. By leveraging dedicated threads for sensors and GPS data, the system effectively manages concurrent data acquisition and LoRaWAN communication.

All mandatory system requirements were fulfilled, including periodic sensor acquisition every 60 seconds and visual feedback through a RGB LED. A critical milestone completed was the optimization of the communication LoRaWAN stack, transitioning to a 30-byte packed binary structure to maximize transmission efficiency. The software architecture utilized atomic variables to protect shared resources, ensuring system stability and preventing data corruption during thread context switches.

The integration with the ResIoT platform enabled a comprehensive data pipeline, from physical sensing to cloud visualization through an interactive dashboard. The implementation of OTAA activation and a custom LUA payload decoder ensured secure connectivity and the accurate transformation of raw hexadecimal data into human-readable metrics. Furthermore, the successful deployment of a downlink command handler allows for real-time remote interaction with the node's hardware.

6.2 Future Works

Although the system meets all specified requirements, several improvements and extensions could be considered in future iterations to enhance performance, accuracy, and scalability:

- **Expanded Sensing Capabilities:** Adding additional simple sensors, such as a CO2 probe or a pH sensor, to provide a more comprehensive view of the plant's physiological health.
- **Physical Protection:** Designing a weather-resistant enclosure for the STM32WL55JC and its breadboard components to allow for testing in actual outdoor garden environments.
- **Power Consumption Profile:** Implementing sleep modes between the 60-second transmission intervals to further extend battery life for long-term field deployment.
- **Adaptive Sampling:** Developing logic to adjust the transmission interval dynamically based on soil moisture levels or environmental criticalities (e.g., increasing frequency during drought conditions).
- **Edge Computing:** Integrating local data processing to detect leaf color anomalies or sudden orientation shifts (acceleration) locally on the MCU, sending "Alarm" flags to reduce unnecessary network traffic.
- **Advanced Visualization:** Expanding the system with predictive analytics to forecast irrigation needs based on historical temperature and soil moisture trends.

7 Bibliography

[1] "STM32WL55JC — Product - STMicroelectronics," Accessed: Dec. 9, 2025. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32wl55jc.html.

[2] "ResIOT® - LoRaWAN® Network Server and IoT Platform," ResIOT®, Accessed: Jan. 4, 2026. [Online]. Available: https://www.resiot.io/en/home/.

[3] A. Industries. "Photo Transistor Light Sensor," Accessed: Nov. 24, 2025. [Online]. Available: https://www.adafruit.com/product/2831.

[4] "SparkFun Soil Moisture Sensor - SparkFun Electronics," Accessed: Nov. 24, 2025. [Online]. Available: https://www.sparkfun.com/sparkfun-soil-moisture-sensor.html.

[5] A. Industries. "Adafruit Si7021 Temperature & Humidity Sensor Breakout Board," Accessed: Nov. 24, 2025. [Online]. Available: https://www.adafruit.com/product/3251.

[6] A. Industries. "RGB Color Sensor with IR filter and White LED - TCS34725," Accessed: Nov. 24, 2025. [Online]. Available: https://www.adafruit.com/product/1334.

[7] A. Industries. "Adafruit Triple-Axis Accelerometer - ±2/4/8g @ 14-bit - MMA8451," Accessed: Nov. 24, 2025. [Online]. Available: https://www.adafruit.com/product/2019.

[8] A. Industries. "Adafruit Ultimate GPS Breakout - 66 channel w/10 Hz updates," Accessed: Nov. 24, 2025. [Online]. Available: https://www.adafruit.com/product/746.

[9] "Zephyr API Documentation: Introduction," Accessed: Nov. 24, 2025. [Online]. Available: https://docs.zephyrproject.org/latest/doxygen/html/index.html.

[10] "Curso: Sensor networks — MOODLE UPM - OFICIALES 25-26," Accessed: Jan. 4, 2026. [Online]. Available: https://moodle.upm.es/titulaciones/oficiales/course/view.php?id=2211.

Appendix A - GitHub

A.1 GitHub Repository - Source Code

The complete source code of the Plant Monitoring System project is hosted on GitHub and can be accessed via the following link: `https://github.com/Estelamb/Sensor_Networks`.

A.2 GitHub Pages - Documentation

The project documentation can be accessed through the following link: `https://estelamb.github.io/Sensor_Networks/`.

A.3 GitHub Action - Workflow

Listing 14: GitHub Action workflow

```
name: Cppcheck and Doxygen

on:
  push:
    branches: [ main ]
  workflow_dispatch:

jobs:
  cppcheck:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Install cppcheck
        run: sudo apt-get update && sudo apt-get install -y cppcheck

      - name: Run cppcheck
        run: |
          if [ -d "plant_monitoring_system" ]; then
            cd plant_monitoring_system
          fi
          cppcheck --enable=all --inconclusive --std=c++17 -I include src

  doxygen:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Install Doxygen
        run: sudo apt-get update && sudo apt-get install -y doxygen graphviz

      - name: Generate documentation for plant_monitoring_system
        run: |
          cd plant_monitoring_system
          doxygen Doxyfile

      - name: Create docs directory if it doesn't exist
        run: |
          cd plant_monitoring_system
          mkdir -p docs/html

      - name: Verify HTML documentation output
        run: |
          ls -la ./plant_monitoring_system/docs/
          ls -la ./plant_monitoring_system/docs/html
```

```
      - name: Deploy to GitHub Pages
        uses: peaceiris/actions-gh-pages@v3
        with:
          github_token: ${{ secrets.GITHUB_TOKEN }}
          publish_dir: ./plant_monitoring_system/docs/html
```