Plant Monitoring System

Embedded Platforms and Communications for IoT

Alejandro Botas Bárcena
Estela Mora Barba

MIoT 2025-2026

Contents

List of Figures

List of Tables

List of Listings

Acronyms

ADC Analog-to-Digital Converter

API Application Programming Interface

ARM Advanced RISC Machine

COM PC Communication

CPU Central Processing Unit

DAC Digital-to-Analog Converter

DMA Direct Memory Access

E East

FIFO First In, First Out

GGA Global Positioning System Fix Data

GND Ground

GNGGA Global Navigation Satellite System Fix Data

GPIO General Purpose Input-Output

GPS Global Positioning System

HDOP Horizontal Dilution of Precision

I2C Inter-Integrated Circuit

IoT Internet of Things

ISR Interruption Service Routine

LED Light Emitting Diode

LoRa Long Range

LPWAN Low-Power Wide-Area Network

MCU Microcontroller Unit

N North

NMEA National Marine Electronics Association

PWM Pulse Width Modulation

RAM Random-Access Memory

RGB Red, Green and Blue

RGBC Red, Green, Blue and Clear

RH Relative Humidity

RTOS Real-Time Operating System

RX Reception

S South

SCL Serial Clock Line

SDA Serial Data

SI International System of Units

SPI Serial Peripheral Interface

SRAM Static Random-Access Memory

TX Transmission

UART Universal Asynchronous Receiver-Transmitter

USART Universal Synchronous Asynchronous Receiver-Transmitter

USB Universal Serial Bus

UTC Coordinated Universal Time

W West

1 Overview and Introduction

1.1 Document Overview

This document defines the technical specifications, development requirements, hardware architecture, software components, and operational modes of the Internet of Things (IoT)-based Plant Monitoring System to be implemented during the course *Embedded Platforms and Communications for IoT*. The objective of this specification is to establish a clear and comprehensive reference for the design, implementation, verification, and assessment of the final embedded system.

1.2 Project Introduction

The purpose of the final project is to design and implement a fully functional IoT platform capable of monitoring the environmental conditions and physiological state of a plant throughout its lifecycle. Such monitoring is essential for applications such as greenhouse automation, precision agriculture, plant health diagnostics, and traceability in plant transportation and storage.

The system must continuously acquire, process, and report multiple physical variables, including temperature, relative humidity, ambient light intensity, soil moisture, and colour characteristics of a plant leaf. Additional inertial data are collected through an accelerometer to detect events such as impacts, falls, tilting, or abnormal movements. The global position of the plant is obtained via a Global Positioning System (GPS) module, ensuring timestamped logging of all monitored parameters.

This project integrates both hardware and software development activities. Students must interface several digital and analog sensors, configure low-level peripherals (Inter-Integrated Circuit (I2C), Universal Asynchronous Receiver-Transmitter (UART), Analog-to-Digital Converter (ADC), General Purpose Input-Output (GPIO), Pulse Width Modulation (PWM)), and implement multitasking using Zephyr Real-Time Operating System (RTOS).

The final embedded application must operate in distinct modes, manage periodic measurements, compute statistical parameters, handle event-based notifications, and provide visual feedback via a Red, Green and Blue (RGB) Light Emitting Diode (LED).

1.3 Summary of the Work Done

This section provides a consolidated overview of the work completed throughout the development of the IoT-based Plant Monitoring System. The tasks performed encompass the full engineering workflow, including requirement analysis, hardware integration, software design, system implementation, testing, and verification.

1.3.1 Requirements Analysis

The project began with an in-depth study of the provided technical specifications and sensor documentation. All functional, hardware, and timing requirements were reviewed to establish a clear design baseline. This included understanding the sensing ranges, communication interfaces, and operating constraints imposed by the STM32WL55JC microcontroller and Zephyr RTOS.

1.3.2 Hardware Integration

The hardware development stage consisted of identifying, wiring, and validating all sensor interfaces:

- STM32WL55JC microcontroller as the central processing unit.
- Integration of the Si7021 temperature and humidity sensor using the I2C bus.
- Connection and calibration of the HW5P-1 phototransistor for ambient light measurement.
- Analog acquisition and scaling of the SEN-13322 soil moisture probe.
- Digital configuration of the TCS34725 colour sensor over I2C.
- Setup of the MMA8451Q accelerometer for multi-axis measurements.
- Interfacing and configuring the Adafruit GPS module via UART.

- Implementation of a RGB LED driver using PWM emulation for status indication.

1.3.3 Software Development

Software implementation was carried out using Zephyr RTOS, structured with a multitasking architecture. Key activities included:

- Configuration of device tree overlays and config options for all peripherals.
- Development of sensor drivers and low-level routines for ADC, I2C, UART, and GPIO.
- Design of individual threads for periodic sampling, data processing, GPS acquisition, and mode management.
- Implementation of Test Mode, Normal Mode, and optional Advanced Mode according to system requirements.
- Integration of statistical processing to compute hourly mean, minimum, and maximum values.
- Implementation of colour-based alert mechanisms based on out-of-range sensor values.

Zephyr's logging and shell utilities were used extensively for debugging and validation.

1.3.4 System Testing and Validation

The complete system was evaluated across all operational modes:

- Verification of measurement accuracy and stability under Test Mode.
- Long-term monitoring and statistical computation under Normal Mode.
- Correct operation of the mode-switching mechanism through the push button.
- Validation of RGB LED behaviour for both colour detection and alert signalling.
- GPS time synchronization and conversion to local time for timestamp generation.
- Stress testing of the application to identify stack usage limits and race conditions.

All mandatory functionalities were confirmed to meet the specifications, with optional enhancements explored where possible.

1.3.5 Final Deliverables

The completed work includes:

- A functional embedded system integrating all sensors and the STM32WL55JC Microcontroller Unit (MCU).
- Clean, documented source code developed under Zephyr RTOS.
- A complete technical report detailing the system design, implementation, and results.
- Final project documentation and code submitted according to course requirements.

Overall, the work conducted demonstrates a full-cycle embedded systems development process, covering hardware, software, real-time processing, testing, and documentation.

2 Specifications

2.1 Specifications required

2.1.1 Hardware

The IoT system is based on the STM32WL55JC microcontroller. Several sensors are required to monitor plant-related environmental and physical parameters. Table 1 summarizes the recommended hardware components, electrical interfaces, and approximate costs.

Table 1: Summary of Suggested Hardware for the IoT System

Parameter	**Sensor / Module**	**Interface**
MCU Board	STM32WL55JC	–
Status Indicator	RGB LED	Digital
Ambient Light	HW5P-1 Phototransistor	Analog
Soil Moisture	SEN-13322	Analog
Temperature / Humidity	Si7021	I2C
Leaf Colour	TCS34725	I2C
Accelerometer	MMA8451Q	I2C
Global Location	Adafruit GPS	UART

2.1.2 Software

The software stack required for the development of the IoT system is summarized in Table 2.

Table 2: Software Requirements

Software Tool	**Description**
Zephyr RTOS	RTOS for Advanced RISC Machine (ARM) Cortex-M devices, used for all system tasks and drivers.
Visual Studio Code	Primary development environment with C/C++, CMake, and Cortex-Debug extensions.
Git / TortoiseGit / GitHub	Version control for project source code.
TeraTerm / PuTTY	Serial terminal emulator for debugging and mode output.

2.1.3 System Requirements

The IoT system must measure environmental and physical parameters and operate in three modes: Test, Normal, and Advanced (optional). Table 3-Table 4 present the system requirements.

Table 3: Sensing Requirements

ID	**Requirement**	**Specification**
SR1	Temperature	Range: -10°C to 50°C. Resolution: 0.1°C
SR2	Relative Humidity	Range: 25%-75% RH. Resolution: 0.1%
SR3	Ambient Light	0-100%. Resolution: 0.1%
SR4	Soil Moisture	0-100%. Resolution: 0.1%
SR5	Leaf Colour	Clear, red, green, blue values
SR6	GPS Location	Coordinates + Coordinated Universal Time (UTC) time
SR7	Acceleration	X, Y, Z axes. Formatted output
SR8	Robustness	System must be stable and robust
SR9	Thread Management	Tasks must be partitioned using multitasking

Table 4: Operating Modes Requirements

Mode	ID	Description
Test Mode	TM1	Verify sensor connections and operation.
	TM2	Sampling period: 2 seconds.
	TM3	Send all measurements every 2 seconds via Universal Serial Bus (USB) virtual PC Communication (COM).
	TM4	RGB LED indicates dominant leaf colour.
	TM5	Blue LED (LED1) must remain ON.
Normal Mode	NM1	Sampling period: 30 seconds.
	NM2	Send all measurements every 30 seconds.
	NM3	Compute hourly mean, max, min for temperature, humidity, light, moisture.
	NM4	Compute hourly dominant colour (frequency-based).
	NM5	Compute hourly max and min accelerometer values.
	NM6	Send GPS location + local time every 30 seconds.
	NM7	Color-coded RGB alert when limits exceeded.
	NM8	Green LED (LED2) must remain ON.
Advanced Mode (Optional)	AM1	Requirements provided during validation stage.
	AM2	Red LED (LED3) must remain ON.

2.2 Additional specifications implemented

In addition to the mandatory requirements defined in the project specifications, several extended functionalities and robustness mechanisms were implemented to improve system reliability, diagnostic capability, maintainability, and user feedback. These additional specifications are summarized in Table 5.

Table 5: Additional Specifications Implemented

ID	Additional Requirement	Description
AS1	**Sensor configuration**	All sensors are configured during system initialization, including adjustable internal parameters such as the colour sensor gain and integration time.
AS2	**Fail-safe initialization**	If any sensor fails to initialize correctly, the entire application halts execution and enters a safe error state.
AS3	**Sensor reconnection**	If a sensor becomes disconnected, a fault condition is reported. If the sensor is later reconnected, the system automatically reinitializes it and resumes normal measurement without requiring a reset.
AS4	**Alarm sequencing**	When operating in NORMAL mode, all active alarms are cycled through sequentially using the RGB LED, with a display cadence of **0.5 seconds per alarm**.
AS5	**Advanced Mode behaviour**	In ADVANCED mode, the system extends the behaviour of the NORMAL mode by reproducing on the RGB LED the exact colour intensity measured by the TCS34725 sensor. Since the existing LED pins cannot be modified, PWM operation is emulated in software to match LED brightness with the RGB sensor readings proportionally. All remaining system behaviour matches NORMAL mode.
AS6	**Stack usage measurement**	The system includes automatic instrumentation to evaluate thread stack consumption at runtime.
AS7	**Verification tools**	External analysis tools were used to inspect the system.
AS8	**Documentation**	All modules, drivers, data-processing routines, and operating modes are fully documented.

3 Hardware Analysis

3.1 Block diagram

The block diagram shown in Figure 1 provides an overview of the complete hardware architecture. It illustrates how the STM32WL55JC microcontroller interacts with the different sensors and output devices integrated into the system. Each peripheral is connected through the appropriate interface, such as analog inputs, I2C buses, UART communication lines, and GPIO pins, allowing the microcontroller to gather environmental data, process it, and generate feedback.

This diagram serves as a high-level representation of the system's structure, highlighting the flow of information between components and the role of the microcontroller as the central control unit.

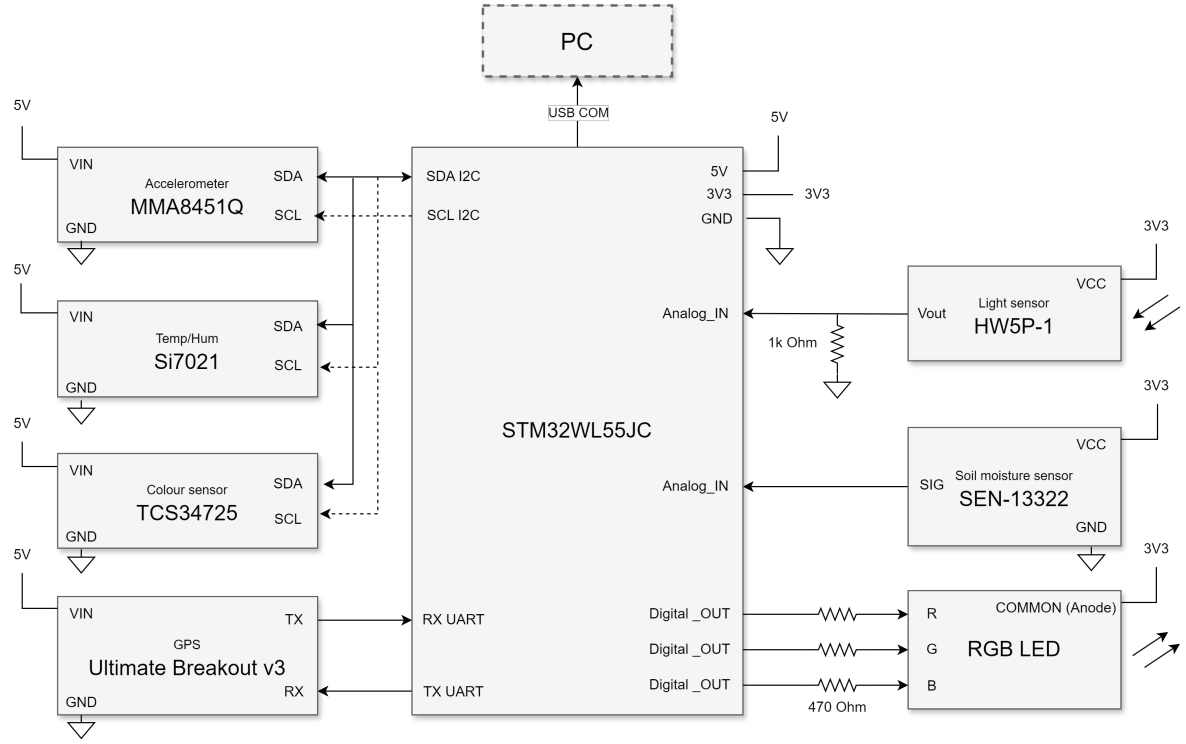

Figure 1: Block Diagram of the Hardware System

The microcontroller employs several of its internal peripherals to interface with the different sensors and modules in the system. One of the available ADC channels is used to read the analog outputs of the soil moisture sensor and the ambient light phototransistor. The I2C2 bus is shared by the temperature and humidity sensor (Si7021), the colour sensor (TCS34725), and the accelerometer (MMA8451Q). A UART interface is dedicated to the GPS module, enabling continuous reception of positioning data.

In addition, three GPIO pins are configured as digital outputs to drive the RGB LED through current-limiting resistors. The system also uses the 3.3V and 5V power rails provided by the board, as well as the ground reference shared by all components. Together, these resources form a compact and energy-efficient hardware configuration that leverages the STM32WL55JC's ADCs, GPIOs, communication peripherals, and power distribution capabilities.

3.2 Interfaces of the system

Table 6 details all electrical interfaces used in the system. Each sensor or module is mapped to the corresponding STM32WL55JC pins, specifying power connections, communication buses, and signal types. The design integrates a mix of digital and analog interfaces, including I2C for multi-sensor communication, UART for GPS data, and ADC channels for analog measurements such as soil moisture and ambient light. In addition, several GPIO pins are used for driving the RGB LED.

Table 6: System Connections

Parameter	**Sensor**	**Pin Description**	**Sensor PIN name**	**STM32WL55JC Connector**	**STM32WL55JC PIN name**	**STM32WL55JC Function**
LED RGB	RGB LED + Resistors (470 Ohm)	Common (Anode)	C	CN6 3V3	3V3	3.3V
		Red + 470 Ohm	R	CN5 PA_6	D12	R
		Green + 470 Ohm	G	CN5 PA_7	D11	G
		Blue + 470 Ohm	B	CN5 PA_9	D9	B
Ambient Light	HW5P-1 Phototransistor + Resistor (1k Ohm)	VCC 3.3V	VCC	CN6 3V3	3V3	3.3V
		Vout	Vout	CN8 PB_1	ADC 1/5	Analog input
		Ground	Ground (GND)	CN6 GND	GND	Ground
Soil Moisture	SEN-13322	VCC 3.3V	VCC	CN6 3V3	3V3	3.3V
		Vout	SIG	CN8 PB_13	ADC 1/0	Analog input
		Ground	GND	CN6 GND	GND	Ground
Temperature and Humidity	Si7021	VCC 5V	VIN	CN6 5V	5V	5V
		Ground	GND	CN6 GND	GND	Ground
		I2C Serial Clock Line (SCL)	SCL	CN5 PA_12	I2C2_SCL	I2C SCL
		I2C Serial Data (SDA)	SDA	CN5 PA_11	I2C2_SDA	I2C SDA
		Output 3.3V	3Vo	–	–	–
Leaf Colour	TCS34725	VIN 5V	VIN	CN6 5V	5V	5V
		Ground	GND	CN6 GND	GND	Ground
		I2C SCL	SCL	CN5 PA_12	I2C2_SCL	I2C SCL
		I2C SDA	SDA	CN5 PA_11	I2C2_SDA	I2C SDA
		Output 3.3V	3V3	–	–	–
		Interrupt out	INT	–	–	–
		LED on/off	LED	–	–	–
Accelerometer	MMA8451Q	VIN 5V	VIN	CN6 5V	5V	5V
		Ground	GND	CN6 GND	GND	Ground
		I2C SCL	SCL	CN5 PA_12	I2C2_SCL	I2C SCL
		I2C SDA	SDA	CN5 PA_11	I2C2_SDA	I2C SDA
		Output 3.3V reg	3Vo	–	–	–
		Inertial Interrupt 1, Output pin	I1	–	–	–
		Inertial Interrupt 2, Output pin	I2	–	–	–
		I2C least significant bit of the device I2C address	A	–	–	–
GPS	Adafruit Ultimate GPS Breakout v3	VIN 5V	VIN	CN6 5V	5V	5V
		Ground	GND	CN6 GND	GND	Ground
		Serial Transmission (TX)	TX	CN9 PB_7	UART1_Reception (RX)	UART RX
		Serial RX	RX	CN9 PB_6	UART1_TX	UART TX
		Output 3.3V reg	3.3V	–	–	–
		Enable	EN	–	–	–
		Fix output	FIX	–	–	–
		Vbackup (battery)	VBAT	–	–	–
		Pulse Per Second output	PPS	–	–	–

3.3 Communication Interfaces used in the system

The system relies on several hardware communication interfaces that allow the STM32WL55JC microcontroller to exchange data efficiently with the different sensors and modules. Each interface is selected based on the nature of the signal (analog or digital), the required data rate, and the number of devices connected.

3.3.1 Analog-to-Digital Converter (ADC)

The STM32WL55JC includes a 12-bit ADC capable of converting analog voltages into digital values. This interface is used for sensors that provide an output voltage proportional to a physical quantity, such as:

- HW5P-1 phototransistor (ambient light)
- SEN-13322 soil moisture sensor

The ADC samples the voltage at the input pin and converts it into a numerical value between 0 and 4095 (12 bits), enabling the microcontroller to process continuous physical signals using digital logic.

3.3.2 I2C Bus

The I2C (Inter-Integrated Circuit) bus is a two-wire digital communication interface consisting of:

- SCL: clock line
- SDA: data line

Multiple sensors can share the same bus because each device has a unique address. In this system, I2C2 is used, and three devices share it:

- Si7021 temperature and humidity sensor
- TCS34725 colour sensor
- MMA8451Q accelerometer

I2C allows simple wiring, energy-efficient transmission, and reliable short-distance communication, making it ideal for embedded sensor networks.

3.3.3 UART Interface

The Universal Asynchronous Receiver/Transmitter (UART) is a serial communication interface used for asynchronous data transfer. It uses two lines:

- TX: microcontroller transmits data
- RX: microcontroller receives data

The Adafruit Ultimate GPS Breakout v3 communicates via a dedicated UART port, continuously streaming National Marine Electronics Association (NMEA) sentences that include position, altitude, speed, and time. UART is preferred here because it supports continuous high-latency streams and long-format messages without requiring a synchronized clock signal.

3.3.4 GPIO Digital Pins

General-Purpose Input/Output (GPIO) pins are used for simple digital control or sensing. In this project, several GPIOs are configured as outputs to drive the RGB LED. Each colour channel (red, green, and blue) is controlled by switching the corresponding GPIO pin on or off.

GPIOs allow:

- Driving LEDs or actuators
- Reading simple digital sensors
- Triggering interrupts

Their flexibility and direct control make them suitable for simple digital signals.

3.3.5 Power Interfaces

The system also uses fixed-voltage power rails:

- **3.3V**: used by analog sensors and logic inputs (e.g., phototransistor, soil sensor)
- **5V**: used by some breakout boards that include internal regulators (e.g., Si7021, TCS34725, MMA8451Q, GPS)
- **GND**: common electrical reference shared by all modules

A shared ground is essential for stable communication because all signal voltages must be referenced to the same electrical level. These interfaces together form an efficient and compact architecture that ensures reliable data acquisition and control across all hardware modules.

3.4 Hardware devices

3.4.1 STM32WL55JC microcontroller

The STM32WL55JC[1] is an ultra-low-power microcontroller that integrates both a processing unit and a long-range sub-GHz radio in a single chip. It combines an ARM Cortex-M4 core for the main application and an ARM Cortex-M0+ core for security and background tasks, providing efficient performance with very low energy consumption. The device includes 256KB of Flash, 64KB of Static Random-Access Memory (SRAM), and a wide set of protection features to ensure firmware integrity.

Its built-in radio supports several Low-Power Wide-Area Network (LPWAN) modulations, including Long Range (LoRa), enabling long-distance communication. The microcontroller also offers a rich collection of peripherals—such as 12-bit ADC/Digital-to-Analog Converter (DAC), multiple timers, Direct Memory Access (DMA) controllers, and interfaces like UART, I2C, and Serial Peripheral Interface (SPI), making it highly adaptable to sensor-based and low-power embedded applications.

3.4.2 RGB LED and 470 Ohm resistors

The system includes a common-anode RGB LED used to provide visual feedback during operation. This type of LED shares a single positive terminal connected to the 3.3,V rail, while each color channel (red, green, and blue) is controlled individually through the microcontroller. The STM32WL55JC drives the three channels using pins PA_6, PA_7, and PA_9, which can be toggled to generate different brightness levels and color combinations.

Each LED channel is connected in series with a 470 Ohm resistor to ensure proper current limiting and protect both the LED and the microcontroller outputs. This simple circuit allows the system to display a wide range of colors, enabling intuitive status indication, such as alerts or measurement feedback.

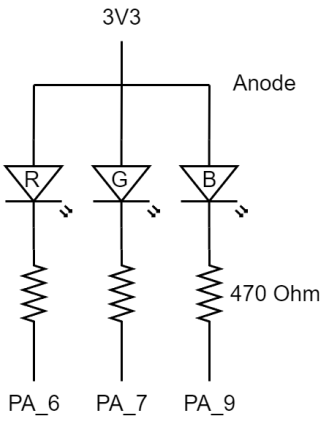

Figure 2: RGB LED circuit diagram

3.4.3 HW5P-1 Phototransistor and 1k Ohm resistor

The system uses a HW5P-1 phototransistor [2] to measure ambient light intensity. The phototransistor is connected in a simple voltage-divider configuration with a 1k Ohm resistor, converting the light-dependent current into a measurable voltage at the junction between the two components. This analog voltage is fed directly into the STM32WL55JC's ADC1/5 channel, allowing the microcontroller to quantify the light level.

By sampling the ADC input, the system can monitor changes in illumination and use this information for environmental sensing or automatic control tasks. The 3.3V supply powers the phototransistor, while a shared ground ensures proper reference for the ADC measurements.

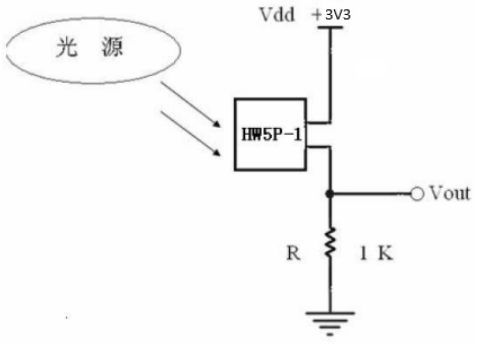

Figure 3: Phototransistor circuit diagram

To express the ambient light as a percentage, the ADC reading is first converted to a voltage using the reference voltage V_{ref} of 3.3V:

$$V_{\text{phototransistor}} = \frac{\text{ADC\_value}}{\text{ADC\_max}} \cdot V_{\text{ref}}$$

Then, this voltage is normalized to a percentage of the maximum measurable light:

$$\text{Light\%} = \frac{V_{\text{phototransistor}}}{V_{\text{ref}}} \cdot 100 = \frac{\text{ADC\_value}}{\text{ADC\_max}} \cdot 100$$

This approach ensures that the ambient light intensity is represented in a standardized form from 0% (dark) to 100% (maximum brightness measurable by the sensor).

The measured light percentage is used by the system to evaluate illumination conditions in the surrounding environment.

3.4.4 SEN-13322 Soil Moisture Sensor

The system uses a SEN-13322 soil moisture sensor[3] to monitor the water content of the soil. This sensor outputs an analog voltage that varies proportionally with the soil's moisture level. The voltage is measured by the STM32WL55JC using ADC1/0 channel, allowing the microcontroller to quantify the moisture.

The sensor is powered by the 3.3V supply from the board, with a common ground shared with the microcontroller to ensure accurate ADC measurements.

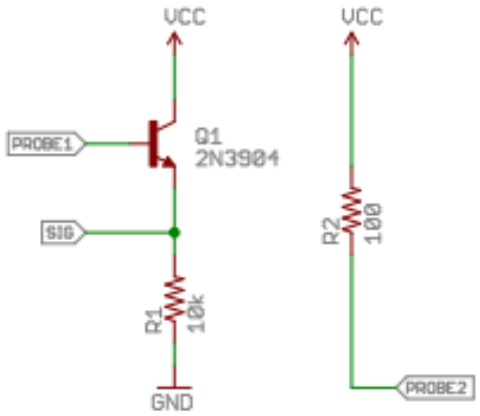

Figure 4: Soil moisture sensor circuit diagram

A direct reading of the ADC value can be converted into a soil moisture percentage using a similar normalization method as for the phototransistor:

$$\text{Moisture\%} = \frac{V_{\text{soil\_sensor}}}{V_{\text{ref}}} \cdot 100 = \frac{\text{ADC\_value}}{\text{ADC\_max}} \cdot 100$$

This provides a simple and effective way to represent soil moisture from 0% (completely dry) to 100% (fully saturated).

By continuously monitoring this value, the system can perform environmental sensing, trigger alerts, or control irrigation mechanisms in an automated manner.

3.4.5 Si7021 Temperature and Humidity Sensor

The system integrates a Si7021 digital temperature and humidity sensor[4] to monitor environmental conditions. This sensor communicates with the STM32WL55JC microcontroller via the I2C2 bus, using pins PA_12 (SCL) and PA_11 (SDA). The sensor is powered by the 5V supply from the board, while a shared ground ensures reliable communication and stable operation.

The Si7021 provides fully digital readings for both temperature and relative humidity, eliminating the need for additional signal conditioning or ADC conversion.

Relative Humidity (RH) is the amount of water vapor present in the air expressed as a percentage of the maximum humidity the air can hold at a given temperature. Mathematically, it is expressed as:

$$\text{RH } (\%) = \frac{P_{\text{humidity}}}{P_{\text{max\_humidity}}(T)} \times 100$$

The microcontroller can query the sensor at regular intervals to obtain accurate temperature in degrees Celsius and relative humidity in percentage. These measurements can then be used for environmental monitoring, data logging, or as input for control algorithms, such as adjusting irrigation based on humidity levels.

3.4.6 TCS34725 Colour Sensor

The system uses a TCS34725 digital colour sensor[5] to measure the color and brightness of objects or ambient light. This sensor communicates with the STM32WL55JC microcontroller via the I2C2 bus, using pins PA_12 (SCL) and PA_11 (SDA). The sensor is powered by the 5V supply from the board, with a shared ground for proper signal reference.

The TCS34725 integrates an array of photodiodes with color-specific filters (red, green, and blue) to detect the intensity of each primary color. Additionally, it includes a clear photodiode that measures the total light intensity without any color filtering. This *clear* channel allows the microcontroller to

compensate for variations in ambient light and normalize the color measurements, improving accuracy under different lighting conditions.

The sensor provides digital output values for each channel (red, green, blue, and clear), which can be read directly by the microcontroller. To calculate the relative intensity of each color, the readings can be normalized against the clear channel:

$$\text{Color\_ratio} = \frac{\text{Color\_value}}{\text{Clear\_value}}$$

This ratio provides a normalized measurement of the color composition independent of the overall light intensity.

The microcontroller can use this information for environmental monitoring, assessing leaf color for plant health, or other applications requiring color detection.

3.4.7 MMA8451Q Accelerometer

The system includes an MMA8451Q 3-axis digital accelerometer[6] to measure linear acceleration along three orthogonal axes (X, Y, and Z). The sensor communicates with the STM32WL55JC microcontroller via the I2C2 bus using pins PA_12 (SCL) and PA_11 (SDA), and is powered by the 5V supply from the board with a common ground reference.

The MMA8451Q outputs digital values corresponding to the acceleration experienced along each axis, which includes static acceleration due to gravity. By convention, the Z-axis measures the acceleration in the vertical direction, while X and Y correspond to horizontal directions.

The raw digital readings from the sensor can be converted into acceleration in units of gravitational acceleration (g), using the sensor's sensitivity parameter (S_{range}), which depends on the configured full-scale range (e.g., ±2g, ±4g, or ±8g):

$$a_{\text{axis}}\,[g] = \frac{\text{Raw\_value}}{2^{12-1}} \cdot S_{\text{range}}$$

Raw_value is the 12-bit signed output from the accelerometer, and $2^{12-1} = 2048$ accounts for the 12-bit resolution with signed values. To convert the acceleration into International System of Units (SI) units (m/s^2), the following relation is used:

$$a_{\text{axis}}\,[\text{m/s}^2] = a_{\text{axis}}\,[g] \cdot g_0$$

where $g_0 = 9.80665\ \text{m/s}^2$ is the standard acceleration due to gravity. This conversion allows the microcontroller to quantify acceleration in physical units, providing meaningful data for motion detection, tilt sensing, or vibration monitoring.

By continuously reading the accelerometer, the system can track orientation changes, detect movement events, and combine the data with other sensors for environmental and behavioral monitoring applications.

3.4.8 Adafruit Ultimate GPS Breakout v3

The system integrates an Adafruit Ultimate GPS Breakout v3 module[7] to obtain accurate geolocation and time information. The GPS communicates with the STM32WL55JC microcontroller via a dedicated UART interface, using pins PB_6 (TX) and PB_7 (RX). The module is powered by the 5V supply from the board, with a shared ground reference.

The GPS module outputs position and time data in the standard NMEA sentence format. Key information includes:

- **Latitude and Longitude:** Provided in degrees and minutes (DDMM.MMMM for latitude and DDDMM.MMMM for longitude) along with a directional indicator: 'N' or 'S' for latitude, and 'E' or 'W' for longitude.

Latitude is referenced to the **Equator** (0°), which divides the Earth into the Northern and Southern hemispheres. Thus, 'N' indicates a position north of the Equator, while 'S' indicates a position south of it.

Longitude is referenced to the **Prime Meridian** (0°), also known as the **Greenwich Meridian**, which separates the Eastern and Western hemispheres. Positions east of Greenwich are marked with 'E', and those to the west with 'W'.

These values can be converted to decimal degrees using:

$$\text{Decimal Degrees} = \text{Degrees} + \frac{\text{Minutes}}{60}$$

For coordinates in the Southern or Western hemispheres, the decimal degrees are taken as negative.

Decimal degrees are used because they provide a **continuous numerical representation** of geographic coordinates, which simplifies mathematical operations such as distance calculations, interpolation, mapping transformations, and data storage. This format is easier for microcontrollers and software libraries to process compared to the degrees-minutes format used in raw NMEA sentences.

- **Altitude:** Measured in meters above mean sea level.
- **UTC Time and Date:** Provided as Coordinated Universal Time (hhmmss.ss for time and ddmmyy for date). UTC is referenced to the **Prime Meridian in Greenwich**, meaning it is the global baseline from which all time zones are defined.

  The system is located in Spain (peninsular), where local time is typically:

$$\text{Local Time} = \text{UTC} + 1\,\text{hour}$$

By parsing the NMEA sentences, the microcontroller can extract and store accurate position coordinates, altitude, and UTC-based timestamps. Continuous reception ensures that the system always has up-to-date location and timing information for real-time applications, enabling georeferenced sensor measurements and time-stamped environmental monitoring.

4 Software Organization

4.1 Description of the Global Software Architecture

The Plant Monitoring System is a multi-threaded embedded application developed using the Zephyr RTOS. It integrates multiple sensors and peripherals, manages several execution threads, uses atomic shared structures for inter-thread communication, and supports three distinct operating modes that govern its behaviour.

This section provides a unified description of system operation, the behaviour of each mode, the internal synchronization mechanisms, and the shared peripheral configuration structures.

4.1.1 Detailed System Behaviour per Operating Mode

The system operates according to the value of the `system_mode_t` enumeration, which cycles through three states: **TEST_MODE**, **NORMAL_MODE**, and **ADVANCED_MODE**. The user button triggers the transitions.

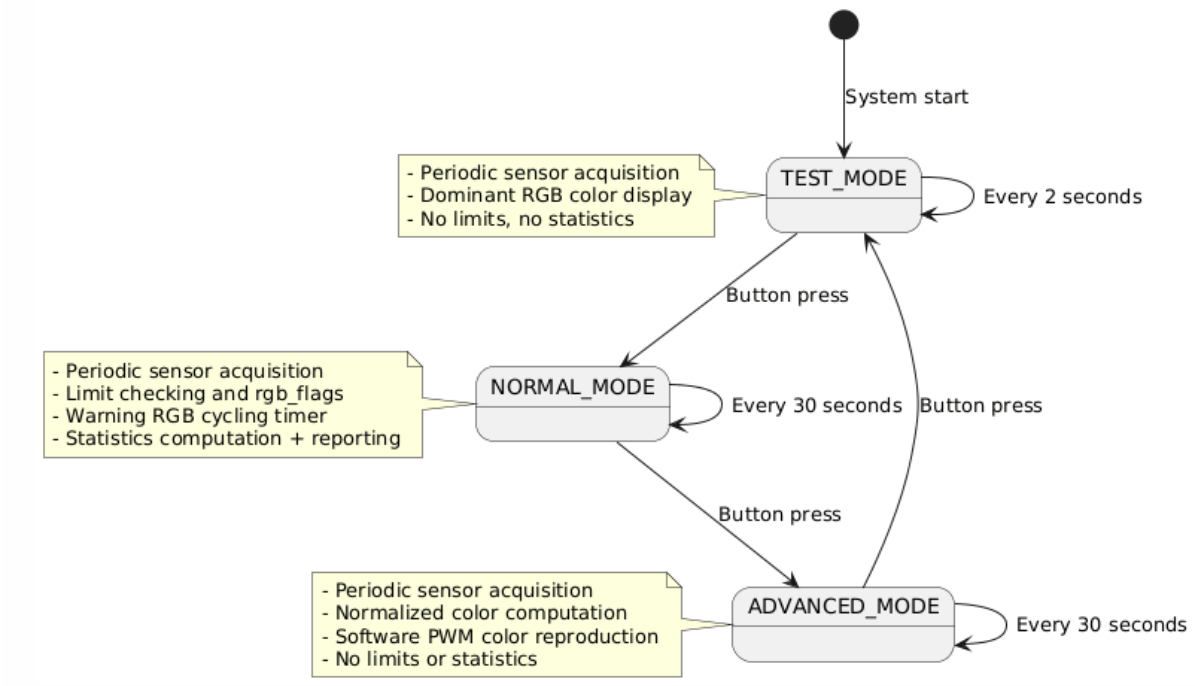

Figure 5: System behaviour

In **TEST_MODE**, the system performs simple acquisition and visualisation tasks:

1. The periodic measurement timer (`main_timer`) is started with the test-mode rate (2 seconds).
2. The sensor and GPS threads are triggered to acquire new samples.
3. The main thread waits for:
   - `main_sensors_sem`: Signals completion of environmental sensor acquisition,
   - `main_gps_sem`: Signals completion of GPS sampling.
4. The main thread reads the shared atomic measurement structure and determines the dominant colour measured by the RGB sensor.
5. The RGB LED is set to the dominant raw channel (red, green, or blue).
6. The measurements are displayed via the serial console.
7. At each timer expiration, the `main_sem` semaphore is released to repeat the mode.

8. No limit checking, no statistics, and no warning indicators are active.

TEST_MODE is primarily a hardware validation mode to verify that all sensors are functional.

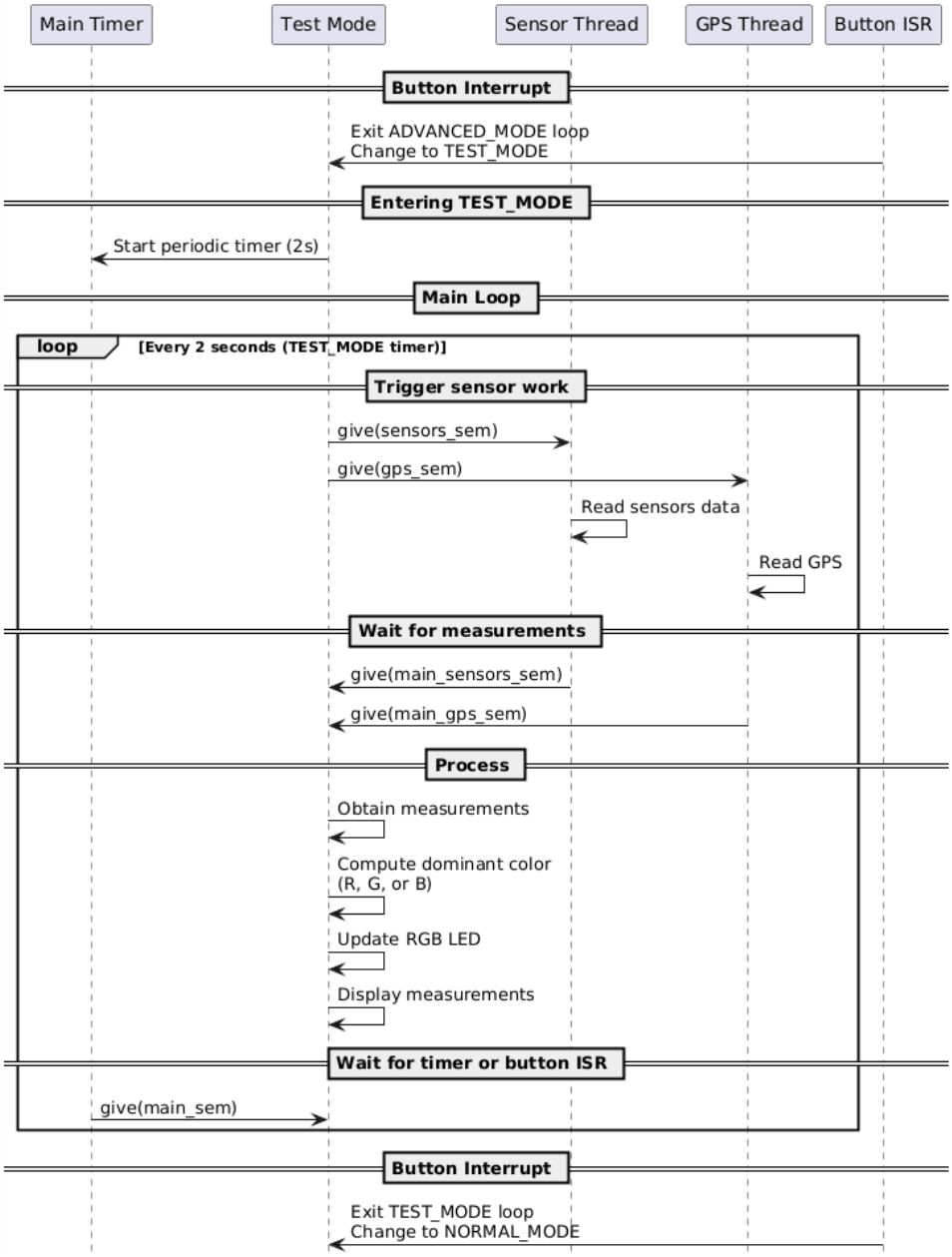

Figure 6: Test mode behaviour

In **NORMAL_MODE**, the system executes the full environmental monitoring workflow:

1. The measurement timer is started with the normal-mode sampling rate (30 seconds).
2. Sensor and GPS threads acquire data and raise their semaphores.
3. The main thread retrieves all sensor values from the atomic shared structure.
4. Limit checking is performed for temperature, humidity, moisture, brightness, and acceleration parameters.
5. Violations raise bits of the atomic `rgb_flags` field, allowing multiple alarms simultaneously.
6. A dedicated RGB warning timer is active: at each tick it reads the `rgb_flags` and updates the RGB LED with colour-coded warnings.

7. The system maintains statistical data, which is shown every hour:
    - running minimum and maximum,
    - running mean values,
    - colour frequency counts.
8. At each timer expiration, the `main_sem` semaphore is released to repeat the mode.

NORMAL_MODE is the complete monitoring and alert mode.

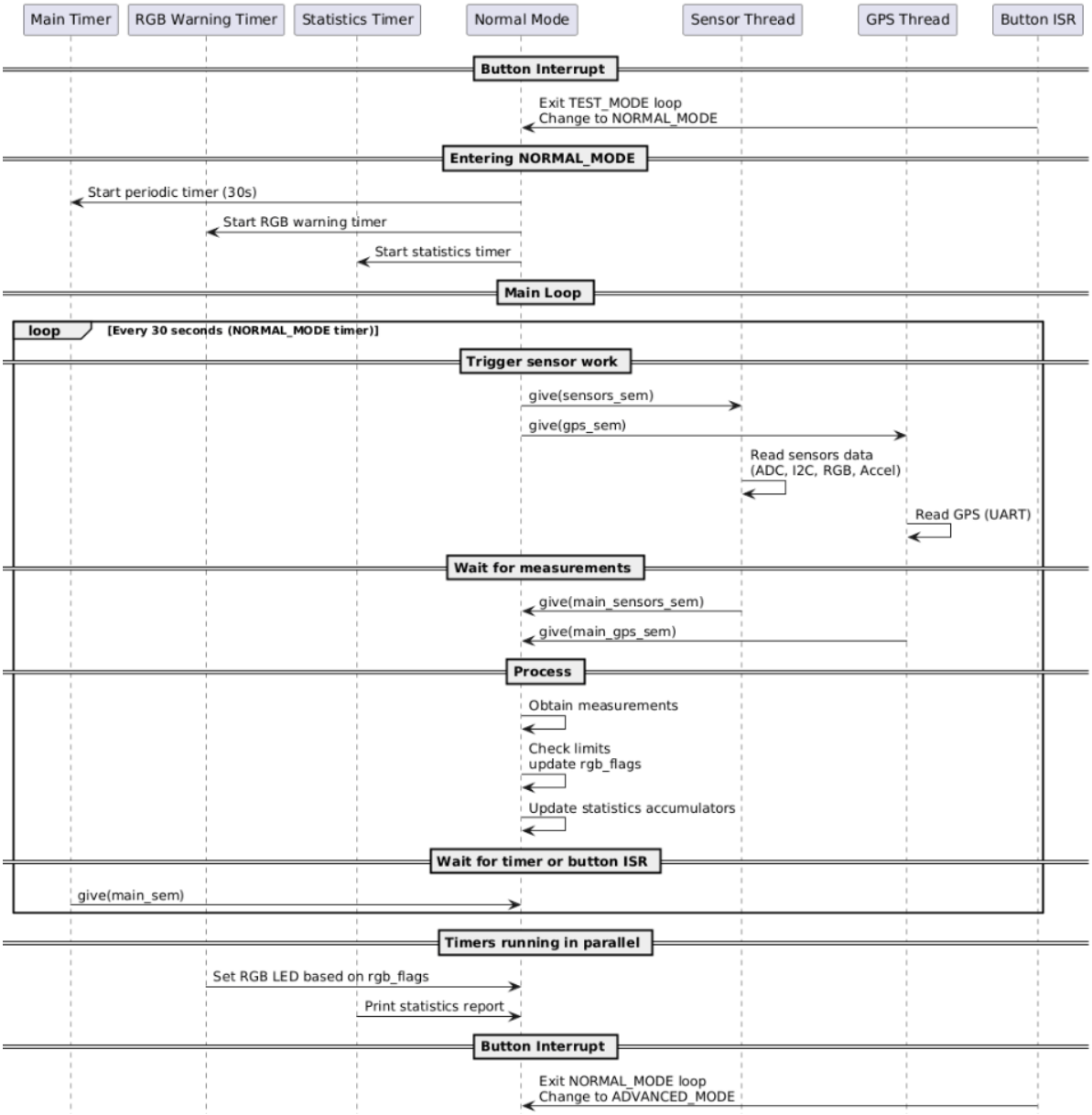

Figure 7: Normal mode behaviour

In **ADVANCED_MODE**, the system focuses on reproducing the sensed colour with high fidelity:

1. The measurement timer is started with the advanced-mode sampling rate (30 seconds).
2. Sensor and GPS threads acquire data and raise their semaphores.
3. The main thread retrieves all sensor values from the atomic shared structure.
4. Colour normalization is performed using the clear channel of the colour sensor, and shown.

5. The RGB LED is set to the normalized colour values for accurate reproduction (emulated PWM).
6. At each timer expiration, the `main_sem` semaphore is released to repeat the mode.
7. No limit checking, no statistics, and no warning indicators are active.

ADVANCED_MODE is a pure high-resolution colour rendering mode.

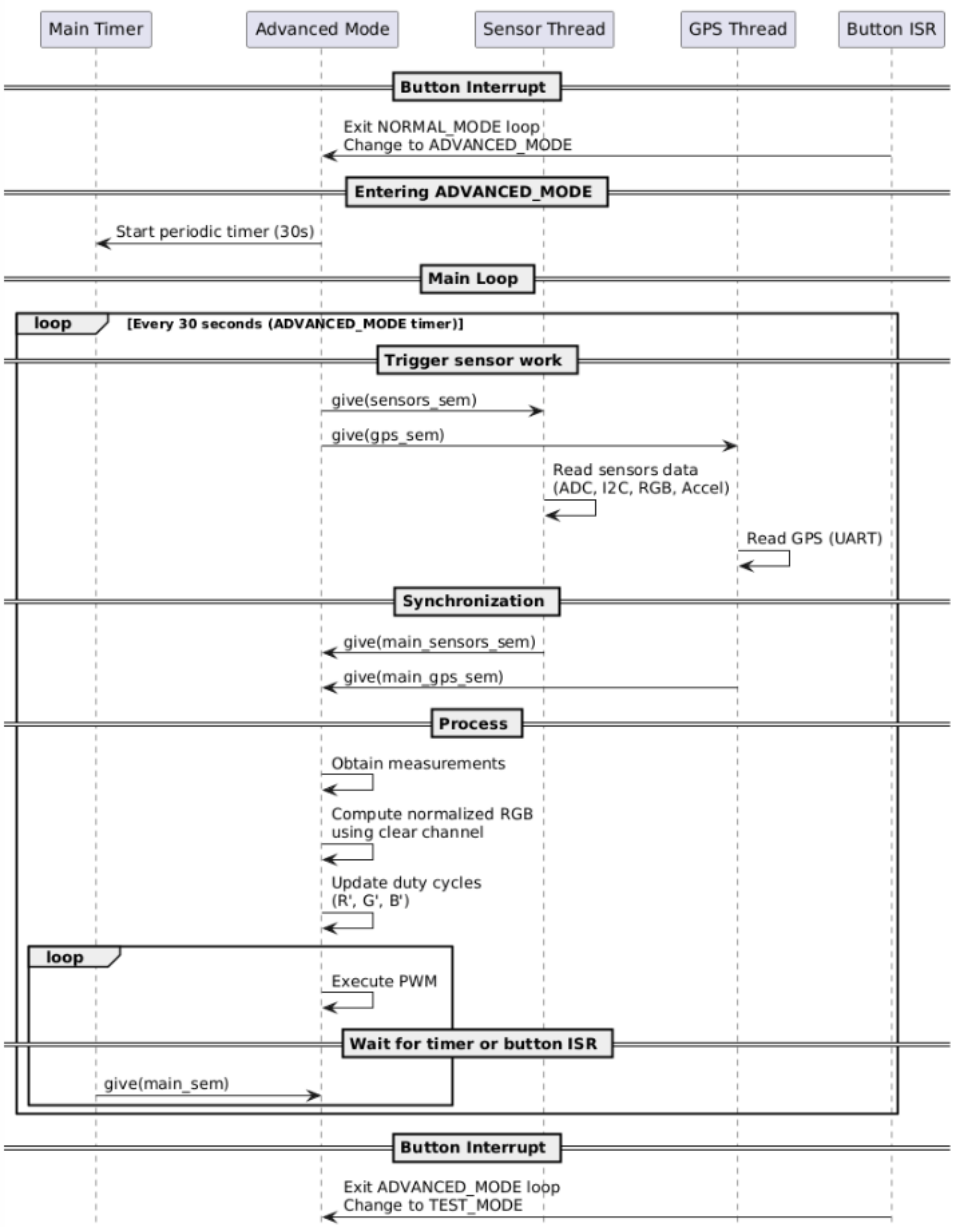

Figure 8: Advanced mode behaviour

4.1.2 Mode Transition Mechanism

The user button controls mode transitions. The button interrupt triggers a deferred work handler, ensuring transitions occur in thread context. The mode progression is cyclic:

$$\texttt{TEST\_MODE} \rightarrow \texttt{NORMAL\_MODE} \rightarrow \texttt{ADVANCED\_MODE} \rightarrow \texttt{TEST\_MODE}$$

During a mode change:

- Timers associated with the previous mode are stopped,

- Mode-specific timers are started (measurement, RGB warning, statistics),
- LEDs are updated to reflect the new mode,
- `main_sem` is released to unblock the main loop.

4.1.3 Semaphore Synchronization and Thread Interactions

Inter-thread coordination relies on multiple named semaphores.

- **Measurement Timer Semaphore (`main_sem`)**

  Released by `main_timer` to:

  - Wake the main thread at the configured sampling frequency to repeat the mode.
  - Wake the main thread when a mode transition occurs (button pressed).

- **Trigger Semaphores (`sensors_sem` and `gps_sem`)**

  These semaphores are released by `main_timer` to notify the respective threads to begin their measurement cycles.

- **Sensor Data Semaphore (`main_sensors_sem`)**

  Raised by the sensor acquisition thread after reading:

  - Temperature and humidity.
  - RGB colour channels.
  - Brightness.
  - Soil moisture.
  - Accelerometer values.

  The main thread blocks until this semaphore is obtained, ensuring consistent data.

- **GPS Data Semaphore (`main_gps_sem`)**

  Raised by the GPS thread when new GPS information (latitude, longitude, altitude, satellites, timestamp) is available.

  The main thread blocks until this semaphore is obtained, ensuring consistent data.

Together, these semaphores serialize measurement flow and guarantee no partial or inconsistent samples.

4.1.4 Atomic Shared Measurement Structure

Sensor data is stored in a global structure using only atomic variables:

Listing 1: Sensor data structure with atomic fields

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

    atomic_t accel_x_g; /**< Latest X-axis acceleration (in g). */
    atomic_t accel_y_g; /**< Latest Y-axis acceleration (in g). */
    atomic_t accel_z_g; /**< Latest Z-axis acceleration (in g). */

    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 (float or encoded). */
};
```

Properties:

- Lock-free thread-safe communication.
- Each measurement updated independently.
- the main thread reads all values without risk of torn writes.

4.1.5 Peripheral Configuration Structure

All peripherals and synchronization objects are referenced through a single shared structure:

Listing 2: Peripheral configuration and synchronization structure

```
struct system_context {
    struct adc_config *phototransistor; /**< Phototransistor ADC configuration. */
    struct adc_config *soil_moisture; /**< Soil moisture ADC configuration. */

    struct i2c_dt_spec *accelerometer; /**< Accelerometer I2C device specification. */
    uint8_t accel_range;            /**< Accelerometer full-scale range (e.g., 2G, 4G, 8G). */

    struct i2c_dt_spec *temp_hum;   /**< Temperature and humidity sensor I2C specification. */
    struct i2c_dt_spec *color;      /**< Color sensor I2C device specification. */
    struct gps_config *gps;         /**< GPS module configuration. */

    struct k_sem *main_sensors_sem; /**< Semaphore for main-to-sensors synchronization. */
    struct k_sem *main_gps_sem;     /**< Semaphore for main-to-GPS synchronization. */
    struct k_sem *sensors_sem;      /**< Semaphore to trigger sensor measurement. */
    struct k_sem *gps_sem;          /**< Semaphore to trigger GPS measurement. */
};
```

In this way, the peripherals are configured once at startup and passed to all threads and modules that require access.

4.1.6 Secure Initialization

In case of initialization failures (e.g., I2C device not found), the system doesn't execute the program:

Listing 3: Secure initialization code

```
    /* Initialize peripherals */
    if (gps_init(&gps)) {
        printk("GPS initialization failed - Program stopped\n");
        return -1;
    }
    if (adc_init(&pt)) {
        printk("Phototransistor initialization failed - Program stopped\n");
        return -1;
    }
    if (adc_init(&sm)) {
        printk("Soil moisture sensor initialization failed - Program stopped\n");
        return -1;
    }
    if (accel_init(&accel, ACCEL_RANGE)) {
        printk("Accelerometer initialization failed - Program stopped\n");
        return -1;
```

```
}
if (temp_hum_init(&th, TEMP_HUM_RESOLUTION)) {
    printk("Temperature/Humidity sensor initialization failed - Program stopped\n");
    return -1;
}
if (color_init(&color, COLOR_GAIN, COLOR_INTEGRATION_TIME)) {
    printk("Color sensor initialization failed - Program stopped\n");
    return -1;
}
if (led_init(&leds) || led_off(&leds)) {
    printk("LED initialization failed - Program stopped\n");
    return -1;
}
if (rgb_led_init(&rgb_leds) || rgb_led_off(&rgb_leds)) {
    printk("RGB LED initialization failed - Program stopped\n");
    return -1;
}
if (button_init(&button)) {
    printk("Button initialization failed - Program stopped\n");
    return -1;
}
if (button_set_callback(&button, button_isr)) {
    printk("Button callback setup failed - Program stopped\n");
    return -1;
}
```

Figure 9 shows an example of the log printed when initialization fails. In this case, the accelerometer is disconnected.

```
*** Booting Zephyr OS build v4.2.0-3924-gccaaaabba261 ***
==== Plant Monitoring System ====
System ON (TEST MODE)

[GPS] - Initializing GPS UART...
[GPS] - GPS initialized successfully
[ADC] - Initializing ADC device adc@40012400...
[ADC] - ADC device adc@40012400 initialized successfully
[ADC] - Initializing ADC device adc@40012400...
[ADC] - ADC device adc@40012400 initialized successfully
[ACCEL] - Initializing ACCEL...
[ACCEL] - ACCEL WHO_AM_I mismatch: 0x00
Accelerometer initialization failed - Program stopped
```

Figure 9: Stopped initialisation

4.1.7 Device Disconnected

In case one device is disconnected during operation, an error message is printed, but the program continues running. It shows the last valid measurement for that sensor. If the device is reconnected, normal operation resumes.

- In GREEN, the log that shows the sensor is disconnected.
- In BLUE is highlighted the same value of the measurement, demonstrating that the system shows the last valid measurement.
- In RED, there isn't log of sensor disconnected, because the sensor is connected again. The value of the measurement is updated.

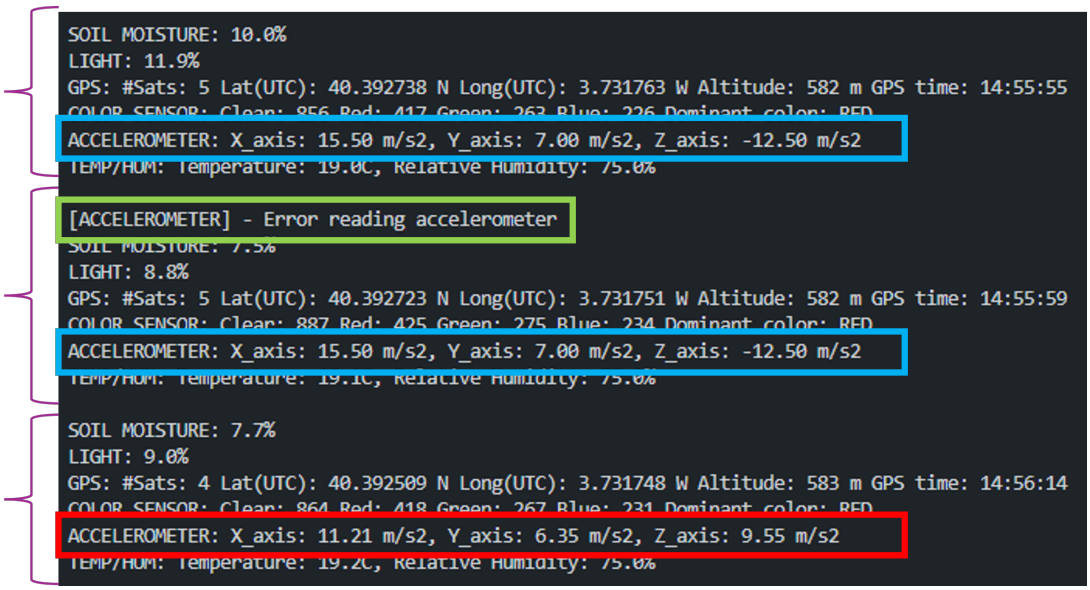

Figure 10: Disconnected and Connected sensor measurements

4.2 Modules

The program is structured into several self-contained modules, each responsible for a specific hardware interface or functionality.

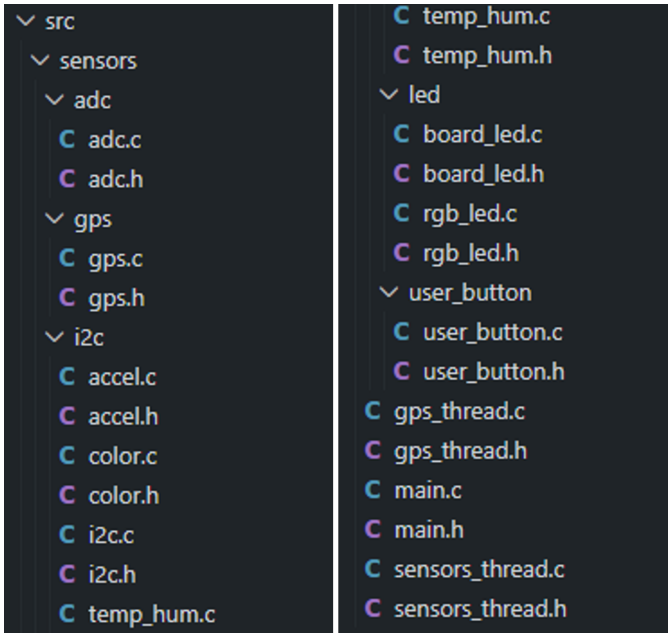

Figure 11: Modules overview

The main modules are used by the main program and the threads to interact with sensors and peripherals. This is done by adding them as source files and include directories in the `CMakeLists.txt` file, as shown below:

Listing 4: CMakeLists.txt module integration

```
# SPDX-License-Identifier: Apache-2.0

cmake_minimum_required(VERSION 3.20.0)

find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
project(plant_monitoring_system)

target_sources(app PRIVATE
    src/main.c
    src/sensors_thread.c
    src/gps_thread.c
    src/sensors/led/rgb_led.c
    src/sensors/led/board_led.c
    src/sensors/adc/adc.c
    src/sensors/user_button/user_button.c
    src/sensors/i2c/i2c.c
    src/sensors/i2c/accel.c
    src/sensors/i2c/color.c
    src/sensors/i2c/temp_hum.c
    src/sensors/gps/gps.c
)

target_include_directories(app PRIVATE
    src/sensors/led
    src/sensors/adc
    src/sensors/user_button
    src/sensors/i2c
    src/sensors/gps
```

```
)
```

4.2.1 adc.c and adc.h

The `adc.c` and `adc.h` modules provide a hardware abstraction layer for reading analogue values from the phototransistor and soil-moisture sensors using the Zephyr ADC Application Programming Interface (API). Their purpose is to encapsulate ADC initialisation, configuration, and acquisition into a self-contained interface that the sensors thread can use without exposing low-level driver details.

This library is responsible for preparing the ADC peripheral, configuring its channels, and performing synchronous conversions on demand. It ensures that all ADC reads comply with a consistent configuration (resolution, reference, acquisition time, and oversampling) and that raw sample values are returned in a unified format to the rest of the system.

- **Peripheral Initialisation**: Loads the ADC device from the device tree and configures the hardware according to project requirements. This includes selecting resolution, reference voltage, acquisition time, and optional oversampling.
- **Channel Setup**: Each analogue sensor has an associated ADC channel configured through a dedicated structure. The module ensures correct pin routing and channel mapping according to the board overlay.
- **Synchronous ADC Sampling**: Provides a blocking API that triggers a single conversion and returns the measured sample. This prevents concurrency issues by guaranteeing that read operations finish before returning control.
- **Unified Abstraction for Higher-Level Modules**: The sensors thread and the system context only interact with a clean, high-level interface without needing to manage ADC device handles, channels, or Zephyr-specific configuration fields.
- **Validation and Error Handling**: Detects device-not-found conditions, invalid configurations, or read failures, forwarding errors to the main system so that appropriate recovery or safe behaviour can occur.
- **Scalability for Additional Channels**: The design allows new analogue sensors to be added by defining a new channel configuration and calling the same acquisition API, without modifying existing code.
- **Separation of Configuration from Logic**: The module centralises all ADC configuration parameters in one place, ensuring future modifications (e.g., resolution, gain, sampling frequency) do not propagate across the project.

This ADC library is intended to be reused by any component that requires analogue-to-digital conversions while maintaining a clean separation between hardware-specific details and system-level functionality.

4.2.2 gps.c and gps.h

The `gps.c` and `gps.h` modules implement a GPS interface based on UART-driven reception of NMEA sentences. Their purpose is to offer a simple, self-contained parser for Global Positioning System Fix Data (GGA) (or Global Navigation Satellite System Fix Data (GNGGA)) frames and provide the rest of the system with clean, validated geographic data without exposing UART or interrupt logic.

The library configures the UART peripheral, enables interrupt-driven reception, reconstructs NMEA lines in the background, and parses the relevant fields when a complete GGA sentence is detected. Once valid GPS information is available, the module updates an internal `gps_data_t` structure and releases a semaphore so that higher-level threads can safely retrieve the most recent fix.

- **UART-Based GPS Initialisation**: The module validates the configuration, checks device readiness, attaches the Interruption Service Routine (ISR), and enables RX interrupts. This ensures autonomous background reception of NMEA data.
- **Interrupt-Driven NMEA Line Reconstruction**: Bytes received from the UART First In, First Out (FIFO) are accumulated into an internal buffer until a newline character is found.

- **GGA Sentence Parsing**: Extracts latitude, longitude, altitude, Horizontal Dilution of Precision (HDOP), satellite count, and UTC time from a standard GGA frame. The parser handles missing or malformed fields gracefully and only accepts complete, coherent entries.
- **NMEA-to-Degrees Conversion**: Converts coordinates from NMEA format (DDMM.MMMM or DDDMM.MMMM) to decimal degrees, applying hemisphere correction. This provides the system with immediately usable geographic values.
- **Thread Synchronisation via Semaphore**: When new valid data is parsed, a semaphore is released so that the GPS thread or main thread can block until a fresh fix is available. This avoids polling and reduces CPU usage.
- **Internal Data Buffering**: The module maintains an internal instance of `gps_data_t` storing the last valid parsed frame. Consumers obtain a copy, ensuring thread safety without exposing shared mutable structures.
- **Timeout-Aware Data Retrieval**: The high-level API allows callers to wait indefinitely, for a fixed period, or return immediately if no new GGA sentence has been received.

This GPS interface provides a robust and maintainable foundation for acquiring geographic data in real time, isolating UART management and parsing details from the rest of the application.

4.2.3 i2c.c and i2c.h

The `i2c.c` and `i2c.h` modules implement a small set of helper functions designed to simplify register-level communication with I2C devices in Zephyr. Their purpose is to provide a clean, reusable interface for reading and writing device registers using only a devicetree `i2c_dt_spec`, avoiding repetitive low-level code in sensor drivers.

The library encapsulates common I2C access patterns into concise functions. These utilities internally rely on Zephyr's `i2c_write_read_dt`, `i2c_write_dt`, and `i2c_is_ready_dt` APIs, ensuring compatibility with any I2C peripheral described in the system devicetree.

Because many sensors require register-based configuration and multi-byte reads, this module centralises these operations and presents a uniform interface that higher-level modules can reuse safely.

- **Multi-Register Read Helper**: Reads an arbitrary number of consecutive registers starting at a given address, abstracting the common write-then-read transaction pattern.
- **Single-Register Write Helper**: Writes one byte to a specified register, a frequent requirement for sensor configuration and control registers.
- **Device Readiness Check**: Verifies that the I2C device is present, powered, and ready before attempting communication. Provides clear error reporting if the device is not reachable.
- **Consistent Devicetree-Based Access**: All functions operate on `i2c_dt_spec` descriptors, ensuring that pin routing, bus selection, addressing and timing come directly from the devicetree.
- **Reusability for Multiple Sensor Drivers**: Higher-level modules (accelerometer, colour sensor, temperature/humidity sensor, etc.) use these helpers to avoid code duplication and maintain consistency across all I2C devices.
- **Error Propagation**: Returns standard negative errno codes, allowing calling modules to handle failures predictably and implement fallback or retry mechanisms.

This I2C helper library provides a clean and robust foundation for register-based communication with any I2C sensor or peripheral in the system.

4.2.4 accel.c and accel.h

The `accel.c` and `accel.h` modules implement a 3-axis accelerometer driver over I2C, providing initialization, configuration, raw data acquisition, and unit conversion utilities. The module abstracts all register-level interaction and exposes a clean interface for obtaining acceleration data in either raw counts, g units, or m/s^2.

During initialization, the library verifies device identity via the `WHO_AM_I` register, transitions the sensor into standby mode, configures its measurement range, and finally activates continuous measurement mode. The project uses the ±2g range, which maximizes sensitivity for environmental and motion-tracking applications.

Raw acceleration values for X, Y, and Z axes are obtained through a single burst read of six consecutive registers. This ensures atomic acquisition of all three axes and prevents axis desynchronization. Each pair of bytes contains a 14-bit left-aligned signed measurement, which the library re-aligns before returning to higher-level modules.

All I2C communication relies on the generic register helpers defined in the I2C module, keeping the driver compact and uniform with the rest of the system.

- **Device Initialization and Identity Check**: Reads the `WHO_AM_I` register to validate the sensor's presence before configuration or data acquisition.
- **Standby and Active Mode Control**: Ensures that configuration registers are only modified while the device is in standby mode, as required by the hardware design. The module automatically returns the device to active mode after configuration.
- **Measurement Range Configuration**: Supports ±2g, ±4g, and ±8g ranges via the `XYZ_DATA_CFG` register. The system uses ±2g for improved resolution and noise performance.
- **Burst Read of 6 Output Registers**: Performs a single multi-register transaction to retrieve X, Y, and Z values consistently, minimizing communication overhead and avoiding partial updates.
- **Raw Data Alignment and Extraction**: Converts the sensor's 14-bit left-aligned format into signed 14-bit integers usable by higher-level modules.
- **Conversion to g Units**: Applies sensitivity scaling based on the configured range, providing a convenient floating-point representation.
- **Conversion to m/s²**: Converts from g units to SI units using standard gravity, enabling direct use in physical calculations or movement detection algorithms.

This accelerometer module provides a robust basis for motion sensing, offering clean abstractions for configuration, raw acquisition, and physical-unit conversion while preserving full compatibility with the system's I2C infrastructure.

4.2.5 color.c and color.h

The `color.c` and `color.h` modules implement a driver for the TCS34725 RGB colour sensor using the Zephyr I2C API. Their purpose is to provide a clean, high-level interface for configuring the device, and acquiring raw Clear/Red/Green/Blue measurements without exposing low-level register logic to the rest of the system.

During initialization, the module validates I2C bus readiness, powers on the device, enables the internal ADC, and applies the user-specified gain and integration time. Raw colour values are retrieved through a single auto-increment burst read of the sensor's Red, Green, Blue and Clear (RGBC) output registers, ensuring consistent sampling of all channels.

- **I2C-Based Sensor Initialisation**: The module verifies that the I2C bus is ready, sends the power-on command, enables the RGBC ADC, and configures the device using register writes to `ATIME` and `CONTROL`. Any communication error propagates as a negative `errno` code.
- **Gain Configuration**: The driver supports all hardware gain settings (1×, 4×, 16×, 60×). Gain determines the analogue pre-amplification applied to each colour photodiode. Lower gains avoid saturation in bright environments, while higher gains improve low-light sensitivity.
- **Integration Time Configuration**: Integration time controls the duration of the ADC light accumulation interval. Short times (e.g. 2.4ms) allow fast updates but with lower resolution, while long times (up to 700ms) significantly improve sensitivity. The selected timing constant is written directly to the `ATIME` register.
- **Burst Read of RGBC Channels**: The function `color_read_rgb()` performs a single multi-register transaction starting at `COLOR_CLEAR_L`, retrieving all Clear, Red, Green, and Blue low/high

bytes using auto-increment addressing. This guarantees coherence between channels and minimizes I2C overhead.

- **16-bit Data Reconstruction**: Raw samples are assembled from consecutive low/high bytes and stored into a `ColorSensorData` structure. The Clear channel is reported alongside RGB values, enabling normalisation or illumination-compensation algorithms at higher layers.
- **Error Handling and Consistent API**: All functions return standard negative `errno` values on failure, ensuring predictable error reporting and uniform behaviour across all I2C-based sensor modules.

This colour-sensor driver offers a compact and maintainable interface for acquiring raw RGBC data, abstracting all register and timing details while remaining fully consistent with the system's I2C infrastructure.

4.2.6 temp_hum.c and temp_hum.h

The `temp_hum.c` and `temp_hum.h` modules implement a driver for the Si7021 temperature and humidity sensor over I2C. Their purpose is to provide a synchronous, resolution-configurable interface for acquiring relative humidity (%RH) and temperature (C) values while abstracting all low-level I2C communication.

The library initializes the sensor, performs a soft reset, sets the desired measurement resolution, and provides functions for reading both temperature and humidity using **Hold Master mode**. In this mode, the sensor holds the SCL line low while a measurement is in progress, ensuring that the master waits synchronously until the data is ready.

- **Device Initialization and Soft Reset**: `temp_hum_init()` validates the I2C bus, issues a soft reset (`TH_RESET`), waits for the sensor to stabilize, and writes the resolution to User Register 1.
- **Hold Master Mode Measurements**: Both `temp_hum_read_humidity()` and `temp_hum_read_temperature()` use Hold Master mode commands (`TH_MEAS_RH_HOLD` and `TH_MEAS_TEMP_HOLD`) to ensure synchronous reading without the need for polling or manual delays.
- **Relative Humidity Conversion**: Converts the 16-bit raw measurement from the sensor into %RH according to the Si7021 datasheet formula, clamping values to the physical range of 0-100%.
- **Temperature Conversion**: Converts the 16-bit raw measurement into degrees Celsius using the datasheet formula, providing accurate environmental temperature readings.
- **Error Handling**: All functions return standard negative `errno` codes in case of communication or configuration failures, allowing higher-level modules to react accordingly.

This I2C-based temperature and humidity driver offers a simple, reliable, and synchronous interface, isolating low-level communication and Hold Master timing from the rest of the application.

4.2.7 rgb_led.c and rgb_led.h

The `rgb_led.c` and `rgb_led.h` modules implement control of a RGB LED connected via three GPIO pins (Red, Green, Blue). This module provides initialization, individual color control, and mixed color combinations through a bitmask-based bus interface. It is important to note that the RGB LED channels are **active-low**, meaning a logical 0 turns the LED on, and a logical 1 turns it off.

The library initializes all GPIO pins, verifies device readiness, and sets them to an inactive state by default. Functions allow activation of standard colors, full white, or complete off states.

- **Bus-Based GPIO Initialization**: `rgb_led_init()` iterates over all pins defined in the `bus_rgb_led` structure, ensures the associated GPIO device is ready, and configures the pins as outputs with an initial inactive state.
- **Bitmask-Controlled Color Output**: `rgb_led_write()` maps each bit of a 3-bit value to a corresponding LED channel (bit0=Red, bit1=Green, bit2=Blue). This bus-like approach allows simultaneous activation of multiple channels to produce mixed colors.

- **Convenience Color Functions**: Functions such as `rgb_red()`, `rgb_green()`, `rgb_blue()`, `rgb_yellow()`, `rgb_cyan()`, `rgb_purple()`, `rgb_white()`, and `rgb_black()` call `rgb_led_write()` internally with predefined bitmasks, providing simple color selection.
- **Active-Low Behavior**: Since the RGB LED channels are active-low, a logical 0 on a GPIO pin activates the LED, while a logical 1 turns it off. All helper functions and the bus interface respect this behavior.
- **Error Handling**: Initialisation and write operations check for GPIO device readiness and return standard negative `errno` codes on failure, allowing safe integration with higher-level modules.

This RGB LED module provides a reliable, bus-oriented, and reusable interface for color control, abstracting low-level active-low pin management while supporting individual colors and combined outputs.

4.2.8 board_led.c and board_led.h

The `board_led.c` and `board_led.h` modules implement an abstraction for controlling board LEDs using GPIO pins. Their purpose is to provide a clean and reusable interface for turning board LEDs on/off, setting specific colors, and combining color channels via bitmask control.

The library initializes the GPIO pins, verifies device readiness, configures outputs, and exposes functions to control individual colors or common combinations. Bitmask-based operations allow multiple channels to be activated simultaneously, producing standard colors (e.g., yellow, cyan, magenta, white).

- **GPIO Initialization**: `led_init()` iterates over all configured pins in the `bus_led` structure, checks device readiness, and configures each pin as an output with a default off state. Errors are reported if a pin or device is not ready.
- **Bitmask-Based Color Control**: `led_write()` maps each bit of the input value to a corresponding LED channel (bit0=Red, bit1=Green, bit2=Blue). This allows direct control of RGB combinations in a single function call.
- **Convenience Functions for Colors**: The module provides higher-level functions (`red()`, `green()`, `blue()`, `red_green()`, `green_blue()`, `red_blue()`, `led_on()`, `led_off()`) that internally call `led_write()` with predefined bitmask values to simplify usage.
- **Error Handling**: All GPIO operations check for errors and return standard negative `errno` codes if configuration or write fails, enabling higher-level modules to respond appropriately.

This GPIO-based LED driver provides a simple, reliable, and reusable interface for LED control, isolating low-level pin management while enabling flexible color and combination handling.

4.2.9 user_button.c and user_button.h

The `user_button.c` and `user_button.h` modules implement a GPIO-based user button interface with interrupt support. This driver allows initialization of a button input pin, configuration of edge-triggered interrupts, and registration of a callback function to handle button press and release events. It is designed for use with a single GPIO pin per button, using pull-up configuration and detecting both rising and falling edges.

The library provides a clean abstraction for integrating physical buttons into the system without exposing low-level GPIO interrupt setup details.

- **GPIO-Based Button Initialization**: `button_init()` verifies the GPIO device is ready, configures the pin as input with pull-up, and enables interrupts on both edges to detect presses and releases.
- **Edge-Triggered Interrupts**: Both rising and falling edges are detected, allowing the application to respond to button presses and releases independently.
- **Callback Registration**: `button_set_callback()` allows the application to attach an ISR handler that will be executed in interrupt context when the configured edge is detected.
- **Safe Error Handling**: All functions return standard negative `errno` codes on failure (e.g., GPIO device not ready, invalid configuration), allowing the calling module to handle errors predictably.

- **Lightweight ISR Support**: The module leaves the implementation of press/release logic to the application via the registered callback, ensuring ISR code remains minimal and safe.
- **Integration with Zephyr GPIO API**: Uses `gpio_pin_configure_dt()`, `gpio_pin_interrupt_configure_dt()`, and `gpio_add_callback()` internally, abstracting Zephyr-specific details from higher-level code.

This user button module provides a robust and reusable interface for integrating physical buttons with interrupt-driven event handling, isolating low-level GPIO configuration and edge detection logic from the application.

4.3 Threads

4.3.1 Sensors Thread

The sensors measurement thread handles the acquisition of data from a heterogeneous set of devices, including:

- **ADC sensors:** ambient brightness and soil moisture.
- **I2C sensors:** accelerometer, temperature/humidity sensor, and RGB color sensor.

The thread stores all gathered data in the shared `system_measurement` structure using atomic operations to guarantee thread-safe data consistency.

The initialization routine creates the sensors thread and assigns its execution parameters. The thread begins running immediately after creation.

Listing 5: Sensors thread initialization

```
void start_sensors_thread(struct system_context *ctx,
                          struct system_measurement *measure) {

    k_thread_create(&sensors_thread_data,
                    sensors_stack,
                    K_THREAD_STACK_SIZEOF(sensors_stack),
                    sensors_thread_fn,
                    ctx, measure, NULL,
                    SENSORS_THREAD_PRIORITY, 0, K_NO_WAIT);

    k_thread_name_set(&sensors_thread_data, "sensors_thread");
}
```

The configuration of the sensors thread includes the definition of its stack, priority, and control block, as shown in Listing 6. Zephyr's `K_THREAD_STACK_DEFINE` macro is used to statically allocate the execution stack.

Listing 6: Sensors thread configuration

```
#define SENSORS_THREAD_STACK_SIZE 1024
#define SENSORS_THREAD_PRIORITY 5

K_THREAD_STACK_DEFINE(sensors_stack, SENSORS_THREAD_STACK_SIZE);
static struct k_thread sensors_thread_data;
```

ADC-based sensors (brightness and soil moisture) are processed using the utility function `read_adc_percentage()`, shown in Listing 7. This function converts the raw ADC voltage into a scaled percentage value, expressed as percentage times ten to preserve one decimal point of precision.

Listing 7: ADC percentage acquisition helper function

```
static void read_adc_percentage(const struct adc_config *cfg, atomic_t *target,
                                const char *label, int32_t *mv)
{
    if (adc_read_voltage(cfg, mv) == 0) {
        int32_t percent10 = ((*mv) * 1000) / cfg->vref_mv;
        atomic_set(target, percent10);
    } else {
        printk("[ADC]: %s read error\n", label);
    }
}
```

The accelerometer is interfaced over I2C and provides raw XYZ readings which are converted to acceleration values in m/s^2 using the device's full-scale range. The processed values are scaled by 100

to preserve two decimal places of resolution. Listing 8 shows the implementation of the accelerometer handling routine.

Listing 8: Accelerometer data acquisition

```
static void read_accelerometer(const struct i2c_dt_spec *dev, uint8_t range,
                               atomic_t *x_ms2, atomic_t *y_ms2, atomic_t *z_ms2) {
    int16_t x_raw, y_raw, z_raw;
    float x_val, y_val, z_val;

    if (accel_read_xyz(dev, &x_raw, &y_raw, &z_raw) == 0) {
        accel_convert_to_ms2(x_raw, range, &x_val);
        accel_convert_to_ms2(y_raw, range, &y_val);
        accel_convert_to_ms2(z_raw, range, &z_val);

        atomic_set(x_ms2, (int32_t)(x_val * 100));
        atomic_set(y_ms2, (int32_t)(y_val * 100));
        atomic_set(z_ms2, (int32_t)(z_val * 100));
    } else {
        printk("[ACCELEROMETER] - Error reading accelerometer\n");
    }
}
```

The temperature and humidity sensor also communicates through I2C. Humidity measurement implicitly triggers a temperature conversion, after which the associated temperature value can be read. Both humidity and temperature values are scaled by a factor of 100.

Listing 9: Temperature and humidity acquisition

```
static void read_temperature_humidity(const struct i2c_dt_spec *dev,
                                      atomic_t *temp, atomic_t *hum) {

    float humidity;

    if (temp_hum_read_humidity(dev, &humidity) == 0) {
        float temperature;
        uint8_t buf[2];

        int ret = i2c_write_read_dt(dev,
                                    (uint8_t[]){ TH_READ_TEMP_FROM_RH },
                                    1, buf, 2);

        if (ret == 0) {
            uint16_t raw_temp = ((uint16_t)buf[0] << 8) | buf[1];
            temperature = ((175.72f * raw_temp) / 65536.0f) - 46.85f;
        } else {
            printk("[TEMP_HUM SENSOR] - Error reading temperature from RH (%d)\n", ret);
            return;
        }

        atomic_set(hum, (int32_t)(humidity * 100));
        atomic_set(temp, (int32_t)(temperature * 100));

    } else {
        printk("[TEMP_HUM SENSOR] - Read error (humidity)\n");
    }
}
```

The RGB color sensor provides raw red, green, blue, and clear-channel information. These values are written directly into the measurement structure without additional scaling, as shown in Listing 10.

Listing 10: Color sensor acquisition

```
static void read_color_sensor(const struct i2c_dt_spec *dev,
                              struct system_measurement *measure) {
    ColorSensorData color_data;

    if (color_read_rgb(dev, &color_data) == 0) {
        atomic_set(&measure->red, color_data.red);
        atomic_set(&measure->green, color_data.green);
        atomic_set(&measure->blue, color_data.blue);
        atomic_set(&measure->clear, color_data.clear);
    } else {
        printk("[COLOR SENSOR] - Read error\n");
    }
}
```

The main execution loop of the sensors thread is shown below. The thread waits for a semaphore signal before performing a complete acquisition cycle across all sensors. Once finished, it releases a semaphore to notify the main thread that new measurements are available.

Listing 11: Sensors thread main loop

```
static void sensors_thread_fn(void *arg1, void *arg2, void *arg3) {
    struct system_context *ctx = (struct system_context *)arg1;
    struct system_measurement *measure = (struct system_measurement *)arg2;

    int32_t mv = 0;

    while (1) {
        k_sem_take(ctx->sensors_sem, K_FOREVER);

        read_adc_percentage(ctx->phototransistor, &measure->brightness, "Brightness", &mv);
        read_adc_percentage(ctx->soil_moisture, &measure->moisture, "Moisture", &mv);
        read_accelerometer(ctx->accelerometer, ctx->accel_range,
                           &measure->accel_x_g, &measure->accel_y_g, &measure->accel_z_g);
        read_temperature_humidity(ctx->temp_hum, &measure->temp, &measure->hum);
        read_color_sensor(ctx->color, measure);

        k_sem_give(ctx->main_sensors_sem);
    }
}
```

The public interface for the sensors thread is shown in Listing 12. It exposes the initialization function and documents the required input structures.

Listing 12: Sensors thread public header

```
#ifndef SENSORS_THREAD_H
#define SENSORS_THREAD_H

#include "main.h"

void start_sensors_thread(struct system_context *ctx,
                          struct system_measurement *measure);

#endif /* SENSORS_THREAD_H */
```

4.3.2 GPS Thread

The GPS measurement thread is responsible for interfacing with the GPS driver, extracting relevant NMEA GGA information, and updating the global `system_measurement` structure. Its main characteristics include:

- Periodic GPS polling synchronized with the system's operational mode.

- Thread-safe shared-memory updates using atomic setters.
- Use of semaphores, thread stacks, and thread control blocks.
- Scaled integer representation of latitude, longitude, altitude, and UTC time.

The GPS measurement thread is created and launched through the function `start_gps_thread()`, shown in Listing 13. This routine initializes the thread with its designated stack, priority, entry function, and arguments.

Listing 13: Initialization of the GPS measurement thread

```
void start_gps_thread(struct system_context *ctx,
                      struct system_measurement *measure) {

    k_thread_create(&gps_thread_data,
                    gps_stack,
                    K_THREAD_STACK_SIZEOF(gps_stack),
                    gps_thread_fn,
                    ctx, measure, NULL,
                    GPS_THREAD_PRIORITY, 0, K_NO_WAIT);

    k_thread_name_set(&gps_thread_data, "gps_thread");
}
```

Memory allocation and priority assignment for the GPS thread are specified as shown in Listing 14. A dedicated stack is defined using Zephyr's `K_THREAD_STACK_DEFINE` macro, and a thread control block is declared to manage its execution context.

Listing 14: GPS thread configuration in Zephyr

```
#define GPS_THREAD_STACK_SIZE 1024
#define GPS_THREAD_PRIORITY 5

K_THREAD_STACK_DEFINE(gps_stack, GPS_THREAD_STACK_SIZE);
static struct k_thread gps_thread_data;
```

The core of the GPS data-handling logic is encapsulated in the helper function `read_gps_data()`, shown in Listing 15. This function waits for a valid NMEA GGA frame, extracts geographic coordinates, altitude, satellite count, and UTC time, and stores them as scaled integers in the shared measurement structure.

Latitude and longitude are scaled by 10^6 to preserve decimal precision, while altitude is scaled by a factor of 100. UTC time is encoded in the `HHMMSS` format as a six-digit integer. Adding 1 to the hour component accounts for timezone adjustment (Spain UTC+1).

Listing 15: GPS data acquisition helper function

```
static void read_gps_data(gps_data_t *data,
                          struct system_measurement *measure,
                          struct system_context *ctx) {

    if (gps_wait_for_gga(data, K_MSEC(1000)) == 0) {
        atomic_set(&measure->gps_lat, (int32_t)(data->lat * 1e6f));
        atomic_set(&measure->gps_lon, (int32_t)(data->lon * 1e6f));
        atomic_set(&measure->gps_alt, (int32_t)(data->alt * 100.0f));
        atomic_set(&measure->gps_sats, (int32_t)data->sats);

        if (strlen(data->utc_time) >= 6) {
            int hh = (data->utc_time[0] - '0') * 10 + (data->utc_time[1] - '0') + 1;
            int mm = (data->utc_time[2] - '0') * 10 + (data->utc_time[3] - '0');
            int ss = (data->utc_time[4] - '0') * 10 + (data->utc_time[5] - '0');

            int time_int = hh * 10000 + mm * 100 + ss;
```

```
            atomic_set(&measure->gps_time, time_int);
        } else {
            atomic_set(&measure->gps_time, -1);
        }

    } else {
        printk("[GPS] - Timeout or invalid data\n");
    }
}
```

The main execution loop of the GPS thread is shown in Listing 16. The thread waits for a semaphore signal indicating that a GPS reading should be performed. Once awakened, it acquires a new GPS sample and signals the main thread upon completion. This mechanism provides deterministic synchronization between system components.

Listing 16: GPS thread entry routine

```
static void gps_thread_fn(void *arg1, void *arg2, void *arg3) {
    struct system_context *ctx = (struct system_context *)arg1;
    struct system_measurement *measure = (struct system_measurement *)arg2;

    gps_data_t gps_data = {0};

    while (1) {
        k_sem_take(ctx->gps_sem, K_FOREVER);
        read_gps_data(&gps_data, measure, ctx);
        k_sem_give(ctx->main_gps_sem);
    }
}
```

The corresponding public interface is declared in the header file `gps_thread.h`. As shown below, it specifies the initialization function and documents the dependency on `system_context` and `system_measurement` structures.

Listing 17: GPS thread public interface

```
#ifndef GPS_THREAD_H
#define GPS_THREAD_H

#include "main.h"

void start_gps_thread(struct system_context *ctx,
                      struct system_measurement *measure);

#endif /* GPS_THREAD_H */
```

4.4 Main

This section describes the main application logic of the system. It includes global configuration parameters, peripheral initialization, shared data structures, and the control flow that governs the system operating modes. The *Main* module acts as the central coordinator, managing timing, user interaction, and synchronization between threads.

4.4.1 Macro definitions

This subsection defines the main compile-time configuration parameters used throughout the application. These macros control system behavior such as operating modes, measurement periods, PWM timing, sensor configuration, and valid measurement ranges. Centralizing these definitions improves maintainability and allows system behavior to be adjusted without modifying the application logic.

Main Configuration

The following macros specify the initial operating mode and the timing parameters associated with each system mode. They also define timer periods for RGB LED updates and statistical reporting, as well as parameters required for software-based PWM generation.

Listing 18: Main configuration macros

```
#define INITIAL_MODE TEST_MODE /**< Initial operating mode at startup. */

#define TEST_PERIOD 2000   /**< Test mode measurement period in milliseconds. */
#define NORMAL_PERIOD 10000 /**< Normal mode measurement period in milliseconds. */

#define RGB_TIMER_PERIOD 500 /**< RGB LED timer period in milliseconds. */
#define STATS_TIMER_PERIOD 60000 /**< Statistics reporting period (ms). */

#define PWM_STEP  1            /**< PWM step in milliseconds. */
#define PWM_PERIOD 15          /**< PWM period in milliseconds. */
#define PWM_STEPS (PWM_PERIOD / PWM_STEP) /**< Number of PWM steps per period. */
```

Sensors Configuration

These macros configure the operating parameters of the connected sensors, including measurement ranges, gain settings, integration times, and resolution. The selected values represent a trade-off between accuracy, response time, and power consumption.

Listing 19: Sensors configuration macros

```
#define ACCEL_RANGE ACCEL_2G /**< Accelerometer full-scale range setting. */

#define COLOR_GAIN GAIN_4X   /**< Color sensor gain setting. */
#define COLOR_INTEGRATION_TIME INTEGRATION_154MS /**< Color sensor integration time in
    milliseconds. */

#define TEMP_HUM_RESOLUTION TH_RES_RH12_TEMP14 /**< Temperature and humidity sensor
    resolution setting. */
```

Measurement Limits

The measurement limit macros define the acceptable operating ranges for each sensor. These thresholds are used to detect abnormal conditions and to trigger visual or logical alarms when sensor readings fall outside predefined bounds.

Listing 20: Measurement limits macros

```
#define TEMP_MIN -10  /**< Minimum temperature in C. */
#define TEMP_MAX 50   /**< Maximum temperature in C. */

#define HUM_MIN  25   /**< Minimum humidity percentage. */
#define HUM_MAX  75   /**< Maximum humidity percentage. */
```

```
#define LIGHT_MIN 0    /**< Minimum brightness percentage. */
#define LIGHT_MAX 100  /**< Maximum brightness percentage. */

#define MOISTURE_MIN 0   /**< Minimum soil moisture percentage. */
#define MOISTURE_MAX 100 /**< Maximum soil moisture percentage. */

#define COLOR_MIN   0    /**< Minimum raw color channel value. */
#define COLOR_MAX   65535 /**< Maximum raw color channel value. */

#define ACCEL_MIN -2   /**< Minimum acceleration in g. */
#define ACCEL_MAX 2    /**< Maximum acceleration in g. */
```

Flags for limits alarms

Each flag represents a specific sensor exceeding its allowed range. These bitwise flags allow multiple alarm conditions to be tracked simultaneously using a single atomic variable.

Listing 21: Measurement limits flags macros

```
#define FLAG_TEMP   (1U << 0)
#define FLAG_HUM    (1U << 1)
#define FLAG_LIGHT  (1U << 2)
#define FLAG_MOISTURE (1U << 3)
#define FLAG_COLOR  (1U << 4)
#define FLAG_ACCEL  (1U << 5)
```

4.4.2 Peripheral configuration

This subsection defines the static configuration structures for all hardware peripherals used by the system. These configurations abstract the hardware details and provide a consistent interface for sensor drivers, communication buses, and user interface components.

Analog peripherals such as the phototransistor and soil moisture sensor are configured through ADC channels, while digital sensors communicate via the I2C bus. The GPS module uses a UART interface, and LEDs and the user button are configured through GPIO abstractions.

Listing 22: Peripheral configuration macros

```
/**
 * @brief Phototransistor ADC configuration.
 */
static struct adc_config pt = {
    .dev = DEVICE_DT_GET(DT_NODELABEL(adc1)),
    .channel_id = 5,
    .resolution = 12,
    .gain = ADC_GAIN_1,
    .ref = ADC_REF_INTERNAL,
    .acquisition_time = ADC_ACQ_TIME_DEFAULT,
    .vref_mv = 3300,
};

/**
 * @brief Soil moisture ADC configuration.
 */
static struct adc_config sm = {
    .dev = DEVICE_DT_GET(DT_NODELABEL(adc1)),
    .channel_id = 0,
    .resolution = 12,
    .gain = ADC_GAIN_1,
    .ref = ADC_REF_INTERNAL,
    .acquisition_time = ADC_ACQ_TIME_DEFAULT,
    .vref_mv = 3300,
```

```
};

/**
 * @brief Accelerometer I2C configuration.
 */
static struct i2c_dt_spec accel = {
    .bus = DEVICE_DT_GET(DT_NODELABEL(i2c2)),
    .addr = ACCEL_I2C_ADDR,
};

/**
 * @brief Temperature and humidity sensor I2C configuration.
 */
static struct i2c_dt_spec th = {
    .bus = DEVICE_DT_GET(DT_NODELABEL(i2c2)),
    .addr = TH_I2C_ADDR,
};

/**
 * @brief Color sensor I2C configuration.
 */
static struct i2c_dt_spec color = {
    .bus = DEVICE_DT_GET(DT_NODELABEL(i2c2)),
    .addr = COLOR_I2C_ADDR,
};

/**
 * @brief GPS UART configuration.
 */
static struct gps_config gps = {
    .dev = DEVICE_DT_GET(DT_NODELABEL(usart1)),
};

/**
 * @brief RGB LED bus configuration.
 */
static struct bus_rgb_led rgb_leds = {
    .pins = {
        GPIO_DT_SPEC_GET(DT_ALIAS(red), gpios),
        GPIO_DT_SPEC_GET(DT_ALIAS(green), gpios),
        GPIO_DT_SPEC_GET(DT_ALIAS(blue), gpios)
    },
    .pin_count = BUS_SIZE,
};

/**
 * @brief Indicator LED bus configuration.
 */
static struct bus_led leds = {
    .pins = {
        GPIO_DT_SPEC_GET(DT_ALIAS(led2), gpios),
        GPIO_DT_SPEC_GET(DT_ALIAS(led1), gpios),
        GPIO_DT_SPEC_GET(DT_ALIAS(led0), gpios)
    },
    .pin_count = BUS_SIZE,
};

/**
 * @brief User button configuration.
 */
static struct user_button button = {
    .spec = GPIO_DT_SPEC_GET(DT_ALIAS(sw0), gpios),
};
```

4.4.3 Data

This subsection presents the data structures and enumerations used for system state management and inter-thread communication.

The `system_mode_t` enumeration defines the three operating modes of the system, each corresponding to a different behavior and level of functionality. The dominant color enumeration is used to classify color sensor readings and to drive RGB LED feedback.

Shared structures such as `system_context` and `system_measurement` enable safe data exchange between threads. Sensor measurements are stored using atomic variables to guarantee consistency in a concurrent execution environment.

The `main_measurement` structure aggregates all processed data required by the main thread, while the statistics structure stores accumulated values used for computing mean, minimum, and maximum measurements over time.

Listing 23: Data structures and enumerations

```
/**
 * @enum system_mode_t
 * @brief System operating modes.
 *
 * These modes define how the system behaves:
 * - **TEST_MODE:** Shows the dominant detected color through the RGB LED.
 * - **NORMAL_MODE:** Periodically measures sensors and alerts if any reading
 *   is out of range.
 * - **ADVANCED_MODE:** Emulates PWM for RGB LED control.
 */
typedef enum {
    TEST_MODE = 0,  /**< Test mode - displays the dominant color. */
    NORMAL_MODE,    /**< Normal mode - periodic measurement and alerts. */
    ADVANCED_MODE   /**< Advanced mode - minimal visual feedback. */
} system_mode_t;

/**
 * @enum dom_color_t
 * @brief Dominant color types.
 */
typedef enum {
    DOM_RED,
    DOM_GREEN,
    DOM_BLUE
} dom_color_t;

const char* dom_color_names[] = { "RED", "GREEN", "BLUE" };

/**
 * @brief Shared system context.
 *
 * The @ref system_context structure holds references to the peripheral
 * configurations. The main, sensor, and GPS threads use this structure
 * to coordinate configuration and mode updates.
 */
static struct system_context ctx = {
    .phototransistor = &pt,
    .soil_moisture = &sm,
    .accelerometer = &accel,
    .accel_range = ACCEL_RANGE,
    .temp_hum = &th,
    .color = &color,
    .gps = &gps,
    .main_sensors_sem = &main_sensors_sem,
    .main_gps_sem = &main_gps_sem,
    .sensors_sem = &sensors_sem,
```

```
    .gps_sem = &gps_sem,
};

/**
 * @brief Shared measurements between threads.
 *
 * The @ref system_measurement structure holds the current sensor readings
 * accessible to all threads.
 */
static struct system_measurement measure = {
    .brightness = ATOMIC_INIT(0),
    .moisture = ATOMIC_INIT(0),
    .accel_x_g = ATOMIC_INIT(0),
    .accel_y_g = ATOMIC_INIT(0),
    .accel_z_g = ATOMIC_INIT(0),
    .temp = ATOMIC_INIT(0),
    .hum = ATOMIC_INIT(0),
    .red = ATOMIC_INIT(0),
    .green = ATOMIC_INIT(0),
    .blue = ATOMIC_INIT(0),
    .clear = ATOMIC_INIT(0),
    .gps_lat = ATOMIC_INIT(0),
    .gps_lon = ATOMIC_INIT(0),
    .gps_alt = ATOMIC_INIT(0),
    .gps_sats = ATOMIC_INIT(0),
    .gps_time = ATOMIC_INIT(0),
};

/**
 * @brief Main data measurement structure.
 */
struct main_measurement {
    system_mode_t mode;
    float light;
    float moisture;
    float lat;
    float lon;
    float alt;
    float x_axis;
    float y_axis;
    float z_axis;
    float hum;
    float temp;
    int sats;
    int time_int;
    int hh, mm, ss;
    float c, r, g, b;
    char ns;
    char ew;
    dom_color_t dom_color;
    atomic_t rgb_flags;
};

/**
 * @brief Main measurement data initialization.
 */
static struct main_measurement main_data = {
    .mode = INITIAL_MODE,
    .light = 0.0f,
    .moisture = 0.0f,
    .lat = 0.0f,
    .lon = 0.0f,
    .alt = 0.0f,
```

```
    .x_axis = 0.0f,
    .y_axis = 0.0f,
    .z_axis = 0.0f,
    .hum = 0.0f,
    .temp = 0.0f,
    .sats = 0,
    .time_int = 0,
    .hh = 0,
    .mm = 0,
    .ss = 0,
    .c = 0,
    .r = 0,
    .b = 0,
    .g = 0,
    .ns = '\0',
    .ew = '\0',
    .dom_color = DOM_RED,
    .rgb_flags = ATOMIC_INIT(0),
};

/**
 * @brief Statistics data structure for mean, max, and min calculations.
 */
struct stats_measurements {
    float temp_mean, temp_max, temp_min;
    float hum_mean, hum_max, hum_min;
    float light_mean, light_max, light_min;
    float moisture_mean, moisture_max, moisture_min;
    float x_axis_max, x_axis_min;
    float y_axis_max, y_axis_min;
    float z_axis_max, z_axis_min;
    int red_count, green_count, blue_count;
    int count;
};

/**
 * @brief Statistics data initialization.
 */
struct stats_measurements stats_data = {0};
```

4.4.4 Initialization

During system startup, all peripherals are initialized sequentially. Each initialization step is validated, and the program execution is halted if any critical component fails to initialize correctly. This approach ensures that the system does not operate in an undefined or partially configured state.

Timers, worker threads, and synchronization primitives are then initialized. Measurement threads for sensors and GPS data acquisition are started, and the system enters its initial operating mode with a visual indication provided by the LEDs.

Listing 24: Initialization function

```
printk("==== Plant Monitoring System ====\n");
printk("System ON (TEST MODE)\n\n");

uint32_t flags = 0;
system_mode_t previous_mode = INITIAL_MODE;
float r_norm = 0.0f, g_norm = 0.0f, b_norm = 0.0f;
int r_duty = 0, g_duty = 0, b_duty = 0, r_value = 0, g_value = 0, b_value = 0;
bool keep_running = true;

/* Initialize peripherals */
if (gps_init(&gps)) {
```

```
        printk("GPS initialization failed - Program stopped\n");
        return -1;
    }
    if (adc_init(&pt)) {
        printk("Phototransistor initialization failed - Program stopped\n");
        return -1;
    }
    if (adc_init(&sm)) {
        printk("Soil moisture sensor initialization failed - Program stopped\n");
        return -1;
    }
    if (accel_init(&accel, ACCEL_RANGE)) {
        printk("Accelerometer initialization failed - Program stopped\n");
        return -1;
    }
    if (temp_hum_init(&th, TEMP_HUM_RESOLUTION)) {
        printk("Temperature/Humidity sensor initialization failed - Program stopped\n");
        return -1;
    }
    if (color_init(&color, COLOR_GAIN, COLOR_INTEGRATION_TIME)) {
        printk("Color sensor initialization failed - Program stopped\n");
        return -1;
    }
    if (led_init(&leds) || led_off(&leds)) {
        printk("LED initialization failed - Program stopped\n");
        return -1;
    }
    if (rgb_led_init(&rgb_leds) || rgb_led_off(&rgb_leds)) {
        printk("RGB LED initialization failed - Program stopped\n");
        return -1;
    }
    if (button_init(&button)) {
        printk("Button initialization failed - Program stopped\n");
        return -1;
    }
    if (button_set_callback(&button, button_isr)) {
        printk("Button callback setup failed - Program stopped\n");
        return -1;
    }

    /* Initialize timers */
    k_timer_init(&main_timer, main_timer_handler, NULL);
    k_timer_init(&rgb_timer, rgb_timer_handler, NULL);
    k_timer_init(&stats_timer, stats_timer_handler, NULL);

    k_timer_start(&main_timer, K_MSEC(TEST_PERIOD), K_MSEC(TEST_PERIOD));
    k_timer_start(&stats_timer, K_MSEC(STATS_TIMER_PERIOD), K_MSEC(STATS_TIMER_PERIOD));

    /* Button handling */
    k_work_init(&button_work, button_work_handler);

    /* Start measurement threads */
    start_sensors_thread(&ctx, &measure);
    start_gps_thread(&ctx, &measure);

    blue(&leds);
```

4.4.5 Modes

The system operates as a finite-state machine with three distinct modes: *Test*, *Normal*, and *Advanced*. Mode transitions are triggered by the user button, while periodic execution within each mode is controlled by a timer.

A semaphore is used to synchronize the main loop with both the main timer and the button interrupt. This mechanism ensures deterministic behavior and prevents race conditions between asynchronous events.

Each mode provides different functionality and board LED feedback, allowing the user to easily identify the current operating state of the system.

```
while (1) {
    switch (main_data.mode) {

        case TEST_MODE:
            blue(&leds);

            ...

            k_sem_take(&main_sem, K_FOREVER);

            break;

        case NORMAL_MODE:
            green(&leds);

            ...

            k_sem_take(&main_sem, K_FOREVER);

            break;

        case ADVANCED_MODE:
            red(&leds);

            ...
                    if (k_sem_take(&main_sem, K_NO_WAIT) == 0) {
                        keep_running = false;
                        break;
                    }
            ...

            break;
    }
}
```

4.4.6 Main timer

The main timer defines the duration of each operating cycle within a given mode. When the timer expires, its handler releases a semaphore that unblocks the main loop, allowing the next iteration of the mode logic to execute.

Listing 25: Main timer handler function

```
static struct k_timer main_timer;

static void main_timer_handler(struct k_timer *timer)
{
    k_sem_give(&main_sem);
}
```

4.4.7 Button

The user button provides manual control over system mode transitions. Button presses are handled through an ISR, which delegates processing to a work queue to avoid executing complex logic in interrupt context.

Each button press advances the system to the next operating mode in a cyclic manner. After updating the mode, the semaphore is released to immediately apply the change in the main execution loop.

Listing 26: Button work handler and ISR functions

```
static void button_work_handler(struct k_work *work)
{
    switch (main_data.mode) {
        case TEST_MODE:
            main_data.mode = NORMAL_MODE;
            printk("\nNORMAL MODE\n");
            break;
        case NORMAL_MODE:
            main_data.mode = ADVANCED_MODE;
            printk("\nADVANCED MODE\n");
            break;
        case ADVANCED_MODE:
            main_data.mode = TEST_MODE;
            printk("\nTEST MODE\n");
            break;
    }

    k_sem_give(&main_sem);
}

static void button_isr(const struct device *dev, struct gpio_callback *cb, uint32_t pins)
{
    if (!gpio_pin_get_dt(&button.spec)) {
        k_work_submit(&button_work);
    }
}
```

4.4.8 TEST MODE

The *Test Mode* is intended for system verification and debugging purposes. In this mode, all sensors are periodically sampled and their raw and processed values are displayed through the serial console. Additionally, the RGB LED provides immediate visual feedback by indicating the dominant color detected by the color sensor.

When entering this mode, the system reconfigures the main timer to operate with the test period and ensures that any previously active RGB timer is stopped. Sensor and GPS acquisition threads are explicitly triggered using semaphores, and the main thread waits until all measurements are available before processing them.

The dominant color is determined by comparing the red, green, and blue channel values, and the corresponding RGB LED color is activated. Execution then blocks on a semaphore, allowing either the main timer or a button event to resume operation.

Listing 27: TEST MODE

```
case TEST_MODE:
    blue(&leds);

    if(previous_mode != TEST_MODE) {
        k_timer_stop(&rgb_timer);
        rgb_led_off(&rgb_leds);
        k_timer_stop(&main_timer);
        k_timer_start(&main_timer, K_MSEC(TEST_PERIOD), K_MSEC(TEST_PERIOD));
        previous_mode = TEST_MODE;
    }

    k_sem_give(ctx.sensors_sem);
    k_sem_give(ctx.gps_sem);
```

```
    k_sem_take(ctx.main_sensors_sem, K_FOREVER);
    k_sem_take(ctx.main_gps_sem, K_FOREVER);

    get_measurements();

    if (main_data.r > main_data.g && main_data.r > main_data.b) {
        rgb_red(&rgb_leds);
        main_data.dom_color = DOM_RED;
    } else if (main_data.g > main_data.r && main_data.g > main_data.b) {
        rgb_green(&rgb_leds);
        main_data.dom_color = DOM_GREEN;
    } else {
        rgb_blue(&rgb_leds);
        main_data.dom_color = DOM_BLUE;
    }

    display_measurements();

    k_sem_take(&main_sem, K_FOREVER);

    break;
```

4.4.9 Get Measurements

This function retrieves the most recent sensor readings from shared atomic variables and converts them into physical units suitable for further processing and display. Scaling factors are applied to raw values in order to obtain meaningful measurements such as percentages, degrees Celsius, meters per second squared, and geographic coordinates.

GPS data is processed to extract latitude, longitude, altitude, satellite count, and time information. Direction indicators (North (N)/South (S) and East (E)/West (W)) are derived from the sign of the coordinates, and the time value is decomposed into hours, minutes, and seconds.

All processed values are stored in the main measurement structure, which serves as the central data container for the current system cycle.

Listing 28: Get measurements function

```
static void get_measurements()
{
    main_data.moisture = atomic_get(&measure.moisture) / 10.0f;

    main_data.light = atomic_get(&measure.brightness) / 10.0f;

    main_data.lat = atomic_get(&measure.gps_lat) / 1e6f;
    main_data.lon = atomic_get(&measure.gps_lon) / 1e6f;
    main_data.alt = atomic_get(&measure.gps_alt) / 100.0f;
    main_data.sats = atomic_get(&measure.gps_sats);
    main_data.time_int = atomic_get(&measure.gps_time);

    main_data.ns = (main_data.lat >= 0) ? 'N' : 'S';
    main_data.ew = (main_data.lon >= 0) ? 'E' : 'W';

    main_data.lat = fabsf(main_data.lat);
    main_data.lon = fabsf(main_data.lon);

    if (main_data.time_int >= 0) {
        main_data.hh = main_data.time_int / 10000;
        main_data.mm = (main_data.time_int / 100) % 100;
        main_data.ss = main_data.time_int % 100;
    } else {
        printk("GPS time: --:--:--\n");
```

```
    }

    main_data.r = atomic_get(&measure.red);
    main_data.g = atomic_get(&measure.green);
    main_data.b = atomic_get(&measure.blue);
    main_data.c = atomic_get(&measure.clear);

    main_data.x_axis = atomic_get(&measure.accel_x_g) / 100.0f;
    main_data.y_axis = atomic_get(&measure.accel_y_g) / 100.0f;
    main_data.z_axis = atomic_get(&measure.accel_z_g) / 100.0f;

    main_data.temp = atomic_get(&measure.temp) / 100.0f;
    main_data.hum = atomic_get(&measure.hum) / 100.0f;
}
```

4.4.10 Display Measurements

This function is responsible for presenting the current system measurements through the serial console. It provides a structured and human-readable output that includes soil moisture, ambient light level, GPS information, color sensor readings, accelerometer values, and temperature and humidity data.

Listing 29: Display measurements function

```
static void display_measurements()
{
    printk("SOIL MOISTURE: %.1f%%\n", (double)main_data.moisture);

    printk("LIGHT: %.1f%%\n", (double)main_data.light);

    printk("GPS: #Sats: %d Lat(UTC): %.6f %c Long(UTC): %.6f %c Altitude: %.0f m GPS time:
        %02d:%02d:%02d\n",
           main_data.sats, (double)main_data.lat, main_data.ns, (double)main_data.lon,
           main_data.ew, (double)main_data.alt, main_data.hh, main_data.mm, main_data.ss);

    printk("COLOR SENSOR: Clear: %.0f Red: %.0f Green: %.0f Blue: %.0f Dominant color: %s
        \n",
           (double)main_data.c, (double)main_data.r, (double)main_data.g,
               (double)main_data.b, dom_color_names[main_data.dom_color]);

    printk("ACCELEROMETER: X_axis: %.2f m/s2, Y_axis: %.2f m/s2, Z_axis: %.2f m/s2 \n",
           (double)main_data.x_axis, (double)main_data.y_axis, (double)main_data.z_axis);

    printk("TEMP/HUM: Temperature: %.1fC, Relative Humidity: %.1f%%\n\n",
           (double)main_data.temp, (double)main_data.hum);
}
```

4.4.11 NORMAL MODE

The *Normal Mode* represents the standard operating state of the system. In this mode, measurements are performed at a lower frequency than in Test Mode to reduce power consumption while still providing continuous monitoring.

Upon entering Normal Mode, the main timer is reconfigured with the normal measurement period, and an additional RGB timer is enabled to handle alarm visualization. Sensor and GPS data are acquired synchronously, and the resulting measurements are processed and validated against predefined limits.

Statistical calculations are updated at each cycle (print each hour), and the current measurements are displayed. The system then waits for the next timer expiration or user interaction.

Listing 30: NORMAL MODE

```
case NORMAL_MODE:
    green(&leds);
    flags = 0;

    if(previous_mode != NORMAL_MODE) {
        k_timer_stop(&main_timer);
        k_timer_start(&main_timer, K_MSEC(NORMAL_PERIOD), K_MSEC(NORMAL_PERIOD));
        k_timer_start(&rgb_timer, K_MSEC(RGB_TIMER_PERIOD), K_MSEC(RGB_TIMER_PERIOD));
        previous_mode = NORMAL_MODE;
    }

    k_sem_give(ctx.sensors_sem);
    k_sem_give(ctx.gps_sem);

    k_sem_take(ctx.main_sensors_sem, K_FOREVER);
    k_sem_take(ctx.main_gps_sem, K_FOREVER);

    get_measurements();

    check_limits(&flags);

    stats_management();

    display_measurements();

    k_sem_take(&main_sem, K_FOREVER);

    break;
```

4.4.12 Check Limits

This subsection describes the mechanism used to validate sensor readings against predefined operational thresholds. Each measurement is compared to its corresponding minimum and maximum values, and any out-of-range condition is clipped to the nearest valid limit.

Whenever a limit violation occurs, a corresponding flag is set using a bitwise representation. These flags are stored atomically and later used to drive the RGB LED alarm system, allowing multiple simultaneous alarm conditions to be represented efficiently.

Listing 31: Check limits function

```
static void check_limit(float *val, float min, float max, uint32_t *flags, uint32_t
    flag_bit)
{
    if (*val < min) {
        *val = min;
        *flags |= flag_bit;
    } else if (*val > max) {
        *val = max;
        *flags |= flag_bit;
    }
}

static void check_limits(uint32_t *flags)
{
    *flags = 0U;

    check_limit(&main_data.temp, TEMP_MIN, TEMP_MAX, flags, FLAG_TEMP);
    check_limit(&main_data.hum, HUM_MIN, HUM_MAX, flags, FLAG_HUM);
    check_limit(&main_data.light, LIGHT_MIN, LIGHT_MAX, flags, FLAG_LIGHT);
    check_limit(&main_data.moisture, MOISTURE_MIN, MOISTURE_MAX, flags, FLAG_MOISTURE);

    check_limit(&main_data.c, COLOR_MIN, COLOR_MAX, flags, FLAG_COLOR);
```

```
    check_limit(&main_data.r, COLOR_MIN, COLOR_MAX, flags, FLAG_COLOR);
    check_limit(&main_data.g, COLOR_MIN, COLOR_MAX, flags, FLAG_COLOR);
    check_limit(&main_data.b, COLOR_MIN, COLOR_MAX, flags, FLAG_COLOR);

    check_limit(&main_data.x_axis, ACCEL_MIN * 9.8f, ACCEL_MAX * 9.8f, flags, FLAG_ACCEL);
    check_limit(&main_data.y_axis, ACCEL_MIN * 9.8f, ACCEL_MAX * 9.8f, flags, FLAG_ACCEL);
    check_limit(&main_data.z_axis, ACCEL_MIN * 9.8f, ACCEL_MAX * 9.8f, flags, FLAG_ACCEL);

    atomic_set(&main_data.rgb_flags, (atomic_val_t)(*flags));
}
```

4.4.13 RGB Alarms

The RGB alarm system provides visual feedback when one or more sensor measurements exceed their allowed ranges. A dedicated timer with a period of 0.5 seconds cyclically updates the RGB LED.

Each active alarm condition is mapped to a specific color. When multiple alarms are present, the LED cycles through the corresponding colors with a period of 0.5 seconds, ensuring that all active warnings are communicated to the user. If no alarms are active, the RGB LED is turned off.

Listing 32: RGB LED alarm timer handler

```
static struct k_timer rgb_timer;

static void rgb_timer_handler(struct k_timer *timer)
{
    static uint8_t color_index = 0;
    uint32_t flags = atomic_get(&main_data.rgb_flags);

    uint8_t colors[6];
    uint8_t count = 0;

    if (flags & FLAG_TEMP)    colors[count++] = 0; // RED
    if (flags & FLAG_HUM)     colors[count++] = 1; // BLUE
    if (flags & FLAG_LIGHT)   colors[count++] = 2; // GREEN
    if (flags & FLAG_MOISTURE) colors[count++] = 3; // CYAN
    if (flags & FLAG_COLOR)   colors[count++] = 4; // WHITE
    if (flags & FLAG_ACCEL)   colors[count++] = 5; // YELLOW

    if (count == 0) {
        rgb_led_off(&rgb_leds);
        color_index = 0;
        return;
    }

    uint8_t color_status = colors[color_index % count];
    color_index++;

    switch (color_status) {
        case 0: rgb_red(&rgb_leds); break;
        case 1: rgb_blue(&rgb_leds); break;
        case 2: rgb_green(&rgb_leds); break;
        case 3: rgb_cyan(&rgb_leds); break;
        case 4: rgb_white(&rgb_leds); break;
        case 5: rgb_yellow(&rgb_leds); break;
        default: rgb_led_off(&rgb_leds); break;
    }
}
```

4.4.14 Stats Management

This subsection describes the statistical processing performed on sensor data during Normal Mode operation. The system maintains running statistics, including mean, maximum, and minimum values, for selected measurements.

The mean values are computed incrementally to minimize memory usage and computational overhead. Maximum and minimum values are updated dynamically as new measurements become available. Additionally, the system tracks the frequency of dominant color detections to determine long-term color trends.

Listing 33: Stats Management

```
static void stats_management()
{
    stats_data.count++;

    mean_calculation();

    max_min_calculation();

    dominant_color_calculation();
}

static void mean_calculation()
{
    if (stats_data.count == 1) {
        stats_data.temp_mean = main_data.temp;
        stats_data.hum_mean = main_data.hum;
        stats_data.light_mean = main_data.light;
        stats_data.moisture_mean = main_data.moisture;
    } else {
        stats_data.temp_mean = ((stats_data.temp_mean * (stats_data.count - 1)) +
            main_data.temp) / stats_data.count;
        stats_data.hum_mean = ((stats_data.hum_mean * (stats_data.count - 1)) +
            main_data.hum) / stats_data.count;
        stats_data.light_mean = ((stats_data.light_mean * (stats_data.count - 1)) +
            main_data.light) / stats_data.count;
        stats_data.moisture_mean = ((stats_data.moisture_mean * (stats_data.count - 1)) +
            main_data.moisture) / stats_data.count;
    }
}

static void max_min_calculation()
{
    if (stats_data.count == 1) {
        // Initialize first values
        stats_data.temp_max = stats_data.temp_min = main_data.temp;
        stats_data.hum_max = stats_data.hum_min = main_data.hum;
        stats_data.light_max = stats_data.light_min = main_data.light;
        stats_data.moisture_max = stats_data.moisture_min = main_data.moisture;

        stats_data.x_axis_max = stats_data.x_axis_min = main_data.x_axis;
        stats_data.y_axis_max = stats_data.y_axis_min = main_data.y_axis;
        stats_data.z_axis_max = stats_data.z_axis_min = main_data.z_axis;
    } else {
        // Temperature
        if (main_data.temp > stats_data.temp_max) stats_data.temp_max = main_data.temp;
        if (main_data.temp < stats_data.temp_min) stats_data.temp_min = main_data.temp;

        // Humidity
        if (main_data.hum > stats_data.hum_max) stats_data.hum_max = main_data.hum;
        if (main_data.hum < stats_data.hum_min) stats_data.hum_min = main_data.hum;
```

```
        // Light
        if (main_data.light > stats_data.light_max) stats_data.light_max = main_data.light;
        if (main_data.light < stats_data.light_min) stats_data.light_min = main_data.light;

        // Moisture
        if (main_data.moisture > stats_data.moisture_max) stats_data.moisture_max =
            main_data.moisture;
        if (main_data.moisture < stats_data.moisture_min) stats_data.moisture_min =
            main_data.moisture;

        // Accelerometer
        if (main_data.x_axis > stats_data.x_axis_max) stats_data.x_axis_max =
            main_data.x_axis;
        if (main_data.x_axis < stats_data.x_axis_min) stats_data.x_axis_min =
            main_data.x_axis;

        if (main_data.y_axis > stats_data.y_axis_max) stats_data.y_axis_max =
            main_data.y_axis;
        if (main_data.y_axis < stats_data.y_axis_min) stats_data.y_axis_min =
            main_data.y_axis;

        if (main_data.z_axis > stats_data.z_axis_max) stats_data.z_axis_max =
            main_data.z_axis;
        if (main_data.z_axis < stats_data.z_axis_min) stats_data.z_axis_min =
            main_data.z_axis;
    }
}

static void dominant_color_calculation()
{
    if (main_data.r > main_data.g && main_data.r > main_data.b) {
        stats_data.red_count++;
    } else if (main_data.g > main_data.r && main_data.g > main_data.b) {
        stats_data.green_count++;
    } else if (main_data.b > main_data.r && main_data.b > main_data.g) {
        stats_data.blue_count++;
    }
}
```

4.4.15 ADVANCED MODE

Explained in section 6.

4.5 Zephyr RTOS

4.5.1 prj_nucleo_wl55jc.conf

The following configuration file (`prj_nucleo_wl55jc.conf`) specifies the system modules required for enabling the serial console, GPIO, ADC, and I2C interfaces within the Zephyr RTOS environment. It also activates several debugging and runtime analysis features, including thread stack initialization, thread information reporting, and the automatic thread analyzer.

Listing 34: prj_nucleo_wl55jc.conf

```
CONFIG_STDOUT_CONSOLE=y
CONFIG_UART_CONSOLE=y
CONFIG_CONSOLE=y
CONFIG_PRINTK=y
CONFIG_CBPRINTF_FP_SUPPORT=y
CONFIG_POLL=y

CONFIG_EVENTS=y
CONFIG_LOG=y

CONFIG_GPIO=y # Enable GPIO

CONFIG_ADC=y # Enable ADC

CONFIG_I2C=y # Enable I2C

CONFIG_SERIAL=y
CONFIG_UART_INTERRUPT_DRIVEN=y # Enable UART interrupt-driven API

CONFIG_INIT_STACKS=y
CONFIG_THREAD_STACK_INFO=y
CONFIG_THREAD_ANALYZER=y
CONFIG_THREAD_ANALYZER_AUTO=y
CONFIG_THREAD_NAME=y
```

4.5.2 nucleo_wl55jc.overlay

Additionally, the DeviceTree overlay file (`nucleo_wl55jc.overlay`) extends the hardware description of the Nucleo-WL55JC board by defining a RGB LED structure implemented through GPIO-controlled LED nodes. Corresponding aliases are included to simplify application-level access to these components. The overlay also configures the Universal Synchronous Asynchronous Receiver-Transmitter (USART)1 peripheral with its associated pin assignments and baud rate, enabling serial communication capabilities required by the console or other UART-based interfaces.

Listing 35: nucleo_wl55jc.overlay

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

/ {
    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
    };


};

&usart1 {
    status = "okay";
    current-speed = <9600>;
    pinctrl-0 = <&usart1_tx_pb6 &usart1_rx_pb7>;
    pinctrl-names = "default";
};
```

4.6 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 768 usage 256 / 1024 (25 %); CPU: 0 %
                     : Total CPU cycles used: 56978
 sensors_thread      : STACK: unused 592 usage 432 / 1024 (42 %); CPU: 0 %
                     : Total CPU cycles used: 698566
 thread_analyzer     : STACK: unused 560 usage 464 / 1024 (45 %); CPU: 0 %
                     : Total CPU cycles used: 119419
 sysworkq            : STACK: unused 808 usage 216 / 1024 (21 %); CPU: 0 %
                     : Total CPU cycles used: 1012
 logging             : STACK: unused 164 usage 604 / 768 (78 %); CPU: 1 %
                     : Total CPU cycles used: 48822096
 idle                : STACK: unused 192 usage 64 / 256 (25 %); CPU: 98 %
                     : Total CPU cycles used: 2830266161
 main                : STACK: unused 544 usage 480 / 1024 (46 %); CPU: 0 %
                     : Total CPU cycles used: 834931
 ISR0                : STACK: unused 1672 usage 376 / 2048 (18 %)
```

Figure 12: Thread stack and CPU usage

It is important to understand the purpose of each thread displayed:

- **gps_thread:** Thread responsible for configuring, reading and parsing GPS data.
- **sensors_thread:** Thread responsible for sensor readings (accelerometer, colour sensor, etc.).
- **thread_analyzer:** Internal diagnostic thread used to collect and report thread metrics such as stack usage. It runs periodically and only consumes CPU during short analysis windows.
- **sysworkq:** The global Zephyr system workqueue. It is used to run small background tasks that do not need their own dedicated thread. Typical examples include executing callbacks.

- **logging:** Internal Zephyr thread in charge of processing log messages.
- **idle:** Lowest-priority thread that runs whenever no other thread is ready. It accounts for the majority of CPU cycles, which is expected and desirable in a low-power sensor system.
- **main:** The initial thread created at system startup.
- **ISR0 (ISR stack):** Not a regular thread, but the shared stack region used by all interrupt service routines.

For each thread, the following metrics are displayed:

- **STACK:** Reports the unused stack space, the amount of stack used, and the total allocated stack size. For example, for `gps_thread`:

  unused 768 B, used 256 B, total 1024 B

  This corresponds to a stack usage of 25%, indicating that the assigned memory is sufficient and no overflow risk is present.

  As a general guideline, **stack usage below 60% is considered safe** in Zephyr, as it leaves enough headroom for context switching, interrupts, and occasional peak loads.

- **CPU:** Shows the percentage of CPU time consumed by each thread. Most application threads such as `gps_thread` or `sensors_thread` show 0% CPU usage because they predominantly sleep while waiting for periodic timers or I/O events.
- **Total CPU cycles used:** Indicates the cumulative processor cycles consumed by each thread since boot.

  Threads like `idle` present extremely large values, which is expected since the idle thread runs whenever no other thread is ready to execute. A high idle count is a positive indicator of energy efficiency.

This is obtained thanks to the following configuration options enabled in `prj_nucleo_wl55jc.conf`, which allow Zephyr to track stack usage, assign human-readable thread names, and automatically generate periodic thread analysis reports:

Listing 36: Thread stack and CPU usage report - prj_nucleo_wl55jc.conf

```
CONFIG_INIT_STACKS=y
CONFIG_THREAD_STACK_INFO=y
CONFIG_THREAD_ANALYZER=y
CONFIG_THREAD_ANALYZER_AUTO=y
CONFIG_THREAD_NAME=y
```

Overall, the reported values confirm that:

- All thread stacks remain within safe usage ranges, with most 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.

4.7 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 13, after linking the executable `zephyr.elf`, the memory report provides the following information:

- **FLASH:** 59.024B used out of 256KB (approximately 22.5%).
- **RAM:** 13.504B used out of 64KB (approximately 20.6%).

This confirms that the firmware comfortably fits within the memory limits of the STM32WL55 microcontroller, leaving sufficient headroom for future improvements or additional functionality.

```
[9/9] Linking C executable zephyr\zephyr.elf
Memory region         Used Size  Region Size  %age Used
           FLASH:       59024 B       256 KB     22.52%
             RAM:       13504 B        64 KB     20.61%
        IDT_LIST:          0 GB        32 KB      0.00%
```

Figure 13: Compilation memory usage report

4.8 Flashing the Firmware onto the STM32WL55

The Figure 14 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 such as:

- **Device:** STM32WLxx.
- **Flash Size:** 256KB.
- **Core:** ARM Cortex-M4.
- **Supply Voltage:** 3.28V.
- **Connection Mode:** Under Reset.

After loading the generated `zephyr.hex` file (57.64KB), the programmer performs the following steps:

1. Erases the internal Flash sectors (0 to 28).
2. Programs the firmware at address `0x08000000`.
3. Verifies the integrity of the written data.
4. Starts the application.

The final message, *"Application is running, Please Hold on..."*, indicates that the microcontroller has successfully been programmed and is now executing the uploaded Zephyr firmware.

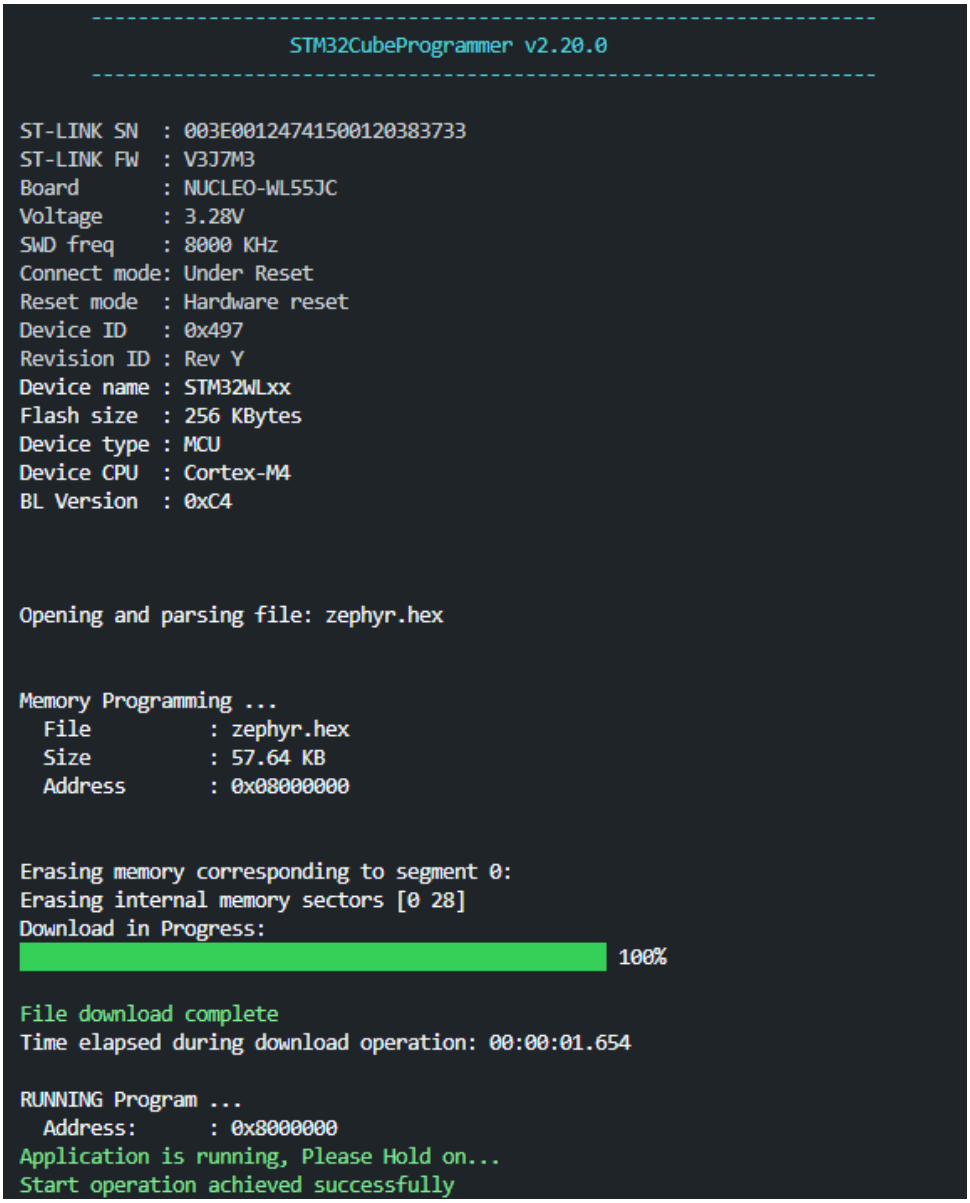

Figure 14: Flashing process using STM32CubeProgrammer

4.9 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/Embedded_IoT/`.

5 Results

This section presents the experimental results obtained during the validation of the system. The tests are divided into unitary tests, which verify the correct operation of individual hardware components, and system-level tests, which evaluate the integrated behavior of the complete system under normal operating conditions.

5.1 Unitary Tests

Unitary tests were performed to validate the correct functionality of each sensor and actuator independently. These tests ensured that individual peripherals were properly configured, responsive, and capable of delivering consistent and reliable measurements before system integration.

5.1.1 Soil Moisture

Figure 15 shows the soil moisture sensor measurements obtained during the unitary test. The sensor responds correctly to variations in moisture level, providing stable and repeatable readings that are consistent with the expected operating range.

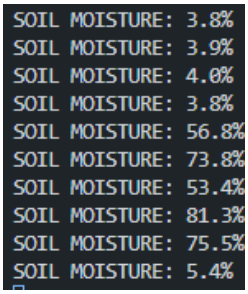

Figure 15: Soil Moisture Measurement

5.1.2 Phototransistor

The phototransistor unitary test, shown in Figure 16, demonstrates the system's ability to measure ambient light intensity. The measured values vary as expected with changes in illumination, confirming correct ADC configuration and signal conditioning.

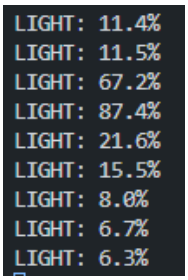

Figure 16: Light Measurement

5.1.3 GPS

Figure 17 presents the output of the GPS module during standalone testing. The module successfully provides geographic coordinates, altitude, satellite count, and time information, validating correct UART communication and data parsing.

```
GPS: #Sats: 6 Lat(UTC): 40.392506 N Long(UTC): 3.731589 W Altitude: 576 m GPS time: 14:51:54
GPS: #Sats: 6 Lat(UTC): 40.392506 N Long(UTC): 3.731589 W Altitude: 576 m GPS time: 14:51:56
GPS: #Sats: 6 Lat(UTC): 40.392502 N Long(UTC): 3.731588 W Altitude: 576 m GPS time: 14:51:58
GPS: #Sats: 6 Lat(UTC): 40.392502 N Long(UTC): 3.731588 W Altitude: 576 m GPS time: 14:52:00
GPS: #Sats: 6 Lat(UTC): 40.392502 N Long(UTC): 3.731586 W Altitude: 576 m GPS time: 14:52:02
GPS: #Sats: 6 Lat(UTC): 40.392502 N Long(UTC): 3.731586 W Altitude: 576 m GPS time: 14:52:04
GPS: #Sats: 6 Lat(UTC): 40.392502 N Long(UTC): 3.731586 W Altitude: 576 m GPS time: 14:52:06
GPS: #Sats: 6 Lat(UTC): 40.392494 N Long(UTC): 3.731586 W Altitude: 576 m GPS time: 14:52:08
GPS: #Sats: 6 Lat(UTC): 40.392494 N Long(UTC): 3.731586 W Altitude: 576 m GPS time: 14:52:10
GPS: #Sats: 6 Lat(UTC): 40.392490 N Long(UTC): 3.731585 W Altitude: 576 m GPS time: 14:52:12
GPS: #Sats: 6 Lat(UTC): 40.392490 N Long(UTC): 3.731585 W Altitude: 576 m GPS time: 14:52:14
GPS: #Sats: 6 Lat(UTC): 40.392483 N Long(UTC): 3.731583 W Altitude: 576 m GPS time: 14:52:16
GPS: #Sats: 6 Lat(UTC): 40.392479 N Long(UTC): 3.731585 W Altitude: 576 m GPS time: 14:52:18
```

Figure 17: GPS Measurement

5.1.4 Accelerometer

The accelerometer test results are shown in Figure 18. The sensor accurately captures acceleration values along the three spatial axes, demonstrating correct configuration of the measurement range and proper I2C communication.

```
ACCELEROMETER: X_axis: -0.20 m/s2, Y_axis: 0.86 m/s2, Z_axis: 9.93 m/s2
ACCELEROMETER: X_axis: -0.18 m/s2, Y_axis: 0.80 m/s2, Z_axis: 10.00 m/s2
ACCELEROMETER: X_axis: -0.22 m/s2, Y_axis: 0.81 m/s2, Z_axis: 9.94 m/s2
ACCELEROMETER: X_axis: -8.65 m/s2, Y_axis: 0.33 m/s2, Z_axis: 4.31 m/s2
ACCELEROMETER: X_axis: -9.55 m/s2, Y_axis: 0.70 m/s2, Z_axis: 1.75 m/s2
ACCELEROMETER: X_axis: -4.19 m/s2, Y_axis: 0.05 m/s2, Z_axis: 8.94 m/s2
ACCELEROMETER: X_axis: 7.93 m/s2, Y_axis: 1.00 m/s2, Z_axis: 7.04 m/s2
ACCELEROMETER: X_axis: 9.38 m/s2, Y_axis: -0.64 m/s2, Z_axis: -1.34 m/s2
ACCELEROMETER: X_axis: 3.13 m/s2, Y_axis: 3.03 m/s2, Z_axis: 9.47 m/s2
ACCELEROMETER: X_axis: 0.66 m/s2, Y_axis: -7.85 m/s2, Z_axis: 5.32 m/s2
ACCELEROMETER: X_axis: 0.37 m/s2, Y_axis: -8.84 m/s2, Z_axis: 3.54 m/s2
ACCELEROMETER: X_axis: 0.84 m/s2, Y_axis: -9.20 m/s2, Z_axis: 2.75 m/s2
ACCELEROMETER: X_axis: -0.40 m/s2, Y_axis: 2.71 m/s2, Z_axis: 8.51 m/s2
```

Figure 18: Accelerometer Measurement

5.1.5 RGB Colour Sensor

Figures 19 and 20 illustrate the performance of the RGB color sensor. The sensor reliably detects variations in color intensity and provides consistent red, green, blue, and clear channel readings, enabling accurate dominant color detection.

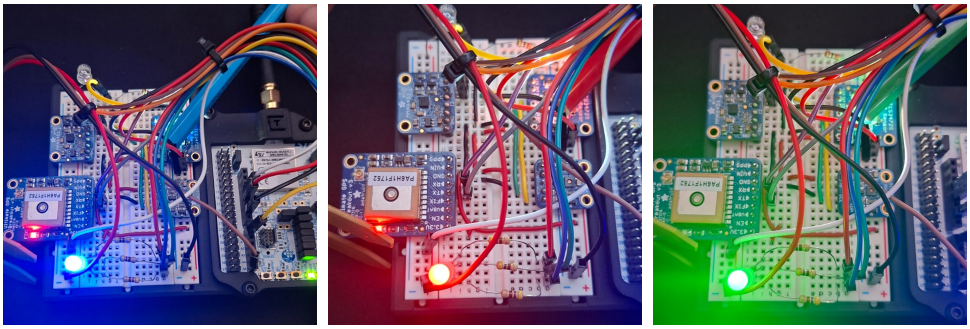

Figure 19: Colours

```
COLOR SENSOR: Clear: 1003 Red: 497 Green: 327 Blue: 281 Dominant color: RED
COLOR SENSOR: Clear: 1001 Red: 497 Green: 326 Blue: 280 Dominant color: RED
COLOR SENSOR: Clear: 994 Red: 492 Green: 322 Blue: 275 Dominant color: RED
COLOR SENSOR: Clear: 21788 Red: 4932 Green: 10544 Blue: 6248 Dominant color: GREEN
COLOR SENSOR: Clear: 6124 Red: 1563 Green: 2938 Blue: 1746 Dominant color: GREEN
COLOR SENSOR: Clear: 1004 Red: 495 Green: 328 Blue: 279 Dominant color: RED
COLOR SENSOR: Clear: 5412 Red: 1257 Green: 2030 Blue: 2326 Dominant color: BLUE
COLOR SENSOR: Clear: 60528 Red: 7177 Green: 24353 Blue: 28802 Dominant color: BLUE
COLOR SENSOR: Clear: 1018 Red: 500 Green: 331 Blue: 289 Dominant color: RED
```

Figure 20: Color Measurement

5.1.6 Temperature and Humidity Sensor

Figure 21 shows the temperature and humidity measurements obtained during testing. The sensor delivers stable and coherent values, confirming the correct resolution configuration and reliable environmental sensing.

```
TEMP/HUM: Temperature: 20.7C, Relative Humidity: 78.9%
TEMP/HUM: Temperature: 20.7C, Relative Humidity: 78.9%
TEMP/HUM: Temperature: 20.7C, Relative Humidity: 78.9%
TEMP/HUM: Temperature: 20.7C, Relative Humidity: 78.6%
TEMP/HUM: Temperature: 22.7C, Relative Humidity: 96.9%
```

Figure 21: Temperature and Humidity Measurement

5.1.7 RGB LED

The RGB LED unitary test, shown in Figure 22, verifies the correct control of individual color channels. Each color can be independently activated, confirming proper GPIO configuration and correct LED driver operation.

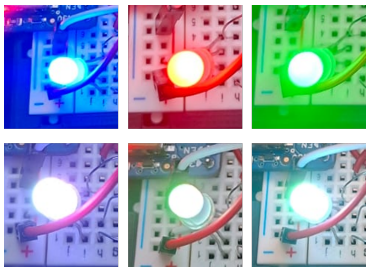

Figure 22: RGB LED

5.1.8 Board LEDs

Figure 23 demonstrates the correct operation of the on-board LEDs. These indicators are used to provide visual feedback regarding system state and operating mode, and their correct behavior confirms proper GPIO handling.

Figure 23: Board LEDs

5.1.9 User button

The user button test, shown in Figure 24, validates correct detection of button presses and reliable interrupt handling. The button is used as the primary user interface element for changing system operating modes.

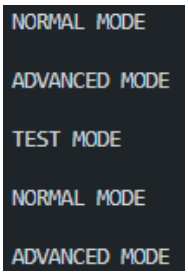

Figure 24: User Button

5.2 System Tests

System-level tests were conducted to evaluate the integrated behavior of all system components working together. These tests verify correct initialization, synchronized data acquisition, mode transitions, and alarm signaling.

5.2.1 Initialization

Figure 25 shows the system output during the initialization phase. All peripherals are correctly initialized, and the system reports its startup state, confirming that the hardware and software components are properly configured before entering normal operation.

```
*** Booting Zephyr OS build v4.2.0-3924-gccaaaabba261 ***
==== Plant Monitoring System ====
System ON (TEST MODE)

[GPS] - Initializing GPS UART...
[GPS] - GPS initialized successfully
[ADC] - Initializing ADC device adc@40012400...
[ADC] - ADC device adc@40012400 initialized successfully
[ADC] - Initializing ADC device adc@40012400...
[ADC] - ADC device adc@40012400 initialized successfully
[ACCEL] - Initializing ACCEL...
ACCEL detected at 0x1D
[ACCEL] - Accelerometer initialized successfully with range 0G
[TEMP_HUM] - Initializing Temp and Hum sensor...
Thread analyze:
 thread_analyzer     : STACK: unused 752 usage 272 / 1024 (26 %); CPU: 0 %
                     : Total CPU cycles used: 9298
 sysworkq            : STACK: unused 808 usage 216 / 1024 (21 %); CPU: 0 %
                     : Total CPU cycles used: 1016
 logging             : STACK: unused 264 usage 504 / 768 (65 %); CPU: 89 %
                     : Total CPU cycles used: 980366
 idle                : STACK: unused 208 usage 48 / 256 (18 %); CPU: 0 %
                     : Total CPU cycles used: 0
 main                : STACK: unused 544 usage 480 / 1024 (46 %); CPU: 7 %
                     : Total CPU cycles used: 79994
 ISR0                : STACK: unused 1816 usage 232 / 2048 (11 %)
[TEMP_HUM] - Resolution set successfully (0x00)
[TEMP_HUM] - Initialization complete
[COLOR] - Initializing Color sensor...
[COLOR] - Color sensor initialized successfully
[LED] - Initializing BOARD LEDs...
[LED] - BOARD LEDs initialized successfully
[RGB LED] - Initializing RGB LED...
[RGB LED] - RGB LED initialized successfully
[USER BUTTON] - Initializing user button...
[USER BUTTON] - User button initialized successfully (edge-interrupt mode)
```

Figure 25: Initialization

5.2.2 Measurements

The measurements test, illustrated in Figure 26, demonstrates the system's ability to acquire, process, and display data from all sensors simultaneously. The results confirm synchronized operation of sensor and GPS threads.

```
SOIL MOISTURE: 7.7%
LIGHT: 9.0%
GPS: #Sats: 5 Lat(UTC): 40.392586 N Long(UTC): 3.731731 W Altitude: 587 m GPS time: 14:57:19
COLOR SENSOR: Clear: 853 Red: 419 Green: 263 Blue: 223 Dominant color: RED
ACCELEROMETER: X_axis: 11.21 m/s2, Y_axis: 6.35 m/s2, Z_axis: 9.55 m/s2
TEMP/HUM: Temperature: 19.5C, Relative Humidity: 83.4%

SOIL MOISTURE: 7.6%
LIGHT: 8.7%
GPS: #Sats: 5 Lat(UTC): 40.392578 N Long(UTC): 3.731744 W Altitude: 587 m GPS time: 14:57:21
COLOR SENSOR: Clear: 859 Red: 421 Green: 266 Blue: 225 Dominant color: RED
ACCELEROMETER: X_axis: 11.21 m/s2, Y_axis: 6.35 m/s2, Z_axis: 9.55 m/s2
TEMP/HUM: Temperature: 19.5C, Relative Humidity: 83.6%
```

Figure 26: Measurements

5.2.3 Mode Changes

Figure 27 presents the system behavior during mode transitions. Each button press correctly advances the system to the next operating mode, and the corresponding visual and textual feedback confirms proper state management.

```
TEST MODE
SOIL MOISTURE: 4.3%
LIGHT: 7.5%
GPS: #Sats: 0 Lat(UTC): 0.000000 N Long(UTC): 0.000000 E Altitude: 0 m GPS time: 01:02:59
COLOR SENSOR: Clear: 1304 Red: 617 Green: 451 Blue: 393 Dominant color: RED
ACCELEROMETER: X_axis: -0.31 m/s2, Y_axis: 0.84 m/s2, Z_axis: 9.77 m/s2
TEMP/HUM: Temperature: 19.2C, Relative Humidity: 75.9%

SOIL MOISTURE: 4.3%
LIGHT: 7.4%
GPS: #Sats: 0 Lat(UTC): 0.000000 N Long(UTC): 0.000000 E Altitude: 0 m GPS time: 01:03:01
COLOR SENSOR: Clear: 1299 Red: 615 Green: 449 Blue: 391 Dominant color: RED
ACCELEROMETER: X_axis: -0.31 m/s2, Y_axis: 0.90 m/s2, Z_axis: 9.97 m/s2
TEMP/HUM: Temperature: 19.2C, Relative Humidity: 75.8%


NORMAL MODE
SOIL MOISTURE: 4.2%
LIGHT: 7.3%
GPS: #Sats: 0 Lat(UTC): 0.000000 N Long(UTC): 0.000000 E Altitude: 0 m GPS time: 01:03:02
COLOR SENSOR: Clear: 1299 Red: 615 Green: 449 Blue: 391 Dominant color: RED
ACCELEROMETER: X_axis: -0.26 m/s2, Y_axis: 0.94 m/s2, Z_axis: 9.83 m/s2
TEMP/HUM: Temperature: 19.2C, Relative Humidity: 75.0%


ADVANCED MODE
SOIL MOISTURE: 4.3%
LIGHT: 7.5%
GPS: #Sats: 0 Lat(UTC): 0.000000 N Long(UTC): 0.000000 E Altitude: 0 m GPS time: 01:03:03
COLOR SENSOR: Clear: 1301 Red: 614 Green: 450 Blue: 394 Dominant color: RED
ACCELEROMETER: X_axis: -0.30 m/s2, Y_axis: 0.80 m/s2, Z_axis: 9.84 m/s2
TEMP/HUM: Temperature: 19.2C, Relative Humidity: 75.7%

NORMALIZED COLOR VALUES: R: 47.19%, G: 34.59%, B: 30.28%


TEST MODE
SOIL MOISTURE: 5.0%
LIGHT: 8.0%
GPS: #Sats: 0 Lat(UTC): 0.000000 N Long(UTC): 0.000000 E Altitude: 0 m GPS time: 01:03:05
COLOR SENSOR: Clear: 1300 Red: 615 Green: 450 Blue: 391 Dominant color: RED
ACCELEROMETER: X_axis: 2.26 m/s2, Y_axis: 0.32 m/s2, Z_axis: 10.03 m/s2
TEMP/HUM: Temperature: 19.2C, Relative Humidity: 75.6%
```

Figure 27: Mode Changes

5.2.4 RGB Dominant color (TEST MODE)

Figure 28 illustrates dominant color detection in Test Mode. The RGB LED accurately reflects the color channel with the highest intensity, validating both color sensor measurements and the dominant color selection logic.

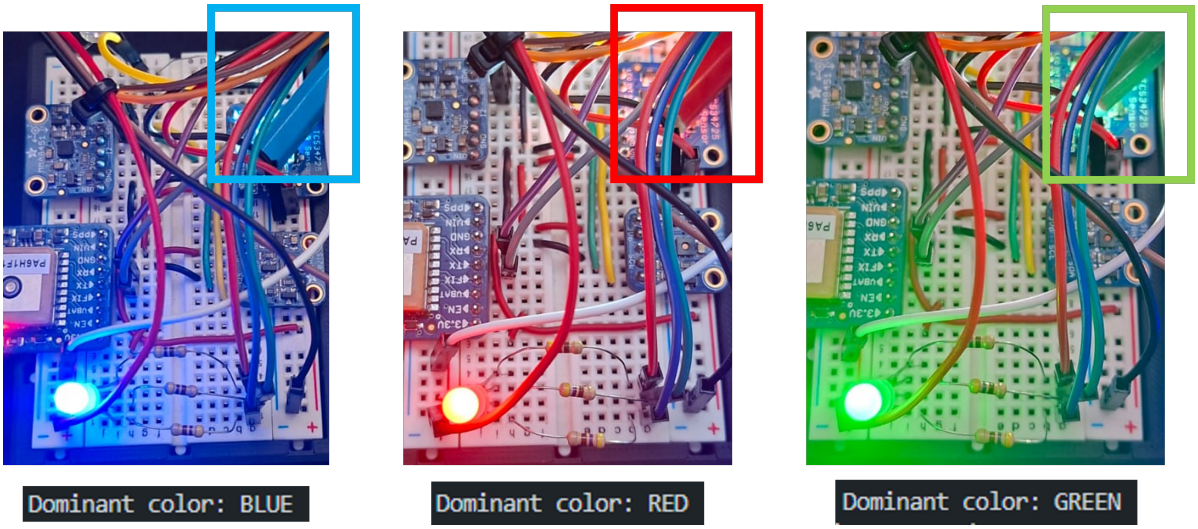

Figure 28: RGB Dominant Color (TEST MODE)

5.2.5 Limits and RGB Alarms (NORMAL MODE)

In Normal Mode, sensor measurements are continuously validated against predefined limits. When a measurement exceeds its allowed range, the corresponding alarm is activated and displayed through the RGB LED. Multiple simultaneous alarms are handled by cyclic color indication, ensuring clear and intuitive visual feedback to the user.

For this example, the maximum temperature is set to 10C, while the minimum soil moisture is set to 10%. As shown in Figure 29, the alarms are triggered when the values exceed these limits. In this case, the temperature is below than 10C (red alarm) and the relative humidity is higher than 75% (blue alarm). Later (second 16), when the soil moisture drops below 10%, the cyan alarm is also activated. These values are clipped to the nearest limit.

Video demonstration (click here)

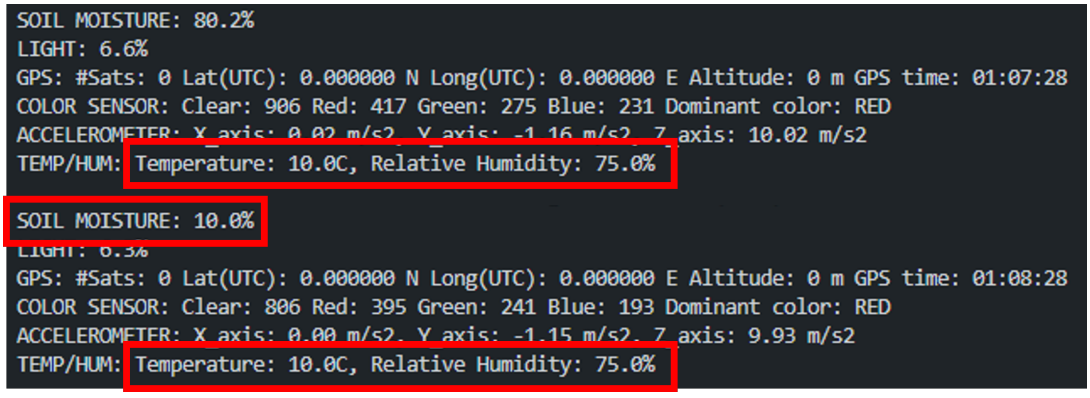

Figure 29: Limits and RGB Alarms (NORMAL MODE)

5.2.6 Statistics Report (NORMAL MODE)

Figure 30 presents the statistical report generated during Normal Mode operation. The system computes and displays mean, minimum, and maximum values for selected measurements, providing a concise

summary of environmental conditions over time.

```
--- STATS REPORT ---
Temperature: Mean: 19.54 C, Max: 19.58 C, Min: 19.51 C
Humidity: Mean: 75.00 %, Max: 75.00 %, Min: 75.00 %
Light: Mean: 7.70 %, Max: 8.40 %, Min: 7.40 %
Soil Moisture: Mean: 4.42 %, Max: 5.00 %, Min: 4.10 %
Acceleration X-axis: Max: -0.29 m/s2, Min: -0.32 m/s2
Acceleration Y-axis: Max: 0.89 m/s2, Min: 0.75 m/s2
Acceleration Z-axis: Max: 9.91 m/s2, Min: 9.79 m/s2
Dominant Color Detected: RED (6 times)
--------------------
```

Figure 30: Measurements Statistics

5.2.7 ADVANCED MODE

The results of the ADVANCED MODE are presented in subsection 6.3.

5.3 External Tests

Both CPPcheck and Valgrind are executed through GitHub Actions as part of the continuous integration pipeline. These external tools are used to perform static and dynamic analysis in order to improve code quality and detect potential issues early in the development process. Due to the specific characteristics of the Zephyr RTOS environment and the target architecture, certain limitations are encountered during their execution.

5.3.1 CPPcheck

CPPcheck is used to perform static code analysis on the C/C++ modules of the project. As shown in Figure 31, the reported warnings are limited and expected.

```
11/12 files checked 90% done
Checking src/sensors_thread.c ...
src/sensors_thread.c:20:0: information: Include file: <zephyr/kernel.h> not found. Please note: Cppcheck does not
#include <zephyr/kernel.h>
^
src/sensors_thread.c:21:0: information: Include file: <zephyr/sys/printk.h> not found. Please note: Cppcheck does
#include <zephyr/sys/printk.h>
^
12/12 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) {
^
src/sensors/i2c/color.c:107:0: style: The function 'color_sleep' is never used. [unusedFunction]
int color_sleep(const struct i2c_dt_spec *dev)
^
src/sensors/i2c/temp_hum.c:120:0: style: The function 'temp_hum_read_temperature' is never used. [unusedFunction]
int temp_hum_read_temperature(const struct i2c_dt_spec *dev, float *temperature)
^
src/sensors/led/board_led.c:83:0: style: The function 'led_on' is never used. [unusedFunction]
int led_on(struct bus_led *led) { return led_write(led, 0x7); }
^
src/sensors/led/board_led.c:127:0: style: The function 'red_green' is never used. [unusedFunction]
int red_green(struct bus_led *led) { return led_write(led, 0x3); }
```

Figure 31: CPPcheck

Most of the messages indicate that Zephyr-specific headers and libraries cannot be found. This is caused by the fact that the GitHub Actions environment does not provide a complete Zephyr build context, and CPPcheck is executed without access to the full toolchain and board support packages required by the operating system.

Additionally, CPPcheck reports several unused functions. This behavior is considered acceptable, as the analyzed modules are implemented as libraries, not directly for the this specific system.

No critical errors or defects were detected during this analysis, and the reported warnings do not indicate functional issues in the codebase.

5.3.2 Valgrind

Valgrind was configured with the objective of performing dynamic memory analysis. However, as illustrated in Figure 32, the execution was not successful.

This limitation is primarily due to the target platform. The project is based on Zephyr RTOS and targets an ARM architecture, while Valgrind is designed to run on Linux-based systems and does not support direct execution on embedded ARM targets without a compatible operating system.

Moreover, multiple attempts were made to build and execute the Zephyr application within the GitHub Actions environment. These attempts were unsuccessful due to the complexity of the Zephyr build process, which requires a specific toolchain, hardware definitions, and board-level configuration that are not easily supported or emulated in a continuous integration environment.

As a result, Valgrind could not be effectively applied to this project.

```
-- Configuring incomplete, errors occurred!
FATAL ERROR: command exited with status 1: /usr/local/bin/cmake -DWEST_PYTHON=/usr/bin/python3 -B/home/runner/work/Embedded_IoT/Embedded_IoT/plant_monitoring_system/build_valgrind -GNinja -DBOARD=native_sim -
S/home/runner/work/Embedded_IoT/Embedded_IoT/plant_monitoring_system
Error: Process completed with exit code 1.
```

Figure 32: Valgrind

6 Advanced Specifications Implemented

6.1 Problem Statement

The objective of the *ADVANCED* mode is to extend the behaviour of the baseline system so that the RGB LED reproduces, the colour measured by the RGB sensor. Unlike the *NORMAL* mode, where the LED output follows predefined patterns, the *ADVANCED* mode must directly map the sensor's RGB data to the LED output using the same relative chromatic composition.

Since the LED pin assignment cannot be modified, the implementation must simulate PWM in software in order to control the LED brightness according to the sensor readings. All other functional parameters must remain identical to those of the *NORMAL* mode.

6.2 Implementation

To reproduce the detected colour, the raw RGB values obtained from the sensor are normalised with respect to the clear channel. This ensures that the emitted light preserves the chromatic ratios of the measured colour independently of the absolute illumination level.

A software-based PWM mechanism is implemented to generate duty cycles proportional to the normalised RGB components. The PWM period is discretised into fixed-size steps, and during each step the LED channels are switched on or off based on whether the current time index is below the computed duty threshold. This emulates real PWM behaviour while keeping the original hardware configuration unchanged.

The following code fragment corresponds to the execution flow in *ADVANCED* mode, after acquiring and displaying the colour measurements:

Listing 37: Software-based PWM implementation for ADVANCED mode

```
#define PWM_STEP  1                          /**< PWM step in milliseconds. */
#define PWM_PERIOD 15                        /**< PWM period in milliseconds. */
#define PWM_STEPS (PWM_PERIOD / PWM_STEP) /**< Number of PWM steps per period. */

case ADVANCED_MODE:
    red(&leds);

    if(previous_mode != ADVANCED_MODE) {
        k_timer_stop(&rgb_timer);
        rgb_led_off(&rgb_leds);

        k_timer_stop(&main_timer);
        k_timer_start(&main_timer, K_MSEC(NORMAL_PERIOD), K_MSEC(NORMAL_PERIOD));
        previous_mode = ADVANCED_MODE;
    }

    k_sem_give(ctx.sensors_sem);
    k_sem_give(ctx.gps_sem);

    k_sem_take(ctx.main_sensors_sem, K_FOREVER);
    k_sem_take(ctx.main_gps_sem, K_FOREVER);

    get_measurements();

    display_measurements();

    if (main_data.c <= 0.0f) {
        printk("[WARN] - Color clear channel == 0\n");
        r_norm = g_norm = b_norm = 0.0f;
    } else {
        r_norm = (main_data.r / main_data.c) * 100.0f;
        g_norm = (main_data.g / main_data.c) * 100.0f;
        b_norm = (main_data.b / main_data.c) * 100.0f;
    }
```

```
        printk("NORMALIZED COLOR VALUES: R: %.2f%%, G: %.2f%%, B: %.2f%%\n\n",
               (double)r_norm, (double)g_norm, (double)b_norm);

        r_duty = (int)(r_norm * PWM_STEPS / 100);
        g_duty = (int)(g_norm * PWM_STEPS / 100);
        b_duty = (int)(b_norm * PWM_STEPS / 100);

        keep_running = true;
        while (keep_running) {
            for (int t = 0; t < PWM_PERIOD; t += PWM_STEP) {
                r_value = (t < r_duty) ? 1 : 0;
                g_value = (t < g_duty) ? 1 : 0;
                b_value = (t < b_duty) ? 1 : 0;

                rgb_led_pwm_step(&rgb_leds, r_value, g_value, b_value);
                k_sleep(K_MSEC(PWM_STEP));

                if (k_sem_take(&main_sem, K_NO_WAIT) == 0) {
                    keep_running = false;
                    break;
                }
            }
        }
```

During development, several timing constraints were encountered when attempting to reduce the PWM step below the millisecond scale. The Zephyr scheduler, together with the hardware capabilities of the target board, prevented reliable delays shorter than one millisecond when using `k_sleep()`.

Alternative approaches were evaluated, including the use of `k_busy_wait()` to achieve microsecond-level blocking delays. However, the board exhibited unstable behaviour and noticeable performance degradation when executing busy-wait loops within the real-time colour reproduction cycle. Consequently, the software PWM design was constrained to millisecond-resolution timing.

6.2.1 Macro definitions and timing parameters

Listing 38: Macros for *ADVANCED* mode

```
#define PWM_STEP  1                          /**< PWM step in milliseconds. */
#define PWM_PERIOD 15                        /**< PWM period in milliseconds. */
#define PWM_STEPS (PWM_PERIOD / PWM_STEP) /**< Number of PWM steps per period. */
```

These macros define the temporal discretisation used by the software PWM:

- `PWM_STEP` specifies the length of a single PWM step in milliseconds. Each step is one atomic time-slot in which the LED channels are either ON or OFF.
- `PWM_PERIOD` is the total duration of one PWM cycle in milliseconds. The perceived brightness is determined by the fraction of this period during which a channel is ON.
- `PWM_STEPS` is the number of discrete steps per PWM period (computed as `PWM_PERIOD / PWM_STEP`). It is used to convert percentage-based normalised colour values into integer duty counts.

This configuration intentionally uses millisecond resolution due to the underlying RTOS scheduler and board timing limitations discussed previously.

6.2.2 Sensor clear-channel safety check and normalisation

Listing 39: Color normalisation based on clear channel

```
if (main_data.c <= 0.0f) {
    printk("[WARN] - Color clear channel == 0\n");
    r_norm = g_norm = b_norm = 0.0f;
```

```
} else {
    r_norm = (main_data.r / main_data.c) * 100.0f;
    g_norm = (main_data.g / main_data.c) * 100.0f;
    b_norm = (main_data.b / main_data.c) * 100.0f;
}
```

Explanation:

- `main_data.c` denotes the clear (ambient) channel returned by the colour sensor. A zero or negative value indicates an invalid or saturated measurement.
- The **safety check** prevents division by zero by forcing the normalised channels to 0% and emitting a warning via `printk` if the clear channel is non-positive.
- When valid, each raw channel (`r`, `g`, `b`) is normalised by the clear channel and converted to percentage units by multiplying by 100. This preserves the chromatic ratios while removing absolute intensity dependence.

6.2.3 Diagnostic logging

Listing 40: Diagnostic logging of normalised color values

```
printk("NORMALIZED COLOR VALUES: R: %.2f%%, G: %.2f%%, B: %.2f%%\n\n",
       (double)r_norm, (double)g_norm, (double)b_norm);
```

Explanation:

- This log statement prints the computed normalised percentages for each channel. It serves as a diagnostic aid during development and in-field debugging to verify that sensor readings and normalisation behave as expected.
- Casting to `double` is used to satisfy the `printk` format specifier and to ensure consistent formatting across platforms.

6.2.4 Duty-cycle computation

Listing 41: Computation of duty counts from normalised percentages

```
r_duty = (int)(r_norm * PWM_STEPS / 100);
g_duty = (int)(g_norm * PWM_STEPS / 100);
b_duty = (int)(b_norm * PWM_STEPS / 100);
```

Explanation:

- The normalised percentages are converted into integer *duty counts* that range from 0 to `PWM_STEPS`.
- Example: with `PWM_STEPS = 15`, a normalised red value of 50% results in `r_duty = 7` (approximately half of the period steps ON).
- Integer truncation is acceptable here because the PWM resolution is limited by `PWM_STEPS`; if higher fidelity is required, increase `PWM_STEPS` by reducing `PWM_STEP`, subject to the timing limits of the platform.

6.2.5 Main PWM loop and per-step evaluation

Listing 42: Main software PWM loop

```
keep_running = true;
while (keep_running) {
    for (int t = 0; t < PWM_PERIOD; t += PWM_STEP) {
        r_value = (t < r_duty) ? 1 : 0;
        g_value = (t < g_duty) ? 1 : 0;
        b_value = (t < b_duty) ? 1 : 0;
```

```
            rgb_led_pwm_step(&rgb_leds, r_value, g_value, b_value);
            k_sleep(K_MSEC(PWM_STEP));

            if (k_sem_take(&main_sem, K_NO_WAIT) == 0) {
                keep_running = false;
                break;
            }
        }
    }
```

Step-by-step explanation:

1. `keep_running` controls the outer loop that maintains continuous PWM operation until an external condition requests termination.
2. The inner `for` loop iterates over the PWM period in increments of `PWM_STEP`. The loop variable `t` effectively indexes the current step within the period.
3. For each step, the boolean channel values (`r_value`, `g_value`, `b_value`) are computed by comparing the current step index `t` with the corresponding duty count. If the index is strictly less than the duty count, the channel is considered ON for that step.
4. `rgb_led_pwm_step(...)` is the hardware abstraction that applies the computed ON/OFF values to the LED pins. It is expected to be non-blocking and to update GPIO (or LED driver) outputs accordingly.
5. `k_sleep(K_MSEC(PWM_STEP))` yields the CPU for the duration of one step. This implements the time base for the PWM emulation while allowing other RTOS threads to run.
6. After sleeping, the loop checks `k_sem_take(&main_sem, K_NO_WAIT)` to determine whether a semaphore has been signalled. If the semaphore is available, the loop exits gracefully. The semaphore is triggered either when the user presses the button or when the 30-second timeout associated with each *ADVANCED* mode cycle elapses.

6.3 Result

The software-based PWM strategy successfully reproduces the colour detected by the RGB sensor. The LED output maintains the chromatic proportions observed in the ambient light, while ensuring smooth transitions and stable operation. The colour is not exact since the PWM resolution is limited to 1 ms steps and the light reflection on the measured object affects the perceived colour, but the representation is sufficiently precise for visual indication applications.

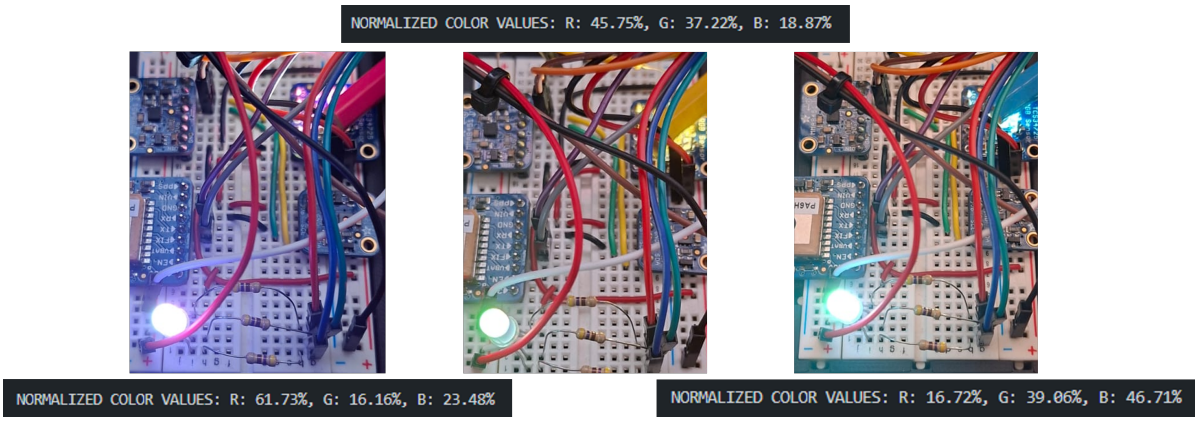

Figure 33: PWM emulated for *ADVANCED* mode

The system preserves full compatibility with the existing hardware configuration and maintains identical behaviour to the *NORMAL* mode regarding synchronisation, timing, and user interaction.

7 Conclusions and Future Works

7.1 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, peripheral management, and real-time operation.

All mandatory system requirements were fulfilled, including periodic sensor acquisition, multi-mode operation (Test, Normal, and Advanced modes), statistical processing, GPS integration, and visual feedback through an RGB LED. The software architecture was carefully structured into modular drivers, dedicated threads, and synchronized shared resources, ensuring scalability, maintainability, and clarity of operation.

The use of Zephyr RTOS enabled efficient thread management, synchronization, and timing control, while also facilitating debugging, logging, and performance analysis. Extensive unit, system, and external testing confirmed the stability, correctness, and robustness of the implementation under different operational conditions.

Overall, the project demonstrates a full embedded IoT systems development lifecycle, from requirements analysis and hardware integration to software design, validation, and documentation. The resulting platform provides a solid foundation for more advanced IoT monitoring applications in agriculture, environmental sensing, and smart systems.

7.2 Future Work

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

- **Hardware-based PWM generation**: The current implementation emulates PWM behavior in software for RGB LED intensity control. A natural improvement would be to use dedicated hardware timers to generate PWM signals. This would reduce CPU load, improve timing accuracy, and provide smoother and more precise brightness control.
- **Improved thread synchronization mechanisms**: The system currently relies on semaphores for thread synchronization. Future versions could explore the use of event-based synchronization primitives (such as Zephyr events or message queues) to achieve more control, improved determinism, and clearer synchronization semantics between threads.
- **Power optimization**: Additional low-power techniques could be implemented, such as deeper sleep states, dynamic sensor activation, and adaptive sampling rates based on environmental conditions.
- **Wireless communication integration**: Leveraging the STM32WL55JC's integrated sub-GHz radio to transmit sensor data wirelessly (e.g., using LoRa or other LPWAN protocols) would significantly enhance the system's applicability in real-world deployments.
- **Data logging and cloud integration**: Persistent storage of measurements and integration with cloud platforms or dashboards would enable long-term analysis, remote monitoring, and predictive analytics.
- **Advanced data processing**: The addition of anomaly detection, trend analysis, or machine learning techniques could further improve plant health assessment and decision-making capabilities.

These potential enhancements would build upon the solid architecture developed in this project, transforming the system into a more powerful, efficient, and scalable IoT monitoring solution.

8 Bibliography

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

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

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

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

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

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

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

[8] "Curso: Embedded platforms and communications for iot — MOODLE UPM - OFICIALES 25-26," Accessed: Nov. 24, 2025. [Online]. Available: https://moodle.upm.es/titulaciones/oficiales/course/view.php?id=3568.

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

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/Embedded_IoT`.

A.2 GitHub Pages - Documentation

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

A.3 GitHub Action - Workflow

Listing 43: GitHub Action workflow

```
name: Cppcheck, Zephyr Build, Valgrind and Doxygen

on:
  push:
    branches: [ main ]
  workflow_dispatch:

env:
  ZEPHYR_BASE: ${{ github.workspace }}/zephyrproject/zephyr
  PROJECT_DIR: plant_monitoring_system

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

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

      - name: Install system dependencies
        run: |
          sudo apt-get update
          sudo apt-get install -y \
            python3-dev \
            python3-pip \
            python3-venv \
            cmake \
            ninja-build \
            gcc \
            g++ \
            valgrind \
            device-tree-compiler
```

```
    - name: Install West and Zephyr dependencies
      run: |
        pip3 install west
        pip3 install -r
            https://raw.githubusercontent.com/zephyrproject-rtos/zephyr/main/scripts/requirements.txt

    - name: Set up Zephyr environment
      run: |
        west init zephyrproject
        cd zephyrproject
        west update
        west zephyr-export

    - name: List available boards to verify native_sim
      run: |
        cd zephyrproject/zephyr
        west boards | grep native

    - name: Build project for native_sim (Valgrind compatible)
      run: |
        cd $PROJECT_DIR
        source $ZEPHYR_BASE/zephyr-env.sh
        west build -b native_sim --build-dir build_valgrind --pristine always

    - name: Find the generated executable
      run: |
        cd $PROJECT_DIR/build_valgrind/zephyr
        find . -name "zephyr*" -type f -executable
        ls -la

    - name: Upload executable for Valgrind
      uses: actions/upload-artifact@v4
      with:
        name: zephyr-native-executable
        path: ${{ env.PROJECT_DIR }}/build_valgrind/zephyr/zephyr.exe
        retention-days: 7

  valgrind:
    runs-on: ubuntu-latest
    needs: build
    steps:
      - name: Install Valgrind
        run: sudo apt-get update && sudo apt-get install -y valgrind

      - name: Download native executable artifact
        uses: actions/download-artifact@v4
        with:
          name: zephyr-native-executable
          path: valgrind_test/

      - name: Make executable
        run: chmod +x valgrind_test/zephyr.exe

      - name: Run Valgrind memory analysis
        run: |
          echo "=== Running Valgrind memory check ==="
          timeout 30s valgrind --leak-check=full --track-origins=yes --show-leak-kinds=all
              valgrind_test/zephyr.exe || echo "Valgrind completed or was terminated"

      - name: Run basic Valgrind check
        run: |
          echo "=== Running basic Valgrind ==="
```

```
        timeout 30s valgrind valgrind_test/zephyr.exe || echo "Valgrind completed or was
            terminated"

  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
```