

Plant Monitoring System

Embedded Platforms and Communications for IoT

Alejandro Botas Bárcena
Estela Mora Barba

MIoT 2025-2026

Contents

List of Contents	i
List of Figures	iii
List of Tables	iii
List of Listings	iii
Acronyms	v
1 Overview and Introduction	1
1.1 Document Overview	1
1.2 Project Introduction	1
1.3 Summary of the Work Done	1
1.3.1 Requirements Analysis	1
1.3.2 Hardware Integration	1
1.3.3 Software Development	2
1.3.4 System Testing and Validation	2
1.3.5 Final Deliverables	2
2 Specifications	3
2.1 Specifications required	3
2.1.1 Hardware	3
2.1.2 Software	3
2.1.3 System Requirements	3
2.2 Additional specifications implemented	4
3 Hardware Analysis	5
3.1 Block diagram	5
3.2 Interfaces of the system	5
3.3 Communication Interfaces used in the system	7
3.3.1 Analog-to-Digital Converter (ADC)	7
3.3.2 I2C Bus	7
3.3.3 UART Interface	7
3.3.4 GPIO Digital Pins	7
3.3.5 Power Interfaces	8
3.4 Hardware devices	8
3.4.1 STM32WL55JC microcontroller	8
3.4.2 RGB LED and 470 Ohm resistors	8
3.4.3 HW5P-1 Phototransistor and 1k Ohm resistor	9
3.4.4 SEN-13322 Soil Moisture Sensor	9
3.4.5 Si7021 Temperature and Humidity Sensor	10
3.4.6 TCS34725 Colour Sensor	10
3.4.7 MMA8451Q Accelerometer	11
3.4.8 Adafruit Ultimate GPS Breakout v3	11
4 Software Organization	13
4.1 Description of the Global Software Architecture	13
4.1.1 Detailed System Behaviour per Operating Mode	13
4.1.2 Mode Transition Mechanism	16
4.1.3 Semaphore Synchronisation and Thread Interactions	17
4.1.4 Atomic Shared Measurement Structure	17
4.1.5 Peripheral Configuration Structure	18
4.1.6 Secure Initialization	18
4.1.7 Device Disconnected	19
4.2 Modules	19
4.2.1 adc.c and adc.h	20

4.2.2	gps.c and gps.h	21
4.2.3	i2c.c and i2c.h	21
4.2.4	accel.c and accel.h	22
4.2.5	color.c and color.h	23
4.2.6	temp_hum.c and temp_hum.h	23
4.2.7	rgb_led.c and rgb_led.h	24
4.2.8	board_led.c and board_led.h	24
4.2.9	user_button.c and user_button.h	25
4.3	Threads	25
4.3.1	Sensors Thread	25
4.3.2	GPS Thread	28
4.4	Main	30
4.5	Zephyr RTOS	30
4.5.1	prj_nucleo_wl55jc.conf	30
4.5.2	nucleo_wl55jc.overlay	31
4.6	Thread Stack and CPU Usage Analysis	31
4.7	Compilation and Flashing Output Analysis	33
4.8	Flashing the Firmware onto the STM32WL55	33
4.9	Code Documentation	34
5	Results	36
5.1	Unitary Tests	36
5.2	System Tests	36
5.2.1	Initialisation	36
5.2.2	Measurements	36
5.2.3	Mode Changes	36
5.2.4	RGB Dominant color (TEST MODE)	36
5.2.5	Limits Check (NORMAL MODE)	36
5.2.6	RGB Alarms (NORMAL MODE)	36
5.2.7	Statistics Comprobatation	36
5.2.8	ADVANCED MODE	36
5.3	External Tests	36
5.3.1	CPP Check	36
5.3.2	Valgrind	36
6	Advanced Specifications Implemented	37
6.1	Problem Statement	37
6.2	Implementation	37
6.2.1	Macro definitions and timing parameters	38
6.2.2	Sensor clear-channel safety check and normalisation	38
6.2.3	Diagnostic logging	38
6.2.4	Duty-cycle computation	39
6.2.5	Main PWM loop and per-step evaluation	39
6.3	Result	40
7	Conclusions and Future Works	41
8	Bibliography	42
Appendix A	- GitHub	43
A.1	GitHub Repository - Source Code	43
A.2	GitHub Pages - Documentation	43
A.3	GitHub Action - Workflow	43

List of Figures

1	Block Diagram of the Hardware System	5
2	RGB LED circuit diagram	8
3	Phototransistor circuit diagram	9
4	Soil moisture sensor circuit diagram	10
5	System behaviour	13
6	Test mode behaviour	14
7	Normal mode behaviour	15
8	Advanced mode behaviour	16
9	Modules overview	19
10	Thread stack and Central Processing Unit (CPU) usage	32
11	Compilation memory usage report	33
12	Flashing process using STM32CubeProgrammer	34

List of Tables

1	Summary of Suggested Hardware for the IoT System	3
2	Software Requirements	3
3	Sensing Requirements	3
4	Operating Modes Requirements	4
5	Additional Specifications Implemented	4
6	System Connections	6

List of Listings

1	Sensor data structure with atomic fields	17
2	Peripheral configuration and synchronization structure	18
3	Secure initialization code	18
4	CMakeLists.txt module integration	20
5	Sensors thread initialization	25
6	Sensors thread configuration	26
7	ADC percentage acquisition helper function	26
8	Accelerometer data acquisition	26
9	Temperature and humidity acquisition	27
10	Color sensor acquisition	27
11	Sensors thread main loop	27
12	Sensors thread public header	28
13	Initialization of the GPS measurement thread	28
14	GPS thread configuration in Zephyr	29
15	GPS data acquisition helper function	29
16	GPS thread entry routine	29
17	GPS thread public interface	30
18	prj_nucleo_wl55jc.conf	30
19	nucleo_wl55jc.overlay	31
20	Thread stack and CPU usage report - prj_nucleo_wl55jc.conf	33
21	Software-based PWM implementation for ADVANCED mode	37
22	Macros for <i>ADVANCED</i> mode	38
23	Color normalisation based on clear channel	38
24	Diagnostic logging of normalised color values	39
25	Computation of duty counts from normalised percentages	39
26	Main software PWM loop	39
27	GitHub Action workflow	43

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
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	Interrupt Service Routine
LED	Light Emitting Diode
LoRa	Long Range
LPWAN	Low-Power Wide-Area Network
MCU	Microcontroller Unit
NMEA	National Marine Electronics Association
OS	Operating System
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
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

Todo list

check resolutions	3
acronimos, vcc, vin?	5
comparar con las diapos	5
comprobar formulas	9
comprobar formulas	10
revisar rangos del acelerometro	11
Añadir captura	19
Añadir captura	19
Añadir captura	19
chequear gs del acelerometro	22
chequear en todo el docuemnto si zephyr es os o rtos	30
capturas	40

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 Operating System (OS).

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 OS.

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 OS, 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 OS.
- 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	Real-Time Operating System (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.

check resolutions

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
GR1	Robustness	System must be stable and robust
GR2	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

acronimos,
vcc, vin?

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.

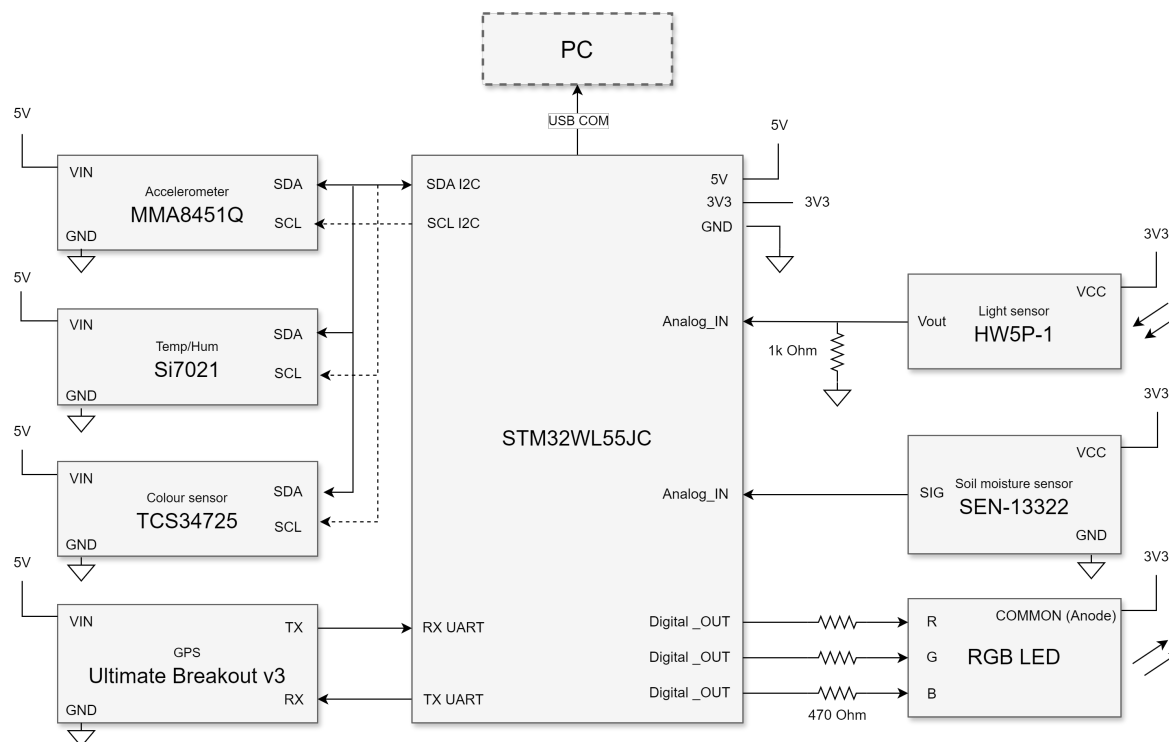


Figure 1: Block Diagram of the Hardware System

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.

comparar
con las
diapos

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 Ground	Vout Ground (GND)	CN8 PB_1 CN6 GND	ADC 1/5 GND	Analog input Ground
Soil Moisture	SEN-13322	VCC 3.3V	VCC	CN6 3V3	3V3	3.3V
		Vout Ground	SIG GND	CN8 PB_13 CN6 GND	ADC 1/0 GND	Analog input 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	Interrupt out	INT	-	-	-
		LED on/off	LED	-	-	-
		VIN 5V	VIN	CN6 5V	5V	5V
		Ground	GND	CN6 GND	GND	Ground
		I2C SCL	SCL	CN5 PA_12	I2C2_SCL	I2C SCL
Accelerometer	MMA8451Q	I2C SDA	SDA	CN5 PA_11	I2C2_SDA	I2C SDA
		Output 3.3V reg	3Vo	-	-	-
		Inertial Interrupt 1,	I1	-	-	-
		Output pin	Output pin	-	-	-
		Inertial Interrupt 2,	I2	-	-	-
		Output pin	Output pin	-	-	-
		I2C least significant bit of the device I2C address	A	-	-	-
		VIN 5V	VIN	CN6 5V	5V	5V
		Ground	GND	CN6 GND	GND	Ground
		I2C SCL	SCL	CN5 PA_12	I2C2_SCL	I2C SCL
GPS	Adafruit Ultimate GPS Breakout v3	I2C SDA	SDA	CN5 PA_11	I2C2_SDA	I2C SDA
		Output 3.3V reg	3Vo	-	-	-
		Inertial Interrupt 1,	I1	-	-	-
		Output pin	Output pin	-	-	-
		Inertial Interrupt 2,	I2	-	-	-
		Output pin	Output pin	-	-	-
		I2C least significant bit of the device I2C address	A	-	-	-
		VIN 5V	VIN	CN6 5V	5V	5V
		Ground	GND	CN6 GND	GND	Ground
		Serial TX	TX	CN9 PB_7	UART1_Reception (RX)	UART RX
GPS	Adafruit Ultimate GPS Breakout v3	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	-	-	-
		Serial 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 3.3V 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.

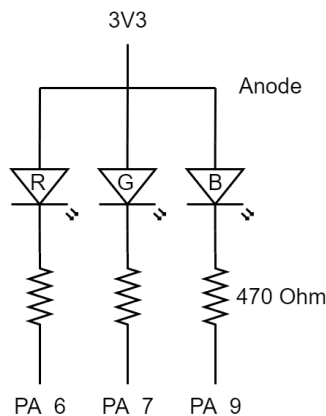


Figure 2: RGB LED circuit diagram

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.

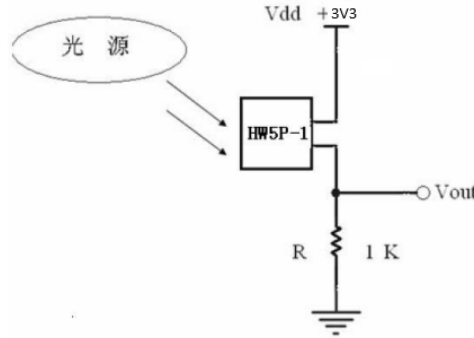


Figure 3: Phototransistor circuit diagram

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:

$$V_{phototransistor} = \frac{ADC_value}{ADC_max} \cdot V_{ref}$$

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

$$Light\% = \frac{V_{phototransistor}}{V_{ref}} \cdot 100 = \frac{ADC_value}{ADC_max} \cdot 100$$

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.

comprobar
formulas

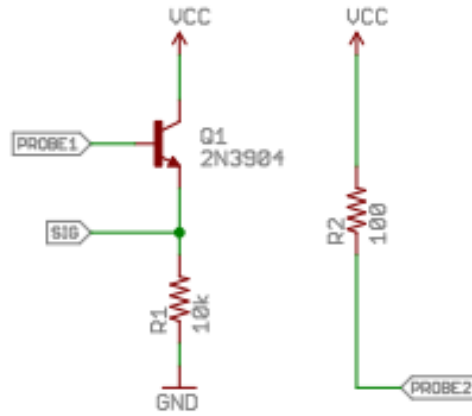


Figure 4: Soil moisture sensor circuit diagram

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

$$\text{Moisture}\% = \frac{\text{ADC_value}}{\text{ADC_max}} \times 100$$

 comprobar
formulas

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., $\pm 2g$, $\pm 4g$, or $\pm 8g$):

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

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.

revisar
rangos del
acelerometro

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.

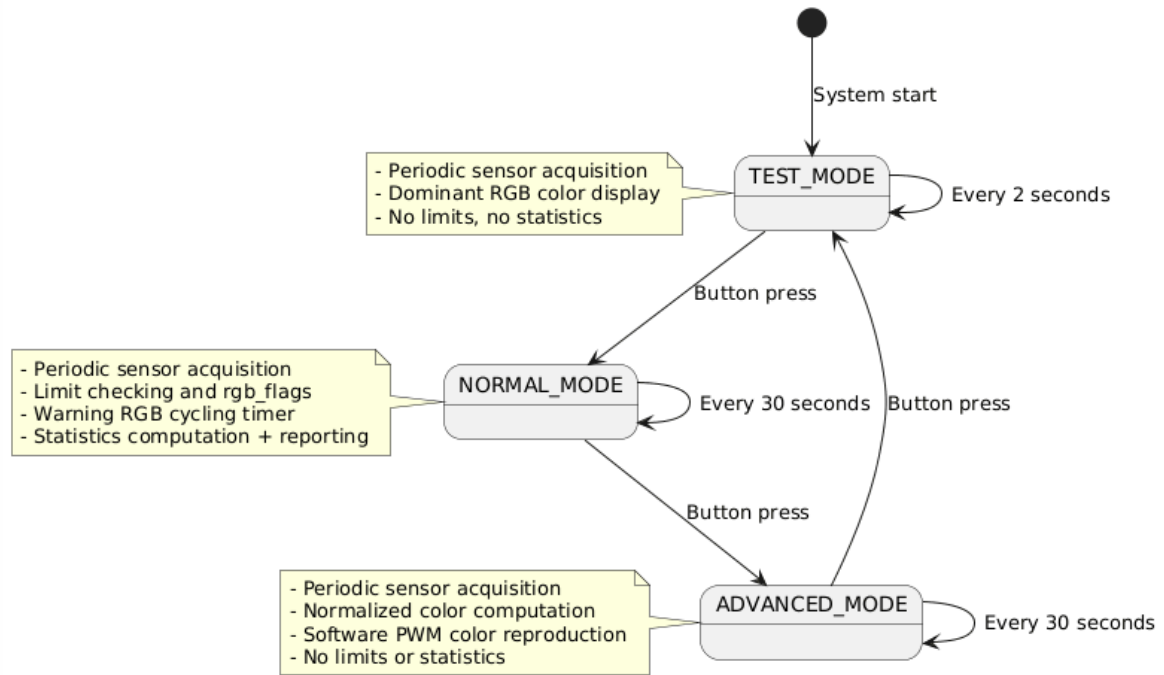


Figure 5: System behaviour

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.

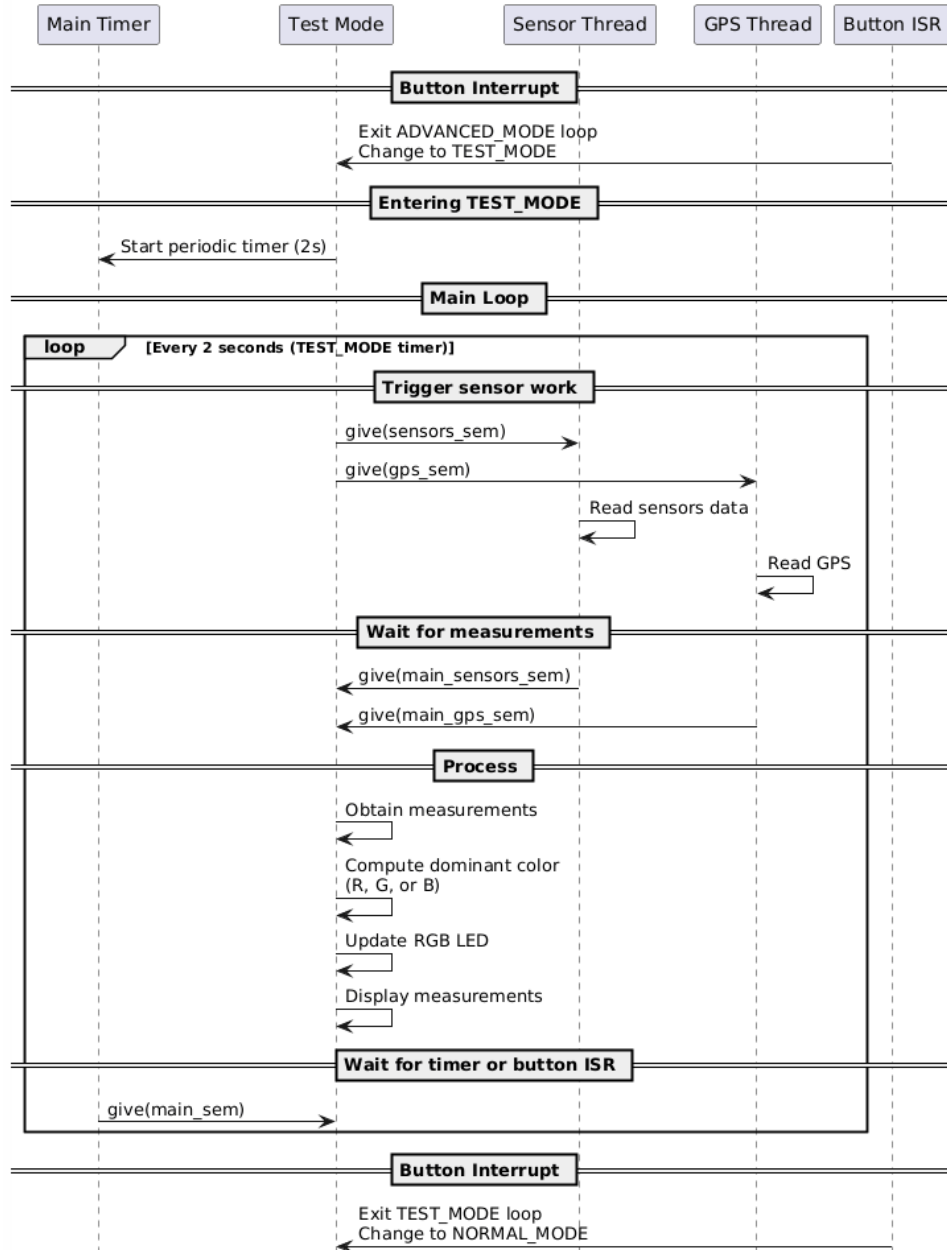


Figure 6: Test mode behaviour

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.

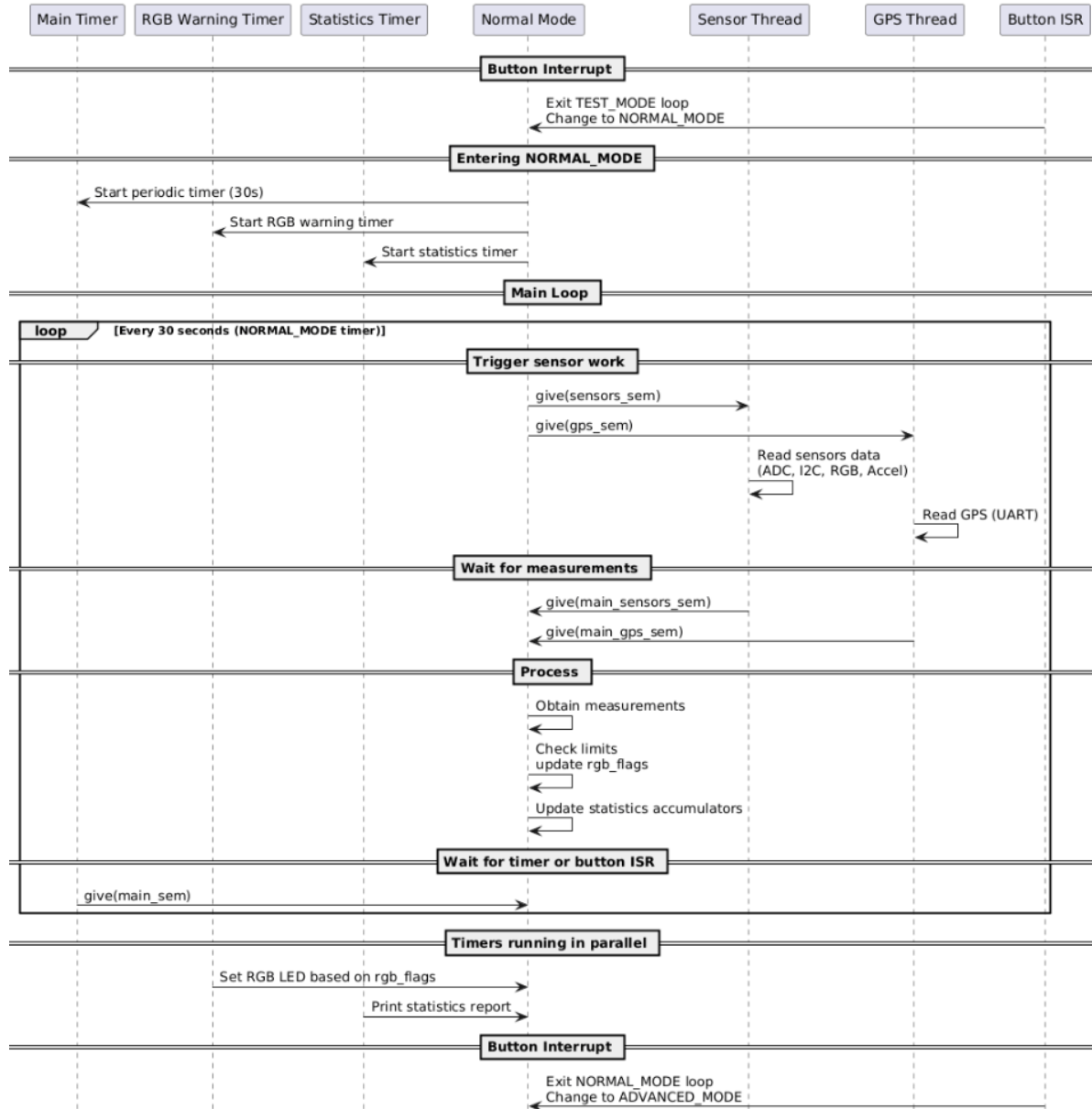


Figure 7: Normal mode behaviour

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.

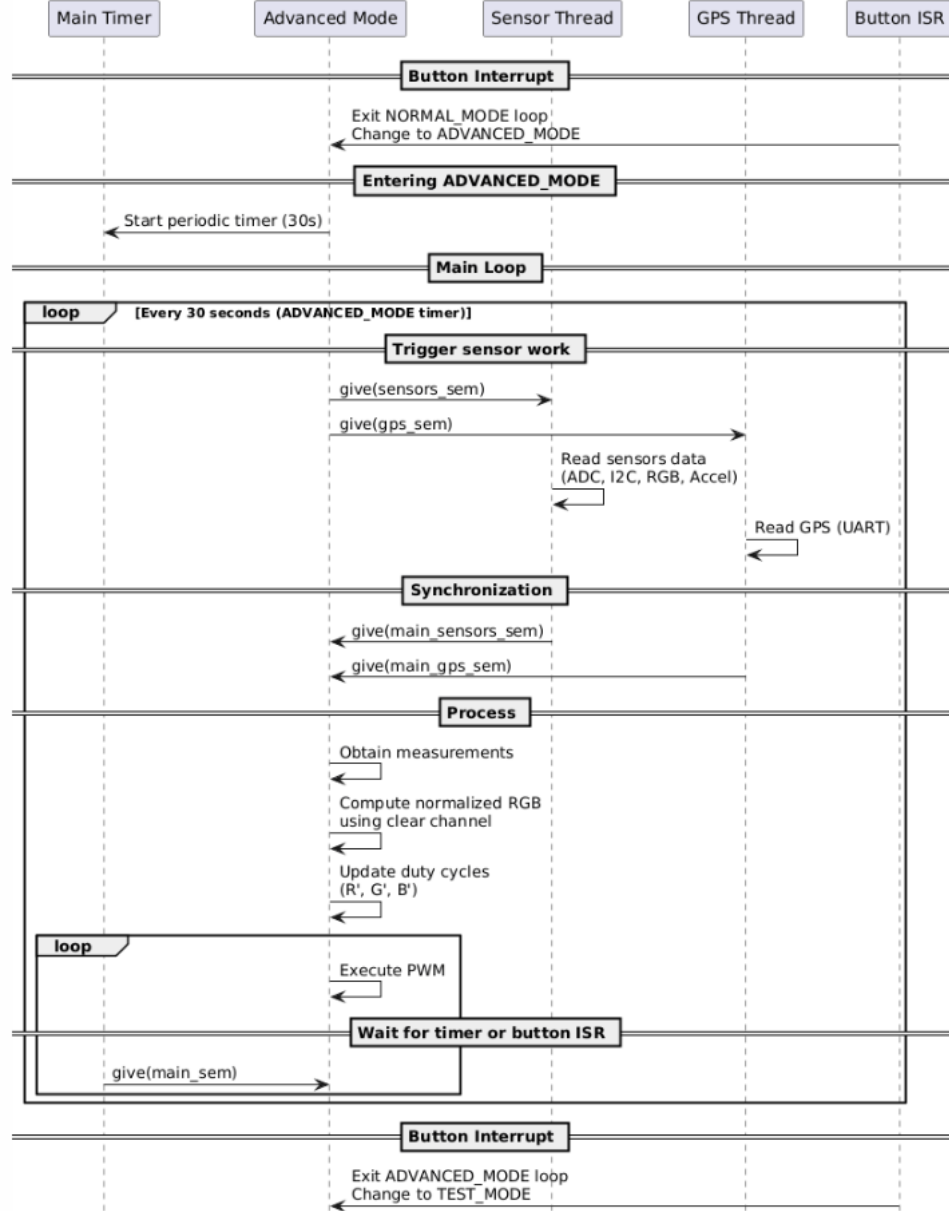


Figure 8: Advanced mode behaviour

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:

TEST_MODE → NORMAL_MODE → ADVANCED_MODE → 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 Synchronisation and Thread Interactions

Inter-thread coordination relies on multiple named semaphores.

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

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 serialise 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 synchronisation 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;
}
}

```

Añadir
captura

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.

Añadir
captura

If the device is reconnected, normal operation resumes.

Añadir
captura

4.2 Modules

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

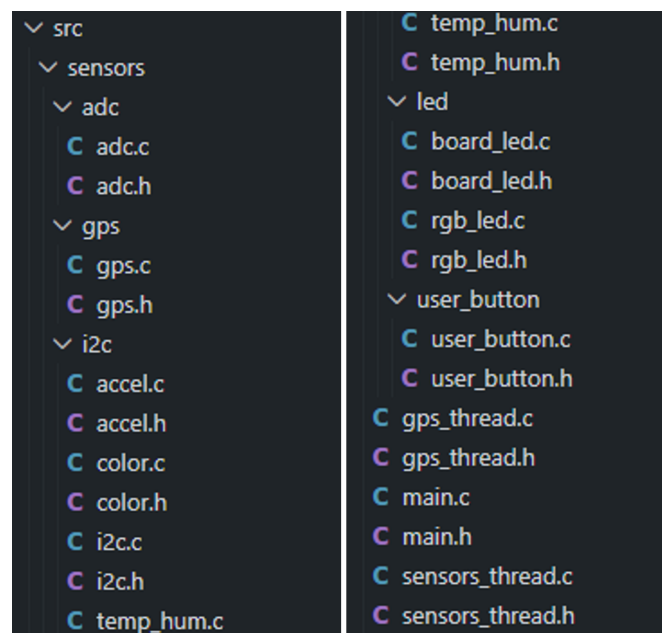


Figure 9: Modules overview

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 (GNSSGA)) 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 $\pm 2\text{g}$ 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 $\pm 2\text{g}$, $\pm 4\text{g}$, and $\pm 8\text{g}$ ranges via the `XYZ_DATA_CFG` register. The system uses $\pm 2\text{g}$ 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.

chequear
gs del
acelerometro

- **Conversion to m/s^2 :** 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 ($1\times$, $4\times$, $16\times$, $60\times$). 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

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 18: `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

```

chequear
en todo el
documento
si zephyr
es os o rtos

```
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 an 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 19: `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 10: 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 20: 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:

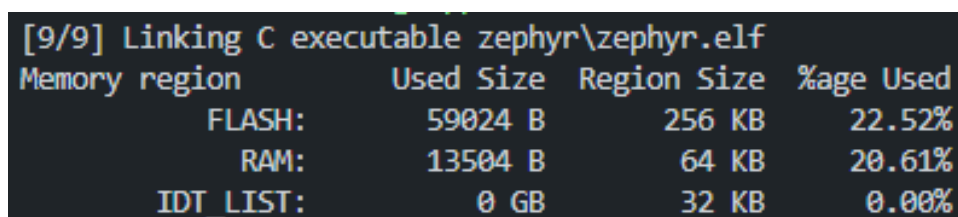
- All thread stacks remain within safe usage ranges, with most below the recommended 60% threshold.
- CPU usage distribution behaves as expected for a sensor-driven, event-based embedded application.
- The idle thread dominates CPU cycles, indicating efficient low-power execution and minimal background 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 11, 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.



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 11: Compilation memory usage report

4.8 Flashing the Firmware onto the STM32WL55

The Figure 12 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.

```
-----
                        STM32CubeProgrammer v2.20.0
-----

ST-LINK SN   : 003E00124741500120383733
ST-LINK FW   : V3J7M3
Board        : NUCLEO-WL55JC
Voltage      : 3.28V
SWD freq     : 8000 KHz
Connect mode : Under Reset
Reset mode   : Hardware reset
Device ID    : 0x497
Revision ID   : Rev Y
Device name   : STM32WLxx
Flash size   : 256 KBytes
Device type   : MCU
Device CPU    : Cortex-M4
BL Version    : 0xC4

Opening and parsing file: zephyr.hex

Memory Programming ...
  File       : zephyr.hex
  Size       : 57.64 KB
  Address    : 0x08000000

Erasing memory corresponding to segment 0:
Erasing internal memory sectors [0 28]
Download in Progress:
 100%

File download complete
Time elapsed during download operation: 00:00:01.654

RUNNING Program ...
  Address    : 0x08000000
Application is running, Please Hold on...
Start operation achieved successfully
```

Figure 12: 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

5.1 Unitary Tests

5.2 System Tests

5.2.1 Initialisation

5.2.2 Measurements

5.2.3 Mode Changes

5.2.4 RGB Dominant color (TEST MODE)

5.2.5 Limits Check (NORMAL MODE)

5.2.6 RGB Alarms (NORMAL MODE)

5.2.7 Statistics Comprobatation

5.2.8 ADVANCED MODE

The results of the ADVANCED MODE are presented in subsection 6.3

5.3 External Tests

5.3.1 CPP Check

5.3.2 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 21: 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. */

/* In ADVANCED mode */
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 22: 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 23: 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 24: 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 `printf` format specifier and to ensure consistent formatting across platforms.

6.2.4 Duty-cycle computation

Listing 25: 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 26: 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

captures

The software-based PWM strategy successfully reproduces the colour detected by the RGB sensor. The LED output maintains the same chromatic proportions observed in the ambient light, while ensuring smooth transitions and stable operation. 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

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 - $\pm 2/4/8g$ @ 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/doxxygen/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 27: 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
```
