Design and Implementation of a Greenhouse Monitoring System Using STM32F303

Sonakshi Agarwal
2410110333
P3

Karanam Avijith
2410110163
P3

Ishita Bhatt
2410110151
P3

Arjun Bose
2410110067
P3

Sarah Chhabra
2410110303
P3

D.J. Ciranthan
2410110112
P3

Laasya Madhuri Dasari
2410110185
P3

***Abstract*—This paper presents the design and development of a greenhouse monitoring system built around the STM32F303 microcontroller. The system integrates multiple environmental sensors, including an LDR for light intensity measurement, a DHT22 for temperature and humidity monitoring, and an MQ-135 for air quality assessment. Sensor data is displayed in real time on an LCD module. This work emphasizes embedded systems design, sensor interfacing, hardware–software co-design, and reliability considerations for greenhouse environments.**

***Index Terms*—Greenhouse Monitoring, STM32F303, LDR, DHT22, MQ135, Embedded Systems, Environmental Sensors.**

I. Introduction

Environmental monitoring systems play a crucial role in modern agriculture, especially inside greenhouses where environmental parameters must be maintained within narrow ranges to ensure optimal plant growth. Factors such as temperature, humidity, light intensity, and air quality directly influence photosynthesis, nutrient uptake, and overall plant health.

The objective of this project is to design and implement a compact, reliable, and low-cost greenhouse monitoring system using the STM32F303 microcontroller. By integrating multiple sensors and displaying real-time data on an LCD module, the system enables growers to closely track key environmental variables without relying on manual measurements. This reduces the likelihood of human error and ensures consistent monitoring throughout the day.

Unlike large-scale automated greenhouse platforms, this project focuses specifically on measuring and reporting environmental conditions rather than controlling them. The scope includes sensor integration, data acquisition, microcontroller-based processing, and data display, making it suitable for research environments, small-scale greenhouse operations, and academic demonstration purposes.

II. Requirements

The greenhouse monitoring system must accurately observe the environmental state of the greenhouse in real time, focusing on parameters critical to plant health such as temperature, humidity, light intensity, and air quality. The system should continuously acquire sensor data, process it reliably, and present it in an accessible format to the user.

To meet these goals, the system must satisfy the following requirements:

- **Accurate sensing:** The system must read and interpret sensor data with sufficient precision to reflect meaningful changes in environmental conditions.
- **Real-time performance:** Data acquisition and display should occur at a rate that allows timely detection of deviations from optimal growing conditions.
- **Reliability:** The system should operate continuously over long periods without frequent resets or failures, even under fluctuating environmental conditions.
- **Scalability:** Additional sensors or expansion modules should be integrable without major redesign.
- **Ease of use:** Installation, configuration, and daily operation must require minimal technical expertise.
- **Power efficiency:** The system should consume minimal power to support extended deployment in greenhouse settings.
- **Clear output display:** All monitored parameters must be presented in a user-friendly format on the LCD module.

Input data for this system includes readings from the DHT22 (temperature and humidity), LDR (light intensity), and MQ-135

(air quality). The output consists of processed, human-readable data displayed on a 16×2 LCD or equivalent interface.

III. STM32F303 in Greenhouse Monitoring

The STM32F303 microcontroller plays a central role in this project due to its mixed-signal processing capabilities, precise timing peripherals, and low-power operation. Key advantages include:

- **High-resolution ADCs:** Essential for stable LDR and MQ-135 readings.
- **Hardware FPU:** Accelerates logarithmic and compensation calculations used in gas sensing.
- **Advanced timers:** Provide microsecond timing accuracy required for the DHT22 protocol.
- **Multiple communication buses:** I2C, SPI, and USART enable easy expansion of sensors and displays.
- **Low-power modes:** Allow continuous operation in greenhouse environments with minimal energy usage.
- **Scalability:** Easily extended to support actuators (fans, pumps), wireless communication, and SD card logging.

The STM32CubeIDE used for development provides integrated debugging, peripheral configuration through STM32CubeMX, and real-time code analysis tools, enabling rapid prototyping and reliable firmware deployment.

IV. Sensor Interfaces and Communication

A. Sensor Technical Specifications

B. Measurement Format

This system uses three primary sensors—DHT22, LDR, and MQ-135—each providing measurements in a different electrical format. Understanding how each sensor encodes its data is essential for reliable acquisition and processing on the STM32F303.

1) DHT22 Temperature and Humidity Format: The DHT22 (AM2302) transmits temperature and humidity using a single-wire digital protocol similar to the DHT11 but with higher resolution and accuracy. After the microcontroller sends the start signal, the DHT22 responds with a fixed 40-bit data frame containing:

- 8 bits: integral humidity value
- 8 bits: decimal humidity value (provides 0.1% resolution)
- 8 bits: integral temperature value
- 8 bits: decimal temperature value (provides 0.1 C resolution)
- 8 bits: checksum

Key specifications that motivated the switch include a typical humidity accuracy of about ±2% RH and temperature accuracy of about ±0.5 °C, with measurement ranges of 0–100% RH and -40–80 °C. The DHT22 requires a minimum sampling interval of approximately 2 seconds (0.5 Hz). Accurate timing using STM32 timers ensures proper decoding of the bitstream.

TABLE II: MQ-135 Gas Sensor — Standard Specifications

Parameter	Specification
Model	MQ-135
Sensor Type	Semiconductor
Encapsulation	Bakelite, metal cap
Target Gases	Ammonia, sulfide, benzene vapors, smoke, VOCs
Detection Range	10–1000 ppm
Circuit Conditions	$V_c \leq 24$ V DC, $V_H = 5.0 \pm 0.1$ V, load R_L adjustable
Characteristics	$R_H = 29\,\Omega \pm 3\,\Omega$, $P_H \leq 950$ mW
Sensitivity	$R_s(\text{air})/R_s(400\,\text{ppm}\,H_2) \geq 5$
Output Voltage	2.0–4.0 V at 400 ppm H_2
Test Conditions	20 °C±2 °C, 55%±5% RH, preheat ≥ 48 h

TABLE I: DHT22 (AM2302) — Technical Specifications

Parameter	Specification
Model	DHT22 (AM2302)
Power supply	3.3–6 V DC
Output signal	Digital single–wire
Sensing element	Polymer capacitive
Operating range	Humidity: 0–100% RH; Temperature: -40 to 80 °C
Accuracy	±2% RH; ±0.5 °C
Resolution	0.1% RH; 0.1 °C
Repeatability	±1% RH; ±0.2 °C
Hysteresis	±0.3% RH
Long-term stability	±0.5% / year
Sensing period	2 seconds typical
Dimensions	14×18×5.5 mm (small), 22×28×5 mm (large)

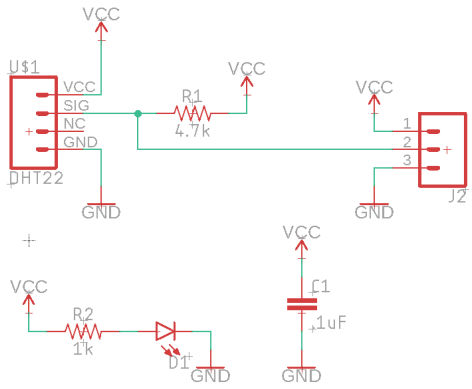

Fig. 1: The circuit of DHT22 sensor.

2) MQ-135 Gas Sensor Format: The MQ-135 outputs an **analog voltage** proportional to changes in air quality. Instead of providing direct ppm values, the sensor's resistance varies with gas concentration. The STM32F303's 12-bit ADC samples this voltage and converts it into a digital value between 0 and 4095. The measurement process involves:

- Sampling the analog output through the ADC
- Converting ADC readings to sensor resistance R_s
- Comparing R_s to a calibrated baseline R_0
- Estimating air quality (AQI trend) through logarithmic models

This measurement format allows continuous tracking of air quality trends within the greenhouse.

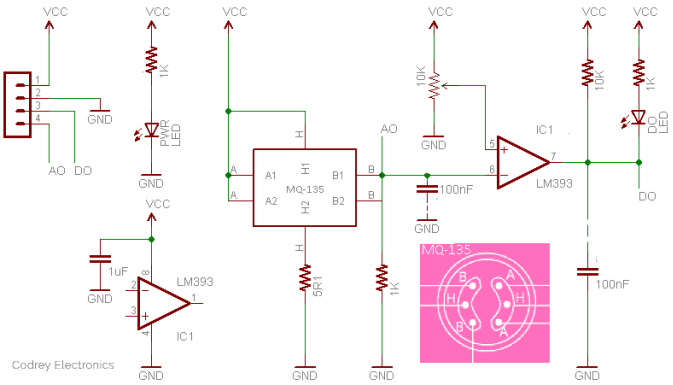

Fig. 2: THe circuit of MQ-135 sensor.

3) LDR Light Sensor Format: The LDR functions as part of a **voltage divider**. Its resistance decreases as light intensity increases, producing a higher ADC voltage under bright conditions. The measurement consists of:

- Reading the ADC voltage from the divider
- Mapping voltage to relative brightness levels
- Optionally converting to lux using empirical calibration curves

Since LDRs are analog and nonlinear, averaging and oversampling techniques are used to stabilize readings.

TABLE III: LDR — Typical Characteristics

Parameter	Value / Notes
Sensor Type	CdS photoresistor (LDR)
Output Type	Analog resistance (voltage divider to MCU ADC)
Dark Resistance	Typically MΩ range
Bright Resistance	Few hundred ohms
Response Time	Slow (10 ms – 200 ms typical)
Linearity	Nonlinear; needs calibration
Illuminance Range	1–10,000 lux (model dependent)
Temperature Effect	Resistance varies with temperature
Usage Notes	Suitable for relative brightness, not accurate lux

4) LCD Display Output Format: The 16×2 LCD (parallel or I2C) receives processed sensor values in ASCII character format. The STM32 updates the display by:

- Converting numerical sensor values to strings
- Sending characters sequentially via 4-bit parallel mode or over I2C
- Refreshing the display at regular intervals (typically 1 Hz)

This ensures real-time visibility of environmental parameters without the need for external devices.

TABLE IV: 16×2 LCD (HD44780-Compatible) — Specifications

Parameter	Specification / Notes
Controller	HD44780-compatible
Display Size	16×2 characters
Supply Voltage	5 V typical (3.3 V with level shifting)
Interface Options	4-bit/8-bit parallel or I^2C (PCF8574 backpack)
Contrast Control	Via VO pin + potentiometer
Backlight	LED-based, requires limiting resistor
Power Consumption	Few mA without backlight
Command Set	Standard LCD commands (clear, home, shift, etc.)
Integration Note	Prefer I^2C to save STM32 GPIO pins

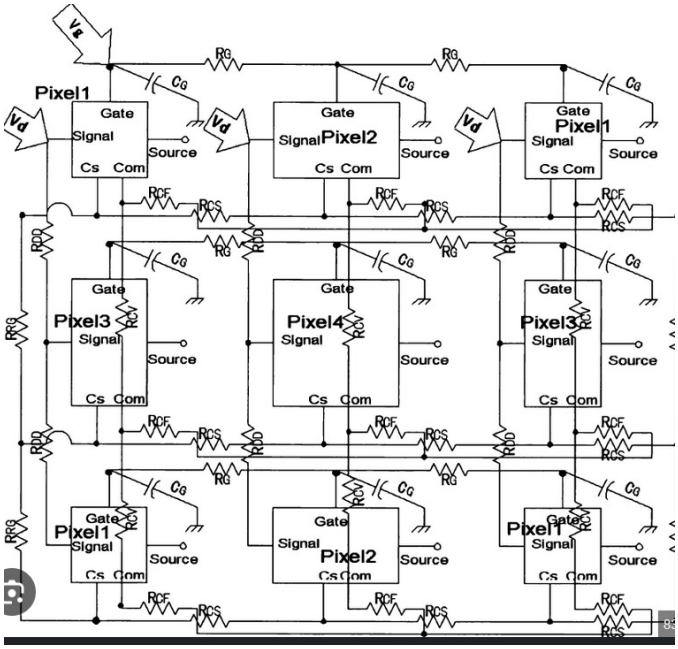

Fig. 3: Internal circuit of LCD Display.

V. System Overview

The system architecture consists of three major components: sensors, microcontroller, and display. The below schematic shows the complete circuit connections along with the sensors that we have utilized in our project. The design integrates three environmental sensors—DHT22, MQ-135, and an LDR module—along with a 16×2 character LCD and discrete status LEDs. Each subsystem connects to the microcontroller through dedicated GPIO, ADC, or timing peripherals, allowing reliable mixed-signal acquisition.

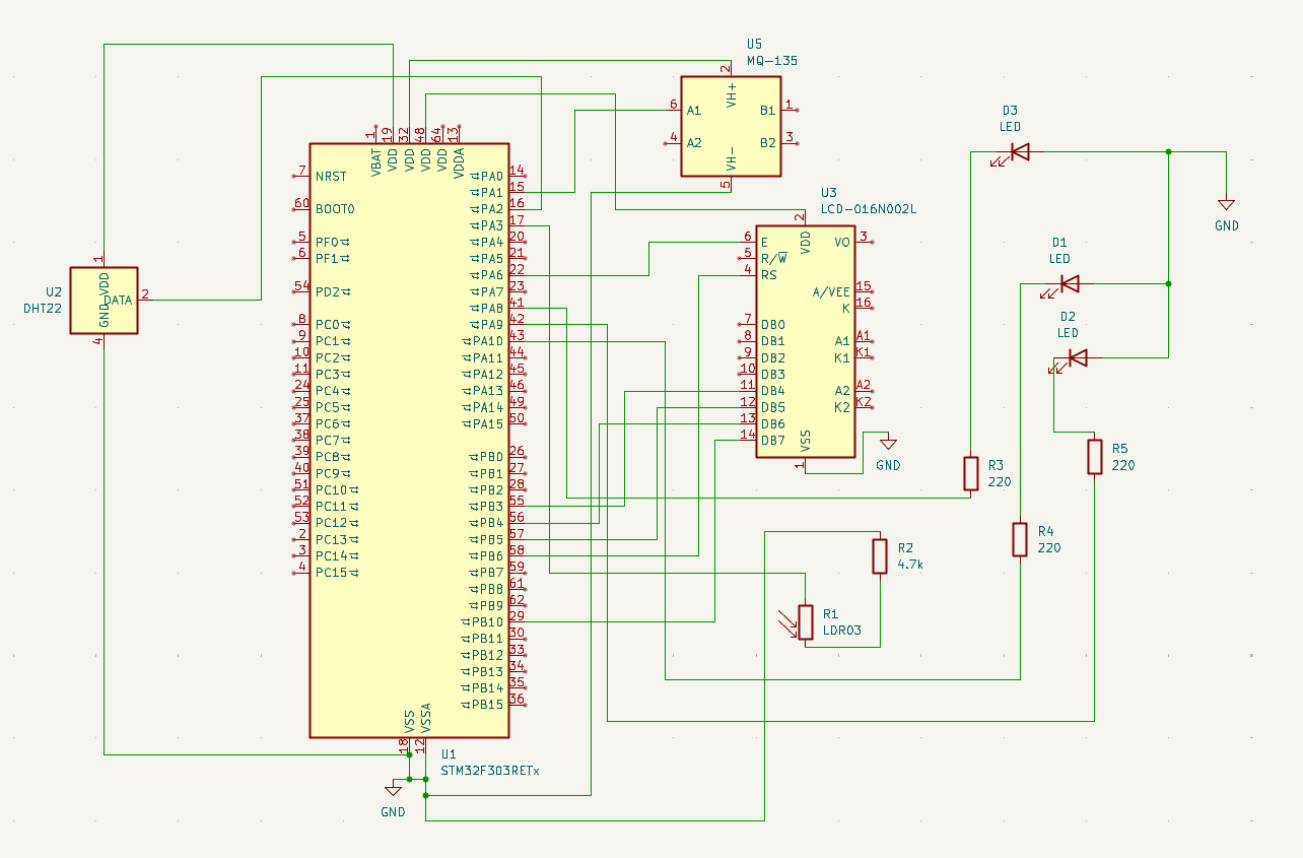

Fig. 4: Complete schematic of the greenhouse monitoring system.

VI. Environmental Sensing Theory

Understanding plant–environment interactions is crucial when designing a monitoring system. Four key parameters influence greenhouse crop productivity: temperature, humidity, light intensity, and air quality. Each parameter affects biological processes such as transpiration, photosynthesis, and nutrient uptake.

A. Temperature

Plant metabolic reactions are temperature-dependent. Most greenhouse crops thrive in the range of 20–30 °C. Deviations trigger:

- Reduced enzyme activity at low temperatures.
- Increased respiration losses at high temperatures.
- Heat stress leading to stomatal closure and reduced CO_2 uptake.

Continuous temperature sensing allows early detection of thermal stress, enabling corrective actions such as activating cooling vents or shading.

B. Humidity

Humidity influences transpiration, nutrient uptake, and stomatal behavior. Optimal levels typically lie between 50–70%. Extremely high humidity promotes fungal growth, whereas low humidity accelerates plant water loss. The DHT22 sensor provides higher-resolution humidity measurements to better track these dynamics.

C. Light Intensity

Photosynthetically Active Radiation (PAR) drives photosynthesis. LDRs offer a cost-effective approximation of ambient light levels. Although LDRs do not measure absolute lux accurately, they capture relative brightness trends, enabling automation for shading or grow-light activation.

D. Air Quality

Air quality inside a greenhouse is heavily influenced by CO_2 levels, volatile organic compounds (VOCs), and ammonia from fertilizers. MQ-135 provides a low-cost proxy measurement. Even though it cannot isolate specific gases, changes in Air Quality Index (AQI) correlate with ventilation needs.

TABLE V: Ideal Environmental Conditions for Greenhouse Growth

Environmental Target Values
Temperature: 22–28°C (day), 15–20°C (night)
Humidity: 50–70% RH
Air Quality Index (AQI): 0–50 optimal, ¡100 acceptable
Light Intensity: 20,000–60,000 lux

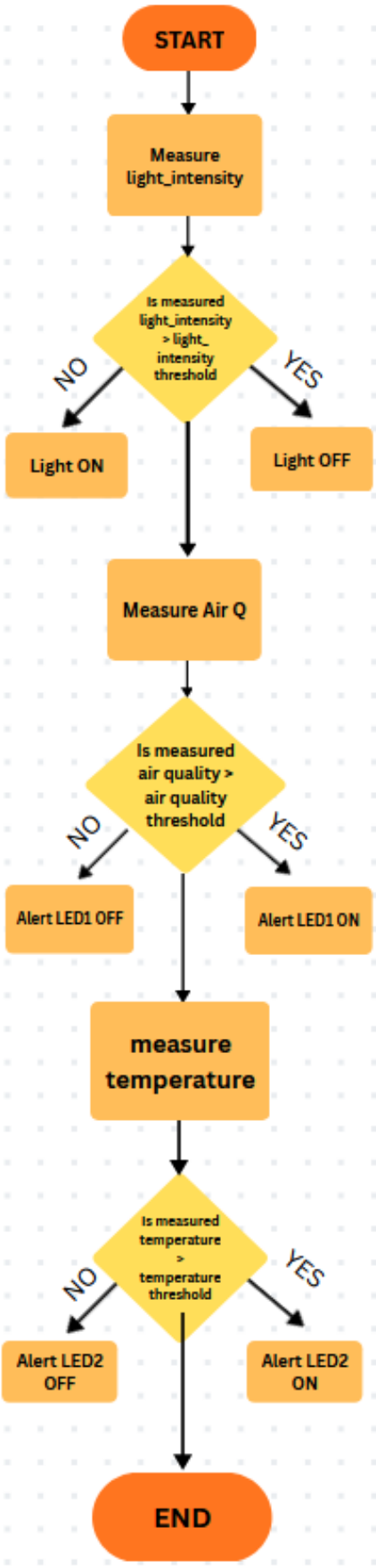

Fig. 5: Flowchart of the working of the greenhouse monitoring system.

VII. Design Challenges

A. Sensor Noise and Drift

Analog sensors (LDR, MQ-135) suffer from noise, drift, and nonlinearity. DHT22 needs precise timing and a 2 s sampling interval. Oversampling and recalibration help improve stability.

B. Environmental Stressors

Humidity and temperature swings can corrode contacts and distort readings. Protective coatings and sealed enclosures improve durability.

C. Timing Constraints

DHT22 requires microsecond timing, making STM32 timers essential for reliable communication.

D. Power Stability

MQ-135's heater draws high current; poor decoupling causes false readings. Proper grounding and filtering are required.

E. Data Interpretation Challenges

Nonlinear sensors and changing conditions require filtering and adaptive thresholds to extract meaningful trends.

VIII. Trade-Off Analysis

A. LDR vs Digital Lux Sensor

LDRs are cheap but nonlinear and slow; digital lux sensors are accurate but costly. LDR was chosen for low cost and relative brightness measurement.

B. MQ-135 vs NDIR CO_2 Sensor

MQ-135 is low-cost but drifts; NDIR is accurate but expensive. MQ-135 was selected to minimize system cost.

C. STM32 vs Arduino vs ESP32

Arduino lacks precision, ESP32 consumes more power, while STM32 offers better ADC quality and timing—making it the preferred choice.

VII. System Integration

Integrating sensors, power systems, timing modules, and user interfaces requires careful hardware–software coordination:

A. Interrupt Management

Timer interrupts handle DHT22 pulse capture, while DMA interrupts manage ADC buffer updates. Prioritizing these interrupts ensures deterministic performance.

B. Clock Configuration

The STM32F303 runs on an HSE. A stable clock is essential for accurate timing in DHT22 and consistent ADC sampling rates.

C. Power Budgeting

The MQ-135 heater draws significant current. Power domains must be isolated to prevent brown-outs. Low-power modes can be used when sensors do not require constant updates.

VIII. Hardware Design

A. Microcontroller: STM32F303

The STM32F303 series features ARM Cortex-M4 cores with floating-point support and rich peripheral sets, making it ideal for sensor-heavy applications.

In the context of a greenhouse monitoring system, the STM32F303 provides several critical advantages:

- **High-speed 12-bit ADC:** The MCU includes multiple 12-bit ADCs with sampling rates high enough to continuously monitor analog sensors such as the LDR and MQ-135. This allows oversampling, averaging, and noise reduction techniques that significantly improve stability in harsh greenhouse environments where lighting conditions fluctuate.
- **Multiple ADC channels:** Since the project includes more than one analog sensor, the STM32F303's multi-channel ADC architecture simplifies simultaneous acquisition without the need for external multiplexers.
- **Floating Point Unit (FPU):** Real-time computation of sensor calibration curves—especially logarithmic functions for MQ-135 gas sensing—benefits greatly from hardware-accelerated floating-point arithmetic.
- **Advanced Timers:** Sensors like the DHT22 require precise microsecond-level timing to decode data frames. The STM32F303's advanced timers enable accurate pulse-width measurements with minimal CPU overhead.
- **I2C Support:** The microcontroller includes robust I2C peripherals that allow reliable communication with the LCD module (through PCF8574) while maintaining low power consumption.
- **Low-power modes:** In greenhouse applications where systems must run continuously, the STM32F303 provides several low-power sleep modes that drastically reduce energy consumption when sensors do not need rapid polling.
- **Scalability and Expansion:** The ecosystem around STM32 allows future integration of wireless modules (LoRa, Wi-Fi), SD card storage, and closed-loop actuation without redesigning the core system.

Features used include:

- 12-bit ADC channels for LDR and MQ-135
- GPIO pins for DHT22
- Timer modules for protocol timing
- I2C/SPI (if using LCD backpack)

IX. Software Tools

This project was developed using **STM32CubeIDE**, an all-in-one development environment provided by STMicroelectronics. STM32CubeIDE integrates project management, code editing, peripheral configuration, compilation, and debugging into a single platform, making it well-suited for embedded applications such as this greenhouse monitoring system.

STM32CubeIDE includes the STM32CubeMX configuration tool, which simplifies peripheral initialization by letting developers visually configure ADC channels, timers, GPIOs, and communication interfaces. The generated initialization code ensures that firmware development is consistent, reliable, and less prone to configuration errors.

Built on the Eclipse®/ CDTTM framework with a GCC-based toolchain, STM32CubeIDE supports:

- Code auto-generation for peripherals
- Real-time debugging via ST-Link
- Register-level and memory inspection
- Integrated build and project management
- Cross-platform support (Windows, macOS, Linux)

These features significantly streamline the development workflow, allowing quick iteration and reliable firmware deployment.

A. Libraries and Inbuilt Functions Used

The firmware of the greenhouse monitoring system makes use of standard C libraries along with a custom LCD driver library. These libraries provide core functionality for data handling, hardware configuration, and display control.

1) Included Libraries

- **stdio.h**
  Provides standard input/output utilities such as `printf()` and `sprintf()` for formatting sensor data into displayable strings.
- **main.h**
  Contains project-specific definitions, pin mappings, peripheral handles, and global configurations generated by STM32CubeIDE. It acts as the central reference for hardware initialization.
- **LCD16x2**
  - **LCD16x2.h**
    A dedicated LCD driver header file that provides APIs to control the 16x2 LCD, including sending commands, printing characters, clearing the display, and cursor positioning.
  - **LCD16x2SR(uint8_t steps)**
    Scrolls the displayed text to the **right** by a specified number of steps. This is useful for displaying messages longer than the LCD's 16-character width.
  - **LCD16x2SL(uint8_t steps)**
    Scrolls the displayed text to the **left** by a given number of steps, enabling smooth marquee-style movement on the LCD.

X. Detailed Design

The detailed design transforms the system requirements into a concrete implementation, describing how the hardware and software components interact within the STM32-based greenhouse monitor.

At the core of the system is the STM32F303 microcontroller, which performs data acquisition, preprocessing, and communication. The system integrates three primary sensors: the DHT22 for temperature and humidity, the MQ-135 for air quality, and the LDR for ambient light measurement. Each sensor interfaces with the STM32 through an appropriate channel—digital GPIO

for the DHT22, ADC inputs for the MQ-135 and LDR, and I2C/parallel communication for the LCD output module.

The firmware includes modular sensor drivers responsible for managing the timing-critical digital protocol of the DHT22, sampling analog signals from the MQ-135 and LDR via the ADC, and formatting output data for the LCD. Data processing algorithms filter noise from the ADC readings, convert raw codes into interpretable environmental values, and verify checksum integrity for the DHT22. These conversions ensure that the displayed values accurately reflect real-time conditions.

System operation begins with an initialization phase where all STM32 peripherals—GPIO, ADC, timers, and communication buses—are configured. Once initialized, the system repeatedly acquires sensor data, applies filtering and conversion routines, and updates the LCD at periodic intervals. By continuously cycling through acquisition, processing, and display, the system maintains an up-to-date representation of the greenhouse environment.

XIII. Future Improvements and Other Possible Features

A. Temperature Control Using PID

A useful future upgrade for the greenhouse system is to move from simple monitoring to **automatic temperature control** using a **PID (Proportional–Integral–Derivative)** controller. Instead of only displaying temperature, the system would actively adjust a heater or fan to keep the temperature close to a desired setpoint.

The PID controller uses the error between the setpoint T_{sp} and the measured temperature $T(t)$:

$$e(t) = T_{sp} - T(t)$$

The control output is:

$$u(t) = K_p e(t) + K_i \int e(t)\,dt + K_d \frac{de(t)}{dt}$$

Here, K_p, K_i, and K_d determine how fast, how accurately, and how smoothly the temperature is corrected.

1) Digital Implementation: On the STM32F303, the PID can run in discrete form:

$$e[k] = T_{sp} - T[k]$$

$$I[k] = I[k-1] + e[k] \cdot T_s \qquad D[k] = \frac{e[k] - e[k-1]}{T_s}$$

$$u[k] = K_p e[k] + K_i I[k] + K_d D[k]$$

The output $u[k]$ can control heater or fan power through a PWM signal (0–100%).

2) Required Hardware:

- Heater or ventilation fan
- MOSFET/relay driver
- Safety temperature cutoff

3) Basic Tuning:

- Increase K_p until response is noticeable
- Add K_i to remove steady-state error
- Use a small K_d to reduce overshoot

4) Expected Benefit: With PID control, the system can:

- Maintain temperature near the setpoint
- Reduce fluctuations and disturbances
- Improve plant growth through stable conditions

Overall, PID control upgrades the system from passive monitoring to **automatic climate regulation**, making it more suitable for long-term greenhouse use.

Acknowledgment

We would like to thank Shiv Nadar University for providing laboratory resources and making it very accessible for us to use, Prof. Rohit Sharma and Prof. Sonal Singhal, for giving us this opportunity to explore, learn and create this project.

References

[1] STMicroelectronics, "STM32F303 Reference Manual," 2024.
[2] Aosong, "DHT22 Datasheet," 2022.
[3] Winsen, "MQ-135 Technical Data," 2020.
[4] STMicroelectronics, "STM32F303xD/E Datasheet," 2024. Available: https://www.st.com/resource/en/datasheet/stm32f303re.pdf
[5] STMicroelectronics, "Getting started with ADC," STM32 MCU Wiki. Available: https://wiki.st.com/stm32mcu/wiki/Getting_started_with_ADC
[6] STMicroelectronics, "Getting started with TIM," STM32 MCU Wiki. Available: https://wiki.st.com/stm32mcu/wiki/Getting_started_with_TIM
[7] Stack Overflow, "1602 LCD module showing garbage output," 2020. Available: https://stackoverflow.com/questions/60701539/1602-lcd-module-showing-garbage-output
[8] HomeGrail, "Ideal greenhouse temperature and humidity," accessed 2025. Available: https://homegrail.com/ideal-greenhouse-temperature-humidity/
[9] Family Handyman, "What is the optimal temperature for a greenhouse?," accessed 2025. Available: https://www.familyhandyman.com/article/what-is-the-optimal-temperature-for-a-greenhouse/
[10] STMicroelectronics, "STM32 MCU Wiki main page," accessed 2025. Available: https://wiki.st.com/stm32mcu/wiki/Main_Page
[11] STMicroelectronics, "Getting started with I2C," STM32 MCU Wiki. Available: https://wiki.st.com/stm32mcu/wiki/Getting_started_with_I2C
[12] LCD-1602A Datasheet, "1602A Character LCD Module," accessed 2025. Available: https://datasheet4u.com/datasheets/CA/LCD-1602A/519148
[13] Components101, "LDR (Light Dependent Resistor) – Working and Datasheet," accessed 2025. Available: https://components101.com/resistors/ldr-datasheet
[14] SparkFun Electronics, "DHT22 Temperature/Humidity Sensor Hookup Guide / Datasheet," accessed 2025. Available: https://cdn.sparkfun.com/assets/f/7/d/9/c/DHT22.pdf
[15] Elprocus, "MQ135 Air Quality Sensor – Working and Applications," accessed 2025. Available: https://www.elprocus.com/mq135-air-quality-sensor/
[16] DeepBlue Embedded, "Arduino I2C LCD Tutorial," accessed 2025. Available: https://deepbluembedded.com/arduino-i2c-lcd/

Code Which Runs the Greenhouse Monitoring System

```
/* USER CODE END Header */

/* Includes */

#include "main.h"

/* Private includes*/

/* USER CODE BEGIN Includes */

#include <stdio.h>
#include "../../ECUAL/LCD16X2/LCD16X2.h"

/* USER CODE END Includes */

/* Private typedef */

/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define */

/* USER CODE BEGIN PD */

#define DHT22_PORT GPIOA
#define DHT22_PIN GPIO_PIN_4
#define MyLCD LCD16X2_1

/* USER CODE END PD */

/* Private macro */

/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables */

ADC_HandleTypeDef hadc1;
ADC_HandleTypeDef hadc2;
ADC_HandleTypeDef hadc3;

TIM_HandleTypeDef htim2;
UART_HandleTypeDef huart2;

/* USER CODE BEGIN PV */

int airQ_threshold=2780;
int temp_threshold=29;
int light_threshold=1200;

int airQ;
int temperature;
int hum1, hum2, temp1, temp2, sumMQ, checkMQ;
int temp_Celsius, temp_Fahrenheit, Humidity;
int ldr_val;

int pMillis, cMillis;

/* USER CODE END PV */

/* Private function prototypes */

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_ADC1_Init(void);
static void MX_TIM2_Init(void);
static void MX_ADC2_Init(void);
static void MX_ADC3_Init(void);

/* USER CODE BEGIN PFP */

#define PUTCHAR_PROTOTYPE int __io_putchar(int ch)

/* USER CODE END PFP */

/* Private user code */

/* USER CODE BEGIN 0 */

// 1. Microsecond Delay Helper

void microDelay (uint16_t delay)
{
	__HAL_TIM_SET_COUNTER(&htim2, 0);
	while (__HAL_TIM_GET_COUNTER(&htim2) < delay);
}

// 2. Set Pin Direction Helpers

void Set_Pin_Output (GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin)
{
	GPIO_InitTypeDef GPIO_InitStruct = {0};
	GPIO_InitStruct.Pin = GPIO_Pin;
	GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
	GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH; // Changed to HIGH for better
	GPIO_InitStruct.Pull = GPIO_NOPULL;
	HAL_GPIO_Init(GPIOx, &GPIO_InitStruct);
}

void Set_Pin_Input (GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin)
{
	GPIO_InitTypeDef GPIO_InitStruct = {0};
	GPIO_InitStruct.Pin = GPIO_Pin;
	GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
	GPIO_InitStruct.Pull = GPIO_PULLUP; // Enable internal pullup just in case
	HAL_GPIO_Init(GPIOx, &GPIO_InitStruct);
}

// 3. DHT22 Start Signal

uint8_t DHT22_Start (void)
{
	uint8_t Response = 0;

	Set_Pin_Output(DHT22_PORT, DHT22_PIN);
	HAL_GPIO_WritePin (DHT22_PORT, DHT22_PIN, 0);

	HAL_Delay(18); // Wait 18ms

	HAL_GPIO_WritePin (DHT22_PORT, DHT22_PIN, 1);

	microDelay (30);

	Set_Pin_Input(DHT22_PORT, DHT22_PIN);

	microDelay (40);

	if (!(HAL_GPIO_ReadPin (DHT22_PORT, DHT22_PIN)))
	{
		microDelay (80);
		if ((HAL_GPIO_ReadPin (DHT22_PORT, DHT22_PIN))) Response = 1;
		else Response = -1;
	}

// Wait for the sensor to finish its response (High signal)

// We use a simple timeout loop instead of HAL_GetTick to be faster

	uint32_t timeout = 0;

	while ((HAL_GPIO_ReadPin (DHT22_PORT, DHT22_PIN)))
	{
		timeout++;
		if(timeout > 10000) break; // Prevent hanging
	}

	return Response;
}

// 4. DHT22 Read Data

uint8_t DHT22_Read (void)
{
	uint8_t x,y = 0;
	for (x=0; x<8; x++)
	{
		uint32_t timeout = 0;

// Wait for the pin to go HIGH (Start of bit transmission)

		while (!(HAL_GPIO_ReadPin (DHT22_PORT, DHT22_PIN)))
		{
			timeout++;
			if(timeout > 10000) return 0; // Timeout
		}
		microDelay(40); // Wait 40us.

// If bit is 0, signal is high for 26us (so it will be LOW now)

// If bit is 1, signal is high for 70us (so it will be HIGH now)

		if (!(HAL_GPIO_ReadPin (DHT22_PORT, DHT22_PIN)))
			y&= ~(1<<(7-x)); // It's a 0
		else
			y|= (1<<(7-x)); // It's a 1

// Wait for the pin to go LOW (End of bit)

		timeout = 0;

		while ((HAL_GPIO_ReadPin (DHT22_PORT, DHT22_PIN)))
```

```
                {
                        timeout++;
                        if(timeout > 10000) return 0; // Timeout
                }
        }
        return y;
}

/* USER CODE END 0 */

/**
* @brief The application entry point.
* @retval int
*/

int main(void)
{
        /* USER CODE BEGIN 1 */
        /* USER CODE END 1 */

        /* MCU Configuration*/

        /* Reset of all peripherals, Initializes the Flash interface and the Systick. */

        HAL_Init();

        /* USER CODE BEGIN Init */
        /* USER CODE END Init */
        /* Configure the system clock */

        SystemClock_Config();

        /* USER CODE BEGIN SysInit */
        /* USER CODE END SysInit */
        /* Initialize all configured peripherals */

        MX_GPIO_Init();

        MX_USART2_UART_Init();

        MX_TIM2_Init();

        MX_ADC1_Init();
        MX_ADC2_Init();
        MX_ADC3_Init();

        /* USER CODE BEGIN 2 */

        HAL_TIM_Base_Start(&htim2);

//LCD

        LCD16X2_Init(MyLCD);
        LCD16X2_Clear(MyLCD);

        char lcd_buffer[32];

        char AirQ_string[30];
        sprintf(AirQ_string, "%d", airQ);

        char temperatureCelsius_string[30];
        sprintf(temperatureCelsius_string, "%d", temp_Celsius);

        char temperatureFahrenheit_string[30];
        sprintf(temperatureFahrenheit_string, "%d", temp_Fahrenheit);

        char humidity_string[30];
        sprintf(humidity_string, "%d", Humidity);

        char ldr_string[30];
        sprintf(ldr_string, "%d", ldr_val);

        LCD16X2_Set_Cursor(MyLCD, 1, 1);

        /* USER CODE END 2 */

        /* Infinite loop */

        /* USER CODE BEGIN WHILE */

        while (1)
        {
                HAL_ADC_Start(&hadc1);
                HAL_ADC_Start(&hadc3);

                DHT22_Start();

//MQ-135
                airQ = HAL_ADC_GetValue(&hadc1);
//MQ-135

                if(airQ>airQ_threshold) {
                        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_SET);
                }
                else {
                        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_RESET);
                }
//DHT22

hum1 = DHT22_Read();
hum2 = DHT22_Read();
temp1 = DHT22_Read();
temp2 = DHT22_Read();
sumMQ = DHT22_Read();
checkMQ = hum1 + hum2 + temp1 + temp2;

if (checkMQ == sumMQ)
{
        if (temp1>127)
        {
                temp_Celsius = temp2/10*(-1);
        }
        else
        {
                temp_Celsius = ((temp1<<8)|temp2)/10;
        }

        temp_Fahrenheit = temp_Celsius * 9/5 + 32;

        if(temp_Celsius > temp_threshold) {
                HAL_GPIO_WritePin(GPIOA, GPIO_PIN_8, GPIO_PIN_SET);
        }
        else {
                HAL_GPIO_WritePin(GPIOA, GPIO_PIN_8, GPIO_PIN_RESET)
        }

        Humidity = (float) ((hum1<<8)|hum2)/10;
}

//debug block- dht-22

        uint8_t presence = DHT22_Start();

                if(presence == 1)
                {
// Sensor Responded! Now read.
                        hum1 = DHT22_Read();
                        hum2 = DHT22_Read();
                        temp1 = DHT22_Read();
                        temp2 = DHT22_Read();
                        sumMQ = DHT22_Read();

// Calculate
                        checkMQ = hum1 + hum2 + temp1 + temp2;

// Combine bytes

                        Humidity = (float)((hum1<<8)|hum2)/10;
                        if (temp1>127) temp_Celsius = temp2/10*(-1);
                        else temp_Celsius = ((temp1<<8)|temp2)/10;

                        printf("Sensor Status: OK | Hum: %d %% | Temp: %d C\r\n", Hu
                }

                else
                {
                        printf("Sensor Status: ERROR. (Response: %d)\r\n", presence)
                }

                HAL_Delay(500); // Read every 2 seconds (DHT22 is slow)

//ldr

        ldr_val = HAL_ADC_GetValue(&hadc3);

                if(ldr_val > light_threshold) {
                        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_9, GPIO_PIN_SET); // Turn
                }
                else {
                        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_9, GPIO_PIN_RESET); // Tur
                }

//print-serial monitor

                printf("Air Quality: %d\n\r", airQ-airQ_threshold);
                printf("Temperature: %d deg C | %d deg F\n\r", temp_Celsius, temp_Fa
                printf("Humidity: %d %%\n\r", Humidity);
                printf("Light: %d \n\r", ldr_val-light_threshold);

//LCD

                  // Air and Temp printing
                  LCD16X2_Clear(MyLCD);

                  LCD16X2_Set_Cursor(MyLCD, 1, 1);
                  sprintf(lcd_buffer, "Air Q: %d", airQ-airQ_threshold);
                  LCD16X2_Write_String(MyLCD, lcd_buffer);

                  LCD16X2_Set_Cursor(MyLCD, 2, 1);
                  sprintf(lcd_buffer, "Temp: %d C", temp_Celsius);
```

```
                        LCD16X2_Write_String(MyLCD, lcd_buffer);

                        HAL_Delay(1000);

                        // Humidity and Light print
                        LCD16X2_Clear(MyLCD);

                        LCD16X2_Set_Cursor(MyLCD, 1, 1);
                        sprintf(lcd_buffer, "Hum: %d %%", Humidity);
                        LCD16X2_Write_String(MyLCD, lcd_buffer);

                        LCD16X2_Set_Cursor(MyLCD, 2, 1);
                        sprintf(lcd_buffer, "Light: %d ", ldr_val-light_threshold);
                        LCD16X2_Write_String(MyLCD, lcd_buffer);

                        HAL_Delay(1000);

                /* USER CODE END WHILE */

                /* USER CODE BEGIN 3 */
        }

        /* USER CODE END 3 */
}

/**

* @brief System Clock Configuration
* @retval None
*/

void SystemClock_Config(void)
{
        RCC_OscInitTypeDef RCC_OscInitStruct = {0};
        RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
        RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};

        /** Initializes the RCC Oscillators according to the specified parameters

        * in the RCC_OscInitTypeDef structure.

        */
        RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
        RCC_OscInitStruct.HSEState = RCC_HSE_ON;
        RCC_OscInitStruct.HSIState = RCC_HSI_ON;
        RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
        RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
        RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL9;
        RCC_OscInitStruct.PLL.PREDIV = RCC_PREDIV_DIV1;

        if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
        {
                Error_Handler();
        }

        /** Initializes the CPU, AHB and APB buses clocks
        */

        RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                                      |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
        RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
        RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
        RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
        RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

        if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) != HAL_OK)
        {
                Error_Handler();
        }

        PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_USART2|RCC_PERIPHCLK_ADC12
                                             |RCC_PERIPHCLK_ADC34|RCC_PERIPHCLK_TIM2;
        PeriphClkInit.Usart2ClockSelection = RCC_USART2CLKSOURCE_PCLK1;
        PeriphClkInit.Adc12ClockSelection = RCC_ADC12PLLCLK_DIV1;
        PeriphClkInit.Adc34ClockSelection = RCC_ADC34PLLCLK_DIV1;
        PeriphClkInit.Tim2ClockSelection = RCC_TIM2CLK_HCLK;
        if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
        {
                Error_Handler();
        }

}

/**

* @brief ADC1 Initialization Function
* @param None
* @retval None
*/

static void MX_ADC1_Init(void)
{
        /* USER CODE BEGIN ADC1_Init 0 */
        /* USER CODE END ADC1_Init 0 */

        ADC_MultiModeTypeDef multimode = {0};
        ADC_ChannelConfTypeDef sConfig = {0};

        /* USER CODE BEGIN ADC1_Init 1 */
        /* USER CODE END ADC1_Init 1 */

        /** Common config

        */

        hadc1.Instance = ADC1;
        hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
        hadc1.Init.Resolution = ADC_RESOLUTION_12B;
        hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
        hadc1.Init.ContinuousConvMode = DISABLE;
        hadc1.Init.DiscontinuousConvMode = DISABLE;
        hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
        hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
        hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
        hadc1.Init.NbrOfConversion = 1;
        hadc1.Init.DMAContinuousRequests = DISABLE;

        hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
        hadc1.Init.LowPowerAutoWait = DISABLE;
        hadc1.Init.Overrun = ADC_OVR_DATA_OVERWRITTEN;

        if (HAL_ADC_Init(&hadc1) != HAL_OK)
        {
                Error_Handler();
        }

        /** Configure the ADC multi-mode
        */
        multimode.Mode = ADC_MODE_INDEPENDENT;
        if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode) != HAL_OK)
        {
                Error_Handler();
        }

        /** Configure Regular Channel

        */

        sConfig.Channel = ADC_CHANNEL_1;
        sConfig.Rank = ADC_REGULAR_RANK_1;
        sConfig.SingleDiff = ADC_DIFFERENTIAL_ENDED;
        sConfig.SamplingTime = ADC_SAMPLETIME_1CYCLE_5;
        sConfig.OffsetNumber = ADC_OFFSET_NONE;
        sConfig.Offset = 0;

        if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
        {
                Error_Handler();
        }

        /* USER CODE BEGIN ADC1_Init 2 */
        /* USER CODE END ADC1_Init 2 */
}

/**
* @brief ADC2 Initialization Function
* @param None
* @retval None
*/
static void MX_ADC2_Init(void)
{

        /* USER CODE BEGIN ADC2_Init 0 */
        /* USER CODE END ADC2_Init 0 */
        ADC_ChannelConfTypeDef sConfig = {0};

        /* USER CODE BEGIN ADC2_Init 1 */
        /* USER CODE END ADC2_Init 1 */

        /** Common config

        */

        hadc2.Instance = ADC2;
        hadc2.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
        hadc2.Init.Resolution = ADC_RESOLUTION_12B;
        hadc2.Init.ScanConvMode = ADC_SCAN_DISABLE;
        hadc2.Init.ContinuousConvMode = DISABLE;
        hadc2.Init.DiscontinuousConvMode = DISABLE;
        hadc2.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
```

```
	hadc2.Init.ExternalTrigConv = ADC_SOFTWARE_START;
	hadc2.Init.DataAlign = ADC_DATAALIGN_RIGHT;
	hadc2.Init.NbrOfConversion = 1;
	hadc2.Init.DMAContinuousRequests = DISABLE;
	hadc2.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
	hadc2.Init.LowPowerAutoWait = DISABLE;
	hadc2.Init.Overrun = ADC_OVR_DATA_OVERWRITTEN;

	if (HAL_ADC_Init(&hadc2) != HAL_OK)
	{
		Error_Handler();
	}

	/** Configure Regular Channel

	*/

	sConfig.Channel = ADC_CHANNEL_1;
	sConfig.Rank = ADC_REGULAR_RANK_1;
	sConfig.SingleDiff = ADC_DIFFERENTIAL_ENDED;
	sConfig.SamplingTime = ADC_SAMPLETIME_1CYCLE_5;
	sConfig.OffsetNumber = ADC_OFFSET_NONE;
	sConfig.Offset = 0;

	if (HAL_ADC_ConfigChannel(&hadc2, &sConfig) != HAL_OK)
	{
		Error_Handler();
	}

	/* USER CODE BEGIN ADC2_Init 2 */

	/* USER CODE END ADC2_Init 2 */

}

/**
* @brief ADC3 Initialization Function
* @param None
* @retval None
*/

static void MX_ADC3_Init(void)
{
	/* USER CODE BEGIN ADC3_Init 0 */

	/* USER CODE END ADC3_Init 0 */

	ADC_MultiModeTypeDef multimode = {0};
	ADC_ChannelConfTypeDef sConfig = {0};

	/* USER CODE BEGIN ADC3_Init 1 */

	/* USER CODE END ADC3_Init 1 */

	/** Common config

	*/

	hadc3.Instance = ADC3;
	hadc3.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
	hadc3.Init.Resolution = ADC_RESOLUTION_12B;
	hadc3.Init.ScanConvMode = ADC_SCAN_DISABLE;
	hadc3.Init.ContinuousConvMode = DISABLE;
	hadc3.Init.DiscontinuousConvMode = DISABLE;
	hadc3.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
	hadc3.Init.ExternalTrigConv = ADC_SOFTWARE_START;
	hadc3.Init.DataAlign = ADC_DATAALIGN_RIGHT;
	hadc3.Init.NbrOfConversion = 1;
	hadc3.Init.DMAContinuousRequests = DISABLE;
	hadc3.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
	hadc3.Init.LowPowerAutoWait = DISABLE;
	hadc3.Init.Overrun = ADC_OVR_DATA_OVERWRITTEN;

	if (HAL_ADC_Init(&hadc3) != HAL_OK)
	{
		Error_Handler();
	}

	/** Configure the ADC multi-mode

	*/

	multimode.Mode = ADC_MODE_INDEPENDENT;

	if (HAL_ADCEx_MultiModeConfigChannel(&hadc3, &multimode) != HAL_OK)
	{
		Error_Handler();
	}

	/** Configure Regular Channel

	*/

	sConfig.Channel = ADC_CHANNEL_12;
	sConfig.Rank = ADC_REGULAR_RANK_1;
	sConfig.SingleDiff = ADC_SINGLE_ENDED;
	sConfig.SamplingTime = ADC_SAMPLETIME_1CYCLE_5;
	sConfig.OffsetNumber = ADC_OFFSET_NONE;
	sConfig.Offset = 0;

	if (HAL_ADC_ConfigChannel(&hadc3, &sConfig) != HAL_OK)
	{
		Error_Handler();
	}

	/* USER CODE BEGIN ADC3_Init 2 */

	/* USER CODE END ADC3_Init 2 */

}

/**

* @brief TIM2 Initialization Function

* @param None

* @retval None

*/

static void MX_TIM2_Init(void)

{

	/* USER CODE BEGIN TIM2_Init 0 */

	/* USER CODE END TIM2_Init 0 */

	TIM_ClockConfigTypeDef sClockSourceConfig = {0};
	TIM_MasterConfigTypeDef sMasterConfig = {0};
	TIM_OC_InitTypeDef sConfigOC = {0};


	/* USER CODE BEGIN TIM2_Init 1 */

	/* USER CODE END TIM2_Init 1 */

	htim2.Instance = TIM2;
	htim2.Init.Prescaler = 71;
	htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
	htim2.Init.Period = 65535;
	htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
	htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;

	if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
	{
		Error_Handler();
	}

	sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;

	if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
	{
		Error_Handler();
	}

	if (HAL_TIM_PWM_Init(&htim2) != HAL_OK)
	{
		Error_Handler();
	}

	sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
	sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;

	if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
	{
		Error_Handler();
	}

	sConfigOC.OCMode = TIM_OCMODE_PWM1;
	sConfigOC.Pulse = 0;
	sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
	sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;

	if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_1) != HAL_OK)
	{
		Error_Handler();
	}

	/* USER CODE BEGIN TIM2_Init 2 */

	/* USER CODE END TIM2_Init 2 */

	HAL_TIM_MspPostInit(&htim2);

}
```

```
/**
* @brief USART2 Initialization Function
* @param None
* @retval None
*/
static void MX_USART2_UART_Init(void)
{
  /* USER CODE BEGIN USART2_Init 0 */
  /* USER CODE END USART2_Init 0 */
  /* USER CODE BEGIN USART2_Init 1 */
  /* USER CODE END USART2_Init 1 */
  huart2.Instance = USART2;
  huart2.Init.BaudRate = 115200;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
  huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
  if (HAL_UART_Init(&huart2) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART2_Init 2 */
  /* USER CODE END USART2_Init 2 */
}
/**
* @brief GPIO Initialization Function
* @param None
* @retval None
*/
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
  /* USER CODE BEGIN MX_GPIO_Init_1 */
  /* USER CODE END MX_GPIO_Init_1 */
  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOF_CLK_ENABLE();
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();
  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6|GPIO_PIN_8|GPIO_PIN_9|GPIO_PIN_10, GPIO_PIN_RESET);
  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOB, GPIO_PIN_10|GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5
                    |GPIO_PIN_6, GPIO_PIN_RESET);
  /*Configure GPIO pins : PA6 PA8 PA9 PA10 */
  GPIO_InitStruct.Pin = GPIO_PIN_6|GPIO_PIN_8|GPIO_PIN_9|GPIO_PIN_10;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
  /*Configure GPIO pins : PB10 PB3 PB4 PB5
  PB6 */
  GPIO_InitStruct.Pin = GPIO_PIN_10|GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5
                        |GPIO_PIN_6;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
  /* USER CODE BEGIN MX_GPIO_Init_2 */
  /* USER CODE END MX_GPIO_Init_2 */
}
/* USER CODE BEGIN 4 */
PUTCHAR_PROTOTYPE
{
  HAL_UART_Transmit(&huart2, (uint8_t *)&ch, 1, 0xFFFF);
  return ch;
}
/* USER CODE END 4 */
/**
* @brief This function is executed in case of error occurrence.
* @retval None
*/
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}
#ifdef USE_FULL_ASSERT
/**
* @brief Reports the name of the source file and the source line number
* where the assert_param error has occurred.
* @param file: pointer to the source file name
* @param line: assert_param error line source number
* @retval None
*/
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line numb
  ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```