

Project Report: ESP32 Smart Fan Controller with FreeRTOS

Students Names/ID's: Yahia Sameh(13001675) , Omar Medhat(13001677), Mahmoud Mabrouk(16001977), Ahmed Adel(19001082)

1. Introduction

Temperature regulation is a critical requirement in modern home automation and industrial electronics. Traditional manual fan operation is often inefficient, leading to unnecessary energy consumption or overheating due to human error.

This project implements a **Smart Fan Controller** using the ESP32 microcontroller and the **FreeRTOS** real-time operating system. The system automates the cooling process by continuously monitoring temperature via a DHT22 sensor and adjusting a DC fan's speed using Pulse Width Modulation (PWM). By leveraging FreeRTOS, the system demonstrates the ability to handle multiple concurrent operations—sensing, processing, user input, and display—without blocking execution, ensuring a responsive and reliable embedded system.

2. Objectives

The primary objectives of this project were achieved as follows:

- **Automatic Monitoring:** Implemented a DHT22 sensor interface to read ambient temperature.
- **Proportional Control:** Utilized PWM (Pulse Width Modulation) to adjust fan speed dynamically based on thermal thresholds.
- **Real-Time Multitasking:** Deployed FreeRTOS to separate critical control tasks from non-critical user interface tasks.
- **User Interface:** Integrated an OLED display to visualize real-time data and physical buttons for manual override control.

3. System Design and Architecture

3.1 Overview

The system is designed around a modular task-based architecture. Unlike a traditional "Super Loop" architecture, this system uses the FreeRTOS scheduler to manage four independent tasks. Data integrity between these tasks is maintained using Inter-Process Communication (IPC) mechanisms: **Queues** for data transfer and **Mutexes** for shared state protection.

3.2 Task Definition and Priorities

The system is divided into the following tasks based on criticality:

Task Name	Priority	Periodicity	Description
TaskTempRead	High (2)	500 ms	Reads the DHT22 sensor. High priority ensures the control loop always has fresh data.
TaskFanControl	High (2)	200 ms	Calculates required speed and updates PWM hardware. High priority ensures immediate reaction to changes.
TaskButtonInput	Medium (1)	100 ms	Polls GPIO pins for user input. Medium priority ensures the UI is responsive but does not interrupt critical control loops.
TaskDisplay	Low (0)	1000 ms	Updates the OLED screen. Lowest priority as visual latency (approx. 1s) is acceptable to save CPU cycles for control logic.

3.3 Data Flow Architecture

- Sensing:** TaskTempRead acquires data and pushes it to a FreeRTOS **Queue** (xTemperatureQueue). This decouples the slow sensor reading (approx. 2ms) from the rest of the system.
- Processing:** TaskFanControl pulls data from the Queue. It checks the global SystemMode (protected by xMutex).
 - If **AUTO**: It maps temperature to speed.
 - If **MANUAL**: It uses the speed set by the buttons.
-
- Actuation:** The fan speed is converted to a PWM duty cycle (0-255) and written to GPIO 25.
- Visualization:** TaskDisplay acquires the Mutex to safely read the current state variables and updates the OLED.

```

// Create FreeRTOS Tasks
xTaskCreatePinnedToCore(
    TaskTemperatureRead,           // Task function
    "TempRead",                   // Task name
    4096,                          // Stack size
    NULL,                          // Parameters
    2,                             // Priority (High)
    &xTempTaskHandle,              // Task handle
    0                             // Core 0
);

xTaskCreatePinnedToCore(
    TaskFanControl,               // Task function
    "FanControl",                 // Task name
    4096,                          // Stack size
    NULL,                          // Parameters
    2,                             // Priority (High)
    &xControlTaskHandle,          // Task handle
    0                             // Core 0
);

xTaskCreatePinnedToCore(
    TaskButtonInput,              // Task function
    "ButtonInput",                // Task name
    2048,                          // Stack size
    NULL,                          // Parameters
    1,                             // Priority (Medium)
    &xButtonTaskHandle,           // Task handle
    1                             // Core 1
);

xTaskCreatePinnedToCore(
    TaskDisplay,                  // Task function
    "Display",                     // Task name
    4096,                          // Stack size
    NULL,                          // Parameters
    0,                             // Priority (Low)
    &xDisplayTaskHandle,          // Task handle
    1                             // Core 1
);

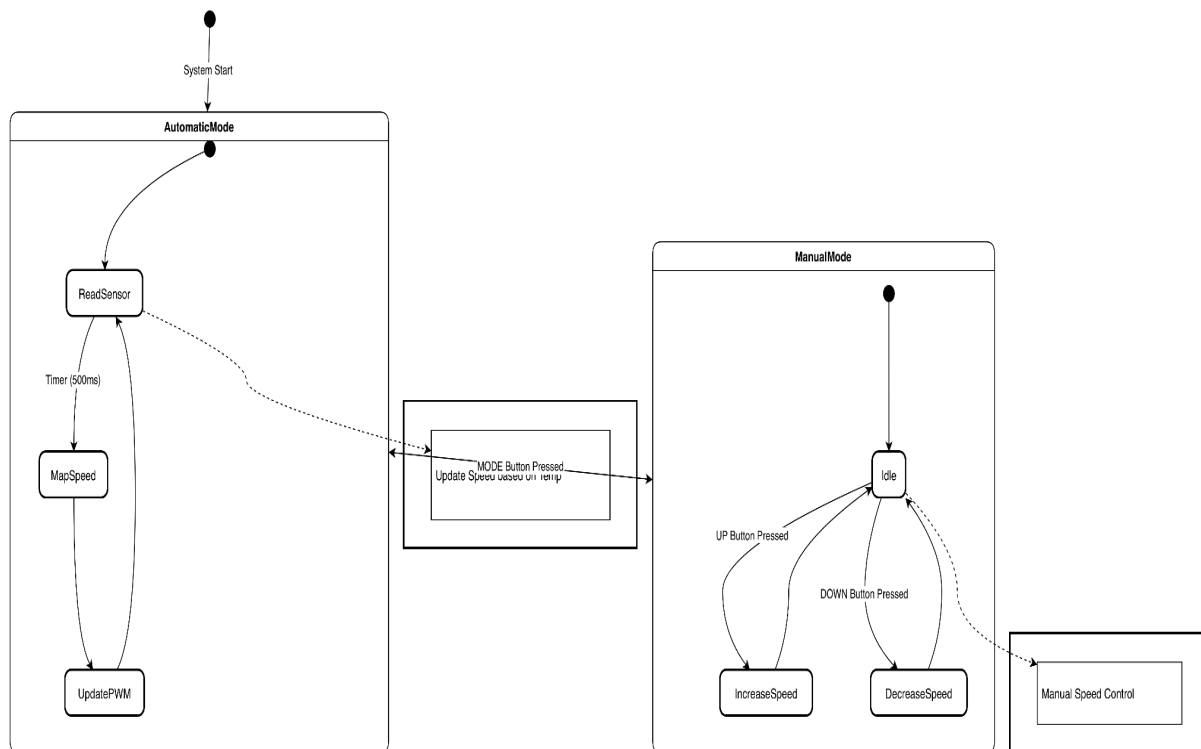
```

Figure 1: FreeRTOS Task Creation and Configuration

4. UML Modeling

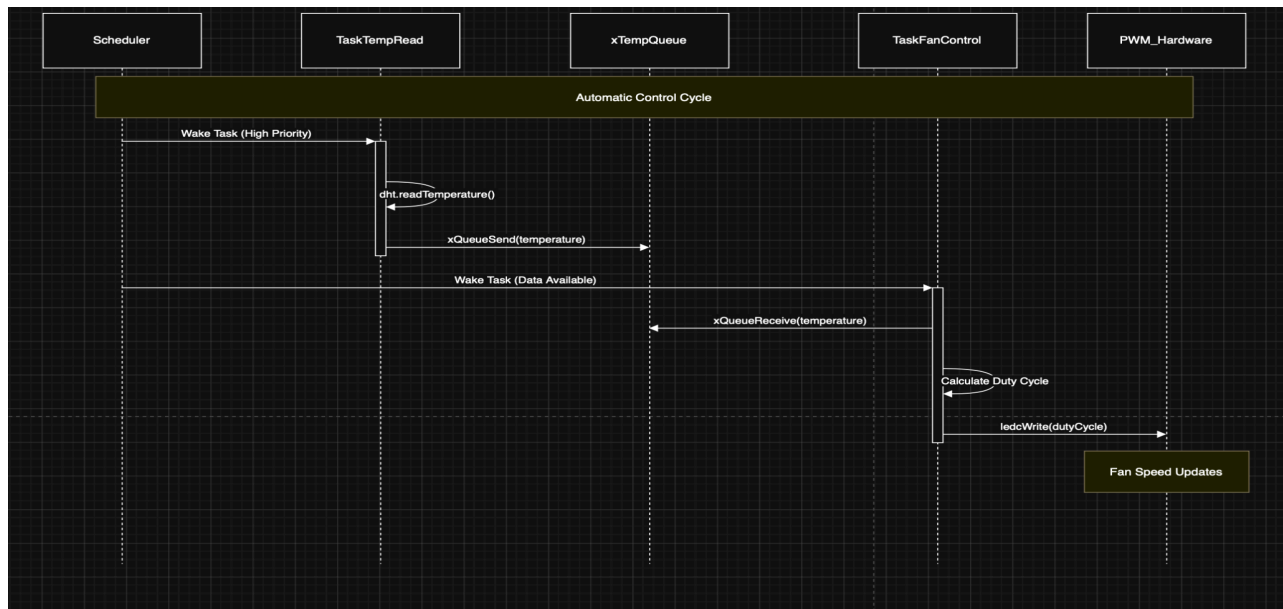
4.1 State Machine Diagram

The system operates in two distinct states. The transition is triggered by the generic "Mode" button.



4.2 Sequence Diagram (Automatic Cycle)

The following sequence describes one regulation cycle:



5. Hardware Implementation

5.1 Schematic Diagram

The hardware is connected according to the following simulation schematic.

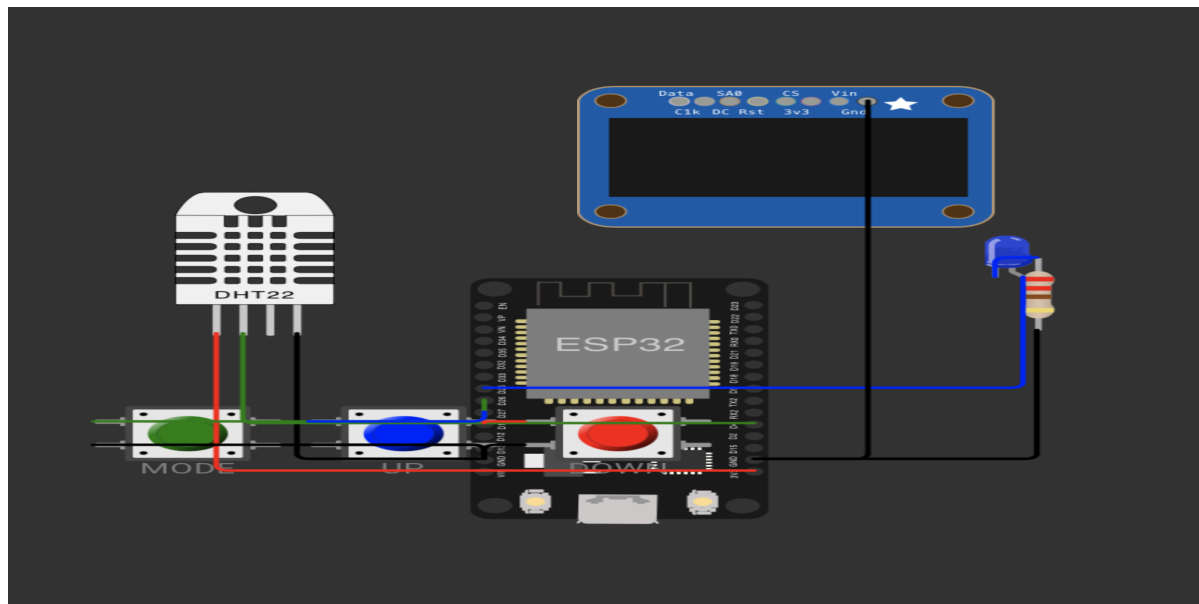


Figure 4: System Wiring Schematic from Wokwi

5.2 Pin Assignment Justification

- **GPIO 25 (Fan PWM):** Selected because it is a clean GPIO pin not used for bootstrapping the ESP32, ensuring the fan does not spin randomly during boot.
- **GPIO 26, 27, 14 (Buttons):** Configured as INPUT_PULLUP. This utilizes the ESP32's internal resistors, eliminating the need for external resistors on the breadboard.
- **GPIO 21 (SDA) & 22 (SCL):** These are the default I2C hardware pins for the ESP32, providing the fastest communication speed for the OLED display.

6. PWM Control Strategy

Pulse Width Modulation (PWM) is used to control the fan speed. The ESP32 ledc peripheral is configured as follows:

- **Frequency:** 25 kHz (Standard for DC fans to reduce audible noise).
- **Resolution:** 8-bit (Providing 256 discrete speed steps).

Control Algorithm (Automatic Mode):

The system uses linear interpolation to map temperature ranges to fan speeds:

- **< 20°C:** Fan OFF (0%)
- **20°C - 25°C:** Linear ramp 0% to 25%
- **25°C - 30°C:** Linear ramp 25% to 50%
- **30°C - 35°C:** Linear ramp 50% to 75%
- **35°C - 40°C:** Linear ramp 75% to 100%
- **> 40°C:** Max Speed (100%)

7. CPU and Memory Utilization

Efficiency was a key design constraint.

7.1 Memory (Stack) Allocation

- **Heavy Tasks (4096 Bytes):** The TempRead and Display tasks handle floating-point calculations and I2C communication (via the Adafruit GFX library), which are memory-intensive. A stack size of 4KB prevents stack overflow.
- **Light Tasks (2048 Bytes):** The ButtonInput task performs simple digital reads and boolean logic, requiring significantly less RAM.

7.2 CPU Efficiency

The design avoids "Busy Waiting" (e.g., `delay()`). Instead, it uses `vTaskDelayUntil()`. This function puts the task into a **Blocked State**, allowing the FreeRTOS scheduler to execute the Idle task or lower-priority tasks (like the Display) while waiting. This maximizes CPU efficiency and reduces power consumption.

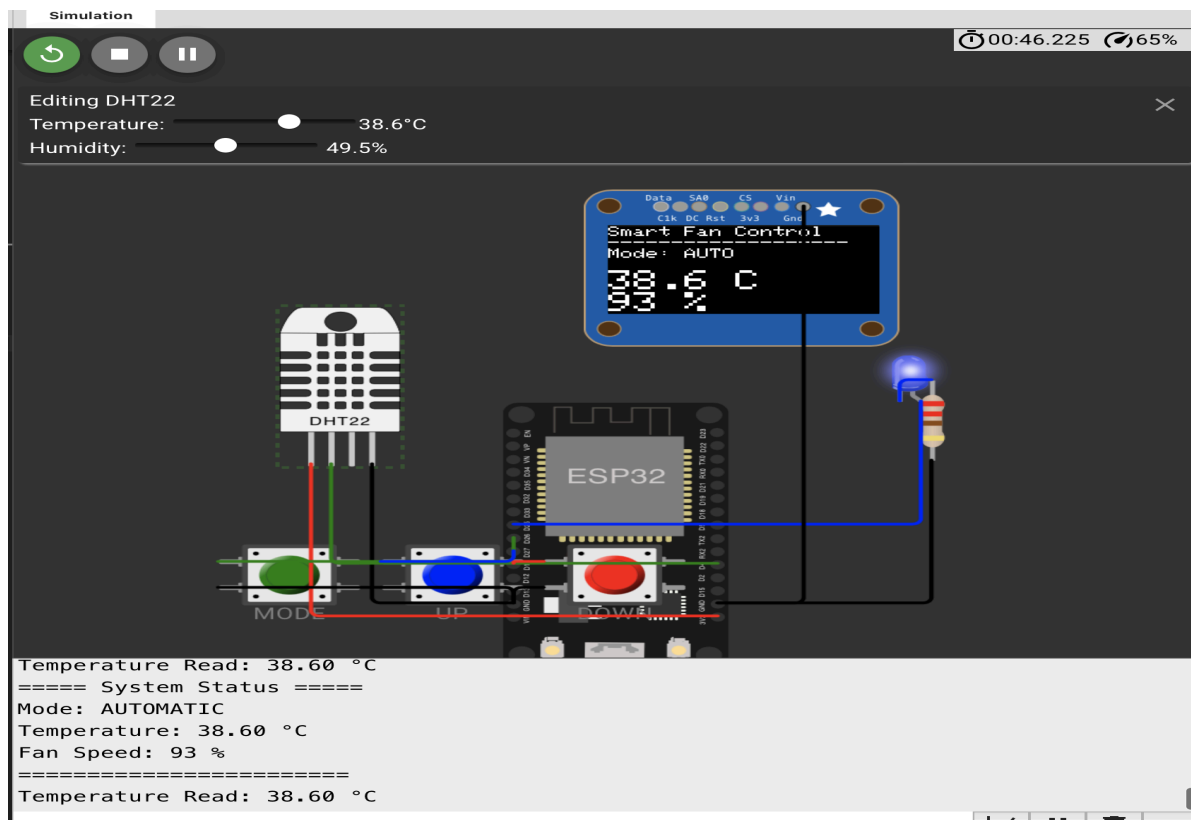
8. Simulation Results

The system was simulated using the Wokwi environment.

Simulation Link: <https://wokwi.com/projects/448972200491964417>

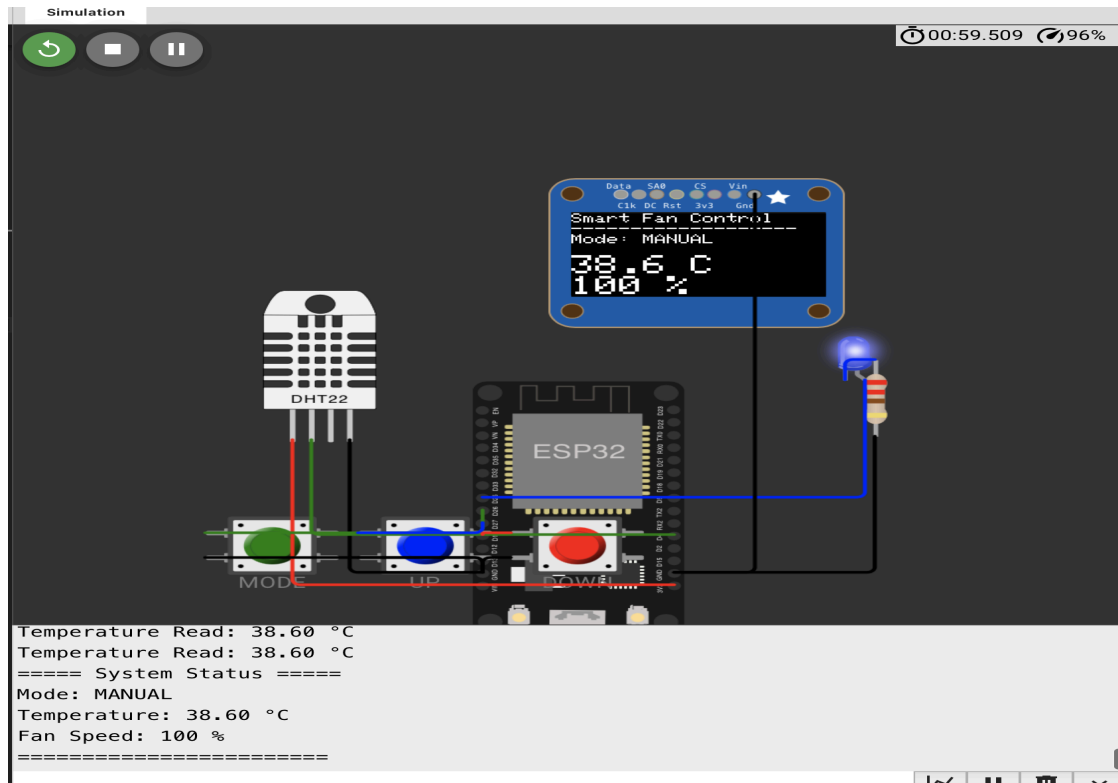
8.1 Test Case: Automatic Mode

When the DHT22 temperature was increased from 20°C to 35°C, the Fan LED brightness increased proportionally, and the OLED updated the speed percentage.



8.2 Test Case: Manual Mode

Pressing the Green "Mode" button successfully switched the system to Manual. Pressing the "Up" button increased the LED brightness regardless of the sensor temperature.



9. Conclusion

This project successfully implemented a Smart Fan Controller utilizing the multitasking capabilities of the ESP32 and FreeRTOS. By segregating sensor reading, control logic, and user interface into concurrent tasks, the system achieves high reliability and responsiveness. The use of synchronization primitives (Mutexes and Queues) prevented race conditions, ensuring data integrity across the system.