ARGB PC Fan Controller with Wireless Remote

1. Introduction

The project represents an intelligent and independent system for controlling the cooling and lighting (**ARGB**) inside a PC case. It is based on a two-node architecture that communicates wirelessly via Wi-Fi protocol, eliminating the need for long cables and bloatware software installed on the operating system.

Project Objectives:

- Monitoring internal PC temperature.
- Automatic/manual RPM (PWM) control.
- Managing ARGB visual effects.
- Remote control interface.

2. Project Description (What it does)

The system is divided into two main units working together:

A. Central Unit (The Controller)

It is the heart of the system, implemented on a **T-HMI (ESP32-S3)** module with an integrated LCD display.

- **DHT22 Sensor:** Constantly reads the case temperature.
- **ARGB Control:** Manages the colors and modes of the Thermalright fan.
- **Real-time Display:** Projects the RPM, temperature, and current mode on the screen.

B. The Remote (Control Unit)

Built on an **ATmega328P Xplained Mini** board connected to an **ESP-01S** module.

- Features an expansion board with **5 buttons**.

- Allows changing operating modes and speed remotely.
- Externally powered for total portability.

3. Operating Modes

- **Auto Mode (Default):**
  - The system automatically adjusts the fan speed using a temperature curve.
  - For every **5°C** threshold reached, the fan speed progressively increases or decreases to maintain the balance between cooling and noise.
- **Manual Mode:**
  - The user takes full control via the remote.
  - Dedicated buttons can be used to increase or decrease the speed regardless of the read temperature.

4. Project Purpose and Motivation

What is its purpose?

The primary purpose is to provide a user-friendly hardware interface that allows users to easily monitor and control fan speeds and ARGB lighting. While connecting a fan directly to the motherboard provides basic cooling, this project focuses on delivering accessible, real-time control without relying on proprietary software. It essentially protects the user from the risk of losing manufacturer support (abandonware), or turning into resource-heavy bloatware over time.

Origin of the Idea

The idea stemmed from the frustration caused by software dependency and forced firmware updates of commercial products, which often break functionality or add unnecessary background processes. I wanted a hardware-level “Plug & Play” device that guarantees longevity and provides clear information on a physical display, completely independent of the PC's software ecosystem.

Usefulness for users

- **For me:** It gives me precise, physical control and immediate visual monitoring over my PC's thermal state and aesthetics without tabbing out of applications or games.
- **For others:** The same as above, but also **Future-Proof & No Obsolescence:** Since it doesn't require a dedicated app, it will never lose functionality due to lack of software support from a manufacturer.
  - **OS Independence:** Works on Windows, Linux, or macOS seamlessly, without requiring any drivers.

  - **Zero Resource Consumption:** Does not load the PC's processor or RAM (crucial for gamers or heavy-duty tasks).
  - **Customization:** While the interface is simple, it ofers all the customization a normal user needs.

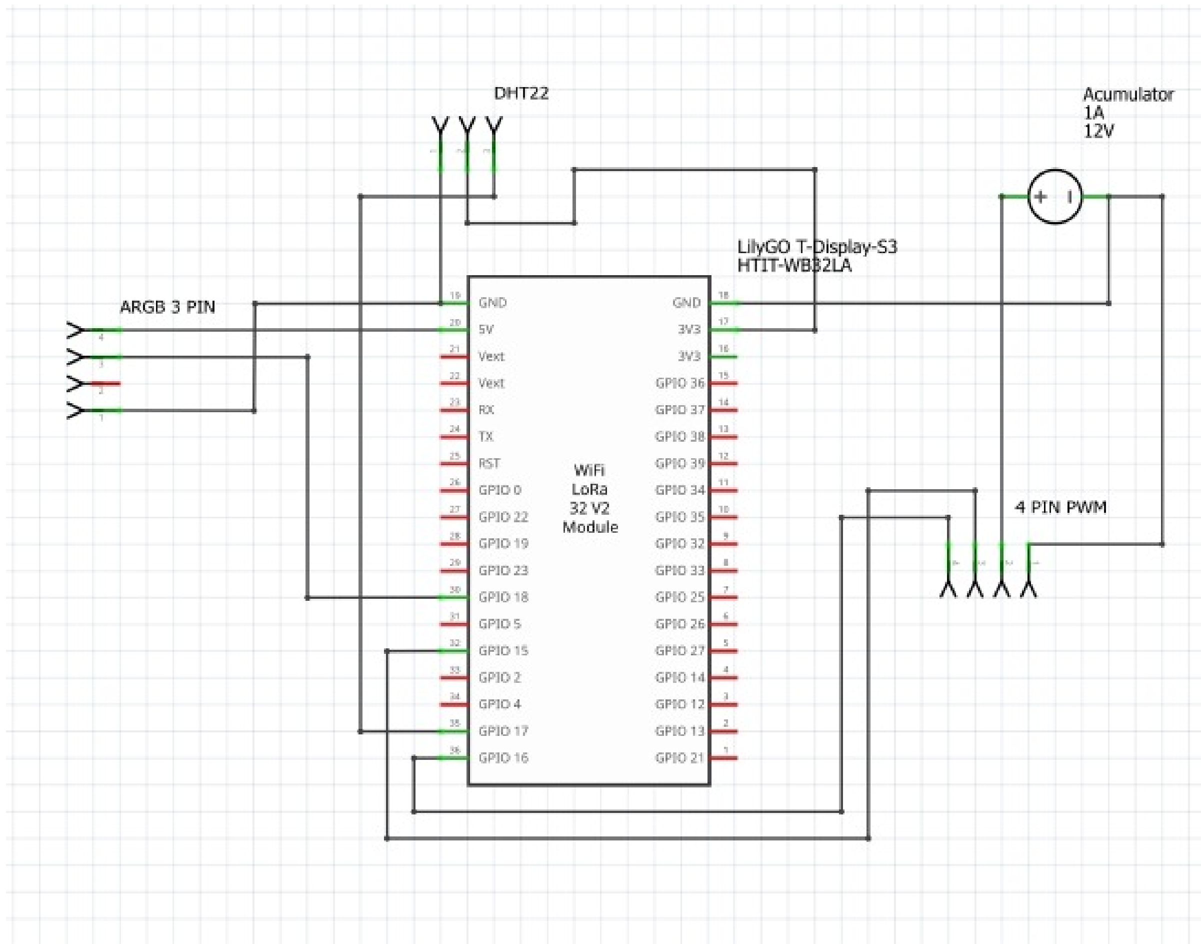
5. Hardware Components Used

- **Microcontrollers:** ESP32-S3 (Central unit) and ATmega328P Xplained Mini (Remote).
- **Remote Communication:** ESP8266 ESP-01S Wi-Fi module.
- **Sensor:** AM2302 (DHT22) for temperature and humidity.
- **Actuator:** Thermalright C12CW-S ARGB Fan.
- **Interface:** LCD Display (integrated on T-HMI) and a 5-button module.
- **Power Management:**
  - **Remote:** External battery and an AMS1117-3.3V voltage regulator (to step down voltage for the Wi-Fi module).
  - **Controller:** Laptop USB-C connection (5V for the ESP32 logic and ARGB LEDs), a 12V 1A Wall Adapter, and a DC Screw Terminal (for the fan motor).

Block Schema

Hardware Design

Schema electrica controller:



Schema electrica telecomanda

Hardware Pin Configuration

1. Remote Node (Telecomanda)

The Remote Node utilizes an ATmega328P microcontroller communicating with an ESP8266 (ESP-201 breakout format) Wi-Fi module. Voltage regulation from 5V to 3.3V is handled by an AMS1117-3.3V

linear regulator.

1.1 ATmega328P Pin Mapping

ATmega328P Pin (Arduino Label)	Connected To	Component / Signal Description
PD0 (D0/RX)	ESP8266 TX	UART Serial Data Receive
PD1 (D1/TX)	ESP8266 RX	UART Serial Data Transmit
PD2 (D2)	Button S1 (Pin 1)	Digital Input for Button 1 (Internal Pull-up)
PD3 (D3)	Button S2 (Pin 1)	Digital Input for Button 2 (Internal Pull-up)
PD4 (D4)	Button S3 (Pin 1)	Digital Input for Button 3 (Internal Pull-up)
PD5 (D5)	Button S4 (Pin 1)	Digital Input for Button 4 (Internal Pull-up)
PD6 (D6)	Button S5 (Pin 1)	Digital Input for Button 5 (Internal Pull-up)
5V	AMS1117 IN (Pin 2)	5V DC Power Source Output from USB
GND	Main GND Rail	Common System Ground Reference

1.2 ESP8266 Wi-Fi Module (ESP-201)

ESP8266 Pin	Connected To	Signal Description
TX	ATmega328P PD0 (D0/RX)	Serial Transmit Data
RX	ATmega328P PD1 (D1/TX)	Serial Receive Data
3.3V	AMS1117 OUT (Pin 3)	Regulated 3.3V Power Input
CHIP_EN	AMS1117 OUT (Pin 3)	Chip Enable Pin (Pulled HIGH to activate)
GND	Main GND Rail	System Ground Reference

1.3 AMS1117-3.3V Voltage Regulator

AMS1117 Pin	Connected To	Function
IN (Pin 2)	ATmega328P 5V	5V DC Input from Microcontroller
OUT (Pin 3)	ESP8266 3.3V & CHIP_EN	3.3V Regulated Output (Max 800mA)
GND (Pin 1)	Main GND Rail	System Ground Reference

1.4 Input Buttons (S1 - S5)

Button Component	Pin 1 (Green Side)	Pin 2 (Red Side)
S1	ATmega328P PD2 (D2)	Main GND Rail
S2	ATmega328P PD3 (D3)	Main GND Rail
S3	ATmega328P PD4 (D4)	Main GND Rail
S4	ATmega328P PD5 (D5)	Main GND Rail
S5	ATmega328P PD6 (D6)	Main GND Rail

2. Controller Node

2.1 Microcontroller Pin Mapping

Module Pin / GPIO	Connected To	Component / Signal Description
5V	ARGB Pin 2	5V VCC Power Output for LED Strip
3V3	DHT22 Pin 1	3.3V VCC Power Output for Sensor
GND	System GND Rail	Shared Reference Ground
GPIO 16	4 PIN PWM Pin 1	Fan Control / Tachometer / PWM Output
GPIO 17	DHT22 Pin 2	DHT22 Data Signal (Bidirectional)
GPIO 18	ARGB Pin 4	ARGB LED Data Signal Output

2.2 Peripheral Devices Connectivity

DHT22 Sensor

Sensor Pin	Connected To	Description
Pin 1	Module 3V3	Power Supply Input
Pin 2	Module GPIO 17	Data Signal Line
Pin 4	Module GND	Ground Reference

ARGB 3-PIN Connector

Connector Pin	Connected To	Description
Pin 1	System GND	Ground
Pin 2	Module 5V	5V DC Power Supply
Pin 3	*Unused*	Not Connected
Pin 4	Module GPIO 18	Data Signal Line

4-PIN PWM Fan Header

Header Pin	Connected To	Description
Pin 1	Module GPIO 16	Control/Feedback Signal 1
Pin 2	Module GPIO 21	Control/Feedback Signal 2
Pin 3	Accumulator (-) / GND	Ground Return Line
Pin 4	Accumulator (+)	+12V Dedicated Power Line

External Power (Accumulator)

Terminal	Connected To	Purpose
Positive (+)	4 PIN PWM Pin 4	High-power +12V Supply for the Fan
Negative (-)	System GND Rail	Common ground loop safety connection

Firmware Description & Software Architecture

1. Development Environment (IDE & Frameworks)

The project utilizes a hybrid software architecture tailored to the specific capabilities of each microcontroller. The entire system is managed via **Visual Studio Code (VS Code)** combined with the **PlatformIO** extension. This IDE was chosen over the standard Arduino IDE due to its superior toolchain management and the ability to handle independent compilations for different architectures (ESP32 and AVR) within the same environment.

- **Central Node (Controller - ESP32-S3 T-HMI):** The firmware is developed using the official **ESP-IDF (Espressif IoT Development Framework)** in C. ESP-IDF was selected to ensure low-level control over hardware peripherals (such as hardware timers for PWM and the I80 parallel interface for the display) and to run the application on top of a Real-Time Operating System (**FreeRTOS**).
- **Remote Node (Control Unit - Arduino Mega 2560):** The firmware utilizes the standard **Arduino framework (C++)**, which significantly simplifies the management of hardware serial communication (`Serial1` for the ESP-01S module) and the debouncing logic for the physical input buttons.

2. 3rd-Party Libraries and Sources

The system is designed with minimal reliance on external third-party libraries, implementing critical logic from scratch to maximize efficiency and stability. For the Central Node, native ESP-IDF components were exclusively used:

- `driver/ledc.h`: For generating the hardware PWM signals required by the fan.
- `driver/gpio.h` & `esp_timer.h`: For managing hardware interrupts (ISR) and high-precision microsecond timing.
- `esp_lcd_panel_*.h`: Native drivers for initializing and pushing data to the ST7789 LCD via the I80 parallel bus.
- *Note:* No external libraries (such as Adafruit DHT) were used for the AM2302/DHT22 temperature sensor. The 1-Wire communication protocol was manually implemented (bit-banging).

3. Planned & Implemented Algorithms and Structures

The software architecture relies on the following core algorithms:

- **Digital Decoding Algorithm (Bit-Banging) for DHT22:** Because the ESP32-S3 FreeRTOS scheduler can cause microsecond desynchronization on the 1-Wire bus, a strict timing algorithm (`dht_wait_state`) was implemented. It measures high/low pulse durations in microseconds. Pulses longer than 40µs are interpreted as a logical 1, successfully reconstructing the 40-bit data packet (16-bit humidity, 16-bit temperature, 8-bit checksum).
- **Tachometric (RPM) Calculation Algorithm:** Utilizes an Interrupt Service Routine (ISR) attached to the falling edge of the fan's TACHO pin. A variable counts the pulses generated by the fan's internal Hall-effect sensor. Periodically, the interrupt is paused, and the pulse count is multiplied by a timing constant (e.g., 15, for a 2-second sampling interval) to calculate the exact Revolutions Per Minute (RPM).
- **Thermal Control Structure (Fan Curve):** A step-based linear conditional block maps the read temperature ranges to an 8-bit Duty Cycle interval (0–255). Predefined thresholds dynamically adjust the RPM (e.g., <25°C equals ~20% power; >35°C equals 100% power) to maintain an optimal

cooling-to-noise ratio.
- **Framebuffer Rasterization for Display:** To prevent visual flickering on the LCD, characters are not drawn directly to the screen. Instead, they are drawn into an allocated memory structure (`uint16_t framebuffer[320 * 240]`). A rasterization algorithm parses a custom 8×8 font dictionary, applies a 3x scaling multiplier, maps the 16-bit RGB565 colors, and pushes the entire buffer to the display in a single block.

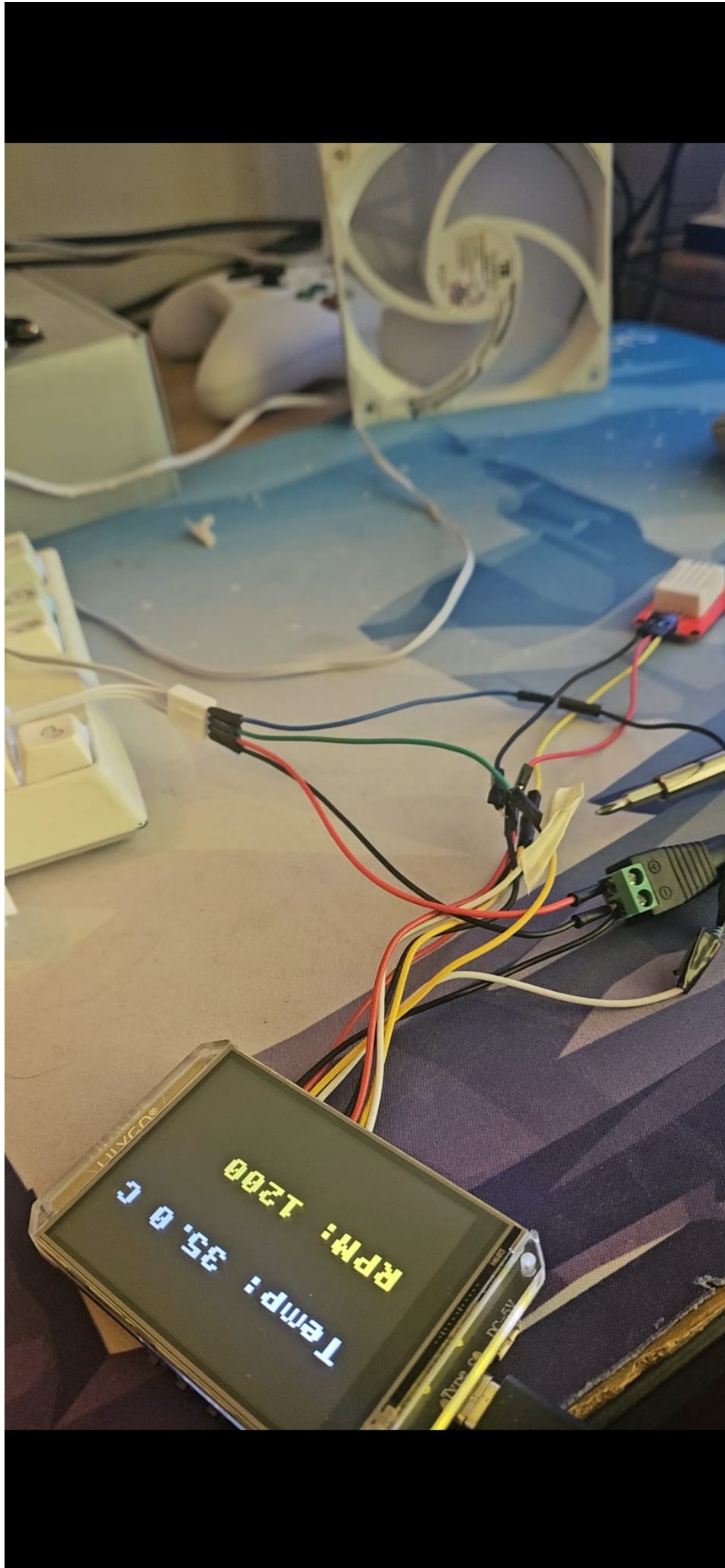
4. Implemented Sources and Functions (Stage 3 - Controller Node)

The following critical C functions were developed and validated for the central unit:

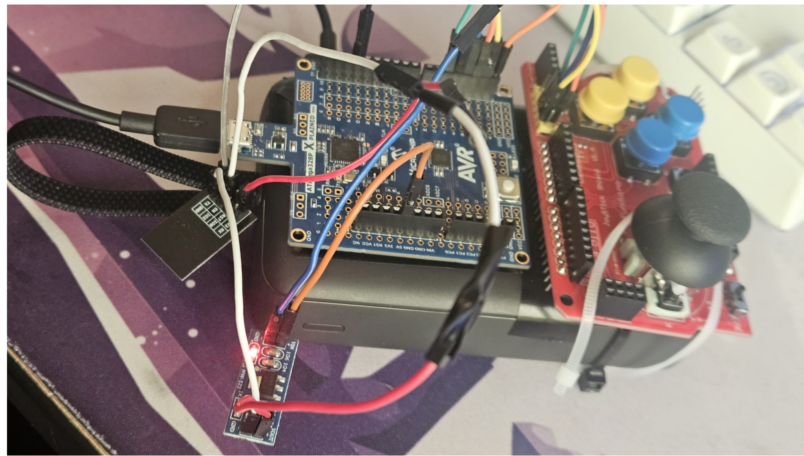
- **`dht_wait_state(int state, int timeout_us)`**: A blocking function equipped with an internal timeout (based on `esp_timer_get_time()`) that measures the duration of a logical state on the GPIO pin, preventing infinite loops in case of sensor disconnection.
- **`read_dht22(float *temp)`**: Initiates the sensor wake-up sequence, temporarily disables global OS interrupts (`taskDISABLE_INTERRUPTS`) to guarantee precise bus timing, decodes the bitstream, and validates data integrity via the checksum.
- **`tacho_isr_handler()`**: The hardware interrupt function. It incorporates a software debouncing mechanism (`now - last_pulse_time > 2500 µs`) to filter out parasitic electrical noise on the tachometer line before incrementing the `pulse_count`.
- **`fan_hardware_init()`**: Configures the LEDC timer for a 25kHz frequency (the industry standard for 4-pin PC fans), sets the 8-bit resolution, and attaches the ISR to the RPM reading pin.
- **`init_display_primitive()`**: Initializes the 8-bit I80 parallel bus and configures the internal ST7789 display controller (including color inversion and XY memory mapping).
- **`draw_char_to_fb()`** / **`draw_string_to_fb()`** / **`clear_framebuffer()`**: A suite of custom graphical functions that take a string, read the corresponding binary pattern from the font matrix (`font8x8`), scale it, and write the pixel data into the framebuffer array.
- **`control_task()`**: The main FreeRTOS task running in an infinite loop (`while(1)`). It orchestrates the overall system logic: reads the DHT22 sensor, calculates the RPM while safely pausing interrupts, applies the thermal curve via `ledc_set_duty`, renders the UI to the framebuffer, flushes it to the screen, and suspends itself for 2 seconds (`vTaskDelay`).

Results / Photos

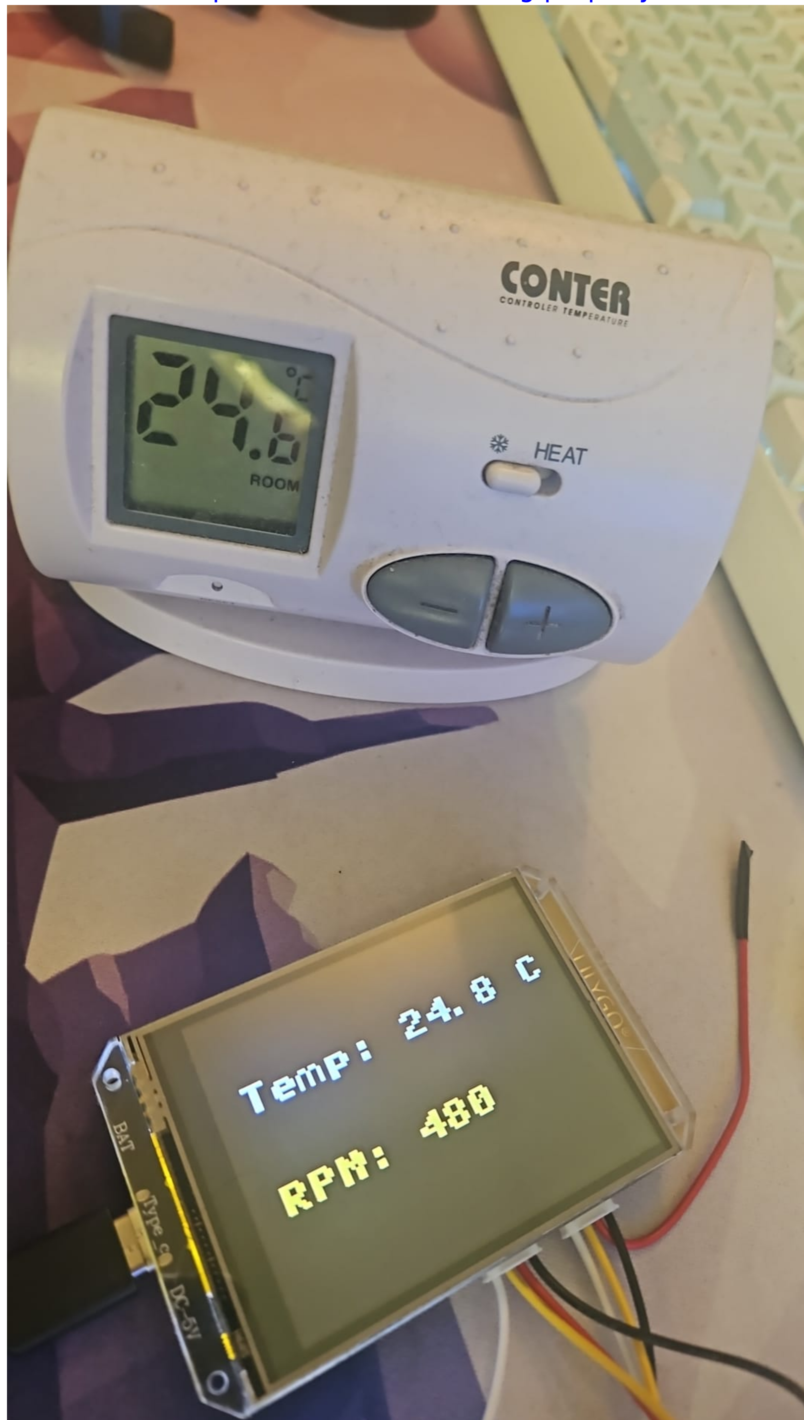
Controller:



Remote Control:



Testing that shows RPM and the temperature sensor working properly:



Concluzii

Download

schema_electrica_pm.fzz_1_.zip videotestareeftimiegabriel.zip

Journal

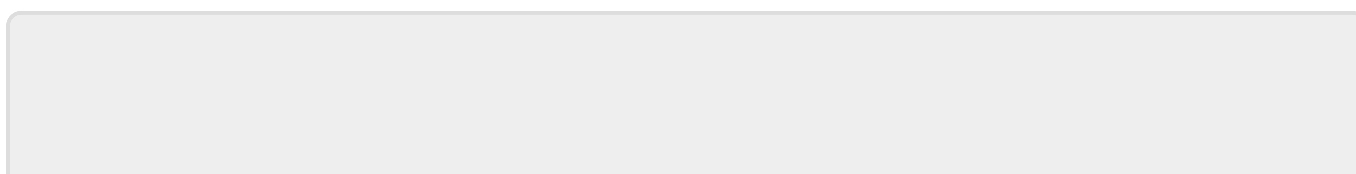
This section serves as the official development log for the project assistant to monitor progress, track milestones, and review troubleshooting steps.

Date	Task / Milestone Description	Status	Notes & Troubleshooting
2026-05-16	Hardware component selection & schematic design in Fritzing. Hardware pin alignment and safety review	**Completed**	Corrected ESP-01S power layout (pin EN pulled HIGH to 3.3V). Confirmed button wiring to GND for INPUT_PULLUP safety.
2026-05-16	Finalized hardware pinout documentation	**Completed**	
[Date]	Firmware development for the Remote Node (ATmega328PB)	*Pending*	Implementing button debouncing and serial UART packet transmission logic.
[Date]	Firmware development for the Controller Node	*Pending*	Writing logic for PWM fan control, DHT22 sensor readings, and ARGB lighting profiles.
[Date]	End-to-end integration and wireless communication testing	*Pending*	Testing packet delivery and response lag between the remote and the main controller.

Bibliografie/Resurse

Listă cu documente, datasheet-uri, resurse Internet folosite, eventual grupate pe **Resurse Software** şi **Resurse Hardware**.

Export to PDF



From:
http://ocw.cs.pub.ro/courses/ - **CS Open CourseWare**

Permanent link:
http://ocw.cs.pub.ro/courses/pm/prj2026/jan.vaduva/gabriel.eftimie

Last update: **2026/05/24 22:09**