Bölüm:day1	Tarih:

Yapılan iş:**IoT Ecosystem Intro: ESP32, Raspberry Pi, Node-RED, MQTT.**

My first day of the internship was an immersive dive into the foundational technologies of our project. The primary goal was to understand the core components and establish a basic communication channel between a sensor node (the ESP32) and a local server (the Raspberry Pi).

Theoretical Learning and System Setup: The first half of the day was dedicated to understanding the architecture of our system. I focused on four key areas:

1. **ESP32 Microcontroller:** I started by learning about the ESP32's fundamental working principles. I understood that it's more than just a microcontroller; it's a powerful System-on-Chip (SoC) with integrated Wi-Fi and Bluetooth capabilities, making it ideal for IoT applications. I also spent time researching the **ESP32's pinout structure** to understand how to connect sensors and actuators in the upcoming phases of the project.

2. **Raspberry Pi as a Local Server:** I learned how the Raspberry Pi would serve as the central hub for our local IoT network. I went through the process of installing the necessary software on it: **Mosquitto**, which acts as our MQTT broker, and **Node-RED**, a visual programming tool for wiring together our data flows.

3. **MQTT Protocol:** I studied the concept of MQTT (Message Queuing Telemetry Transport). I grasped its lightweight, publish-subscribe model, which is highly efficient for devices with limited resources. The key takeaway was understanding "topics" – how a device can publish messages to a specific topic (e.g., `merhaba/giden`) and how other devices can subscribe to that topic to receive the messages.

4. **Node-RED Interface:** I explored the Node-RED web interface running on the Raspberry Pi. I learned how to create simple flows by dragging and dropping nodes, and how it can be used to listen to MQTT topics and visualize the data received from the ESP32.

A crucial practical lesson from the setup phase was the **importance of organized IP address management**. Ensuring the ESP32 and Raspberry Pi were on the same network and that I had the correct, static IP for the Raspberry Pi was essential for the entire system to function.

Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day2	Tarih:

Yapılan iş:**IoT Ecosystem Intro: ESP32, Raspberry Pi, Node-RED, MQTT.**

Practical Implementation & First Code

With the theoretical foundation in place, I moved on to the practical part: writing and deploying my first piece of code to the ESP32. The goal was to make the ESP32 connect to our local Wi-Fi, then connect to the MQTT broker on the Raspberry Pi, and send a simple "Hello" message.

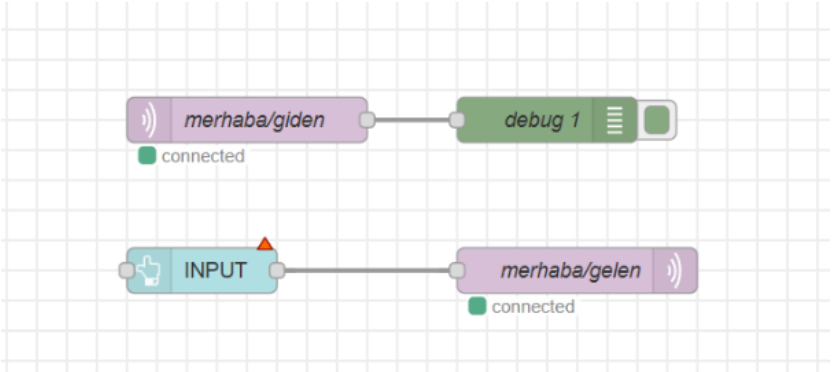
The code I implemented performed the following steps:

1. **Wi-Fi Connection:** The `setup_wifi()` function handled the connection to the local network. I used the serial monitor in the Arduino IDE to track the connection status, which was helpful for debugging.

2. **MQTT Server Configuration:** I configured the client by setting the Raspberry Pi's IP address and the MQTT port (`client.setServer(mqtt_server, mqtt_port)`).

3. **Establishing MQTT Connection:** The `mqtt_connect()` function was designed to connect the ESP32 to the Mosquitto broker with a unique client ID. It also included a crucial feature: if the connection failed, it would print the client state error code (`client.state()`) and retry after 5 seconds. This was my first encounter with writing resilient, self-recovering code.

4. **Publishing and Subscribing:** In the main `loop()`, the device periodically published the message "Merhaba Arduino'dan" to the `merhaba/giden` topic. Simultaneously, it was subscribed to the `merhaba/gelen` topic, with the `callback()` function ready to process any incoming messages.



Challenges and Solutions:

I encountered a minor issue where the ESP32 initially failed to connect to the MQTT broker. By observing the error code printed to the serial monitor by `client.state()`, I realized I had mistyped the Raspberry Pi's IP address in my code. Correcting this simple error immediately solved the problem and reinforced the importance of careful configuration and using debug messages.

Summary of Learnings & Equipment Used:

- **Concepts Learned:** ESP32 architecture and pinouts, the role of an MQTT broker, the publish/subscribe pattern, and data flow creation in Node-RED.
- **Practical Skills:** Installing and configuring Mosquitto and Node-RED on a Raspberry Pi, writing C++ code for the ESP32 using the `PubSubClient` library, and debugging network connectivity issues using the serial monitor.
- **Hardware/Software Used:** ESP32 Development Board, Raspberry Pi 4, Arduino IDE, Mosquitto MQTT Broker, Node-RED.

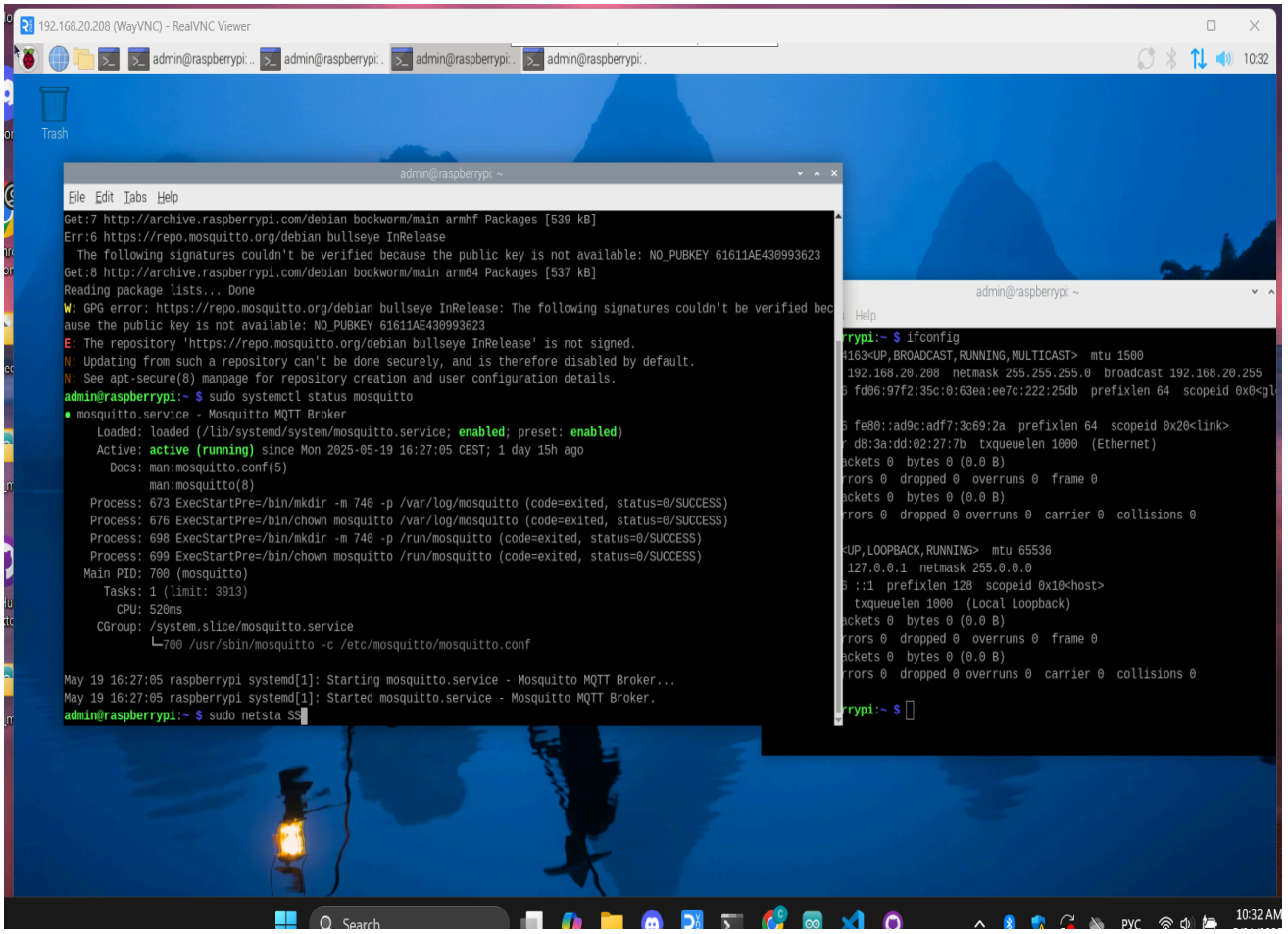
By the end of the day, I had successfully sent a message from the ESP32 and could see it appear in my Node-RED debug window, marking a successful first step in the project.

The first step was to upgrade the ESP32's capabilities. The initial code was only for publishing messages. The new code I developed introduces a more robust, two-way communication logic.

Key improvements in the ESP32 Code:

- **Dual MQTT Topics:** I defined two separate topics for communication:
  - `esp32/veri/giden`: Used by the ESP32 to publish data to Node-RED.
  - `nodered/komut/gelen`: Used by the ESP32 to subscribe to commands sent from Node-RED.
- **Dynamic MQTT Client ID:** Instead of a static ID, I implemented a randomized client ID (`String clientId = "ESP32-ManualClient-" + String(random(0xffff), HEX);`). This is a best practice to avoid connection conflicts with the MQTT broker, especially if multiple devices are on the network or if a device reconnects frequently.
- **Interactive Serial Monitor Publishing:** The main `loop()` was modified to check for input from the Arduino IDE's Serial Monitor (`if (Serial.available() > 0)`). This allowed me to manually type a message into the serial monitor, press Enter, and have the ESP32 publish that message to the `esp32/veri/giden` topic. This was a fantastic way to test the publishing mechanism without needing a physical sensor yet.
- **Robust Callback Function:** The `mqtt_callback` function was enhanced to not only print any received message but also specifically check if the message came from the `nodered/komut/gelen` topic, confirming that a command was successfully received from the UI.

During this phase, I also confirmed that the Mosquitto MQTT broker was running correctly on the Raspberry Pi using the `sudo systemctl status mosquitto` command, ensuring the service was active and enabled.



Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

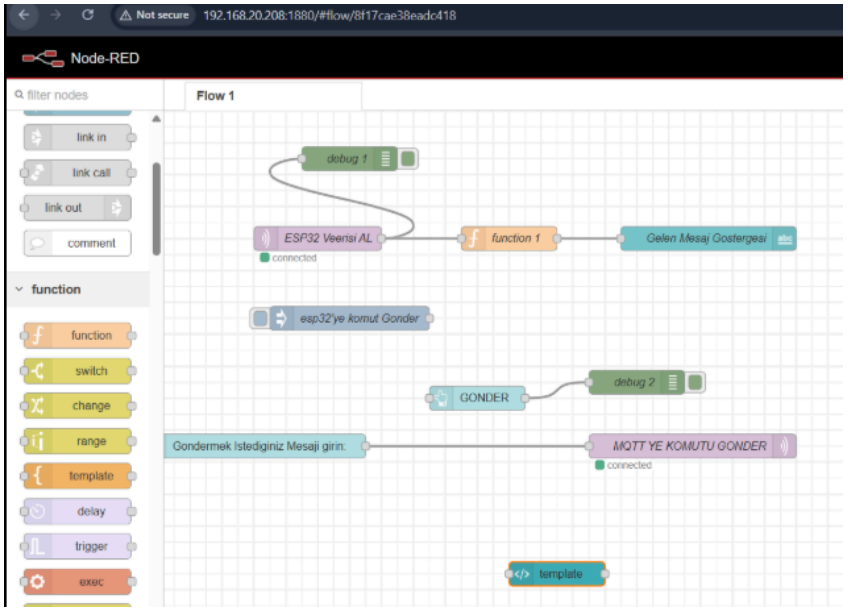
Bölüm:day3	Tarih:

Yapılan iş:**Building the Node-RED Flow and Dashboard UI**

With the ESP32 ready for two-way traffic, the next step was to build a system in Node-RED to manage this. I created a complete flow to handle incoming data and to send commands back to the ESP32.

Node-RED Flow Components:

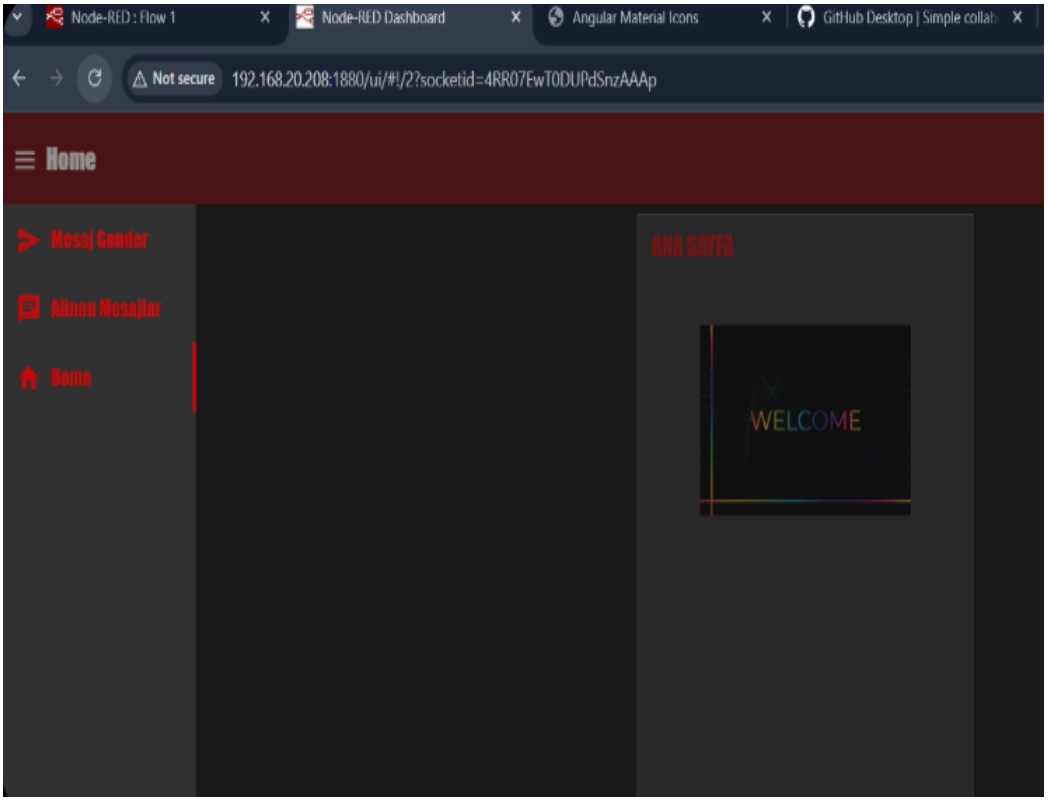
1. **Listening to ESP32:** An `mqtt in` node was configured to subscribe to the `esp32/veri/giden` topic. This node captures any data sent from the ESP32.
2. **Displaying Received Data:** The data from the `mqtt in` node was passed through a `function` node to format it and then displayed in two ways: a `debug` node for real-time logging and a `ui_text` node on the dashboard titled "ESP32'den Gelen Mesajlar".
3. **Sending Commands:** I created a UI group for sending messages. It consists of a `ui_text_input` node for typing a message, a `ui_button` to trigger the sending, and an `mqtt out` node configured to publish the message to the `nodered/komut/gelen` topic.

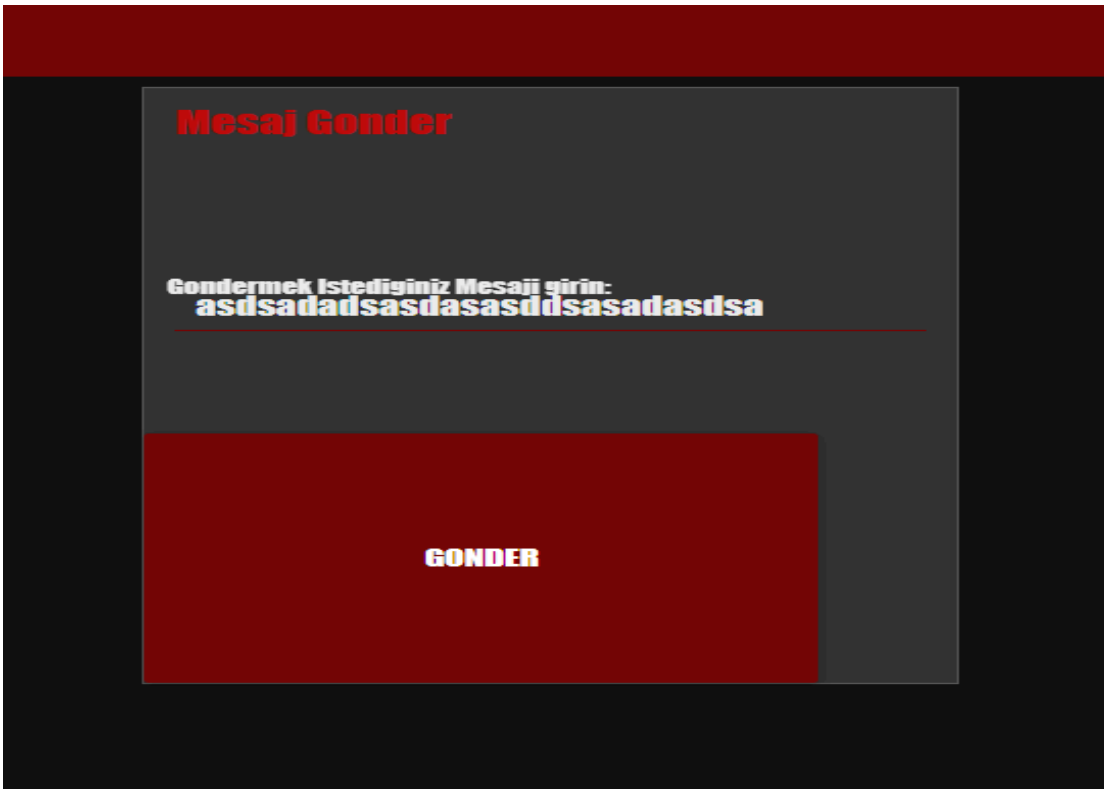


The Node-RED Dashboard UI

The final step was to interact with the system through the user-friendly dashboard created by Node-RED. The dashboard I designed has three main sections: "Mesaj Gönder" (Send Message), "Alınan Mesajlar" (Received Messages), and a "Home" page.

- The **"Mesaj Gönder"** page allows a user to type any string and send it directly to the ESP32.
- The **"Alınan Mesajlar"** page displays a list of all messages that the ESP32 has sent and that Node-RED has received.
- The main page provides a simple welcome screen.


Node-RED : Flow 1
Node-RED Dashboard
Angular Material Icons
GitHub Desktop | Simple collab
Not secure
192.168.20.208:1880/ui/#!/2?socketid=4RR07EwT0DUPdSnzAAAp
Home
Mesaj Gonder
Alinan Mesajlar
Home
ANA SAYFA
WELCOME


Mesaj Gonder
Gondermek Istediginiz Mesaji girin:
asdsadadsasdasasddsasadasdsa
GONDER

Challenges and Conclusion

The primary challenge was ensuring the data formats were consistent. The ESP32's `PubSubClient` library works best with character arrays (`char*`), while I was reading `String` objects from the serial port. This required a careful conversion using `messageFromSerial.toCharArray()` before publishing to MQTT, which taught me a valuable lesson in data type handling in C++.

By the end of Day 3, I had a fully functional, bidirectional communication system, laying a robust foundation for the next phase of the project, which will involve attaching and reading data from actual sensors.

Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day4	Tarih:

Yapılan iş:**System Design, ESP32 Programming, and Hardware Setup**

Project Goal & Initial Challenge:

My primary goal was to control each of the 15 LEDs on the strip independently, allowing for unique colors to be assigned to each one. However, upon inspecting the hardware, I encountered a significant limitation: the LED strip was not a standard WS2812B where every LED is individually addressable. Instead, it was designed with one driver IC for every three LEDs. This meant I had to adapt my project goal. While I couldn't control every single LED, I could control **each group of 3 LEDs** independently, which still provided a strong proof-of-concept for addressable control.

ESP32 Software:

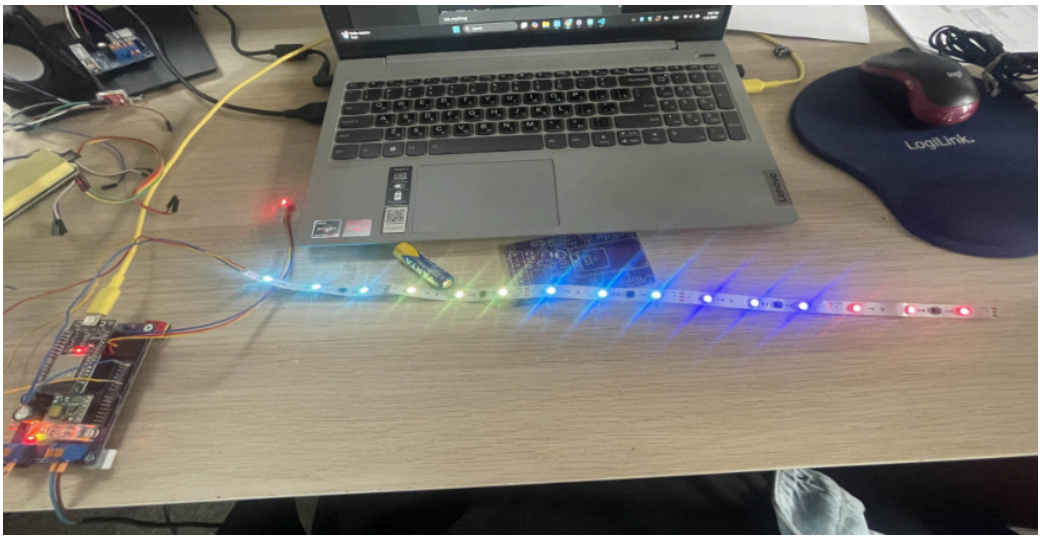
The ESP32 code was more complex than the previous projects. It required several libraries working in tandem:

- **Adafruit_NeoPixel:** To control the ARGB LEDs.
- **PubSubClient:** For MQTT communication.
- **ArduinoJson:** A crucial library for parsing the incoming JSON-formatted data from Node-RED.

The code was designed to listen on a specific MQTT topic (`esp32/setSingleLed`). When a message arrived, it would parse the JSON payload to extract two key pieces of information: the target LED `index` and the desired `color` in hexadecimal format. This data was then used to call the `strip.setPixelColor()` function and update the physical LED strip with `strip.show()`.

Hardware Assembly:

The hardware setup involved connecting the ESP32 to the ARGB LED strip. The strip was powered by an external 5V power supply to ensure stable operation and prevent overloading the ESP32. The ESP32's GPIO33 pin was connected to the Data-In (DIN) pin of the strip to send the color commands.



Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day5	Tarih:

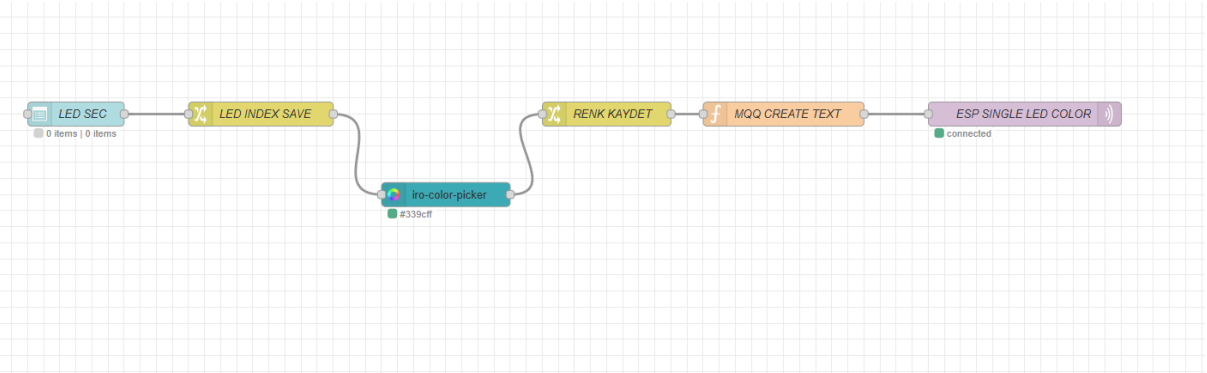
Yapılan iş:**Node-RED Flow and Dashboard UI Implementation**

With the ESP32 and hardware ready, the focus shifted to creating the user interface in Node-RED.

Node-RED Flow:

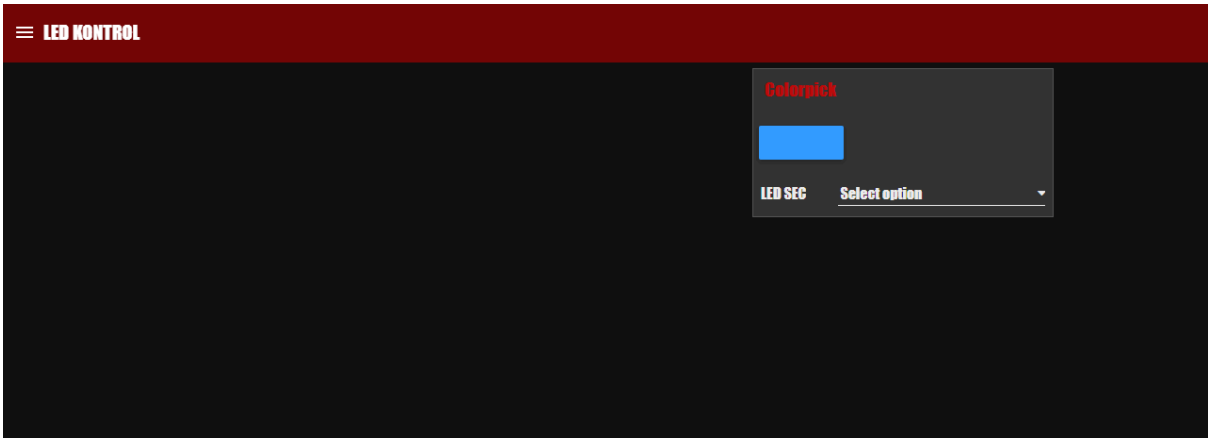
The flow was designed to be user-friendly and intuitive. It consisted of several key nodes:

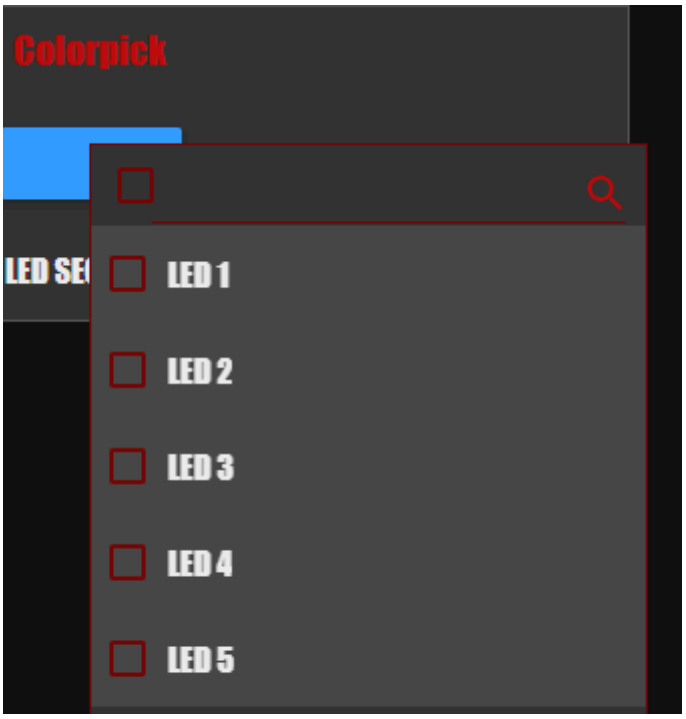
1. **ui_dropdown (LED SEC):** To create a dropdown menu allowing the user to select which group of LEDs to control (from 1 to 15). This node outputs the selected index (0-14).
2. **node-red-contrib-ui-iro-color-picker:** An external node I installed that provides a professional, wheel-based color picker for a great user experience. It outputs an object containing the r, g, b values of the selected color.
3. **function node (MQTT CREATE TEXT):** This was the core of the UI logic. It takes the selected index from the dropdown and the color object from the color picker, then formats them into a single JSON string. This JSON payload matches the exact format the ESP32 code expects: `{"index": 5, "color": "#ff0000"}`.
4. **mqtt out node (ESP SINGLE LED COLOR):** This node publishes the final JSON payload to the `esp32/setSingleLed` topic, sending the command to the ESP32.

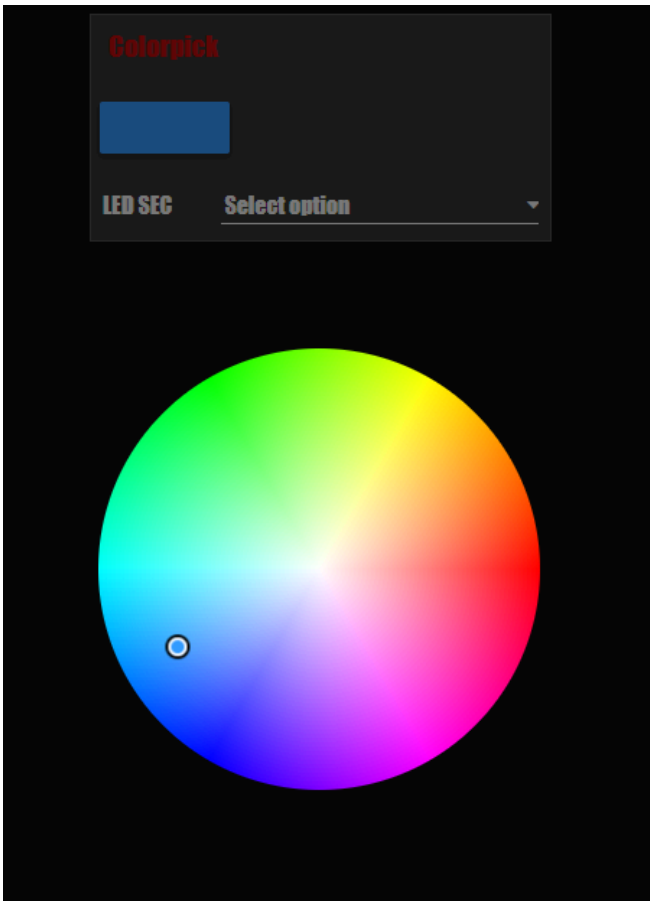


The Final User Interface:

The resulting Node-RED dashboard provides a simple yet powerful interface for controlling the LEDs. A user can select a LED group from the dropdown menu, pick a vibrant color from the color wheel, and instantly see the physical LED strip update. This provided a tangible and satisfying result, directly linking a web interface to a hardware action.


LED KONTROL
Colorpick
LED SEC
Select option


Colorpick
LED SE
LED 1
LED 2
LED 3
LED 4
LED 5



Conclusion:

This project was a major step forward, integrating hardware control with a custom-built web UI. The initial challenge with the LED strip hardware provided a valuable real-world lesson in adapting project scope based on available components. Successfully parsing JSON on an ESP32 and creating a dynamic Node-RED flow were key technical achievements that will be invaluable for future, more complex IoT applications.

Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day6	Tarih:

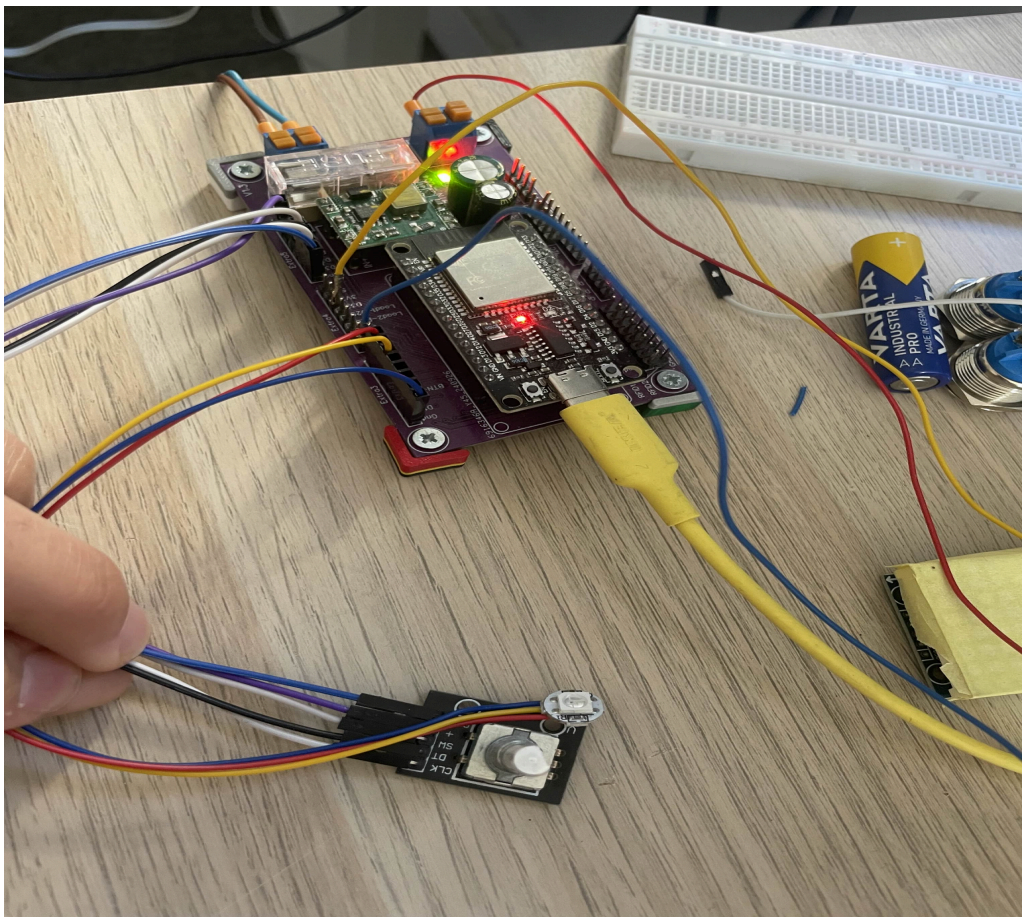
Yapılan iş:**Integrating Physical Controls with a Rotary Encoder and Status Feedback**

Following the successful implementation of the web-based UI, the project's next phase was to add a physical control mechanism. This allows for quick, tactile adjustments without needing to use the web interface. I chose a rotary encoder module to manage the global brightness of the LED strip and to toggle it on and off. This creates a hybrid control system, combining the precision of the web UI for color selection with the convenience of a physical knob for brightness.

Hardware Integration & Challenges:

The main task was to connect the rotary encoder to the ESP32. The encoder has three main signal pins: Clock (CLK), Data (DT), and Switch (SW). I connected these to the respective GPIO pins on the ESP32 (GPIO 32, 35, and 34).

A significant challenge arose from the choice of pins. GPIOs 34 and 35 on the ESP32 are input-only and do not have internal pull-up resistors. For the encoder's switch (SW) pin to function reliably, an external pull-up resistor (connecting the pin to 3.3V) is required to prevent the pin from "floating" and causing false readings. This was a critical lesson in ESP32 hardware specifics and the importance of reading pinout documentation.



ESP32 Code Enhancements:

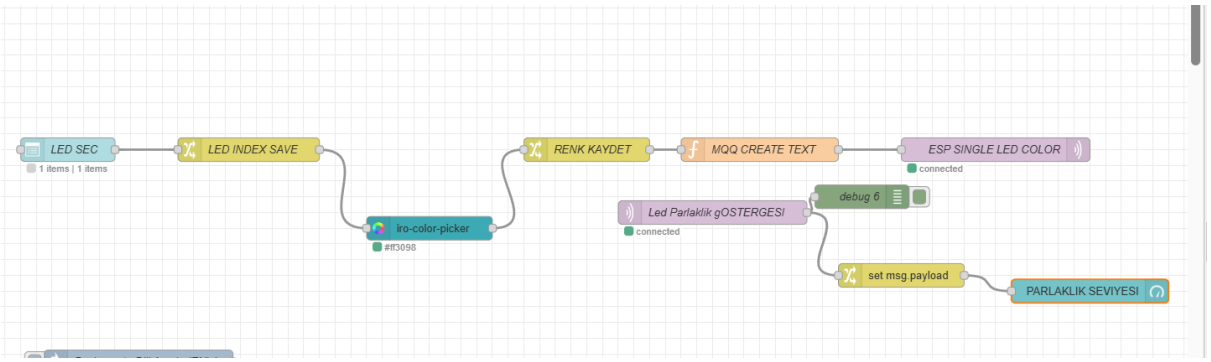
The ESP32 code required substantial additions to handle the new hardware and logic:

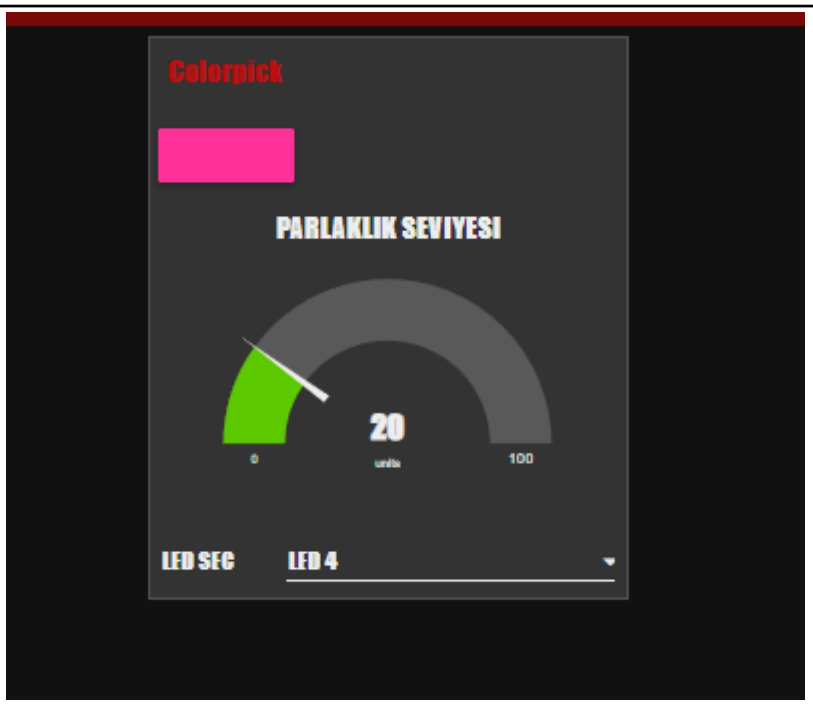
1. **ESP32Encoder Library:** I integrated the `ESP32Encoder` library to efficiently read the rotation of the encoder. This library simplifies the process of tracking the encoder's position.
2. **Global Brightness Control:** The encoder's rotation is now directly mapped to a `currentBrightness` variable (from 0 to 100). Turning the knob updates this variable.
3. **On/Off Toggle:** The encoder's built-in push-button is used as an on/off switch. The code includes a "debounce" mechanism to ensure that a single press is registered cleanly, without electrical noise causing multiple triggers. When toggled off, the code stores the last brightness level, so when it's turned back on, it returns to its previous state.
4. **Color and Brightness Separation:** A key part of the new logic is how colors and brightness are managed. The individual colors set from Node-RED are stored in a `baseLedColors` array. The `refreshLedStrip()` function now takes these base colors and applies the `currentBrightness` value to them before sending the final color data to the LED strip. This allows global brightness changes without losing the individually set colors.
5. **MQTT Status Feedback:** To ensure the web UI and physical controls are in sync, I implemented a feedback loop. Whenever the brightness is changed using the encoder, the ESP32 publishes the new brightness level to a new MQTT topic: `esp32/brightnessFeedback`.

Node-RED Flow and UI Updates:

The Node-RED flow was updated to accommodate the new feedback feature:

1. **MQTT In Node:** A new `mqtt in` node was added to subscribe to the `esp32/brightnessFeedback` topic.
2. **Dashboard Gauge:** The data from this topic is channeled into a `ui_gauge` node on the dashboard. This gauge provides a real-time visual representation of the current brightness level of the LED strip, whether it was changed physically by the encoder or not.





Conclusion:

This stage successfully merged web-based control with physical, tactile control. The system is now significantly more versatile and user-friendly. The implementation of a feedback mechanism where the physical device reports its state back to the UI is a fundamental concept in robust IoT design. The project has evolved from a simple one-way command system to a fully interactive and synchronized smart device.

Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day7	Tarih:

Yapılan iş:**Advanced Visual Feedback with MAX7219 LED Matrix Display**

To complete the project, I decided to add another layer of visual feedback. While the Node-RED dashboard provides an excellent remote interface, a local, at-a-glance display on the device itself makes the system's status much more accessible. For this, I integrated a 4-module MAX7219 8x8 LED matrix display.

Objective:

The goal was to have the matrix display show real-time, critical information about the ARGB LED strip's status. Specifically, I wanted it to cycle through the current global brightness and the individual color codes of the first five LED groups.

Hardware and Library Integration:

This addition required new hardware and corresponding software libraries:

- **Hardware:** Four 8x8 LED matrix modules driven by the MAX7219 chipset were daisy-chained together and connected to the ESP32 using the SPI communication protocol.
- **Libraries:** I incorporated three new libraries into the ESP32 code:
  - `MD_Parola` & `MD_MAX72xx`: Powerful libraries for controlling and creating text effects on MAX7219-based displays.
  - `SPI`: The standard library for handling SPI communication between the ESP32 and the matrix display.

ESP32 Code Logic:

The core logic of the ESP32 code was significantly expanded to manage the new display alongside its existing tasks.

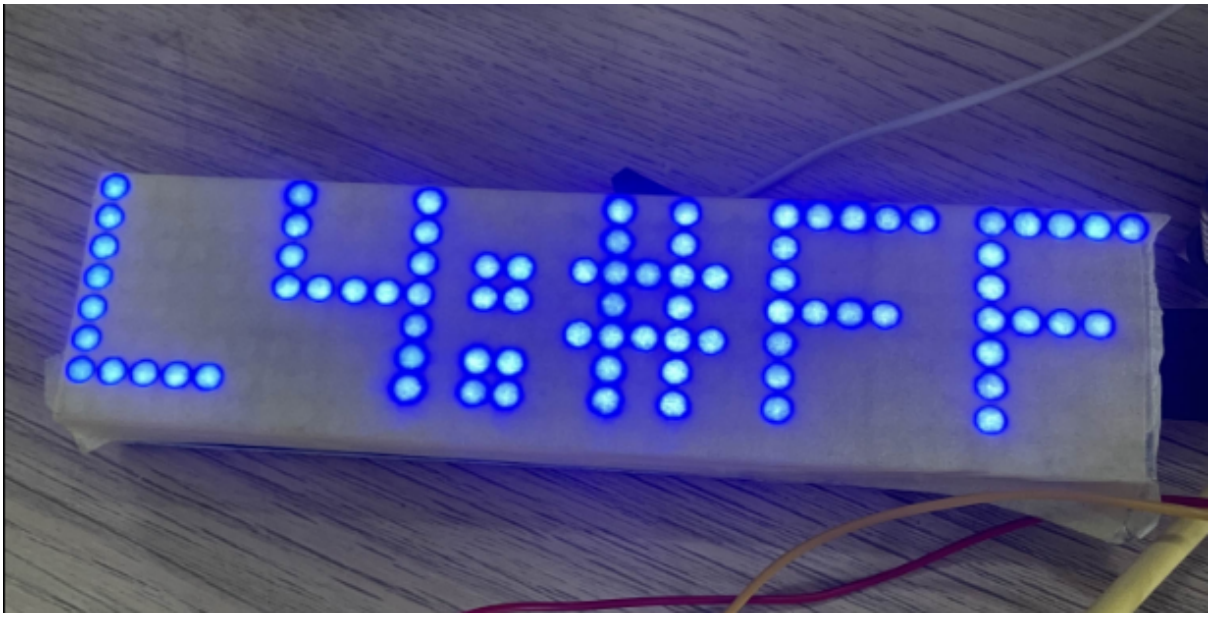
1. **Display State Machine:** I implemented a simple state machine using an `enum` to manage what is currently being shown on the matrix. The states cycle between displaying brightness, LED 0's color, LED 1's color, and so on.
2. **Cyclical Updates:** The display automatically cycles through this information every 3 seconds.
3. **Forced Updates:** To ensure the display is always showing the most current information, I implemented a `forceMatrixUpdate` flag. This flag is set to `true` whenever a change occurs—either a color change from Node-RED or a brightness change from the rotary encoder. This immediately triggers a refresh of the matrix display with the new data, rather than waiting for the next 3-second cycle.
4. **Data Conversion:** A helper function, `colorToHex()`, was created to convert the `uint32_t` color values stored in the `baseLedColors` array back into a readable HEX string format (e.g., `#FF0000`) for display on the matrix.

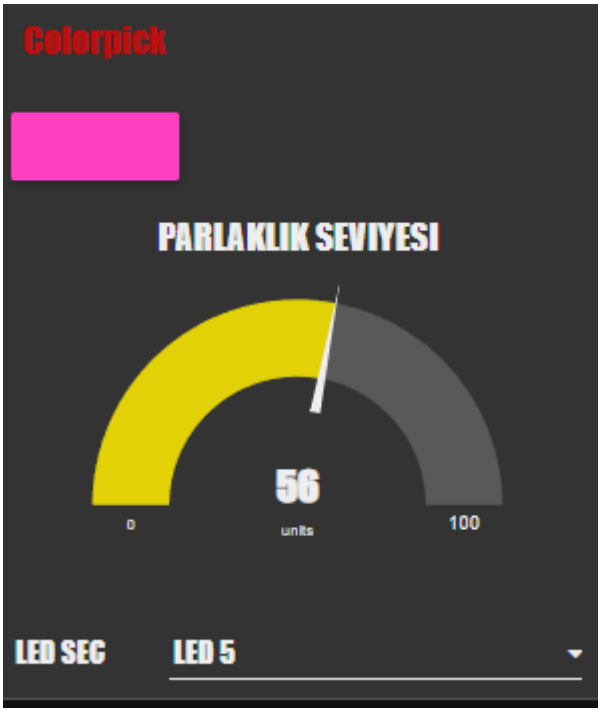
Final Functionality:

The fully integrated system now provides a comprehensive feedback loop:

- The **Node-RED UI** allows for remote color selection and shows the brightness level.
- The **rotary encoder** allows for physical control of brightness and toggling the power.
- The **ARGB LED strip** instantly reflects these changes.
- The **MAX7219 matrix display** provides a local, real-time text-based status update, showing the brightness percentage and HEX color codes.

For example, when the brightness is adjusted to 56%, the matrix displays "B: 56%". When a color is changed, it will show messages like "L4:#FFFF00" in its cycle.





Conclusion:

This final addition of the matrix display successfully closes the information loop of the project. It transforms the device from a simple remote-controlled object into a self-contained, smart device that clearly communicates its own status. This project spanned from basic MQTT communication to a multi-faceted system involving a web UI, physical controls, and multiple visual feedback mechanisms, providing an extensive and valuable learning experience in IoT system design.

Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day8	Tarih:

Yapılan iş:**Time Synchronization with Network Time Protocol (NTP)**

After building several projects that involved sending and receiving data, I recognized the need for accurate timekeeping. Many IoT applications require precise timestamps for logging data, scheduling actions, or ensuring accurate records. To address this, today's project focused on connecting the ESP32 to Network Time Protocol (NTP) servers to synchronize its internal clock with the Coordinated Universal Time (UTC).

Objective:

The goal was to program the ESP32 to connect to a Wi-Fi network, fetch the current time from public NTP servers, and correctly display it as local time on the Serial Monitor, accounting for my specific time zone and Daylight Saving Time (DST).

Hardware and Software:

- **Hardware:** This project was purely software-focused and only required a standard ESP32 development board.
- **Software:** I utilized the built-in libraries of the ESP32 core for Arduino IDE, specifically `WiFi.h` and the ESP's SNTP functionalities.

Implementation and Code Logic:

The process was implemented with the following key steps in the ESP32 code:

Wi-Fi Connection: As in previous projects, the first step is to connect the ESP32 to the local Wi-Fi network.

NTP Server Configuration: I configured the ESP32 to use two reliable public NTP servers (`pool.ntp.org` and `time.nist.gov`) to ensure redundancy.

Time Zone and DST: To convert the received UTC time to my local time, I configured the time zone settings. This was done by defining the GMT offset and the daylight saving offset in seconds. I used a POSIX time zone string (`"CET-1CEST,M3.5.0,M10.5.0/3"`) which is a powerful method to automatically handle DST changes for the Central European Time zone.

Time Synchronization and Callback: I used the `configTime()` function to initiate the time synchronization process. A key feature I implemented was the use of `sntp_set_time_sync_notification_cb(timeavailable)`. This registers a callback function, `timeavailable()`, which is automatically triggered only once when the ESP32 successfully receives the time from the NTP server for the first time. This provides a clear confirmation that the synchronization was successful.

Displaying Time: In the main loop, the `printLocalTime()` function is called every five seconds. This function retrieves the time using `getLocalTime()` and prints it to the Serial Monitor in a human-readable format, such as "Friday, June 20 2025 08:05:00".

```
#include "time.h"

#include <WiFi.h>
const char* ntpServer = "pool.ntp.org";
const char* time_zone = "CET-1CEST,M3.5.0,M10.5.0/3";
void timeavailable(struct timeval *t) {
  Serial.println("NTP'den zaman bilgisi başarıyla senkronize edildi!");
}
void setup() {
  Serial.begin(115200);
  sntp_set_time_sync_notification_cb(timeavailable);

  configTzTime(time_zone, ntpServer);
}
```

Conclusion:

This project successfully established a foundational capability for all future work. By ensuring the ESP32 has accurate, synchronized time, any data logged or actions performed can now be reliably timestamped. This is a critical step in moving from simple control systems to more advanced data logging and automation applications where timing is essential. The use of a callback function for the initial sync confirmation also proved to be an elegant and efficient programming practice.

Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day9	Tarih:

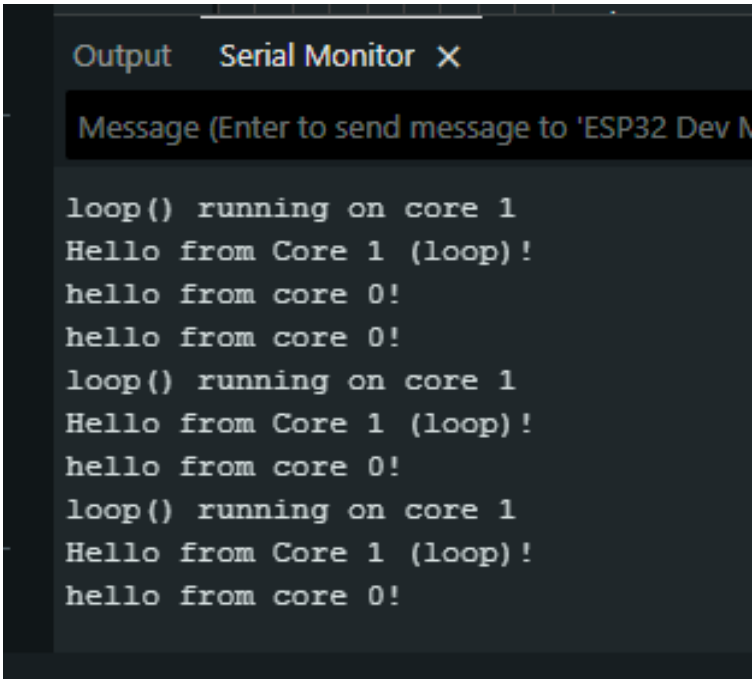
Yapılan iş:**Introduction to FreeRTOS and Multitasking on ESP32**

Having built several single-threaded applications, I dedicated today to understanding one of the ESP32's most powerful features: its dual-core architecture and the underlying FreeRTOS (Real-Time Operating System). The goal was to move beyond the simple `setup()` and `loop()` structure and learn how to run multiple, independent tasks concurrently.

Project 1: Basic Dual-Core Operation

My first experiment was to verify that I could assign different tasks to each of the ESP32's two cores.

- **Objective:** Create a simple task and pin it to Core 0, while letting the default `setup()` and `loop()` functions run on Core 1.
- **İmplementation:**
  - I wrote a function named `Task1code` that prints a message to the Serial Monitor every second.
  - In the `setup()` function, I used `xTaskCreatePinnedToCore()`. This function is the cornerstone of FreeRTOS multitasking on the ESP32. I configured it to create a new task that runs the `Task1code` function and explicitly "pinned" it to Core 0.
  - The standard `loop()` function continued to run on its default core, which is Core 1.
- **Outcome:** The Serial Monitor clearly showed interleaved messages: "hello from core 0!" and "Hello from Core 1 (loop)!". This successfully demonstrated true parallel execution on the ESP32.



```
void IRAM_ATTR button_ISR() {
  xSemaphoreGiveFromISR(buttonSemaphore, NULL);
}

//(Core 1).
void buttonTask(void *pvParameters) {
  for (;;) {
    if (xSemaphoreTake(buttonSemaphore, portMAX_DELAY) == pdTRUE) {
      isLedPaused = !isLedPaused;
      if (isLedPaused) {
        vTaskSuspend(ledTaskHandle); // LED görevini askıya al
      } else {
        vTaskResume(ledTaskHandle);
      }
    }
  }
}
```

The `button_ISR` function runs immediately on a hardware button press. To maintain system stability, an ISR cannot perform complex operations. Instead, it safely signals another task by "giving" a semaphore with `xSemaphoreGiveFromISR`. The `buttonTask` waits efficiently for this signal using `xSemaphoreTake`. Once the signal is received, the task awakens and uses the task handle (`ledTaskHandle`) to either suspend or resume the separate `ledTask`. This pattern is fundamental for creating responsive and stable multi-threaded embedded systems.

Project 2: Task Suspension and Synchronization with Semaphores

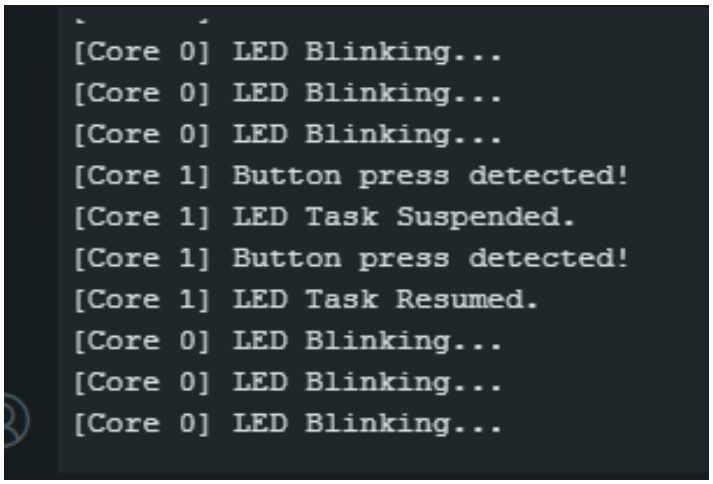
Building on the first experiment, I explored more advanced FreeRTOS concepts: inter-task communication and control.

- **Objective:** Create two tasks—one to blink an LED and another to listen for a button press. The button press should pause or resume the LED blinking task. This requires a way for the two tasks to signal each other.

- **İmplementation:**
  1. **Tasks:** I created `ledTask` (pinned to Core 0) to blink an LED every 500ms and a `buttonTask` (pinned to Core 1) to wait for a signal.
  2. **Interrupt Service Routine (ISR):** I attached a hardware interrupt to the button pin. When the button is pressed, a very fast function called an Interrupt Service Routine (ISR) is executed.
  3. **Semaphores:** It is unsafe for an ISR to directly manipulate tasks. Instead, I used a "binary semaphore" (`xSemaphoreCreateBinary()`). The ISR's only job is to "give" the semaphore, which is like raising a flag.

4. **Task Synchronization:** The `buttonTask` waits indefinitely until the semaphore is given (`xSemaphoreTake()`). Once it receives the signal, it proceeds to suspend or resume the `ledTask` using its unique handle (`ledTaskHandle`). The `vTaskSuspend()` and `vTaskResume()` functions were used to control the execution of the LED task.

- **Outcome:** The system worked perfectly. The LED blinked continuously until the button was pressed, at which point it stopped. Pressing the button again resumed the blinking. The serial monitor provided a clear log of tasks being suspended and resumed on different cores.



Conclusion:

Today was a crucial step in understanding embedded systems programming. Learning to use FreeRTOS, create tasks pinned to specific cores, and use semaphores for safe inter-task communication are fundamental skills for developing complex and responsive IoT devices. This knowledge allows for the separation of concerns—for example, handling network communication on one core while managing real-time sensor readings on the other—preventing tasks from blocking one another and leading to much more efficient and reliable applications.

Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day10 **Project - ESP32 Advanced Control Hub**	Tarih:

Yapılan iş:**System Architecture and Backend Development**

I undertook a large-scale project to consolidate many of the concepts and skills I've learned so far. The goal was to create a versatile "Advanced Control Hub" using the ESP32. This hub integrates an ARGB LED strip, a DFPlayer Mini MP3 module, and a rotary encoder, all manageable through a sophisticated, self-hosted web interface.

The first day was focused on designing the system, selecting the necessary libraries, and writing the core backend logic on the ESP32.

Key Features and Components:

- **Web Server:** Instead of MQTT, this project uses the ESP32 as a standalone web server. This allows any device on the same network to control the hub via a web browser without needing an external broker.
- **DFPlayer Mini MP3 Module:** For audio playback capabilities.
- **ARGB LED Strip:** For visual feedback and effects.
- **Rotary Encoder:** For physical, tactile control over the system's functions.
- **Web Interface:** A feature-rich user interface written in HTML, CSS, and JavaScript, served directly from the ESP32.

ESP32 Code - The "Backend":

The code is the most complex I have written so far, combining multiple functionalities:

1. **Library Integration:** It required integrating a suite of libraries: `WebServer`, `ArduinoJson` (for handling data between the ESP32 and the web UI), `Adafruit_NeoPixel`, `DFRobotDFPlayerMini`, and `AiEsp32RotaryEncoder`.
2. **Web Server Endpoints:** I created several "endpoints" (URLs) that the web interface can call to trigger actions. For example:
   - `/` serves the main HTML page.
   - `/status` provides a JSON object with the real-time status of all components (LED brightness, DFPlayer volume, button states, etc.).
   - `/set_brightness?value=...` sets the LED brightness.
   - `/play_track?track=...` plays a specific MP3 file.
3. **Modal Rotary Encoder:** The rotary encoder's functionality is modal. Pressing its button cycles through what it controls: LED Brightness, LED Color Selection (from a predefined list), and DFPlayer Volume. This makes a single physical control incredibly versatile.
4. **Hardware Abstraction:** The code is structured to separate concerns. Functions like `updateLeds()`, `handleRotaryEncoder()`, and various web handlers (`handleSetBrightness`, etc.) keep the code organized and readable.

```
// loop()
if (IrReceiver.decode()) {
    processIRCode(IrReceiver.decodedIRData.decodedRawData);
    IrReceiver.resume();
}
```

```
// Web sunucusu IR endpoint
void handleIrActionFromWeb() {
    if (server.hasArg("cmd")) {
        String cmdName = server.arg("cmd");
        uint32_t hexCode = getHexForIrAction(cmdName);
        if (hexCode != 0) {
            processIRCode(hexCode);
        }
    }
    server.send(200, "text/plain", "OK");
}
```

```
// MQTT callback func
} else if (topicStr == "esp32/command/ir_action") {
    String cmdName = String(msg);
    uint32_t hexCode = getHexForIrAction(cmdName);
    if (hexCode != 0) {
        processIRCode(hexCode);
    }
}
```

This shows the project's core design principle: abstraction. Whether the input comes from the physical IR remote (IrReceiver.decode), a button on the web interface (handleIrActionFromWeb), or an MQTT command (mqttCallback), all paths lead to the same processIRCode function. This centralizes the action logic, making the system easier to maintain and ensuring consistent behavior across all control methods.

Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day11	Tarih:

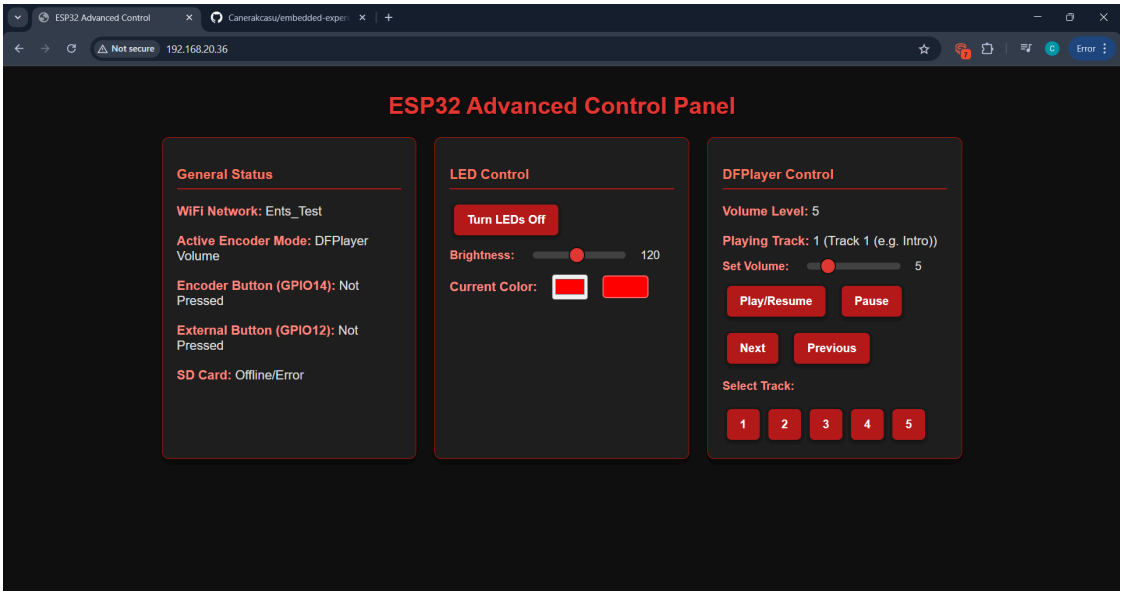
Yapılan iş:**Frontend Development and System Integration**

The second day was dedicated to creating the web interface and ensuring seamless integration between the hardware, backend, and frontend.

The Web Interface (UI):

The UI is a single HTML file stored in the ESP32's program memory using the `PROGMEM` directive.

- **Structure & Styling:** I used modern HTML5 and CSS to create a responsive, dark-themed dashboard with cards for different control sections (General Status, LED Control, DFPlayer Control).
- **Dynamic Content:** JavaScript is used to make the UI dynamic:
  - A `updateStatus()` function periodically fetches the JSON data from the `/status` endpoint and updates all the values on the page in real-time.
  - Buttons and sliders are programmed to call the corresponding API endpoints on the ESP32 when used (e.g., the brightness slider calls `/set_brightness`).
  - This asynchronous communication (using `fetch()`) ensures the UI is responsive and doesn't need to reload the entire page for every action.



Conclusion:

This project serves as a powerful demonstration of building a complete IoT product from the ground up, combining hardware control, network communication, and a user-facing interface. It successfully integrates multiple complex components into a cohesive system, controlled both physically and through a custom web application.

Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day12	Tarih:

Yapılan iş:**Re-architecting the Control Hub with Node-RED and MQTT**

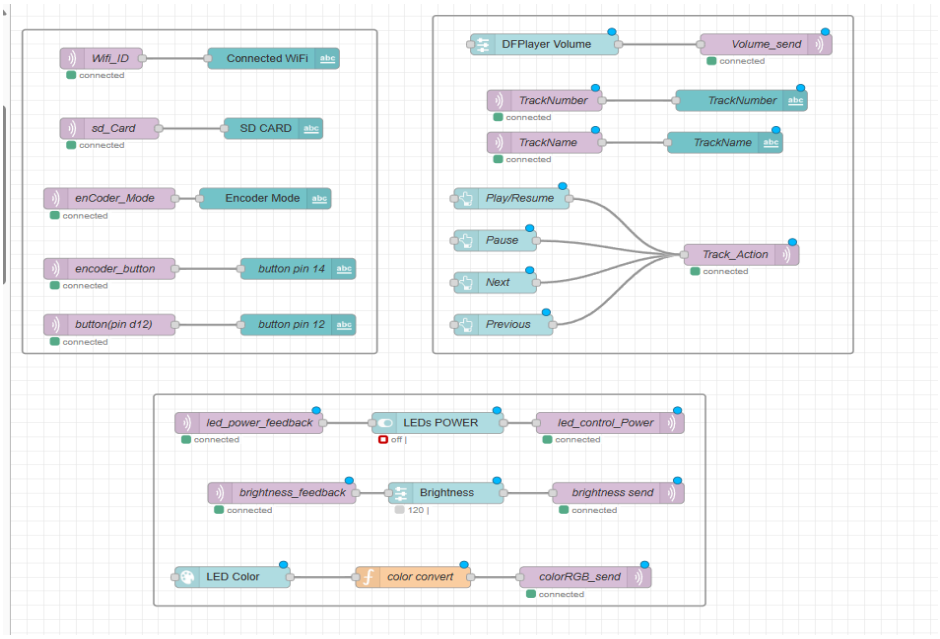
Following the successful creation of the self-hosted web server on the ESP32, I decided to take the project a step further by re-architecting it using a more powerful and industry-standard IoT protocol: MQTT. The goal was to decouple the ESP32 from the user interface, converting it into a true "headless" device that communicates its status and receives commands via an MQTT broker. The entire user interface would be rebuilt from scratch on the Node-RED platform.

Architectural Migration and Node-RED Flow Development:

The first step was to rewrite the ESP32 firmware to remove the web server and implement the `PubSubClient` library for MQTT communication. A new, well-defined API based on MQTT topics was established for commands and status updates.

The main focus of the day was building the new control logic and user interface within Node-RED. I created a comprehensive flow, visually organizing the logic into distinct functional groups: System Status, DFPlayer Control, and LED Control.

- **Status Reception:** A series of `mqtt in` nodes were configured, each subscribing to a specific status topic that the ESP32 publishes (e.g., `esp32/status/encoderMode`, `esp32/status/sd_card`). These nodes listen for real-time data from the device.
- **Command Publishing:** UI elements like buttons and sliders were wired to `mqtt out` nodes. When a user interacts with a control on the dashboard, the corresponding node publishes a formatted message to a command topic that the ESP32 is subscribed to.
- **Data Processing:** `Function` nodes were used to process data where needed. For instance, a "color convert" function was created to format the output from the color picker into a simple string that the ESP32 can easily parse.



The New Dashboard Interface:

The result of the Node-RED flow is a clean, powerful, and real-time dashboard. The interface is logically divided into cards that mirror the project's functionality:

- **System Status/General Info:** Displays live data such as the connected WiFi network, SD card status, and the current mode and state of the rotary encoder and external buttons.
- **LED CONTROL:** Provides controls for toggling the LED power, adjusting brightness with a slider, and selecting a specific color.
- **DFPlayer:** Shows the current volume, playing track number, and track name, along with buttons for Play, Pause, Next, and Previous.

All elements on the dashboard are updated instantly as new MQTT messages arrive, providing a much more responsive and robust user experience compared to the previous polling-based web server.

Conclusion:

This migration was a critical learning experience in modern IoT architecture. By moving to an MQTT-based system, the project became significantly more modular, scalable, and standards-compliant. This approach not only creates a more efficient system but also opens the door for easy integration with other platforms like Home Assistant, cloud services, or mobile applications in the future, all without needing to modify the core ESP32 firmware.

Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day13	Tarih:

Yapılan iş:**Hybrid Control System with IR Remote and Multi-Platform Synchronization**

I focused on integrating a universal control layer via an infrared (IR) remote into the MQTT-enabled control hub. This aimed to establish a truly hybrid system, allowing seamless device control through various inputs: a physical rotary encoder, a local web interface, a Node-RED dashboard, and now, a standard IR remote. This implementation necessitated substantial firmware modifications and a new approach to synchronize multiple control inputs.

IR Signal Processing and Firmware Integration

The first day was focused on the low-level task of capturing and interpreting IR signals, and then mapping them to actions within the existing firmware.

1. IR Code Mapping: The foundational step was to listen to the IR remote and systematically record the unique hexadecimal code for every button. Using the `IRremote` library and an IR receiver connected to GPIO 18, I captured the raw data for each press. These codes were then documented in a clear mapping table, associating each remote button with its intended action and its specific hex code.

Remote Button	Action	HEX Code
Mute	DFPlayer Mute Toggle	`0xED127F80`
Power	Toggle LEDs On/Off	`0xE11E7F80`
Music	Play Track 1	`0xFD027F80`
Play/Stop	Play/Pause Toggle	`0xFB047F80`
Up Arrow	D-Pad Up (depends on mode)	`0xFA057F80`
Down Arrow	D-Pad Down (depends on mode)	`0xE41B7F80`
Left Arrow	D-Pad Left (depends on mode)	`0xF8077F80`
Right Arrow	D-Pad Right (depends on mode)	`0xF6097F80`
Enter	Toggle D-Pad Mode	`0xF7087F80`
Volume Up	DFPlayer Volume Up	`0xF30C7F80`
Volume Down	DFPlayer Volume Down	`0xFF007F80`
FFWD	DFPlayer Next Track	`0xF00F7F80`
REW	DFPlayer Previous Track	`0xF20D7F80`
Next	DFPlayer Next Track	`0xF10E7F80`
Back	DFPlayer Previous Track	`0xE6197F80`

2. Firmware Logic (`processIRCode` function): With the code map established, a large `switch-case` statement was implemented in the `processIRCode` function. This function acts as the central translator for IR commands. It takes a received hex code as input and executes a corresponding action.

- The Power button (`0xE11E7F80`) toggles the `ledsOn` boolean variable.
- The Volume Up button (`0xF30C7F80`) calls the `myDFPlayer.volumeUp()` function.

- The Mute button (`0xED127F80`) triggers a custom mute/unmute logic for the DFPlayer.

3. Modal D-Pad Control: To maximize the utility of the remote's D-pad (up, down, left, right arrows), I implemented a modal system controlled by the "Enter" button. This allows the D-pad to perform different functions depending on the selected mode:

- **In "LED D-Pad Control" mode:** The arrows control LED brightness and cycle through predefined colors.
- **In "DFPlayer D-Pad Control" mode:** The arrows control volume and switch between the previous/next tracks.

4. Hybrid Communication Architecture: A crucial part of the firmware update was ensuring all control methods remained synchronized. The ESP32 was programmed to operate both a **Web Server** (for a direct-access local UI) and an **MQTT Client** simultaneously. Whenever a state change is triggered—whether from the IR remote, the rotary encoder, or the web UI—the `publishStatusMQTT()` function is called. This function immediately sends the new status to the MQTT broker, ensuring the Node-RED dashboard is always up-to-date, regardless of how the device was controlled.

```
void processIRCode(uint32_t irCode) {
    switch (irCode) {
        case 0xF7087F80:
            irDpadMode = (irDpadMode + 1) % 2;
            break;
        case 0xFA057F80:
            if (irDpadMode == 0) { // Mode 0: LED Control
                // Increase LED brightness
                globalBrightness = min(255, globalBrightness + 15);
                updateLeds();
            } else { // Mode 1: DFPlayer Control
                // Increase DFPlayer volume
                if (dfPlayerAvailable) {
                    myDFPlayer.volumeUp();
                    currentDFPlayerVolume = myDFPlayer.readVolume();
                }
            }
            break;
```

1. **Direct Code-to-Action Mapping:** The switch (irCode) statement directly maps a raw hexadecimal code from a remote button to a specific action. For example, the code 0xF7087F80 received from the "Enter" button is assigned to toggle the irDpadMode variable.

2. **Modal Control:** This is best illustrated by the logic for the "Up Arrow" button (0xFA057F80). Instead of having one fixed function, its behavior is conditional, based on the irDpadMode state variable.

    - If irDpadMode is 0, the button controls the **LEDs** (increasing brightness).
    - If irDpadMode is 1, the same button controls the **DFPlayer** (increasing volume).

This modal approach is a powerful technique for maximizing the utility of a remote with a limited number of buttons, creating a richer and more context-aware user experience.

Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm: **Day 14:**	Tarih:

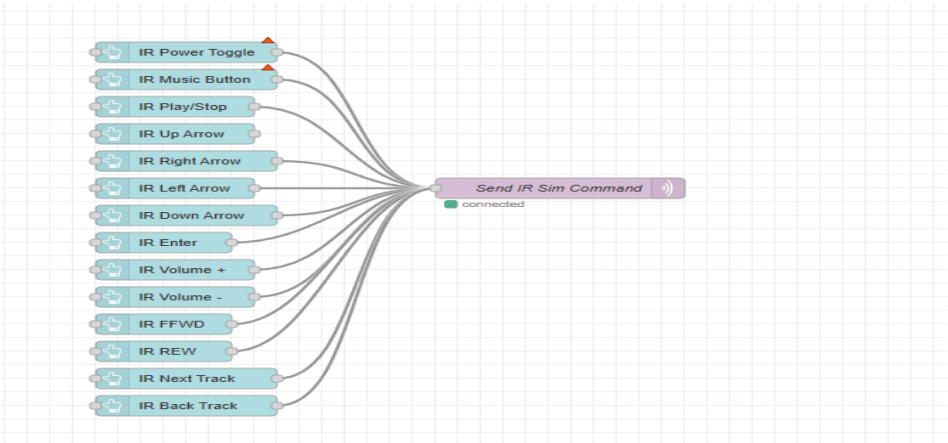
Yapılan iş: **Web & Node-RED Interface Updates**

The second day was dedicated to updating the user interfaces to reflect and fully utilize the new IR capabilities.

Web Interface Enhancements: The self-hosted web page was updated with a new card titled "Simulate IR Remote". This section contains buttons for every function of the physical IR remote. When a button on the web page is clicked, JavaScript sends a request to a new `/ir_action` endpoint on the ESP32, which then simulates a physical remote press by calling the same `processIRCode` function.

Node-RED Dashboard Integration: The Node-RED dashboard, which provides the primary remote interface, was also expanded.

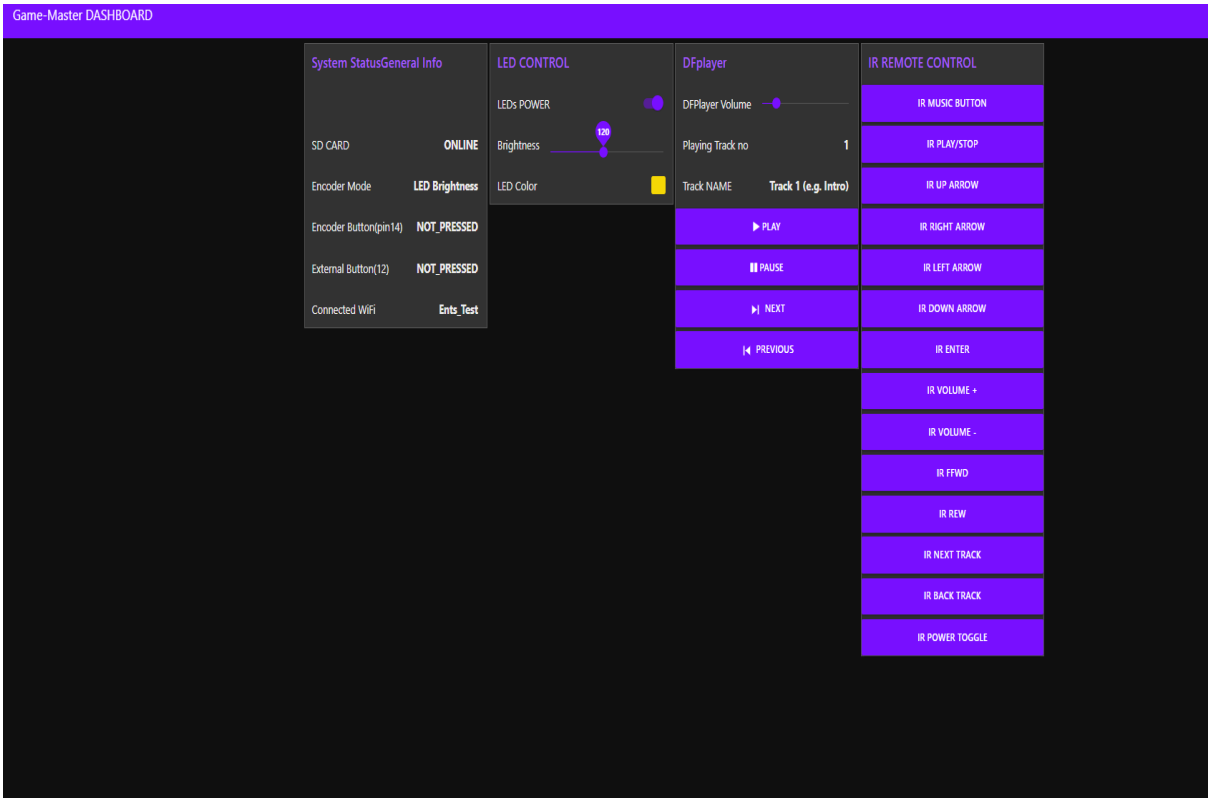
> **IR Simulation Flow:** A new flow was created with `ui_button` nodes for each IR command (e.g., "IR MUSIC BUTTON", "IR UP ARROW"). All of these buttons are wired to a single `mqtt out` node that publishes a command string to the `esp32/command/ir_action` topic.
>
> **ESP32 MQTT Handler:** The ESP32's `mqttCallback` function was updated to listen to this new topic, allowing it to process IR commands sent from the Node-RED dashboard.

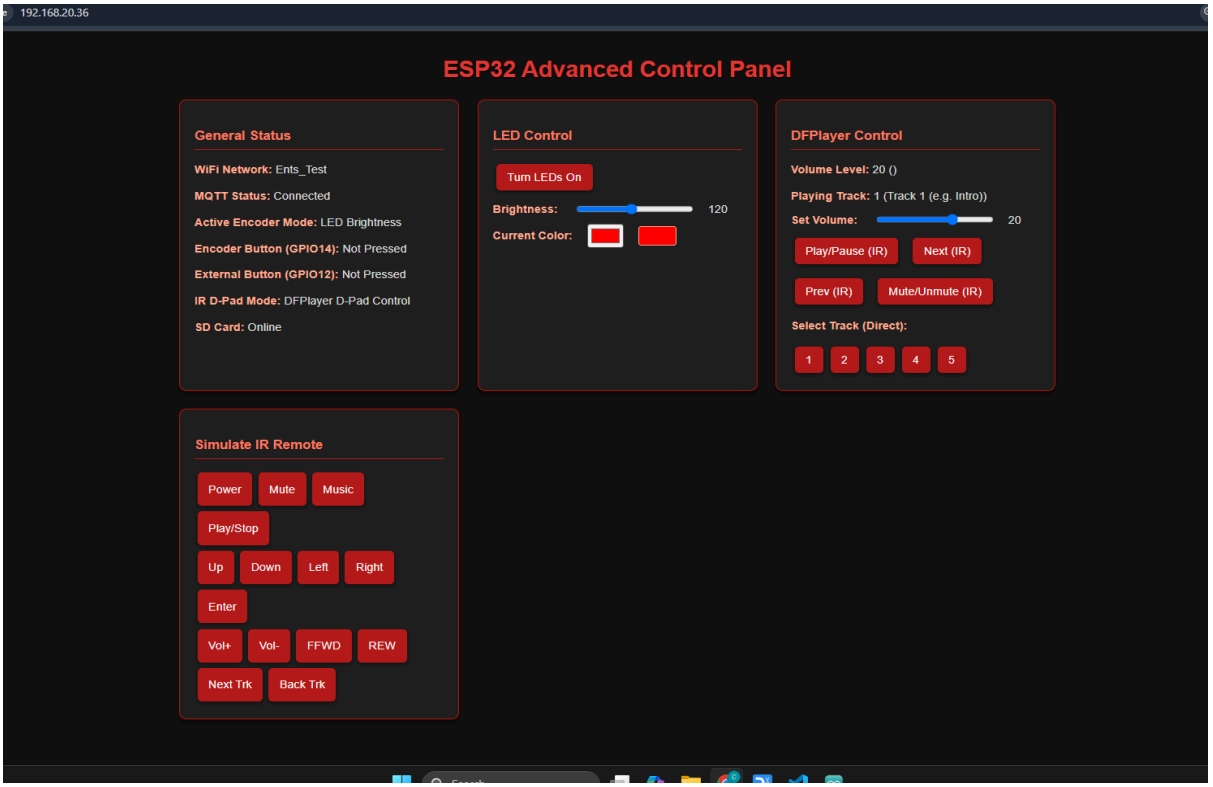


Conclusion:

This final phase of the project successfully created a truly multi-modal control system. The device can now be managed by physical controls (encoder), a direct web connection (local UI), a remote dashboard (Node-RED), and a universal IR remote. The integration of web-based and MQTT-based IR simulators demonstrates a deep understanding of how to abstract control logic and create flexible, synchronized communication systems. This project serves as a comprehensive culmination of the skills acquired during the internship, showcasing an ability to design and implement a complex, feature-rich, and user-friendly IoT device from the ground up.

Final UI: The dashboard now features a comprehensive "IR REMOTE CONTROL" section alongside the existing status and control panels, offering a complete remote management solution.





Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day15	Tarih:

Yapılan iş:**RFID Dual Reader Access System with ARGB LED Feedback**

Today's project focused on building a foundational access control system. The goal was to use an ESP32 to manage two separate MFRC522 RFID readers simultaneously. The system is designed to check scanned RFID cards against a predefined list of authorized UIDs for each reader and provide clear visual feedback using an ARGB LED strip.

System Architecture and Hardware Setup:

The core of the project is the ESP32's ability to handle two SPI devices. Since multiple SPI devices cannot share the same Chip Select (SS/CS) pin, I assigned unique SS and Reset (RST) pins for each RFID reader:

- **Reader 1:** SS on GPIO 15, RST on GPIO 22.
- **Reader 2:** SS on GPIO 4, RST on GPIO 21.
- **Visual Feedback:** A 15-pixel ARGB LED strip, connected to GPIO 2, was used for providing user feedback.

Firmware and Logic:

The firmware was designed to be straightforward and efficient.

1. **Initialization:** In the `setup()` function, the SPI bus, both MFRC522 readers, and the ARGB LED strip are initialized. The code also performs a version check on each reader to confirm they are correctly wired and communicating.
2. **Authorization Lists:** Two separate arrays, `allowedUIDs_Reader1` and `allowedUIDs_Reader2`, were created to store the 4-byte UIDs of the cards authorized for each specific reader.
3. **Main Loop:** The `loop()` function continuously polls both RFID readers to see if a new card is present (`rfid1.PICC_IsNewCardPresent()`).
4. **UID Checking:** When a card is scanned, its UID is read. The `checkUID()` function is then called, which compares the scanned UID against the appropriate authorization list.
5. **Visual Feedback (`feedbackARGB` function):**
    - If the `checkUID()` function returns `true` (indicating a match), the `feedbackARGB()` function is called with a success status. This causes the entire LED strip to blink green three times.
    - If the function returns `false`, the strip blinks red three times, clearly indicating an unauthorized attempt.

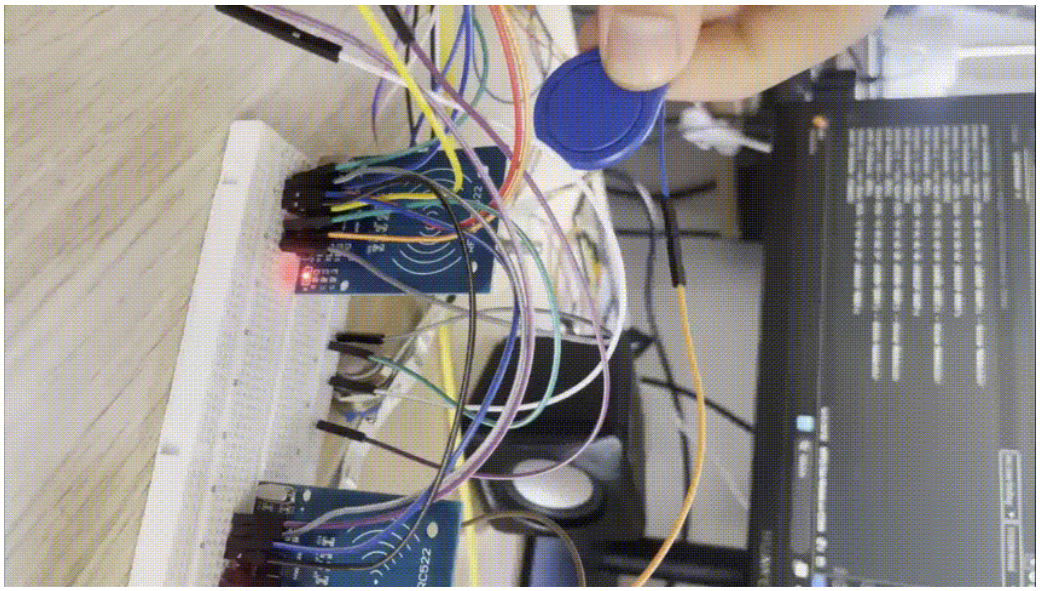
```
const byte allowedUIDs_Reader1[][4] = {
    {0xFA, 0x44, 0xA9, 0x00},
    {0x71, 0x83, 0xA1, 0x27}
};
const int numAllowedUIDs_Reader1 = sizeof(allowedUIDs_Reader1) / sizeof(allowedUIDs_Reader1[0]);

const byte allowedUIDs_Reader2[][4] = {
    {0x75, 0x41, 0x61, 0x9A},
    {0xC9, 0x51, 0x82, 0x83}
};
const int numAllowedUIDs_Reader2 = sizeof(allowedUIDs_Reader2) / sizeof(allowedUIDs_Reader2[0]);
```

Separate Authorization Lists: The system defines two distinct arrays, `allowedUIDs_Reader1` and `allowedUIDs_Reader2`. This allows you to assign different sets of cards to each reader, making it possible to have, for example, an "entry" reader and an "exit" reader with different permissions.

```
bool checkUID(MFRC522::Uid uid, int readerNumber) {
    bool match = false;
    const byte (*allowedUIDs)[4]; // A pointer to an array of 4 bytes
    int numAllowedUIDs;
    if (readerNumber == 1) {
        allowedUIDs = allowedUIDs_Reader1;
        numAllowedUIDs = numAllowedUIDs_Reader1;
    } else if (readerNumber == 2) {
        allowedUIDs = allowedUIDs_Reader2;
        numAllowedUIDs = numAllowedUIDs_Reader2;
    } else {
        return false; // Invalid reader number
    }

    // Compare the scanned UID against every UID in the selected list
    for (int i = 0; i < numAllowedUIDs; i++) {
        // memcmp performs a byte-by-byte comparison
        if (memcmp(uid.uidByte, allowedUIDs[i], 4) == 0) {
            match = true;
            break; // Exit the loop once a match is found
        }
    }

    return match;
}
```

Dynamic and Reusable Logic: The `checkUID` function is designed to be highly efficient and reusable. Instead of having two separate functions for each reader, it accepts a `readerNumber` as an argument. The `if-else` block at the beginning of the function intelligently points its `allowedUIDs` pointer to the correct array (`allowedUIDs_Reader1` or `allowedUIDs_Reader2`).

Efficient Comparison: After selecting the correct list, the function iterates through it and uses `memcmp` to perform a fast, byte-by-byte comparison between the scanned card's UID and the authorized UIDs. This is more efficient than comparing UIDs as strings.



Conclusion:

This project was a successful exercise in managing multiple SPI-based peripherals and implementing a basic but effective access control logic. The use of an ARGB LED strip for feedback proved to be a much more intuitive and user-friendly method than simple LEDs or serial monitor outputs. This system forms a solid foundation that could be expanded in the future with features like logging access attempts to an SD card, integrating with a central server via MQTT, or controlling physical locks and doors.

Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day16	Tarih:

Yapılan iş:**Research and Theoretical Understanding of Stepper Motor Control**

Today marked the beginning of a new project phase, moving from simple outputs like LEDs to the precise control of mechanical motion using a stepper motor. To prevent component damage and build a solid foundation, the entire day was dedicated to research and studying the documentation for the NEMA 17 stepper motor and the A4988 driver.

My key research findings were:

- **The Necessity of a Stepper Driver:** My primary discovery was that a microcontroller like the ESP32 cannot directly power a stepper motor. Its GPIO pins lack the required current and cannot generate the complex, phased electrical signals needed. A specialized driver, the A4988, is essential. The driver's role is to act as a bridge, interpreting simple digital signals from the ESP32 and translating them into the high-current, precisely sequenced pulses that energize the motor's coils. This elegantly simplifies the control interface to just three pins:
    - STEP: A single pulse on this pin commands the motor to move one discrete step.
    - DIR (Direction): The logic level (HIGH or LOW) on this pin determines the direction of rotation.
    - EN (Enable): This pin functions as an on/off switch for the motor's coils.
- **The Critical Role of Current Limiting:** I learned that the most critical safety procedure is current limiting. The A4988 driver is capable of delivering more current than the NEMA 17 motor is rated for, which could quickly cause the motor to overheat and suffer permanent damage. This adjustment is not an optional tweak but a mandatory calibration step performed by setting a reference voltage (Vref) on the driver's potentiometer.

- **Holding Torque and Power Management:** I researched a key characteristic of stepper motors: "holding torque." This is the motor's ability to resist external force and hold a fixed position. I discovered that this torque is only active when the motor coils are energized, a state directly controlled by the driver's `EN` pin.
    - `EN` pin LOW: The driver is active. The motor can step and will hold its position firmly.
    - `EN` pin HIGH: The driver is disabled. The coils are de-energized, which significantly reduces power consumption and allows the motor shaft to be turned freely by hand. This is crucial for manual positioning or for applications where power saving is important.

With this theoretical knowledge established, I was prepared to move on to the hardware assembly and firmware development on the following day.

Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day17	Tarih:

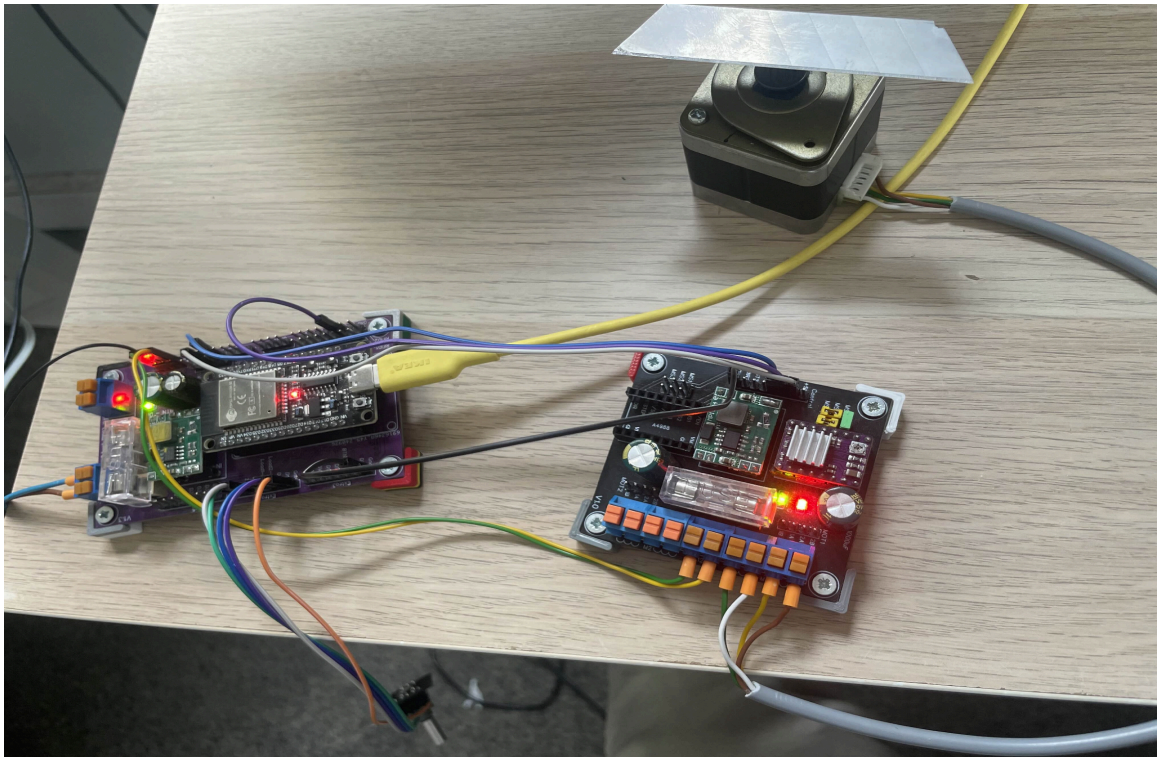
Yapılan iş: **Hardware Assembly, Calibration, and Firmware Implementation**

Building on the previous day's research, today was focused on the practical implementation of the stepper motor control system.

- **Hardware Assembly and Wiring:** The components—ESP32, A4988 driver, NEMA 17 motor, and rotary encoder—were wired together. The ESP32's GPIOs 21, 22, and 23 were connected to the driver's EN, DIR, and STEP pins. A separate 12V power supply was connected to the driver to power the motor, and all grounds were tied together to create a common reference.

- **Driver Calibration: Current Limiting (Vref Adjustment):** The most delicate and mandatory step of the hardware setup was to correctly set the current limit on the A4988 driver. I followed a careful procedure to perform this critical calibration:

  1. I first disconnected the motor from the driver to prevent any accidental movement.
  2. I powered the driver board with the 12V supply.
  3. Using a multimeter, I measured the reference voltage (Vref) between the driver's GND pin and the center of the tiny metal potentiometer on the board.
  4. I carefully turned the potentiometer with small, precise movements until the multimeter read a value in the safe range for my NEMA 17 motor (approximately 0.6V – 0.7V).
  5. After setting the correct Vref, I powered down the system and reconnected the motor, ensuring it would now operate within a safe current range



```
void loop() {
    // Check if the encoder's button has been clicked
    if (rotaryEncoder.isEncoderButtonClicked()) {
        motorAktif = !motorAktif; // Toggle the motor's active state

        if (motorAktif) {
            digitalWrite(EN_PIN, LOW); // Activate driver and holding torque
            Serial.println("Motor AKTIF Edildi!");
        } else {
            digitalWrite(EN_PIN, HIGH);
            Serial.println("Motor DURDURULDU!");
        }
        delay(50);
    }
    if (motorAktif) {
        if (rotaryEncoder.encoderChanged()) {
            mevcutEncoderPozisyonu = rotaryEncoder.readEncoder();
            if (mevcutEncoderPozisyonu > sonEncoderPozisyonu) {
                digitalWrite(DIR_PIN, HIGH);
                tekAdimAt(hizGecikmesi);
            } else if (mevcutEncoderPozisyonu < sonEncoderPozisyonu) {
                digitalWrite(DIR_PIN, LOW);
                tekAdimAt(hizGecikmesi);
            }
            sonEncoderPozisyonu = mevcutEncoderPozisyonu;
        }
    }
}
```

This code block perfectly demonstrates the project's control strategy:

1. **Motor State Control:** The `isEncoderButtonClicked()` check implements the enable/disable feature. When the button is pressed, it toggles the `motorAktif` flag and, more importantly, sets the driver's `EN_PIN` to `LOW` (active) or `HIGH` (inactive). This directly controls the motor's holding torque, as discovered during the research phase.

2. **Positional Control:** The `encoderChanged()` check detects when the user turns the knob. The logic inside is a direct translation of the control theory:

    - It reads the new encoder value.
    - It compares the `mevcutEncoderPozisyonu` (current position) with the `sonEncoderPozisyonu` (last position) to determine the direction of rotation.
    - It sets the `DIR_PIN` accordingly (HIGH for one way, LOW for the other).
    - Finally, it calls the `tekAdimAt()` function, which sends a single, brief pulse to the `STEP_PIN`, causing the motor to move by one discrete step.

Hardware Setup and Key Concepts:

The core of this project was understanding the relationship between the microcontroller, the stepper driver, and the motor itself.

1. **Stepper Driver (A4988):** I learned that a stepper motor cannot be driven directly from the ESP32's GPIO pins. A specialized driver, in this case, the A4988, is required. The driver's role is to take simple commands from the ESP32—`STEP` (move one step), `DIR` (direction), and `EN` (enable)—and translate them into the precisely sequenced and high-current electrical pulses needed to energize the motor's coils.
2. **Current Limiting (Vref Adjustment):** A critical and mandatory step before operation was setting the current limit on the A4988 driver. I learned that failing to do this could damage the motor or the driver. Using a multimeter, I measured the reference voltage (Vref) on the driver's potentiometer and carefully adjusted it to a safe level (around 0.6V-0.7V) suitable for the NEMA 17 motor's specifications.
3. **Wiring:** The components were wired according to the standard configuration. The ESP32's GPIO pins 21, 22, and 23 were connected to the driver's EN, DIR, and STEP pins, respectively. The rotary encoder was connected to other GPIOs to provide user input, and a separate 12V power supply was used for the motor and driver.

Firmware and Control Logic:

- **Rotary Encoder for Position Control:** The `AiEsp32RotaryEncoder` library was used to read input from the rotary encoder. When the encoder is turned, the `loop()` function detects a change in its value (`rotaryEncoder.encoderChanged()`).
- **Direction and Stepping:** The code compares the new encoder position to the last known position.
    - If the new value is higher, the `DIR` pin is set to `HIGH`, and a single pulse is sent to the `STEP` pin via the `tekAdimAt()` function, causing the motor to turn one step in one direction.
    - If the new value is lower, the `DIR` pin is set to `LOW`, and a pulse is sent, turning the motor one step in the opposite direction.
- **Enable/Disable Motor:** The push-button on the rotary encoder is used to toggle the motor's active state. When pressed, it toggles the `EN` (Enable) pin on the driver.
    - When the motor is active (`EN` = LOW), the driver energizes the motor coils. This allows the motor to move and also to hold its position firmly against external forces.
    - When disabled (`EN` = HIGH), the coils are de-energized. The motor shaft can be turned freely by hand, and it consumes minimal power.

The serial monitor provided real-time feedback on the motor's state, showing its current position and whether it was active or disabled.

```
16:08:55.389 -> Pozisyon: 552
16:08:55.477 -> Pozisyon: 553
16:08:56.141 -> Pozisyon: 554
16:08:56.230 -> Pozisyon: 555
16:08:56.300 -> Pozisyon: 556
16:14:59.346 -> Motor DURDURULDU!
16:15:01.077 -> Motor AKTIF Edildi!
16:15:01.157 -> Pozisyon: 555
16:15:01.426 -> Pozisyon: 554
16:15:01.426 -> Pozisyon: 555
16:15:01.426 -> Pozisyon: 554
16:15:01.500 -> Pozisyon: 553
16:15:01.533 -> Pozisyon: 552
16:15:01.575 -> Pozisyon: 551
16:15:01.575 -> Pozisyon: 550
16:15:01.575 -> Pozisyon: 549
16:15:01.575 -> Pozisyon: 548
16:15:01.609 -> Pozisyon: 547
16:15:01.652 -> Pozisyon: 546
16:15:01.652 -> Pozisyon: 545
```

Conclusion:

This project was a crucial learning experience in mechatronics and embedded control. Successfully controlling a stepper motor requires an understanding of both software logic and key hardware principles like the function of a driver and the importance of current limiting. This project provides a solid foundation for more advanced applications requiring precise motion control, such as building 3D printers, CNC machines, or robotic arms.

Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day18	Tarih:

Yapılan iş::**Remote Hardware Control via a Telegram Bot**

Today's project explored a new method of remote communication and control: creating a Telegram bot to manage the ARGB LED strip. The goal was to build a system where I could send simple text commands from anywhere in the world using the Telegram messaging app and have the ESP32 respond by controlling the connected hardware. This moves beyond local network control (like web servers or local MQTT) into the realm of cloud-based IoT interaction.

System Architecture and Key Components:

- **ESP32 as a Bot Client:** The ESP32 was programmed to act as a client for the Telegram Bot API. It connects to the internet via Wi-Fi and periodically polls Telegram's servers for new messages sent to a specific bot.
- **Telegram Bot:** I created a new bot using Telegram's "BotFather" and obtained a unique Bot Token. This token authorizes the ESP32 to interact with the API on behalf of my bot.
- **Security:** To prevent unauthorized control, the system was configured with a specific `CHAT_ID`. The ESP32 checks the Chat ID of every incoming message and will only process commands from the authorized user.

Firmware and Logic:

The firmware was built around the `UniversalTelegramBot` library, which simplifies communication with the Telegram API.

1. **Library Integration:** The `FastLED` library was used for controlling the ARGB strip, while `WiFiClientSecure` and `UniversalTelegramBot` were essential for handling the secure internet communication with Telegram.
2. **Command Handling (`handleNewMessages` function):** The core of the logic resides in the `handleNewMessages` function. This function is responsible for parsing the text of incoming messages and translating them into actions.
    - **Simple Commands:** Commands like `/strip_on` and `/strip_off` directly call the corresponding `turnStripOn()` or `turnStripOff()` functions.

- **Commands with Arguments:** For more complex commands, the `sscanf` function was used to parse arguments from the message string. For example, for a message like `/rgb 255 0 100`, `sscanf` extracts the three integer values for Red, Green, and Blue. Similarly, `/brightness 200` extracts the brightness value.

- **Status Command:** A `/strip_status` command was implemented to query the current state of the LED strip. The ESP32 replies with a formatted message showing the power state (ON/OFF), the current RGB color values, and the brightness level.

3. **User Feedback:** After successfully executing a command, the ESP32 uses the `bot.sendMessage()` function to send a confirmation message back to the user via Telegram. This provides a closed feedback loop, assuring the user that their command was received and processed.

```
#include <UniversalTelegramBot.hh>

UniversalTelegramBot bot(BOT_TOKEN, secured_client);

if (text.startsWith("/rgb ")) {
    int r, g, b;

    sscanf(text.c_str() + 5, "%d %d %d", &r, &g, &b);

    setStripColor(r, g, b); // Function call to control the hardware
    bot.sendMessage(chat_id, "LED color set: R=" + String(r), ""); // Send feedback
}
```

This code demonstrates the core logic of the project. The `UniversalTelegramBot` library provides the `bot` object, which handles all communication with the Telegram API. The `if` statement checks if an incoming message string begins with the `/rgb` command. If it does, the `sscanf` function is used to efficiently parse the three numerical values for Red, Green, and Blue directly from the text message. These values are then passed to a function that controls the physical LED strip, and a confirmation message is sent back to the user.

```
void handleNewMessages(int numNewMessages) {
    for (int i = 0; i < numNewMessages; i++) {
        String chat_id_from_message = bot.messages[i].chat_id;
        String text = bot.messages[i].text;

        // Security
        if (chat_id_from_message != CHAT_ID){
            bot.sendMessage(chat_id_from_message, "Unauthorized user", "");
            continue;
        }
        String lowerText = text;
        lowerText.toLowerCase();
        if (lowerText == "ledon" || lowerText == "/strip_on") {
            turnStripOn();
            bot.sendMessage(chat_id_from_message, "LED Strip is ON.", "");
        } else if (lowerText.startsWith("/rgb ")) {
            int r_val, g_val, b_val;
            int num_parsed = sscanf(text.c_str() + 5, "%d %d %d", &r_val, &g_val, &b_val);
            if (num_parsed == 3) {
                setStripColor(r_val, g_val, b_val);
                String msg = "LED color set: R=" + String(r_val) + " G=" + String(g_val) + " B=" + String(b_val);
                bot.sendMessage(chat_id_from_message, msg, "");
            } else {
                bot.sendMessage(chat_id_from_message, "Invalid format. Use: /rgb R G B", "");
            }
        }
    }
}
```

This code block is the brain of the Telegram bot, responsible for interpreting user commands and acting on them. It demonstrates three core concepts of the project:

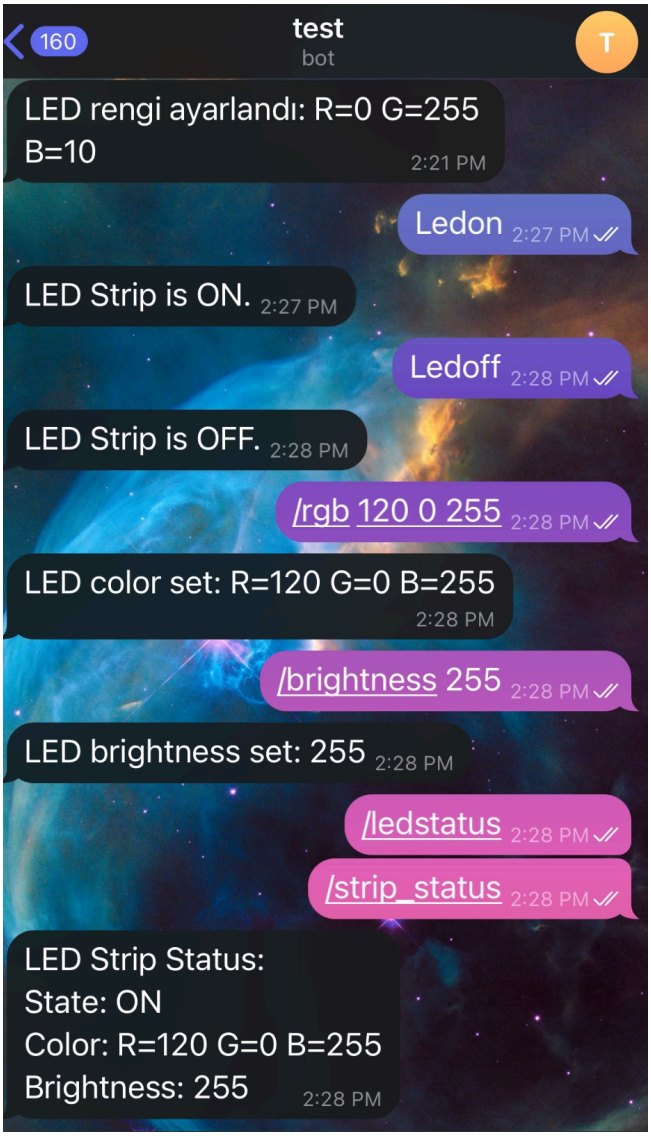
1. **Security First:** The very first check inside the loop verifies that the `chat_id` of the incoming message matches the pre-defined `CHAT_ID`. This is a critical security measure that ensures only you, the authorized user, can control the device. Any message from an unknown user is immediately rejected with an "Unauthorized" reply.

2. **Simple Command Parsing:** For basic commands like `/strip_on` or its alias `ledon`, the code performs a simple string comparison. If the message text is an exact match, it calls the corresponding function (`turnStripOn()`) and sends a confirmation message back to the user.

3. **Advanced Command Parsing with `sscanf`:** The most powerful feature is handling commands with arguments, like `/rgb 255 0 100`. Instead of complex string manipulation, the code uses the C-style function `sscanf` for this task. The expression `sscanf(text.c_str() + 5, "%d %d %d", ...)` elegantly tells the program to:

    - Look at the message text (`text.c_str()`).
    - Skip the first 5 characters (the `/rgb` part).
    - Scan for three integers (`%d %d %d`) separated by spaces.
    - Store these integers directly into the `r_val`, `g_val`, and `b_val` variables.

This method provides a robust and efficient way to extract data from user messages, making it easy to implement complex, multi-argument commands. Finally, the use of `bot.sendMessage()` creates a responsive feedback loop, confirming to the user that their command was received and executed successfully.



Conclusion:

This project was a valuable exploration into cloud-based IoT control. Using a third-party messaging platform like Telegram as a user interface is a powerful and efficient way to create globally accessible control systems without the need to develop a custom mobile application or web portal. The project successfully demonstrated how to securely connect to a public API, parse text-based commands, and provide remote feedback, adding another critical skill set for developing real-world IoT solutions.

Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day19	Tarih:

Yapılan iş:**Hybrid Control of a Stepper Motor via Telegram and Rotary Encoder**

Today's project combined the learnings from previous work in mechanical motion and cloud-based communication to create a sophisticated hybrid control system. The objective was to control a NEMA 17 stepper motor with an ESP32, allowing for precise, high-level commands to be sent from anywhere in the world via a Telegram bot, while also retaining an immediate, physical control method using a local rotary encoder.

System Architecture and Design Philosophy:

The system was intentionally designed with two distinct control pathways to offer maximum flexibility and reliability, a common requirement in real-world applications.

Remote Control (Telegram Bot): The ESP32 connects to the internet and actively listens for commands from a dedicated Telegram bot. This allows for powerful, non-local operation, such as starting a long process or moving the motor to a predefined position from a remote location.

Local Control (Rotary Encoder): A rotary encoder is directly connected to the ESP32's GPIO pins. This provides an immediate, tactile interface for an operator on-site to perform fine-tuning, make small adjustments, or manually operate the system if an internet connection is unavailable.

A key part of the design was ensuring these two methods could not conflict with each other. A global state variable was used to synchronize the motor's enabled/disabled state across both control methods.

Firmware and Implementation Details:

The project's firmware successfully integrated the `UniversalTelegramBot` library for handling the Telegram API and the `AiEsp32RotaryEncoder` library for managing the physical encoder inputs.

1. Telegram Command Handling: The handleNewMessages function was developed to be the central processing unit for all incoming Telegram commands. It robustly parses different types of text messages:

State Control: Simple, one-word commands like `motoro` and `motoroff` directly call the `enableMotor()` or `disableMotor()` functions. These functions control the logic level of the `EN` (Enable) pin on the A4988 driver, which in turn energizes or de-energizes the motor coils.

Positional Control: The bot was programmed to interpret two types of movement commands for greater flexibility:

`turn <degrees>`: This user-friendly command accepts a degree value (e.g., `turn 90`, `turn -45`). The firmware then calculates the exact number of steps required to achieve this rotation based on the motor's predefined `STEPS_PER_REVOLUTION`.

`step <steps>`: For more direct, low-level control, this command allows the user to specify the exact number of steps to move.

Security and Feedback: All commands are first validated against a predefined `CHAT_ID` to ensure only the authorized user can control the motor. After executing any command, the bot provides a confirmation message back to the user, closing the feedback loop.

This code block from the `handleNewMessages` function shows how different text commands are parsed and acted upon.

```
// ... inside handleNewMessages(int numNewMessages) ...

String lowerText = text;
lowerText.toLowerCase();

if (lowerText == "motoron") {
    enableMotor();
    bot.sendMessage(chat_id_from_message, "Motor driver ENABLED.", "");
} else if (lowerText == "motoroff") {
    disableMotor();
    bot.sendMessage(chat_id_from_message, "Motor driver DISABLED.", "");
} else if (lowerText.startsWith("turn")) {
    String valStr = text.substring(4);
    float degrees = valStr.toFloat();

    // Check if the motor is enabled before moving
    if (!motorGloballyEnabled) {
        bot.sendMessage(chat_id_from_message, "Motor is disabled. Use 'motoron' first.", "");
    } else {
        rotateStepperByDegrees(degrees); // Call the function to perform the move
        bot.sendMessage(chat_id_from_message, "Motor command: turn " + valStr + " degrees.", "");
    }
}
```

This code demonstrates the text-based command processing logic. It first converts the incoming message to lowercase to handle commands like `motoron` or `MotorON` equally. It then uses an `if/else` if structure to identify the command. For commands with arguments like `turn`, it uses `.startsWith()` to match the keyword and then `.substring()` and `.toFloat()` to extract the numerical value. Crucially,

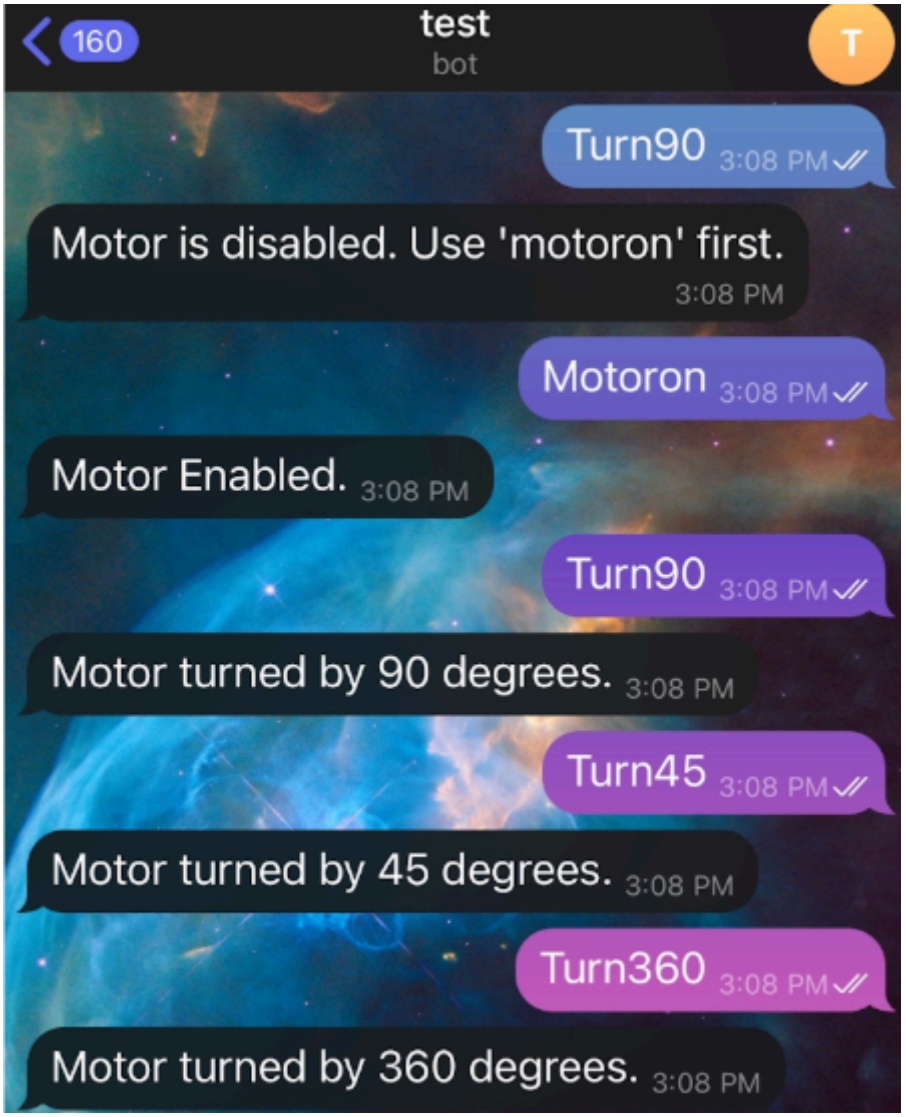
it checks the `motorGloballyEnabled` flag before calling the movement function, ensuring that remote commands respect the physical state of the device.

2. Local Encoder Logic and Synchronization: The rotary encoder provides a seamless physical interface.

- **Rotation:** Turning the encoder knob triggers the `stepOnce()` function, moving the motor one discrete step at a time in the corresponding direction. This is ideal for precise manual adjustments.
- **Button Press & Synchronization:** Pressing the encoder's built-in button toggles the `motorGloballyEnabled` boolean variable. This variable is the key to synchronizing the system. Both the local encoder's movement logic and the Telegram command handlers check the state of this variable before attempting to move the motor. This prevents any movement command from being executed if the system has been physically disabled by the operator.

```
volatile boolean motorGloballyEnabled = false;

if (rotaryEncoder.isEncoderButtonClicked()) {
    if (motorGloballyEnabled) {
        disableMotor();
    } else {
        enableMotor();
    }
    while(rotaryEncoder.isEncoderButtonClicked()) delay(10);
}
void stepOnce(bool direction, int speedDelay) {
    // CRITICAL: Do not move if the motor is not enabled.
    if (!motorGloballyEnabled) return;

    digitalWrite(DIR_PIN, direction);
    digitalWrite(STEP_PIN, HIGH);
    delayMicroseconds(speedDelay);
    digitalWrite(STEP_PIN, LOW);
    delayMicroseconds(speedDelay);
}
```

Explanation: This code illustrates the core synchronization mechanism. The `motorGloballyEnabled` flag acts as the single source of truth for the motor's powered state. The first block shows how the physical encoder button directly toggles this state by calling `enableMotor()` or `disableMotor()`. The second block, the `stepOnce` function, contains the critical "guard clause": `if (!motorGloballyEnabled) return;`. This ensures that no step pulse can be generated—regardless of whether the request came from the local encoder or a remote Telegram command—if the system has been physically disabled. This makes the system safe and robust.



Conclusion: This project successfully demonstrates the creation of a sophisticated hybrid control system. By combining a cloud-based interface (Telegram) with a direct physical interface (rotary encoder), the system offers both flexibility and reliability. This approach is highly relevant in real-world scenarios where a machine might need to be controlled remotely for major operations, but also requires fine-tuning by an operator on-site. The project showcases a strong understanding of handling multiple input sources, parsing command strings, and managing the state of a mechanical system in a synchronized and safe manner.	
Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day20	Tarih:

Yapılan iş:**Project Scoping and Initial Hardware Setup**

Today, I began the most significant undertaking of my internship: the final capstone project. My goal was to synthesize all the skills I've developed—from hardware control to network communication—into a single, cohesive, and impressive final product. After some brainstorming, I finalized the concept for an interactive escape room puzzle I've named the **"Deadlight Game System."**

My first task was to move from a general idea to a concrete plan. I spent the morning whiteboarding the entire user journey and system architecture. I decided to create a more engaging experience by designing two separate but interconnected puzzles, each running on its own ESP32.

- **ESP1 - The "Deadlight Game":** This would be the main event for the players. I designed it as a two-part test of skill, combining a fast-paced **reflex challenge** using an LED strip, followed by a **memory challenge** using a dot-matrix display and buttons.
- **ESP2 - The "Hardware Puzzle":** To add depth and a sense of progression, I designed a prerequisite puzzle. Players would need to solve this more physical, mechanical challenge first to "unlock" the ability to play the main Deadlight game.
- **The Communication Link:** The crucial link between these two independent systems would be **MQTT**. This would allow the puzzles to communicate their states and trigger each other, creating a single, seamless game flow for the players.

With the plan solidified, I moved to the workbench for the hardware assembly of the first module, ESP1. This involved carefully wiring the **ESP32** to:

- A **WS2811 addressable LED strip** for the reflex game's visual effects.
- A four-module **MAX7219 matrix display** to show game information and progress.
- A **DFPlayer Mini** sound module for audio cues and voiceovers.
- Five **push buttons** for all player interactions.

Before writing any custom game logic, I followed a critical best practice: validating each component individually. For each peripheral, I loaded a basic example sketch from its respective library. I ran the `strandtest` example for the `FastLED` library to ensure every pixel on the strip was working, the `Parola_Test` sketch for the matrix display to check the wiring, and a simple playback sketch for the DFPlayer Mini to verify the SD card and UART communication.

By the end of the day, I had a fully defined project scope and a breadboard with all the core hardware for ESP1, individually tested and verified. This foundational work was crucial, as it prevents difficult

hardware-related debugging down the line. I now have a stable platform to begin developing the core game logic tomorrow.	
Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day21	Tarih:

Yapılan iş:**Core Logic and State Machine for the "Reflex Challenge" (Step 1)**

With the hardware for the ESP1 module tested and ready, my entire focus today was on bringing the first puzzle, the "Reflex Challenge," to life. This meant moving from hardware verification to writing the core software that would define the gameplay and player experience.

Architectural Foundation: The State Machine

Before writing a single line of animation code, I knew I needed a robust structure to manage the game's complex flow. A simple series of `if-else` statements would quickly become messy and difficult to debug as more features were added. For this reason, I decided to build the entire game around a **state machine**, a fundamental and highly effective practice in embedded systems programming.

I implemented this concept using a C++ `enum` called `GameState`. This clearly defines every possible state the game can be in—from waiting for a player to start, to actively playing, to completing a stage. This approach ensures the program is always in a well-defined state, making its behavior predictable and much easier to manage.

```
enum GameState {
  WAITING_TO_START, // Game is idle, waiting for a start command
  STEP1_INIT,       // Initializing the first game stage
  STEP1_PLAYING,    // The reflex game is active
  STEP1_COMPLETE,   // The reflex game has been won
  WAITING_FOR_STEP2,// Waiting to start the memory game
  STEP2_INIT,       // Initializing the memory game
  STEP2_PLAYING,    // The memory game is active
  GAME_WON          // Both stages have been won
};
GameState currentState = WAITING_TO_START; // The game starts in an idle state
```

Explanation: Using this state machine structure dramatically cleans up the main `loop()` function. Instead of complex conditional logic, the loop simply contains a `switch` statement that checks the `currentState` and calls the appropriate function for that state (e.g., `handleStep1()`, `handleStep2()`). This makes the code highly organized and scalable.

Gameplay Logic Implementation

With the state machine structure in place, I began coding the logic for the `STEP1_PLAYING` state. I used the `FastLED` library to create a moving "progress bar" effect with a green color. The core challenge is a randomly timed event: at an unpredictable point in the animation, a block of LEDs briefly turns red.

The firmware constantly checks for a button press from the player. If the press occurs during the brief "alert window" when the LEDs are red, the system registers a success. Upon a successful catch, the program plays a positive sound effect via the DFPlayer, flashes the entire LED strip green, and uses the `MD_Parola` library to add the next letter of the word "D-E-A-D" to the matrix display. This provides clear, multi-sensory feedback to the player, confirming their success and showing their progress towards completing the stage.

By the end of the day, I had a fully playable prototype of the first game stage. The state machine architecture proved invaluable in keeping the code clean and manageable. The system can now successfully run the reflex game loop, handle player input, and provide clear feedback on both the LED strip and the matrix display. The next logical step is to build a user interface to control and monitor this game remotely.

Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day22	Tarih:

Yapılan iş:**Gameplay Implementation and Dynamic Difficulty for the "Reflex Challenge"**

Building upon the state machine architecture from yesterday, my entire focus today was on bringing the first puzzle, the "Reflex Challenge," to life. My goal wasn't just to make it functional, but to make it feel like a real, engaging game with clear feedback for the player and a progressively increasing challenge.

Gameplay Loop Implementation

The core logic was built within the `handleStep1()` function, which acts as the engine for this stage of the game. I started by coding the main visual element: a green "progress bar" that sweeps across the addressable LED strip. To make the game unpredictable, I programmed the red "alert" block to appear at a new random position (`targetStep`) in each round. When the animation's progress reaches this target step, a flag (`alertIsActive`) is set to `true`, and the LEDs at that position turn red for a brief period defined by `currentReactionWindow`.

Success, Failure, and Progression Logic

The most critical part was handling the player's input and providing immediate, multi-sensory feedback.

- **Successful Catch:** The firmware constantly checks if the main button is pressed. If a press is detected, it first checks if the `alertIsActive` flag is true. If it is, the system registers a success. A positive sound effect is played via the DFPlayer, the entire LED strip flashes green, and the `showOnMatrix()` function is called to add the next letter of the target word "D-E-A-D" to the matrix display.
- **Failure:** If the player presses the button when `alertIsActive` is false (either too early or after the red light has disappeared), it's registered as a failure. An error sound is played, and a "FAIL" message is shown on the matrix, forcing the player to retry the level.
- **Dynamic Difficulty:** To prevent the game from becoming repetitive, I implemented a dynamic difficulty system. After each successful catch, the `resetStep1Round()` function is called. This function slightly decreases the values of `currentAnimationInterval` (making the bar move faster) and `currentReactionWindow` (giving the player less time to react). This ensures the challenge escalates as the player progresses.

- **Stage Completion:** Once the player successfully completes all four levels and spells out "DEAD" on the matrix, the game state transitions to `STEP1_COMPLETE`. A final success sound for this stage is played, and the system prepares to introduce the second puzzle.

```
void handleStep1() {
    // Check for a button press at any time
    if (digitalRead(BUTTON_1) == LOW) {
        // Was it a valid press (during the red light alert)?
        if (alertIsActive && (millis() - alertTimer < currentReactionWindow)) {
            // --- SUCCESS ---
            step1Level++;
            playSound(SOUND_S2_SUCCESS);
            fill_solid(leds, NUM_PIXELS, SUCCESS_COLOR);
            FastLED.show();

            // Build and display the progress word (e.g., "D", "DE", "DEA", "DEAD")
            String progressWord;
            for(int i=0; i<step1Level; i++) {
                progressWord += STEP1_TARGET_WORD[i];
            }
            showOnMatrix(progressWord.c_str());
            if (step1Level >= 4) {
                currentState = STEP1_COMPLETE;
            } else {
                resetStep1Round();
            }
        } else {
            // --- FAILURE (Pressed too early or missed) ---
            playSound(SOUND_S2_ERROR);
            showOnMatrix("FAIL!");
        }
        delay(200);
        return;
    }
}
```

Explanation: This function perfectly encapsulates the game's core loop. It continuously checks for player input while managing the LED animation in the background. The logic cleanly separates the conditions for success (pressing the button during the `alertIsActive` window) and failure, providing immediate auditory and visual feedback and advancing the game state accordingly. The call to `resetStep1Round()` after a success is what implements the dynamic difficulty.

Conclusion for the Day: By the end of Day 3, the "Reflex Challenge" was a fully implemented, polished, and challenging mini-game. The combination of visual feedback, sound effects, and escalating difficulty created an immersive experience. The game now correctly transitions to the next stage upon completion, setting the stage for developing the "Memory Challenge" tomorrow.	
Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day23	Tarih:

Yapılan iş:**Deadlight Game System - Core Logic and Hardware Integration**

Day 4: "Memory Challenge" (Step 2) and Final Integration

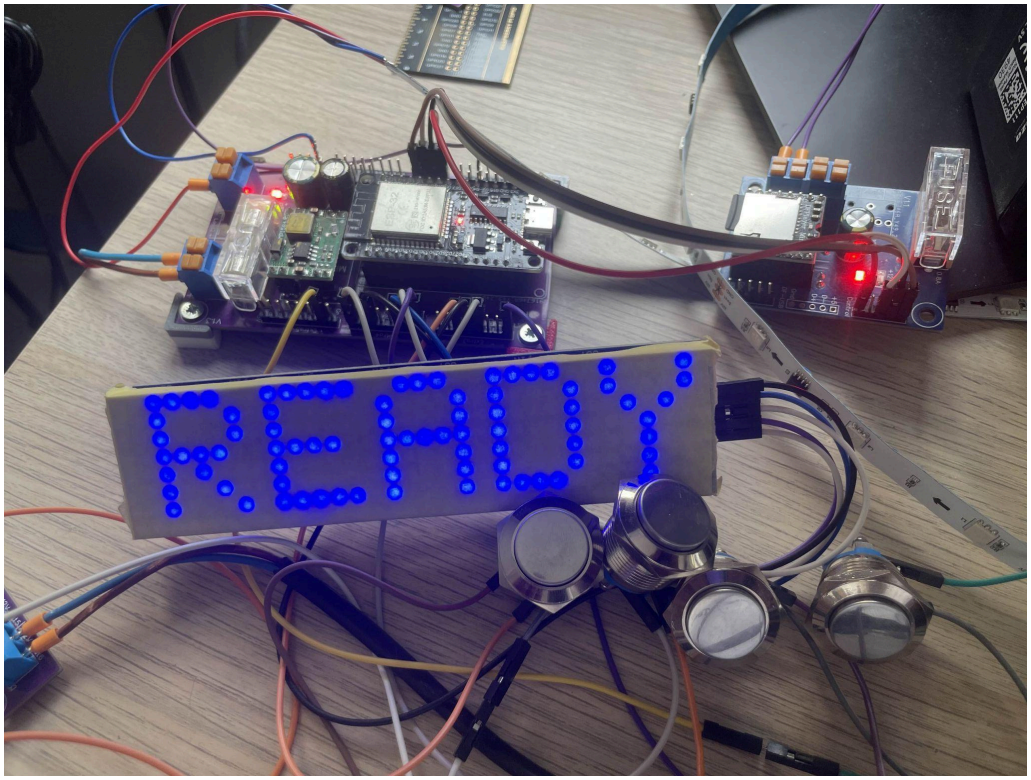
This day focused on building the second game stage and integrating it into the main state machine to create a complete game from start to finish.

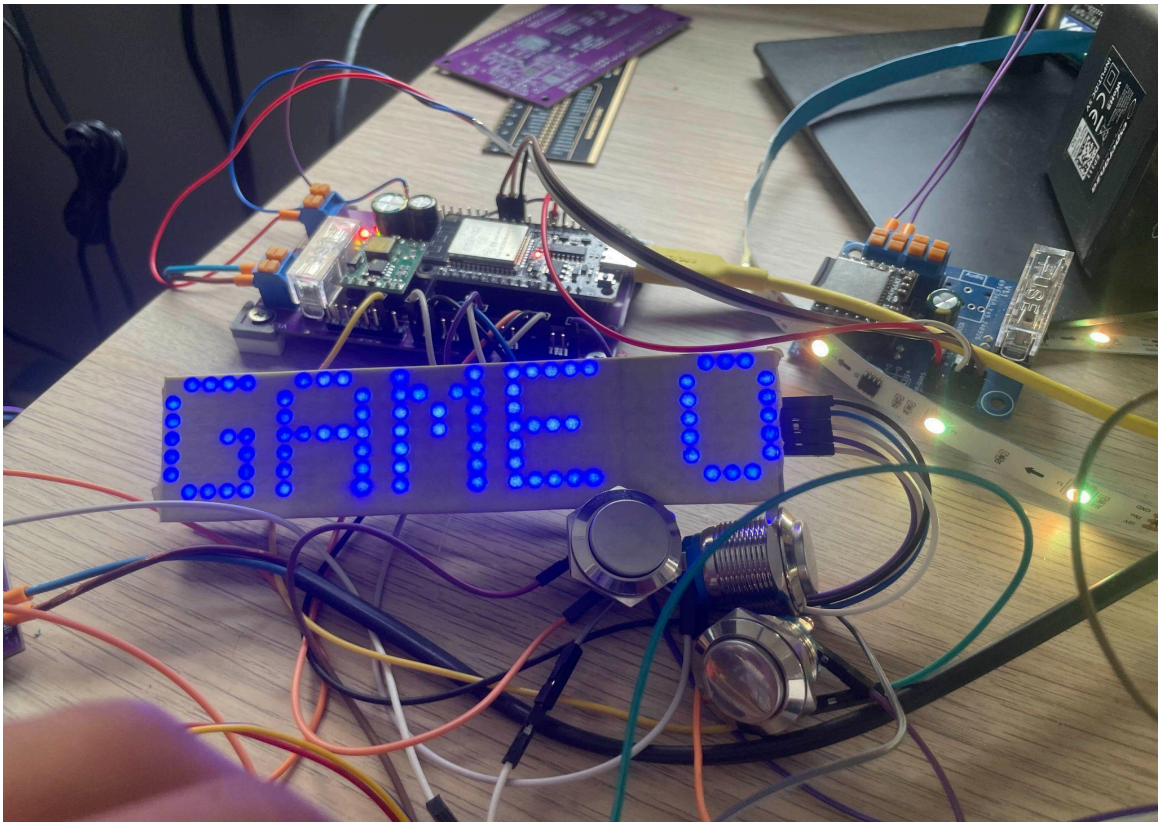
- **Transition Phase:** After Step 1 is won, the system enters the `WAITING_FOR_STEP2` state. The matrix displays "STEP 2?", and a sound prompts the player to get ready for the next challenge. The game proceeds only after the player presses the main button to confirm.

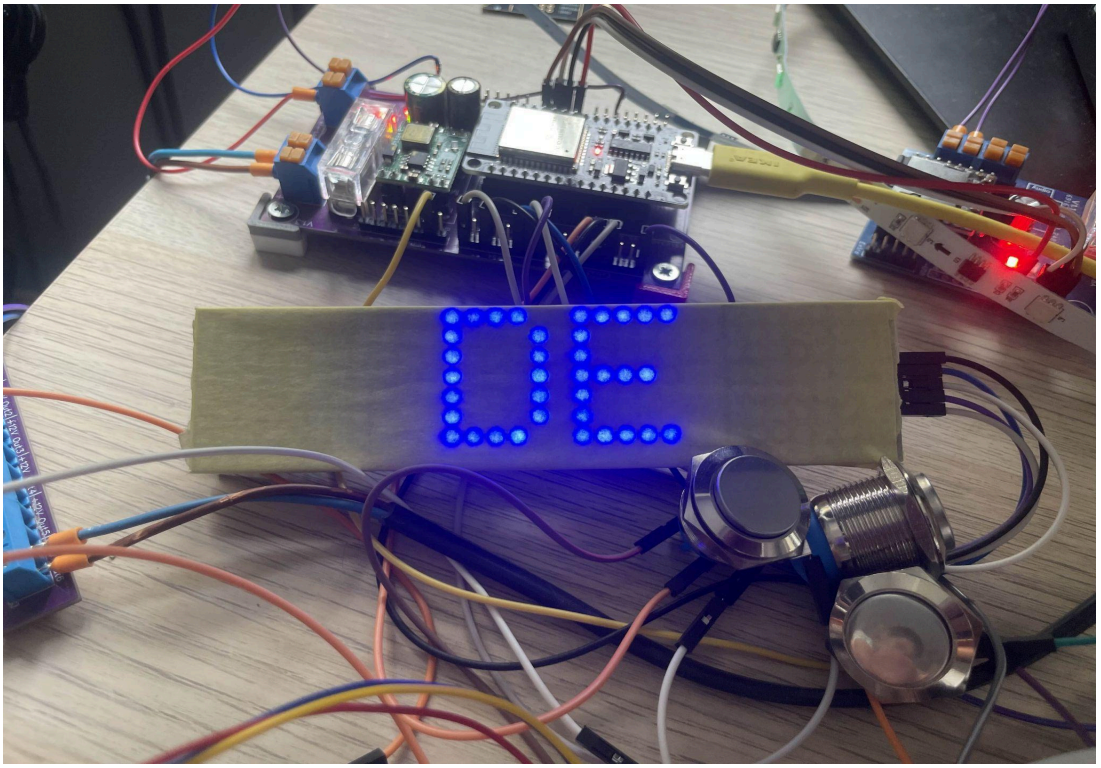
- **Memory Game Implementation:**

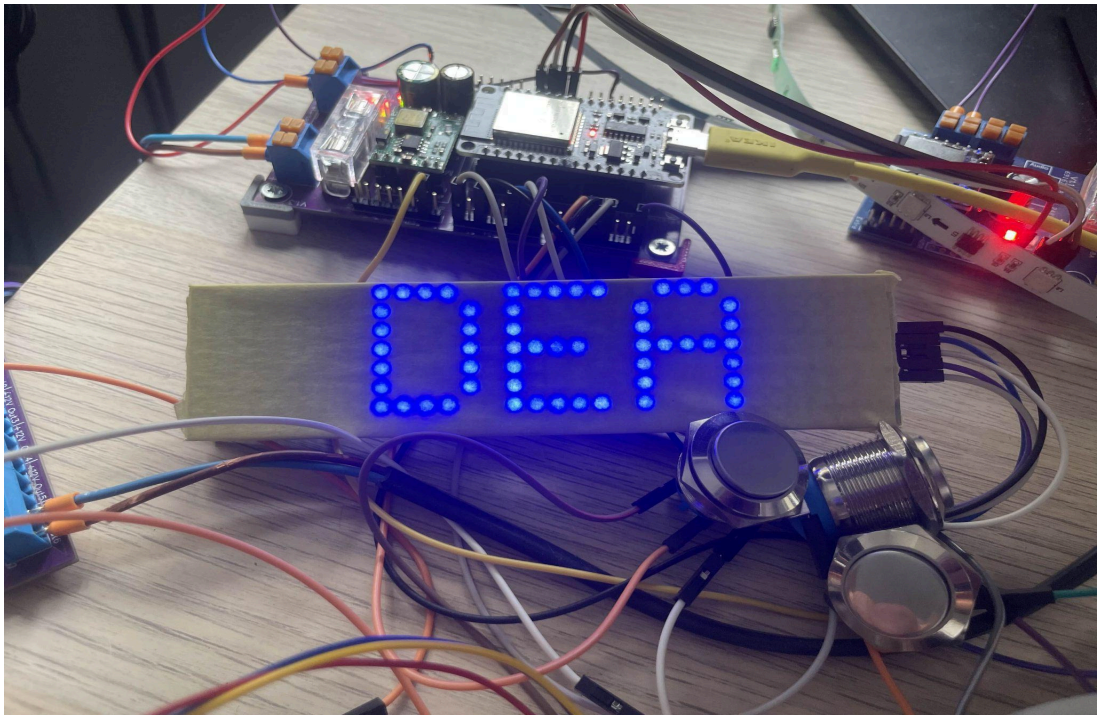
    1. **Display Code:** The game enters `STEP2_INIT`. The matrix briefly displays a 4-character pattern from the `STEP2_LEVELS` array (e.g., "OXOX").
    2. **Player Input:** The game transitions to `STEP2_PLAYING`. The matrix clears to "- - - -", and the player is given five seconds to input the pattern. The four secondary buttons correspond to the four positions on the matrix. As the player presses buttons, "X"s appear on the screen to provide feedback.
    3. **Validation:** After the input window closes, the firmware compares the player's input with the correct pattern.
    4. **Success/Win Condition:** If the input is correct, the matrix displays "OK!" or "SUCCESS", a success sound plays, and the game advances to the next level. If it's incorrect, "FAIL" is shown, and the level repeats. After completing all four memory levels, the main game state changes to `GAME_WON`.

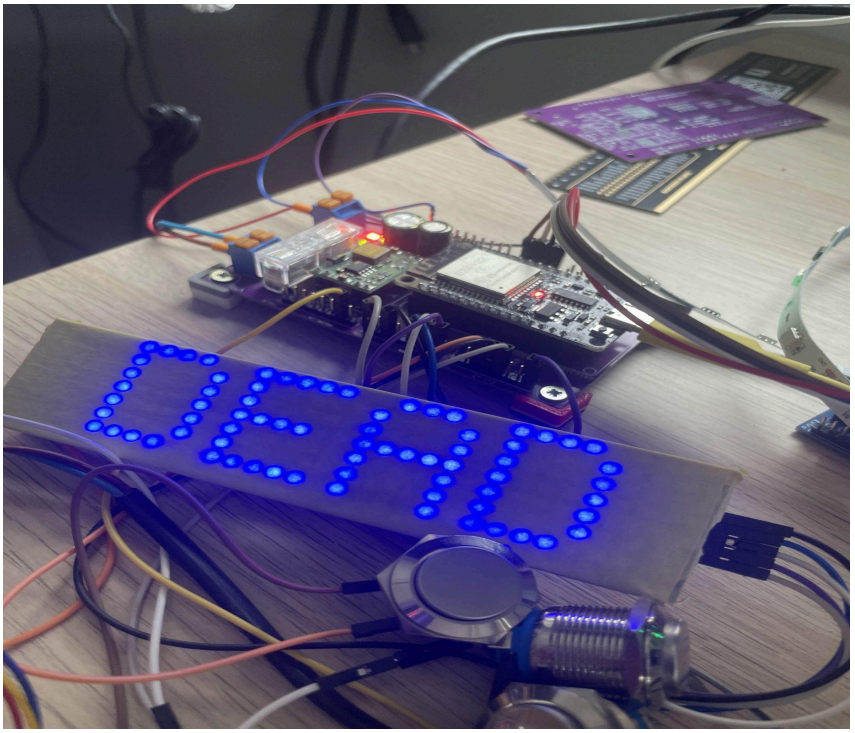
- **Final Win Sequence:** In the `GAME_WON` state, the firmware triggers a final victory sequence. The matrix displays "FINISH", a grand finale sound is played, and the LED strip begins to blink green, signifying that the player has successfully beaten the entire "Deadlight" game.

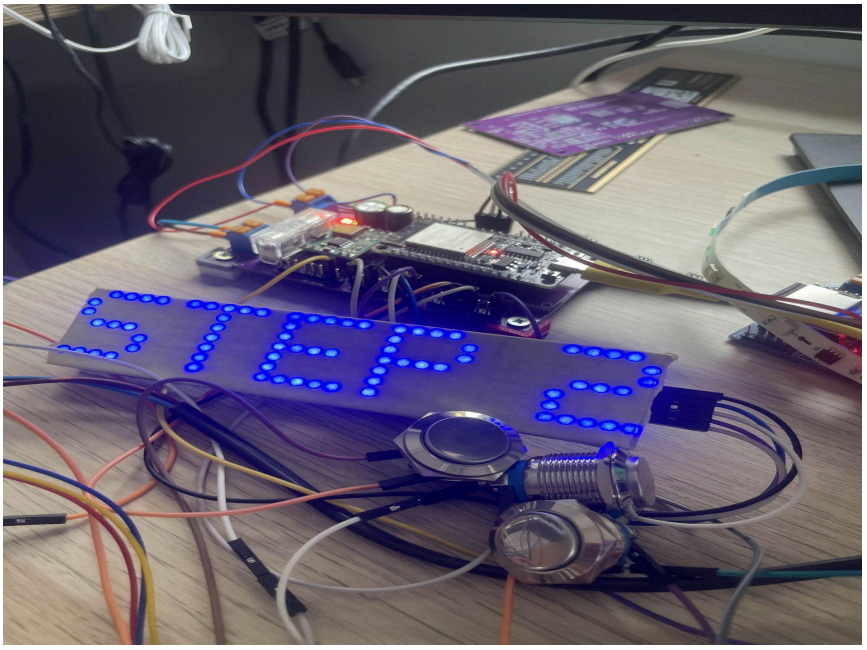

READY

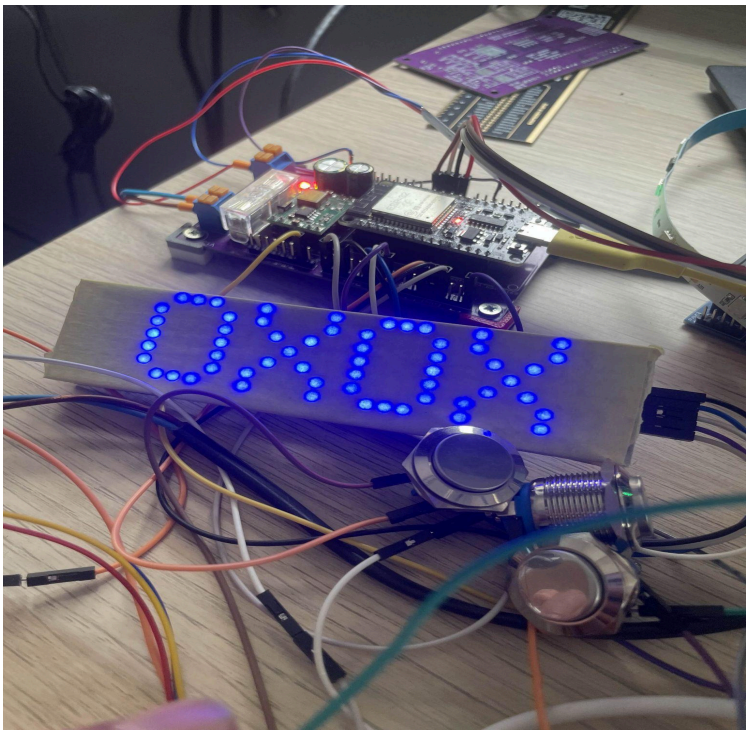

GAME 1

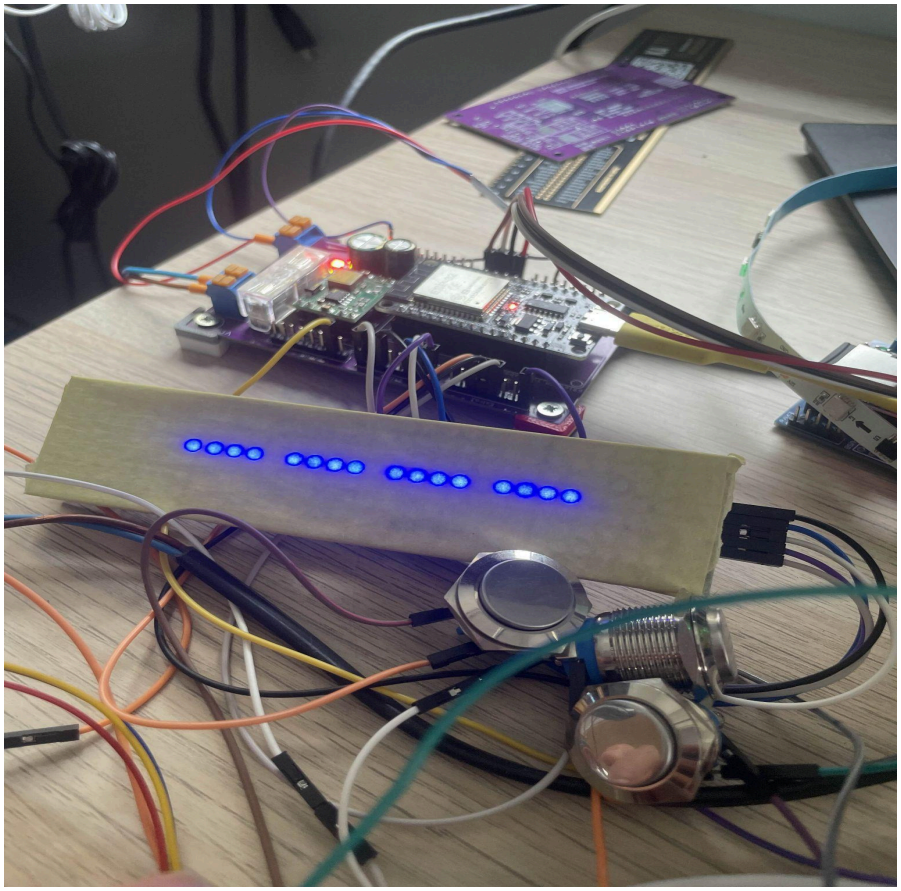

OF

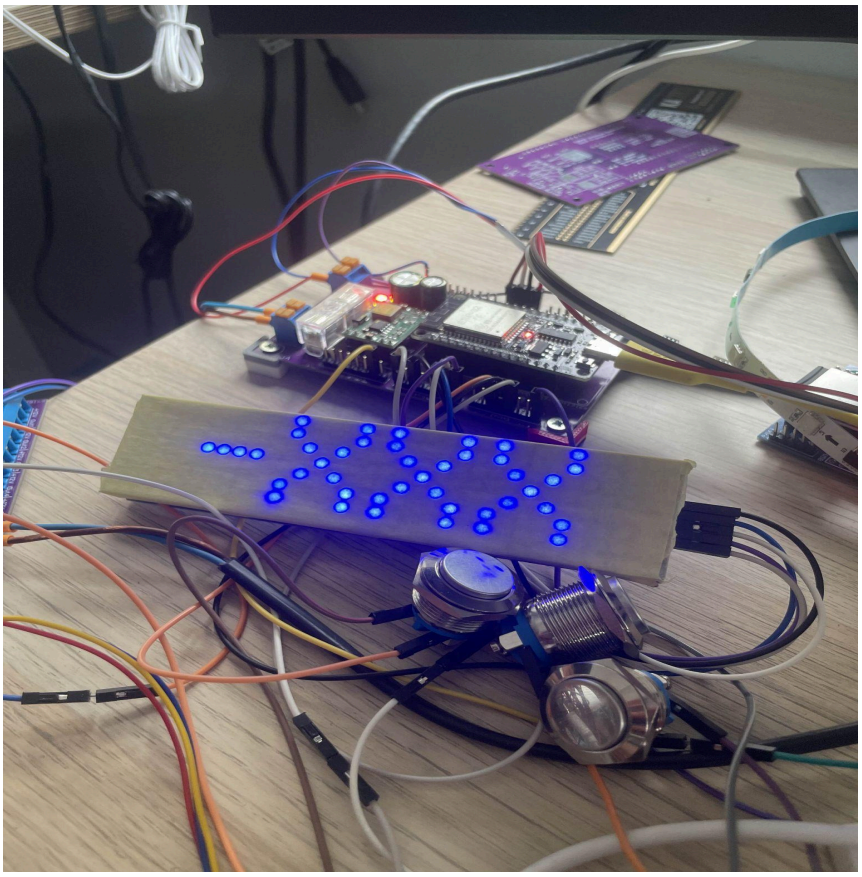

OFFA

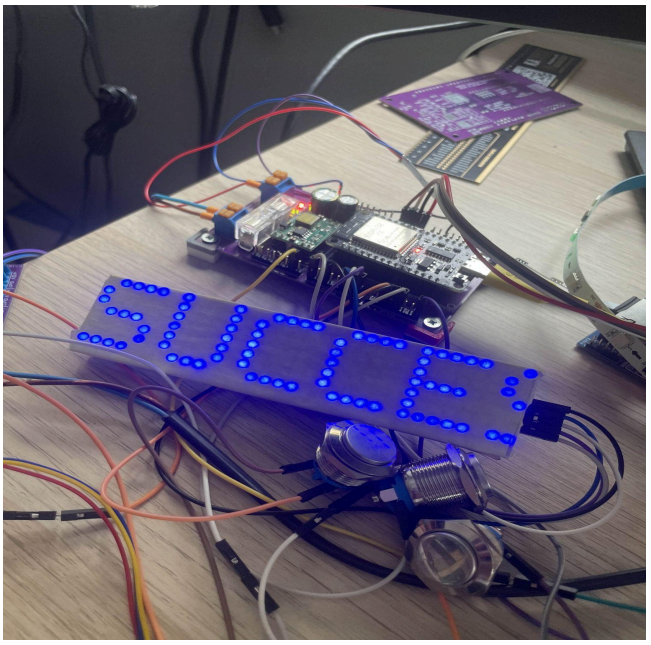

DEAD


STEAL


ONOFF







Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

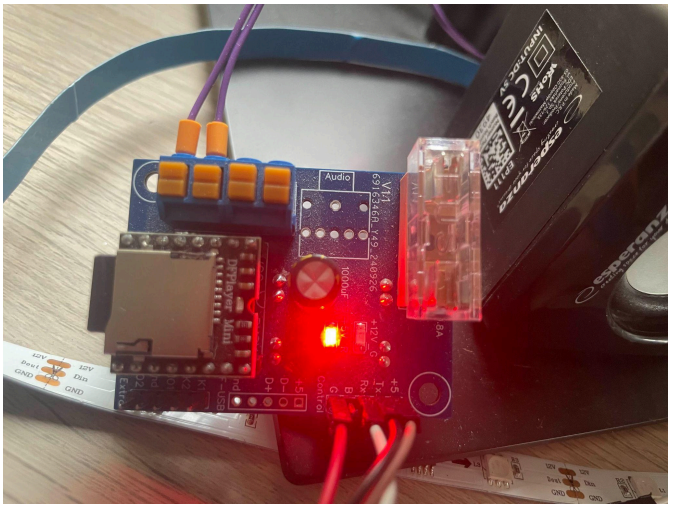
Bölüm:day24	Tarih:

Yapılan iş:**Sound System Design and DFPlayer Mini Integration**

With the core game mechanics prototyped, today's focus was on adding a critical layer of immersion to the "Deadlight" game: the audio system. The objective was to integrate the **DFPlayer Mini MP3** module to provide players with instructions, feedback (success/error chimes), and atmospheric sounds.

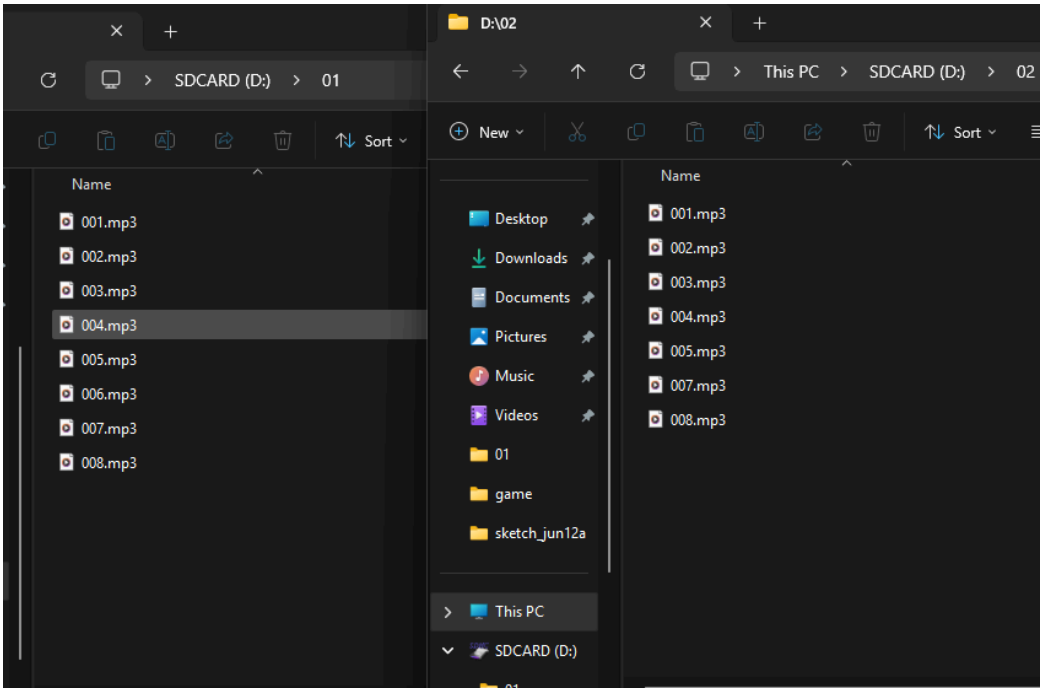
Hardware Setup and SD Card Preparation:

The DFPlayer Mini module was integrated into the ESP1 ("Deadlight Game") circuit. It communicates with the ESP32 via a serial connection (UART), requiring only two GPIO pins (RX and TX).



A crucial part of the setup was the preparation of the Micro SD card. I learned that the DFPlayer Mini requires a specific file structure and naming convention to function correctly.

- **Folder Structure:** To support the bilingual (Turkish and English) audio feature planned for the future, I created two folders on the root of the SD card: `/01` for Turkish sounds and `/02` for English sounds.
- **File Naming:** Inside each folder, the audio files were named sequentially starting from `001.mp3`, `002.mp3`, and so on. This numerical naming is essential, as the `DFRobotDFPlayerMini` library identifies tracks by their number, not their filename



Firmware Integration and Control Logic:

The firmware was designed to make playing sounds simple and modular, directly tying audio cues to specific game events.

1. Sound File Definitions: First, I created clear definitions for each sound effect using `const uint8_t`. This practice makes the code much more readable and maintainable than using "magic numbers" (like `myDFPlayer.play(1);`) throughout the logic.

2. Bilingual Playback Function: To handle the multilingual feature, I wrote a helper function called `playSound()`. This function takes a track number as input, checks the currently selected language (`currentGameLanguage`), and then instructs the DFPlayer to play the specified track from the corresponding folder (`/01` for Turkish, `/02` for English).

```
enum GameLanguage { LANG_TR, LANG_EN };
GameLanguage currentGameLanguage = LANG_TR; // Default language is Turkish

void playSound(uint8_t track) {
    // Determine which folder to play from based on the current language setting
    uint8_t folder = (currentGameLanguage == LANG_TR) ? 1 : 2;
    Serial.printf("--> SOUND: Playing file #%d from folder #%d.\n", track, folder);

    // Send the command to the DFPlayer Mini module if it's online
    if (dfPlayerStatus) {
        myDFPlayer.playFolder(folder, track);
    }
}
```

This function abstracts the complexity of the bilingual system. The main game logic no longer needs to worry about which folder to use; it simply requests a "type" of sound (e.g., success, error), and the `playSound` function handles the rest based on the user's language setting. This implementation provides the project with a crucial layer of immersion and is robust and easily expandable for any new sounds or languages that might be added in future stages of development.

Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day25	Tarih:

Yapılan iş:**"Hardware Puzzle" Assembly & Voltmeter Game**

With the core software for the first game on ESP1 established, the focus for today shifted to building the second, entirely separate, and more physical part of the escape room experience. This module, running on a second ESP32 (ESP2), is designed as a prerequisite challenge that players must solve to unlock the main game.

- **Hardware Assembly:** Today was dedicated to the physical construction of this complex interactive device. A new set of peripherals was integrated with the ESP2:
    - A **NEMA 17 stepper motor** was connected to an **A4988 driver board**. A disk was mounted on the motor shaft, and two different RFID tags were attached to it, creating a physical game selector mechanism.
    - An **MFRC522 RFID reader** was positioned to read the tags on the rotating disk as it passed by.
    - An analog **voltmeter** was wired to a PWM-capable GPIO pin. This allows the ESP32 to control the needle's position, turning a simple display into an interactive game element.
    - A **digital microphone sensor** was connected to a digital input pin for a future sound-based puzzle.
    - A custom-designed PCB was used to neatly distribute power and signals, ensuring a stable and reliable hardware foundation.
- **Firmware - The Voltmeter Puzzle:** I developed the firmware for the first challenge on this device. The goal is for the player to stop a fluctuating needle at a specific point.
    - The ESP32 generates a sweeping PWM signal using ledcWrite, which causes the voltmeter's needle to move back and forth between 0V and 10V.
    - When the player presses a button, the firmware reads the needle's "position" at that exact moment.
    - This "captured" value (e.g., 5V) is then used as an input for the next stage of the puzzle, determining how many times the stepper motor will rotate.

```
void runGame1_VoltmeterSelect() {
    unsigned long currentTime = millis();
    if (currentTime - game1_lastSelectionUpdateTime >= game1_selectionInterval) {
        game1_lastSelectionUpdateTime = currentTime;
        game1_targetVoltage += game1_voltageDirection;
        if (game1_targetVoltage >= 10.0) { game1_voltageDirection = -0.1; }
        else if (game1_targetVoltage <= 0.0) { game1_voltageDirection = 0.1; }
        int dutyCycle = multiMap(game1_targetVoltage * 100, ...);
        ledcWrite(pwmChannel, dutyCycle);
    }
    if (digitalRead(GAME1_BUTTON_PIN) == LOW && (millis() - game1_lastButtonPressTime >
game1_debounceDelay)) {
        game1_lastButtonPressTime = millis();
        game1_selectedLevel = round(game1_targetVoltage);
        // Calculate how many steps the motor needs to move based on the selected level
        long totalSteps = (long)(game1_selectedLevel) * STEPS_PER_360_DEG;
        stepper.moveTo(totalSteps);
        currentGameMode = MODE_STEPPER_MOVING;
    }
}
```

This code demonstrates real-time hardware control for a game purpose. Instead of just reading a sensor, the ESP32 is actively *creating* an analog effect with a digital signal (PWM). The `multiMap` function is a crucial piece of calibration, translating the desired linear voltage (0-10V) to the non-linear PWM values required to drive the physical voltmeter needle accurately.

Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day26	Tarih:

Yapılan iş:**RFID Trigger, Microphone Game, and System Logic**

Today's work focused on completing the logic for the ESP2 puzzle, implementing its second game, and establishing the communication link that makes it a selector for the main game.

- **The RFID Trigger Logic:** After the stepper motor completes its rotation (as determined on Day 6), the ESP2 must identify which game to start. This is done by reading the RFID tag that is now positioned in front of the reader. The `handleCardAndStartGame()` function was written for this. It reads the unique ID (UID) of the card and compares it against two hardcoded UIDs.

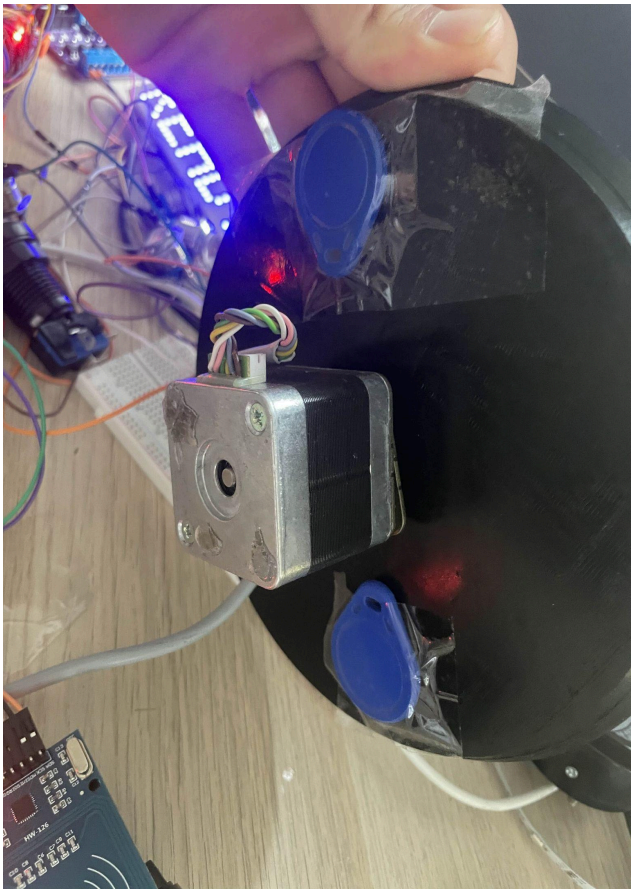
- **The Microphone Puzzle:** If the "local game" card is scanned, the ESP2 initiates its second puzzle. An ARGB LED lights up red. Players must blow into the microphone sensor. The firmware detects this sound via a digital pin and reduces the LED's brightness. The puzzle is solved when the players successfully "extinguish" the light completely. Upon winning, the ESP2 sends an MQTT message to the system, signaling its completion.

- **Inter-Device Communication:** This is the core of the integration.

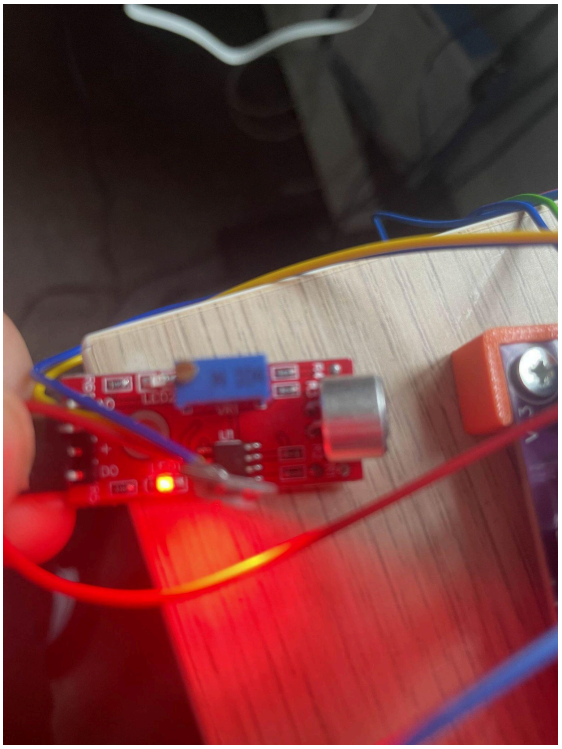
    - If the player scans the **first RFID card**, the ESP2 publishes an MQTT message to ESP1's control topic (`game/control`) with a command to `start_game`.
    - If the player scans the **second RFID card**, the ESP2 starts its local microphone game. Once this game is won, ESP2 publishes an event message (`{"event":"game2_won"}`) to a status topic. This message will later be used by the Node-RED dashboard to automatically trigger the next game in the sequence.

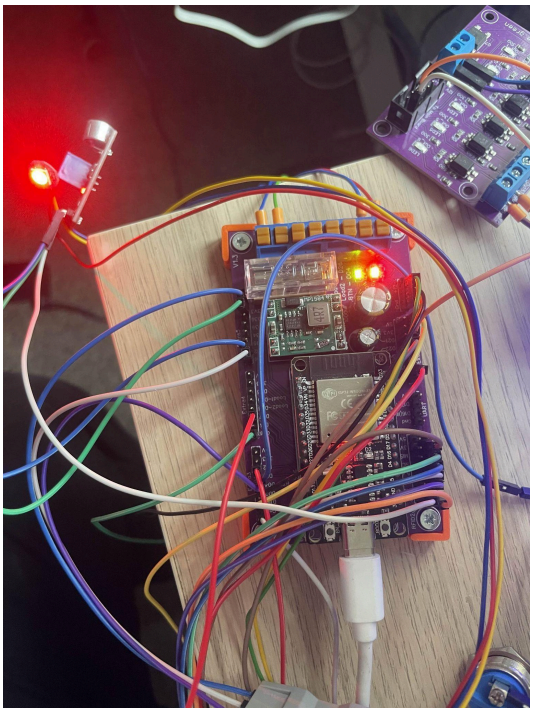
```
void handleCardAndStartGame() {
    // If no new card is present, do nothing.
    if (!mfrc522.PICC_IsNewCardPresent() || !mfrc522.PICC_ReadCardSerial()) {
        return;
    }
    const byte CARD_UID_GAME1[] = {0x75, 0x41, 0x61, 0x9A}; // Starts the "Deadlight" game on ESP1
    const byte CARD_UID_GAME2[] = {0xFA, 0x44, 0xA9, 0x00}; // Starts the local microphone game
    if (compareUIDs(mfrc522.uid.uidByte, CARD_UID_GAME2, 4)) {
        startMicrophoneGame();
    } else if (compareUIDs(mfrc522.uid.uidByte, CARD_UID_GAME1, 4)) {
        char jsonPayload[50];
        sprintf(jsonPayload, "{\"action\":\"start_game\", \"level\":%d}", game1_selectedLevel);
        client.publish(topic_game_control, jsonPayload);
        currentGameMode = MODE_HOMING;
    }
    mfrc522.PICC_HaltA();
}
```

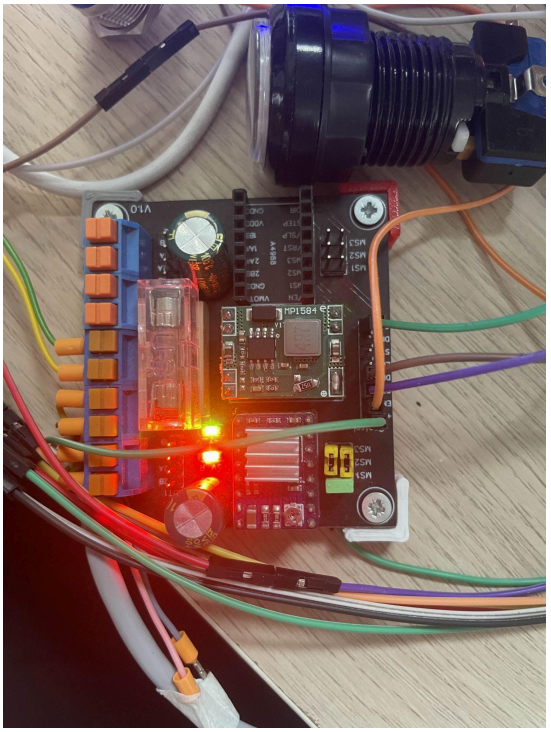
This code demonstrates a powerful and flexible design pattern. The system uses physical objects (RFID cards) as keys to unlock different branches of the game flow. The logic cleanly separates the local action (`startMicrophoneGame()`) from the remote action (`client.publish(...)`), showcasing a clear understanding of how to orchestrate a multi-device system over a network.

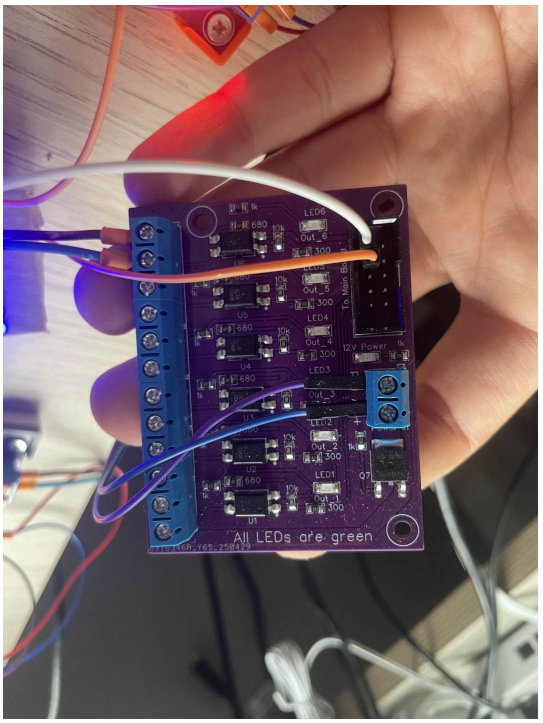


HW-126
RFID-RC522









Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day27	Tarih:

Yapılan iş:**Developing the Polished Web Interface for ESP1 ("Deadlight Game")**

With the core game logic and hardware for the ESP1 module fully functional, today's objective was to create a professional, feature-rich, and user-friendly web interface for real-time control and monitoring. The goal was to move beyond a simple diagnostic tool and build a polished "Game Control Panel" that a game master could intuitively use.

Architectural Choice: WebSockets for Real-Time Communication

A key decision was to use **WebSockets** for communication between the ESP32 and the browser, powered by the `ESPAsyncWebServer` and `AsyncTCP` libraries. Unlike traditional HTTP requests where the browser must constantly ask the server for updates (polling), WebSockets create a persistent, two-way connection. This allows the ESP32 to instantly "push" status updates to the web interface the moment a change occurs (e.g., a player completes a level, the timer decrements), resulting in a truly real-time experience with minimal latency.

Backend Implementation (ESP32):

The ESP32 firmware was developed to act as a sophisticated backend for the web UI.

- **Self-Contained UI:** The entire user interface (HTML, CSS, and JavaScript) was embedded directly into the ESP32's firmware in a separate header file (`index_h.h`) and stored in program memory (`PROGMEM`). This makes the device completely self-contained, requiring no external files to be hosted.
- **JSON-Based Communication:** All data exchange between the ESP32 and the web UI is handled using structured JSON objects.
    - **Sending Status:** The `getStatusJSON()` function gathers all relevant data—current game state, level details, remaining time, system settings, and hardware status—and packages it into a single JSON string. This string is then broadcast to all connected web clients via the WebSocket.
    - **Receiving Commands:** The `processAction()` function parses incoming JSON messages from the web UI. It checks for an `action` key (e.g., `"start_game"`, `"apply_settings"`) and triggers the corresponding function in the main game logic.

```
String getStatusJSON() {
    JSONVar gameStateJson;
    gameStateJson["gameState"] = gameStateNames[currentState];
    gameStateJson["levelDetail"] = getLevelDetailString();
    gameStateJson["matrix"] = String(matrixBuffer);
    gameStateJson["timer"] = getRemainingSeconds();
    gameStateJson["wifiStatus"] = wifiStatus;

    JSONVar settings;
    settings["volume"] = gameVolume;
    settings["base_animation_interval"] = base_animation_interval;
    gameStateJson["settings"] = settings;
    return JSON.stringify(gameStateJson);
}

void notifyClients() {
    ws.textAll(getStatusJSON());
}
```

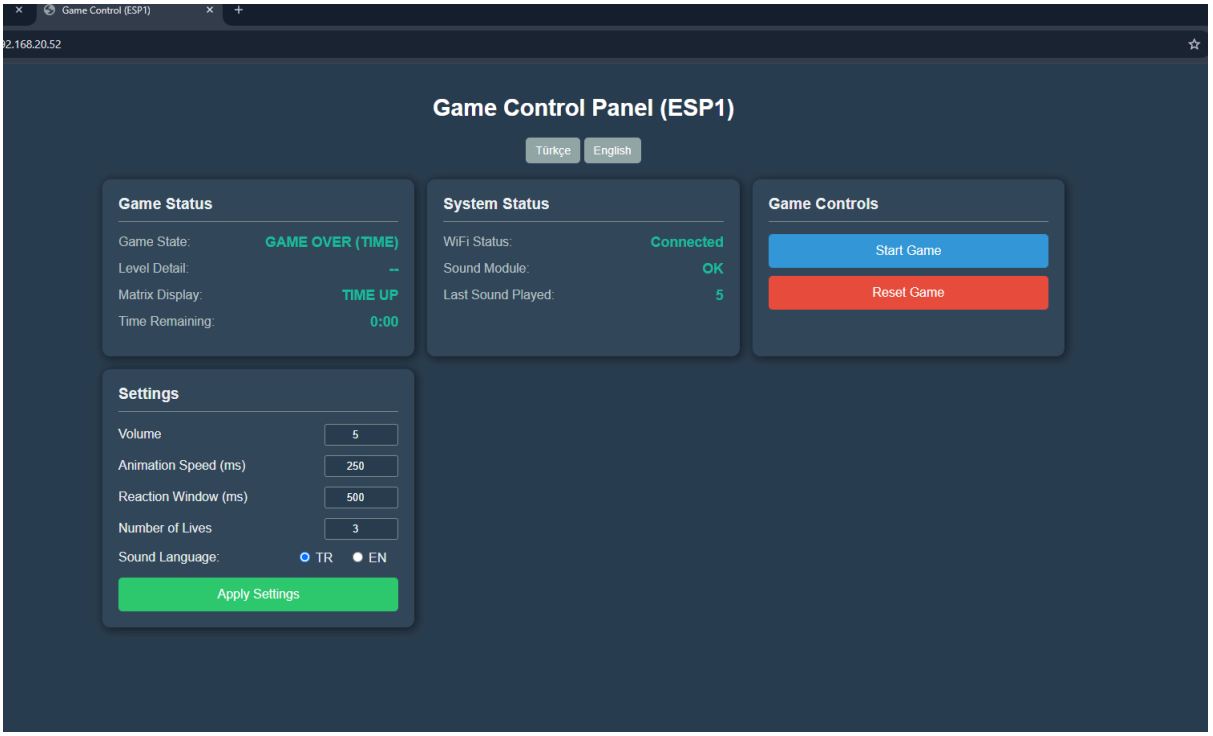
This approach is highly efficient. Instead of sending multiple small messages, the ESP32 compiles a complete "snapshot" of its current state into a single JSON object. The web interface can then parse this one object to update all its display elements simultaneously.

Frontend Development and UI Features:

The web interface was designed to be clean, intuitive, and comprehensive.

- **Layout:** The UI is organized into logical cards: "Game Status," "System Status," "Game Controls," and "Settings."
- **Real-Time Data Display:** Using JavaScript, the page listens for WebSocket `onmessage` events. When a new JSON status object is received, the script parses it and dynamically updates the inner HTML of the corresponding elements on the page, showing the live game state, remaining time, and more.
- **Interactive Controls:** Buttons for "Start Game" and "Reset Game" send simple JSON commands. The "Settings" panel allows the game master to change parameters like volume, animation speed, reaction time, and the number of lives. Clicking "Apply Settings" packages all these values into a JSON object and sends them to the ESP32 for processing.

- **Multilingual Support:** A key feature is the JavaScript-driven language switcher. With buttons for "Türkçe" and "English," the UI can dynamically change all text labels and titles without reloading the page by pulling the appropriate text from a `translation` object.



Conclusion:

This day's work resulted in a highly polished and professional control panel for the ESP1 game module. The use of WebSockets created a truly interactive and responsive user experience, far superior to basic web pages. This task successfully demonstrated skills in full-stack embedded development, from writing the C++ backend and WebSocket server on the ESP32 to designing a dynamic and user-friendly frontend with HTML, CSS, and modern JavaScript.

Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day28	Tarih:

Yapılan iş:**Developing the Hardware Diagnostic and Control Panel for ESP2**

With the core firmware for the "Hardware Puzzle" (ESP2) established, today's objective was to create a dedicated web interface for it. Unlike the UI for the main game (ESP1), which is player-focused, this panel was designed as a powerful diagnostic and engineering tool. Its purpose is to give the game master a detailed, real-time view of all mechanical and sensor states, and to provide manual overrides and fine-tuning capabilities.

Web Interface Architecture and Communication:

The architecture mirrors the successful implementation on ESP1, ensuring consistency and reliability.

- Self-Hosted Server: The `ESPAsyncWebServer` library was used to host the entire interface directly from the ESP2.
- Real-Time Data with WebSockets: A WebSocket connection (`/ws`) provides a persistent, low-latency communication channel. This allows the ESP32 to instantly "push" status updates to the browser whenever a sensor value changes or the motor's state is updated.
- JSON for Data Structuring: All communication is handled through structured JSON objects. This is especially important for ESP2 due to the large number of different data points (motor position, sensor readings, game states, etc.).

Backend Firmware (ESP2):

The firmware was designed to collect a comprehensive set of data points and make them available to the web UI.

- Data Aggregation (`getStatusJSON`): This function was developed to be the single source of truth for the UI. It gathers all relevant data—the current game mode, RFID status, motor position and speed, microphone status, live voltmeter voltage, and all configurable settings—and packages them into a single, well-structured JSON object. This object is then broadcast to all connected WebSocket clients.
- Command and Settings Handler: The `handleWebSocketMessage` and `applySettings` functions were written to parse incoming JSON messages from the web UI. This allows the game master to:
    - Manually Control the Motor: Send commands to move the stepper motor by a specific number of steps.
    - Start the Game Sequence: Remotely trigger the homing and game sequence.

- Apply Settings in Real-Time: Adjust critical parameters like motor speed, acceleration, and voltmeter needle speed without needing to reprogram the ESP32.

```
String getStatusJSON() {
    JSONVar status;

    // Main game state
    status["gameMode"] = getGameModeString();
    JSONVar hardware;
    hardware["buttonState"] = (digitalRead(GAME1_BUTTON_PIN) == LOW) ? "Pressed" : "Released";
    hardware["micStatus"] = (digitalRead(DIGITAL_MIC_PIN) == HIGH) ? "ACTIVE (Sound)" : "Passive";
    hardware["rfidStatus"] = last_rfid_uid;
    hardware["ledBrightness"] = strip.getBrightness();
    hardware["liveVoltage"] = game1_targetVoltage;
    hardware["motorPosition"] = stepper.currentPosition();
    hardware["motorRunning"] = (stepper.distanceToGo() != 0);
    status["hardware"] = hardware;

    // Group all settings into another nested object
    JSONVar settings;
    settings["voltmeterSpeed"] = game1_selectionInterval;
    settings["stepperSpeed"] = stepper_max_speed;
    settings["stepperAccel"] = stepper_acceleration;

    status["settings"] = settings;

    return JSON.stringify(status);
}
```

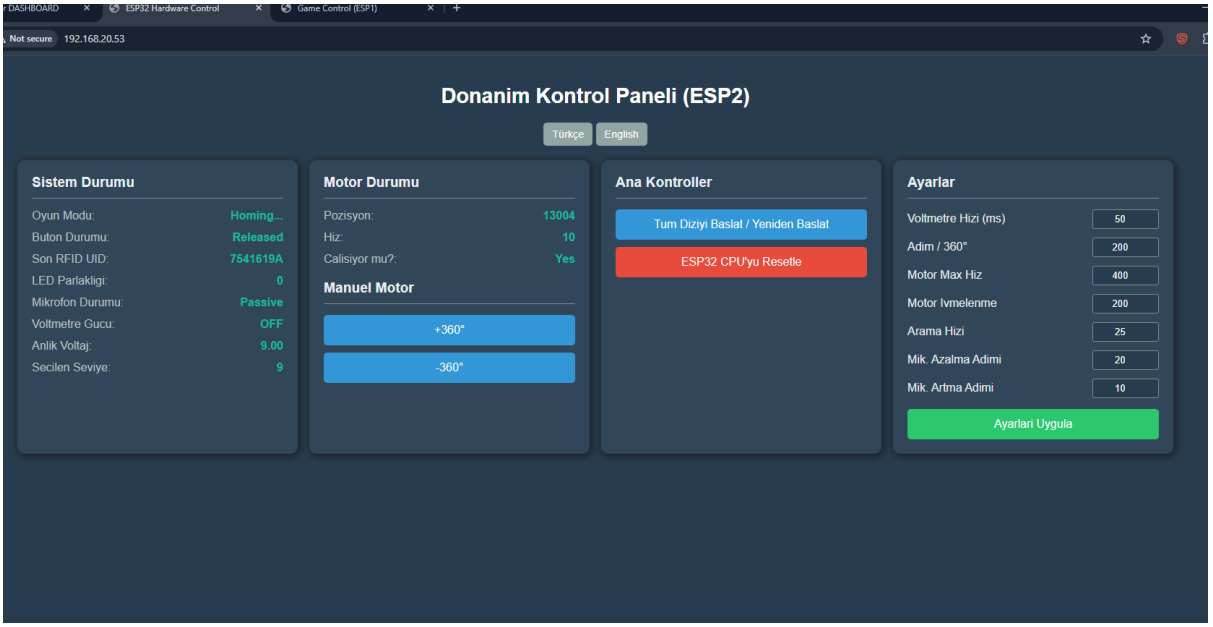
This JSON structure is more advanced than that of ESP1. By grouping data into nested `hardware` and `settings` objects, the code is highly organized. This allows the JavaScript on the frontend to easily access specific data points (e.g., `data.hardware.motorPosition` or `data.settings.stepperSpeed`) and update the UI with high precision. It's a robust method for managing a large number of variables.

Frontend "Hardware Control Panel":

The web interface for ESP2 was specifically designed for technical oversight and control.

- **Multi-Card Layout:** The UI is divided into clear sections: "System Status," "Motor Status," "Main Controls," and "Settings."

- **Live Data Monitoring:** Every field on the page, from "Game Mode" to "Motor Position" and "Mic Status," is updated in real-time by the WebSocket connection, providing an instant view of the hardware's behavior.
- **Manual Overrides:** The panel includes buttons to start the game sequence and manually move the motor, which was invaluable for testing the homing routine and game mechanics without needing to physically trigger the game's start condition.
- **Real-Time Tuning:** The "Settings" card provides sliders and input fields to fine-tune all major parameters of the game. Changes made here are sent to the ESP32 and applied immediately, allowing for rapid prototyping and balancing of the puzzle difficulty.



Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day29	Tarih:

Yapılan iş:**Node-RED Dashboard Foundation & Status Monitoring**

With both ESP32 modules programmed and functioning independently, today's work shifted to creating the central nervous system of the project: the "Game Master" dashboard in Node-RED. The objective for today was to lay the foundation by establishing a robust system for receiving, parsing, and displaying all real-time status data from both ESPs.

1. MQTT Architecture and Data Flow Design

The first step was to configure Node-RED to securely connect to the local MQTT broker. I then designed the data ingestion flow by creating `mqtt in` nodes to subscribe to the four essential status topics defined in the ESP firmware:

- **`game/status`**: Listens for the detailed JSON status from ESP1 (The "Deadlight" Game).
- **`esp32-gamemaster/status`**: Listens for the detailed JSON status from ESP2 (The "Hardware Puzzle").
- **`esp/esp1/connection`**: Monitors the connection status of ESP1 using MQTT's Last Will and Testament (LWT) feature.
- **`esp/esp2/connection`**: Monitors the connection status of ESP2.

2. Advanced Data Parsing with `function` Nodes

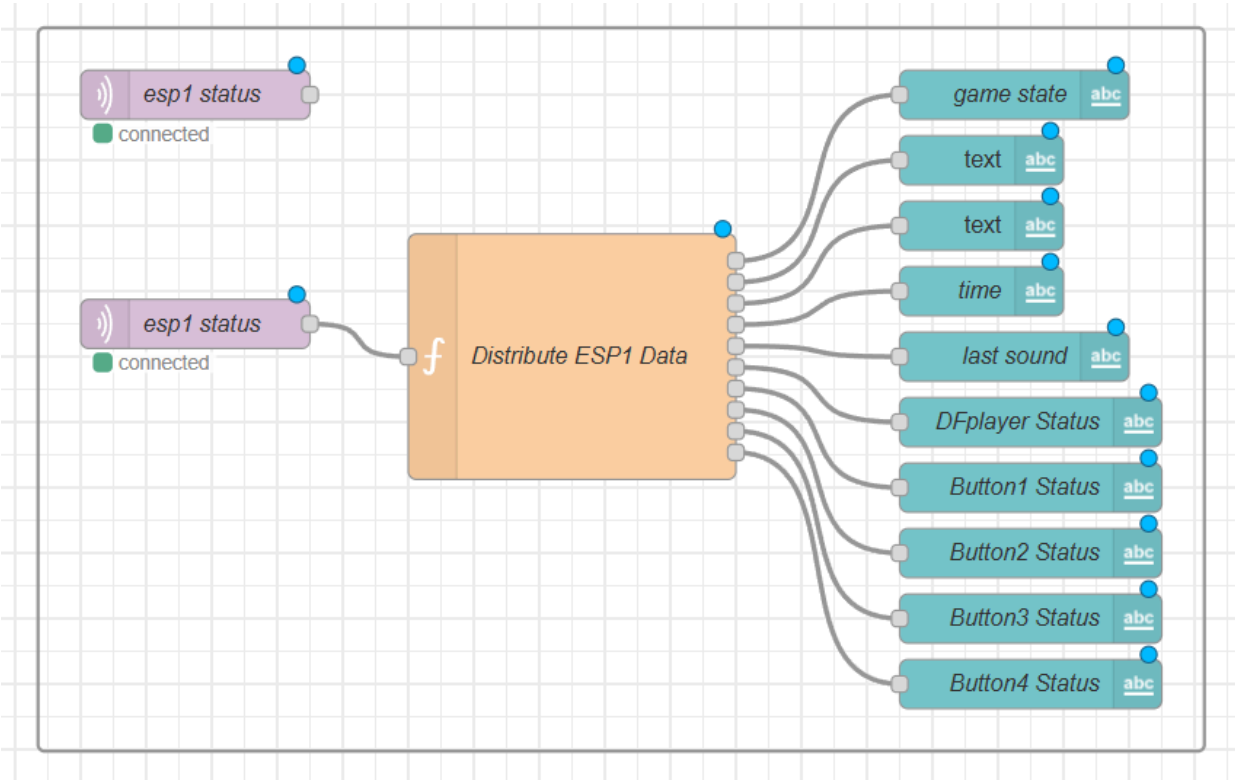
A key architectural choice was to use a single `function` node for each ESP to parse its entire status JSON. This node acts as a central distributor, receiving one complex message and outputting many simple ones. This design keeps the overall flow clean and highly modular. If a new status variable is added to an ESP in the future, only this single function node needs to be updated.

```json
{
  "gameMode": "Awaiting RFID Scan",
  "hardware": {
    "buttonState": "Released",
    "micStatus": "Passive",
    "rfidStatus": "None",
    "ledBrightness": 0,
    "motorPosition": 13004,
    "motorRunning": false
  },
  "settings": {
    "voltmeterSpeed": 50,
    "stepperSpeed": 400
  }
}
```

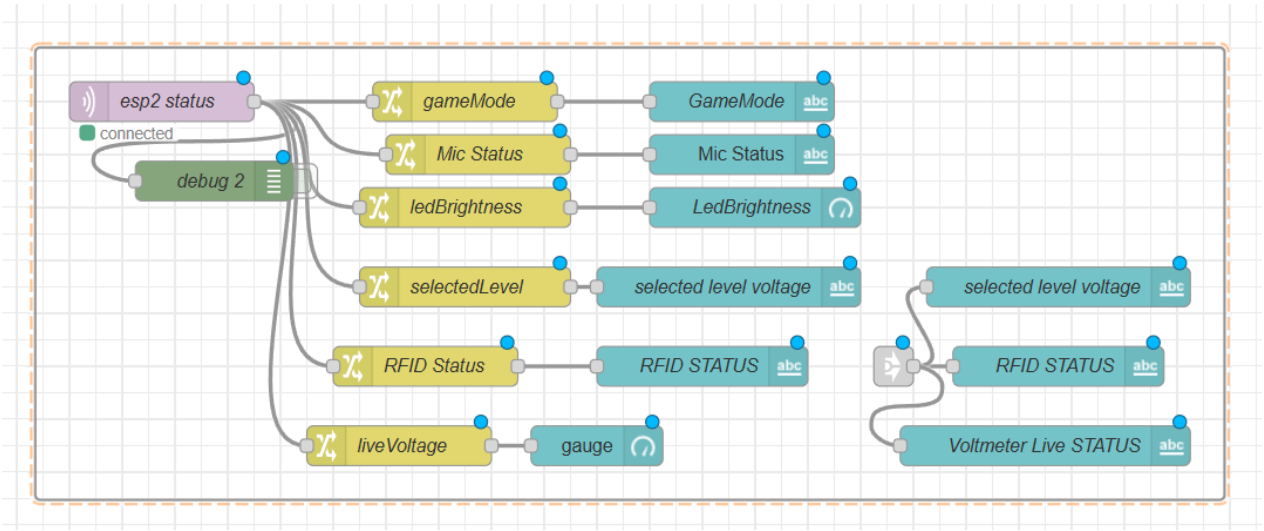
This JSON payload demonstrates an organized data structure. By nesting related values inside `hardware` and `settings` objects, the data is easy to manage. The Node-RED `function` node can access specific values using dot notation, such as `data.hardware.motorPosition`.

```
const data = JSON.parse(msg.payload);
const messages = new Array(8);
if (data.hardware) {
    messages[0] = { payload: data.gameMode };
    messages[1] = { payload: data.hardware.micStatus };
    messages[2] = { payload: data.hardware.ledBrightness };
    messages[3] = { payload: data.hardware.rfidStatus };
    // ... and so on for other hardware statuses
}

// Return the array of messages.
return messages;
```

This function acts as a smart, central distributor. It safely unpacks the complex JSON object and routes each individual piece of information to the correct part of the UI. This approach is highly efficient and scalable for managing a large number of status variables.



The Node-RED flow for distributing status data from ESP1. A single MQTT input is parsed and routed to multiple dashboard elements.



The corresponding data distribution flow for the more complex hardware data coming from ESP2.

Conclusion for the Day: By the end of the day, the monitoring foundation of the dashboard was complete. Node-RED was successfully receiving, parsing, and preparing all real-time status data from both complex ESP32 modules. The next step is to build the interactive controls to send commands back to the devices.

Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day30	Tarih:

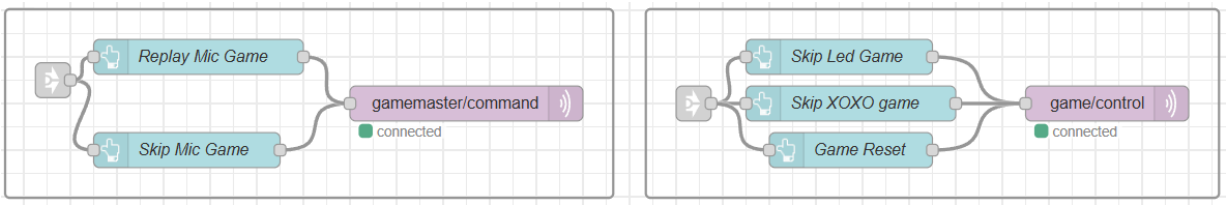
Yapılan iş: **Implementing Game Controls & Finalizing the Dashboard**

With the status monitoring systems in place, today's objective was to build the interactive control part of the Node-RED dashboard and finalize the unified user interface. This would complete the bidirectional communication loop, giving the game master full command over the entire escape room experience.

1. "Game Master" Command Implementation

I added a new section to the dashboard for high-level game master controls. This involved using `ui_button` nodes for critical actions like "Game Reset," "Skip LED Game," and "Skip Mic Game." The logic for these controls was designed to be simple and robust:

- A `ui_button` node is clicked on the dashboard.
- The button sends a predefined string payload (e.g., `"skip_led_game"`).
- A `function` node wraps this string into the required JSON command format (e.g., `{"action":"skip_led_game"}`).
- An `mqtt out` node publishes this JSON payload to the appropriate command topic for either ESP1 (`game/control`) or ESP2 (`esp32-gamemaster/command`).

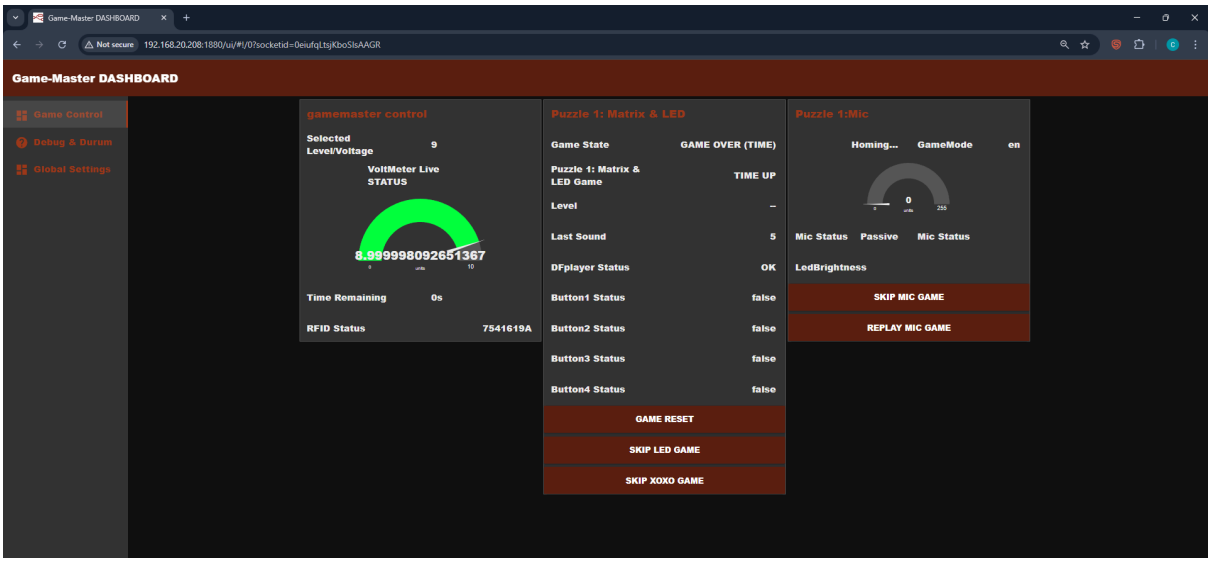


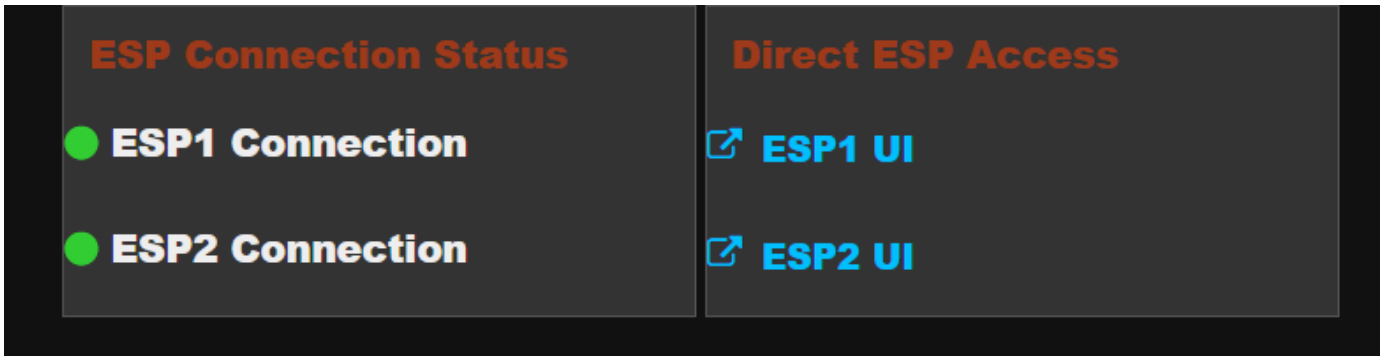
2. The Final Unified Dashboard

The work from the past two days culminated in the final, unified "Game-Master DASHBOARD." This dashboard successfully integrates all aspects of the two-part escape room game into a single, cohesive interface.

- **Live Status Monitoring:** The panel provides a real-time view of every crucial variable, from the game state on ESP1 to the live voltmeter value and motor position on ESP2.
- **Centralized Control:** The game master has full authority to start, stop, reset, or bypass any part of the game sequence for both puzzles, which is essential for managing the player experience.

- **Connection Health:** A dedicated panel, driven by the MQTT LWT messages, provides an instant visual indicator (a red or green dot) of whether each ESP32 is online and connected to the network. This is critical for quick troubleshooting during a live game.





Conclusion for the Day: The completion of the Node-RED dashboard marks a major project milestone. The system now possesses a robust, centralized, and real-time interface for both monitoring and management. This demonstrates a key skill in modern IoT: using platform tools like Node-RED to rapidly build powerful applications that can elegantly control and interact with complex, multi-device embedded systems. The project is now functionally complete, with the final day reserved for the implementation of an additional hint system.

Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day31	Tarih:

Yapılan iş:**Centralizing Language Control via the Node-RED Dashboard**

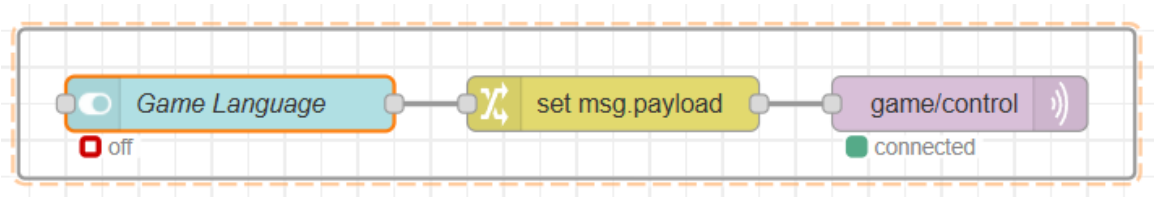
With the individual web interfaces for both ESP modules complete, today's goal was to centralize all language settings into the main "Game Master" dashboard on Node-RED. This involved implementing two distinct features:

1. Remotely controlling the **spoken language** of the audio cues played by ESP1.
2. Dynamically changing the **display language** of the Node-RED dashboard itself.

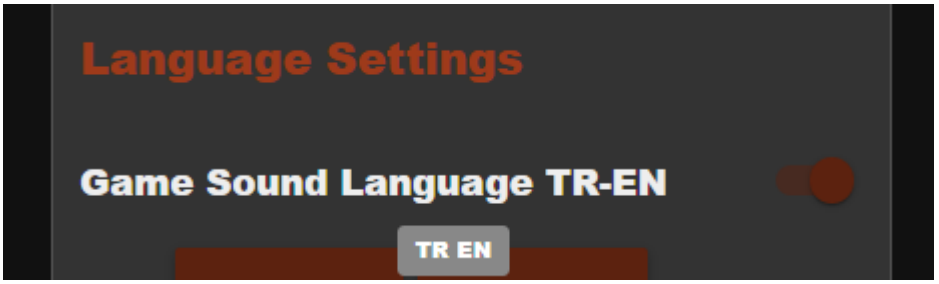
1. Remote Control of Game Audio Language

To allow the game master to change the language of the in-game voiceovers (played by ESP1's DFPlayer Mini), I created a control flow in Node-RED that sends a command via MQTT.

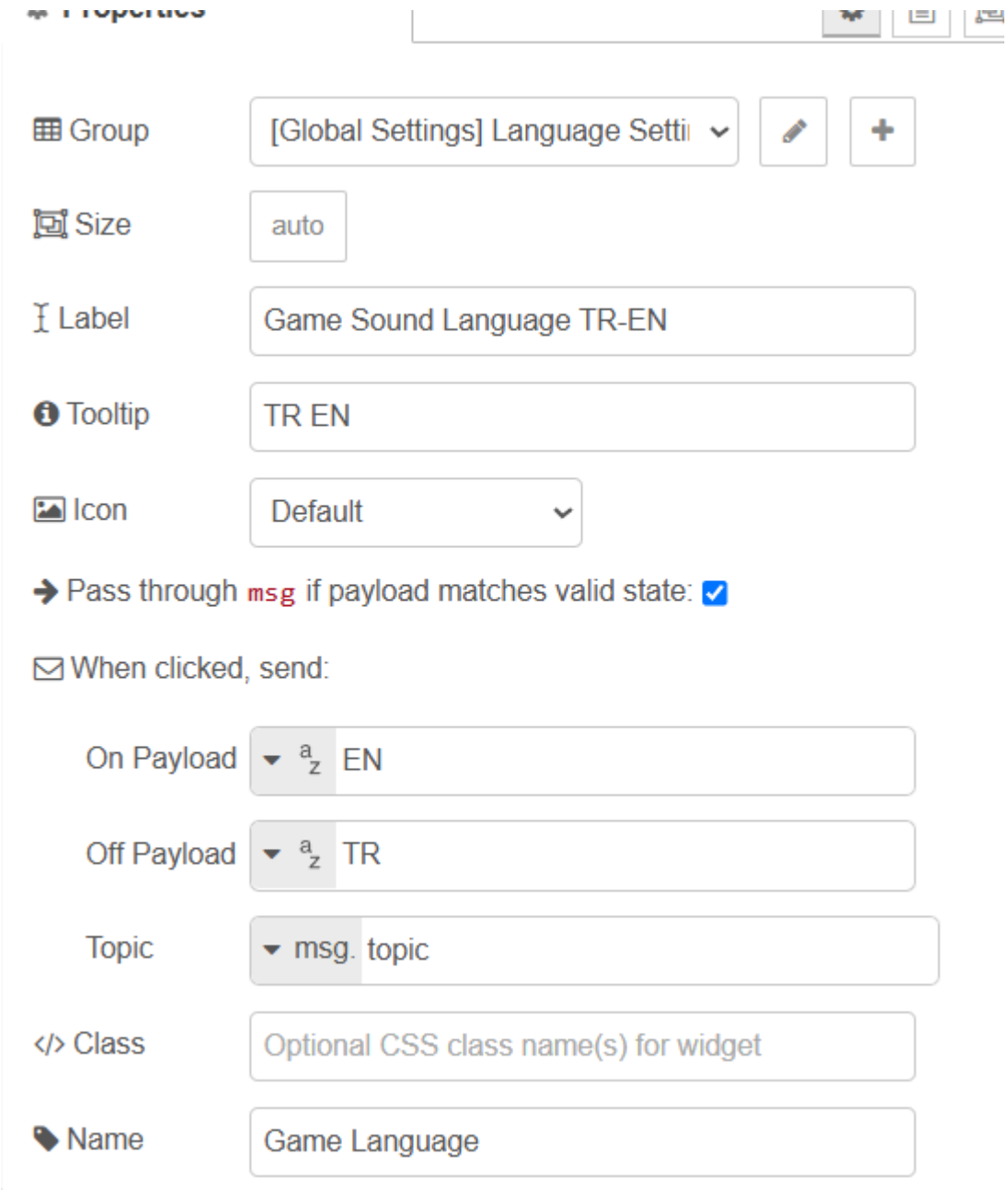
- **UI Control:** A `ui_switch` node was added to the dashboard, labeled "Game Sound Language TR-EN". This switch is configured to send a payload of `"EN"` when in the "On" state and `"TR"` when in the "Off" state.



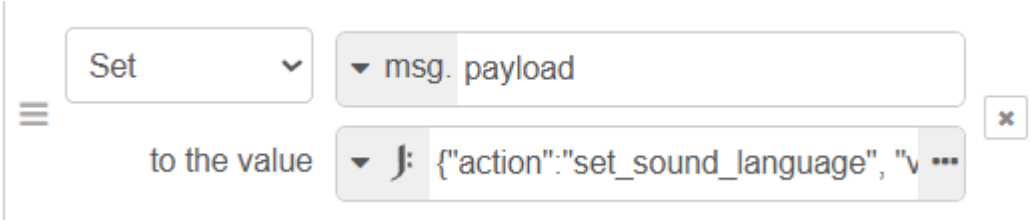
- **Command Formatting:** The output of this switch is passed to a `change` node. This node formats the payload into a proper JSON command string that the ESP1 can understand. For example, if "EN" is selected, it creates the message: `{"action":"set_sound_language", "value":"EN"}`.
- **MQTT Publishing:** Finally, this JSON command is sent to an `mqtt out` node, which publishes it to the `game/control` topic. The ESP1 firmware, which is always listening to this topic, receives the command, updates its `soundLanguage` variable, and from that point on, plays all audio from the English sound folder.



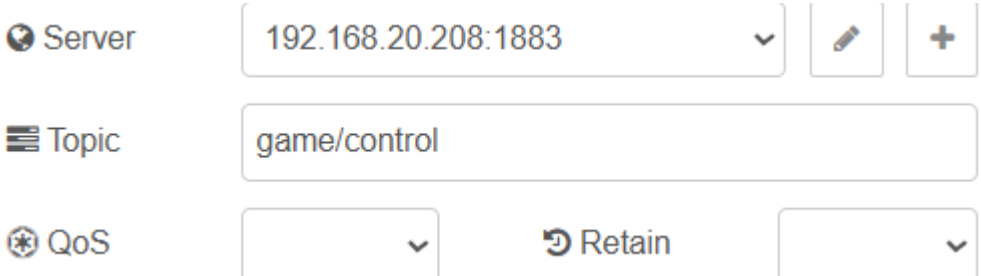
The language switch for game audio on the Node-RED dashboard.



The configuration of the `ui_switch` node, defining the "EN" and "TR" payloads.



The `change` node used to format the MQTT command payload.



The `mqtt out` node configured to send the command to the `game/control` topic.

2. Dynamic Translation of the Node-RED Dashboard

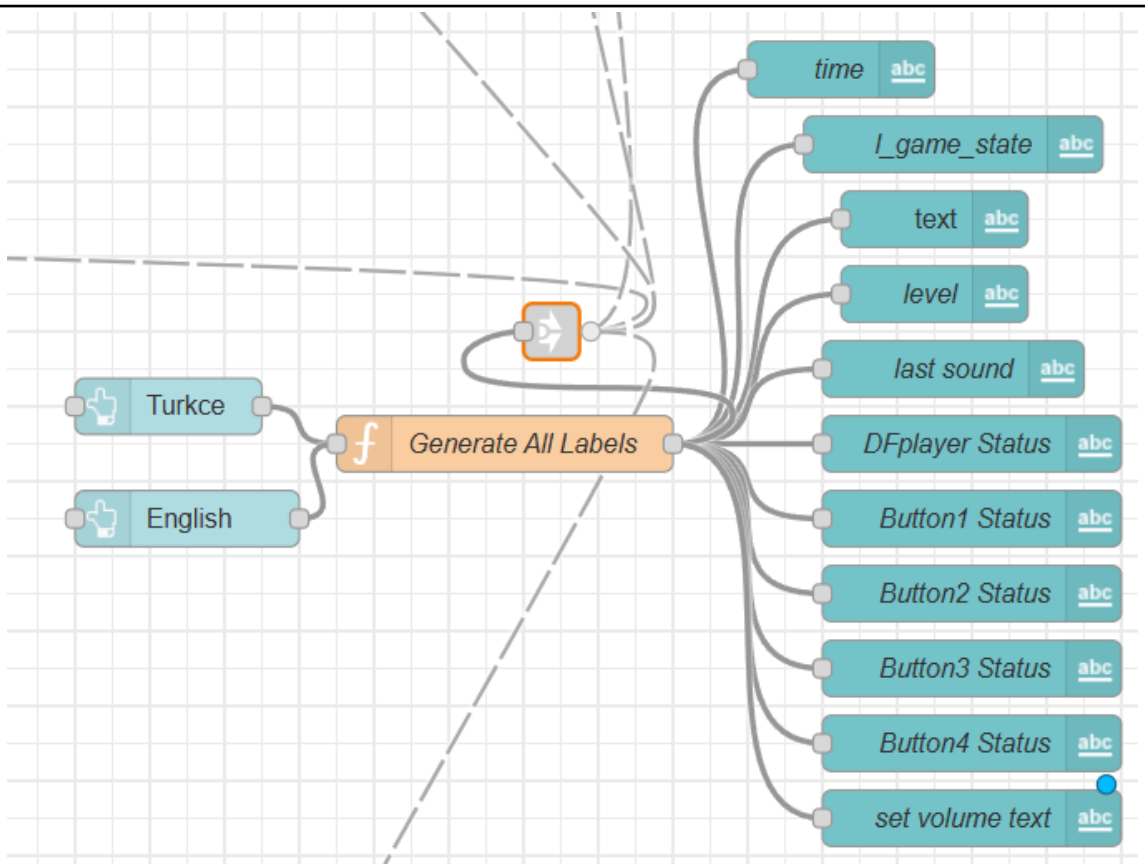
To make the dashboard itself multilingual, a more advanced, internal Node-RED technique was used.

- **UI and Flow:** Two `ui_button` nodes, labeled "Türkçe" and "English," were added. These buttons do not send MQTT messages. Instead, they trigger a central `function` node named "Generate All Labels".
- **Central Translation Logic:** This `function` node contains a large JavaScript object called `translations`, which holds all the UI text (button labels, titles, status descriptions) for both languages. When a language button is pressed, the function receives the language code (`tr` or `en`), selects the corresponding set of translations, and attaches each translated string as a property to the output message (e.g., `msg.l_game_state = "Oyun Durumu"`).
- **Dynamic Update:** The multiple outputs of this function node are wired directly to the "label" field of every text, button, and group element on the dashboard. This allows a single button click to

instantly update the entire user interface to the selected language without any delay or page refresh.

```
// A large dictionary containing all text labels for both languages
const translations = {
    en: {
        g_master_control: "Master Control",
        g_puzzle1: "Puzzle 1: Matrix & LED Game",
        l_game_state: "Game State",
        b_game_reset: "Game Reset",
        // ... and all other English labels
    },
    tr: {
        g_master_control: "Ana Kontrol",
        g_puzzle1: "Puzzle 1: Matrix & LED Oyunu",
        l_game_state: "Oyun Durumu",
        b_game_reset: "Oyunu Sıfırla",
        // ... ve diğer tüm Türkçe etiketler
    }
};

// Get the chosen language ('tr' or 'en') from the input message
const lang = msg.payload;

// Select the correct set of translations, defaulting to Turkish if there's an error
const labels = translations[lang] || translations.tr;

msg.l_game_state = labels.l_game_state;
msg.b_game_reset = labels.b_game_reset;
// ... and so on for every single label on the dashboard

return msg;
```

This centralized approach is extremely efficient for managing UI translations. All text is kept in one location, making it easy to add new languages or edit existing strings. The function acts as a powerful "translation engine" within the Node-RED flow itself, dynamically rebuilding the UI text based on the operator's selection.



Conclusion: Today's work centralized all language settings into the master dashboard, demonstrating an advanced use of Node-RED for both remote device configuration (sending MQTT commands) and dynamic, server-side UI generation. This makes the entire system more professional and operator-friendly.

Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

Bölüm:day32	Tarih:

Yapılan iş:**Implementing the Centralized, Bilingual Audio Hint System**

The final day of the project was dedicated to implementing a crucial feature for any live escape room: a flexible and robust audio hint system. The objective was to allow the game master to play helpful audio hints for the players directly from the Node-RED dashboard. To ensure high-quality, uninterrupted audio and to avoid adding any processing load to the game's ESP32s, the system was architected to handle all audio playback directly on the Raspberry Pi that hosts the Node-RED instance.

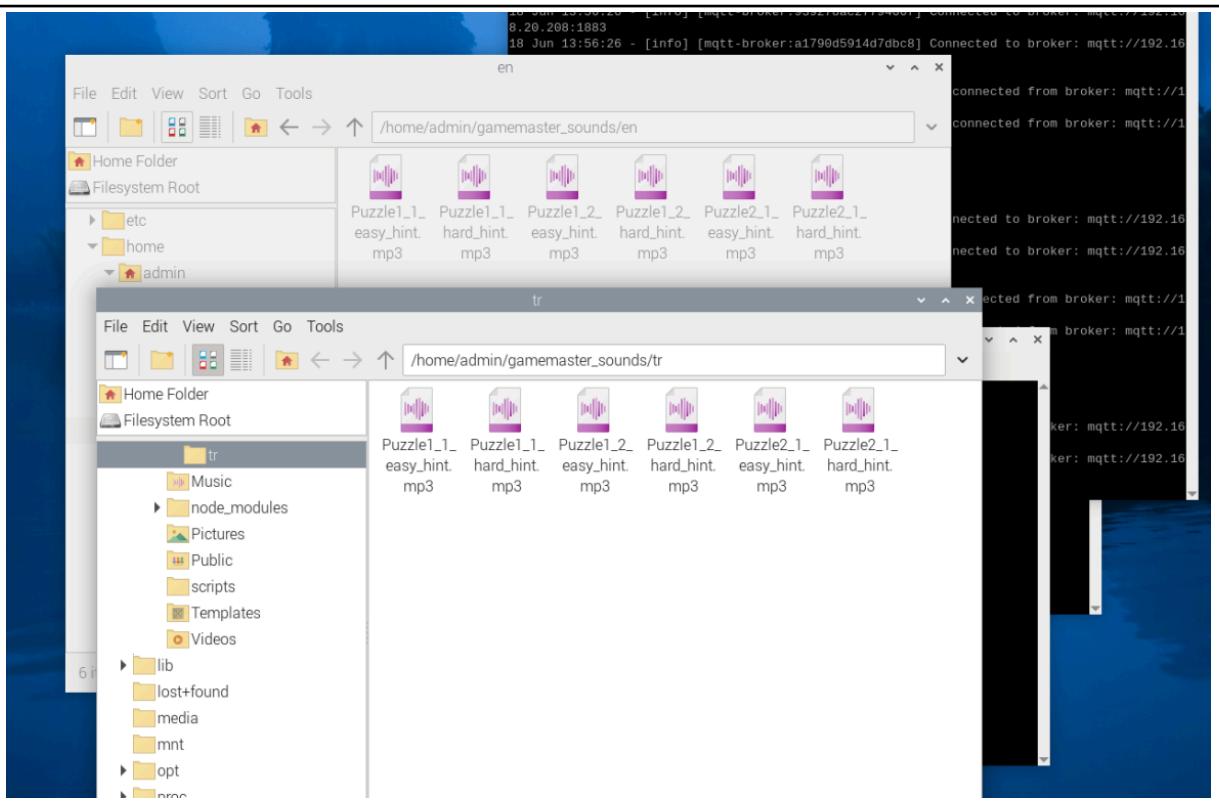
System Architecture and Design:

The hint system was designed based on four core principles:

1. **Centralized Audio Files:** All hint `.mp3` files are stored on the Raspberry Pi's filesystem. This makes managing, adding, or changing hints incredibly easy, as it only requires uploading a file to a folder, with no need to reprogram any microcontrollers.
2. **Bilingual Support:** The system fully supports both Turkish and English hints. This is achieved by organizing the audio files into language-specific subdirectories.
3. **Non-Overlapping Playback:** A Python script utilizing the `mpg123` command-line player ensures that if a new hint is triggered while another is playing, the old one is immediately stopped before the new one begins. This prevents confusing audio clashes.
4. **ESP-Independent Logic:** The entire hint system operates independently of the ESP32 modules. The ESPs are completely unaware that hints are being played, as the process is managed entirely between the Node-RED dashboard and the Raspberry Pi's operating system.

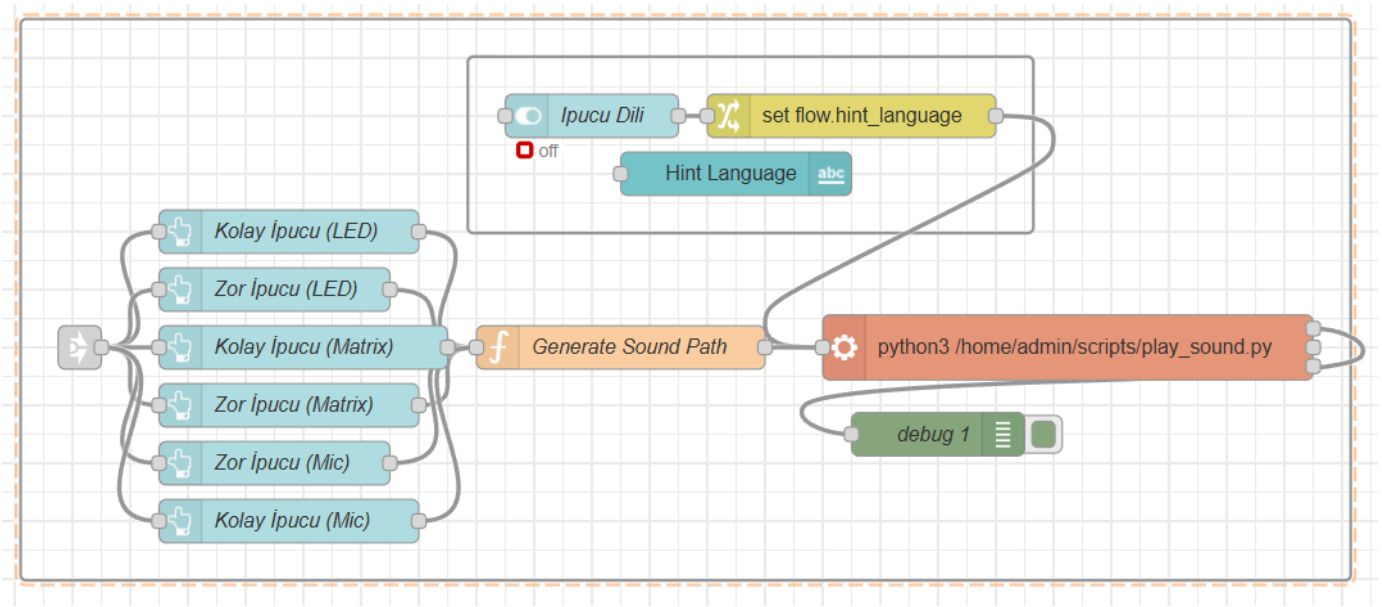
Implementation on Raspberry Pi and Node-RED:

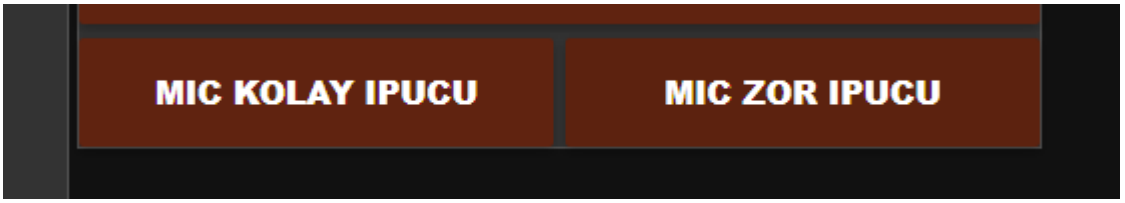
- **Raspberry Pi File Structure:** A main directory, `/home/admin/gamemaster_sounds/`, was created. Inside, two subdirectories, `tr` and `en`, were made to hold the Turkish and English audio files, respectively. The filenames are kept consistent across languages (e.g., `Puzzle1_1_easy_hint.mp3`).

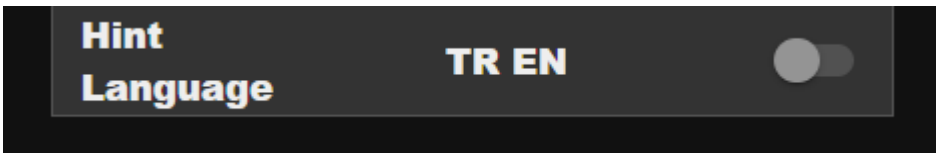


Node-RED Hint Flow: A dedicated flow was created in Node-RED to manage the hint logic.

- UI Controls: `ui_button` nodes were created for each specific hint (e.g., "LED EASY HINT," "MIC HARD HINT"). A `ui_switch` node was also added to allow the game master to toggle the hint language between Turkish and English.
- Path Generation & Execution: When a hint button is pressed, its payload (the filename) is sent to a `function` node. This node dynamically builds the full file path based on the selected language stored in a flow variable. The complete path is then passed to an `exec` node, which runs a Python script to play the sound.

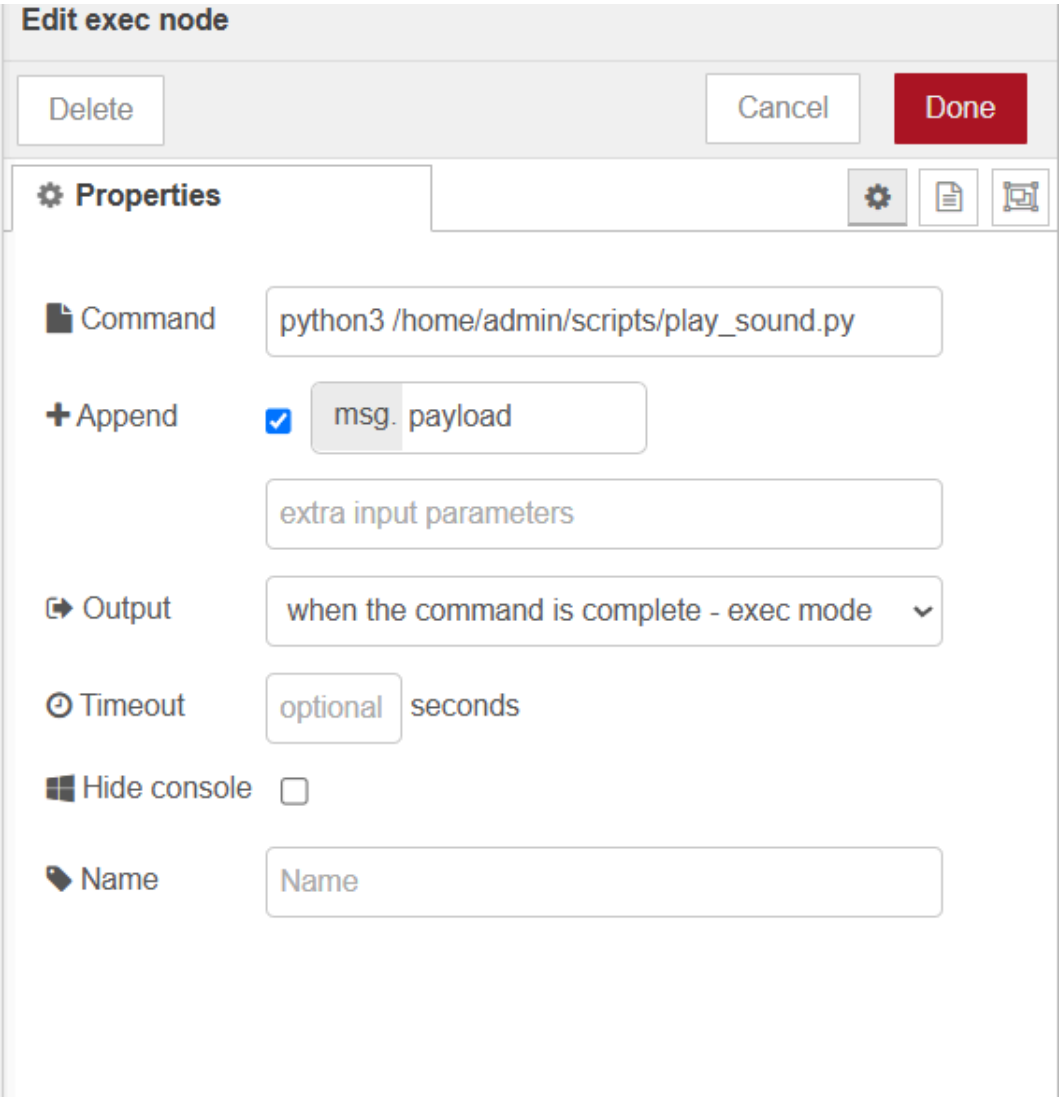






Node-RED "Generate Sound Path" Function This JavaScript code runs inside the central `function` node. It intelligently constructs the command that the `exec` node will run.

```
const fileName = msg.payload;
const lang = flow.get("hint_language") || "tr";
const volume = flow.get("hint_volume") || 80;
const basePath = "/home/admin/gamemaster_sounds/";
const fullPath = basePath + lang + "/" + fileName;
msg.payload = `"${fullPath}" ${volume}`;
return msg;
```



This function is the "brain" of the hint system. It demonstrates a key Node-RED concept: using flow variables (`flow.get()`) to maintain state (like the selected language). It dynamically constructs a shell command argument string, which provides the Python script with all the information it needs to play the correct file at the correct volume.

Project Conclusion: The implementation of this centralized audio hint system marks the successful completion of the entire 12-day final project. This last feature perfectly illustrates the power of a component-based, networked architecture. By combining different technologies—real-time embedded systems on the ESP32s, a flexible message bus with MQTT, a powerful visual programming and UI tool with Node-RED, and system-level scripting with Python—a truly robust, scalable, and professional-grade interactive experience was created. The final system fulfills all initial objectives and serves as a comprehensive demonstration of the skills and IoT principles acquired throughout the internship.	
Stajyer İmzası	İşyeri Bölüm Sorumlusunun İmzası

AÇIKLAMALAR

- Staj defteri her iş gününe bir sayfa olacak şekilde 30 işgünü için ayarlanmıştır. İhtiyaç halinde sayfa sayısı artırılabilir.
- Staj Türü kısmına Yazılım/Donamım/Diğer(Belirtin) ifadelerini girmelisiniz.
- Yapılan Staj Türü ve Yapılan Staj kısımlarına Staj I ya da Staj II seçeneklerinden birisini girmelisiniz.
- Bu sayfa bilgilendirme amaçlıdır, çıktıya eklemeyin!
- Defterin okunabilirliğini kolaylaştırmak adına lütfen defteri ciltletin ya da spiralletin.