**Agile-Scrum with phased hardware development**

- to ensure iterative progress, testing, and refinements

**Project Development Lifecycle**

1. Project Planning & Requirement Analysis (Weeks 1-2)

🔹 Define core objectives → What problems does the development kit solve?

🔹 List MCU, sensors, and communication protocols

🔹 Determine software stack (C++, Python, Web, APIs, GUI)

🔹 Define project scope, constraints & budget

🔹 Create initial roadmap

2. Hardware Architecture & PCB Design (Weeks 3-6)

🔹 Finalize schematic designs for ESP32, STM32, ATmega, and PIC integration

🔹 Map sensor and actuator connections

🔹 Design multi-layer PCB ensuring power efficiency & signal integrity

🔹 Prototype initial PCB with basic components

🔹 Test voltage, GPIO functionality, and communications (UART, I2C, SPI)

3. Firmware Development & MCU Programming (Weeks 7-10)

🔹 Write modular drivers for sensors, motors, displays, communication modules

🔹 Develop hardware abstraction layers (HAL) for cross-platform compatibility

🔹 Integrate multitasking using FreeRTOS for ESP32/STM32

🔹 Implement data logging & debugging interfaces

🔹 Test peripheral functionality with unit tests & debugging tools

4. Web-Based GUI & API Development (Weeks 11-14)

🔹 Design Node-RED-like drag-and-drop interface

🔹 Develop RESTful APIs for remote programming and monitoring

🔹 Implement secure MQTT/WebSockets for real-time data streaming

🔹 Optimize UI/UX for seamless program selection & execution

🔹 Deploy test version for internal validation

5. Prototype Testing & Iteration (Weeks 15-18)

🔹 Test complete hardware & software stack together

🔹 Validate sensor readings, actuator control, and wireless communication

🔹 \*\*Measure power consumption, response time, and latency

🔹 Optimize firmware for performance & security enhancements

🔹 Iterate based on results & improve system robustness

6. Final Product & Documentation (Weeks 19-22)

🔹 Create full documentation for hardware, firmware, and web systems

🔹 Prepare learning guides, tutorials, and project samples

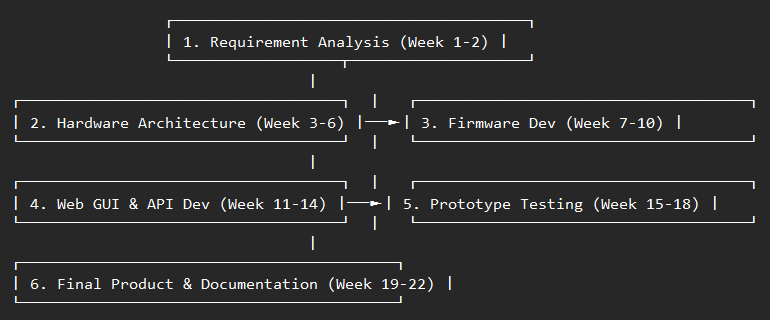
🔹 Manufacture final PCB version & refine case/enclosure design

🔹 Deploy software & firmware for production use

🔹 Release online community & educational resources

Development Flow Diagram

Here’s a visual Agile-Scrum workflow that fits your hardware-software hybrid development:



**Refined Project Development Lifecycle**

1. Project Planning & Requirement Analysis (Weeks 1-2)

🔹 Define objectives → MCU compatibility, communication protocols, automation functions.

🔹 List all hardware & software components → ESP32, STM32, ATmega, GSM800L, CAN/LIN, Zigbee, LoRaWAN.

🔹 Set up GitHub repositories for version control & collaborative development.

🔹 Draft system architecture, data flow models, and initial firmware specifications.

2. Hardware Architecture & PCB Design (Weeks 3-6)

🔹 MCU Selection & Pin Mapping - Assign dedicated I/O pins for sensors, motors, displays, relays, and communication modules. - Map STM32 CAN/LIN interfaces, ensuring proper signal integrity. 🔹 Power Circuit Optimization - Implement voltage regulation, battery management, and efficiency calculations. - Ensure robust PCB layout for minimal interference and EMI shielding. 🔹 Peripheral Integration - Connect GSM800L for cellular communication. - Design LoRaWAN/Zigbee/Z-Wave interfaces for IoT. 🔹 First Prototype PCB Design & Fabrication - Validate initial board for MCU communication, power distribution & signal integrity.

3. Firmware Development & MCU Programming (Weeks 7-12)

Sub-Modules for Specific Hardware Components:

✅ Sensor Interface Module

- Develop drivers for POT, LDR, temperature sensors, ACS712 current sensor.

- Optimize ADC/DAC readings & calibration algorithms.

✅ Stepper Motor & Servo Control

- Implement DRV8825 stepper driver logic with precise step control & acceleration profiles.

- Develop servo PWM control routines.

✅ Display Interfaces

- Write drivers for 16x2 LCD, OLED, and future TFT touch screens.

- Implement menu-based visualization & dynamic refresh rate optimizations.

✅ Wireless Communication Stack

- Implement LoRaWAN, Zigbee, BLE, RF data protocols for IoT.

- Develop MQTT integration & WebSockets support for real-time interaction.

✅ GSM & Remote Connectivity

- Integrate SIM800L GSM control API for SMS & remote communication.

- Enable HTTP-based control for cloud sync capabilities.

✅ CAN/LIN Communication

- Develop STM32 CAN bus interfaces for automotive-grade control.

- Ensure modular API for expanding industrial automation compatibility.

✅ Storage & File Handling

- Optimize SD card read/write operations.

- Store log files for debugging and historical data.

✅ Real-Time Scheduling & Multitasking

- Implement FreeRTOS or multi-threading for ESP32 & STM32.

- Ensure efficient task execution, avoiding bottlenecks.

✅ Security & Access Control

- Develop firmware-level authentication for program execution.

- Implement AES encryption for secure IoT device communication.

4. Web-Based GUI & API Development (Weeks 13-16)

✅ Node-RED-Like Programming Interface

- Build a drag-and-drop logic framework for flow-based automation.

- Allow visual programming for selecting hardware functions dynamically.

✅ IoT Dashboard for Real-Time Monitoring

- Create a web-based GUI displaying sensor values, motor status, and automation events.

- Implement graphing & cloud logging features.

✅ Firmware Over-the-Air (OTA) Updates

- Enable wireless firmware updates via cloud API.

- Implement secure boot mechanisms for MCU updates.

✅ User Authentication & Role-Based Access

- Add account management for selecting preloaded projects.

- Ensure role-based permissions (admin, student, guest, etc.).

✅ Embedded Edge AI Features (Future Expansion)

- Introduce basic machine learning functions for ESP32.

- Process camera-based object recognition using ESP32-CAM.

5. Prototype Testing & Iteration (Weeks 17-20)

✅ Hardware Debugging

- Validate sensor readings, motor accuracy, wireless connectivity.

- Fix power optimization issues & EMI concerns.

✅ Software & Firmware Testing

- Perform unit tests, integration tests & stress tests.

- Implement error handling & watchdog mechanisms.

✅ Performance Optimization

- Profile execution time, latency, and power consumption.

- Optimize code efficiency for minimal delay & improved real-time responsiveness.

✅ Beta Release for Field Testing

- Deploy test units to verify real-world usage scenarios.

- Gather feedback for usability improvements.

6. Final Product Release & Documentation (Weeks 21-24)

✅ Comprehensive Documentation

- Create hardware setup guides, firmware development tutorials, and API references.

- Provide educational materials for beginners & advanced users.

✅ Mass Production & Deployment

- Fabricate final PCB batch with optimized firmware.

- Expand hardware capabilities based on testing outcomes.

✅ Community & Open-Source Development

- Set up GitHub repository for community contributions.

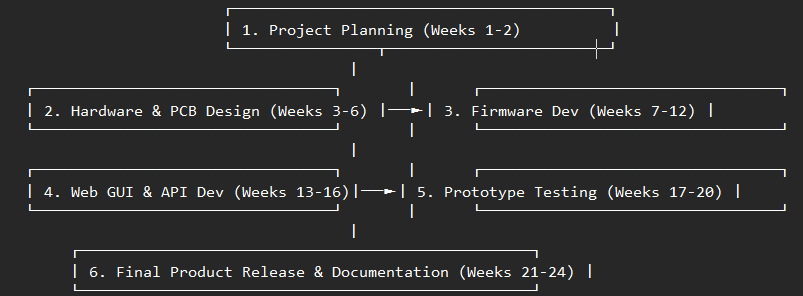
- Develop a user forum for troubleshooting and feature expansion.

✅ Marketing & Industry Collaboration

- Reach out to academic institutions, IoT companies, and automation industries.

- Publish research papers or case studies showcasing system capabilities.

**Final Optimized Flow Diagram**



Project Architecture

1. System Components

✔ MCUs & Processing Units

- ESP32

- STM32

- ATmega328

- PIC12F

✔ Sensors & Peripherals

- POT, LDR, Temperature Sensors

- Stepper Motors, Servo Motors, DRV8825

- ACS712 Current Sensor

- 16x2 LCD, OLED, TFT Display

✔ Communication Modules

- GSM800L

- LoRaWAN, Zigbee, BLE, RF

- CAN/LIN for automotive/industrial control

- WebSockets & MQTT for IoT connectivity

✔ Software Stack

- Firmware: C/C++ (ESP-IDF, STM HAL, FreeRTOS)

- Web GUI: React.js, Node.js, WebSockets

- API Services: RESTful API, MQTT, Firebase

✔ Cloud & Data Storage

- SD Card Logging

- IoT Cloud Connectivity

System Architecture

Core Subsystems

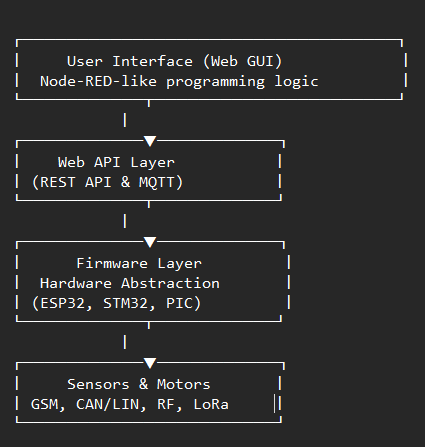
🔹 Hardware Abstraction Layer (HAL) → Ensures universal firmware compatibility across ESP32, STM32, ATmega, and PIC.

🔹 Task Scheduler → Manages real-time execution of firmware (FreeRTOS or event-driven model).

🔹 Communication Stack → Includes Wi-Fi, BLE, LoRaWAN, MQTT, WebSockets, CAN/LIN for seamless peripheral interfacing.

🔹 Data Processing Engine → Captures sensor values, processes them for automation and decision-making.

🔹 User Interface & Web API → GUI for drag-and-drop programming, project selection, and device configuration.



Data Flow Model

1. Sensor Data Processing

🔹 Analog/Digital Inputs → Read sensor values (POT, LDR, Temp Sensor, ACS712).

🔹 Data Conversion → ADC readings processed into usable formats.

🔹 Event Triggers → If temperature exceeds threshold, activate fan/relay.

🔹 Web Transmission → Publish data to API for live monitoring.

2. Actuator Control Flow

🔹 User Input via GUI → Select stepper motor or relay control logic.

🔹 Firmware Execution → Sends movement parameters to driver circuits.

🔹 Response Handling → Sends feedback to UI (motor position reached).

🔹 Log Data → Store event activity in SD Card or cloud database.

Initial Firmware Specifications

Core Features

✔ Multi-MCU Support → ESP32, STM32, ATmega, PIC via HAL.

✔ Sensor Integration → I2C, SPI, UART peripheral drivers.

✔ Motor Control → Stepper driver interface, PWM servo operation.

✔ Web API Communication → WebSockets & MQTT messaging.

✔ Event-Based Processing → FreeRTOS task scheduling for real-time automation.

✔ Security Features → AES-encrypted data transmission.

✔ OTA Update Support → Remote firmware updates via cloud.

Code Framework

void setup() {

initSensors();

setupCommunication();

configureUI();

}

void loop() {

readSensorData();

processCommands();

updateMotors();

sendDataToCloud();

}

**📖 Technical Documentation Structure**

**Each guide should follow a logical progression, from introduction to advanced concepts.**

**1️⃣ Introduction**

**✔ Overview of Development Kit**

**- Explanation of supported MCUs & peripherals.**

**- Hardware capabilities (ESP32, STM32, ATmega, PIC).**

**- Key features (wireless protocols, motor control, sensors, web API).**

**✔ System Architecture**

**- Block diagrams showing hardware interactions.**

**- Explanation of data flow between components.**

**- How Node-RED-like visual programming integrates with firmware.**

**2️⃣ Hardware Setup & Wiring Guide**

**✔ PCB Layout & Pin Mapping**

**- Step-by-step connection diagrams for each MCU.**

**- Power distribution strategies & voltage requirements.**

**- Peripheral wiring (sensors, motors, GSM, LoRa, Zigbee, CAN/LIN, etc.).**

**✔ Safety Guidelines**

**- Preventing short circuits & overheating.**

**- Best practices for grounding & EMI shielding.**

**3️⃣ Firmware Development Guide**

**✔ Programming Basics**

**- How to flash firmware on ESP32, STM32, ATmega, PIC.**

**- Configuring UART, I2C, SPI, PWM for peripherals.**

**- Using FreeRTOS for task scheduling & optimization.**

**✔ Stepper Motor Control**

**- Microstepping techniques (DRV8825).**

**- Writing PWM-based motion control algorithms.**

**✔ Wireless Communication (LoRa, BLE, Zigbee, MQTT, WebSockets)**

**- Setting up remote control via cloud API.**

**- Implementing secure encryption in data transmission.**

**✔ Firmware OTA Updates**

**- Step-by-step guide for updating firmware remotely.**

**- Debugging OTA issues & recovery mechanisms.**

**4️⃣ Web-Based GUI & API Integration**

**✔ Web Dashboard Usage**

**- Connecting the board to the browser interface.**

**- Selecting & running predefined automation programs.**

**✔ Node-RED-Like Drag-and-Drop Logic**

**- How users define automation workflows visually.**

**- Interfacing with sensors, relays, motors, and cloud services.**

**✔ REST API Reference**

**- Endpoints for controlling hardware remotely.**

**- Examples of JSON-based requests & responses.**

**5️⃣ Troubleshooting & FAQs**

**✔ Common Errors & Fixes**

**- Firmware crashes, sensor misreadings, connectivity issues.**

**- Step-by-step debugging methods.**

**✔ Hardware Debugging**

**- Diagnosing power fluctuations, EMI noise, and peripheral failures.**

**- Using oscilloscopes & logic analyzers for deeper diagnostics.**

**✔ Future Expansion & Customization**

**- Modifying firmware for additional features.**

**- Expanding hardware capabilities (new sensors, more I/O options).O options).**

📖 1. Firmware Development Guide

Overview

The firmware runs on ESP32, STM32, ATmega, and PIC, ensuring seamless hardware integration with sensors, actuators, displays, and communication modules.

Core Firmware Features

✅ Multi-MCU Support → ESP32 (Wi-Fi/Bluetooth), STM32 (industrial control), ATmega/PIC (basic automation).

✅ Modular Peripheral Control → Stepper motors, servos, GSM, LoRaWAN, Zigbee, RF.

✅ FreeRTOS for Multitasking → Handles scheduling, event-driven programming.

✅ Secure Communication (MQTT, WebSockets) → IoT cloud integration.

✅ Node-RED-like flow logic → Drag-and-drop automation.

System Setup & Programming

void setup() {

initSensors();

setupCommunication();

configureMotors();

}

void loop() {

readSensorData();

executeLogic();

sendDataToCloud();

}

🖥️ 2. Web GUI & API Development Guide

Overview

The web-based GUI provides real-time interaction, allowing users to configure, visualize, and automate tasks via drag-and-drop logic workflows.

Core Web Features

✅ Node-RED-like Programming Interface → Visual logic builder for GPIO control.

✅ IoT Dashboard for Real-Time Monitoring → Displays sensor values & device statuses.

✅ Firmware OTA Updates → Remote program selection.

✅ Multi-User Support & Authentication → Secure device access.

Sample API Request (Node.js)

app.post('/setMotorSpeed', (req, res) => {

let speed = req.body.speed;

mqttClient.publish('motor/speed', speed.toString());

res.json({ success: true });

});

🔧 3. Hardware Setup & Wiring Guide

Overview

Provides step-by-step wiring for MCUs, power distribution, and peripheral integration.

Core Hardware Components

✅ MCU Selection & Pin Mapping → ESP32, STM32, ATmega.

✅ Power Circuit Optimization → Efficient voltage regulation & EMI shielding.

✅ Sensor Interfacing → POT, LDR, Temp, ACS712, NeoPixel LEDs.

✅ Motor Drivers → DRV8825 stepper & H-Bridge servo control.

✅ Communication Modules → GSM800L, LoRaWAN, BLE, Zigbee, CAN/LIN.

Pin Mapping Example

ESP32 Pin Configuration:

GPIO12 → Stepper Motor Step Signal

GPIO14 → Direction Control

GPIO16 → Enable Pin

📡 4. API Logic & IoT Connectivity Guide

Overview

Defines how devices interact remotely, enabling cloud control and automation.

Core API Features

✅ MQTT & WebSockets for Real-Time Communication → Publish/subscribe model.

✅ Node-RED Logic Processing → API commands dynamically affect devices.

✅ Encryption & Secure Connectivity → AES-secured device messaging.

API Structure (RESTful Format)

POST /setMotorSpeed

Payload: { "speed": 500 }

Response: { "status": "ok" }

🚀 Block Categories for Flow-Based Programming

1️⃣ I/O Blocks (Input & Output)

✅ Digital Input Blocks → Reads switches, buttons, logic states (HIGH/LOW).

✅ Analog Input Blocks → Reads values from POT, LDR, Temperature sensors, ACS712.

✅ Digital Output Blocks → Controls LEDs, relays, stepper drivers, actuators.

✅ Analog Output Blocks (PWM) → Controls motors, dimmable lighting.

2️⃣ Conditional Logic Blocks

✅ IF-ELSE Block → Executes logic based on conditions (If temperature > 30°C, turn fan ON).

✅ COMPARE Block → Compares values (Is LDR brightness > 50%?).

✅ AND / OR / NOT Blocks → Implements Boolean logic (If Button1 AND Button2 are pressed, activate motor).

3️⃣ Timer & Delay Blocks

✅ Time-Based Triggers → Set delays (Activate relay after 5 seconds).

✅ Repeat Loops → Create timed actions (Blink LED every 2 seconds).

✅ Event Scheduler → Execute tasks at specific intervals (Measure sensor every 10 minutes).

4️⃣ Sensor Blocks

✅ Temperature Sensor Block → Reads data from DS18B20, DHT11, LM35.

✅ Light Sensor Block → Reads LDR & NeoPixel LED brightness levels.

✅ Motion Sensor Block → Uses PIR & Ultrasonic for object detection.

5️⃣ Actuator Blocks

✅ Stepper Motor Block → Controls speed, direction, steps using DRV8825.

✅ Servo Motor Block → Moves servo to defined angles.

✅ Relay Control Block → Activates AC appliances, industrial controls.

6️⃣ Communication Blocks

✅ Wi-Fi/MQTT Block → Sends or receives cloud-based messages.

✅ GSM Block → Sends SMS alerts (Send temperature warning to phone).

✅ LoRa/Zigbee Block → Sends long-range wireless data packets.

7️⃣ AI & Camera Processing Blocks

✅ ESP32-CAM Image Processing Block → Object detection tasks (Identify color & shape).

✅ Edge ML Block → Runs lightweight AI models for pattern detection.

🛠️ Block Interactions in Firmware Logic

Each block must correspond to a modular function in firmware, allowing plug-and-play logic execution.

1️⃣ Conditional Logic Blocks (IF-ELSE, Compare, Boolean)

✔ IF-ELSE Block Implementation

void checkTemperature(int temp) {

if (temp > 30) {

activateFan();

} else {

deactivateFan();

}

}

✔ COMPARE Block

bool compareValues(int sensor1, int sensor2) {

return sensor1 > sensor2;

}

✔ AND / OR / NOT Logic

if (button1 && button2) {

activateMotor();

}

2️⃣ Sensor Blocks (Reading Data & Processing)

✔ Temperature Sensor Block

int readTemperature() {

return analogRead(TEMP\_SENSOR\_PIN);

}

✔ Light Sensor Block

int readLDR() {

return analogRead(LDR\_PIN);

}

3️⃣ Actuator Blocks (Motors, Relays, Servos)

✔ Stepper Motor Control Block

void moveStepper(int steps, bool direction) {

digitalWrite(DIR\_PIN, direction);

for (int i = 0; i < steps; i++) {

digitalWrite(STEP\_PIN, HIGH);

delayMicroseconds(800);

digitalWrite(STEP\_PIN, LOW);

delayMicroseconds(800);

}

}

✔ Relay Control Block

void toggleRelay(bool state) {

digitalWrite(RELAY\_PIN, state);

}

📌 GUI Block Design & Connection

In the drag-and-drop interface, blocks visually snap together to form execution flows.

Block Representation

✅ Each block corresponds to a firmware function call.

✅ Blocks connect using data inputs and outputs.

✅ Conditional and event-driven blocks trigger actions dynamically.

Example GUI workflow:

[Sensor Block] ➝ [IF Condition] ➝ [Activate Motor]

🖥️ Block-Based GUI Flow Design

Each block connects logically, forming an automation sequence.

1️⃣ Block Example: Motor Activation Based on Sensor Input

[📡 Sensor Block] → [🔄 IF Condition] → [🚀 Actuator Block]

- Sensor Block: Reads temperature, light, or motion.

- IF Condition Block: Defines logic (If temperature > 30°C).

- Actuator Block: Controls motor (Activate cooling fan).

2️⃣ Block Example: Timer-Based Automation

[⏳ Timer Block] → [🔄 Repeat Loop] → [💡 LED Control Block]

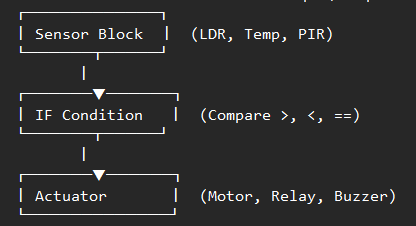
- Timer Block: Defines a trigger delay (Blink every 2 seconds).

- Repeat Loop Block: Ensures repeated execution.

- LED Control Block: Toggles LED state (ON/OFF).

🔗 Visual Representation of Blocks

Each block has connectors for input/output data:



🖥️ GUI Layout for Block-Based Programming

The interface should be intuitive, allowing users to drag-and-drop blocks to create automation sequences.

1️⃣ Main Interface Sections

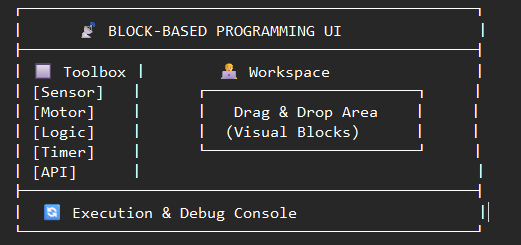
🔹 Block Workspace → Users assemble logic blocks visually.

🔹 Toolbox (Block Library) → Predefined blocks (I/O, Sensors, Motors, Timers, API).

🔹 Live Preview Panel → Shows real-time data (sensor values, motor status, logs).

🔹 Execution & Debug Console → Displays system output and errors.

2️⃣ GUI Layout Design



✔ Users drag blocks from the Toolbox to the Workspace.

✔ Blocks snap together dynamically, defining execution flow.

✔ Console shows real-time program execution results.

🔗 Refining Firmware Interaction with Blocks

Each GUI block triggers a corresponding function in the firmware.

🔹 Example Block: Sensor → IF Condition → Motor

(Temperature Sensor) ➝ (IF Temp > 30°C) ➝ (Activate Cooling Fan)

✔ Block in GUI triggers firmware function

int temp = readTemperature();

if (temp > 30) {

activateMotor();

}

🔹 Example Block: Timer → LED Blinking

(⏳ Timer Block: 2s) ➝ (💡 LED ON/OFF)

✔ Firmware executes timer-based LED blinking

void blinkLED() {

digitalWrite(LED\_PIN, HIGH);

delay(2000);

digitalWrite(LED\_PIN, LOW);

}

🖥️ Visual Block Interactions in JSON

1️⃣ Basic Block Structure

Each block should have: ✅ ID → Unique identifier for referencing.

✅ Type → Defines functionality (Sensor, Motor, Logic).

✅ Inputs → Values from sensors or user parameters.

✅ Outputs → Actions triggered (Motor ON/OFF).

✅ Connections → Links to other blocks.

Example JSON format for a Motor Control Block:

{

"block\_id": "motor1",

"type": "ACTUATOR",

"name": "Stepper Motor",

"inputs": {

"direction": "clockwise",

"speed": 500

},

"outputs": ["motor\_start"],

"connections": ["sensor\_block", "logic\_block"]

}

2️⃣ Example: Conditional Logic Block (IF-ELSE)

{

"block\_id": "if\_condition1",

"type": "LOGIC",

"name": "IF Condition",

"inputs": {

"sensor\_value": 30

},

"logic": "sensor\_value > threshold",

"outputs": ["motor1"]

}

3️⃣ Example: Timer Block

{

"block\_id": "timer1",

"type": "TIME",

"name": "Repeat Every 5s",

"interval": 5000,

"outputs": ["led\_block"]

}

📡 How GUI Elements Update in Real-Time

To ensure smooth block interactions, GUI updates should use: ✔ WebSockets for live data streaming (sensor updates, motor status).

✔ Drag-and-Drop event listeners for repositioning blocks.

✔ JSON-based execution mapping for dynamic firmware calls.

Example GUI rendering logic (JavaScript)

function updateBlockConnection(blockID, newConnection) {

let block = blockDatabase[blockID];

block.connections.push(newConnection);

sendBlockDataToFirmware(block);

}

🖥️ Expanding Real-Time GUI Rendering

1️⃣ Dynamic Block Placement & Connection

When users drag a block onto the workspace, the GUI should: ✅ Assign a unique block ID.

✅ Store the block type, inputs, and outputs.

✅ Establish connections to other blocks dynamically.

Example JavaScript logic for rendering a new block:

function addBlock(blockType) {

let blockID = "block\_" + Date.now();

let newBlock = {

id: blockID,

type: blockType,

inputs: [],

outputs: [],

connections: []

};

renderBlockOnGUI(newBlock);

}

✔ Blocks snap together visually, forming execution sequences.

2️⃣ Drag-and-Drop Event Handling

GUI should handle block movement, repositioning, and linking.

Example drag logic in JavaScript (React or Vue)

document.addEventListener("dragstart", function(event) {

event.dataTransfer.setData("blockID", event.target.id);

});

document.addEventListener("drop", function(event) {

let droppedBlockID = event.dataTransfer.getData("blockID");

updateBlockPosition(droppedBlockID, event.clientX, event.clientY);

});

✔ Blocks move smoothly, allowing users to reorganize logic flow.

3️⃣ Live Updates for Sensor & Motor Status

GUI should fetch real-time sensor data and actuator responses.

Example WebSockets data stream to GUI

socket.on("sensor\_update", function(data) {

document.getElementById("sensor\_display").innerText = "Temp: " + data.temperature;

});



✔ Users see live sensor readings instantly.

🔗 JSON Block Structure & Firmware Execution Flow

Each GUI block translates into a structured JSON command, which is then processed by the firmware.

1️⃣ JSON Format for Block-Based Execution

Each block consists of: ✅ Block Type → Defines functionality (Sensor, Motor, Timer, Logic).

✅ Inputs & Outputs → Parameter values (Speed, Direction, Threshold).

✅ Connections → Links to dependent blocks.

Example JSON for a Stepper Motor Block:

{

"block\_id": "motor1",

"type": "ACTUATOR",

"name": "Stepper Motor",

"inputs": {

"direction": "clockwise",

"speed": 500

},

"connections": ["sensor\_block", "logic\_block"]

}

✔ Firmware reads JSON values to execute motor functions.

2️⃣ Firmware Execution Engine Reads & Processes Blocks

Firmware parses JSON, identifies block type, and executes the corresponding function.

Example Firmware Code Processing JSON Commands:

void executeBlock(String jsonData) {

DynamicJsonDocument doc(512);

deserializeJson(doc, jsonData);

String blockType = doc["type"];

String blockID = doc["block\_id"];

if (blockType == "ACTUATOR") {

String direction = doc["inputs"]["direction"];

int speed = doc["inputs"]["speed"];

moveStepper(speed, direction);

}

}

✔ Maps each JSON block to a firmware function dynamically.

3️⃣ Real-Time Data Feedback from Firmware to GUI

To ensure smooth response handling, firmware sends execution results back to the GUI.

✔ Firmware sends stepper motor status (position, speed updates) to GUI:

{

"block\_id": "motor1",

"status": "running",

"current\_speed": 500,

"direction": "clockwise"

}

✔ GUI displays live status updates:

socket.on("motor\_update", function(data) {

document.getElementById("motor\_status").innerText =

"Speed: " + data.current\_speed + " | Direction: " + data.direction;

});



✔ Users see real-time execution feedback.

🔗 Multi-Block Execution Flow

When multiple blocks are connected, their execution must follow logical sequencing.

Each block reads input, processes conditions, and triggers output actions.

Example Flow:

✔ Sensor → Conditional → Timer → Actuator

[📡 Temperature Sensor] → [🔄 IF Condition: Temp > 30°C] → [⏳ Timer: Delay 5s] → [🚀 Turn ON Cooling Fan]



📄 JSON Structure for Multi-Block Mapping

To ensure seamless execution, each block should define: ✅ Block ID → Unique reference for execution order.

✅ Type → Sensor, Conditional, Timer, Actuator.

✅ Inputs → Sensor values, comparison logic, timings.

✅ Outputs → Actions triggered (motor ON/OFF, LED toggle).

✅ Connections → Which block executes next.

Example JSON for Sensor → IF Condition → Timer → Motor

[

{

"block\_id": "sensor1",

"type": "SENSOR",

"name": "Temperature Sensor",

"output": 32.5,

"connections": ["condition1"]

},

{

"block\_id": "condition1",

"type": "LOGIC",

"name": "IF Temp > 30",

"condition": "sensor1.output > 30",

"outputs": ["timer1"]

},

{

"block\_id": "timer1",

"type": "TIME",

"name": "Delay 5s",

"interval": 5000,

"outputs": ["motor1"]

},

{

"block\_id": "motor1",

"type": "ACTUATOR",

"name": "Cooling Fan",

"action": "ON"

}

]



✔ Blocks execute sequentially, ensuring a smooth automation flow.

📡 Firmware Handling Multi-Block Execution

Firmware must: ✅ Parse JSON structure to extract execution flow.

✅ Process sensor readings and condition validation dynamically.

✅ Trigger next block based on successful logic execution.

Example Firmware Code to Process Multi-Block Mapping:

void executeBlocks(JsonArray blocks) {

for (JsonObject block : blocks) {

String blockType = block["type"];

if (blockType == "SENSOR") {

int temp = readTemperature();

block["output"] = temp;

} else if (blockType == "LOGIC") {

bool conditionMet = evaluateCondition(block["condition"]);

if (conditionMet) {

triggerNextBlock(block["outputs"]);

}

} else if (blockType == "TIME") {

delay(block["interval"]);

triggerNextBlock(block["outputs"]);

} else if (blockType == "ACTUATOR") {

activateMotor();

}

}

}



✔ Dynamically executes blocks, following their connection sequence.

🛠️ Parallel Execution Handling

1️⃣ Key Requirements

✅ Multi-Actuator Control → Motors, relays, LEDs must activate at the same time.

✅ Asynchronous Execution → No delays in sequential processing.

✅ FreeRTOS or Interrupt-Based Scheduling → Ensures efficiency.

2️⃣ JSON Format for Parallel Execution

Multiple blocks must run concurrently, each triggering independently.

[

{

"block\_id": "sensor1",

"type": "SENSOR",

"name": "Temperature Sensor",

"output": 32.5,

"connections": ["condition1"]

},

{

"block\_id": "condition1",

"type": "LOGIC",

"name": "IF Temp > 30",

"condition": "sensor1.output > 30",

"outputs": ["motor1", "relay1", "led1"]

},

{

"block\_id": "motor1",

"type": "ACTUATOR",

"name": "Cooling Fan",

"action": "ON"

},

{

"block\_id": "relay1",

"type": "ACTUATOR",

"name": "Water Pump",

"action": "ON"

},

{

"block\_id": "led1",

"type": "ACTUATOR",

"name": "Warning Light",

"action": "ON"

}

]



✔ Outputs include multiple actuators, ensuring simultaneous execution.

3️⃣ Firmware Logic for Parallel Processing

To execute multiple blocks at once, the firmware should: ✅ Use FreeRTOS tasks to handle concurrent execution.

✅ Implement non-blocking functions for seamless responsiveness.

✅ Utilize interrupt-based triggering instead of delay().

Example Parallel Execution with FreeRTOS (ESP32 & STM32)

void activateParallelBlocks(void \*pvParameters) {

JsonArray blocks = parseBlocksFromJSON();

for (JsonObject block : blocks) {

if (block["type"] == "ACTUATOR") {

xTaskCreatePinnedToCore(triggerActuator, "ActuatorTask", 2048, (void\*)&block, 1, NULL, 0);

}

}

}

void triggerActuator(void \*pvParameters) {

JsonObject \*block = (JsonObject\*) pvParameters;

String action = (\*block)["action"];

if ((\*block)["name"] == "Cooling Fan") activateMotor();

else if ((\*block)["name"] == "Water Pump") toggleRelay();

else if ((\*block)["name"] == "Warning Light") blinkLED();

vTaskDelete(NULL);

}



✔ Executes all actuators in parallel using FreeRTOS tasks.

🛠️ Implementing Interrupt-Based Execution

1️⃣ Key Requirements

✅ Interrupt Service Routines (ISRs) → Handles real-time events.

✅ Non-Blocking Execution → Avoid delays (delay() functions should be removed).

✅ External Trigger Handling → Button presses, motion detection, sensor thresholds.

✅ Debouncing & Filtering → Prevent false activations.

2️⃣ Example: External Interrupt for Button Press

✔ Connect a button to trigger a function instantly.

Hardware Wiring

Button → GPIO Pin (ESP32/STM32)

✔ Firmware Interrupt Setup

#define BUTTON\_PIN 12

void IRAM\_ATTR handleButtonPress() {

Serial.println("Button Pressed!");

}

void setup() {

pinMode(BUTTON\_PIN, INPUT\_PULLUP);

attachInterrupt(digitalPinToInterrupt(BUTTON\_PIN), handleButtonPress, FALLING);

}

void loop() {

// Other tasks can run while button is monitored asynchronously

}



✔ ISR executes instantly upon button press, avoiding polling.

3️⃣ Example: Interrupt-Based Motion Detection

✔ Motion sensor triggers an action immediately.

#define PIR\_SENSOR\_PIN 15

void IRAM\_ATTR motionDetected() {

Serial.println("Motion Detected!");

}

void setup() {

pinMode(PIR\_SENSOR\_PIN, INPUT);

attachInterrupt(digitalPinToInterrupt(PIR\_SENSOR\_PIN), motionDetected, RISING);

}

void loop() {

// The main program continues running, no delay needed

}



✔ Triggers real-time events without delay.

4️⃣ Example: Timer Interrupt for Periodic Tasks

✔ Execute functions at fixed intervals without delay().

hw\_timer\_t \*timer = NULL;

void IRAM\_ATTR onTimer() {

Serial.println("Timer Triggered!");

}

void setup() {

timer = timerBegin(0, 80, true);

timerAttachInterrupt(timer, &onTimer, true);

timerAlarmWrite(timer, 1000000, true); // Trigger every 1s

timerAlarmEnable(timer);

}

void loop() {

// Other functions run normally

}



✔ Efficient background task execution.

🛠️ Multi-Interrupt Handling Strategy

✅ Interrupt Prioritization → Ensuring high-priority events get immediate execution.

✅ Interrupt Debouncing → Avoid unwanted multiple activations (for buttons).

✅ Task Scheduling via FreeRTOS → Allows real-time event handling.

✅ Concurrent Interrupt Execution → Multiple triggers work without interfering.

📡 Firmware Implementation

1️⃣ Setting Up Multiple Interrupt Sources

✔ Define multiple interrupt pins (Button, Motion Sensor, Timer).

#define BUTTON\_PIN 12

#define PIR\_SENSOR\_PIN 15

hw\_timer\_t \*timer = NULL;

// Interrupt Service Routines (ISRs)

void IRAM\_ATTR handleButtonPress() {

Serial.println("Button Pressed! Executing action...");

}

void IRAM\_ATTR motionDetected() {

Serial.println("Motion Detected! Triggering response...");

}

void IRAM\_ATTR onTimer() {

Serial.println("Timer Triggered! Performing scheduled task...");

}

void setup() {

pinMode(BUTTON\_PIN, INPUT\_PULLUP);

pinMode(PIR\_SENSOR\_PIN, INPUT);

// Attach interrupts

attachInterrupt(digitalPinToInterrupt(BUTTON\_PIN), handleButtonPress, FALLING);

attachInterrupt(digitalPinToInterrupt(PIR\_SENSOR\_PIN), motionDetected, RISING);

// Configure Timer Interrupt

timer = timerBegin(0, 80, true);

timerAttachInterrupt(timer, &onTimer, true);

timerAlarmWrite(timer, 2000000, true); // Trigger every 2s

timerAlarmEnable(timer);

}

void loop() {

// Main loop continues, interrupts handle actions separately

}



2️⃣ Handling Interrupt Prioritization

✔ Some interrupts require higher priority (e.g., emergency shutdown vs. routine update).

✔ Use flags to manage execution priority.

volatile bool emergencyTriggered = false;

void IRAM\_ATTR handleEmergencyButton() {

emergencyTriggered = true;

}

void loop() {

if (emergencyTriggered) {

Serial.println("Emergency Mode Activated!");

shutdownSystem();

}

}

3️⃣ Ensuring No Interrupt Conflicts

✔ Avoid race conditions by structuring interrupts efficiently.

✔ Use FreeRTOS tasks if multiple interrupt executions are needed concurrently.

🛠️ Interrupt Handling Using FreeRTOS

1️⃣ Key Requirements

✅ Independent Interrupt Service Routines (ISRs) → Each event executes separately.

✅ Task Scheduling for Concurrent Execution → Ensures smooth multi-event triggering.

✅ Minimal Latency & No Blocking → Events must respond instantly without affecting system flow.

✅ Priority-Based Execution → Critical interrupts execute first (e.g., emergency stop overrides routine tasks).

2️⃣ FreeRTOS Task-Based Interrupt Handling

✔ Instead of handling interrupts directly in the ISR, we send event data to FreeRTOS tasks, ensuring smooth execution.

Example: Multi-Interrupt Handling in FreeRTOS (ESP32 & STM32)

#include <Arduino\_FreeRTOS.h>

#define BUTTON\_PIN 12

#define PIR\_SENSOR\_PIN 15

volatile bool buttonPressed = false;

volatile bool motionDetected = false;

// FreeRTOS Task Handles

TaskHandle\_t buttonTaskHandle;

TaskHandle\_t motionTaskHandle;

// Interrupt Service Routines

void IRAM\_ATTR handleButtonPress() {

buttonPressed = true;

xTaskNotifyFromISR(buttonTaskHandle, 0, eNoAction, NULL);

}

void IRAM\_ATTR handleMotion() {

motionDetected = true;

xTaskNotifyFromISR(motionTaskHandle, 0, eNoAction, NULL);

}

// FreeRTOS Task Execution

void buttonTask(void \*parameter) {

while (true) {

ulTaskNotifyTake(pdTRUE, portMAX\_DELAY);

Serial.println("Button Pressed! Executing action...");

}

}

void motionTask(void \*parameter) {

while (true) {

ulTaskNotifyTake(pdTRUE, portMAX\_DELAY);

Serial.println("Motion Detected! Triggering response...");

}

}

void setup() {

Serial.begin(115200);

pinMode(BUTTON\_PIN, INPUT\_PULLUP);

pinMode(PIR\_SENSOR\_PIN, INPUT);

attachInterrupt(digitalPinToInterrupt(BUTTON\_PIN), handleButtonPress, FALLING);

attachInterrupt(digitalPinToInterrupt(PIR\_SENSOR\_PIN), handleMotion, RISING);

xTaskCreate(buttonTask, "Button Task", 2048, NULL, 1, &buttonTaskHandle);

xTaskCreate(motionTask, "Motion Task", 2048, NULL, 1, &motionTaskHandle);

}

void loop() {

// Main loop remains free for additional tasks

}



✔ Interrupts notify FreeRTOS tasks, avoiding delays.

✔ Tasks execute events independently, ensuring non-blocking behavior.

3️⃣ Optimizing Interrupt Execution Order & Priority

✔ Assign priorities to ensure high-criticality events execute first.

✔ Use FreeRTOS priority levels (xTaskCreate(..., priority)).

Example Emergency Stop Interrupt Handling

TaskHandle\_t emergencyTaskHandle;

void IRAM\_ATTR handleEmergencyStop() {

xTaskNotifyFromISR(emergencyTaskHandle, 0, eNoAction, NULL);

}

void emergencyTask(void \*parameter) {

while (true) {

ulTaskNotifyTake(pdTRUE, portMAX\_DELAY);

Serial.println("Emergency STOP Triggered! Shutting down...");

shutdownSystem();

}

}

void setup() {

attachInterrupt(digitalPinToInterrupt(EMERGENCY\_PIN), handleEmergencyStop, FALLING);

xTaskCreate(emergencyTask, "Emergency Task", 2048, NULL, 5, &emergencyTaskHandle);

}



✔ High-priority tasks like emergency stop run first, preventing system failure.

🛠️ Firmware Architecture

The firmware is structured into layers, ensuring cross-platform compatibility across ESP32, STM32, ATmega, and PIC microcontrollers.

1️⃣ Layered Firmware Design

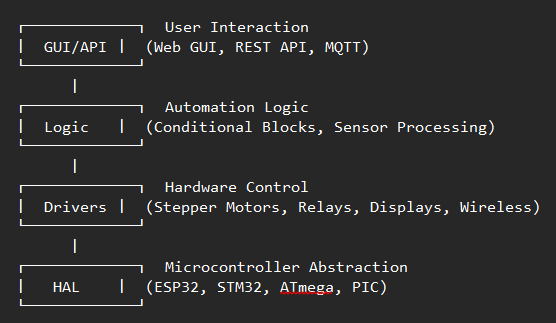
✅ Hardware Abstraction Layer (HAL) → Ensures driver compatibility for various MCUs.

✅ Peripheral Driver Layer → Manages sensors, motors, communication modules.

✅ Logic Processing Layer → Handles conditional logic, timers, and automation workflows.

✅ Application Layer → GUI and API communication for remote control & monitoring.

Example Firmware Execution Structure:



✔ Ensures modular software updates without affecting hardware drivers.

✔ Supports multiple microcontrollers, allowing flexible expansion.

🔧 Hardware Architecture

1️⃣ Core System Components

✅ Processing Units → ESP32 (Wireless IoT), STM32 (Industrial Automation), ATmega/PIC (Basic Control).

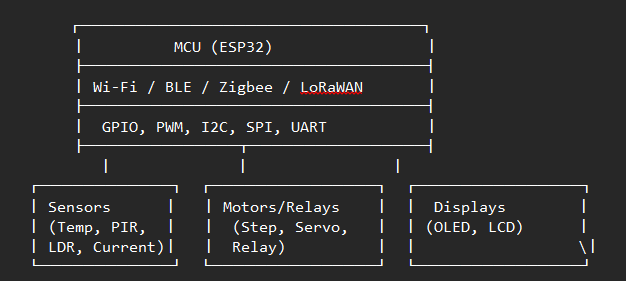
✅ Sensors → POT, LDR, Temperature Sensors, PIR Motion, ACS712 Current Sensor.

✅ Actuators → Stepper Motors (DRV8825), Servos, Relays, Buzzer, OLED/LCD Displays.

✅ Communication Modules → GSM800L, LoRaWAN, Zigbee, BLE, RF, CAN/LIN.

✅ Power Management → Voltage regulators, battery monitoring, stable distribution.

2️⃣ Hardware Block Diagram



✔ Supports real-time sensor and motor control.

✔ Ensures modular connectivity, allowing easy upgrades (new sensors, actuators).

🔄 Real-Time Execution Flow Strategy

✅ Concurrent Sensor Readings → Poll multiple sensors without delay.

✅ Parallel Actuator Control → Stepper motors, servos, relays must activate simultaneously.

✅ Asynchronous Communication → Wireless protocols (MQTT, LoRaWAN, BLE) execute without bottlenecks.

✅ Interrupt & Timer-Based Execution → Ensures immediate response while preventing CPU overload.

📡 Firmware Implementation for Parallel Processing

1️⃣ Multi-Sensor Parallel Processing

✔ Using FreeRTOS tasks to manage multiple sensor readings concurrently.

#include <Arduino\_FreeRTOS.h>

#define TEMP\_SENSOR\_PIN A0

#define LDR\_SENSOR\_PIN A1

TaskHandle\_t tempTaskHandle;

TaskHandle\_t ldrTaskHandle;

void readTemperatureTask(void \*pvParameters) {

while (true) {

int temp = analogRead(TEMP\_SENSOR\_PIN);

Serial.println("Temperature: " + String(temp));

vTaskDelay(1000 / portTICK\_PERIOD\_MS);

}

}

void readLDRTask(void \*pvParameters) {

while (true) {

int light = analogRead(LDR\_SENSOR\_PIN);

Serial.println("Light Level: " + String(light));

vTaskDelay(500 / portTICK\_PERIOD\_MS);

}

}

void setup() {

Serial.begin(115200);

xTaskCreate(readTemperatureTask, "Temp Sensor Task", 2048, NULL, 1, &tempTaskHandle);

xTaskCreate(readLDRTask, "LDR Sensor Task", 2048, NULL, 1, &ldrTaskHandle);

}

void loop() {

// Main loop remains free for additional tasks

}



✔ Each sensor runs in its own task, ensuring smooth parallel execution.

2️⃣ Parallel Actuator Control

✔ Running motors, LEDs, and relays in separate FreeRTOS tasks.

TaskHandle\_t motorTaskHandle;

TaskHandle\_t ledTaskHandle;

void activateMotorTask(void \*pvParameters) {

while (true) {

Serial.println("Stepper Motor Running...");

moveStepper(200, true);

vTaskDelay(3000 / portTICK\_PERIOD\_MS);

}

}

void blinkLEDTask(void \*pvParameters) {

while (true) {

digitalWrite(LED\_PIN, !digitalRead(LED\_PIN));

Serial.println("LED Toggled");

vTaskDelay(1000 / portTICK\_PERIOD\_MS);

}

}

void setup() {

pinMode(LED\_PIN, OUTPUT);

xTaskCreate(activateMotorTask, "Motor Task", 2048, NULL, 1, &motorTaskHandle);

xTaskCreate(blinkLEDTask, "LED Task", 2048, NULL, 1, &ledTaskHandle);

}

void loop() {

// Main loop remains free for other executions

}



✔ Stepper motor and LED run in parallel without interference.

3️⃣ Asynchronous Communication Handling

✔ MQTT, LoRa, and BLE should execute without delay.

TaskHandle\_t mqttTaskHandle;

TaskHandle\_t bleTaskHandle;

void mqttPublishTask(void \*pvParameters) {

while (true) {

client.publish("sensor/temp", String(analogRead(TEMP\_SENSOR\_PIN)).c\_str());

vTaskDelay(5000 / portTICK\_PERIOD\_MS);

}

}

void bleBroadcastTask(void \*pvParameters) {

while (true) {

BLE.broadcastData("Light: " + String(analogRead(LDR\_SENSOR\_PIN)));

vTaskDelay(2000 / portTICK\_PERIOD\_MS);

}

}

void setup() {

xTaskCreate(mqttPublishTask, "MQTT Task", 2048, NULL, 1, &mqttTaskHandle);

xTaskCreate(bleBroadcastTask, "BLE Task", 2048, NULL, 1, &bleTaskHandle);

}



✔ Ensures seamless IoT communication without interfering with sensor or actuator tasks.

🛠️ Key Debugging Techniques

✅ Task Monitoring & Logging → Track execution flow using serial debugging.

✅ Error Handling & Recovery → Use watchdogs and error detection mechanisms.

✅ Memory & Performance Profiling → Monitor stack overflow, RAM usage, and execution time.

✅ Live Sensor & Actuator Feedback → Get real-time system diagnostics for immediate troubleshooting.

1️⃣ Task Monitoring & Serial Debugging

Debugging FreeRTOS Task Execution

✔ Check if tasks are running and detect crashes.

void monitorTasks() {

Serial.println("Checking active tasks...");

Serial.print("Task Button: "); Serial.println(eTaskGetState(buttonTaskHandle));

Serial.print("Task Motion: "); Serial.println(eTaskGetState(motionTaskHandle));

}

void setup() {

Serial.begin(115200);

xTaskCreate(monitorTasks, "Task Monitor", 2048, NULL, 1, NULL);

}



✔ Displays real-time task status (Running, Suspended, Deleted).

2️⃣ Error Handling & System Recovery

Watchdog Timer to Detect Freezes

✔ Prevent system crashes by resetting after deadlocks.

void setup() {

esp\_task\_wdt\_init(5, true); // 5-second watchdog timeout

esp\_task\_wdt\_add(NULL); // Watchdog monitors main loop

}

void loop() {

if (checkSystemHealth()) {

esp\_task\_wdt\_reset(); // Reset watchdog on normal execution

} else {

Serial.println("System Error Detected! Restarting...");

ESP.restart(); // Emergency recovery mechanism

}

}

✔ Detects system hangs and recovers automatically.

3️⃣ Memory & Performance Profiling

Check Stack Overflow & RAM Usage

✔ Ensure tasks don’t exceed memory limits.

void checkMemoryUsage() {

Serial.print("Free Heap: "); Serial.println(ESP.getFreeHeap());

Serial.print("Task Stack Usage: "); Serial.println(uxTaskGetStackHighWaterMark(NULL));

}



✔ Tracks heap allocation to prevent memory crashes.

4️⃣ Live Sensor & Actuator Feedback

Monitor Hardware Status in Real-Time

✔ Validate sensor readings and actuator state.

void monitorSystem() {

Serial.print("Temperature: "); Serial.println(readTemperature());

Serial.print("Motor Speed: "); Serial.println(getMotorSpeed());

Serial.print("Relay Status: "); Serial.println(digitalRead(RELAY\_PIN));

}



✔ Displays system diagnostics for fast debugging.