

# UNIVERSIDAD NACIONAL DE COLOMBIA SEDE MANIZALES

## SISTEMA DE CONTROL DE TEMPERATURA IMPLEMENTACIÓN DE SOFTWARE EN TIEMPO REAL CON ESP32

PRESENTADO POR:

JHONATAN YARA LOPEZ EDWIN SANTIAGO  
RODRIGUEZ

ASIGNATURA: SISTEMAS EN TIEMPO REAL  
MANIZALES, COLOMBIA 2025

Documentación del Proyecto: Sistema de Control de  
Temperatura

### 1. Descripción del Hardware

El sistema está construido sobre un SoC **ESP32**, el cual gestiona la adquisición de datos, el control de actuadores y la conectividad WiFi. Los componentes principales son:

- **Unidad Central de Proceso (MCU):** ESP32 (DevKit V1 o similar). Se encarga de ejecutar FreeRTOS, manejar la lógica de control (PID/ON-OFF), servir la página web y gestionar la seguridad.
- **Interfaz de Usuario Local (HMI):**
  - **Pantalla OLED (SH1106):** Pantalla gráfica I2C de 1.3" utilizada para mostrar el estado del sistema, temperatura actual, porcentaje del motor y máscaras de seguridad para la contraseña.
  - **Teclado Matricial 4x4:** Permite el ingreso de contraseñas y configuración manual sin necesidad de red.
- **Sensores:**
  - **Sensor de Temperatura (LM35):** Sensor analógico lineal. Se utiliza una atenuación de 11dB/12dB en el ADC del ESP32 para leer el rango completo de voltaje.
  - **Sensor de Movimiento (PIR HC-SR501):** Sensor digital que detecta presencia para activar funciones de seguridad o ahorro de energía.
- **Actuadores:**

- **Ventilador DC (Motor):** Controlado mediante una señal PWM (Modulación por Ancho de Pulso) a 5kHz con una resolución de 13 bits, permitiendo un ajuste suave de velocidad.

---

## 2. Diagrama de Bloques del Hardware

(Nota para el usuario: Puedes usar este esquema lógico para dibujar tu diagrama en Visio, Draw.io o PowerPoint).

Fragmento de código

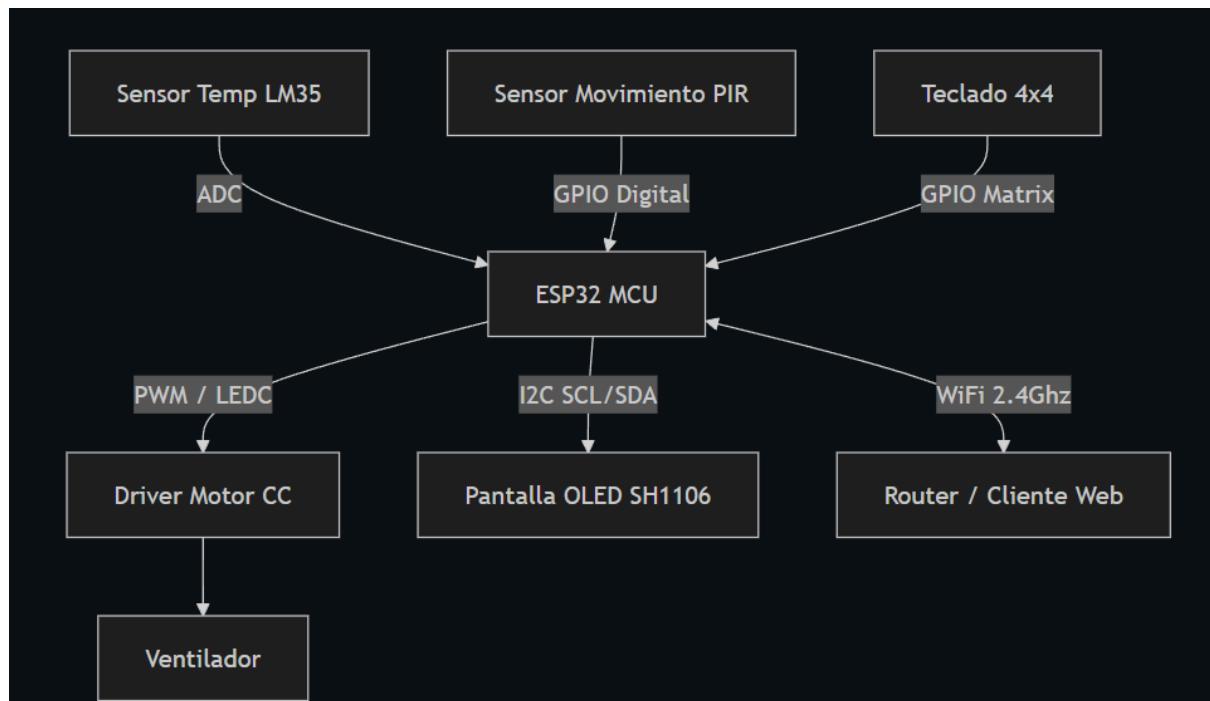
```
None

graph TD
    subgraph Entradas [Sensores y Entrada de Datos]
        PIR[Sensor PIR] -->|GPIO 15 (Digital)| ESP32
        LM35[Sensor LM35] -->|GPIO 35 (ADC)| ESP32
        Keypad[Teclado 4x4] -->|8 GPIOs| ESP32
    end

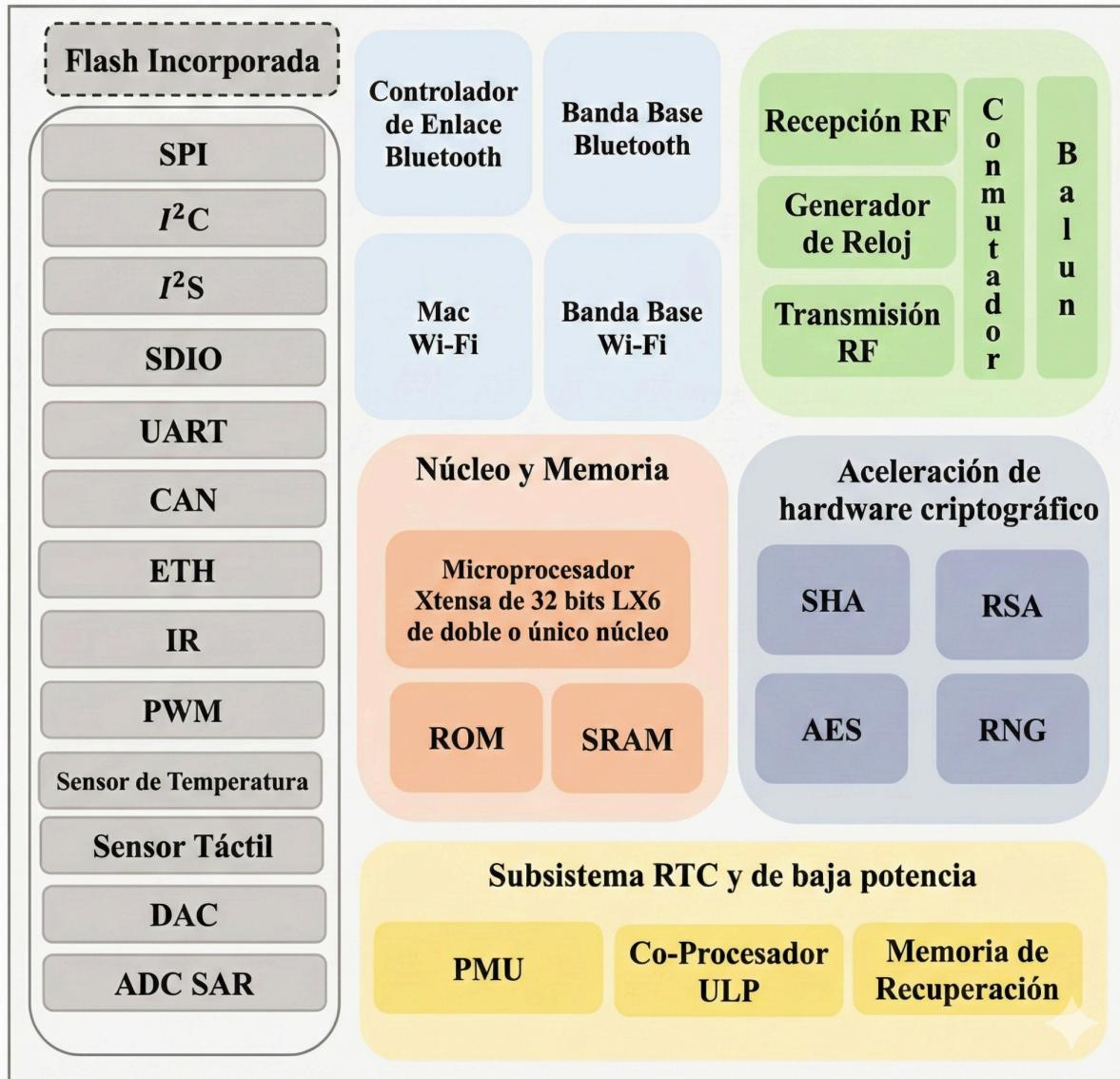
    subgraph Procesamiento
        ESP32[Microcontrolador ESP32]
        WiFi[Módulo WiFi Interno]
    end

    subgraph Salidas [Actuadores e Interfaz]
        ESP32 -->|I2C (SDA/SCL)| OLED[Pantalla OLED SH1106]
        ESP32 -->|GPIO 23 (PWM)| Motor[Driver Motor / Ventilador]
    end

    subgraph Conectividad
        WiFi <-->|HTTP/JSON| Cliente[Navegador Web / App]
    end
```



## Diagrama de Bloques de Funciones ESP32



### 3. Diagrama de Conexiones (Pinout)

La siguiente tabla detalla las conexiones físicas entre el ESP32 y los periféricos, basadas en la configuración del firmware ([Display.h](#), [Motor.h](#), [Sensor.h](#), [keypad.h](#), [Temp\\_LM35.h](#)).

| Componente              | Pin del Componente | Pin ESP32 (GPIO) | Función / Notas        |
|-------------------------|--------------------|------------------|------------------------|
| OLED (I <sup>2</sup> C) | SDA                | GPIO 21          | Datos I <sup>2</sup> C |

|                    |                |                |                                     |
|--------------------|----------------|----------------|-------------------------------------|
|                    | SCL            | <b>GPIO 22</b> | Reloj I2C                           |
|                    | VCC / GND      | 3.3V / GND     | Alimentación                        |
| <b>Motor (Fan)</b> | IN / Signal    | <b>GPIO 23</b> | Salida PWM (LEDC Timer 0)           |
| <b>Sensor PIR</b>  | OUT            | <b>GPIO 15</b> | Entrada Digital (Pull-down interno) |
| <b>Sensor LM35</b> | Vout           | <b>GPIO 35</b> | Entrada Analógica (ADC1_CH7)        |
| <b>Teclado 4x4</b> | Fila 1 (R1)    | <b>GPIO 13</b> | Salida de barrido                   |
|                    | Fila 2 (R2)    | <b>GPIO 12</b> | Salida de barrido                   |
|                    | Fila 3 (R3)    | <b>GPIO 14</b> | Salida de barrido                   |
|                    | Fila 4 (R4)    | <b>GPIO 27</b> | Salida de barrido                   |
|                    | Columna 1 (C1) | <b>GPIO 26</b> | Entrada con Pull-up                 |
|                    | Columna 2 (C2) | <b>GPIO 25</b> | Entrada con Pull-up                 |
|                    | Columna 3 (C3) | <b>GPIO 33</b> | Entrada con Pull-up                 |
|                    | Columna 4 (C4) | <b>GPIO 32</b> | Entrada con Pull-up                 |

# Documentación del Proyecto - Parte 2: Firmware

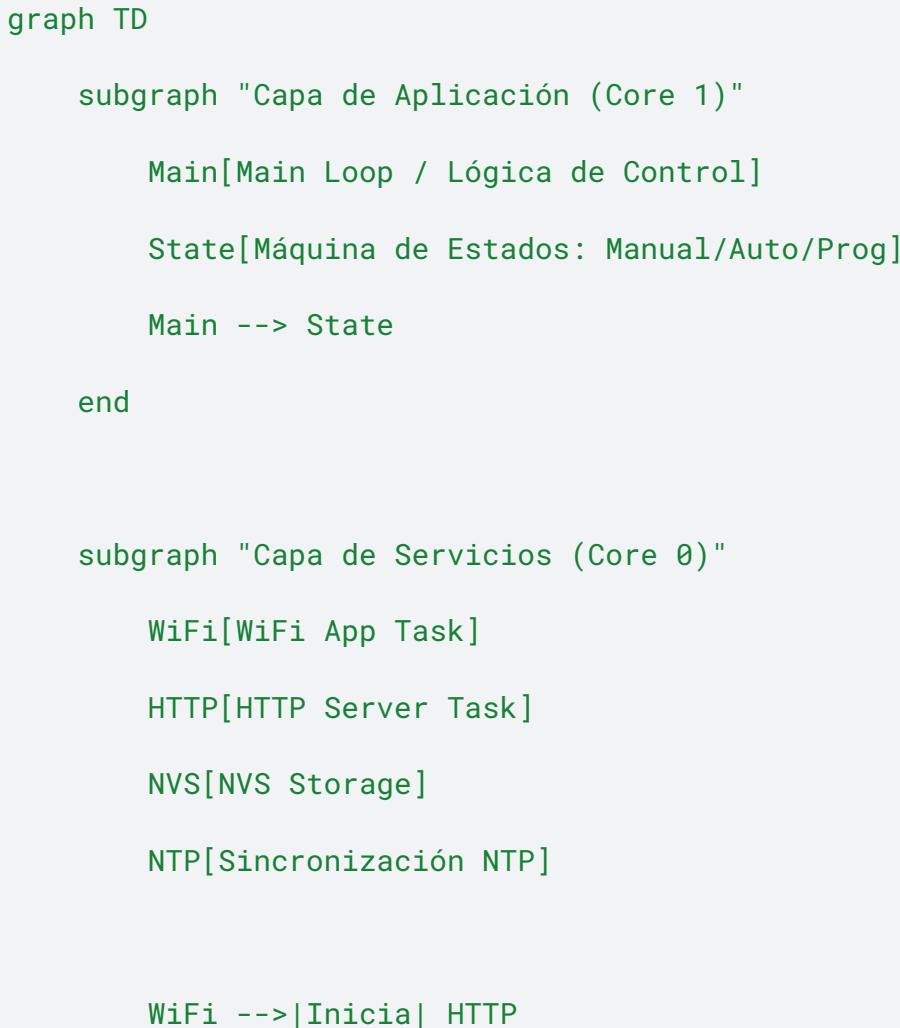
## 4. Diagrama de Bloques del Firmware

El firmware está diseñado bajo una arquitectura modular basada en **FreeRTOS**. Se divide en tres capas principales:

1. **Capa de Aplicación (Main)**: Contiene la lógica de negocio, la máquina de estados (Manual/Auto/Prog) y el bucle de control principal.
2. **Capa de Servicios**: Maneja la conectividad (WiFi, NTP) y la interfaz remota (Servidor HTTP, API REST).
3. **Capa de Drivers (HAL)**: Abstacta el hardware específico (motores, sensores, pantalla) del resto del sistema.

Fragmento de código

None



```

WiFi -->|Sincroniza| NTP

HTTP <-->|Lee/Escribe Config| NVS

end

subgraph "Capa de Hardware / Drivers"
    OLED[Driver Pantalla]
    Key[Driver Teclado]
    Sensors[Driver Sensores Temp/PIR]
    Motor[Driver PWM Motor]
end

%% Relaciones

Main <-->|Variables Globales Compartidas| HTTP

Main -->|Actualiza| OLED

Main -->|Lee| Key

Main -->|Lee| Sensors

Main -->|Controla| Motor

```

## 5. Descripción de Tareas (FreeRTOS Tasks)

El sistema utiliza el sistema operativo en tiempo real FreeRTOS para gestionar la concurrencia. Las tareas están configuradas en el archivo `tasks_common.h` y distribuidas para optimizar el uso de los dos núcleos del ESP32.

### Tabla de Tareas del Sistema

| Nombre de la Tarea          | Prioridad      | Tamaño de Pila (Stack) | Core (Núcleo) | Frecuencia / Comportamiento  | Descripción   |
|-----------------------------|----------------|------------------------|---------------|------------------------------|---|
| <b>Main Task (app_main)</b> | 1<br>(Default) | Default                | 1             | Bucle Infinito (100ms delay) | <b>Tarea Principal.</b> Ejecuta la lógica de control. Lee el teclado y sensores, calcula la salida PWM según el modo (Manual/Auto/Prog) y actualiza la pantalla OLED. |
| <b>WiFi App Task</b>        | 5              | 4096 bytes             | 0             | Basado en Eventos (Colas)    | Gestiona la conexión WiFi (Station/SoftAP), reconexiones automáticas y obtiene la hora vía NTP. Al conectarse, notifica al sistema para iniciar el servidor web.      |
| <b>HTTP Server Task</b>     | 4              | 8192 bytes             | 0             | Bajo Demanda (Request)       | Escucha peticiones HTTP en el puerto 80. Sirve la interfaz web ( <a href="#">index.html</a> ) y procesa la API JSON para monitoreo y control remoto.                  |

|                          |   |            |   |                           |   |
|--------------------------|---|------------|---|---------------------------|---|
| <b>HTTP Monitor Task</b> | 3 | 4096 bytes | 0 | Cíclico (Baja frecuencia) | Watchdog del servidor web. Supervisa la salud de la tarea HTTP y gestiona el ciclo de vida de las conexiones para liberar recursos. |
|--------------------------|---|------------|---|---------------------------|---|

### Mecanismo de Comunicación entre Tareas

- Colas (Queues):** Se utiliza `wifi_app_queue_handle` para enviar mensajes asíncronos entre las interrupciones del sistema (eventos WiFi) y la tarea de aplicación WiFi, evitando bloqueos.
- Variables Globales Compartidas:** La comunicación entre el Servidor Web (Core 0) y el Bucle Principal (Core 1) se realiza mediante variables globales protegidas (`system_mode`, `manual_pwm_val`, `current_temp`). Esto permite que la web cambie la configuración y el `main` reaccione casi instantáneamente.

## Documentación del Proyecto - Parte 3: Documentación del Código

### 6. Estándar de Documentación (Doxygen)

El código fuente ha sido documentado siguiendo el estándar Doxygen. Esto permite la generación automática de manuales de referencia en formatos HTML y PDF. A continuación, se presentan los ejemplos clave de cómo están documentadas las interfaces de hardware y lógica.

#### 6.1. Ejemplo: Módulo de Control de Motores (`Motor.c`)

Este módulo gestiona la señal PWM para el ventilador.

C

```
None
/** 
 * @file Motor.c
 *
 * @brief Controlador del driver PWM para el ventilador DC.
```

```
* @author Tu Nombre  
* @date 2023-10-27  
  
* * Este archivo implementa la configuración del timer LEDC  
del ESP32  
  
* para generar una señal PWM de 5kHz con resolución de 13  
bits.  
  
*/  
  
#include "Motor.h"  
  
/**  
 * @brief Inicializa el hardware PWM.  
 * * Configura el temporizador (Timer 0) y el canal (Channel  
0) del periférico LEDC.  
 * Asigna el GPIO 23 como salida de la señal.  
 * * @note Debe llamarse antes de intentar establecer  
cualquier velocidad.  
 * @return void  
*/  
  
void motor_init(void) {  
    // ... implementación del código ...  
  
}  
  
/**  
 * @brief Establece la velocidad de giro del ventilador.
```

```

 * * Convierte un porcentaje (0-100%) en un ciclo de trabajo
(Duty Cycle)

 * para la resolución de 13 bits (0-8191).

 * * @param percent Porcentaje de velocidad deseado (Entero
entre 0 y 100).

 * Si es < 0 se asume 0, si es > 100 se asume 100.

 * @return void

 */

void motor_set_speed_percent(int percent) {

    // ... implementación del código ...

}

```

## 6.2. Ejemplo: Módulo de Sensores (**Sensor.c**)

**Encapsula la lectura digital del sensor de movimiento PIR.**

**C**

```

None

/**


 * @file Sensor.c

 * @brief Driver para el sensor de movimiento PIR HC-SR501.

 */




/**

 * @brief Verifica el estado del sensor de presencia.

 * * Lee el estado lógico del GPIO 15.

```

```
* * @return true Si se detecta movimiento (Nivel Alto).  
* @return false Si no hay movimiento (Nivel Bajo).  
*/  
  
bool sensors_get_pir_state(void) {  
  
    return gpio_get_level(PIR_PIN);  
  
}
```

### 6.3. Ejemplo: Servidor Web ([http\\_server.c](#))

Maneja las peticiones HTTP y la API REST.

C

```
None  
/**  
  
 * @file http_server.c  
  
 * @brief Servidor HTTP embebido y API REST JSON.  
  
 * * Gestiona las rutas para servir la interfaz web  
(index.html) y  
  
 * los endpoints de control (/api/settings, /api/status).  
  
 */  
  
/**  
  
 * @brief Inicia la tarea del servidor web.  
  
 * * Configura la estructura `httpd_config_t`, asigna memoria  
para el stack (8192 bytes)  
  
 * y registra los manejadores de URI (URI handlers).  

```

```

* * @see webpage_get_handler

* @see settings_post_handler

* * @return httpd_handle_t Puntero al manejador del servidor
creado, o NULL si falló.

*/

httpd_handle_t start_webserver(void) {

    // ... implementación del código ...

}

/**

 * @brief Handler para peticiones GET de la página principal.

 * * Responde con el contenido del archivo `index.html`
embebido en la memoria flash.

 * @param req Estructura de la petición HTTP entrante.

 * @return esp_err_t ESP_OK si se envió correctamente.

*/

static esp_err_t webpage_get_handler(httpd_req_t *req) {

    // ... implementación del código ...

}

```

## Documentación del Proyecto - Parte 3: Documentación del Código

### 6. Estándar de Documentación (Doxygen)

El código fuente ha sido documentado siguiendo el estándar Doxygen. Esto permite la generación automática de manuales de referencia en formatos HTML y PDF. A continuación, se presentan los ejemplos clave de cómo están documentadas las interfaces de hardware y lógica.

## 6.1. Ejemplo: Módulo de Control de Motores ([Motor.c](#))

Este módulo gestiona la señal PWM para el ventilador.

C

```
None

/**


 * @file Motor.c

 * @brief Controlador del driver PWM para el ventilador DC.

 * @author Tu Nombre

 * @date 2023-10-27

 * * Este archivo implementa la configuración del timer LEDC
 * del ESP32

 * para generar una señal PWM de 5kHz con resolución de 13
 * bits.

 */

#include "Motor.h"



/**

 * @brief Inicializa el hardware PWM.

 * * Configura el temporizador (Timer 0) y el canal (Channel
 * 0) del periférico LEDC.

 * Asigna el GPIO 23 como salida de la señal.
```

```

    * * @note Debe llamarse antes de intentar establecer
    cualquier velocidad.

    * @return void

    */

void motor_init(void) {

    // ... implementación del código ...

}

/**

 * @brief Establece la velocidad de giro del ventilador.

 * * Convierte un porcentaje (0-100%) en un ciclo de trabajo
(Duty Cycle)

 * para la resolución de 13 bits (0-8191).

 * * @param percent Porcentaje de velocidad deseado (Entero
entre 0 y 100).

 * Si es < 0 se asume 0, si es > 100 se asume 100.

 * @return void

*/

void motor_set_speed_percent(int percent) {

    // ... implementación del código ...

}

```

## 6.2. Ejemplo: Módulo de Sensores (`Sensor.c`)

Encapsula la lectura digital del sensor de movimiento PIR.

C

```

None

/**


 * @file Sensor.c

 * @brief Driver para el sensor de movimiento PIR HC-SR501.

 */





/**

 * @brief Verifica el estado del sensor de presencia.

 * * Lee el estado lógico del GPIO 15.

 * * @return true Si se detecta movimiento (Nivel Alto).

 * @return false Si no hay movimiento (Nivel Bajo).

 */

bool sensors_get_pir_state(void) {

    return gpio_get_level(PIR_PIN);

}

```

### 6.3. Ejemplo: Servidor Web ([http\\_server.c](#))

Maneja las peticiones HTTP y la API REST.

C

```

None

/**


 * @file http_server.c

 * @brief Servidor HTTP embebido y API REST JSON.

 * * Gestiona las rutas para servir la interfaz web
(index.html) y

```

```
* los endpoints de control (/api/settings, /api/status).  
*/  
  
/**  
 * @brief Inicia la tarea del servidor web.  
 * * Configura la estructura `httpd_config_t`, asigna memoria  
 para el stack (8192 bytes)  
 * y registra los manejadores de URI (URI handlers).  
 * * @see webpage_get_handler  
 * @see settings_post_handler  
 * * @return httpd_handle_t Puntero al manejador del servidor  
 creado, o NULL si falló.  
 */  
  
httpd_handle_t start_webserver(void) {  
    // ... implementación del código ...  
}  
  
/**  
 * @brief Handler para peticiones GET de la página principal.  
 * * Responde con el contenido del archivo `index.html`  
 embebido en la memoria flash.  
 * * @param req Estructura de la petición HTTP entrante.  
 * @return esp_err_t ESP_OK si se envió correctamente.  
 */
```

```
static esp_err_t webpage_get_handler(httpd_req_t *req) {  
    // ... implementación del código ...  
}
```