UACM
Universidad Autónoma
de la Ciudad de México
*Nada humano me es ajeno*

# UNIVERSIDAD AUTONOMA DE LA CIUDAD DE MÉXICO.

Documento de Arquitectura de Software: Sistema de Control Indoor v3.0 (ESP32 + FreeRTOS + Firebase dentro de um sistema web))

Version 3.0.

Profesor: Maximo Eduargo Gutierrez Sanchez.

Ramos Jain Ángel.
Navarrete Pérez Diego Ángel.

Tabla de contenido

Sistema de interés: “Sistema Indoor”

## Sistema de interés: “Sistema Indoor” (ESP32-FreeRTOS + Firebase dentro de un sistema web)

## Nuestro sistema.

El sistema Indoor desarrollado se centra en el cultivo dentro de un espacio cerrado, el sistema planteado para ello consiste en el control de humedad, luz y riego por medio de una interfaz web que provee información a una base de datos conocida como Firebase.

El sistema Indoor paso de tener elementos físicos (pantalla LCD y botones) a trabajar dentro de un sistema web (interfaz) que cumple la misma funcionalidad que si poseyera los elementos físicos.

¿Qué es?

El propósito de los sistemas Indoor se centra en la práctica de cultivo tradicional, plantas dentro de un entorno cerrado, así como controlado haciendo uso de técnicas específicas para la simulación de condiciones óptimas para su crecimiento.

Este método de cultivo se ha vuelto más popular desde los aficionados que buscan cuidar sus plantas, hasta cultivadores experimentados puesto que el sistema Indoor provee la oportunidad de cultivar una variedad de especímenes de forma independiente, ajustando a las condiciones climáticas externas.

Esto provee control sobre algunos elementos como lo son:

- Luz
- Humedad
- Ventilación
- Riego

Esto maximiza el rendimiento del crecimiento de las plantas para obtener cosechas de una muy alta calidad. Una de las ventajas de hacer este tipo de sistemas es poder llevar a cabo cultivos durante todo el año, donde las estaciones no interfieren en los cultivos que estén siendo desarrollados dentro del sistema Indoor.

Hacer esto nos brinda la oportunidad de elegir el momento en que queramos cultivar, buscan la optimización del agua y nutrientes promoviendo un crecimiento más rápido y eficiente.

El llevar a cabo desarrollar un sistema Indoor consiste en elegir el equipo adecuado, que va desde la iluminación, la ventilación, como está el control ambiental, cada uno de los componentes del sistema desarrollan un papel fundamental para tener éxito en el cultivo.

Ventajas.

La ventaja principal de llevar el cultivo en interiores es tener control total sobre las condiciones ambientales pues sabemos que el desarrollo de una planta o cultivo depende mucho de ciertos factores como lo son:

- Humedad
- Luz
- Ventilación
- Riego

Estos elementos pueden ser ajustados según las necesidades de nuestras plantas o cultivos, optimizando el crecimiento, así como la salud de nuestros cultivos haciendo mínima las variaciones que podrían afectar de forma negativa.

Así mismo otra ventaja es la reducción de plagas, así como enfermedades siendo un beneficio muy significativo pues existe una limitación enorme a los insectos y microorganismos.

## Versión.

La primera versión V1.0 del sistema se construyó sobre un enfoque procedimental y secuencial, típico de los entornos de desarrollo Arduino.
El flujo de ejecución se concentraba en la función loop(), la cual ejecutaba de forma cíclica y lineal todas las tareas del sistema (lectura de sensores, control de relés, actualización de LCD.

En la segunda versión V2.0, se adoptó una arquitectura basada en un sistema operativo en tiempo real (FreeRTOS), nativo del microcontrolador ESP32. Este cambio marcó una transición hacia un diseño modular y concurrente, donde cada funcionalidad principal (control de riego, iluminación, lectura de sensores, comunicación con Firebase, interfaz de usuario, etc.) se ejecuta como una tarea independiente (task) dentro del scheduler de FreeRTOS.

En la versión V2.1 se implementa el uso del módulo de hardware RTC (DS3231) esto brinda el beneficio de llevar un mejor control dentro de la iluminación, El tiempo se

calcula mediante código lo que aumenta mucho la complejidad del código y la sincronización de los objetos físicos y red.

La versión v3.0 representa la evolución natural del sistema Indoor hacia un modelo más económico, más estable y más sencillo de mantener, eliminando componentes físicos innecesarios como son botones, ahora solo dejando uno para el reseteo del wifi, así como retirando el módulo RTC, trasladando la interfaz del usuario al entorno web.

## 1. Identificación y Resumen de la Arquitectura

El sistema fue rediseñado, abandonando la arquitectura original basada en un Patrón State (documentada en Diseño de Software Indoor que se ejecutaba en un único hilo loop(), para adoptar una arquitectura multitarea concurrente basada en FreeRTOS. El sistema es un dispositivo embebido basado en un microcontrolador ESP32 que utiliza el sistema operativo de tiempo real (RTOS) FreeRTOS.

Su propósito es automatizar y monitorear un entorno de cultivo interior mediante:

- La lectura concurrente de sensores (humedad del suelo).
- El control de actuadores (relés para bomba de agua, ventilador y luz).
- Una interfaz de usuario local y en formato web para configuración y monitoreo.
- La conectividad Wi-Fi para reportar el estado a una base de datos en la nube (Firebase).

Evolución del Diseño: Esta arquitectura es una nueva evolución de la versión V2.1 la versión anterior se implementaba el uso de una interfaz LCD física con botones, así como el sistema web, sin embargo, se genera una problemática para el correcto funcionamiento del sistema físico y web, pues se pueden presentar fallas referentes a la responsividad.

Está diseñada para la concurrencia real y la separación de preocupaciones (un concepto clave de ISO 42010, Anexo A.3), asignando tareas específicas a los dos núcleos del procesador del ESP32.

La arquitectura actual (v3.0), que se basa en el sistema operativo de tiempo real FreeRTOS en un ESP32. Esta versión obtiene la hora para el control de iluminación y los *timestamps* mediante NTP (Network Time Protocol) a través de la conexión Wi-Fi,

modificando el uso de elementos físicos a un sistema completamente web pues el hardware como son los botones, al igual que el modulo RTC y la pantalla LCD empezaron a generar una problemática con el código.

## 1.1 Justificación de la Arquitectura

Esta sección describe las decisiones clave que llevaron a la arquitectura v3.0 actual.

*Decisión Clave (v1 → v2): Migración de Patrón State (Single-Loop) a FreeRTOS*

- Contexto (v1.0): La primera versión (documentada en Diseño de Software (Indoor) se basó en un Patrón de Diseño State ejecutado secuencialmente en el loop(). El EstadoOperacional era responsable de *todo*: leer sensores, actualizar LCD, verificar botones y controlar relés.
- Problema: Al agregar la conexión a Firebase (C-3), una operación lenta y bloqueante, el diseño v1.0 se volvió inviable. Una llamada a Firebase habría "congelado" la lectura de botones y el control de relés, violando las preocupaciones C-1 (Responsividad) y C-2 (Fiabilidad).
- Justificación (v2.0): Se migró a FreeRTOS para obtener mayor simplicidad y control.
    - Simplicidad: La lógica se separó en tareas independientes (taskMenu, taskControlRele, taskFirebaseUpdate), haciendo que cada módulo sea más simple de depurar y mantener que un único EstadoOperacional monolítico.
    - Control: Se asignaron tareas a núcleos específicos. La taskFirebaseUpdate (lenta) se fijó al Núcleo 1, mientras que taskMenu (rápida) se fijó al Núcleo 0. Esto garantiza que la conexión a la nube *nunca* pueda bloquear la interfaz de usuario.

*Decisión Clave (v2 → v2.1): Inclusión de Módulo RTC (DS3231) sin alterar la arquitectura FreeRTOS.*

- *Contexto (v2.0):*

*Tras migrar a FreeRTOS, el sistema ya contaba con tareas concurrentes y comunicación con base de datos, pero aún no disponía de un método estable de obtención de hora.*

*El control automático de luces, timestamps y registros necesitaban una fuente temporal confiable.*

- *Problema:*

*El ESP32 no mantenía la hora entre reinicios.*
*Para evitar inconsistencias temporales, se decidió añadir un RTC DS3231.*

- *Justificación (v2.1): Uso de RTC sin alterar estructura de v2 o Estabilidad Temporal:*

*El módulo RTC proporciona hora precisa incluso sin conexión Wi-Fi. o No*

*se modificó la arquitectura base:*

*Las tareas FreeRTOS se conservaron sin cambios: solo se añadió el RTC como componente de soporte.*

*o Fiabilidad:*

- *No depende de conexión Wi-Fi para tener hora correcta.*
- *Mantiene precisión gracias al cristal interno del DS3231.*

*o Soporte a funciones clave:*

- *Control de luces basado en horario*
- *Generación de timestamps para BD*
- *Lógica definida en taskControlRele y taskDatabaseUpdate*

*Decisión Clave (v2.1 → v3): Eliminación del RTC y Migración a NTP*

- *Contexto (v2.1):*

*El sistema utilizaba el RTC_DS3231 como fuente principal de hora.*
*Aunque preciso, este introducía hardware adicional, costos y un punto de falla (batería y cristal).*

- *Problema:*

- *Se aumentaba el costo final del Indoor.*
- *El módulo RTC añadía complejidad y era un componente más susceptible a fallos.*
- *El sistema ya usaba Wi-Fi para comunicación con base de datos → se podía reutilizar.*
- *Justificación (v3.0): Eliminación del RTC → uso de NTP o Reducción de costos y complejidad (C-5): Se elimina completamente un componente físico. o Eficiencia (No Redundancia):*

*El Indoor ya necesita Wi-Fi → se aprovecha esa conexión para obtener hora exacta vía NTP. o Fuente Única y Consistente de Tiempo:*

*Tanto:*

- *taskControlRele*
- *taskDatabaseUpdate*

*usan la misma función getLocalTime(), sincronizada por el ESP32.*

*o Degradación Controlada ante fallos de Wi-Fi:*

- *En setup():*
  *Intento de conexión con timeout, sin bloquear el sistema.*
- *En taskControlRele:*
  *Si getLocalTime() falla → se usa hora por defecto (0) evitando estados indefinidos.*
- *En taskDatabaseUpdate: Si la sincronización falla →*
  *se genera timestamp con epoch interno, permitiendo enviar datos sin interrupciones.*

Estas modificaciones volviéndose una evolución de:

- V3.0 Pagina web con uso adecuado de ESP32 , se retira el modulo RTC y se hace uso de una página web.

- v2.1 (FreeRTOS-RTC): Que utilizaba un componente de hardware dedicado (RTC DS3231).
- V2.0 Se cambia del sistema Loop a un sistema llamado FreeRTOS, implementando una base de datos.
- v1.0 (Patrón State): Que se basaba en un Patrón de Diseño State secuencial (documentado en Diseño de Software (Indoor)).

Donde la justificación de estas migraciones es una parte central de este documento.

## 2. Stakeholders y Concerns (ISO 5.3)

- S-1: Usuario / Operador: La persona que gestiona el cultivo y necesita una interfaz fiable.
- S-2: Desarrollador / Mantenedor: La persona que escribe, depura y actualiza el software del sistema.

2.2. Concerns (Preocupaciones)

| Código | Stakeholder(s) | Descripción / Preocupación |
|---|---|---|
| C-1 | S-1 | Responsividad de la UI. La interfaz debe ser fluida y responder instantáneamente, sin “congelarse” al conectar Wi-Fi o ejecutar otras tareas (problema del diseño basado en loop()). |
| C-2 | S-1, S-2 | Fiabilidad del Control. La lógica de control (riego y luz) debe operar de manera robusta y predecible. |
| C-3 | S-2 | Conectividad. El sistema debe reportar datos a la nube (Firebase) sin interrumpir las funciones críticas C-1 y C-2. |
| C-4 | S-2 | Complejidad de Gestión. El código debe ser mantenible a medida que se agregan nuevas funcionalidades. |
| C-5 | S-2 | Reducción de Costos y Complejidad de Hardware. El diseño debe ser simple y económico, eliminando puntos de falla innecesarios. |
| C-6 | S-2 | Concurrencia. El sistema debe ejecutar múltiples acciones simultáneamente (leer sensor, actualizar LCD, controlar relés, enviar datos). |
| C-7 | S-2 | Asignación de Recursos. Las tareas deben distribuirse entre los dos núcleos del ESP32 para optimizar el rendimiento (especialmente Wi-Fi). |
| C-8 | S-1, S-2 | Fiabilidad Operativa. El control de luz y humedad debe funcionar 24/7 sin fallos. |
| C-9 | S-2 | Flujo de Datos y Seguridad. Las tareas deben comunicarse de forma segura, sin corrupción de datos ni condiciones de carrera. |
| C-10 | S-1, S-2 | Conectividad Confiable. El sistema debe reportar su estado a Firebase de manera estable y continua. |

¿Porque usar Firebase?

Firebase es una plataforma de Google que facilita la conexión de dispositivos IoT (como el ESP32) conservicios en la nube. En este sistema indoor, se utiliza Firebase para monitorear, almacenar y controlar las variables ambientales desde cualquier lugar. A continuación, se explican las razones principales:

1. Monitoreo remoto: El ESP32 envía a Firebase los datos de humedad, luz y ventilación. Desde una aplicación o página web, el usuario puede visualizar el estado del sistema en tiempo real sin estar físicamente presente.
2. Almacenamiento histórico: Los datos se guardan con una marca de tiempo (timestamp), permitiendo crear registros históricos y analizar la evolución de las condiciones ambientales.
3. Comunicación bidireccional (IoT): Firebase no solo recibe datos, también puede enviar órdenes al ESP32. Por ejemplo, desde una app se puede cambiar el valor de humedad mínima o encender la luz manualmente.
4. Seguridad y autenticación: Firebase requiere un API Key y configuración de autenticación, garantizando que solo dispositivos autorizados puedan acceder o modificar los datos.
5. Integración sencilla con ESP32: La librería oficial `Firebase_ESP_Client` facilita el envío y recepción de datos en formato JSON, simplificando la comunicación con la base de datos.
6. Sin necesidad de servidor propio: No se requiere crear ni mantener un servidor. Firebase proporciona toda la infraestructura, reduciendo costos y complejidad de implementación.
7. Actualización en tiempo real: La base de datos se actualiza instantáneamente cuando cambian los valores, permitiendo reflejar los datos al instante en la interfaz del usuario.

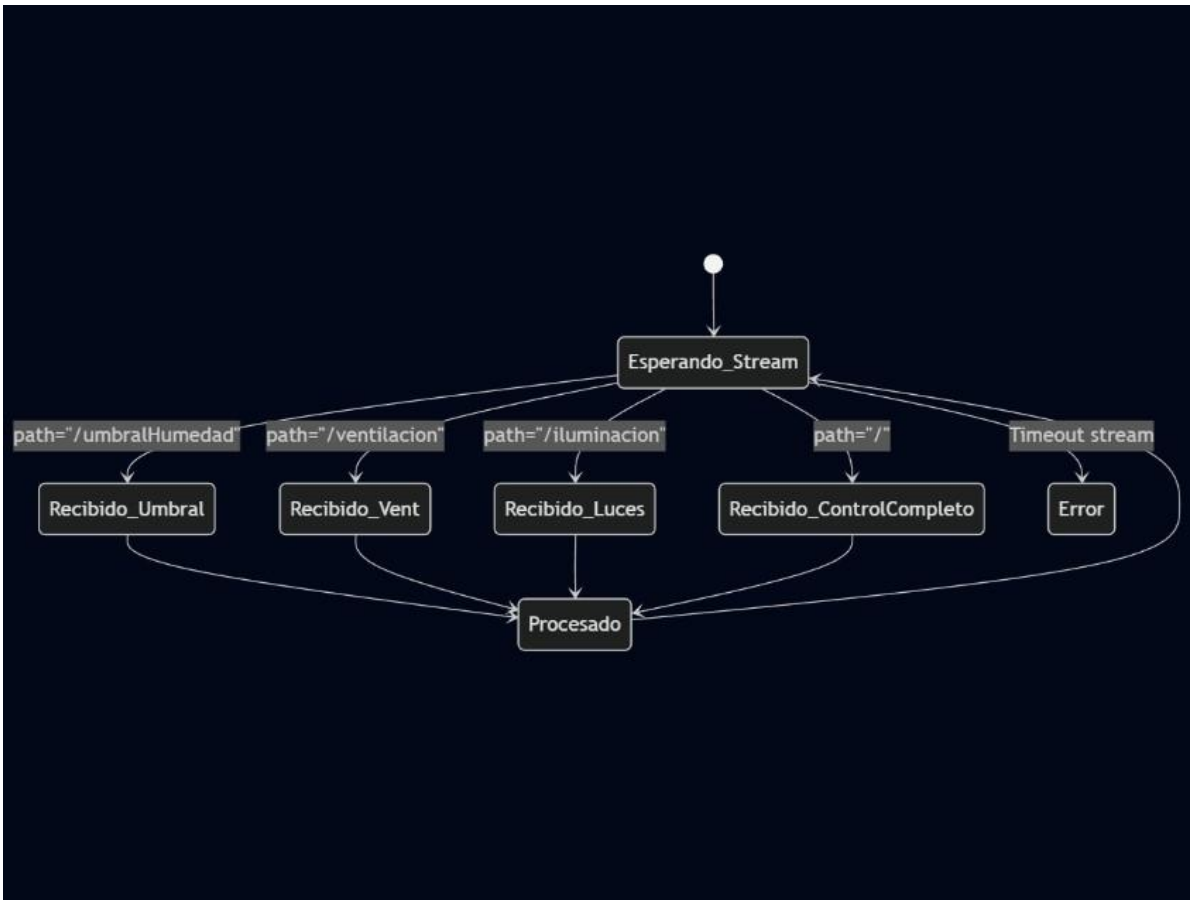



*Ilustración 1 Diagrama del funcionamiento de Firebase dentro de un ciclo de comunicación entre la base de datos y el ESP32 para el procesamiento de los datos.*

Para que el ESP32 y la Web se comuniquen, deben acordar una estructura de datos (un "contrato"). La estructura de pagina.html es la correcta:

JSON

{

"sensores": {

"humedadSuelo": 58.4 // (Escrito por ESP32)

},

"estado": {

"bomba": "OFF",  // (Escrito por ESP32)

"ventilador": "ON",  // (Escrito por ESP32)

"luces": "ON"  // (Escrito por ESP32)

},

"control": {

"umbralHumedad": 40,  // (Escrito por Web, Leído por ESP32)

"ventilacion": true,  // (Escrito por Web, Leído por ESP32)

"iluminacion": {  // (Escrito por Web, Leído por ESP32)

"horaEncendido": "06:00",

"horaApagado": "22:00"

}

}

}

Nuevo Desafío: Conflicto de Autoridad

Preocupación: ¿Qué pasa si el usuario local (Botón LCD) y el usuario remoto (Web) dan órdenes opuestas al mismo tiempo?

Solución de Arquitectura (Propuesta):

"La Nube es la Fuente de Verdad" (Cloud is the Source of Truth).

El ESP32 nunca cambia su estado directamente desde un botón. Flujo de

Botón Local: Usuario presiona "Ventilador ON" en el LCD. taskMenu

(ESP32) no cambia ventilador_encendido = true.

En su lugar, taskMenu escribe en Firebase (set("control/ventilacion", true)).

Flujo de Sincronización: 4. taskControlListener (ESP32) recibe el cambio (que él mismo causó) desde Firebase. 5. taskControlListener actualiza ventilador_encendido = true.

Resultado: La web se actualiza porque lee de la misma "fuente de verdad". Eliminando el conflicto.

## 3. Vistas de Arquitectura (Sistema de Control Indoor – Arquitectura v3.0 (ESP32 + FreeRTOS + Firebase dentro de la web))

La arquitectura actual (v3.0) se describe con algunas de las siguientes vistas, si embargo antes de ello se explica el uso de diversos patrones.

| Patrón / Técnica | Ejemplo / referencia en el código | Propósito / Por qué está ahí | Beneficios | Observaciones / Mejoras sugeridas |
|---|---|---|---|---|
| **Observer (Listener / Callback)** | Firebase.RTDB.beginStream(&fbdo, "/control") + streamCallback | Recibir notificaciones de cambios remotos en /control y reaccionar en tiempo real. | Reacción inmediata sin sondeo constante. | Bien implementado; cuidar manejo de errores y validar datos entrantes. |
| **Singleton (instancias globales compartidas)** | FirebaseData fbdo, fbdo_writer, FirebaseConfig config, FirebaseAuth auth, timeClient | Mantener una única instancia de objetos que representan recursos globales (Firebase, NTP). | Evita duplicar recursos y consumo; facilita control centralizado. | Documentar alcance global; para pruebas sería útil abstraer mediante interfaces. |
| **Active Object / Task-based Concurrency** | xTaskCreatePinnedToCore (vTaskReadSensors, …), vTaskActuatorControl | Encapsular trabajo concurrente en tareas periódicas independientes. | Paralelismo real con FreeRTOS y separación clara de responsabilidades. | Correcto; vigilar stacks y prioridades. Considerar watchdogs por tarea. |
| **Monitor / Mutex (Sincronización)** | xStateMutex, xFirebaseWriteMutex, uso de xSemaphoreTake / xSemaphoreGive | Proteger acceso al estado compartido y operaciones no reentrantes (Firebase). | Evita condiciones de carrera y corrupción de datos. | Buen uso con timeouts; revisar orden de adquisición para evitar deadlocks. Posible RAII si se migra a C++. |
| **Producer–Consumer (acoplamiento débil)** | vTaskReadSensors produce g_currentSoilMoisture; vTaskActuatorControl consume esos datos | Separar productor (sensores) y consumidor (actuadores) mediante estado compartido protegido. | Pipeline simple de sensores → acción. | Para mayor robustez usar colas (FreeRTOS queue) para enviar eventos o históricos. |
| **Snapshot / Copy-on-read** | Copia local de estado dentro del mutex en vTaskActuatorControl (ej. localSoil, localUmbral) | Minimizar tiempo dentro del mutex copiando el estado localmente. | Menor contención y tareas más responsivas. | Excelente práctica; documentar invariantes del estado copiado. |

| | | | | |
|---|---|---|---|---|
| **Batching / Bulk Update** | json_estado.set(...) + Firebase.RTDB.updateNode(&fbdo_writer, "/estado", &json_estado) | Agrupar múltiples cambios en una sola operación de red. | Menor overhead de red y estado consistente en DB. | Vigilar tamaño del JSON y gestionar reintentos si hay fallos parciales. |
| **Retry / Backoff** | initNTP() con intentos limitados; initFirebase() con bucles de espera | Intentar conectar servicios externos hasta un límite. | Robustez ante fallos temporales de red. | Considerar backoff exponencial en lugar de reintentos lineales. Añadir logs detallados. |
| **Fail-fast / Restart en error fatal** | ESP.restart() en fallas críticas (mutex no creado, NTP no sincronizado) | Recuperarse mediante reinicio cuando el sistema no puede operar correctamente. | Evita estados incoherentes; útil en sistemas embebidos. | Documentar razones de reinicios y, si es posible, registrarlas en EEPROM o RTC RAM. |
| **Separation of Concerns / Single Responsibility** | Tareas separadas: vTaskReadSensors para sensado; vTaskActuatorControl para actuadores | Mantener módulos con responsabilidades claras. | Código más mantenible, testeable y modular. | Extraer lógica de Firebase/NTP a módulos más pequeños para mayor claridad. |
| **Guarded Suspension (espera con timeout)** | xSemaphoreTake(xFirebaseWriteMutex, pdMS_TO_TICKS(1500)) | Esperar un recurso sin bloquear indefinidamente. | Evita deadlocks y tareas congeladas. | Homogeneizar los valores de timeout entre tareas. |
| **Hardware Abstraction (nivel básico)** | Definición de pines y macros (RELAY_ON, RELAY_OFF, SENSOR_SECO_VAL) | Separar detalles de hardware en constantes para evitar mezclarlos con la lógica. | Facilita cambios de hardware sin modificar la lógica. | Encapsular en funciones o structs para facilitar portabilidad a otras placas. |

## 3.1 PATRONES.

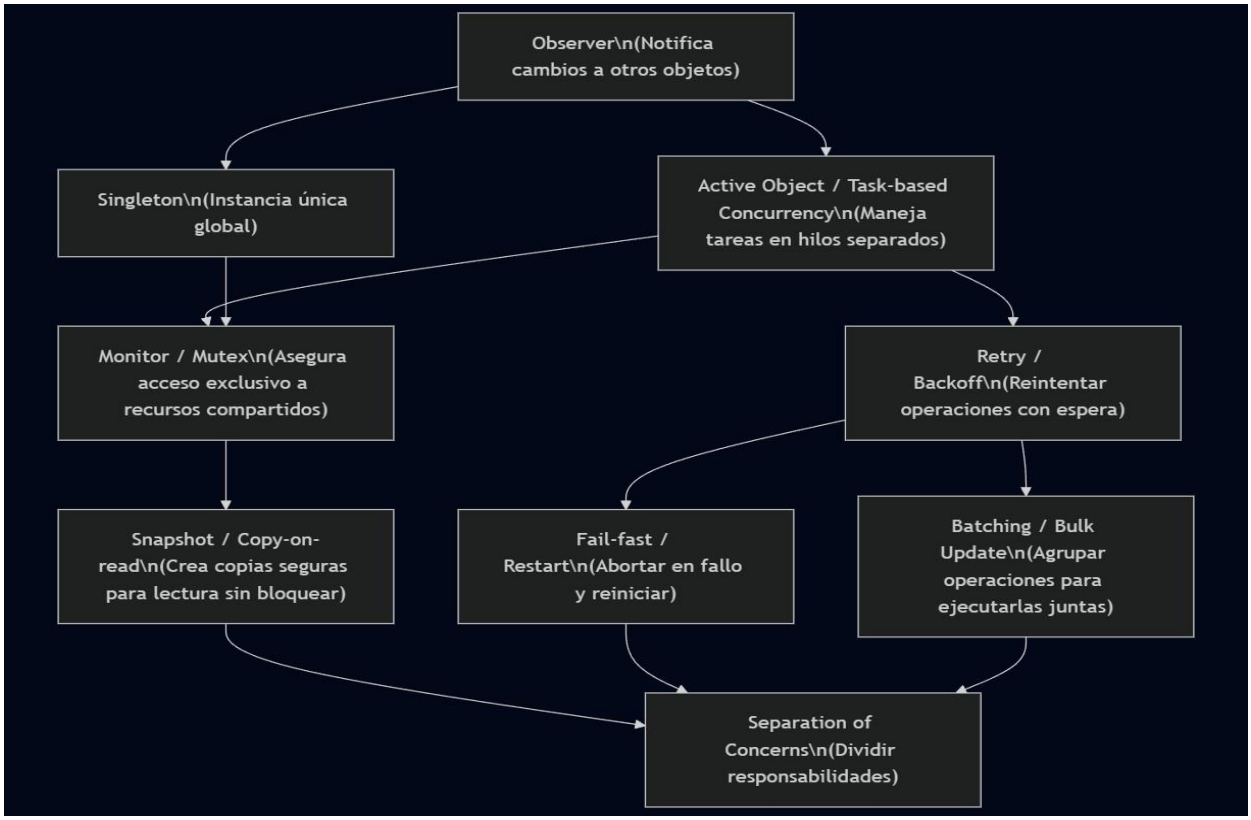



*Ilustración 2 Diagrama de patrones Como es el  funcionamiento y comunicación entre los patrones, úesto que todos deben llegar a cubrir las responsabilidades.*

VISTA GENERAL.

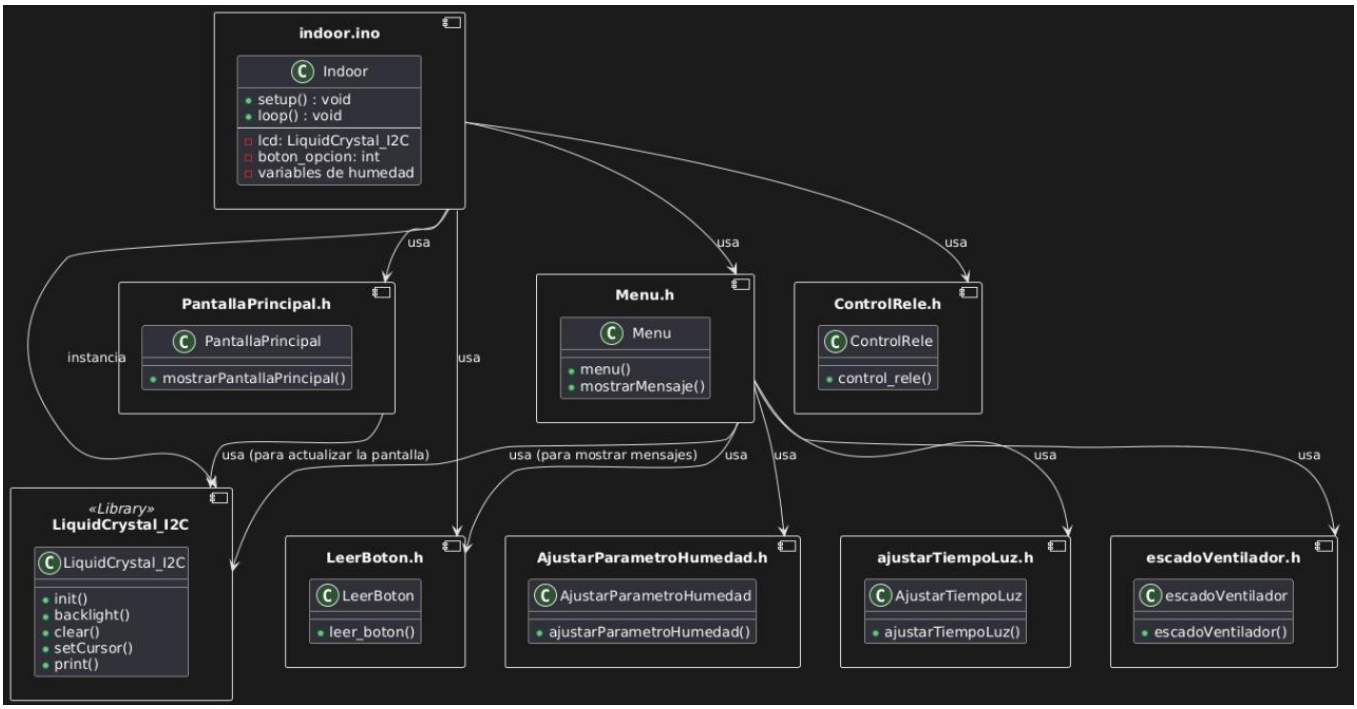



*Ilustración 3 Diagrama general V1.0 indoorState.ino(Prueba de Concepto)Bucle único.*

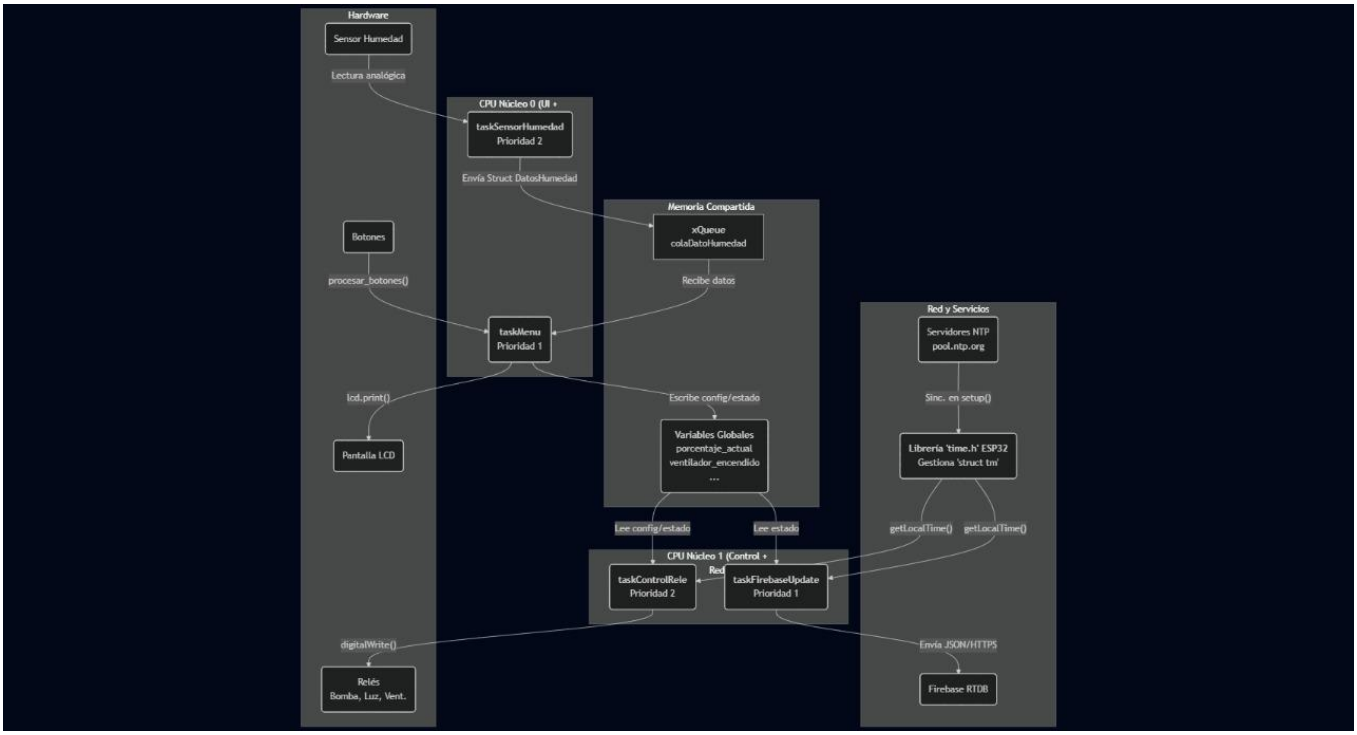



*Ilustración 4 Diagrama general V2.0 indoorFreertos2.ino(Concurrencia + Nube)Añade FreeRTOS y Firebase..*

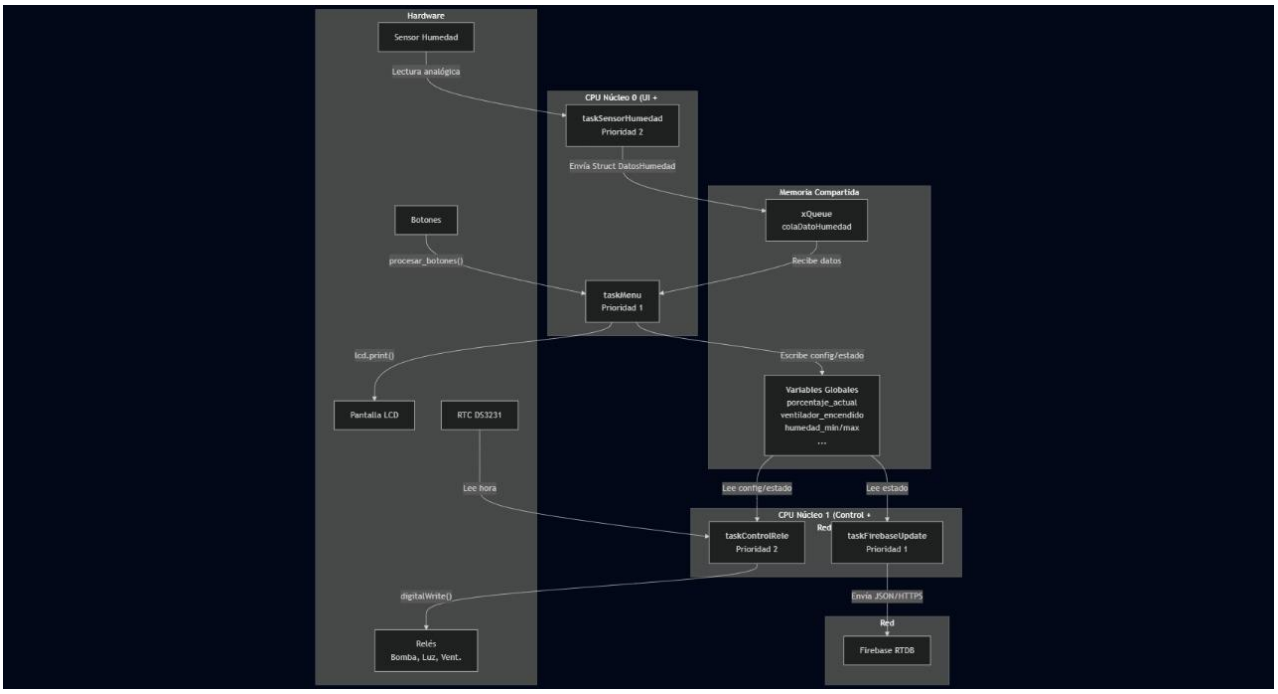



*Ilustración 5 Diagrama general V2.1 indoorFreertos3.ino(Refinamiento + Costo)Optimiza el hardware y la lógica de tiempo..*

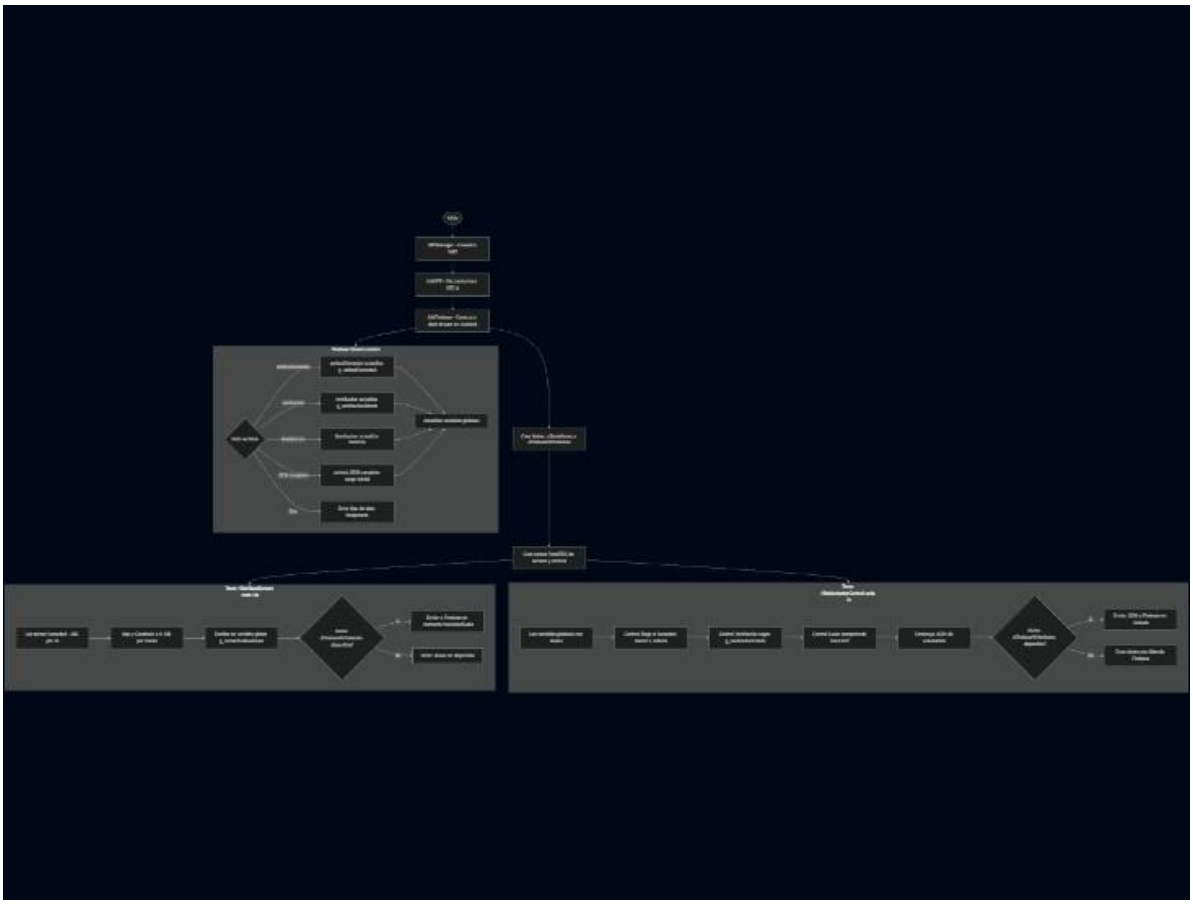
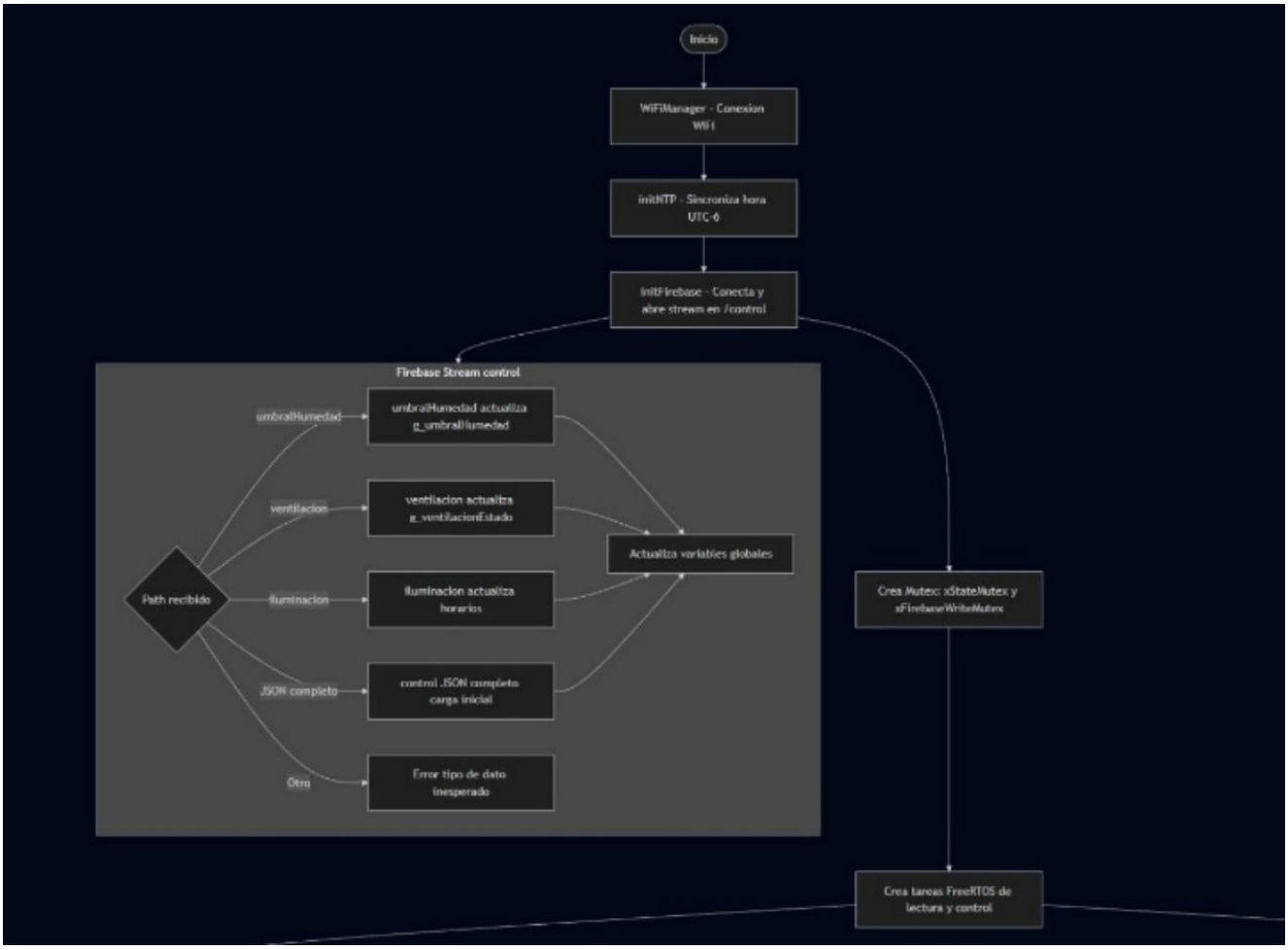
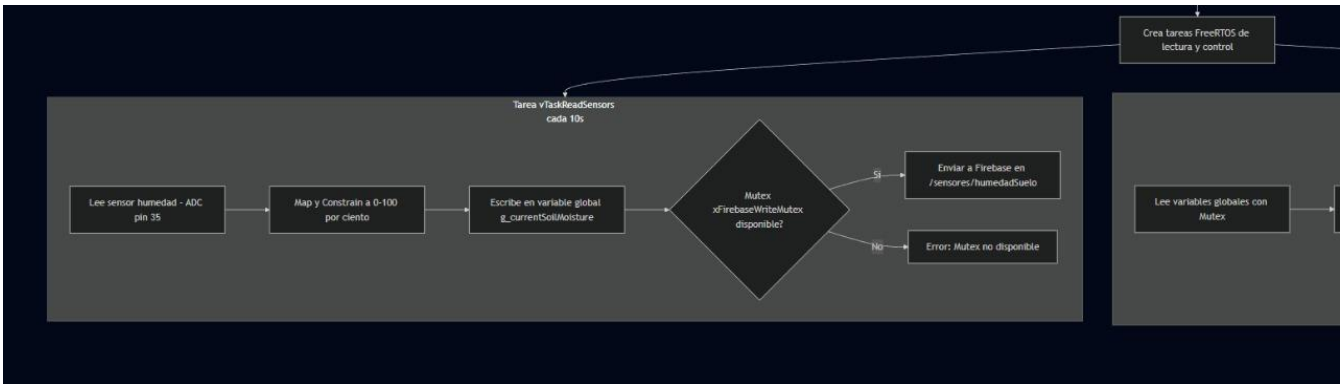
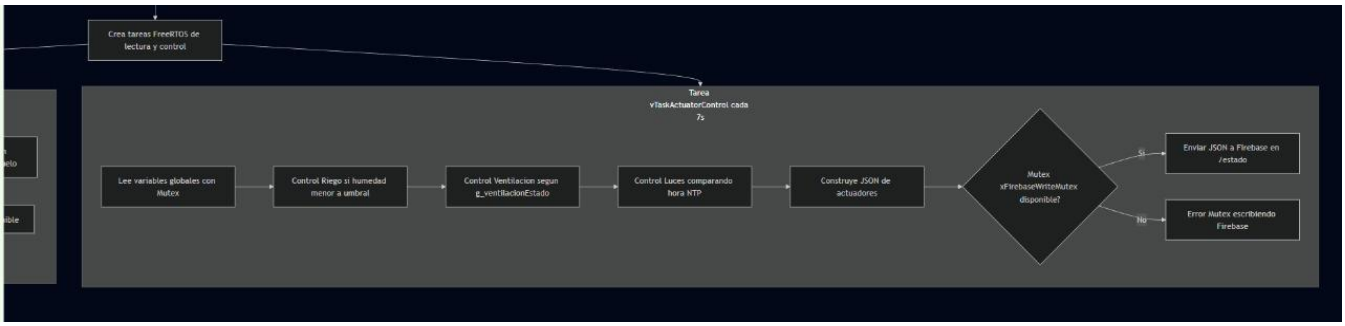


*Ilustración 6 Diagrama general del Sistema Indoor V3.0 indoorFreertos4.ino(Refinamiento y Costo)Optimiza el hardware, la lógica de tiempo y el uso de un sistema WEB.*

.


Inicio
WiFiManager - Conexion WiFi
initNTP - Sincroniza hora UTC-6
initFirebase - Conecta y abre stream en /control
Firebase Stream control
umbralHumedad
umbralHumedad actualiza g_umbralHumedad
ventilacion
ventilacion actualiza g_ventilacionEstado
Path recibido
iluminacion
iluminacion actualiza horarios
Actualiza variables globales
JSON completo
control JSON completo carga inicial
Otro
Error tipo de dato inesperado
Crea Mutex: xStateMutex y xFirebaseWriteMutex
Crea tareas FreeRTOS de lectura y control


Crea tareas FreeRTOS de lectura y control
Tarea vTaskReadSensors cada 10s
Lee sensor humedad - ADC pin 35
Map y Constrain a 0-100 por ciento
Escribe en variable global g_currentSoilMoisture
Mutex xFirebaseWriteMutex disponible?
Si
Enviar a Firebase en /sensores/humedadSuelo
No
Error: Mutex no disponible
Lee variables globales con Mutex




El sistema funciona a partir del inicio de sesión, se crea la red cautiva, se genera la comunicación con Firebase y FreeRTOS, por parte de Firebase existe el recibimiento y actualización de los datos del sistema, por parte de FreeRTOS se crean las tareas por tiempo para el correcto funcionamiento del sistema.

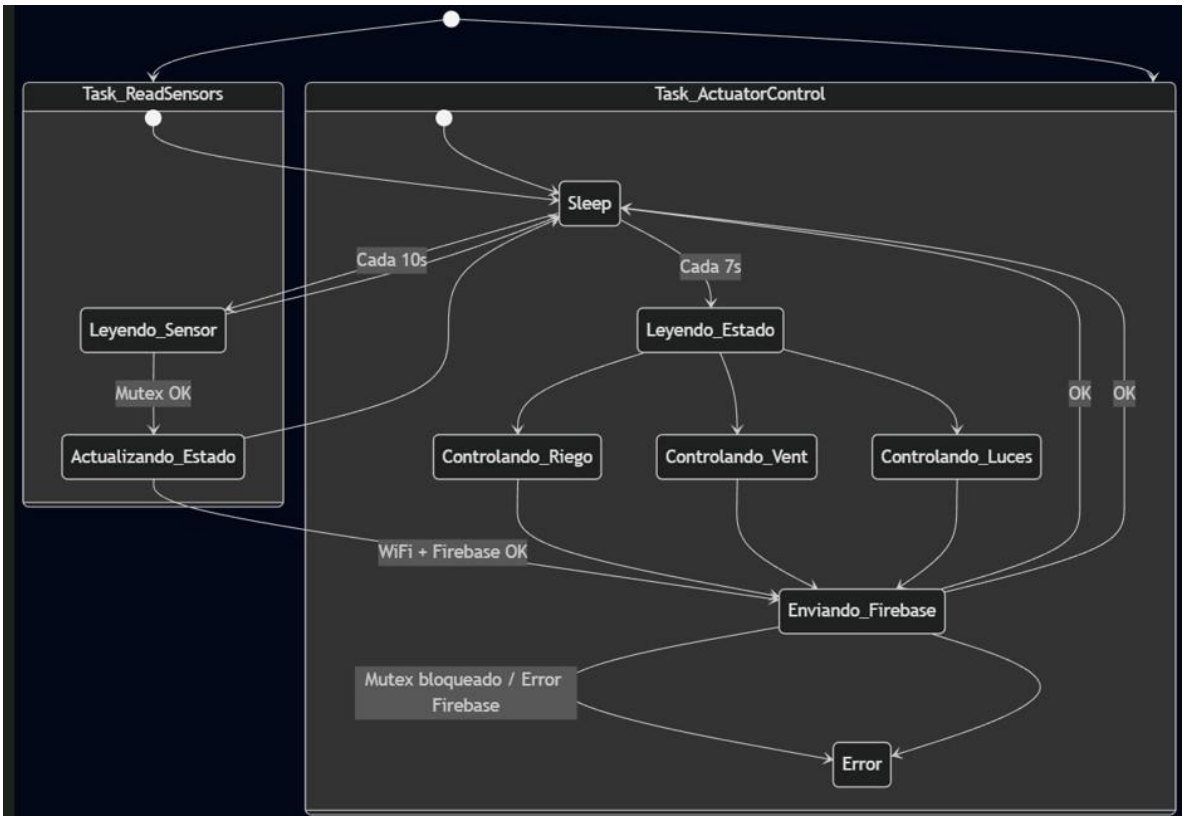



*Ilustración 7 Diagrama de estados de las tareas. Correcto funcionamiento del sistema Indoor por medio de las tareas, puesto que todas deben cumplir un correcto funcionamiento, pero sin saturar los recursos.*

El sistema se divide entre la lectura de los sensores y el control de los actuadores, desde el lado de la lectura de estos se centra en la lectura y en la actualización de estados, por otra parte los actuadores deben funcionar de tal modo que envían los datos a Firebase sin saturar los recursos, si el actuador no es necesario su funcionamiento, este se manda a Hibernar.

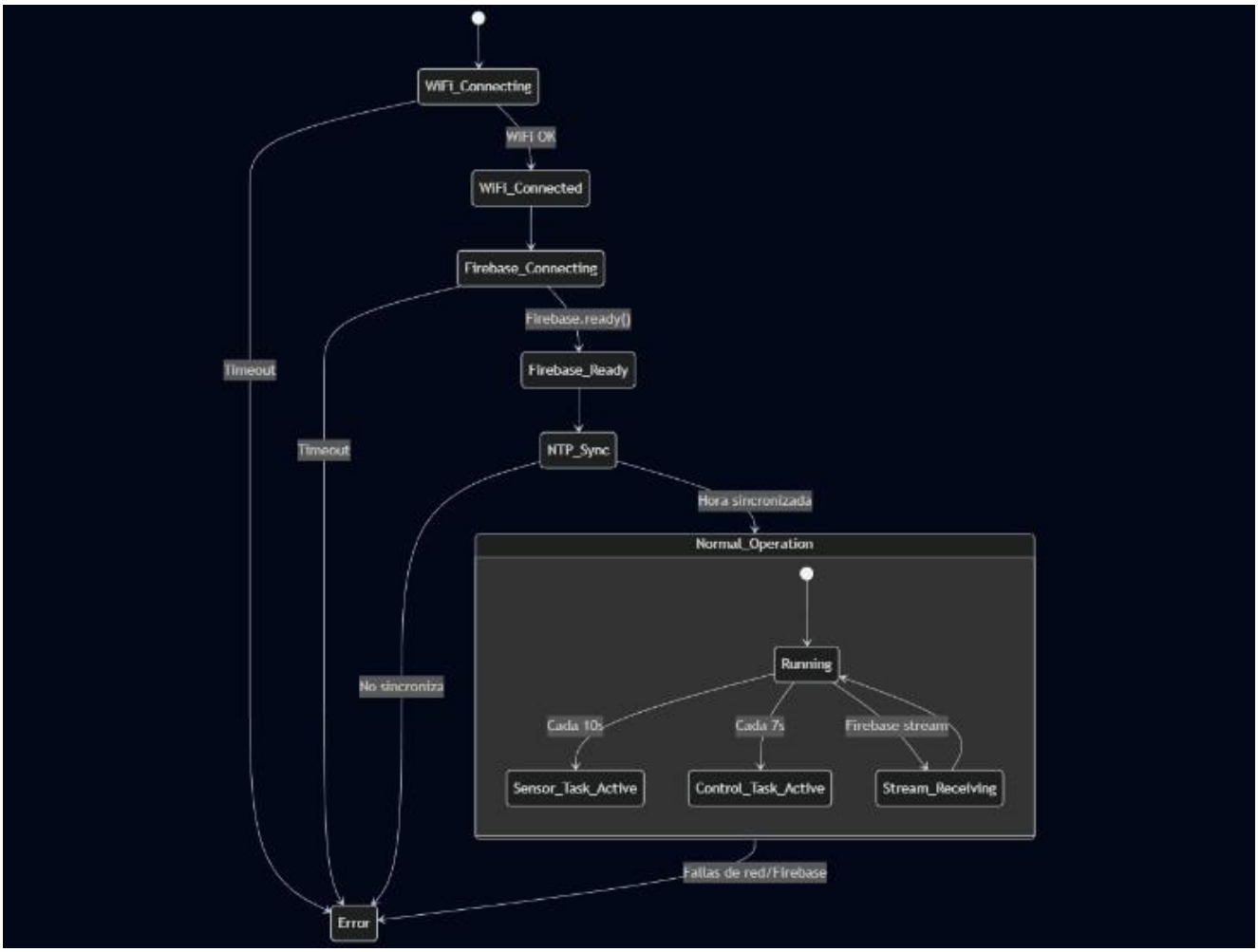



*Ilustración 8 Diagrama de estados del wifi dentro del sistema Indoor*

El usuario crea una conexión wifi cautiva para la comunicación con Firebase, posteriormente entra al paquete Operación Normal que tiene dentro el funcionamiento del sistema, se muestran posibles timeout que pueden resultar en la no sincronidad.

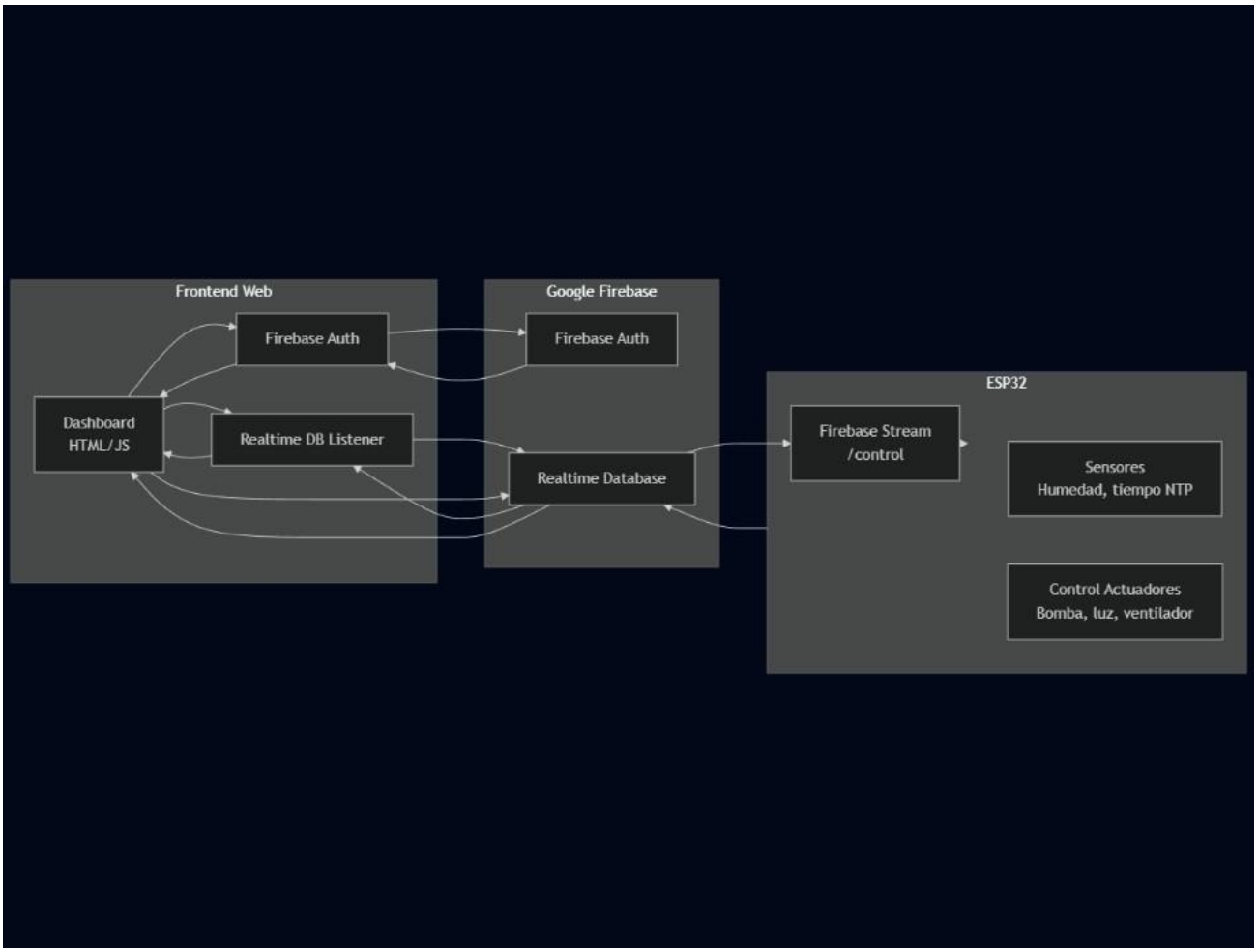



*Ilustración 9 Diagrama de arquitectura Como funciona la arquitectura del sistema, Frontend -> Comunicación con Firebase -> Comunicación bidireccional con ESP32.*

El sistema a partir del HTML implementando JavaScript tiene comunicación con Firebase y el escucha de la Base de Datos, implementan comunicación con Google Firebase por medio de la autenticación y el aspecto de tiempo real, creando comunicación con el ESP32.

3.2 Vista 1: Concurrencia (Tareas RTOS)

Esta vista aborda las preocupaciones C-1, C-2 y C-3. La arquitectura descompone el sistema en tareas (Tasks) de FreeRTOS independientes.

Modelo 1: Diagrama de Tareas, Núcleos y Flujo de Datos

Nota: Observa que el HW_RTC ha sido eliminado. La hora ahora proviene de NTP, es gestionada por la librería time.h del ESP32, y consumida por las tareas taskControlRele y taskFirebaseUpdate. Fragmento de código

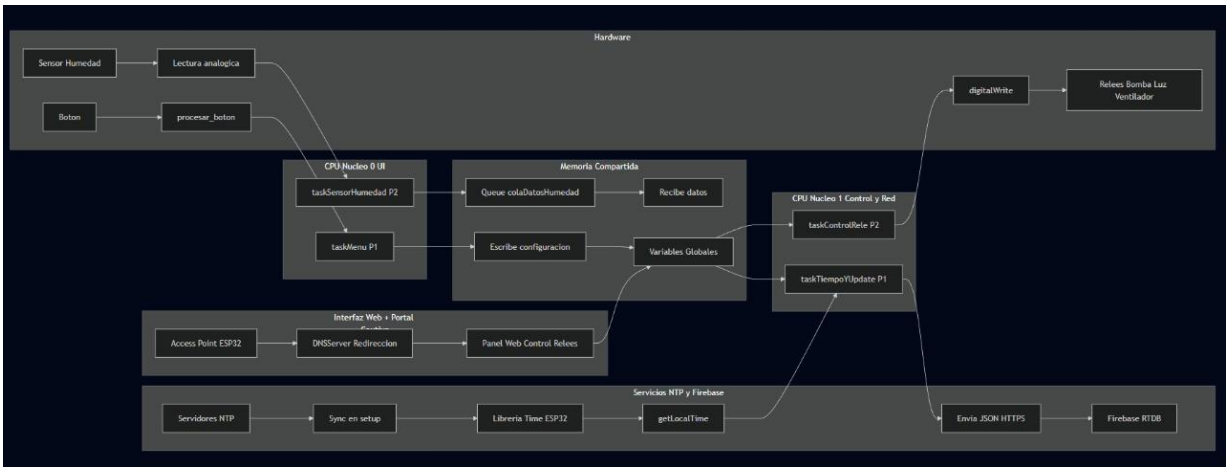



*Ilustración 10 Comunicación dentro del sistema ESP32*

Este diagrama muestra la comunicación del sistema, desde el modulo de hardware se ve los componentes físicos que componen el sistema Indoor, las tareas que pueden realizar cada uno de los modulos que posee el ESP32 , como funciona la interfaz web por medio de una UI.

3.3 Vista 2: Máquina de Estados de UI

Esta vista cambia con la v2.1, ya que la lógica de la interfaz de usuario está completamente desacoplada. Esta versión retira el modulo RTC para abaratar costos, así como el hardware LCD para cambiarlo por una UI de usuario en un entorno web.

Estos cambios generan un menor costo en la implementación del sistema, nulos bloqueos, un simple botón para el reseteo del wifi, la implementación de una red cautiva.

Modelo 2: Diagrama de Estados de taskMenu.

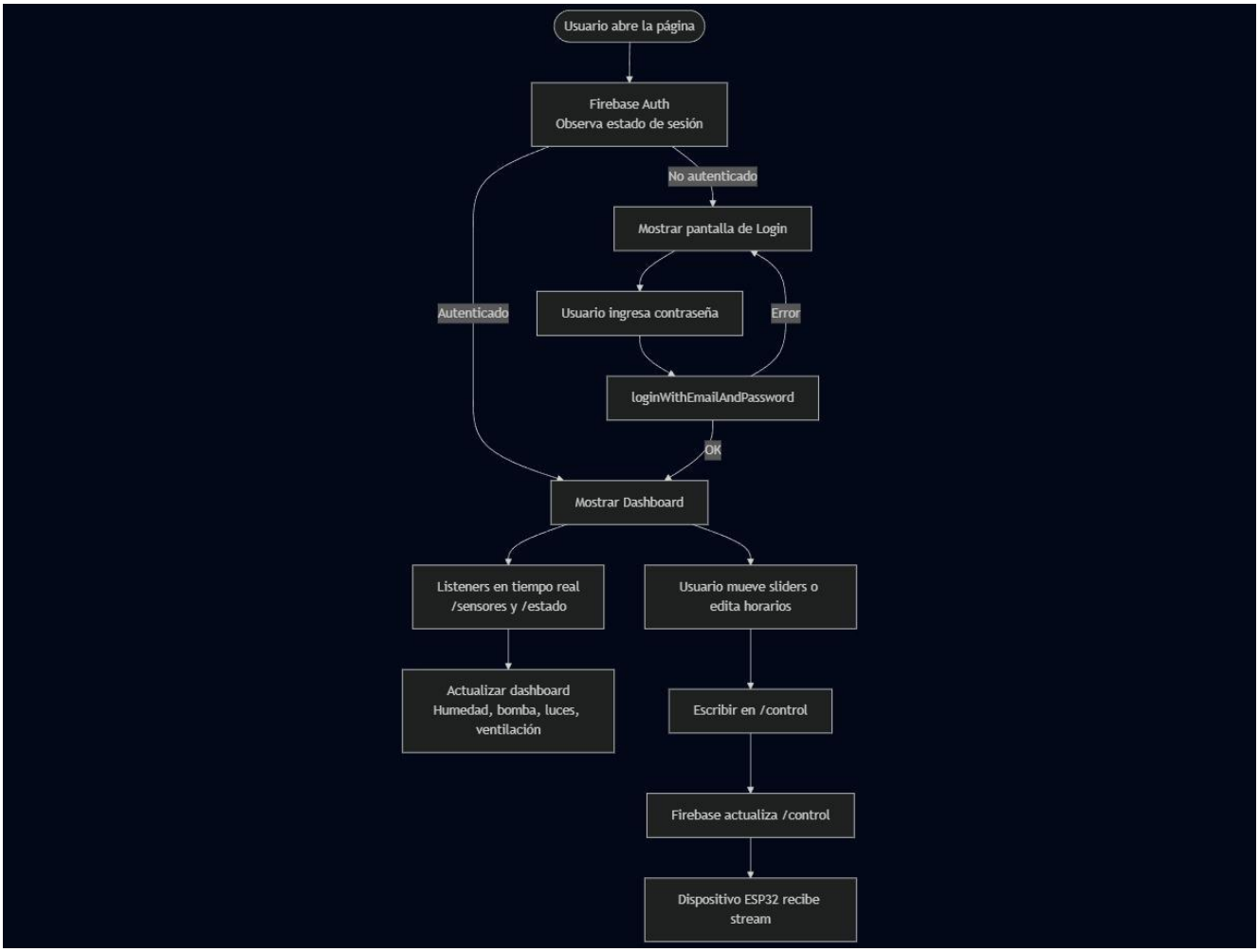



*Ilustración 11 Lógica de la página web.*

El usuario por medio de la UI de la pagina web se autentica, lo que a su vez inicia Firebase por medio del inicio de sesión, mandando el login, el usuario ingresa la contraseña para después llamar el dashboard que da la opción de ver los datos en tiempo real o modificar los datos del sistema Indoor para actualizar el correcto funcionamiento del ESP32.

3.4 Vista 3: Vista Lógica

Como funciona FIREBASE dentro del sistema:

1. Funcionamiento de Firebase en el Sistema

1.1. Propósito

Firebase Realtime Database (RTDB) es la plataforma utilizada para sincronizar el estado del invernadero en tiempo real, recibir órdenes desde la aplicación móvil o dashboard web, subir continuamente las mediciones de sensores y actualizar el estado de actuadores. Firebase permite comunicación bidireccional y en tiempo real entre el ESP32 y la base de datos.

1.2. Mecanismos principales

A. Autenticación
El ESP32 utiliza un usuario y contraseña dedicados dentro de Firebase Authentication con email del sistema, password del sistema, API Key para habilitar acceso al proyecto y URL de la base de datos que identifica el RTDB correspondiente. FirebaseAuth y FirebaseConfig inicializan auth.user.email, auth.user.password, config.api_key, config.host y config.database_url. Esto habilita tokens válidos para lectura y escritura.

B. Conexión y Modo Reconnect
El sistema utiliza Firebase.begin() y Firebase.reconnectWiFi(true), lo que mantiene la conexión activa y almacena los tokens para evitar reconexiones constantes.

1.3. Comunicación con Firebase
A. Escritura en RTDB
El sistema sube humedad del suelo en la ruta /sensores/humedadSuelo y el estado de la bomba, ventilador y luces en /estado. La escritura está protegida con un mutex xFirebaseWriteMutex para evitar conflictos entre tareas concurrentes.
Ejemplo de escritura: Firebase.RTDB.setFloat(&fbdo_writer, "/sensores/humedadSuelo", humedad) y Firebase.RTDB.updateNode(&fbdo_writer, "/estado", &json_estado). Se usan objetos globales para evitar fragmentación de memoria en el ESP32.

B. Lectura mediante Streaming
El ESP32 mantiene un stream persistente con Firebase mediante Firebase.RTDB.beginStream(&fbdo, "/control"). Esto permite recibir en tiempo real actualizaciones de umbralHumedad, ventilacion, iluminacion y toda la configuración inicial. Cada cambio ejecuta streamCallback(FirebaseStream data), que actualiza las variables globales protegidas por mutex como g_umbralHumedad, g_ventilacionEstado, g_horaEncendido y g_horaApagado. El stream garantiza que el ESP32 mantenga su configuración sincronizada con Firebase.

C. Manejo de Timeouts
Si el stream se interrumpe se ejecuta streamTimeoutCallback(true) y Firebase se reconecta automáticamente.

1.4. Protección mediante Mutex
Existen dos mutex: xStateMutex que protege las variables globales del estado del sistema y xFirebaseWriteMutex que protege las escrituras hacia Firebase.

2. Funcionamiento del WiFi cautivo (WiFiManager)

2.1. Propósito
El sistema debe conectarse a la red WiFi del usuario sin necesidad de reprogramar el ESP32. Para esto se implementa un portal cautivo mediante la librería WiFiManager.

2.2. Flujo de Funcionamiento
A. Arranque del Dispositivo

Cuando el ESP32 arranca, WiFiManager intenta conectarse a la última red WiFi conocida. Si falla o si el usuario presiona el botón de reset WiFi, se borran las credenciales con wm.resetSettings().

B. Creación del Punto de Acceso
Si no hay conexión válida, WiFiManager crea un punto de acceso local con SSID Indoor, seguridad opcional abierta y un DHCP interno que asigna IP a los teléfonos conectados. El ESP32 funciona como router temporal.

C. Portal Cautivo
Cuando un usuario se conecta al AP, WiFiManager abre un portal cautivo. El usuario abre cualquier página y el dispositivo lo redirige automáticamente a la interfaz de configuración. El usuario selecciona la red WiFi de su casa, introduce la contraseña, y el ESP32 almacena las credenciales en la memoria NVS. La función clave wm.autoConnect("Indoor") crea el AP, abre el portal, guarda credenciales y reconecta automáticamente.

D. Conexión Automática
Una vez configurada la red, en futuros arranques el ESP32 se conectará automáticamente y únicamente si la red falla volverá a abrir el portal cautivo.

2.3. Botón de Reset WiFi
El ESP32 incluye un botón físico que al ser presionado ejecuta wm.resetSettings(), restableciendo credenciales y activando el portal cautivo.

1. Arquitectura de Red del Sistema WiFi + Captive Portal

El dispositivo utiliza dos modos WiFi.
Modo Access Point AP para configuración inicial, donde el ESP32 crea una red propia y un servidor DNS que redirige todo a la IP local, y un servidor HTTP que muestra la página de configuración. También incluye un servidor DHCP. Flujo: el ESP32 inicia en modo AP, crea la red Invernadero Setup, inicia el DNS hijacker, el servidor HTTP, el usuario se conecta, introduce SSID y contraseña, el ESP32 guarda en NVS y reinicia en modo Station.

2. Arquitectura de Red en Modo Estación

Cuando el dispositivo ya está configurado, activa el modo Station, obtiene IP por DHCP del router, mantiene la conexión con Firebase mediante REST y Streaming, y sincroniza la hora con NTP.

3. Arquitectura de Comunicación con Firebase El dispositivo utiliza comunicación híbrida.

REST para envío de datos de sensores y estados, con respuestas HTTP síncronas.
Streaming Listener para recibir actualizaciones instantáneas desde la ruta /control, eliminando necesidad de polling y permitiendo control en tiempo real.
El ESP32 utiliza token API o claves de autenticación del RTDB en modo seguro.

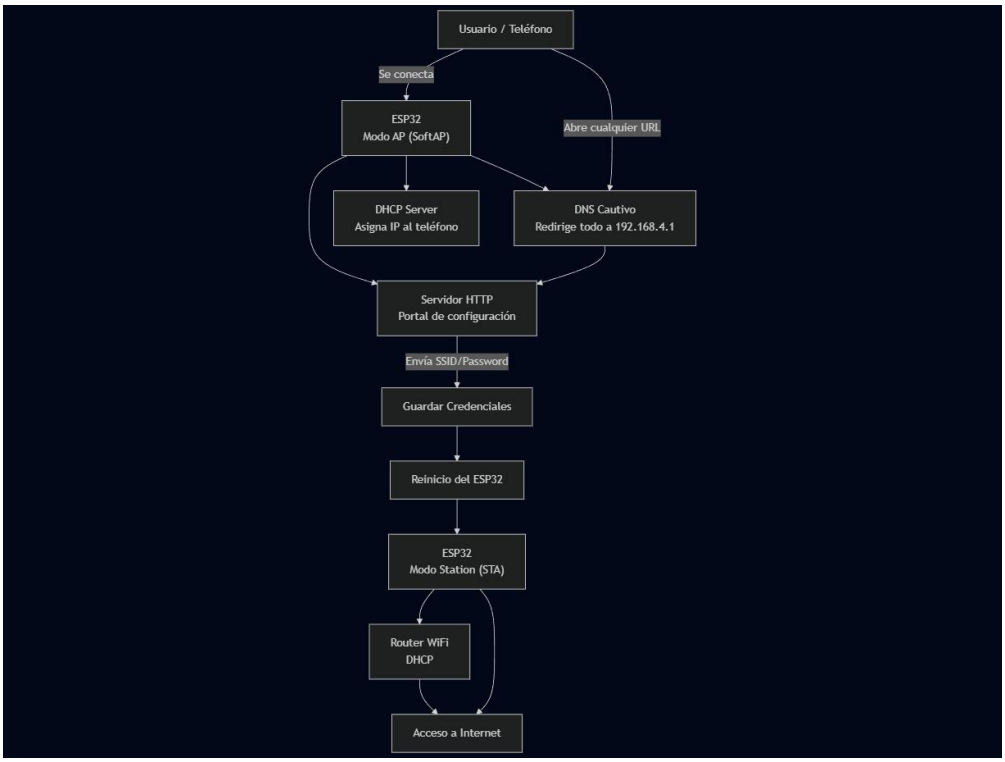



*Ilustración 12 Arquitectura wifi cautivo.*

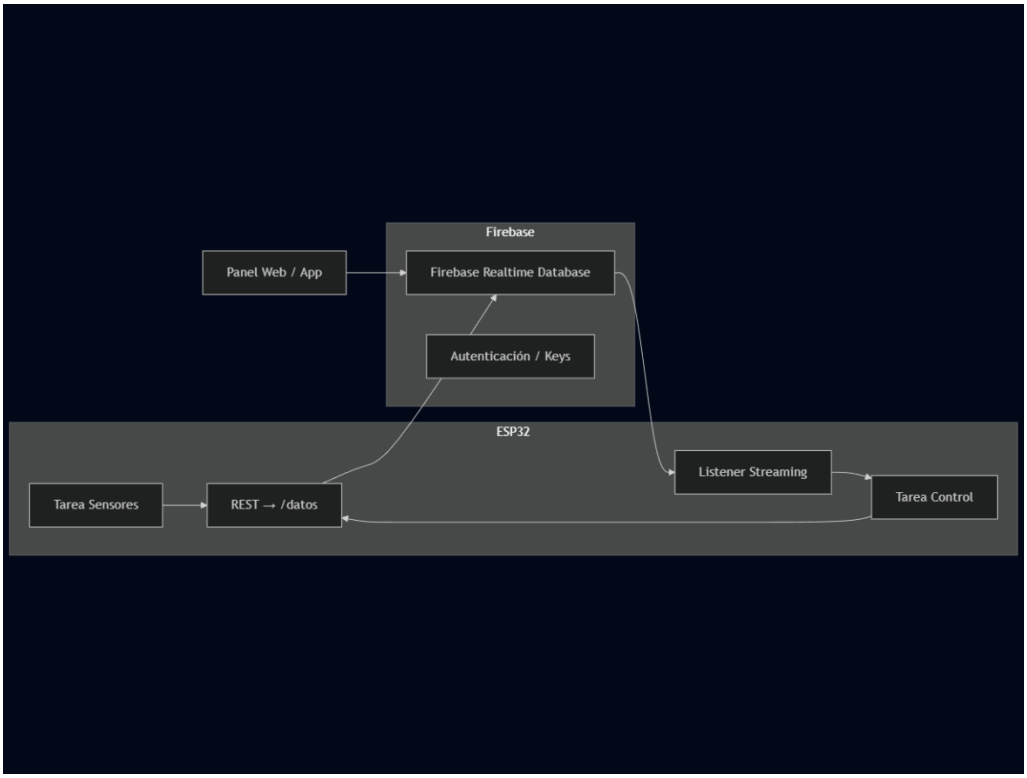



*Ilustración 13 Arquitectura Firebase.*

# 4. VISTA FISICA

‘La Vista Física describe cómo se distribuyen e interconectan los elementos hardware que componen el sistema Indoor, así como la infraestructura necesaria para su operación. Esta vista se concentra en los componentes físicos reales (sensores, actuadores, red, microcontrolador, servicios en la nube) y cómo se relacionan entre sí para permitir la ejecución del software descrito en las demás vistas.

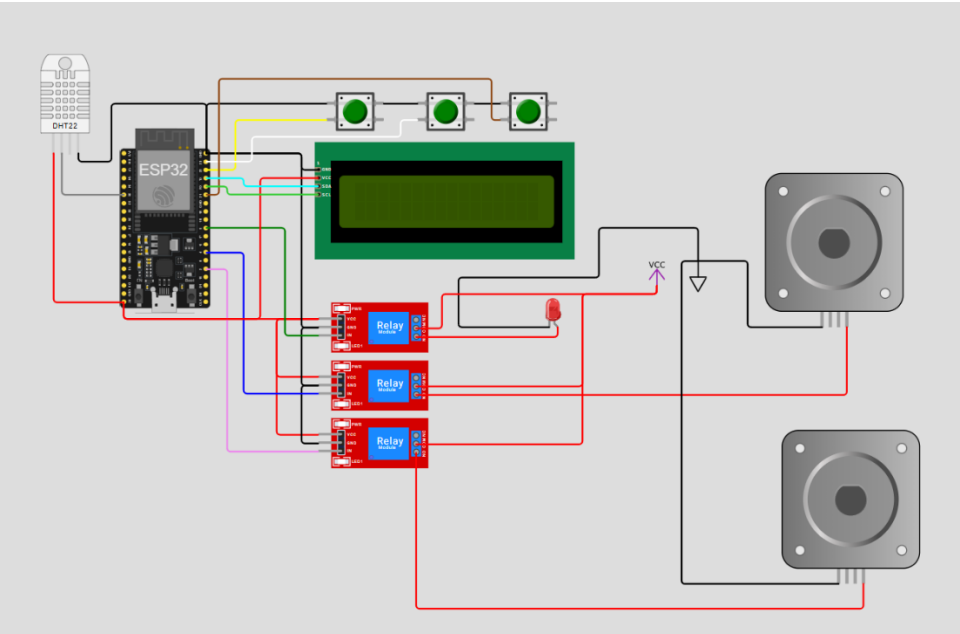



*Ilustración 14 Versiondel Sistema fisico anterior.*

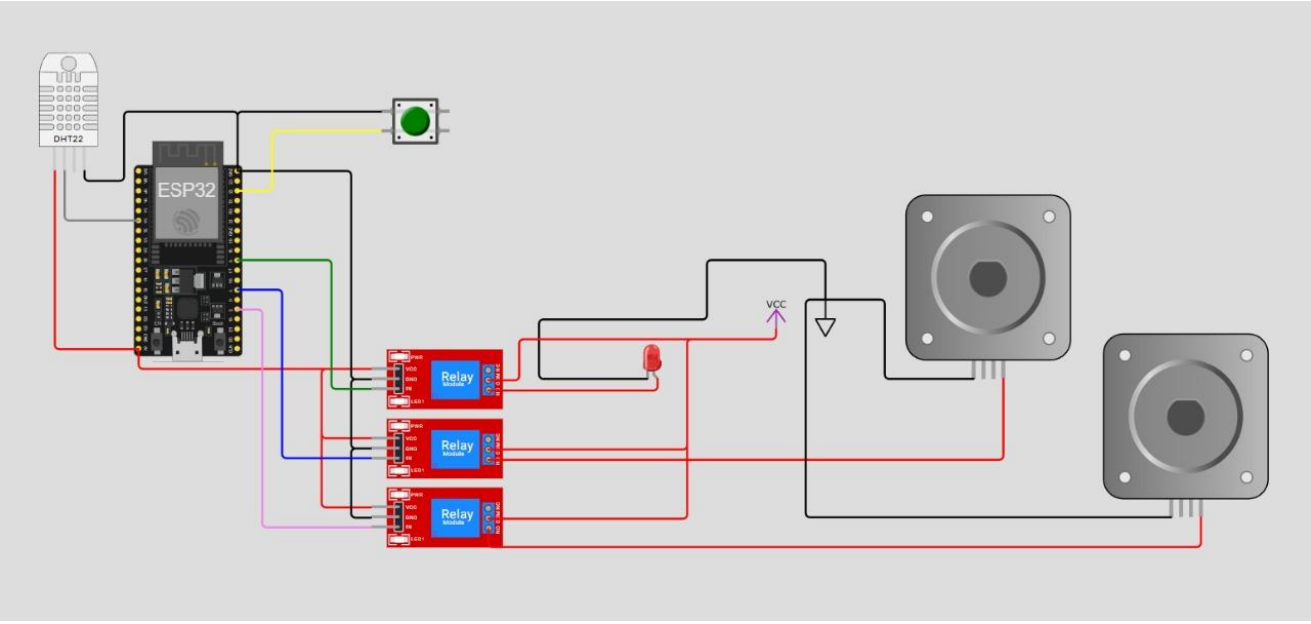
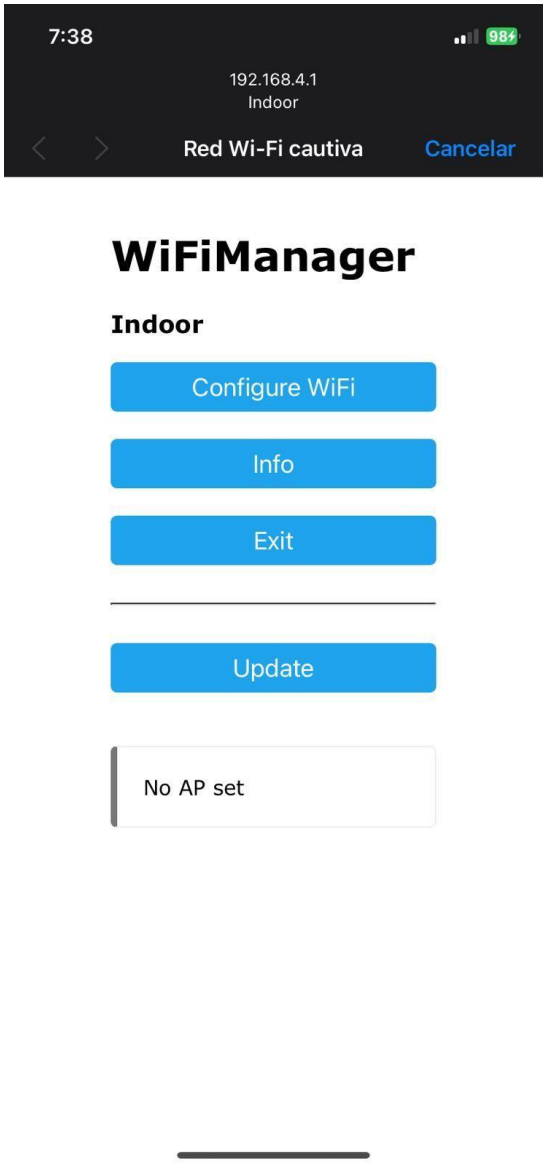



*Ilustración 15 Implementación física del sistema dentro de un simulador, los reles, el ESP32.*

Los cambios físicos mas notorios dentro del sistema físico es la eliminación de la pantalla LCD, 2 botones que se retiran para no interferir en el código, así dejando solo 1 botón que tiene como función el reseteo del wifi.

El diagrama 13 de la parte superior muestra la comunicación del ESP32 con los reles que dan las instrucciones al sistema de iluminación y riego, asi mismo se muestra un botón e color verde, este tiene como función el reinicio del sistema Wifi.

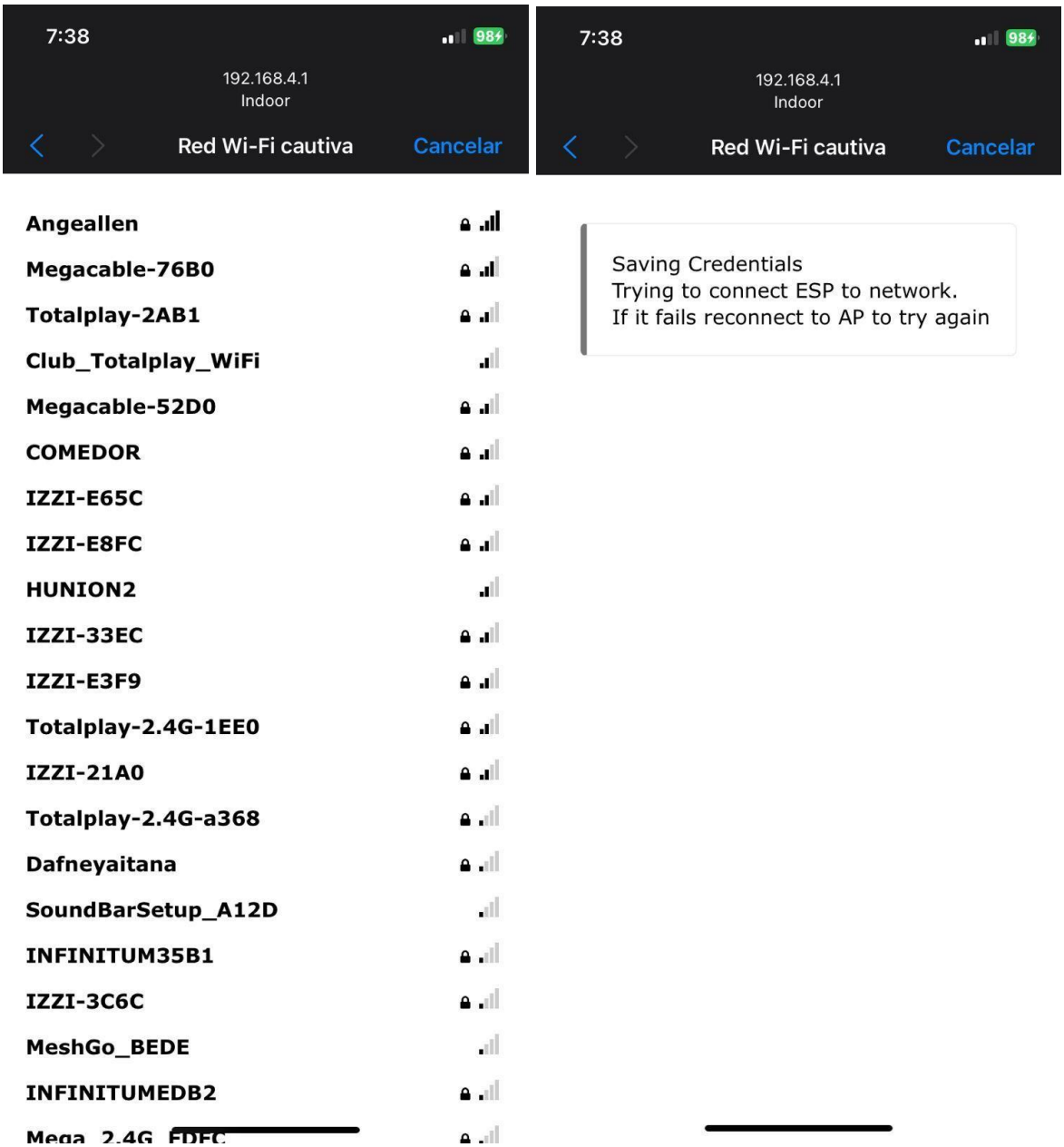



*Ilustración 16 Las imágenes de la parte superior plasman la creación de una red wifi cautiva, puesto que el sistema debe conectarse a la red WiFi del usuario sin necesidad de reprogramar el ESP32. Para esto se implementa un portal cautivo mediante la librería WiFiM*

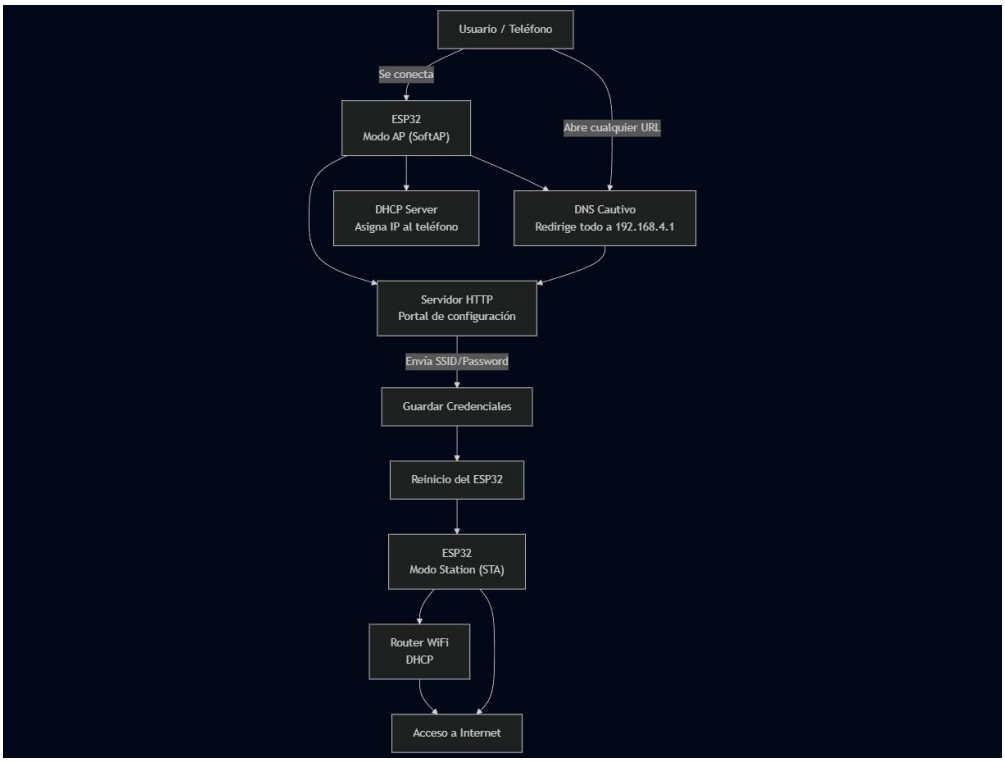



*Ilustración 17 Arquitectura wifi cautivo.*

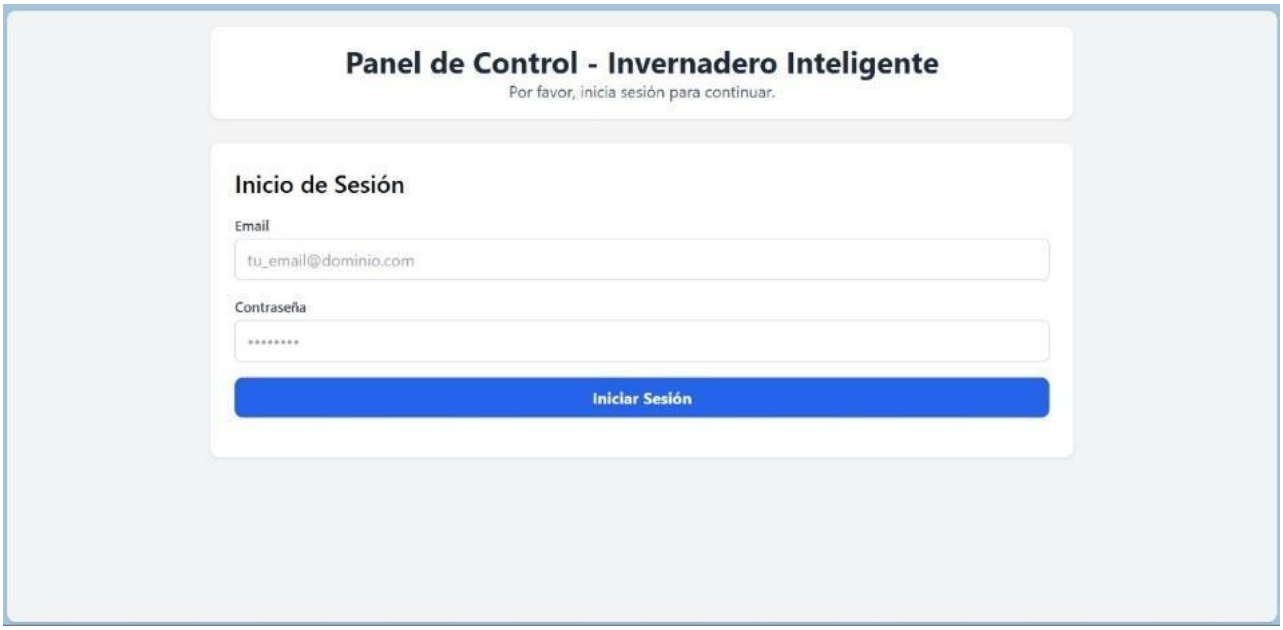



*Ilustración 18 Inicio de sesion. Por medio de una interfaz web para ingresar al sistema web del control Indoor.*

Esto es el primer paso para generar la comunicación dentro del sistema, solicitando las credenciales para dar paso al dashboard.

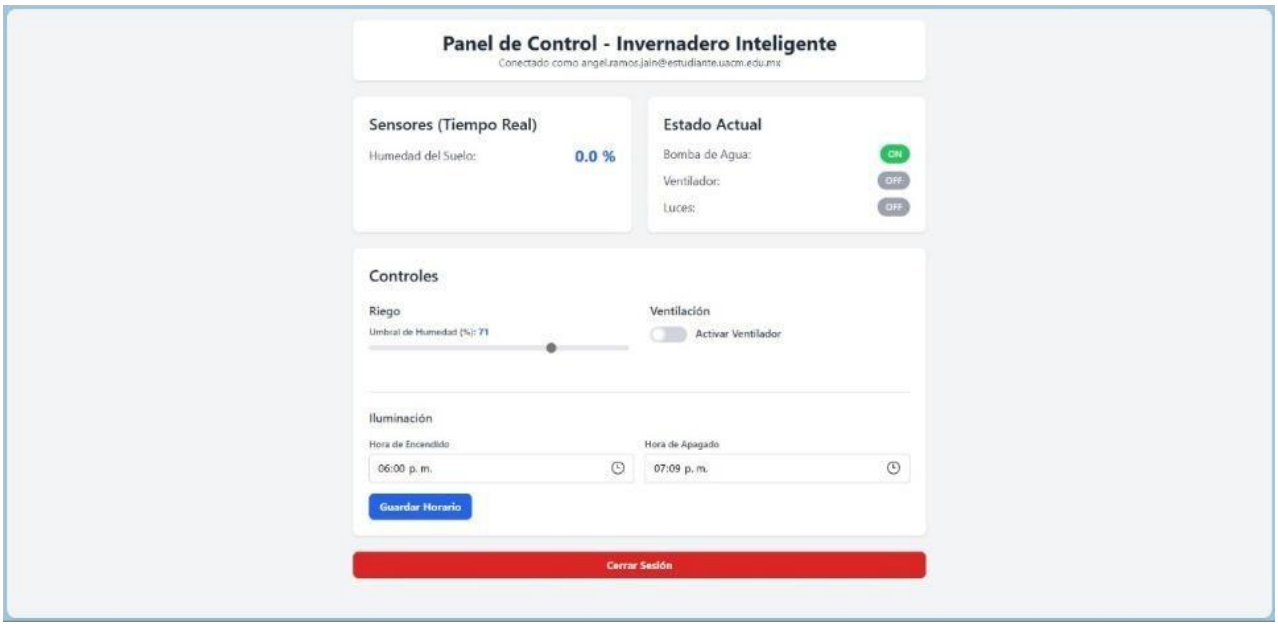



*Ilustración 19 Interfaz gráfica del control del sistema Indoor, mostrando todas las opciones que existen en el sistema.*

El dashboard muestra los sensores en funcionamiento, el estado de la bomba, la ventilación, asó como las luces, las opciones del encendido de riego, la ventilación y ajuste de la iluminación.

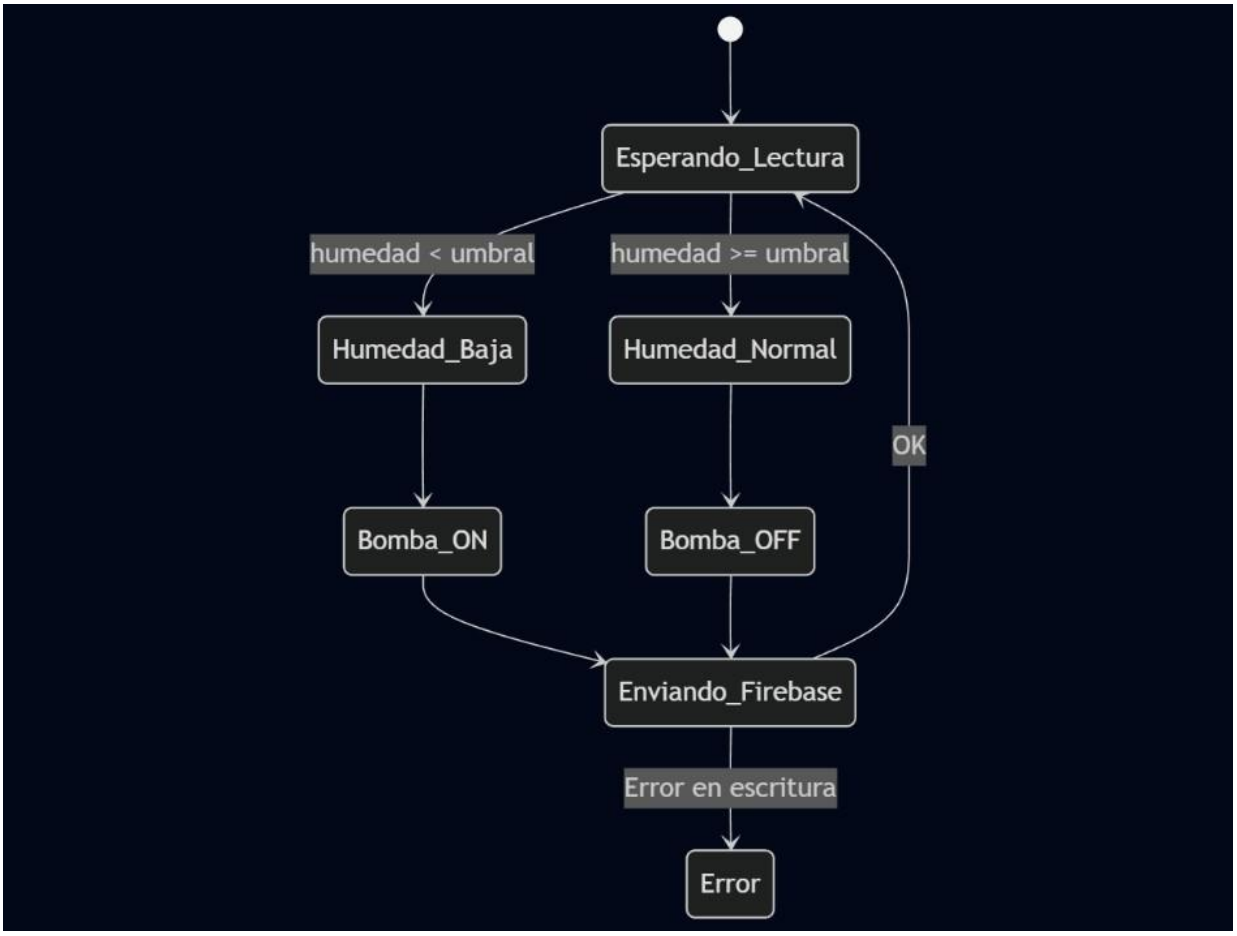



*Ilustración 20 Control de riego Diagrama de estados del sistema de riego dentro del sistema Indoor, comparando el humbral de humedad para saber si se prende o permanece apagada la bomba de agua para transmitir esos registros a Firebase*

# 5. Relaciones y Racionalización

## 5.1. Correspondencias entre Vistas

- (C-1) Tarea-a-Hardware: La tarea taskControlRele (Vista de Concurrencia) es responsable de operar los [HW Relés] (Vista Física).
- (C-2) Tarea-a-Flujo: La tarea taskSensorHumedad (Vista de Concurrencia) es la *fuente* del flujo de datos "Lectura analógica raw" (Vista de Flujo de Datos).
- (C-3) Flujo-a-Hardware: El flujo "String JSON vía HTTPS" (Vista de Flujo de Datos) depende del periférico "Wi-Fi" (Vista Física) para alcanzar su destino.

## 5.2. Inconsistencias Conocidas

- IN-1: Acceso concurrente a variables globales sin protección.
    - Descripción: Múltiples tareas (taskMenu, taskControlRele, taskFirebaseUpdate) leen y escriben en variables globales (ej. porcentaje_actual, ventilador_encendido) que residen en memoria compartida.
    - Problema: Existe un riesgo teórico de *condición de carrera* (race condition), donde una tarea podría leer un valor parcialmente escrito por otra tarea en el otro núcleo.
    - Racionalización (ISO 5.8): Esta inconsistencia se registra. Se considera de bajo riesgo en la implementación actual porque las operaciones son atómicas o muy rápidas y los *delays* de las tareas son largos (50ms, 500ms, 60000ms), haciendo una colisión improbable. Una arquitectura más robusta implementaría un mutex (mecanismo de exclusión mutua) para proteger el acceso a estas variables.

## 5.3. Racionalización de Decisiones

- Decisión R-1: Migración de Patrón State (Single-Loop) a FreeRTOS (Multitarea).
    - Decisión: El sistema fue rediseñado, abandonando la arquitectura original basada en un Patrón State (documentada en Diseño de Software (Indoor).pdf ) que se ejecutaba en un único hilo (loop()), para adoptar una arquitectura multitarea concurrente basada en FreeRTOS.

- Racionalización: Esta es la decisión de diseño clave que resuelve las preocupaciones fundamentales del diseño anterior:
  - Resuelve C-1 (Responsividad de la UI): En el diseño original (Patrón State), cualquier operación de bloqueo o semibloqueante (como esperar la lectura de un sensor o un delay) en el loop() principal "congelaba" la lectura de los botones. Al mover la taskMenu a un núcleo dedicado (Núcleo 0), la UI se vuelve completamente independiente y responsiva, sin importar lo que hagan las tareas de red o control en el Núcleo
    1.
  - Resuelve C-2 (Concurrencia): El Patrón State original simula la concurrencia pero es fundamentalmente secuencial (primero lee sensor, luego controla riego, luego actualiza LCD, luego revisa botones). FreeRTOS permite que el sensor esté leyendo (Núcleo 0) *al mismo tiempo* que se envían datos a

Firebase (Núcleo 1), logrando una verdadera concurrencia.

  - Separación de Preocupaciones (Anexo A.3): El nuevo diseño separa limpiamente las preocupaciones: taskMenu (UI), taskSensorHumedad (Datos), taskControlRele (Lógica de Negocio), taskFirebaseUpdate (Red). El diseño anterior mezclaba estas preocupaciones dentro de un único estado (EstadoOperacional).

- Decisión R-2: Asignación de Tareas a Núcleos Específicos.
  - Decisión: Las tareas de Wi-Fi/Red (taskFirebaseUpdate) y control (taskControlRele) se fijaron al Núcleo 1. Las tareas de UI (taskMenu) y Sensores (taskSensorHumedad) se fijaron al Núcleo 0.
  - Racionalización: Esta decisión refuerza la Decisión R-1. La pila de Wi-Fi y TCP/IP del ESP32 es intensiva y puede introducir latencia. Al aislarla en el Núcleo 1, se asegura que la taskMenu en el Núcleo 0 siempre tenga recursos de CPU para responder a los botones y actualizar el LCD sin lag.
- Decisión R-3: Uso de Mecanismos de Comunicación Mixtos (Cola + Globales).
  - Decisión: Se usó una cola (xQueue) para los datos del sensor, pero variables globales para el estado y la configuración.

    - Racionalización: La cola es ideal para el patrón productorconsumidor de "muchos-a-uno" (taskSensorHumedad -> taskMenu) para datos de alta frecuencia. Las variables globales se usaron para datos de "baja velocidad" (configuración) que necesitan ser *leídos* por múltiples tareas (patrón "uno-a-muchos"). Esto simplifica el código, aunque introduce la inconsistencia IN-1.
- Decisión R-4: Lógica de Botones No Bloqueante.
    - Decisión: Se implementó una máquina de estados personalizada (procesar_botones) para gestionar el *debounce* y las presiones cortas/largas, en lugar de usar delay().
    - Racionalización: En un RTOS, delay() es una función de bloqueo que duerme la tarea *completa*. Si se usara delay() en la taskMenu, la pantalla no podría actualizarse con los datos de la cola de humedad mientras se espera el *debounce*. La solución implementada permite que el bucle de la taskMenu se ejecute rápidamente, procesando tanto eventos de botón como actualizaciones de la cola de forma concurrente.

## 5. Problemáticas vividas en cada versión.

V1 Bloqueos y retrasos con la interfase física, los sensores, el menú.

V2 El tiempo se calculaba mediante código lo que complicaba por mucho la complejidad del código y la sincronización de los objetos físicos y red.

V2.1 El módulo aunque ayudó a tener una mejor forma de contar el tiempo aun seguía siendo complejo y costosa la implementación de un módulo extra.

V3 Para simplificar se optó quitar el módulo RTC y usar un servidor NTP para obtener la hora y controlar mejor y también por desincronización entre la interfase física y la web se optó por quitar la física y dejar el control únicamente por un sistema web.