TAREA #5

Integrantes:
Daniela Basilio

Shirley Esquibel

Melisa Castro

Alexander Gallegos

Ejercicio 1

Descripción del ejercicio.

El objetivo de esta práctica es implementar un sistema de comunicación serial bidireccional utilizando la UART2 de un microcontrolador ESP32 bajo el framework ESP-IDF. El sistema debe actuar como un intérprete de comandos que permita interactuar con el hardware (un LED en el GPIO 2) y gestionar datos internos (un contador de eventos) a través de una terminal serial configurada a 115200 bps.

Los requisitos funcionales incluyen:

- Configuración de pines específicos para UART2 (RX:16, TX:17).
- Implementación de lectura de datos no bloqueante con un tiempo de espera definido.
- Reconocimiento de comandos de texto: status, led on, led off, info y reset.

Explicación del funcionamiento del sistema.

El sistema opera mediante una arquitectura de bucle de control por consulta (polling) asistido por los controladores (drivers) nativos del ESP32.

- Inicialización: Se reserva memoria para los buffers de la UART y se configuran los parámetros de comunicación (8 bits de datos, sin paridad, 1 bit de parada). Se instalan los controladores para manejar las colas de transmisión y recepción.
- Gestión del Buffer de Línea: Debido a que la comunicación serial recibe datos carácter por carácter, el sistema implementa un Buffer de Acumulación. Cada carácter recibido se almacena en un arreglo (line_buffer) hasta que se detecta el carácter de fin de línea (\n o \r).
- Procesamiento de Comandos: Una vez recibida una línea completa, se aplica una función de limpieza (clean_buffer) para eliminar espacios o caracteres invisibles. Luego, se utiliza la función strcmp para comparar la cadena limpia contra los comandos predefinidos:

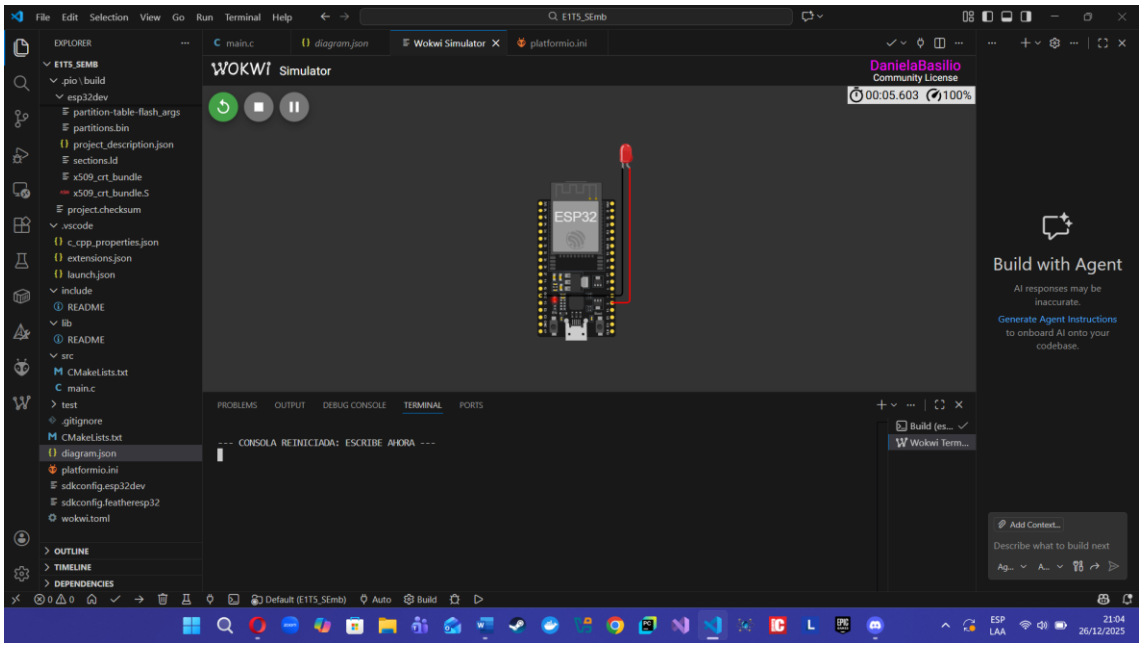
- led on/off: Cambia el estado del pin GPIO 2 mediante gpio_set_level.
- info: Genera una cadena dinámica utilizando snprintf para reportar el valor del contador de comandos procesados.
- reset: Reinicia la variable de conteo a cero.

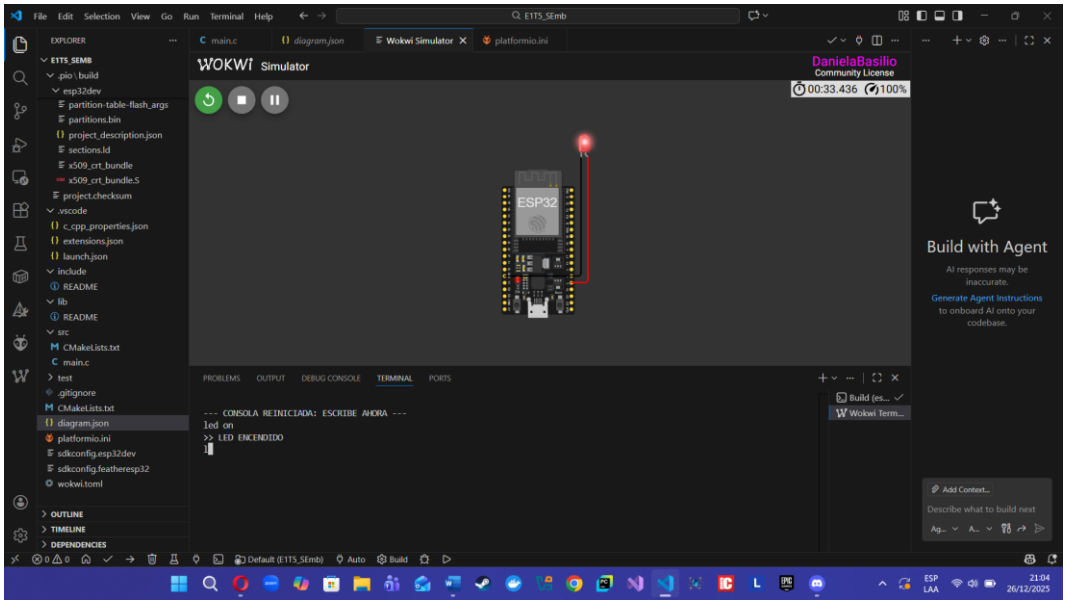
- Eco (Echo): Para mejorar la experiencia del usuario, cada carácter recibido se reenvía inmediatamente a la terminal para que el usuario pueda ver lo que está escribiendo en tiempo real.

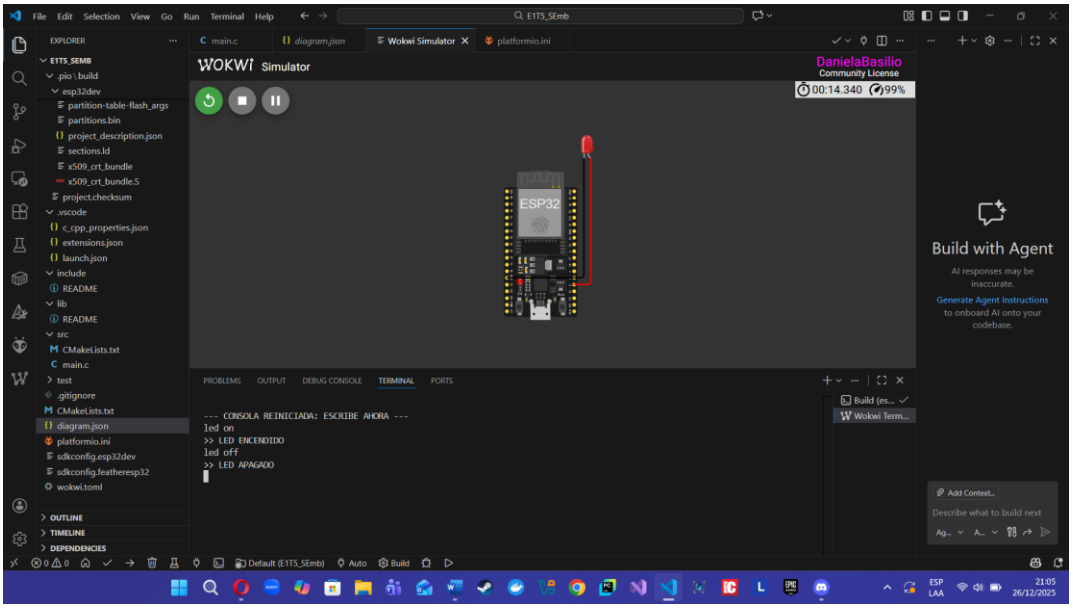
Diagrama de flujo o arquitectura.

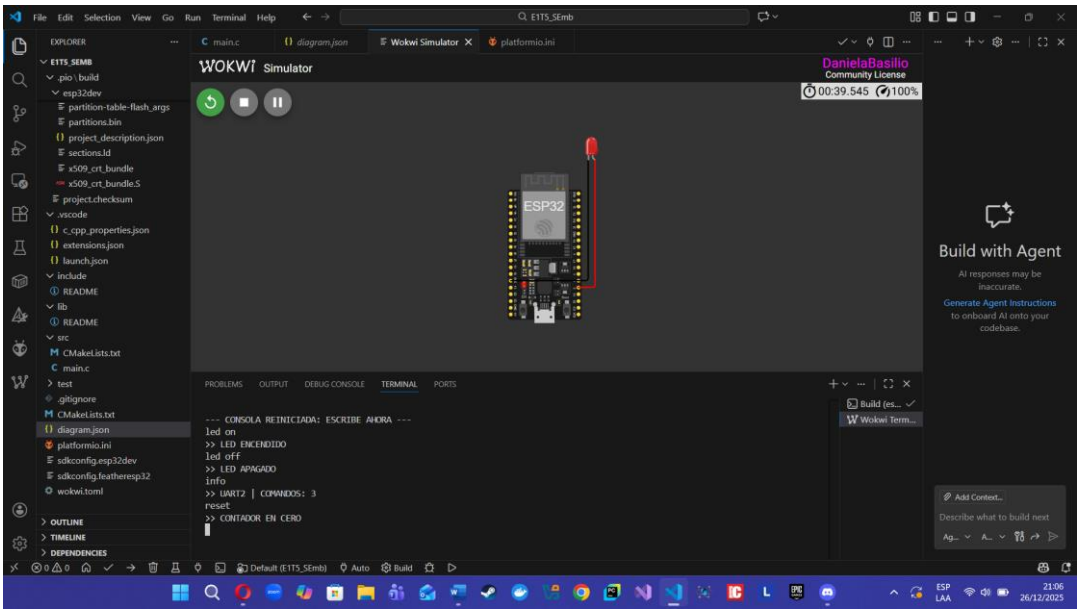
1. Capa de Hardware: ESP32 (UART2 y GPIO 2).
2. Capa de Driver (ESP-IDF): Funciones uart_read_bytes y uart_write_bytes.
3. Capa de Aplicación:
   - Módulo de Recepción: Escanea el puerto serial con un timeout de 20ms.
   - Módulo de Limpieza: Normaliza el texto recibido.
   - Módulo de Control: Toma decisiones basadas en comparaciones de cadenas y actúa sobre el hardware o las variables internas.

Capturas de simulación.




WOKWI Simulator
DanielaBasilio
Community License
00:33.436
100%
ESP32
--- CONSOLA REINICIADA: ESCRIBE AHORA ---
led on
>> LED ENCENDIDO
Build with Agent
AI responses may be inaccurate.
Generate Agent Instructions
to onboard AI onto your codebase.


WOKWI Simulator
DanielaBasilio
Community License
00:14.340
99%
ESP32
--- CONSOLA REINICIADA: ESCRIBE AHORA ---
led on
>> LED ENCENDIDO
led off
>> LED APAGADO
Build with Agent
AI responses may be inaccurate.
Generate Agent Instructions
to onboard AI onto your codebase.


WOKWI Simulator
DanielaBasilio
Community License
00:39.545
100%
ESP32
--- CONSOLA REINICIADA: ESCRIBE AHORA ---
led on
>> LED ENCENDIDO
led off
>> LED APAGADO
info
>> UART2 | COMANDOS: 3
reset
>> CONTADOR EN CERO
Build with Agent
AI responses may be inaccurate.
Generate Agent Instructions
to onboard AI onto your codebase.

Ejercicio 2

Descripción del ejercicio.

El objetivo es diseñar e implementar un sistema embebido multitarea sobre un ESP32 utilizando el sistema operativo de tiempo real FreeRTOS. El sistema debe gestionar de manera concurrente (simultánea) tres procesos con diferentes responsabilidades y prioridades:

- Gestión de Interfaz (UART2): Un intérprete de comandos que permite al usuario modificar el comportamiento del sistema en tiempo real.
- Control de Actuador (LED): Una salida digital que cambia su estado (encendido/apagado) a frecuencias variables según la configuración del usuario.
- Monitoreo del Sistema: Un proceso automático que reporta el estado actual de las variables internas y el conteo de comandos procesados cada 5 segundos.

Explicación del funcionamiento del sistema.

El sistema se basa en un modelo de planificación por prioridades gestionado por el scheduler de FreeRTOS. A diferencia de un programa secuencial, aquí el procesador alterna entre tareas tan rápido que parecen ejecutarse al mismo tiempo:

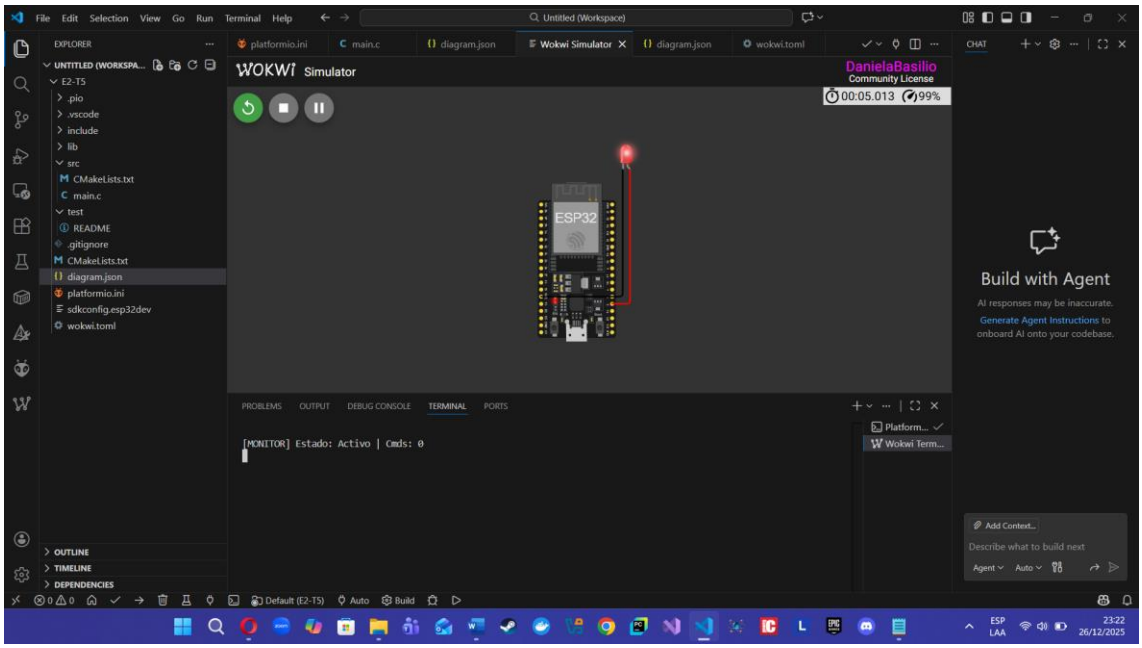
- Multitarea No Bloqueante: Se utiliza vTaskDelay() en lugar de retardos simples. Esto permite que, cuando una tarea está "esperando" (por ejemplo, el LED apagado), el CPU se libere para atender la terminal UART o la tarea de monitoreo.
- Comunicación Inter-Tarea: Las tareas comparten información a través de variables globales seguras (blink_speed, blink_active, total_commands). Cuando la tarea de la UART detecta un comando válido, modifica estas variables, y las otras dos tareas adaptan su comportamiento inmediatamente.
- Priorización de Recursos: * La Tarea de Consola tiene la prioridad más alta (5) para asegurar que el sistema responda de inmediato a la entrada del usuario.
    - La Tarea del LED tiene prioridad media (4) para mantener un ritmo de parpadeo estable.
    - La Tarea de Monitoreo tiene la prioridad más baja (3) ya que su reporte periódico no es crítico para el funcionamiento del hardware.

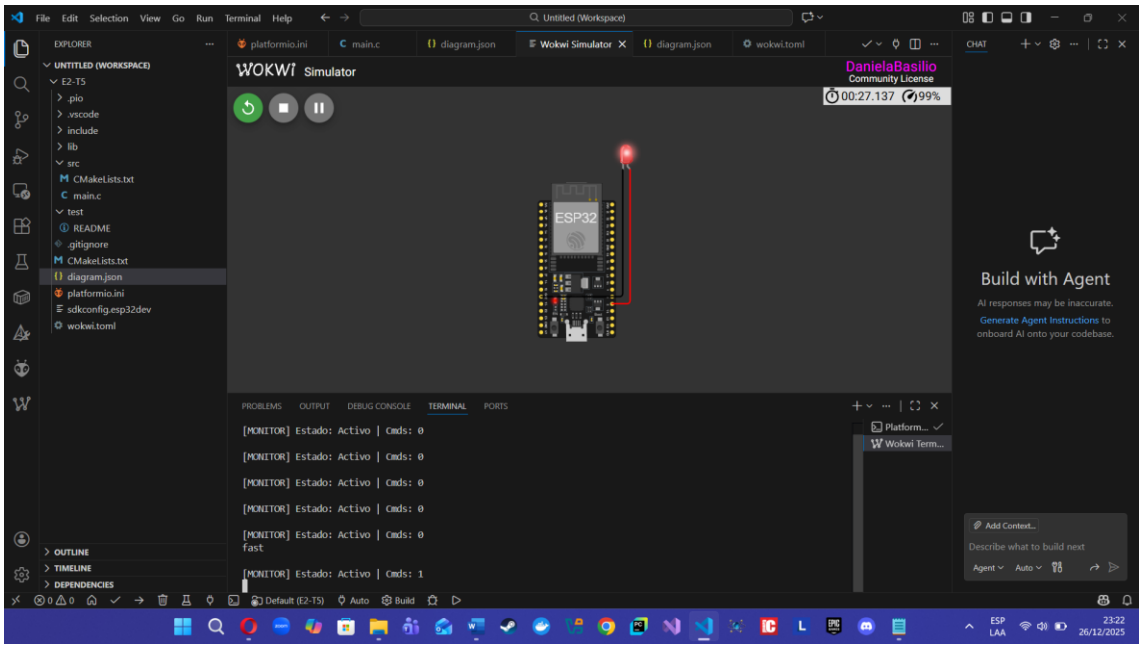
Diagrama de flujo o arquitectura.

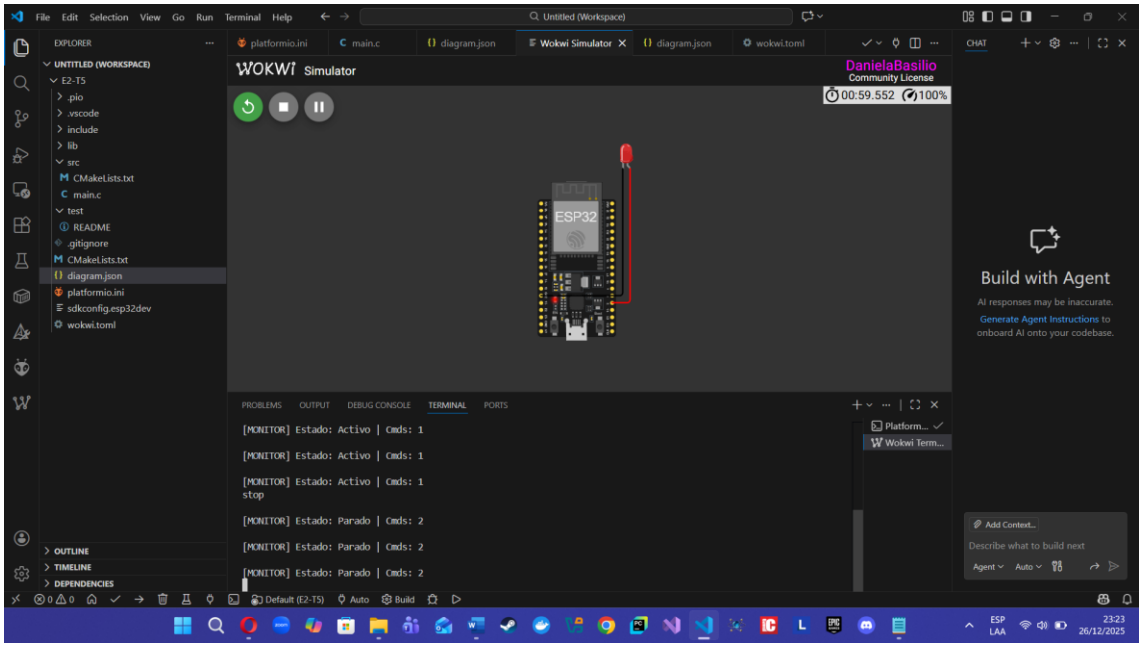
1. Inicio (app_main): Se inicializa el hardware (UART2 y GPIO 2).
2. Creación de Tareas: Se lanzan vTaskUART, vTaskBlink y vTaskMonitor con sus respectivas prioridades y tamaños de stack.

3. Bucle de Ejecución:
    - vTaskUART: Queda a la espera de un carácter. Si detecta un comando (fast, slow, etc.), actualiza las variables globales.
    - vTaskBlink: Lee la variable blink_speed y ejecuta el ciclo de encendido/apagado.
    - vTaskMonitor: Cada 5000ms genera un reporte de texto y lo envía a la terminal.

Capturas de simulación (si aplica).







Ejercicio 3

Descripción del ejercicio.

Implementación de mecanismos de comunicación entre tareas (Inter-Task Communication) mediante el uso de colas (Queues) de FreeRTOS. El objetivo es desacoplar la tarea de recepción de datos (UART) de la tarea de ejecución (LED), eliminando la dependencia de variables globales y mejorando la seguridad del hilo.

Explicación del funcionamiento del sistema.

- Productores y Consumidores: La tarea UART actúa como "Productora", traduciendo el texto del usuario en un valor numérico y colocándolo en la cola mediante xQueueSend().
- Seguridad de Datos: La cola actúa como un buffer seguro. Si el usuario envía muchos comandos seguidos, la cola los almacena en orden (FIFO) hasta que la tarea del LED los pueda procesar con xQueueReceive().
- Bloqueo Inteligente: Si la cola está llena, el productor puede esperar automáticamente; si está vacía, el consumidor puede entrar en estado de espera, optimizando al máximo el uso del CPU.

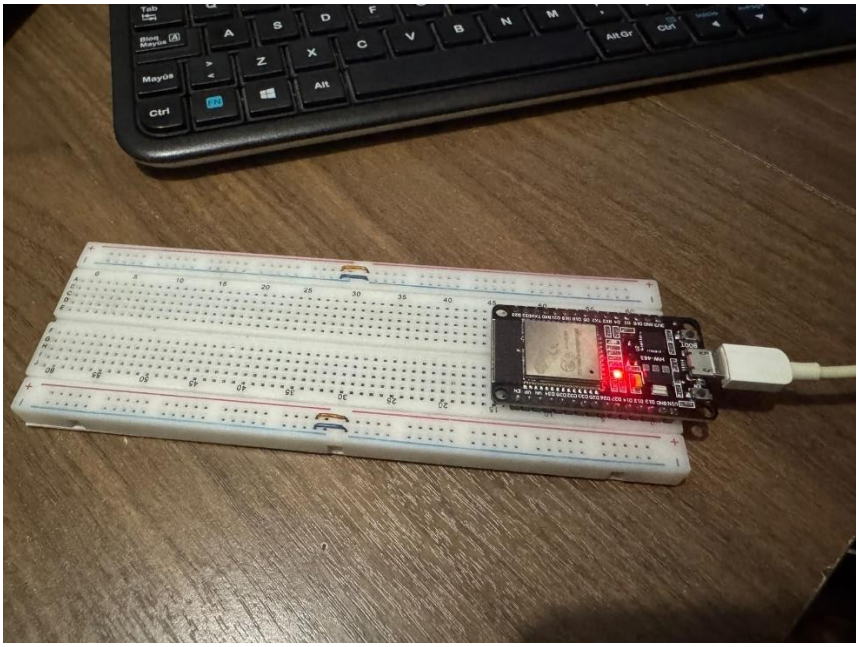
Diagrama de flujo o arquitectura.

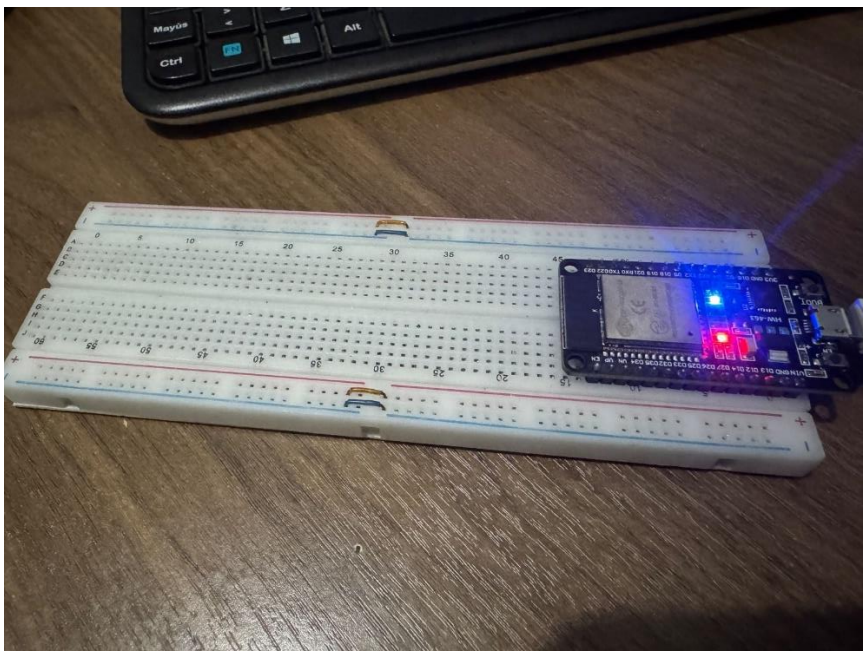
La arquitectura cambia de una estructura de "Memoria Compartida" a una de "Paso de Mensajes":

1. Entrada: UART recibe fast.

2. Proceso A: Tarea UART identifica el comando y envía el entero 100 a la cola xQueueComandos.
3. Medio: La cola almacena el valor de forma segura.
4. Proceso B: Tarea LED extrae el 100 de la cola y actualiza su registro de tiempo para el siguiente parpadeo.

Evidencia de pruebas en hardware real.





Ejercicio 4

Descripción del ejercicio.

Implementación de técnicas avanzadas de sincronización de tareas mediante el uso de Semáforos Binarios para la señalización de eventos y Mutex para la protección de secciones críticas (recursos compartidos). El ejercicio demuestra cómo una tarea puede permanecer en estado de "bloqueo eficiente" hasta que otra tarea le notifica que hay trabajo disponible.

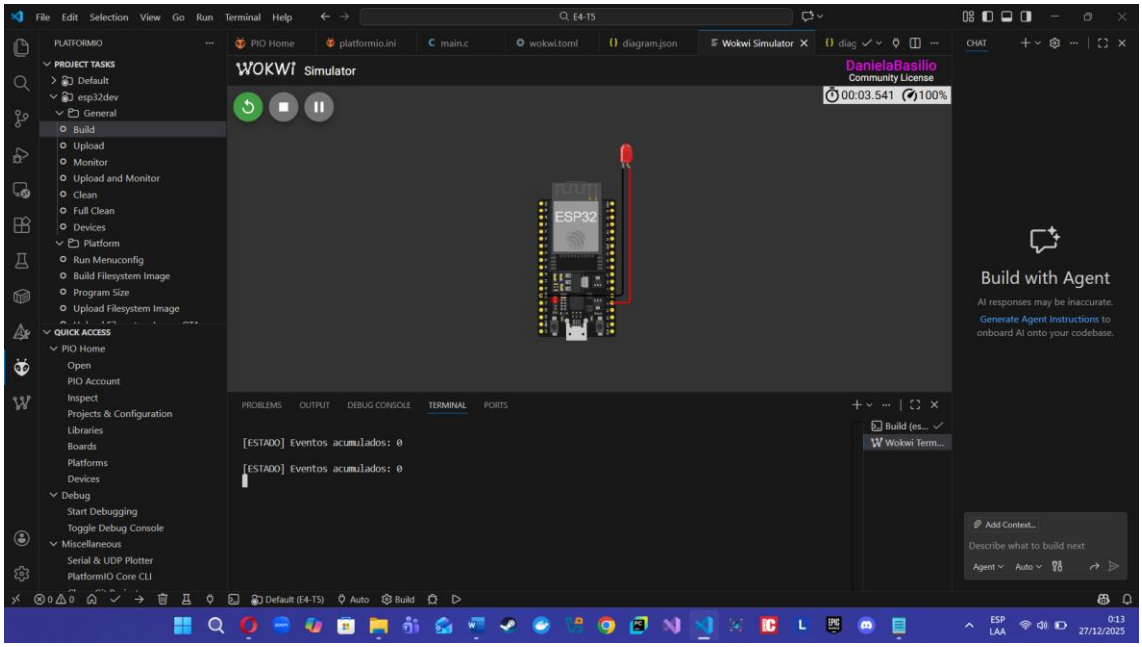
Explicación del funcionamiento del sistema.

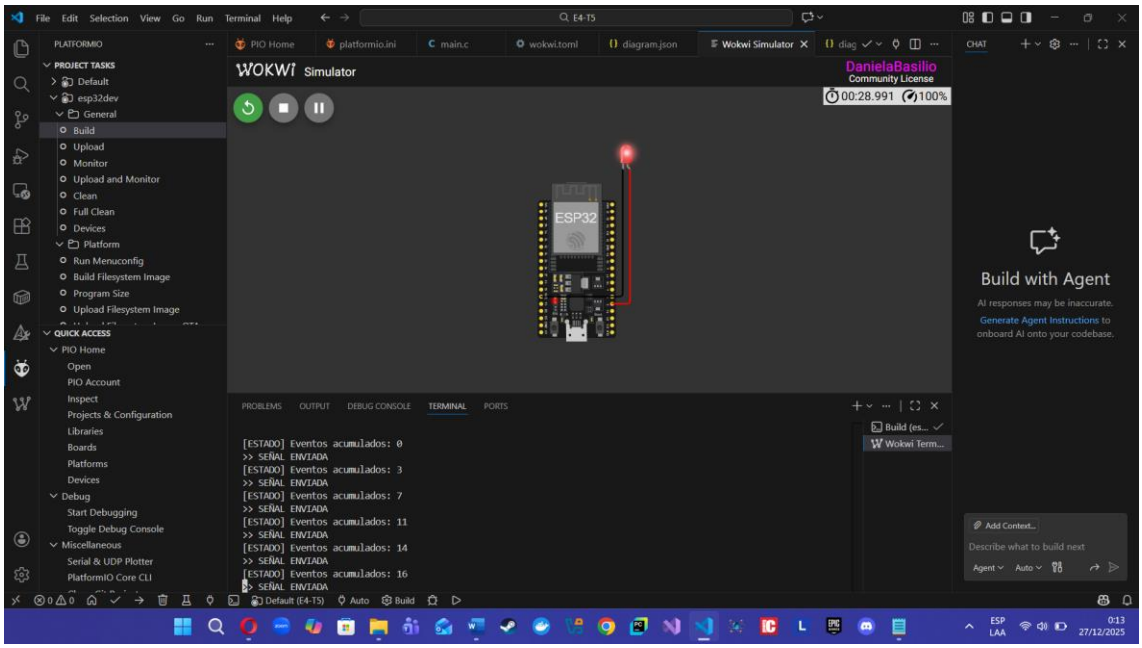
- Señalización (Semáforo): La tarea vTaskUART detecta la interacción del usuario. En lugar de procesar los datos ella misma, "da" un semáforo (xSemaphoreGive). La tarea vTaskProcessor, que estaba dormida, detecta el semáforo, se despierta y realiza el trabajo pesado.
- Exclusión Mutua (Mutex): Para incrementar la variable eventos_totales, el procesador debe "adquirir el token" del Mutex. Esto garantiza que si en el futuro añadimos más tareas, ninguna intentará modificar el contador al mismo tiempo, evitando errores de memoria.

Diagrama de flujo o arquitectura.

1. Evento Externo: El usuario presiona una tecla en la terminal UART2.
2. Tarea A (Productor de Señal): Captura el carácter y libera el Semáforo Binario.
3. Planificador (Scheduler): Detecta que el Semáforo está disponible y despierta a la Tarea B.
4. Tarea B (Consumidor de Señal): * Intenta tomar el Mutex.
    - Incrementa el recurso compartido de forma segura.
    - Libera el Mutex.
    - Realiza una acción física (flash del LED).
    - Vuelve a bloquearse esperando el siguiente semáforo.

Capturas de simulación.





Enlace al repositorio GitHub con el código.

https://github.com/danibasilio887/Tarea-5---SEmb.git

Conclusiones y recomendaciones

Conclusiones:

- El uso de una UART secundaria (UART2) para el control del sistema permite separar el tráfico de depuración (UART0) de la interfaz de usuario. Esto garantiza que el envío de comandos no interfiera con los mensajes de error del sistema,

proporcionando un canal de comunicación determinista y profesional en entornos industriales.

- A diferencia de un bucle tradicional (loop), FreeRTOS permite que las tareas entren en estado de "Bloqueo" (Blocked) mientras esperan eventos como un semáforo o una cola. Esto elimina el desperdicio de ciclos de CPU en esperas activas (*polling*), permitiendo que el microcontrolador gestione múltiples procesos como la consola UART y el parpadeo de LEDs de forma verdaderamente concurrente.
- La implementación de funciones como vTaskDelay() en lugar de retardos por hardware es clave para el ahorro de energía. Al invocar estas funciones, el planificador de FreeRTOS puede poner al microcontrolador en un modo de bajo consumo o ejecutar una tarea de prioridad mínima (*Idle Task*), reduciendo significativamente el consumo eléctrico del ESP32 entre ejecuciones de tareas.

Recomendaciones:

- Priorización y Protección de Recursos: Al diseñar sistemas embebidos complejos, se recomienda asignar prioridades altas a las tareas de comunicación (como UART o Wi-Fi) para evitar la pérdida de datos, y utilizar siempre mecanismos de exclusión mutua (Mutex) al acceder a variables compartidas. Esto previene condiciones de carrera y asegura la integridad de los datos en sistemas donde varias tareas intentan leer o escribir simultáneamente.
- Modularidad y Escalabilidad: Diseñe cada tarea con una única responsabilidad (por ejemplo, una tarea solo para sensores, otra solo para actuadores). Esta modularidad, facilitada por FreeRTOS, permite escalar el proyecto añadiendo nuevas funcionalidades sin necesidad de reescribir la lógica de tiempo del sistema completo, lo cual es fundamental para el desarrollo de firmware robusto y fácil de mantener.