IoT

LABORATORIO 3

FreeRTOS

Estudiantes:

Marcos Giombini

Fernando Larrica

Diego Massaferro

16/06/2025

Objetivo

El objetivo de este laboratorio fue implementar una arquitectura multitasking en un sistema embebido FreeRTOS, haciendo uso de timers, queues, semáforos y manejo dinámico de memoria. Se buscó, además, aplicar las herramientas vistas en el Módulo 2 para el diseño, sincronización e interacción entre tareas concurrentes.

Descripción del sistema

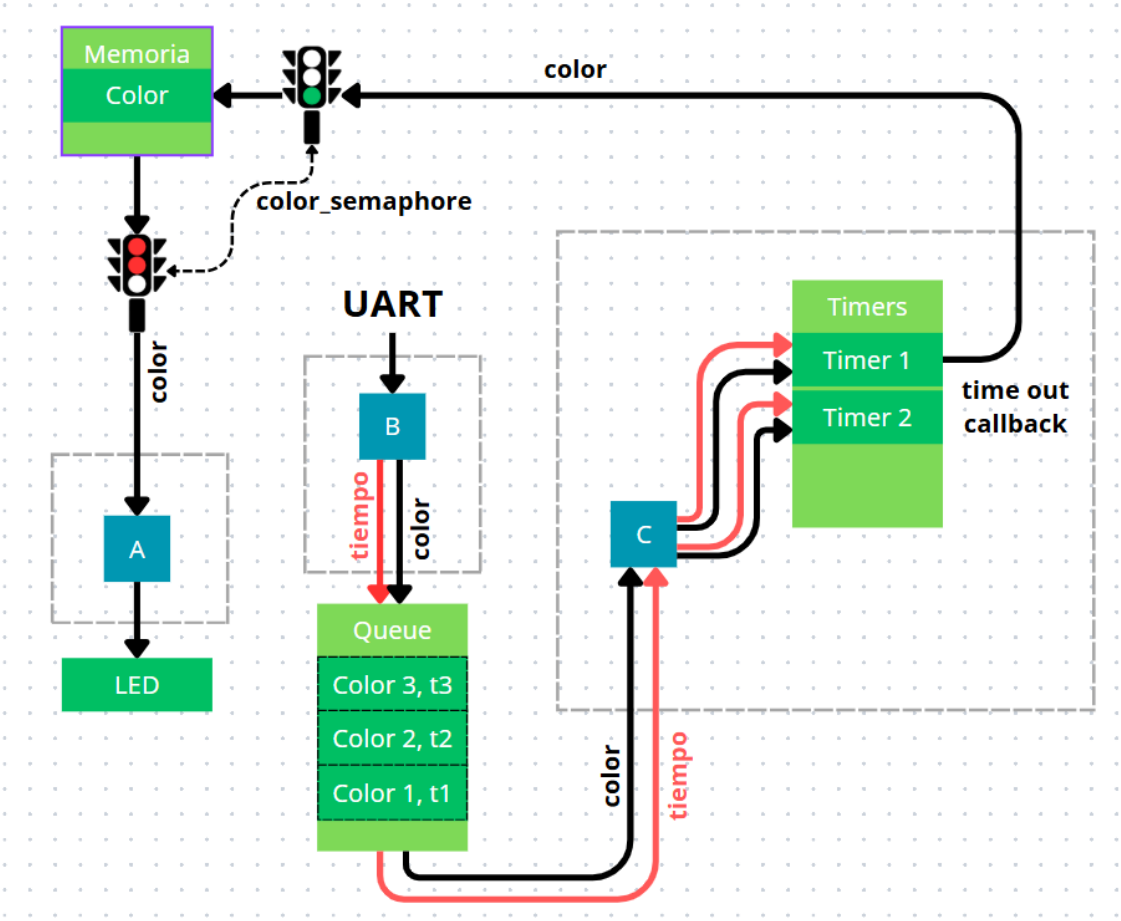
El sistema está compuesto por un grupo de tres tareas principales

- Task A: Controla el parpadeo de un LED RGB, alternando entre encendido y apagado según el color activo.
- Task B: Lee comandos recibidos por UART, los interpreta y los envía a través de una cola de comandos.
- Task C: Procesa los comandos recibidos, gestiona los temporizadores (timers) y activa los eventos de cambio de color.

Para la comunicación y sincronización entre tareas se utilizaron:

- Semáforos binarios (color_semaphore) para notificar cambios de color al expirar un timer en la Task C.
- Colas (command_queue) para transferir los comandos leídos desde UART hacia la lógica de ejecución.
- Timers FreeRTOS configurados como one-shot, que manejan el retraso ingresado por el usuario antes de ejecutar el cambio de color.

Diagrama de flujo y pseudocódigo general

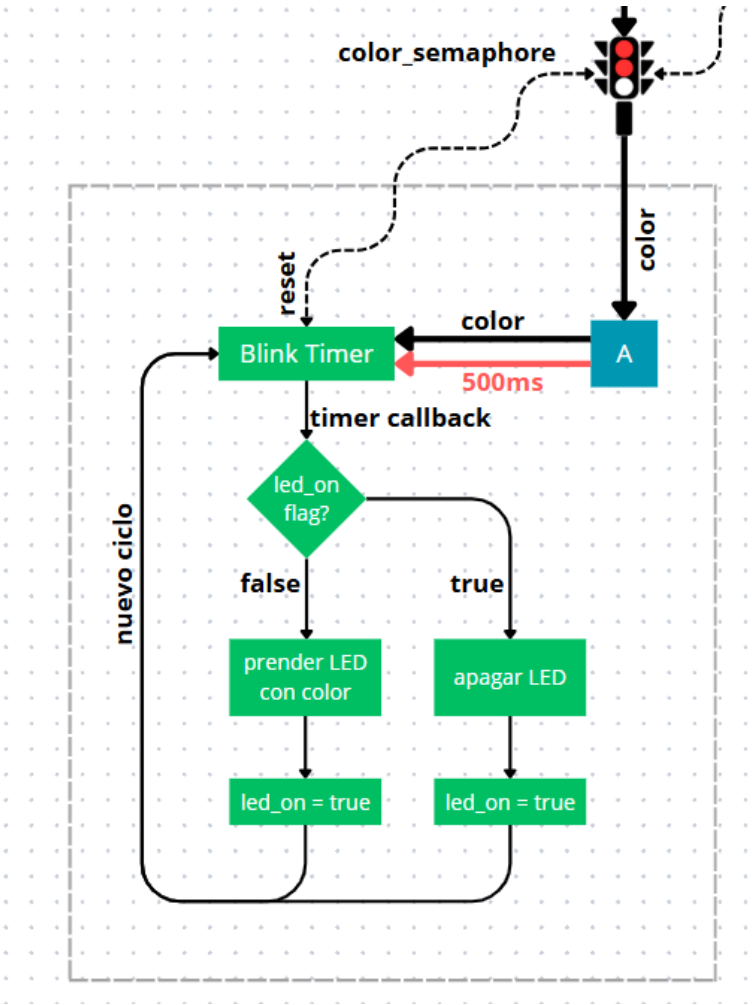


Flujo general del sistema:

1. El usuario envía una línea de comando por UART con el formato: color,delay
2. Task B lee e interpreta el comando y lo envía como un **`uart_commant_t`**. a la cola **`command_queue`**.
3. Task C recibe el comando **`cmd`** de la cola, creando un timer one-shot, con tiempo de disparo **`cmd.delay_seconds`** y con el color asociado al ID del timer **`cmd.color`**.
4. Al disparar el timer, el callback actualiza **`current_color`** y libera el **`color_semaphore`**.
5. Task A actualiza el color luego de detectar el semáforo y reinicia el parpadeo.

Flujos y pseudocódigos de funciones principales

Diagrama de flujo de parpadeo LED en task A



Flujo de Task A:

1) Se crea un timer con reinicio, que dispara cada 500ms.
2) La función de callback cambia el estado del flag led_on cada vez que entra, ejecutando de forma condicional:
   a) Si **`led_on=true`**: Apaga LED, invierte flag **`led_on`**.
   b) Si **`led_on=false`**: Prende LED con **`current_color`**, invierte flag **`led_on`**.
3) Cuando se habilita **`color_semaphore`**:
   a) Se actualiza **`current_color`**
   b) Se fija el flag **`led_on=false`**
   c) Se apaga el LED
   d) Con esto, en el próximo disparo de timer se muestra el nuevo color de LED

Pseudocódigo de Task B

```
INICIO Task B
Configurar UART
Instalar driver UART
Inicializar buffers
Bucle infinito
     Lee 1 byte desde UART
     Byte es fin de línea? ('\n' o '\r')
          Terminar línea con carácter nulo
          Extraer color_str y delay_val a partir de la línea
almacenada en line_buffer
          Mapear color a cmd.color:
               "rojo" → LED_EVENT_ROJO
                         "verde" → LED_EVENT_VERDE
                         "azul" → LED_EVENT_AZUL
                         "amarillo" → LED_EVENT_AMARILLO
                         "cian" → LED_EVENT_CIAN
                         "blanco" → LED_EVENT_BLANCO
                         "apagar" → LED_EVENT_APAGAR
          Cargar delay a cmd.delay_seconds
          Enviar cmd a command_queue
          Resetear line_buffer y line_pos
     Sino:
          Agregar byte a line_buffer
          Incrementar line_pos
Fin de Task B
```

Pseudocódigo de Task C

```
INICIO Task C

     BUCLE INFINITO

     Crea estructura info que contiene: Info.color(guarda el
color traído desde el comando de la cola)

/* Se crea esta estructura como forma de escalar en un futuro
y pasar mas información por el ID del callback del timer, en
caso de ser necesario. Por ahora solo pasamos color*/

     Crea timer one-shot con:

          Tiempo de disparo: cmd.delay_seconds

          ID del timer: puntero a info

          Callback: vTimerCallback

     Inicia el timer

FIN Task C


INICIO vTimerCallback

     Toma la variable info del ID del timer

     Setea current_color a info.color

     Libera memoria de info

     Entrega color_semaphore

FIN vTimerCallback
```

Recursos de FreeRTOS utilizados

Empleamos varios mecanismos de FreeRTOS para cumplir con la consigna del laboratorio:

Tareas

Se definieron tres tareas independientes, cada una con una prioridad asignada según su criticidad:

- Task B (prioridad 10): encargada de recibir los comandos UART, parsearlos y encolar los comandos resultantes. Su prioridad elevada garantiza una rápida respuesta frente a la llegada de datos desde el puerto serie, evitando pérdidas de información.

- Task C (prioridad 8): toma los comandos desde la cola y crea, en caso de ser necesario, los timers one-shot correspondientes. Al centralizar la creación de timers en esta tarea, se garantiza un manejo ordenado de los retardos.

- Task A (prioridad 6): mantiene el parpadeo del LED en el color activo. Se mantiene en espera bloqueante hasta ser notificada de un nuevo color mediante el semáforo.

Esta división de tareas favorece la separación clara de funciones y evita bloqueos innecesarios durante la ejecución.

Semáforos

Se utilizó un semáforo binario (**`color_semaphore`**) como mecanismo de señalización desde los callbacks de los timers hacia Task A. Cada vez que un timer vence, el callback asociado actualiza el color actual (**`current_color`**) y libera el semáforo, permitiendo a Task A reaccionar al cambio. Este mecanismo permitió una sincronizar la lógica temporizada y la tarea de parpadeo. Durante la implementación, entendimos que podría tener algunas limitaciones en situaciones de alta concurrencia de timers. Presentamos el análisis en las conclusiones.

Colas

El pase de información entre Task B y Task C se resolvió mediante una cola (**`command_queue`**). Cada comando UART recibido es transformado en una estructura **`uart_command_t`**, que contiene el color y el retardo especificado, y es encolado para su posterior procesamiento. Esto permite desacoplar completamente la recepción de comandos de la lógica de temporización, evitando bloqueos en la tarea de UART y ordenando los comandos en una estructura FIFO.

Timers

Se implementaron dos tipos de timers dentro del sistema:

- Timers one-shot en Task C: Cada vez que Task C recibe un comando con un retardo distinto de cero, crea un timer one-shot con la API de FreeRTOS. Este timer se activa una única vez luego del tiempo especificado por el usuario. La información de configuración (color a aplicar) se pasa a través del timerID, mediante una estructura timer_info_t asignada dinámicamente. Al vencer el timer, el callback actualiza el color actual (**`current_color`**) y notifica a Task A mediante el semáforo.

- Timer periódico en Task A: Independientemente de los comandos UART, Task A mantiene un timer periódico (auto-recargable) que determina el ciclo de parpadeo

del LED. Este timer alterna el estado del LED encendiéndolo o apagándolo cada 500 ms, utilizando la última configuración de color activa. Este esquema asegura que el LED continúe parpadeando automáticamente mientras no se reciba un nuevo comando.

Memoria dinámica

Para almacenar los parámetros individuales de cada timer, se reserva memoria dinámica (malloc) al crear cada timer_info_t. Este mecanismo permite generar múltiples timers simultáneos sin conflictos, ya que cada uno opera sobre su propio contexto de datos. Una vez ejecutado el callback y aplicado el nuevo color, se libera la memoria correspondiente para evitar fugas.

Conclusiones

Semáforo implementado y posibles mejoras

El uso del semáforo binario permitió sincronizar de forma simple el aviso de nuevos colores entre los timers y la tarea de parpadeo, evitando que Task A tenga que hacer polling constante. Si bien esta solución funcionó bien para los casos normales de uso, se detectó que ante múltiples timers activos en paralelo podría perder algunos eventos por las limitaciones que tiene este tipo de semáforo.

Para cubrir este tipo de escenarios, una cola de eventos sería una opción más robusta.

Inconvenientes con recursos compartidos e implementación de shared_lib

Al avanzar en el desarrollo del laboratorio, surgió la necesidad de compartir ciertas variables globales (como los semáforos, la cola de comandos y el color actual) entre las distintas tareas del sistema. Inicialmente, al intentar definir estas variables en varios archivos, comenzaron a aparecer errores de compilación por múltiples definiciones o variables no encontradas.

Esto ocurría porque cada tarea necesitaba acceso a los mismos recursos, pero si las variables no estaban centralizadas, el compilador generaba conflictos al vincular el proyecto.

Para resolver este problema, se creó el módulo shared_lib, compuesto por un archivo de cabecera (shared_lib.h) y su implementación (shared_lib.c). Allí se definieron de forma única todas las variables compartidas, y se implementó la función inicializar_recursos_globales(), que centraliza la creación de las colas y semáforos.

De esta forma, cada tarea puede incluir el shared_lib.h y acceder a los recursos globales sin generar conflictos ni duplicaciones.

Otra forma de haberlo solucionado era pasando los recursos como parámetros a cada tarea al momento de crearla, pero se optó por shared_lib por ser más simple, directo y escalable para el tamaño y complejidad de este laboratorio.

Conclusiones finales

Se logró implementar un sistema multitarea funcional y estable, aplicando los recursos principales de FreeRTOS de forma ordenada. Cada tarea tiene una función clara, las colas permiten desacoplar la recepción de comandos del procesamiento, y los timers manejan los retardos sin bloquear el sistema. La creación de shared_lib resolvió los problemas de integración de recursos compartidos y simplificó el manejo global de las variables. Además, se identificaron posibles mejoras en la sincronización que podrían optimizar el sistema ante situaciones de alta concurrencia.