Universidad Católica del Uruguay

Laboratorio 3

17 de Junio de 2025

Diseño de IoT y Sistemas Embebidos

Integrantes del equipo:

Gonzalo Soto

Jerónimo Ventos

Santiago El Ters

Contenido

Introducción

En este laboratorio se trabajará con un sistema basado en FreeRTOS, un sistema operativo en tiempo real que nos permite ejecutar varias tareas al mismo tiempo en un entorno embebido.

El objetivo principal es organizar el programa en tres tareas independientes, que interactúan para controlar un LED RGB. A través de este laboratorio se busca aplicar conceptos fundamentales como colas, semáforos, temporizadores y manejo dinámico de memoria, integrados dentro de una arquitectura modular (una librería por tarea) y concurrente.

El sistema en general se compone de tres tareas principales. Por un lado, la tarea A es la encargada de hacer parpadear el LED con un color que se define en una variable compartida. Por otro lado, la tarea B se encarga de recibir comandos del usuario mediante UART (por USB), los cuales indican un color y un tiempo de retardo, y luego los envía a una cola. Finalmente, la tarea C extrae los comandos de dicha cola, crea un temporizador one-shot por cada uno y, cuando este se cumple, actualiza el color que la tarea A usará para parpadear el LED.

Para asegurar que la información del color no se corrompa al ser accedida por más de una tarea, se utiliza un semáforo tipo mutex que protege la variable compartida. Además, cada tarea fue organizada dentro de su propia librería para fomentar la modularidad del código.

Desarrollo

A continuación, se describen en detalle las implementaciones de cada una de las tareas del sistema. En cada caso se explicará primero su funcionamiento general, seguido de un pseudocódigo a alto nivel que resume su lógica, y luego se detallarán las decisiones tomadas durante el desarrollo, las alternativas consideradas, y las soluciones que finalmente se implementaron.

Task A - Parpadeo de LED RGB

Funcionamiento general

La tarea A es responsable de hacer que el LED RGB parpadee utilizando un color que se encuentra almacenado en una variable compartida de memoria. Esta variable es modificada por la tarea C y la tarea A accede a ella de manera protegida utilizando un semáforo mutex, para evitar conflictos de concurrencia.

El comportamiento de esta tarea es cíclico: en cada iteración del bucle, intenta obtener el semáforo, accede al color actual, aplica ese color al LED, y luego alterna el estado del LED para simular el parpadeo. Todo esto se repite indefinidamente, con un pequeño retardo entre cada cambio de estado.

Pseudocódigo

```
INICIO
    Bucle infinito:
        Tomar semáforo
        Leer color actual de la variable compartida
        Liberar semáforo

        Aplicar color al LED
        Alternar estado del LED (encendido/apagado)
        Esperar 2 segundos
FIN
```

Implementación

Para facilitar la comunicación entre las tareas y evitar la repetición de código, se creó una pequeña librería llamada color_shared, que contiene únicamente un archivo .h con la definición de la estructura led_color_t. Esta estructura encapsula los valores RGB del color y el tiempo en segundos que se debe esperar antes de aplicar el cambio. Al estar contenida en un único header (color_shared.h), se evita la necesidad de duplicar la definición en múltiples archivos, y permite a todas las tareas acceder de forma consistente a la misma representación de color compartido.

Uso de la estructura led_color_t: Como se mencionaba se utilizó una estructura para agrupar los valores RGB junto al tiempo de retardo. Esto facilita el manejo y la transferencia de información entre tareas. En esta tarea solo se usan los componentes r, g, b.

Protección con semáforo (mutex): Dado que el color puede ser actualizado por la tarea C mientras la tarea A lo está leyendo, fue necesario proteger el acceso a esta variable con un semáforo de tipo mutex. Esto garantiza que no se produzcan condiciones de carrera o lecturas inconsistentes.

Función led_embebido_toggle(): Se reutilizó funciones implementadas en laboratorios anteriores, en este caso toggle, que alterna el estado del LED, y combinándola con la función led_embebido_set_color() para que el color actual sea respetado en cada encendido del parpadeo.

Retraso entre parpadeos: Se consideró esencial introducir una pausa entre cada parpadeo, utilizando vTaskDelay(), para que el efecto visual del LED sea claro. El retardo fue configurado en 2000 ms (2 segundos), y se puede ajustar si fuera necesario.

Asignación de prioridad: La tarea A se creó con una prioridad más baja que la tarea B, ya que esta última maneja la entrada del usuario y se considera más crítica para la interacción del sistema.

Descartes

Inicialmente se pensó en que la tarea A pudiera manejar el delay que se pase por línea de comando, pero se descartó porque la arquitectura del laboratorio exige que esta solo lea el color rgb de una variable compartida. La lógica de temporización y asignación de color queda completamente delegada a la tarea C.

En resumen, la tarea A actúa pasivamente, aplica el color que se le indica y mantiene el LED parpadeando. Toda la lógica de actualización del color ocurre fuera de esta tarea, lo que simplifica su comportamiento y facilita la sincronización mediante semáforos.

Task B - lectura de UART

Funcionamiento general

La tarea B tiene la responsabilidad de gestionar la entrada de comandos desde el usuario mediante la interfaz UART (puerto USB). Se encarga de interpretar estos comandos, que consisten en un color (rojo, verde o azul) y un número que indica cuántos segundos deben pasar antes de ejecutar el cambio de color. Una vez procesado el comando, lo coloca en una queue compartida, desde donde será leído por la tarea C.

Esta tarea debe ejecutarse con prioridad mayor que las demás, ya que depende de eventos externos y se espera que procese los comandos con rapidez para que el sistema no pierda datos o respuestas del usuario.

Pseudocódigo

```
INICIO

    Inicializar UART
    Reservar memoria para buffers de entrada

    Bucle infinito:
        Leer bytes desde UART
        Formar líneas de texto completas hasta detectar un '\n'
        Parsear la línea en formato: "color segundos"
        Validar color y tiempo
        Formar estructura led_color_t con los valores RGB y delay
        Enviar estructura por la queue a Task C

    Liberar buffers al finalizar (nunca debería ocurrir)

FIN
```

Implementación

Para la implementación de la tarea B fue necesario configurar la comunicación UART, ya que es la interfaz por la que el usuario envía comandos al sistema. Esta configuración se realizó en la función taskB_uart_init() incluida dentro del archivo taskB.c.

En la inicialización del UART se utilizó el puerto UART número 0, ya que por defecto está conectado al puerto USB en la mayoría de las placas de desarrollo basadas en ESP32. Esto simplifica la comunicación sin necesidad de configuraciones adicionales. No fue necesario definir manualmente los pines TX y RX, ya que se usaron los valores predeterminados que el sistema asigna al UART0.

El baud rate utilizado fue de 115200, un valor estándar y confiable que permite buena velocidad de comunicación sin comprometer estabilidad. Esta elección se basó en ejemplos oficiales de ESP-IDF, los cuales también utilizan UART0 a 115200 bps como configuración base.

Además de la inicialización del UART, la librería tiene una función parse_command() que toma una línea de texto recibida (por ejemplo "rojo 5"), separa el color y el tiempo, y traduce el color a componentes RGB. Si el color no es reconocido, la función asigna un valor neutro (r=0, g=0, b=0) y un delay igual a cero. Esta estructura de comando es almacenada en una variable de tipo led_color_t, una estructura definida en la librería común color_shared.h, que mencionamos anteriormente. La utilización de esta estructura en común evita duplicación de código y asegura consistencia en los datos que se manejan.

La función vTaskB(), que representa la ejecución principal de la tarea, se encarga de leer continuamente datos desde la UART. A medida que los caracteres llegan, se almacenan en un buffer temporal hasta completar una línea de texto. Una vez que se detecta el final de línea (\n o \r), se interpreta el comando completo y se crea un objeto led_color_t con los datos del color y el tiempo.

Este objeto se envía a una cola (queue) compartida con la tarea C, la cual se encarga de procesar estos comandos. La cola fue creada específicamente para contener elementos del tipo led_color_t, lo cual garantiza que los datos enviados estén bien formateados y sean interpretables por la tarea receptora.

Además, el uso de la función xQueueSend() con portMAX_DELAY como argumento asegura que el sistema espere indefinidamente a que haya espacio disponible en la cola si esta se encuentra llena. Esto evita la pérdida de comandos en momentos de alto tráfico (en este caso es casi imposible que suceda), y garantiza que cada orden emitida por el usuario eventualmente sea procesada.

La función start_task_b() se encarga de lanzar la tarea asociada (vTaskB) y vincular la cola de comandos al contexto interno de la librería, permitiendo su uso en todo el ciclo de vida de la tarea.

Descartes

Durante la implementación de Task B, se evaluaron distintas formas de interpretar los comandos recibidos por UART. En un principio, consideramos usar comandos más estructurados o explícitos, como “rojo, 5” o “set rojo 5”, lo cual hubiera requerido una lógica de parsing más compleja y una validación más robusta. Sin embargo, esto aumentaba considerablemente el tiempo de desarrollo y el margen de error, por lo que se optó por un formato más simple y directo: “rojo 5”.

También se tuvo que mejorar el procesamiento de lectura del UART, ya que al principio la tarea intentaba parsear comandos de manera prematura. Por ejemplo, al recibir solo una letra como "r", ya intentaba interpretarla como un comando completo. Para resolver este problema, se implementó un buffer de línea (line_buffer) que acumula los caracteres hasta detectar un salto de línea (\n o \r), lo que garantiza que solo se procese el comando una vez esté completo.

Además, no se configuraron manualmente los pines de la UART porque la intención fue simplificar el setup utilizando la UART por defecto del puerto 0 de la placa, lo cual funcionaba correctamente con los valores estándar. Con esto evitamos el uso de la función uart_set_pin(), que dió bastantes problemas en su configuración y en la inicialización del UART, no pudiendo leer, debuggear o testear funcionalidades concretas de cada tarea.

Task C - lectura de Queue y timers

Funcionamiento general

La tarea C se encarga de recibir comandos desde una cola compartida con la Task B, y ejecutarlos luego de un retardo especificado. Cada comando incluye un color RGB y un tiempo de espera (en segundos), y la ejecución se realiza mediante un timer one-shot.

Pseudocódigo

```
INICIAR
    Bucle infinito:
        RESERVAR memoria para nuevo comando

        SI se recibió un comando:
            CREAR un timer one-shot con el retardo especificado
            ASOCIAR el comando como identificador del timer (timerID)
        SINO:
            LIBERAR la memoria del comando
TERMINAR
```

```
AL ejecutar el callback del timer:
    OBTENER el comando desde el timerID

    TOMAR el semáforo para acceder al color compartido
        COPIAR el color del comando a la variable compartida
    LIBERAR el semáforo
    LIBERAR la memoria del comando
    ELIMINAR el timer
```

Implementación

En la función vTaskC(), la tarea entra en un bucle infinito donde espera recibir elementos de tipo led_color_t desde la cola. Esta estructura se definió previamente en la librería común color_shared.h, que es utilizada por todas las tareas para unificar el formato de los comandos.

Cuando la tarea recibe un nuevo comando, se realiza una reserva dinámica de memoria (malloc) para alojar sus datos. Este bloque de memoria es luego pasado como timerID al timer que se crea con xTimerCreate(), lo que permite que el callback acceda a los valores del comando cuando llegue el momento de ejecutarlo. Si la creación o el inicio del timer falla, la memoria se libera.

El callback del timer (timer_callback) es el encargado de actualizar la variable global compartida que contiene el color actual del LED. Esta variable está protegida mediante un semáforo binario (xSemaforoColor) para asegurar que la lectura y escritura se realicen de forma segura y sin condiciones de carrera. El color se copia campo por campo (r, g, b) y, al finalizar, se libera la memoria del comando y se elimina el timer.

La función start_task_c() es la encargada de lanzar la tarea. Recibe como parámetros la cola, el semáforo y un puntero a la variable compartida led_color_t, y los guarda en variables internas para que puedan ser utilizadas tanto por vTaskC() como por el callback.

Descartes

Durante el desarrollo se consideró que la tarea C fuera la encargada de comunicarse con el LED, es decir, de aplicar el color. Sin embargo, se descartó esta opción para mantener una separación clara de responsabilidades entre las tareas: la tarea A se encarga exclusivamente del parpadeo y del acceso al LED, mientras que la tarea C se limita a gestionar los comandos recibidos, programar el retardo correspondiente mediante timers one-shot, y actualizar la variable compartida de color.

Rutina en el main

Pseudocódigo

```
INICIO

    Inicializar el LED embebido
    Inicializar la UART para comunicación en Task B

    semaforo_color ← CrearMutex()
    SI semaforo_color == NULL ENTONCES
        Imprimir "Error al crear el mutex"
        TERMINAR
    FIN SI

    xQueueComandos ← CrearCola(longitud=10,
tamañoElemento=sizeof(led_color_t))

    SI xQueueComandos == NULL ENTONCES
        Imprimir "Error al crear la queue"
        TERMINAR
    FIN SI

    color_global ← {R=255, G=0, B=0, intensidad=1}

    Iniciar Task A con argumentos (semaforo_color, &color_global)

    Iniciar Task B con argumento (xQueueComandos)

    Iniciar Task C con argumentos (xQueueComandos, semaforo_color,
&color_global)

FIN
```

En la rutina implementada en el main.c se inicializan los periféricos necesarios (LED y UART), se crea un mutex para proteger el acceso al color compartido y una cola para comunicar comandos entre tareas. Luego, establece un color inicial y se crean las tres tareas principales.

Conclusiones

Se logró implementar un sistema multitarea basado en FreeRTOS que controla un LED RGB mediante la cooperación de tres tareas independientes. La modularidad del diseño, con cada tarea organizada en su propia librería, facilitó el desarrollo y le brinda escalabilidad al código.

Se demostró la importancia de utilizar mecanismos de sincronización como semáforos mutex para proteger variables compartidas y evitar condiciones de carrera. Asimismo, el uso de colas y temporizadores one-shot permitió una comunicación eficiente y la ejecución de comandos con retardos controlados.

Por último, vale la pena recalcar que el enfoque de delegar claramente las responsabilidades entre tareas, parpadeo en la tarea A, recepción de comandos en tarea B y temporización en la tarea C, simplificó la lógica general (se logró un main sencillo y entendible) y mejoró la robustez del sistema, que en caso de errores es muy accesible mantener el sistema.