PRÁCTICA 7

Índice

1. Objetivos

- FASE 1: Creación de tareas y uso de los retrasos temporales: hacer intermitencias continuas con los leds.
- FASE 2: Crear un proyecto con tres tareas, dos iguales que en la fase 1 y la otra que imprima un mensaje en el LCD.
- FASE 3: Crear un proyecto con cuatro tareas, dos iguales que en la fase 1 y las otras dos que impriman un mensaje en el display LCD, cada una en una línea diferente del LCD.
- FASE 4: Realizar la fase 3 de la práctica 5 usando el STM32CubeIDE y con el Freertos.

2. Introducción

Hasta ahora hemos realizado las prácticas sin apoyo de un SOTR, en esta práctica vamos a introducir algunos conceptos básicos y a utilizar algunos de los recursos que ofrecen este tipo de sistemas operativos en los microcontroladores. El objetivo principal de la práctica es aprender a utilizar los recursos más básicos de los SOTR: tareas, semáforos, colas y retrasos. Hay que entender esta práctica como una introducción de conceptos que posteriormente (en cursos superiores) serán tratados con más profundidad.

Con respecto al sistema operativo tiempo real que vamos a utilizar es el Freertos, este SOTR es de los más populares que podemos encontrar para microcontroladores, hay más de 35 portes a diferentes familias de microcontroladores de este sistema operativo (ver Wikipedia), desde microcontroladores pequeños de 8 bits, como AVR, PIC, 8051..., hasta los modelos más avanzados de ARM. En el tema 7 de teoría se explican algunas de las llamadas nativas de dicho sistema operativo, pero en esta práctica vamos a utilizar una envoltura (wrapper) para los sistemas operativos tiempo real en Cortex M denominada CMSIS-RTOS versión 2. La primera pregunta que nos podríamos hacer es

qué ventaja tiene usar CMSIS-RTOS, pues no es más que una envoltura, frente a usar el Freertos directamente, con sus llamadas primitivas; pues la ventaja está en que el CMSIS-RTOS es una envoltura que pretende ser universal para cualquier sistema operativo tiempo real que pueda funcionar en un Cortex M, por lo que aprender esas llamadas nos permite controlar multitud de SOTR en Cortex M sin tener que aprender detalles de cada SOTR. Fijaros en esta dicotomía, el Freertos es probablemente el más extendido de los SOTR para microcontroladores, por lo que interesaría conocerlo… pero por otra parte ARMKeil nos proporciona una interface para los Cortex M que pretende ser universal para cualquier SOTR, con la que no tengas que aprender ninguna llamada de ningún sistema operativo tiempo real en particular, y sólo usar siempre las llamadas al CMSIS-RTOS. Observad en la figura siguiente a qué nos referimos, el RTOS Kernel Third Party en este caso es el Freertos, pero podría ser cualquier otro, como uCOS (microC/OS en Wikipedia), y el programador sólo tendría que saber las llamadas al CMSIS-RTOS, independientemente del SOTR que usáramos en el nivel más bajo. Es decir, conocer las llamadas al Freertos ofrece la ventaja de que puedes usar ese sistema operativo tiempo real hasta en 35 familias diferentes de microcontroladores, y conocer CMSIS-RTOS te ofrece que puedes usar cualquier SOTR en cualquier Cortex M sin aprender las particularidades de dicho SOTR, usando siempre las mismas llamadas.

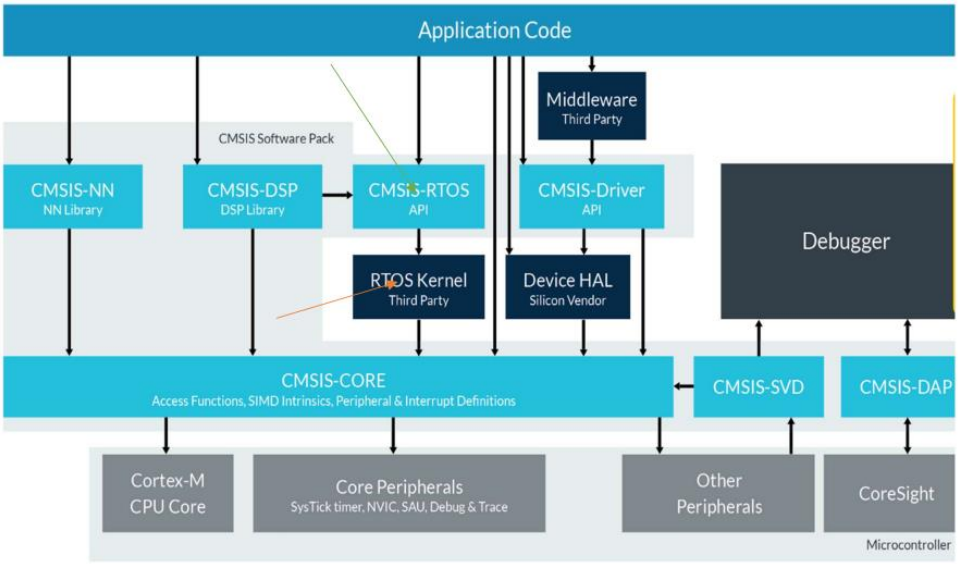
En la imagen de abajo aparecen cuatro actores principales que proporcionan elementos para hacer realidad un producto basado en Cortex:

- ARM que proporciona todo lo que está en azul claro (y ya sabemos que algo más, el core, debugger…).

- El fabricante del Cortex (Silicon Vendor), entre otras cosas es el que fabrica el microcontrolador. También puede proporcionar una interface para los dispositivos (Device HAL, que ya hemos tratado en prácticas anteriores), esta interface pretende simplificar el acceso a los mismos, es una capa por encima del CMSIS-CORE que proporciona ARM.

- La “tercera parte” que proporciona diversas librerías, entre ellas el RTOS (como el Freertos). También el Middleware, que pueden ser, por ejemplo, las librerías de manejo del USB.

- Por último, seguramente nosotros, los desarrolladores de aplicaciones (Application Code).



En las transparencias del tema 7 de teoría se muestran unas referencias bibliográficas que pueden ayudarnos a entender o profundizar algunos conceptos especialmente complejos de esta práctica. Por otra parte, en esta práctica puede ser de gran ayuda el manual de referencia que se encuentra en la web de Keil-ARM sobre el CMSIS-RTOSv2:

https://www.keil.com/pack/doc/CMSIS/RTOS2/html/rtos_api2.html

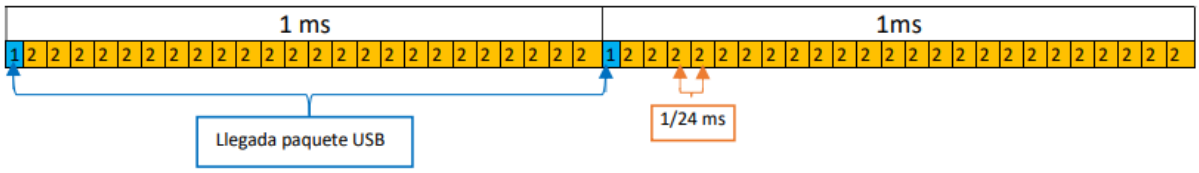
En esa página (hay que desplazarse hacia abajo) vienen todas las funciones de CMSIS-RTOSv2 y ejemplos que nos pueden servir para cuando tengamos dudas a la hora de hacer la práctica. Es curioso porque en el desarrollo de nuestro proyecto el generador de código nos configurará el FREERTOS, pero después, cuando escribamos nuestro código, usaremos siempre las llamadas al CMSIS-RTOS… y éste al FREERTOS.

Estrategias para implementar un sistema tiempo real

Una pregunta habitual que se hace cualquier desarrollador es cuándo conviene utilizar un SOTR. Hemos visto en las clases de teoría que hay tres estrategias para diseñar sistemas empotrados tiempo real, es decir, con requisitos temporales: Bucle scan, primer plano/segundo plano y SOTR.

- La estrategia del bucle scan (o procesamiento secuencial) permite desarrollar sistemas tiempo real simples con requerimientos temporales relativamente laxos, cuando en la fase 4 de la práctica 5 mandamos las notas musicales al PC cada cierto tiempo usamos dicha estrategia, en realidad estamos haciendo una tarea y es fácil, si ahora queremos cambiar de instrumento, hicimos una chapuza y cambiamos sobre la marcha, pero se podría haber testeado los botones para indicar el cambio de instrumento. Observad que si las especificaciones se van complicando conviene ir dividiendo la funcionalidad de nuestro sistema en trozos más o menos independientes, así ha surgido una primera definición de tarea: mandar notas musicales, cambiar instrumento, testear teclas… Definir la funcionalidad como tareas ayuda a manejar especificaciones complejas, después habrá que ver como se lleva a cabo dichas tareas. Lo cierto es que cuando aumenta el número de tareas y sus requerimientos temporales es bastante difícil resolver el problema con bucle scan.

- Primer plano/segundo plano. Si nos fijamos en las fases 2 y 3 de la práctica 5, podríamos decir que el sistema funciona con dos tareas, una que espera a recibir los datos del USB y otra que periódicamente manda cada muestra recibida al conversor digital analógico para ir cambiando la tensión del altavoz y que se reproduzca el sonido. Aquí tenemos un sistema en principio simple, en cuanto a la funcionalidad, pero con requerimientos temporales muy exigentes, este sistema es imposible de realizar (y que suene medianamente bien) con un simple bucle de scan, puesto que cualquier desviación temporal a la hora de sacar las muestras por el DAC implica un mal funcionamiento, y con un bucle de scan es casi imposible conseguir tal "finura temporal" . Pero la solución con la que se resolvió el problema es bastante simple: un bucle de scan que espera recibir datos del USB (tarea 1) y una interrupción periódica (tarea 2) que va sacando las muestras por el DAC en cada momento establecido (algo parecido a lo que se muestra en la siguiente imagen, aunque la tarea 1 ocurre en paralelo con la 2).



El problema lo encontramos cuando queremos comprobar los botones (tarea 3) y mostrar algo en el LCD (tarea 4); si lo que hacemos es añadir simplemente dichas tareas al bucle de scan, junto a la tarea 1, el retraso que imponen las tareas 3 y 4 es tal que dejamos de recibir los datos del USB cada milisegundo. La solución a este problema puede ser hacer otra interrupción periódica mucho más lenta que la correspondiente a la tarea 2 que realice esas nuevas tareas 3 y 4 (puede ser la misma interrupción periódica)... pero cuando hay más tareas que encajar con diferentes tiempos llega un momento en que ya no se dispone de más timers con los que hacer interrupciones periódicas y resolver el problema. Aunque siempre podemos usar un timer maestro que genere un tick temporal y vayamos indicando a qué tarea le toca ejecutarse... llegado a este punto no sigas y usa ya un sistema operativo tiempo real, PUES ESTAMOS CONSTRUYENDO UN PLANIFICADOR Y REINVENTANDO LA RUEDA. Lo único que justificaría no usar un SOTR cuando estamos ante especificaciones complejas, con requerimientos temporales exigentes, es que el microcontrolador no tuviera recursos suficientes (fuera relativamente pequeño) y no pudiéramos recurrir a uno de más prestaciones.

- El usar un SOTR para las fases 2 y 3 de la práctica 5 en realidad no nos hubiera evitado tener que utilizar interrupciones, pues tanto para el USB como para la interrupción periódica del DAC/altavoz son imprescindible, lo que sí nos hubiera resultado más simple es hacer cualquier otra cosa, como por ejemplo comprobar las teclas y escribir en el LCD, como veremos en la última fase de esta práctica. En muchas ocasiones, aunque se pueda llevar a cabo las tareas sin echar mano de un SOTR, cuando el microcontrolador está sobrado de recursos se acaba utilizando el SOTR, pues permite resolver el problema de forma mucho más simple sin perder demasiado tiempo en el desarrollo. Una gran ventaja de los SOTR es que se puede asociar las tareas en las que

se ha dividido las especificaciones a las tareas propias del SOTR (2ª definición), de forma que se realiza una ejecución paralela (seudo-paralela) de las mismas con facilidad para que cada una cumpla los requerimientos temporales (y como ya sabemos, el SOTR además nos proporciona los mecanismos para la comunicación y sincronización entre tareas).

Las tareas

Todas las tareas disponen de una FUNCIÓN PRINCIPAL parecida a lo que sería la función main() en un programa de microcontrolador normal (sin SOTR). Esa función principal comienza declarando las variables que se necesiten, después llamando a las funciones de inicialización o configuración que se requieran, y por último el famoso bucle infinito tipo while(1). Cuando se crea una tarea, ya sea con las funciones del Freertos o con las de CMSIS-RTOS, se deben determinar, al menos, los siguientes parámetros:

- Función principal de la tarea, en CMSIS-RTOS le pone de nombre Start... seguido del nombre de la tarea (pero se podría denominar de cualquier forma (sin el start).

- Nombre de la tarea para depuración.

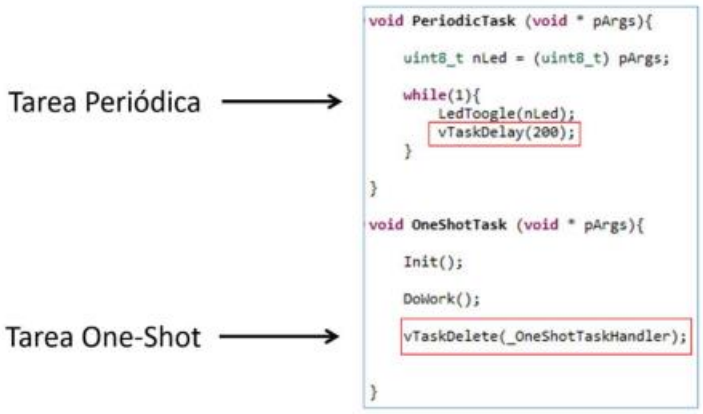
- Tamaño de la memoria RAM que va a utilizar la tarea en palabras del procesador, si el microcontrolador es de 32 bits (como es nuestro caso) y ponemos 128, se ocuparía en RAM 128x4=512 bytes sólo para las variables de la tarea. No podríamos, por ejemplo, declarar un array tipo char de más de 512 bytes.

- Un parámetro que se le pasa a la función principal de la tarea como argumento.

- La prioridad de la tarea.

Y al crear la tarea el SOTR nos devuelve un manejador de la misma, que en el CMSIS-RTOS se suele llamar siempre a estos manejadores “loquesea....handle”. El manejador es importante, pues cualquier operación que queramos hacer sobre la tarea es el parámetro que hay que pasar para que el sistema operativo sepa a qué tarea nos referimos (ya sea suspender o terminar tarea, por ejemplo).

Normalmente se manejan dos tipos de tareas: tareas one-shot y tareas periódicas (o recurrentes). Las tareas one-shot son aquellas que se crean, se realizan y se autodestruyen, es decir, consiste en un código que se realiza una vez y después se lleva a cabo el propio borrado de la tarea. Pero las que más se utilizan son las tareas periódicas.



TODAS LAS TAREAS PERIÓDICAS DEBEN ENTRAR EN BLOQUEO EN ALGÚN MOMENTO. Si la tarea más prioritaria nunca se bloqueara, las menos prioritarias nunca se ejecutarían. Hay que recordar que cuando dos tareas tienen la misma prioridad se comparte el tiempo de ejecución en Freertos (round-robin), a pesar de ello es buena política bloquear las tareas cuando no sean necesarias (e incluso suspenderlas).

Las prioridades en Freertos son muy simples, número más alto mayor prioridad que el más bajo. En CMSIS-RTOS es igual, pero para facilitar la legibilidad asigna etiquetas de texto sobre la prioridad, no debemos liarnos con esto, la etiqueta (más un número detrás, si no tiene número es 0) indica cuál es su nivel de prioridad y ya está. Por ejemplo, osPriorityNormal es más baja prioridad que osPriorityNormal1, y osPriorityBelowNormal7 es más baja que osPriorityNormal.

En SETR1 no hemos tratado nada sobre el sistema de gestión de la memoria de las tareas, en los sistemas empotrados basados en microcontroladores casi siempre se utiliza una gestión estática de la misma, y se ha justificado de diversos puntos de vista (siempre se ejecuta el mismo código y las tareas son las mismas…). Pero en realidad eso no es cierto del todo, ya que se puede crear y borrar tareas, y por tanto cabe aprovechar mejor los recursos de memoria usando una gestión dinámica, es por ello que Freertos permite asignación dinámica de memoria, se estudia en SETR2.

Semáforos

Todos los semáforos sirven para sincronizar tareas en general. Pero desde nuestro punto de vista académico podemos encontrar dos tipos de semáforos:

- Los que sirven para sincronizar el acceso a un recurso compartido (es el uso más clásico de los semáforos). Imaginaros que dos tareas necesitan escribir sobre el mismo display, si ambas lo hacen sin sincronizarse el display acaba no mostrando ningún mensaje legible, mostrando mensajes erróneos o, en el peor de los casos, colgado (como experimentaremos en la fase 3). El semáforo es el contenedor del “token”, cuando una tarea necesita acceder al recurso común pide antes el semáforo, si obtiene el “token” es que tiene acceso al recurso común, si no tiene acceso no obtiene el token, y tendría que esperar a que la tarea que está accediendo devuelva el token para cogerlo

ella. OBSERVAD QUE LA MISMA TAREA QUE HA OBTENIDO EL TOKEN ES LA QUE LO DEVUELVE PARA LIBERAR EL RECURSO.

- Los semáforos que sirven para sincronizar la ejecución de tareas “por disparo”. En este caso también hay una tarea que desbloquea a otra mediante el semáforo, pero en realidad hay una tarea que siempre pone el token en el semáforo y otra que siempre está esperando para recogerlo, pero ésta nunca lo devuelve, sólo lo recoge. Imaginemos una interrupción que está esperando que se pulse un botón, pero es otra tarea la que va a realizar la función necesaria del botón, pues esta tarea estaría bloqueada esperando el semáforo, y cuando la interrupción se produjera pondría el token que a su vez desbloquearía la tarea que estaba esperando, ésta haría lo que corresponda, y volvería a bloquearse esperando que de nuevo la interrupción detectara el botón y pusiera un nuevo token. Siempre la interrupción pondrá el token que dispara la ejecución de la tarea que estaba esperándolo. La interrupción podría usar un tipo de semáforo múltiple que se denomina “semáforo contador”, que permite soltar varios tokens para que la tarea se dispare el número de veces deseado.

En esta práctica vamos a usar el primer tipo de semáforo. Y en la siguiente práctica (la práctica8) el segundo tipo de semáforo (pero sin interrupciones), será un semáforo contador que dependiendo del número de tokens que una tarea le pase hará que otra tarea produzca el número de intermitencias deseadas, al desbloquearse tantas veces como tokens se le pase.

Cuando se inicializa un semáforo los parámetros más importantes son:

- El número máximo de tokens que puede contener.

- El número de tokens activos con los que se inicializa (veremos que a nosotros nos interesa empezar con cero, pero que el generador de código pone siempre el máximo).

- El manejador para utilizar el semáforo, es el parámetro que se pasan en las funciones para que el SOTR sepa a qué semáforo nos referimos.

Las dos funciones básicas para utilizar los semáforos en CMSIS-STOR son:

```
- osSemaphoreRelease(loqueseaHandle);
- osSemaphoreAcquire (loqueseaHandle,0xFFFFFFFF);
```

La primera pone el semáforo (token) correspondiente al manejador que se le pasa y la segunda espera el semáforo del manejador correspondiente un determinado tiempo, si ese tiempo es cero, no lo espera (sabremos si tenemos o no el token por lo que devuelve la función). Y si en el parámetro de tiempo ponemos 0xFFFFFFFF es un tiempo muy largo… si no obtiene el token se queda bloqueada indefinidamente.

Colas de mensaje

Las colas de mensajes son muy prácticas para pasar información entre tareas (como es natural son particularmente interesantes en las comunicaciones tipo USB, serie asíncrona, ethernet…). Las colas y los semáforos tienen gran parecido en lo que

respecta al funcionamiento, un semáforo es como una cola que no pasa mensajes. En la inicialización de las colas los parámetros fundamentales son:

- El tamaño de la cola. Determina cuantos ítems puede contener la cola antes de saturarse.

- El tamaño de cada ítem, si los ítems son variables tipo char, el tamaño del mismo será un byte, si es tipo short será 2 bytes y si es tipo int será 4 bytes. Peros lo ítems pueden ser también arrays completos, por ejemplo si declaramos un char pepe[10], el ítem será de tamaño 10 bytes. Por tanto, una cola de tamaño 4, que cada ítem sea del tipo char pepe[10], ocuparía al menos 40 bytes.

- El manejador de la cola vuelve ser el elemento que se usa en las funciones para referenciar la cola. Las dos funciones principales de las colas son:

```
- osMessageQueuePut(colaloqueseaHandle,&conta,0,0);
- osMessageQueueGet(colaloqueseaHandle, &respuesta, 0, 0xFFFFFFFF);
```

La primera función mete un mensaje en la cola correspondiente al manejador que se le pasa, el puntero al dato que se introduce es el siguiente parámetro. El tercer parámetro permite cambiar el orden de la cola, esto normalmente no se utiliza (las colas suelen utilizarse como tipo FIFO, sin saltarse el orden de la cola), por lo que se suele poner a cero. Y el último parámetro indica un timeout… si la cola se ha llenado la tarea que mete el dato se queda bloqueada esperando a que haya hueco para colocar el siguiente dato, el tiempo de bloqueo viene dado por ese cuarto parámetro de la función. Si ponemos cero no se espera nunca, incluso si no hay hueco para meter el dato.

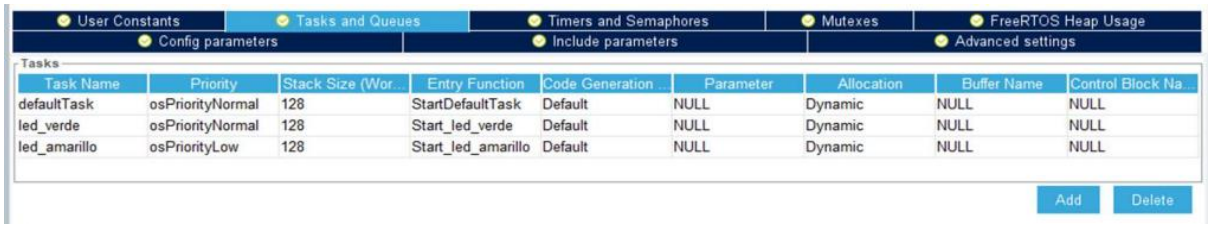
La segunda función es la encargada de recoger datos de la cola, los parámetros son prácticamente iguales, en el ejemplo que hemos puesto esperamos el tiempo que haga falta a que haya un dato, el tiempo es muy grande y la tarea se quedará bloqueada mientras no haya dato.

3. Desarrollo de la práctica

▪ En la fase 1, configuraremos tareas y utilizaremos retrasos temporales para crear intermitencias continuas con los LEDs. Comenzaremos creando un nuevo proyecto STM32 usando el selector de placa y siguiendo los mismos pasos que en prácticas anteriores. Usaremos el LED2 (verde) conectado al pin PB14, y el LED3 (amarillo) conectado al pin PC9. Este último GPIO también está conectado al LED4 (azul), pero invertido respecto al LED3, de modo que al encender uno, el otro se apaga, y viceversa. En la configuración, hemos etiquetado estos pines como “verde” y “amarillo” para mayor comodidad.

Para generar el proyecto con las tareas, primero debemos configurar FREERTOS. Para ello, seguiremos esta ruta: “Middleware” → “FREERTOS” → “Interface” → “CMSIS_V2” y en la misma pestaña de “FREERTOS” → “Tasks and Queues” → “Tasks” → “Add”. Pulsamos “Add” y se abre una ventana para declarar la nueva tarea. Modificaremos “Task Name”, “Priority” y “Entry Function”, asignando “led_verde”, “osPriorityNormal” y

“Start_led_verde” respectivamente. Luego, añadiremos otra tarea para el LED amarillo, pero en este caso, configurando la prioridad en bajo, quedando tal que así:



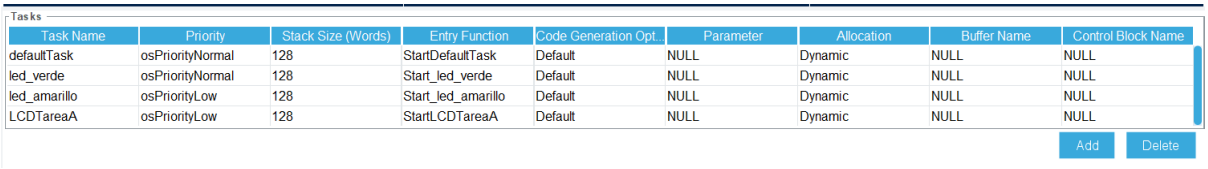
Cuando generemos el código, iremos al “main.c” y escribiremos las funciones “Start_led_verde” y “Start_led_amarillo”:

```
void Start_led_verde(void *argument)
{
 /* USER CODE BEGIN Start_led_verde */
 /* Infinite loop */
 for(;;)
 {

        HAL_GPIO_WritePin(GPIOB, verde_Pin, GPIO_PIN_SET);
        osDelay(300);
        HAL_GPIO_WritePin(GPIOB, verde_Pin, GPIO_PIN_RESET);
        osDelay(300);

 }
 /* USER CODE END Start_led_verde */
}

void Start_del_amarillo(void *argument)
{
 /* USER CODE BEGIN Start_del_amarillo */
 /* Infinite loop */

for(;;)
 {
        HAL_GPIO_WritePin(GPIOC, amarillo_Pin, GPIO_PIN_SET);
        osDelay(200);
        HAL_GPIO_WritePin(GPIOC, amarillo_Pin, GPIO_PIN_RESET);
        osDelay(200);

 }
 /* USER CODE END Start_del_amarillo */
}
```

Al ejecutar, veremos cómo los dos LEDs parpadean a ritmos diferentes y ambas tareas se ejecutan en paralelo. Si queremos que las intermitencias se hagan al mismo ritmo, podemos cambiar todos los retrasos en las funciones principales de las dos tareas a “osDelay(300)”.

▪ En la fase 2, continuaremos con una copia de la práctica anterior. En esta fase, crearemos un proyecto con tres tareas: dos iguales a las de la fase 1 y otra que imprima un mensaje en el LCD. Primero, añadiremos las librerías necesarias para el funcionamiento del LCD, que podemos obtener de la Práctica 3.

Una vez hecho esto, volveremos a la configuración del proyecto para añadir una nueva interrupción. Esta se llamará "LCDTareaA" y tendrá una prioridad baja, quedando tal que así:


Tasks

Task Name	Priority	Stack Size (Words)	Entry Function	Code Generation Opt..	Parameter	Allocation	Buffer Name	Control Block Name
defaultTask	osPriorityNormal	128	StartDefaultTask	Default	NULL	Dynamic	NULL	NULL
led_verde	osPriorityNormal	128	Start_led_verde	Default	NULL	Dynamic	NULL	NULL
led_amarillo	osPriorityLow	128	Start_led_amarillo	Default	NULL	Dynamic	NULL	NULL
LCDTareaA	osPriorityLow	128	StartLCDTareaA	Default	NULL	Dynamic	NULL	NULL

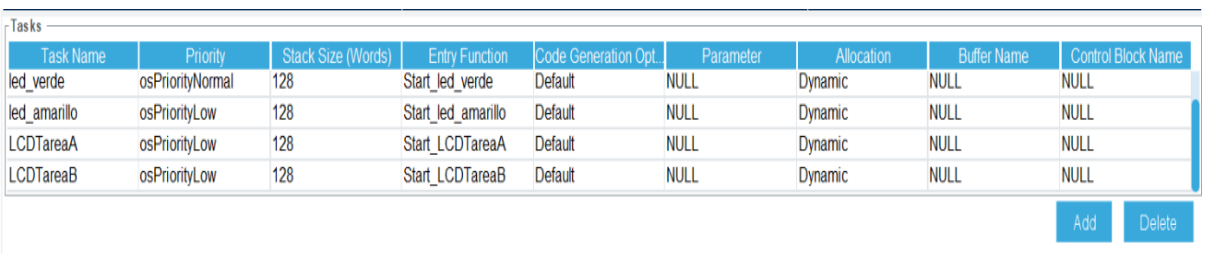
Add Delete

Luego, vamos a "main.c" y escribiremos el código de la función "StartLCDTareaA":

```
void StartLCDTareaA(void *argument)
{
 /* USER CODE BEGIN StartLCDTareaA */
 /* Infinite loop */
 int contador=0;
 lcd_reset();
 lcd_display_settings(1,0,0);
 lcd_clear();

 for(;;)
 {
        moveToXY(0,0);
        lcd_print("TAREA A:");
        writeIntegerToLCD(contador++);
        osDelay(200);
 }
 /* USER CODE END StartLCDTareaA */
}
```

▪ En la fase 3, continuaremos con una copia de la práctica anterior. En esta fase, crearemos un proyecto con cuatro tareas: dos iguales a las de la fase 1 y dos más que impriman un mensaje en el display LCD, cada una en una línea diferente del LCD.

Primero, volveremos a la configuración del proyecto para añadir una nueva interrupción. Esta se llamará "LCDTareaB" y tendrá una prioridad normal, quedando tal que así:


Tasks

Task Name	Priority	Stack Size (Words)	Entry Function	Code Generation Opt..	Parameter	Allocation	Buffer Name	Control Block Name
led_verde	osPriorityNormal	128	Start_led_verde	Default	NULL	Dynamic	NULL	NULL
led_amarillo	osPriorityLow	128	Start_led_amarillo	Default	NULL	Dynamic	NULL	NULL
LCDTareaA	osPriorityLow	128	Start_LCDTareaA	Default	NULL	Dynamic	NULL	NULL
LCDTareaB	osPriorityLow	128	Start_LCDTareaB	Default	NULL	Dynamic	NULL	NULL

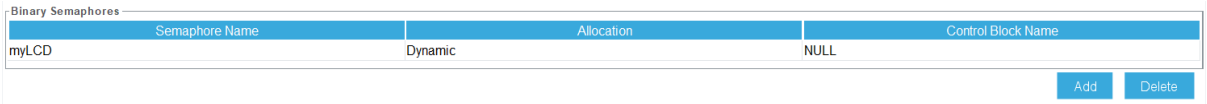
Add Delete

Una vez añadida la interrupción, nos enfrentaremos a varios problemas:

1. **Exceso de memoria utilizada por el SOTR:** Hemos configurado TOTAL_HEAP_SIZE en 3000 bytes, pero la suma del consumo de memoria de las tareas creadas es de 3160 bytes. Para solucionar esto, podemos reducir el consumo de memoria de las tareas o aumentar el valor de TOTAL_HEAP_SIZE. Dado que este microcontrolador tiene 128 KB de memoria, podemos permitirnos aumentar dicho parámetro a 30000 bytes.
2. **Errores en la tarea B:** Si hemos copiado el código de la tarea A para la tarea B, puede haber elementos que solo deben ser declarados una vez. Después de eliminar esas líneas duplicadas, puede que siga fallando debido a que la tarea B debe inicializarse después de la tarea A, ya que la inicialización del LCD se realiza en la tarea A. El código final de la tarea B es el siguiente:

```
void Start_LCDTareaB(void *argument)
{
 /* USER CODE BEGIN Start_LCDTareaB */
 /* Infinite loop */
 int contadorB = 0;
 osDelay(20);
 for(;;)
 {

        moveToXY(1, 0);
        lcd_print("TAREA B: ");
        writeIntegerToLCD(contadorB++);
        osDelay(100);

 }
 /* USER CODE END Start_LCDTareaB */
}
```

3. **Problemas de acceso al recurso compartido:** Aunque parezca que todo funciona, el programa falla con el tiempo. Esto se debe a que ambas tareas acceden al LCD, que es un recurso compartido. Para solucionar esto, utilizaremos un semáforo. Volveremos a la herramienta de configuración para crearlo siguiendo esta ruta: “Middleware” → “FREERTOS” → “Interface” → “CMSIS_V2” y en esa misma pestaña “FREERTOS” → “Timers and Semaphores” → “Binary Semaphores” → “Add”. Al pulsar “Add”, se abrirá una ventana para declarar el nuevo semáforo. Solo modificaremos “Semaphore Name”, donde pondremos “myLCD”, quedando así:.



Una vez hecho esto, deberemos ir a las funciones “Start_LCDTareaA” y “Start_LCDTareaB” y añadir el código:

- `osSemaphoreAcquire (myLCDHandle,0xFFFFFFFF);`
- `osSemaphoreRelease(myLCDHandle);`

Las funcionas quedarían tal que así:

```
void Start_LCDTareaA(void *argument)
{
 /* USER CODE BEGIN Start_LCDTareaA */
 /* Infinite loop */
 int contadorA=0;
 lcd_reset();
 lcd_display_settings(1, 0, 0);
 lcd_clear();

 for(;;)
 {
        osSemaphoreAcquire (myLCDHandle,0xFFFFFFFF);
        moveToXY(0, 0);
        lcd_print("TAREA A: ");
        writeIntegerToLCD(contadorA++);
        osSemaphoreRelease(myLCDHandle);
        osDelay(200);
 }
 /* USER CODE END Start_LCDTareaA */
}

void Start_LCDTareaB(void *argument)
{
 /* USER CODE BEGIN Start_LCDTareaB */
 /* Infinite loop */
 int contadorB = 0;
 osDelay(20);
 for(;;)
 {
        osSemaphoreAcquire (myLCDHandle,0xFFFFFFFF);
        moveToXY(1, 0);
        lcd_print("TAREA B: ");
        writeIntegerToLCD(contadorB++);
        osSemaphoreRelease(myLCDHandle);
        osDelay(100);
 }
 /* USER CODE END Start_LCDTareaB */
}
```

Ahora podemos ejecutar el código y ver que todo funciona correctamente. Sin embargo, hay un pequeño problema: el contador de la tarea B no funciona al doble de la velocidad de la tarea A, como debería. Esto se debe a las prioridades de las tareas y se soluciona

aumentando la prioridad de la tarea B. Una vez modificado esto, el código funcionará perfectamente.

▪ En la fase 4, continuamos con una copia de la práctica anterior. En esta fase, realizaremos la fase 3 de la Práctica 5 utilizando STM32CubeIDE y FreeRTOS. Primero, agregamos las librerías necesarias para el funcionamiento del USB de clase audio, que vienen adjuntas en la práctica.

Una vez agregadas, configuramos un dispositivo USB de clase audio en el proyecto. Para esto, seguimos la ruta: Connectivity → USB_OTG_FS → Mode → Device_Only y en esa misma pestaña de USB_OTG_FS → NVIC Settings → USB OTG FS global interrupt → Enable → ✓. Luego, sin salir del configurador, vamos a Middleware → USB_DEVICE → Class For FS IP → Audio Device Class y en esa pestaña de USB_DEVICE → Parameter Settings → USBD_AUDIO_FREQ → 24000.

Configuramos un timer contador de recarga automática para producir una interrupción periódica fija de 24000Hz. Para esto, seguimos la ruta: Timers → TIM7 → Activated → ✓ y en esa misma pestaña de TIM7 → Parameter Settings → Counter Period → 3320. Luego, en esa misma pestaña de nuevo, NVIC Settings → TIM7 global interrupt → Enable → ✓. Ahora configuramos el conversor digital analógico: Analogs → DAC1 → OUT2 mode → Connected to external pin only.

Generado el código, modificamos partes del mismo. Primero, en Usbd_audio.c, comentamos una definición y declaramos algunas variables:

```
#define AUDIO_SAMPLE_FREQ(frq)      (uint8_t)(frq), (uint8_t)((frq >> 8)), (uint8_t)((frq >> 16))

/*
#define AUDIO_PACKET_SZE(frq)(uint8_t)(((frq * 2U * 2U)/1000U) & 0xFFU), \(uint8_t)((((frq * 2U * 2U)/1000U) >> 8) & 0xFFU)
*/
#define AUDIO_PACKET_SZE(frq)(uint8_t)(((frq * 2U)/1000U) & 0xFFU), \(uint8_t)((((frq * 2U)/1000U) >> 8) & 0xFFU)

/**
 * @}
 */

uint8_t disponible=0;
int16_t *audiobuf;
int16_t mibuf[256][24];
uint8_t jj=0;
```

En el mismo fichero, en la sección /* USB Speaker Audio Type III Format Interface Descriptor */, cambiamos un 0x02 por un 0x01:

```
/* USB Speaker Audio Type III Format Interface Descriptor */
 0x0B,                       /* bLength */
 AUDIO_INTERFACE_DESCRIPTOR_TYPE,     /* bDescriptorType */
 AUDIO_STREAMING_FORMAT_TYPE,       /* bDescriptorSubtype */
 AUDIO_FORMAT_TYPE_I,             /* bFormatType */
```

```
 0x01, //0x02,                  /* bNrChannels */
 0x02,                        /* bSubFrameSize :  2 Bytes per frame (16bits) */
 16,                         /* bBitResolution (16-bits per sample) */
 0x01,                        /* bSamFreqType only one frequency supported */
 AUDIO_SAMPLE_FREQ(USBD_AUDIO_FREQ),   /* Audio sampling frequency coded on
3 bytes */
 /* 11 byte*/
```

También modificamos la función USBD_AUDIO_DataOUT en ese fichero:

```
static uint8_t USBD_AUDIO_DataOut(USBD_HandleTypeDef *pdev, uint8_t epnum)
{
 USBD_AUDIO_HandleTypeDef *haudio;
 haudio = (USBD_AUDIO_HandleTypeDef *) pdev->pClassData;
 int i;

 if (epnum == AUDIO_OUT_EP)
 {

        audiobuf = (int16_t *) &haudio->buffer[haudio->wr_ptr];

        for (i = 0; i < 24; i++)

                mibuf[jj][i] = audiobuf[i];
                jj++;
                disponible = 1;
                /* Increment the Buffer pointer or roll it back when all buffers are full */
                haudio -> wr_ptr += AUDIO_OUT_PACKET;

                if (haudio -> wr_ptr == AUDIO_TOTAL_BUF_SIZE)
                {
                        /* All buffers are full: roll back */
                        haudio -> wr_ptr = 0U;
x
                }
                /* Prepare Out endpoint to receive next audio packet */
                USBD_LL_PrepareReceive(pdev, AUDIO_OUT_EP, &haudio ->
buffer[haudio -> wr_ptr], AUDIO_OUT_PACKET);

 }

 return USBD_OK;

}
```

Luego, vamos al fichero stm32l4xx_it.c y declaramos las variables a usar en la sección /* USER CODE BEGIN PV */:

```
/* USER CODE BEGIN PV */
extern int16_t *audiobuf;
extern unsigned char disponible;
extern DAC_HandleTypeDef hdac1; //manejador del DAC que se declara en el fichero main.c
extern int16_t mibuf[256][24];
extern uint8_t jj;
float ganancia = 1;
uint8_t efecto = 100;
/* USER CODE END PV */
```

Hay que hacer un último cambio en este fichero, y es la función “TIM7_IRQHandler”:

```
void TIM7_IRQHandler(void)
{
	/* USER CODE BEGIN TIM7_IRQn 0 */
	static int i=0;
	float dato;

	if (disponible==1)
	{

		dato = ((float) audiobuf[i] + (float) (ganancia * mibuf[(unsigned char)(jj - efecto)][i]) + (float) (ganancia * mibuf[(unsigned char)(jj - efecto - 1)][i])) / 3;
		dato = dato * 2048.0 / 32768; //escalado de 16 bits del PC a 12
		dato = dato + 2048; //centrado positivo DAC
		i++;

		if (i >= 24)
		{

			i = 0;
			disponible = 0;

		}

		HAL_DAC_SetValue(&hdac1, DAC1_CHANNEL_2, DAC_ALIGN_12B_R, (unsigned short int) dato);

	}

	__HAL_TIM_CLEAR_IT(&htim7, TIM_IT_UPDATE);
	/* USER CODE END TIM7_IRQn 0 */

	/* USER CODE BEGIN TIM7_IRQn 1 */
```

```
    /* USER CODE END TIM7_IRQn 1 */
}
```

Finalmente, en el main.c, agregamos dos líneas de código en la sección /* USER CODE BEGIN 2 */:

```
/* USER CODE BEGIN 2 */
  HAL_DAC_Start(&hdac1, DAC1_CHANNEL_2);
  HAL_TIM_Base_Start_IT(&htim7);
/* USER CODE END 2 */
```

Ejecutamos el programa y verificamos que todo funcione correctamente. Los LEDs seguirán con su intermitencia, las tareas A y B continuarán, y podremos reproducir audio a través del altavoz conectado a la placa.

4. Conclusión

Se han completado satisfactoriamente todas las fases debido a que se han seguido las explicaciones que figuraban en la hoja de la práctica. Personalmente, la práctica ha sido fácil y +divertida, ya que, a pesar de pensar que no se podrían conectar más interacciones simultáneas, hemos conseguido hacerlo. También considero que ha sido muy útil porque hemos aprendido a usar colas de prioridad y semáforos, así como la ejecución de múltiples tareas simultáneamente.