

PRÁCTICA 8

Índice

1. Objetivos.
2. Introducción.
3. Desarrollo de la práctica.
4. Conclusión.

1. Objetivos

- FASE 1: Creación de tareas y uso de los retrasos temporales: hacer intermitencias continuas con los leds (igual que práctica 7).
- FASE 2: Utilización de semáforos contadores: pulsar los botones un número determinado de veces y provocar el mismo número de intermitencias en los leds.
- 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: Hacer un juego de capacidad de observación con los dos leds y los dos botones, usando los elementos de las fases anteriores.

2. Introducción

En esta práctica vamos a comprobar el funcionamiento de los semáforos contadores y de las colas. Para hacer la práctica vamos a usar elementos muy parecidos a los de la práctica 1, el STM32CubeIDE como sistema de desarrollo y una placa de Cortex M3 simulada con Qemu, además del sistema operativo Freertos (con CMSIS-RTOS de interfaz de programación). Se va a utilizar el simulador en lugar de la placa real para que se pueda ir devolviendo los equipos prestados sin necesitarlos para la última práctica, y poder entregar la memoria de práctica cuando corresponda.

No crean que encontrar un simulador gratuito que funcione correctamente con Cortex M y Freertos es fácil. En el Qemu el principal problema lo encontramos en la simulación de la gestión de prioridades de las interrupciones. Por una parte, utilizan los 8 bits del registro de prioridades IPR, cuando estos Cortex sólo usan los 4 bits superiores (el simulador trata el IRP como ARM y no como Cortex M), y por otra, no simulan características como son PriorityGroup o incluso la coma flotante hardware (en el caso del Qemu). Por lo que cuando usamos el Freertos en nuestros programas suele dar

bastantes fallos. Ya veremos que precisamente el error en la simulación de las prioridades del Qemu nos obliga a comentar una línea en el código de Freertos que testea los niveles de prioridad del microcontrolador en cuestión, eso no deberíamos hacerlo si estuviéramos con el Cortex M3 de verdad. La placa que vamos a usar de Qemu va a ser una diferente a la de la práctica 1, con la que podamos hacer algo más, no mucho más, pues el simulador es muy limitado, simplemente vamos a poder controlar dos botones y dos leds (en lugar de un botón y un led de la práctica uno).

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.

En esta práctica vamos a realizar un pequeño juego que denominamos “simoncín” (basado en el famoso juego de memoria y observación “simón dice”). Consiste en que el microcontrolador va a generar una secuencia aleatoria de parpadeos con dos leds y el jugador tiene que indicar la suma total de veces que se encienden los dos leds, si acierta se señala con un led verde, si fracasa con el led amarillo. Evidentemente es un juego simple que se podría hacer fácilmente sin necesidad de un SOTR, y que cabría soluciones mucho más cortas de las que se plantean en esta práctica. Pero el juego sólo es la excusa para ver cómo funcionan las tareas, los retrasos, los semáforos contadores y las colas. El objetivo académico es ilustrar el funcionamiento de los semáforos contadores y las colas. La interfaz de usuario de dos leds y dos botones no da para mucho a la hora de hacer prácticas.

IMPORTANTE:

- Si habéis tenido que borrar el Qemu, por falta de espacio en disco, tendréis que instalarlos de nuevo, acudid a la práctica 1 para ver cómo se realiza dicho proceso.
- En esta práctica no se van a repetir los detalles de uso del STM32CubeIDE ni de su generador de código que ya se indicaron en prácticas anteriores. Por ejemplo, diremos que vamos a iniciar un proyecto nuevo, en qué pestaña picar y qué hacer se sabe de otras prácticas.
- Recordar también lo explicado en las clases de teoría y en la práctica anterior sobre SOTR.

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 (la práctica presencial sobre SOTR trataba dicho problema). 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 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 contador 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(semaamarilloHandle);  
  
- osSemaphoreAcquire (semaamarilloHandle,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. Pero los ítems pueden ser también arrays completos, por ejemplo si declaramos un char titor[10], el ítem será de tamaño 10 bytes. Por tanto, una cola de tamaño 4, que cada ítem sea del tipo char titor[10], ocuparía al menos 40 bytes.
- El manejador de la cola vuelve a ser el elemento que se usa en las funciones para referenciar la cola. Las dos funciones principales de las colas son:

```
- osMessageQueuePut(num_pulsacionesHandle,&conta,0,0);
```

```
- osMessageQueueGet(num_pulsacionesHandle, &respuesta, 0, 0xFFFFFFFF);
```

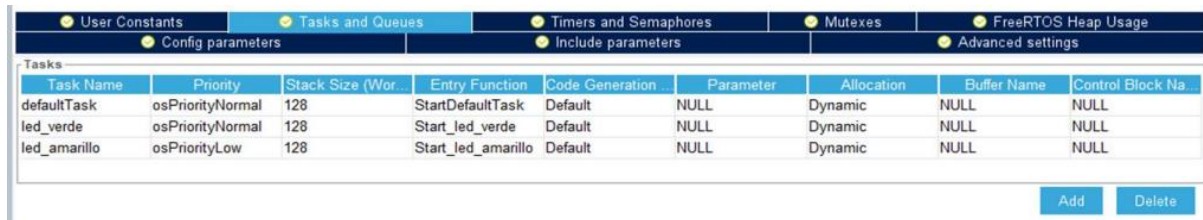
La primera función mete un mensaje en la cola correspondiente al manejador que se le pasa, el puntero al dato que se introduce es el siguiente parámetro. El tercer parámetro permite cambiar el orden de la cola, esto normalmente no se utiliza (las colas suelen utilizarse como tipo FIFO, sin saltarse el 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. En nuestro caso usamos este mecanismo, y si no hay hueco simplemente el dato se pierde.

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

3. Desarrollo de la práctica

■ En esta primera fase crearemos tareas y usaremos retrasos temporales para hacer intermitencias continuas con los LEDs. Para comenzar, crearemos un nuevo proyecto STM32 utilizando el selector de placa y siguiendo los mismos pasos que en prácticas anteriores. Utilizaremos 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 manera que cuando encendemos 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. Seguiremos esta ruta: “Middleware” -> “FREERTOS” -> “Interface” -> “CMSIS_V2”. En la misma pestaña de “FREERTOS” -> “Tasks and Queues” -> “Tasks” -> “Add”. Al pulsar en “Add”, se abrirá una ventana para declarar la nueva tarea. Solo modificaremos “Task Name”, “Priority” y “Entry Function”, donde pondremos “led_verde”, “osPriorityNormal” y “Start_led_verde” respectivamente. Después, pulsamos de nuevo en “Add” y rellenamos los mismos campos para el LED amarillo, pero con una prioridad baja, quedándonos así:



Task Name	Priority	Stack Size (Words)	Entry Function	Code Generation	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

Una vez generado 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_led_amarillo(void *argument)
{
    /* USER CODE BEGIN Start_led_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 */
}

```

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

■ En la fase 2, continuaremos donde lo dejamos en la fase anterior. Pulsaremos los botones un número determinado de veces para provocar el mismo número de intermitencias en los LEDs. Seguiremos esta ruta: “Middleware” -> “FREERTOS” -> “Timers and Semaphores” -> “Counting Semaphores” -> “Add”. Al pulsar en “Add”, se abrirá una ventana para declarar la nueva tarea. Solo modificaremos “Semaphore Name” y “Count”, donde pondremos “semaverde” y “5” respectivamente. Luego, repetimos el proceso para el semáforo amarillo, quedándonos así:

Semaphore Name	Count	Allocation	Control Block Name
semaamarillo	5	Dynamic	NULL
semaverde	5	Dynamic	NULL

Después, iremos al “main.c” y escribiremos el siguiente código en el apartado “/* creation of semaverde */” y “/* creation of semaamarillo */”:

```

/* Create the semaphores(s) */
/* creation of semaamarillo */
semaamarilloHandle = osSemaphoreNew(5, 5, &semaamarillo_attributes);

/* creation of semaverde */
semaverdeHandle = osSemaphoreNew(5, 5, &semaverde_attributes);

```

También modificaremos ambos bucles de las tareas de los LEDs, quedándonos así:

```

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

        osSemaphoreAcquire (semaverdeHandle,0xFFFFFFFF);
    }
}

```

```

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

}
/* USER CODE END Start_led_verde */
}

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

        osSemaphoreAcquire (semaamarilloHandle,0xFFFFFFFF);
        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_led_amarillo */
}

```

Finalmente, escribiremos la función que controlará los LEDs:

```

void Start_control_leds(void *argument) {
    /* USER CODE BEGIN Start_control_leds */
    /* Infinite loop */
    unsigned short conta, i;
    while(HAL_GPIO_ReadPin(GPIOC, tamper_Pin)==0) osDelay(50);

    for (;;) {

        if (HAL_GPIO_ReadPin(GPIOA, wkup_Pin) == 1) {

            conta++;

            while (HAL_GPIO_ReadPin(GPIOA, wkup_Pin) == 1) osDelay(10);

        }

        if (HAL_GPIO_ReadPin(GPIOC, tamper_Pin) == 0) {

            for (i = 0; i < conta; i++) osSemaphoreRelease(semaverdeHandle);

```

```

        for (i = 0; i < conta; i++) osSemaphoreRelease(semaamarilloHandle);

        conta = 0;

    }

    osDelay(10);
}
/* USER CODE END Start_control_leds */
}

```

Tras realizar todo lo anterior, podremos ejecutar el código y comprobar que funciona correctamente.

- En esta fase 3, usaremos una cola de mensajes y pulsaremos los botones un número determinado de veces para indicar si el número es par o impar con el LED verde o amarillo. Primero, iremos de nuevo a la configuración del proyecto y añadiremos una nueva interrupción llamada “respuesta” con una prioridad alta, quedándonos 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	osPriorityNormal	128	Start_led_amarillo	Default	NULL	Dynamic	NULL	NULL
control_led	osPriorityNormal	128	Start_control_led	Default	NULL	Dynamic	NULL	NULL
respuesta	osPriorityHigh	128	Start_respuesta	Default	NULL	Dynamic	NULL	NULL

Luego, iremos al “main.c” y escribiremos el siguiente código en el apartado “Start_respuesta”:

```

void Start_respuesta(void *argument)
{
    /* USER CODE BEGIN Start_respuesta */
    /* Infinite loop */
    unsigned short conta;
    for(;;)
    {
        if (HAL_GPIO_ReadPin(GPIOA, wkup_Pin) == 1) {
            conta++;
            while (HAL_GPIO_ReadPin(GPIOA, wkup_Pin) == 1)
                osDelay(10);
        }
        if (HAL_GPIO_ReadPin(GPIOC, tamper_Pin) == 0 && conta!=0) {
            osMessageQueuePut(num_pulsacionesHandle,&conta,0,0);
            conta = 0;
        }
        osDelay(10);
    }
    /* USER CODE END Start_respuesta */
}

```



```
}
```

También realizaremos cambios en la función de control de los LEDs, quedándonos así:

```
void Start_control_leds(void *argument) {  
/* USER CODE BEGIN Start_control_leds */  
/* Infinite loop */  
unsigned short respuesta; //enteros de 16 bits sin signo  
while (HAL_GPIO_ReadPin(GPIOC, tamper_Pin) == 0)  
    osDelay(100);  
osMessageQueueReset (num_pulsacionesHandle);  
  
for (;;) {  
    osMessageQueueGet(num_pulsacionesHandle, &respuesta, 0, 0xFFFFFFFF);  
    if (respuesta % 2 == 0) {  
        HAL_GPIO_WritePin(GPIOC, amarillo_Pin, GPIO_PIN_RESET);  
        HAL_GPIO_WritePin(GPIOC, verde_Pin, GPIO_PIN_SET);  
    } else {  
        HAL_GPIO_WritePin(GPIOC, verde_Pin, GPIO_PIN_RESET);  
        HAL_GPIO_WritePin(GPIOC, amarillo_Pin, GPIO_PIN_SET);  
    }  
    osDelay(2000);  
    HAL_GPIO_WritePin(GPIOC, verde_Pin, GPIO_PIN_RESET);  
    HAL_GPIO_WritePin(GPIOC, amarillo_Pin, GPIO_PIN_RESET);  
    osDelay(10); //dejar esta línea fuera del if en la fase 4  
}  
/* USER CODE END Start_control_leds */  
}
```

Tras realizar esta implementación, podremos ejecutar el código y ver que funciona correctamente, mostrando con los LEDs si las pulsaciones han sido pares o impares.

4. Conclusión

Se han completado satisfactoriamente todas las fases, siguiendo las explicaciones de la práctica, aunque debo admitir que la cuarta fase me ha resultado un poco complicada y no he logrado entenderla del todo.

En cuanto a mi experiencia personal, encontré la práctica fácil y muy entretenida. Considero que esta práctica ha sido muy útil, ya que nos ha permitido profundizar en el uso de colas de prioridad y semáforos, ampliando así nuestros conocimientos en programación concurrente.