Universidad Tecnológica Nacional

Facultad Regional Avellaneda

Departamento de Ingeniería Electrónica

Cátedra: Técnicas Digitales III

Título del Trabajo Práctico:	Trabajo Práctico Nº3: Contenidos analizados: • Gestión y API correspondiente a Semáforos • Gestión y API correspondiente a Interrupciones • Exclusión Mutua • Inversión de Prioridades • Tareas Guardianas
Autores:	
Resumen:	
El objetivo de este trabajo practico es aplicar en un problema concreto el concepto de semáforos en FreeRTOS, ver su utilidad para sincronismo de tareas y diferenciar los contextos de aplicación del semáforo binario, counting y mutex.	

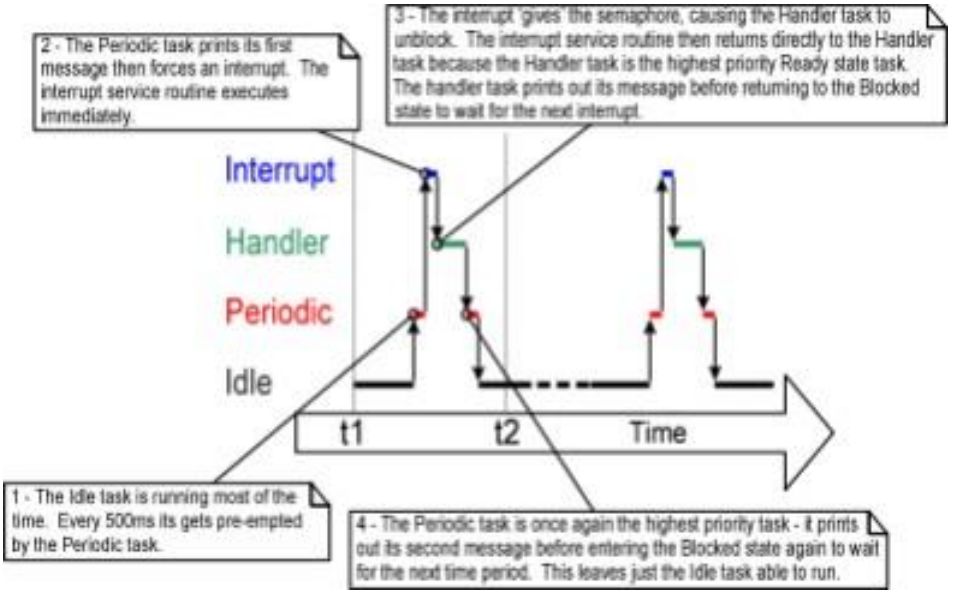
Sistemas Operativos en Tiempo Real: FREERTOS (Trabajo Práctico Nº3)

Parte Teórica:

1) Desarrolle el significado de "Procesamiento diferido por interrupción" utilizando un ejemplo concreto implementado con código en FREERTOS. Desarrolle una consigna que materialice el ejemplo y luego resuélvalo (El ejemplo debe estar desarrollado bajo la plataforma actual de la cátedra).

2) Explique el uso del parámetro "**pxHigherPriorityTaskWoken**".

3) Utilice los apuntes de la cátedra para desarrollar el contexto del siguiente gráfico:



¿Cuál es la problemática que se analiza y cuál es el recurso que permite solucionar la problemática? Desarrolle.

4) Explique los diferentes casos de Exclusión Mutua existentes en FREERTOS. En cada caso proponga un ejemplo aplicado a la plataforma actual utilizada en la cátedra.

5) Secciones Críticas. Desarrolle un ejemplo de cada caso utilizando la plataforma actual de la cátedra. Evalúe las ventajas y desventajas de cada caso en forma crítica.

6) Explicar el concepto de "Exclusión Mutua" y el concepto de "Punto Muerto".

7) ¿Cuál es el uso que poseen las Tareas Guardianas? ¿Cuál es la problemática que soluciona? Proporcione un ejemplo implementado con la plataforma utilizada por la cátedra actualmente.

Parte Práctica:

1) A) Realizar el siguiente trabajo práctico en la sección correspondiente al repositorio en GitHub "6.trabajos_practicos" realizando un commit denominado "tp3_23-06-07".

B) Se desea armar un contador de frecuencia inyectando la señal de un generador de pulsos a través de una interrupcion externa.

- Se utilizará un semáforo counting (máximo 5000) como recurso para contabilizar la cantidad de veces que el microcontrolador entró en la interrupción (cantidad de pulsos del generador).
- Se debe implementar una tarea que verifique periódicamente la cantidad de pulsos contados en un segundo y muestre en un LCD 16x2 con I2C el mensaje "Cant de pulsos xxxx".

AYUDA: Ya cuentan con los archivos lcd1602_i2c.c y lcd1602_i2c.h (que manejan la inicialización y comandos básicos para el LCD). Algunas funciones útiles para esta consigna incluyen:

- lcd_init(hi2c1, address) para la Bluepill o lcd_init(I2C0, address) para LPC1769. Estas inicializan el LCD suponiendo que el I2C1 o I2C0 respectivamente fueron inicializados correctamente. El valor de address corresponde a la dirección de 7 bits del PC8574 que usualmente es 0x27 o 0x3f.
- lcd_clear() para poder limpiar la pantalla.
- lcd_string(str) para pasarle una cadena de caracteres que se escriba en el display. Pueden armar las cadenas de caracteres incluyendo stdio.h al proyecto y haciendo uso del sprintf.

2) CONDICIÓN DE PROMOCIÓN: Hacer las modificaciones necesarias para que una segunda tarea escriba en la primera línea del display el mensaje "TD3 TP3" a intervalos regulares, mientras que la anterior tarea muestre el mensaje "Cant de pulsos xxxx" en la segunda línea periódicamente.

AYUDA: La función lcd_set_cursor(línea, posición) permite ubicar el cursor en una posición indicada siendo para línea 0 o 1 los valores posibles y 0 a 15 los posibles para posición.

Parte Teórica:

1) El "Procesamiento diferido por interrupción" se refiere a la manera de trabajar donde la rutina de atención de una interrupción no realiza más funciones que avisar por medio de un semáforo por ejemplo que ocurrió una interrupción.

   Y el procesamiento, en lugar de estar en la interrupción, se realiza en una tarea de FREERTOS que atiende dicho semáforo.

   Esto permite que las rutinas de interrupción sean cortas y rápidas para que no se pierda tiempo si se interrumpió una tarea de mayor prioridad. Además, evita problemas de sincronización y mejora la modularidad y la flexibilidad del código.

   Ejemplo: Supongamos que dada una interrupción proveniente de un pulsador de marcha se accione un mecanismo para abrir un portón. La rutina de interrupción sólo tendrá que dar un semáforo y la tarea en cuestión leerá el semáforo y al desbloquearse accionará el portón.

   Rutina de interrupción:

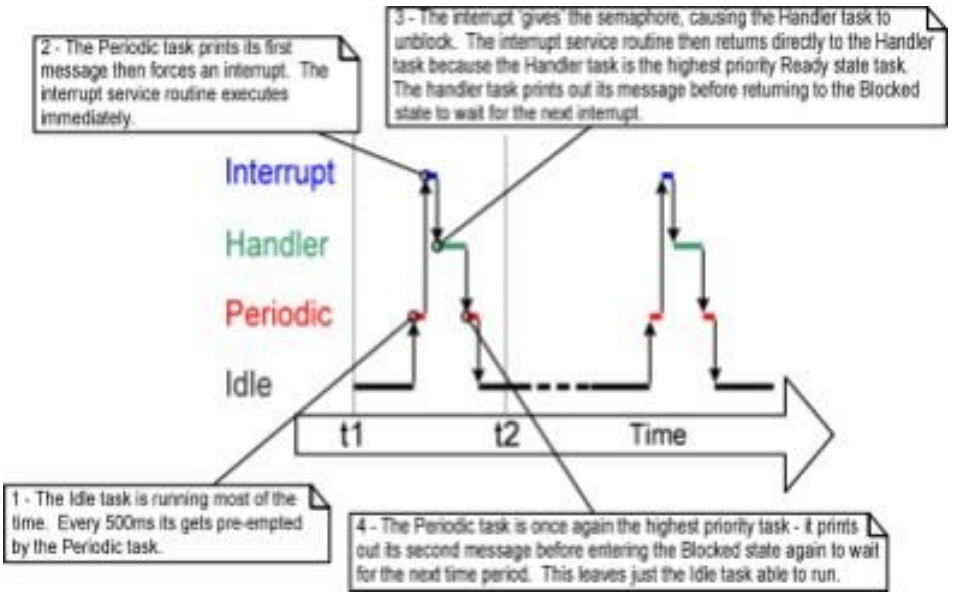
```
/* USER CODE BEGIN 4 */

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    static portBASE_TYPE xHigherPriorityTaskWoken;
    xHigherPriorityTaskWoken = pdFALSE;

    if(GPIO_Pin == GPIO_PIN_9) // INT Source is pin A9
    {
        xSemaphoreGiveFromISR( s1, &xHigherPriorityTaskWoken );
    }
}
```

Tarea de procesamiento:

```
static void tareaPorton(void *pvParameters){

    while(1){
        xSemaphoreTake(s1, portMAX_DELAY);

        accionarPorton();

        vTaskDelay(200/portTICK_PERIOD_MS);

    }
}
```

2) El parámetro "**pxHigherPriorityTaskWoken**" se utiliza cuando una tarea o semáforo pueden bloquear más de una tarea. Lo que se hace es colocar este parámetro en pdTRUE para que al realizarse un cambio de contexto el scheduler ejecute la tarea de mayor prioridad que está esperando el recurso.

3) Utilice los apuntes de la cátedra para desarrollar el contexto del siguiente gráfico:



En el gráfico, se observa una secuencia donde una interrupción es tratada con un semáforo binario:

- Primero, ocurre una interrupción
- La rutina de servicio de la interrupción es ejecutada. Dando el semáforo para desbloquear la tarea correspondiente.

- Dicha tarea es ejecutada cuando finaliza la rutina de la interrupción. Y toma el semáforo.
- La tarea procesa el evento y después intenta tomar el semáforo nuevamente. Si el semáforo no estaba nuevamente disponible entra en estado bloqueada.

Lo que sucede con esta secuencia, es que la atención de la/s interrupciones va a funcionar bien siempre y cuando la frecuencia de interrupción no supere el tiempo de procesamiento de la tarea. Porque por ejemplo si el semáforo ya fue tomado por la tarea y mientras la tarea se ejecuta, se interrumpe 2 veces más. Una de estas interrupciones dará el semáforo y la otra se perderá. Por lo que, estaríamos perdiendo atender la última de estas interrupciones en la tarea.

Para evitar estos casos, lo que se puede utilizar es un semáforo counting. Donde no importa si el semáforo ya había sido dado, sino que cuando se vuelve a interrumpir se aumenta la cuenta del semáforo para que después la tarea tenga que atender esa cuenta de interrupciones.

4) Explique los diferentes casos de Exclusión Mutua existentes en FREERTOS. En cada caso proponga un ejemplo aplicado a la plataforma actual utilizada en la cátedra.
Las herramientas de Exclusión Mutua que nos permite utilizar FREERTOS son:
   - Semáforos binarios: Que funciona como una cola unitaria donde las tareas pueden tomar o dar el semáforo. En caso de que quiera tomarlo, previamente alguna otra tarea o interrupción debió haber dado dicho semáforo, sino la tarea se bloquea.
   - Semáforos Mutex: Funcionan como los semáforos binarios, pero, además resuelven problemas que pueden darse por la prioridad y el acceso a los recursos. De tal forma, que si una tarea de menor prioridad toma el semáforo y está utilizando dicho recurso. Cuando una tarea de mayor prioridad quiere tomar ese semáforo y se bloque porque ya estaba tomado por la primera tarea. Dicha primera tarea va a aumentar su prioridad para que cuando termine se ejecute la tarea de mayor prioridad y no una tarea de prioridad entre ambas.

   Ejemplos:
   Semáforo binario utilizado en la parte práctica de este trabajo, para que 2 tareas utilicen el lcd

```
static void Lcd1(void *pvParameters){

    static int count = 0;
    static char cadena[16];
    TickType_t xLastWakeTime;

    while(1){
        xSemaphoreTake(sLcd, portMAX_DELAY);
        count = uxSemaphoreGetCount(sCount);
        xQueueReset(sCount);

        sprintf(cadena,"Can de Puls %d    ", count);
        lcd_set_cursor(1,0);
        lcd_string(cadena);

        xSemaphoreGive(sLcd);
        //vTaskDelay(1000/portTICK_PERIOD_MS);
        vTaskDelayUntil( &xLastWakeTime, 1000/portTICK_PERIOD_MS );
        xLastWakeTime = xTaskGetTickCount();
    }
}


static void Lcd2(void *pvParameters){

    while(1){
        xSemaphoreTake(sLcd, portMAX_DELAY);
        lcd_set_cursor(0,0);
        lcd_string("TD3 TP3");


        xSemaphoreGive(sLcd);
        vTaskDelay(200/portTICK_PERIOD_MS);

    }
}
```

Semáforo mutex, en el siguiente ejemplo tenemos 3 tareas. Todas de distintas prioridades, donde la tarea 1 y 3 utilizan el mismo recurso. Y también, hay una tarea 2 de procesamiento continuo. Si no utilizará un mutex, y la tarea 2 se desbloquea cuando yo estaba ejecutando la de menor prioridad. Lo que ocurrirá es que la de mayor prioridad quedará bloqueada por una tarea de menor prioridad (Inversión de prioridades).

```
static void tarea_mayor_prioridad(void *pvParameters){

    while(1){
        xSemaphoreTake(sMutex, portMAX_DELAY);
        lcd_set_cursor(0,0);
        lcd_string("Escribe la tarea de mayor prioridad");
        xSemaphoreGive(sLcd);
        vTaskDelay(2000/portTICK_PERIOD_MS);

    }
}

static void tarea_prioridad_procesamiento_continuo(void *pvParameters){

    while(1){
        xSemaphoreTake(s1, portMAX_DELAY);
        procesamientoContinuo();
    }
}

static void tarea_menor_prioridad(void *pvParameters){

    while(1){
        xSemaphoreTake(sMutex, portMAX_DELAY);
        lcd_set_cursor(0,0);
        lcd_string("Escribe la tarea de menor prioridad");
        xSemaphoreGive(sLcd);
        vTaskDelay(1000/portTICK_PERIOD_MS);

    }
}
```

5) Las secciones críticas básicas son regiones de código que están rodeadas por llamados a los macros taskENTER_CRITICAL() y taskEXIT_CRITICAL(). Las cuales van a garantizar que ninguna interrupción se ejecute durante la sección crítica. Las únicas interrupciones que se pueden producir son aquellas cuya prioridad sea mayor al valor asignado a configMAX_SYSCALL_INTERRUPT_PRIORITY.

```
static void tarea_con_acceso_dedicado(void *pvParameters){

    while(1){
        taskENTER_CRITICAL();
        accesoAMemoria();
        taskEXIT_CRITICAL();
        vTaskDelay(3000/portTICK_PERIOD_MS);

    }
}
```

Otra forma de crear secciones críticas, es suspendiendo al Scheduler para que ninguna otra tarea me interrumpa en el proceso. En este caso, las interrupciones permanecen habilitadas. Se debe considerar en que caso se utiliza esta estrategia, debido a que reanudar el Scheduler es una operación relativamente larga. Las funciones para suspender o reanudar el Scheduler son: vTaskSuspendAll() y xTaskResumeAll().

```
static void tarea_con_acceso_dedicado(void *pvParameters){

    while(1){
        vTaskSuspendAll();
        accesoAMemoria();
        xTaskResumeAll();
        vTaskDelay(3000/portTICK_PERIOD_MS);

    }
}
```

6) La Exclusión Mutua sucede para que las tareas no accedan a un recurso al mismo tiempo. La idea de la exclusión mutua es para que una vez que una tarea tome un recurso tenga acceso exclusivo a ese recurso.
   El punto muerto es un escenario donde, por ejemplo, 2 tareas se bloquean a la espera del mismo recurso. Esto sucede cuando, por ejemplo:
   1- Las Tarea A y B ambas necesitan adquirir el mutex X y el mutex Y, con el fin de realizar una acción.
   2- La tarea A se ejecuta y toma el mutex X.
   3- La tarea B adelanta a la tarea A.
   4- La tarea B toma el mutex Y, y luego intenta tomar el mutex X, pero no lo logra dado que lo tiene la tarea A. La tarea B entonces se bloquea, esperando a que el mutex X esté disponible.
   5- La tarea A vuelve a ejecutarse e intenta tomar el mutex Y, pero no lo logra dado que la tarea B tiene tomado este mutex. Entonces la tarea A también se bloquea, esperando a que el mutex Y esté disponible. Al final, la Tarea A está esperando al mutex tomado por la Tarea B, y Tarea B está esperando a un mutex retenido por la Tarea A. Se produce entonces un interbloqueo, porque ni tarea puede proceder más allá.

7) Las tareas Guardianes se encargan de controlar un periférico y que ninguna otra tarea lo controle, evita el uso de semáforos mutex y las problemáticas de cambio de prioridad que estos traen.
   Ejemplo: Supongamos que mas de una tarea quiere escribir en un lcd. En lugar de que todas accedan al periférico, lo que se puede hacer es que sólo la tarea guardiana escriba el periférico y las demás escriban en una cola.

```
static void tareaGuardianaLcd( void *pvParameters) {
    xRead valueLCD;
    while (1){
        xQueueReceive(queueLCD, &valueLCD, portMAX_DELAY);

        lcd_set_cursor(valueLCD.posicionX, valueLCD.posicionY);
        lcd_string(valueLCD.text);
    }
}


static void escribir_LCD1( void *pvParameters) {
    static xRead valueReading;
    while (1){
        valueReading.posicionX = 0;
        valueReading.posicionY = 0;
        valueReading.text = "Soy la tarea 1";
        xQueueSendToBack(queueLCD, &valueReading , portMAX_DELAY);
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}

static void escribir_LCD2( void *pvParameters) {
    static xRead valueReading;
    while (1){
        valueReading.posicionX = 0;
        valueReading.posicionY = 0;
        valueReading.text = "Soy la tarea 2";
        xQueueSendToBack(queueLCD, &valueReading , portMAX_DELAY);
        vTaskDelay(1500 / portTICK_PERIOD_MS);
    }
}
```