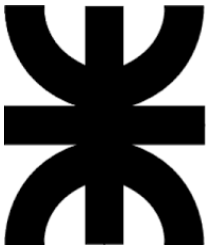


Universidad Tecnológica Nacional

Facultad Regional Avellaneda

Departamento de Ingeniería Electrónica

Cátedra: Técnicas Digitales III

Título del Trabajo Práctico:	Trabajo Práctico INTEGRADOR: Contenidos analizados: • Cap1: Manejo de Tareas • Cap2: Manejo de Colas • Cap3: Gestión de Interrupciones / Semáforos • Cap4: Administración de Recursos / Exclusión • Cap5: Uso de Memoria (Contenidos Integrados)
Autores:	PARDO ERRECARRET, Matías
	RIVAS, Martín
	RODRÍGUEZ, Hernán

IMPORTANTE:

Condición de Aprobación (Nota: 6): Sujeta a funcionamiento general del equipo con base en su funcionamiento mínimo y exposición oral.

Condición de Promoción (Nota: 7-10): Tener el 100% de la consigna desarrollada con informe y presentación oral. Todos los integrantes del grupo DEBEN estar presentes el día de la entrega del Parcial, de no ser así el parcial se debe recuperar en la instancia de recuperación.

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

Parte Teórica:

Consigna: se deberá adjuntar todas las partes Teóricas; con sus consignas, de cada Trabajo Práctico anterior.

Parte Práctica:

A). Realizar el siguiente trabajo práctico en la sección correspondiente al repositorio en GITHub "6.trabajos_practicos" realizando un commit denominado "tp_INT_03-07-23".

B). Se solicita realizar un sistema que permita medir el nivel respecto a la horizontal utilizando la lectura de un módulo acelerómetro. La salida de nivel buscado en grados será indicada por un buzzer. El sistema debe ser programable por PC o teclado, en diferentes posiciones seleccionadas por el grupo de trabajo. Se debe prever la necesidad de almacenar dicha información en la E2PROM o en un módulo SD.

- El sistema debe realizarse con el uC LPC 1769 o BluePill utilizando como Sistema Operativo, FreeRTOS. Como mínimo deben utilizarse dos tareas sincronizadas.

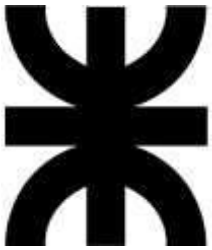
- Se debe llevar a cabo el montaje del sistema en placa experimental donde puedan ensayarse las características del mismo bajo los requerimientos solicitados.

- Se debe entregar el workspace de trabajo el día de entrega del parcial, con el cual se ensayará el funcionamiento del mismo.

- Se debe realizar un informe en formato IEEE con, como mínimo, las siguientes características:

  a) Índice
  b) Abstract
  c) Fundamentación teórica de Hardware y Software
  d) Alcance logrado
  e) Diagrama en bloques (Completo y con detalles de interconexión de bloques)
  f) Sobre el Hardware: se debe incluir las especificaciones técnicas, valores esperados, rangos de medición y precisión obtenida de cada módulo, sensor o dispositivo utilizado.
  g) Sobre el Software: Se debe incluir una guía de código donde se especifique la relación entre las funciones, tareas y recursos del sistema operativo.

NOMBRE Y APELLIDO	CALIFICACIÓN EXPOSICIÓN ORAL
1. Matías Pardo Errecarret	
2. Martín Rivas	
3. Hernán Rodríguez	



Universidad Tecnológica Nacional

Facultad Regional Avellaneda

Departamento de Ingeniería Electrónica

Cátedra: Técnicas Digitales III

Título del Trabajo Práctico:	Trabajo Práctico Nº1: FreeRTOS - Tareas
Autores:	

Resumen:
Contenidos analizados: • Manejo y API correspondiente a Tareas • Conmutación de Tareas • Scheduling/Política de Scheduling

Sistemas Operativos en Tiempo Real: FREERTOS

OBSERVACION: La entrega del presente trabajo se realizará mediante GitHub, en una carpeta llamada **tp1_tareas** dentro de la sección **2.trabajos_practicos**. El correspondiente branch será nombrado **tp1_tareas** y el commit **tp1_tareas_entrega**. La parte teórica debe ser entregada en formato PDF.

Parte Teórica

1) Sistemas Operativos - Generalidades

1-a) ¿Quién es el encargado de administrar el tiempo de la CPU y cuál es su función principal dentro del sistema?

1-b) Explique a que se denomina "Contexto de Ejecución". ¿Cuáles son las variables intervinientes?

1-c) ¿A qué se denomina Sistema Operativo de Tiempo Real? ¿Por qué usar un Sistema Operativo de Tiempo Real?

2) Kernel de FREERTOS

2-a) Explique y desarrolle las características de una "Tarea" en FreeRTOS. Proponga un ejemplo de una tarea manteniendo la sintaxis correcta.

2-b) Explique y desarrolle los "Estados de una Tarea" y sus transiciones. Indique las funciones que permiten las transiciones.

2-c) ¿A qué nos referimos con el concepto de "tareas de procesamiento continuo"? ¿y a qué nos referimos con el concepto de "tareas manejadas por eventos"?

2-d) ¿Cuáles son los tipos de eventos por los que una tarea puede estar esperando al entrar en el estado bloqueado? Explicar.

2-e) ¿Cuál es la función de la IDLE TASK? ¿Puede el procesador no estar ejecutando ninguna tarea en algún momento? Explicar.

Parte Práctica

1) Se desea generar el parpadeo de un LED mediante dos tareas, las cuales cumplirán las siguientes funciones:

Tarea 1: Controlará únicamente el encendido del LED. El mismo debe permanecer encendido por 600ms.

Tarea 2: Controlará únicamente el apagado del LED. El mismo debe permanecer apagado por 400ms.

* No se permite el uso de variables globales a modo de "flags". El ejercicio deberá resolverse íntegramente

2) CONDICION DE PROMOCION: Habiendo cumplido con lo solicitado anteriormente, se solicita: añadir un pulsador, el cual al ser presionado permitirá mantener el estado que el LED tenía al momento de la activación. El LED no cambiará de estado mientras mantenga presionado el pulsador.

Parte Teórica

1-a) ¿Quién es el encargado de administrar el tiempo de la CPU y cuál es su función principal dentro del sistema?

El scheduler es el núcleo del sistema operativo.
Su función principal es administrar la ejecución de las tareas, dándole a cada una un tiempo de la CPU dependiendo de la política de Scheduling.
Dicha política nos indica como llevar a cabo las tareas de una forma organizada y planificada en base a los recursos, prioridades, recursos bloqueantes y al estado de las tareas. El scheduler ya está programado y forma parte del Kernel.
Hay distintos tipos de scheduler con distintas políticas para decidir la ejecución de las tareas.

1-b) Explique a que se denomina “Contexto de Ejecución”. ¿Cuáles son las variables intervinientes?

El contexto de ejecución es aquello que necesita una tarea guardar cuando deja de ser ejecutada para la próxima vez poder retomar.
Lo que debe realizarse al cambiar una tare:

- Salvar el estado (registros, información de punteros de memoria) que se están ejecutando.
- Cambiar el estado de la tarea que estaba ejecutando al que corresponda.
- Cargar el estado asignado a la CPU.
- Cambiar el estado, de la siguiente tarea a ejecutarse, a “ejecutando”

1-c) ¿A qué se denomina Sistema Operativo de Tiempo Real? ¿Por qué usar un SistemaOperativo de Tiempo Real?

Un Sistema Operativo de Tiempo Real es un sistema operativo ligero capaz de poder ser integrado en microprocesadores, por lo cual son de tamaño ligero para poder entrar en dicho micro. Se llaman de tiempo real porque tienen una respuesta determinística en el tiempo más que velocidad,

lo que prioriza es que el tiempo de ejecución sea siempre preciso. Por lo cual, satisface los requerimientos de tiempo real duro debido a su determinismo.
Además, debe tener recursos que van a ser líneas de códigos especificas que en su conjunto permitan administrar las funciones del sistema operativo. Para lo cual, contará con APIS, una asociada a las tareas, otra asociada a las colas, semáforos y a las distintas funciones del sistema.
Este tipo de sistemas operativos se utilizan en aplicaciones donde es importante que el sistema responda a un tiempo específico. Por ejemplo, el mecanismo de disparo de un Airbag debe desplegarse en un cierto tiempo después del impacto, si la respuesta se demora, el conductor podría sufrir heridas serias.

2-a) Explique y desarrolle las características de una "Tarea" en FreeRTOS. Proponga unejemplo de una tarea manteniendo la sintaxis correcta.

Una tarea en FreeRTOS es la unidad básica de ejecución de código que puede ser programada y administrada por el sistema operativo. Para nosotros una tarea va a ser una función en C con algunas características particulares. ¡Observación: Una tarea es una función en C pero una función en C no es una tarea!
Las características particulares son:

- Prioridad: Cada tarea tiene su propia prioridad, lo que le permite al Scheduler asignar recursos de CPU de manera eficiente en función de la importancia de cada tarea.
- Contexto de ejecución: Cada tarea tendrá su propio contexto de ejecución para poder ejecutarse sin afectar a las otras tareas.
- Estados: Una tarea puede estar en estado Ejecución y No Ejecución. Dentro de este último puede estar en Bloqueada, Suspendida o Ready (lista para ejecutarse).
- Tiempo de ejecución asignado: El Scheduler asigna un tiempo de ejecución a cada tarea, lo que evita que una tarea monopolice la CPU y garantiza que todas las tareas reciban una cantidad justa.

Ejemplo de una tarea que hace titilar un led:

```
xTaskCreate(tarea_blinky,
"blinky",
configMINIMAL_STACK_SIZE,
NULL,
1,
NULL);

void tarea_blinky(void *p)
{
while(1)
{
        HAL_GPIO_TogglePin(GPIOC, GPIO_PIN13);
        vTaskDelay(1000);
}
}
```

2-b) Explique y desarrolle los "Estados de una Tarea" y sus transiciones. Indique lasfunciones que permiten las transiciones.

Estados de una Tarea:

- Ejecución: La tarea se está ejecutando en la CPU.
- No Ejecución: El cual se puede expandir en los siguientes estados:
- Bloqueado: La tarea está esperando un evento para estar disponible, los eventos que espera pueden ser:
    - Eventos temporales: Se debe cumplir un retardo de un cierto tiempo y una vez pase dicho tiempo la tarea deja de estar bloqueada. Por ejemplo, con el uso de al función vTaskDelay()
    - Eventos de Sincronización: Donde el evento es originado desde otra tarea o

interrupción. Por ejemplo, una tarea puede bloquearse con un semáforo o cola.

- Suspendido: Las tareas en este estado no están disponibles para que el Scheduler las pase a Ejecución. La única manera de entrar en este estado es con la función vTaskSuspend(), y la forma de salir de este estado es con las funciones vTaskResume() p vTaskResumeFromISR().
- Disponible: Las tareas que no se están ejecutando, pero no se encuentran en ninguno de los otros 2 estados. Significa que el scheduler les puede asignar tiempo de CPU. Con vTaskCreate() se crea una tarea y se le asigna este estado.

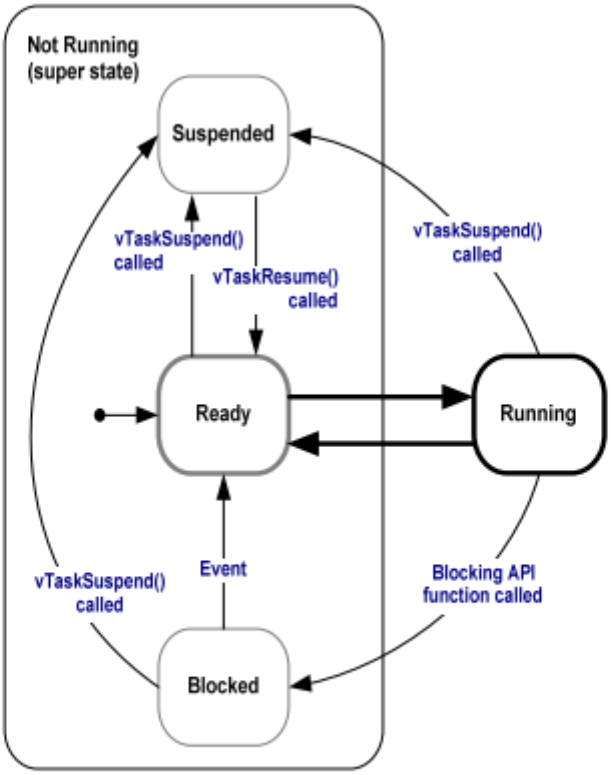


Figura 7 – Diagrama completo de estados y transiciones.

Nota: Suspend = Suspendido, Ready = Listo o Disponible, Blocked = Bloqueado, Running = En Ejecución.

2-c) ¿A qué nos referimos con el concepto de “tareas de procesamiento continuo”? ¿y a quénos referimos con el concepto de “tareas manejadas por eventos”?

Las “tareas de procesamiento continuo” son aquellas que están siempre disponibles para su ejecución, ya que no deben esperar ningún tiempo o evento para ser ejecutadas. Tienen una utilidad limitada y siempre deberán tener el menor nivel de prioridad disponible, porque sino impedirán la ejecución de tareas de menor prioridad.

Las “tareas manejadas por eventos” sólo entran en ejecución luego que haya ocurrido cierto evento que la dispara, y no pueden ejecutarse antes de que dicho evento ocurra. Estas tareas pueden tener distintos niveles de prioridad, porque si una tarea de mayor prioridad se encuentra bloqueada debido a que está esperando un evento, una tarea de menor prioridad que se encuentra disponible puede ser ejecutada por el Scheduler.

2-d) ¿Cuáles son los tipos de eventos por los que una tarea puede estar esperando al entrar enel estado bloqueado? Explicar.

Los eventos que puede esperar una tarea al entrar en estado Bloquedo son

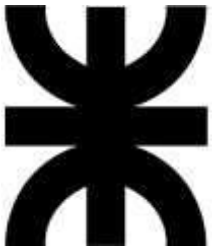
- Eventos temporales: Se debe cumplir un retardo de un cierto tiempo y una vez pase dicho tiempo la tarea deja de estar bloqueada. Por ejemplo, con el uso de al función vTaskDelay()

- Eventos de Sincronización: Donde el evento es originado desde otra tarea o interrupción. Por ejemplo, una tarea puede bloquearse con un semáforo o cola.

2-e) ¿Cuál es la función de la IDLE TASK? ¿Puede el procesador no estar ejecutandoninguna tarea en algún momento? Explicar.

El procesador no puede estar ejecutando ninguna tarea, por lo que existe una tarea ociosa (IDLE TASK) que se crea automáticamente cuando se llama a la función vTaskStartScheduler(). Dicha tarea no hace más que entrar en un bucle y siempre es capaz de ejecutarse.
Esta tarea cuenta siempre con la misma prioridad (cero 0) para asegurarse permitir que las tareas de mayor prioridad puedan ejecutarse.
La tarea IDLE cuenta con ciertas si así se desea:

- Ejecución de baja prioridad, de fondo o de procesamiento continuo, estando siempre disponible para ejecución.
- Medir la capacidad de procesamiento extra. Debido a que esta tarea, entra en ejecución cuando ninguna tarea está consumiendo tiempo del procesador, es muy practica para medir el tiempo que no se está utilizando.
- Colocar el procesador en un modo de bajo consumo. Poner en ahorro de energía al procesador siempre que no haya procesamiento a realizar.



Universidad Tecnológica Nacional

Facultad Regional Avellaneda

Departamento de Ingeniería Electrónica

Cátedra: Técnicas Digitales III

Título del Trabajo Práctico:	Trabajo Práctico Nº2: FreeRTOS - Colas
Autores:	RIVAS, Martín

Resumen:
Contenidos analizados: • Gestión y API correspondiente a Colas • Gestión de ADC • Gestión de Interrupciones

Sistemas Operativos en Tiempo Real: FREERTOS

OBSERVACION: La entrega del presente trabajo se realizará mediante GitHub, en una carpeta llamada tp2_colas dentro de la sección 2.trabajos_practicos. El correspondiente branch será nombrado tp2_colas y el commit tp2_colas_entrega. La parte teórica debe ser entregada en formato PDF.

Parte Teórica

1) Desarrolle como es el almacenamiento de datos en el recurso "Cola" proporcionado por el Kernel.

2) Explique utilizando un ejemplo con código el porqué es común que el recurso Cola posea múltiples tareas escritoras y una sola tarea lectora.

3) Explique el proceso de bloqueo por escritura y bloqueo por lectura en el recurso Cola.

4) A la hora de enviar datos compuestos: explique cómo se lleva a cabo dicho proceso y por qué no genera conflictos con la definición de la macro del recurso. ¿Cuál sería la implementación si los datos fueran "grandes", lo que ocasionaría un problema con la memoria RAM?

1) Una cola puede contener un número finito de elementos de datos, todos del mismo tamaño fijo. El número máximo de elementos que una cola puede contener se llama su "longitud".
Tanto la longitud y el tamaño de cada elemento de datos se establecen cuando la cola es creada.
Normalmente las colas son usadas como buffers primero en entrar-primero en salir (FIFO) donde los datos se escriben en el final de la cola y se retira de la parte frontal de la cola. También es posible escribir en la parte delantera de una Cola. La escritura de datos a una cola se lleva a cabo copiando byte por byte de los datos a ser almacenados en la cola en sí misma.
La lectura de datos de una cola provoca una copia de los datos que se eliminaran de la cola.

2) Es común que el recurso cola tenga múltiples tareas escritoras y una sola lectora debido al uso de la comunicación asincrónica y la necesidad de compartir información entre tareas de manera eficiente y sincronizada.
Por ejemplo, podría ser el caso de múltiples tareas que cada una cumpla la función de leer un parámetro en la cola, tal vez proveniente de un sensor y haya una única tarea lectora (la controladora) que va a ser la que controle la escritura de una pantalla LCD.

```
void writeTask1 (void *pvParameters){
        xRead value;

        while(1){
                leerSensor1();
                xQueueSendToBack(queue, &value, portMAX_DELAY;
                vTaskDelay(200/portTICK_PERIOD_MS);
        }
}
```

```
void writeTask2 (void *pvParameters){
        xRead value;

        while(1){
                leerSensor2();
                xQueueSendToBack(queue, &value, portMAX_DELAY;
                vTaskDelay(200/portTICK_PERIOD_MS);
        }
}
void readTask (void *pvParameters){
        xRead value;

        while(1){

                xQueueReceive(queue, &value, portMAX_DELAY);
                escribirLCD();
        }
}
```

3) El proceso de bloqueo por escritura se produce cuando una tarea va a escribir sobre una cola y la misma ya está llena. La tarea va a pasar a estado Lista en 2 casos. Cuando se cumpla el tiempo máximo de bloqueo aclarado o cuando una tarea libere un dato de la cola leyéndolo.
El proceso de bloqueo por lectura se produce cuando una tarea va a obtener un dato sobre una cola y la misma esta vacía. La tarea pasará a estado Lista cuando otra tarea escriba un dato sobre la misma o cuando se cumpla el tiempo máximo de bloqueo establecido.
En el primer caso, se puede dar que más de una tarea este necesitando el dato, en ese caso se liberará la tarea de mayor prioridad que estaba esperando más tiempo. Lo mismo puede pasar en el caso de escritura, si más de una tarea estaba esperando para escribir, en el momento que se libere la cola, escribirá sobre ella la de mayor prioridad que estaba esperando más tiempo.

4) Para transferir datos compuestos, lo que se hace es definir una estructura y transferir el valor de los datos y la fuente de los datos estén contenidos en los campos de la estructura.
Cuando los datos son grandes, lo que se puede hacer es mandar por la cola un puntero al dato. El cual va a ser más fácil de copiar byte a byte y más liviano. Lo que hay que tener en cuenta al pasar punteros, es que la tarea receptora pueda acceder a la memoria apuntada por el puntero y sea la misma que cuando se guardo el dato. Una tarea debe ser la responsable de liberar la memoria porque sino la memoria apuntada sigue siendo válida.

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.	

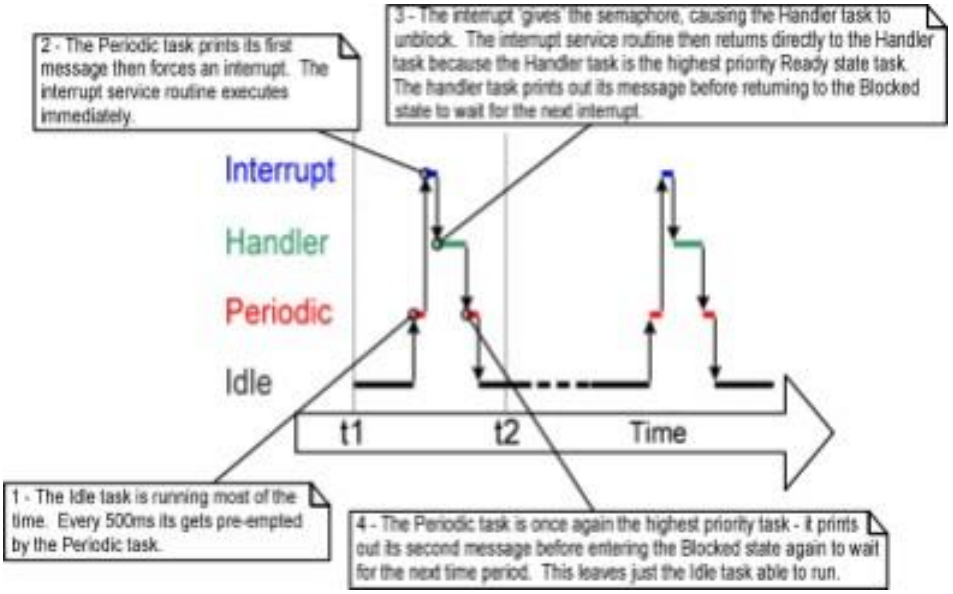
Sistemas Operativos en Tiempo Real: FREERTOS (Trabajo Práctico Nº3)

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:

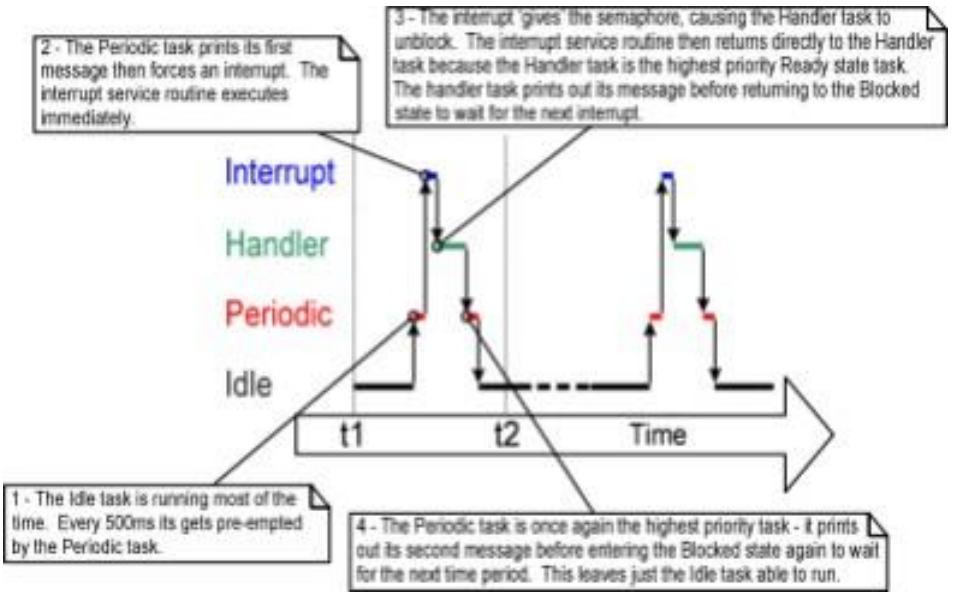
```
/* USER CODE BEGIN 4 */

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);
    }
}
```