

Usando el Kernel de Tiempo Real FreeRTOS:

(Traducción de la Guía práctica por Richard Barry)

Capítulo 3: Gestión de Interrupciones

3.1 Introducción y alcance de este capítulo:

Eventos:

Los sistemas embebidos de tiempo real tienen que tomar acciones en respuesta a los eventos que se originan en el ambiente. Por ejemplo, un paquete que llega en un periférico Ethernet (evento) podría requerir pasar a una pila TCP / IP para el procesamiento (acción). Los sistemas no triviales tendrán que servir eventos que se originan a partir de múltiples fuentes, los cuales tendrán diferentes cargas de procesamiento y requisitos de tiempo de respuesta. En cada caso, un juicio tiene que ser hecho en cuanto a la mejor estrategia de procesamiento de eventos:

1. ¿Cómo se debe detectar el evento? Normalmente se utilizan las interrupciones, pero las entradas también pueden ser consultadas.
 2. Cuando se utilizan interrupciones, ¿cuánto procesamiento se debe realizar dentro del servicio de interrupción de rutina (ISR), y cuánto afuera? Normalmente es deseable mantener cada ISR lo más corto posible.
 3. ¿Cómo pueden comunicarse los eventos al código principal (no-ISR), y cómo puede ser estructurado el código para acomodar mejor el procesamiento de sucesos potencialmente asíncronos?
- FreeRTOS no impone ninguna estrategia específica de procesamiento de eventos, pero proporciona características que permitirán a la estrategia elegida ser implementada de modo simple y mantenible.

Cabe señalar que sólo las funciones de la API y macros que terminan en 'FromISR' o 'FROM_ISR' nunca deberían ser utilizadas dentro de una rutina de servicio de interrupción.

Visión:

Este capítulo tiene como objetivo dar a los lectores una buena comprensión de:

- Qué funciones API de FreeRTOS se pueden utilizar dentro de una rutina de servicio de interrupción.
- Cómo se puede implementar un esquema de interrupción diferidos.
- Cómo crear y utilizar semáforos binarios y semáforos counting.
- Las diferencias entre semáforos binarios y counting.
- Cómo utilizar una cola para pasar los datos hacia y desde una rutina de servicio de interrupción.
- El modelo de anidación de interrupción disponible con algunos puertos FreeRTOS.

3.2 Procesamiento diferido de interrupción:

Semáforos binarios usados para sincronización:

Un semáforo binario se puede utilizar para desbloquear una tarea cada vez que una interrupción en particular se produce, efectivamente sincronizando la tarea con la interrupción. Esto permite que la mayoría de los eventos de procesamiento de interrupción sean implementados dentro de la tarea sincronizada, con sólo una porción muy corta restante directamente en el ISR. El proceso de interrupción se dice que ha sido "diferida" a una tarea "manipuladora".

Si el proceso de interrupción es particularmente crítico temporalmente, la prioridad de la tarea manipuladora puede setearse para asegurarse que ésta siempre se adelante a la otra tarea. Será entonces la tarea a la cual el ISR retorna cuando termina de ejecutar. Esto tiene el efecto de asegurar que el procesamiento de eventos se ejecuta de forma contigua en el tiempo, como si todo hubiera sido implementado dentro del ISR. Este esquema se muestra en la Figura 26.

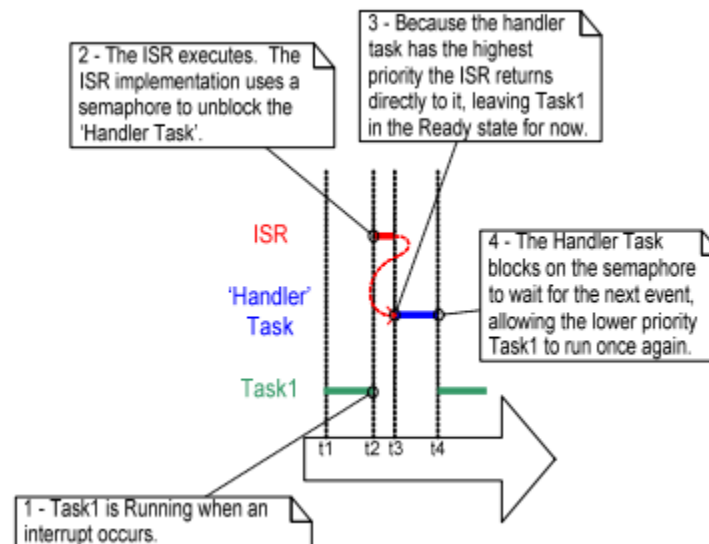


Figura 26 – La interrupción interrumpe una tarea, pero retorna a otra.

La tarea manipuladora usa un "Tomar semáforo" para entrar en el estado Bloqueado, y esperar a que el evento ocurra. Cuando el evento ocurre el ISR usa un "Dar semáforo" en el mismo semáforo, para desbloquear la tarea y que el evento pueda ser procesado.

"Tomar" y "Dar" un semáforo son conceptos que tienen diferentes significados dependiendo el escenario en el cual se usan. En la terminología clásica, tomar un semáforo es equivalente a una operación P(), y dar un semáforo es equivalente a una operación V().

En este escenario de sincronización de interrupciones, los semáforos pueden ser pensados conceptualmente como una cola de un solo elemento. La cola puede tener un largo máximo de un elemento, por lo que siempre se encontrará o llena o vacía (binario). Llamando a xSemaphoreTake() la tarea manipuladora intenta leer el valor de la cola, causando que la tarea se bloquee si la cola estaba vacía. Cuando el evento ocurre el ISR simplemente usa la función xSemaphoreGiveFromISR() para posicionar un elemento (el semáforo) en la cola, dejándola llena. Esto produce que la tarea manipuladora salga del

estado bloqueado y obtenga el elemento de la cola, dejándola nuevamente vacía. Una vez que la tarea manipuladora ha completado su procesamiento, intenta nuevamente leer la cola, que, como se encuentra vacía, vuelve a entrar en el estado bloqueado, esperando al próximo evento. Esta secuencia se muestra en la Figura 27.

Función API vSemaphoreCreateBinary():

Los manipuladores de todos los diferentes tipos de semáforos de FreeRTOS se almacenan en una variable del tipo xSemaphoreHandle.

Previo a que un semáforo pueda ser usado, debe ser creado. Para crear un semáforo binario se usa la función API vSemaphoreCreateBinary().

```
Void vSemaphoreCreateBinary( xSemaphoreHandle xSemaphore ) ;
```

Listado 41 – Prototipo de la función API vSemaphoreCreateBinary().

Tabla 12 Parámetros de la función vSemaphoreCreateBinary().

Nombre del Parámetro	Descripción
xSemaphore	Semáforo que se creará. Notar que esta función es en realidad implementada como una macro, por lo que la variable del semáforo debería pasarse directamente y no por referencia. Los ejemplos en este capítulo incluyen llamados a esta función que pueden ser usados como referencia.

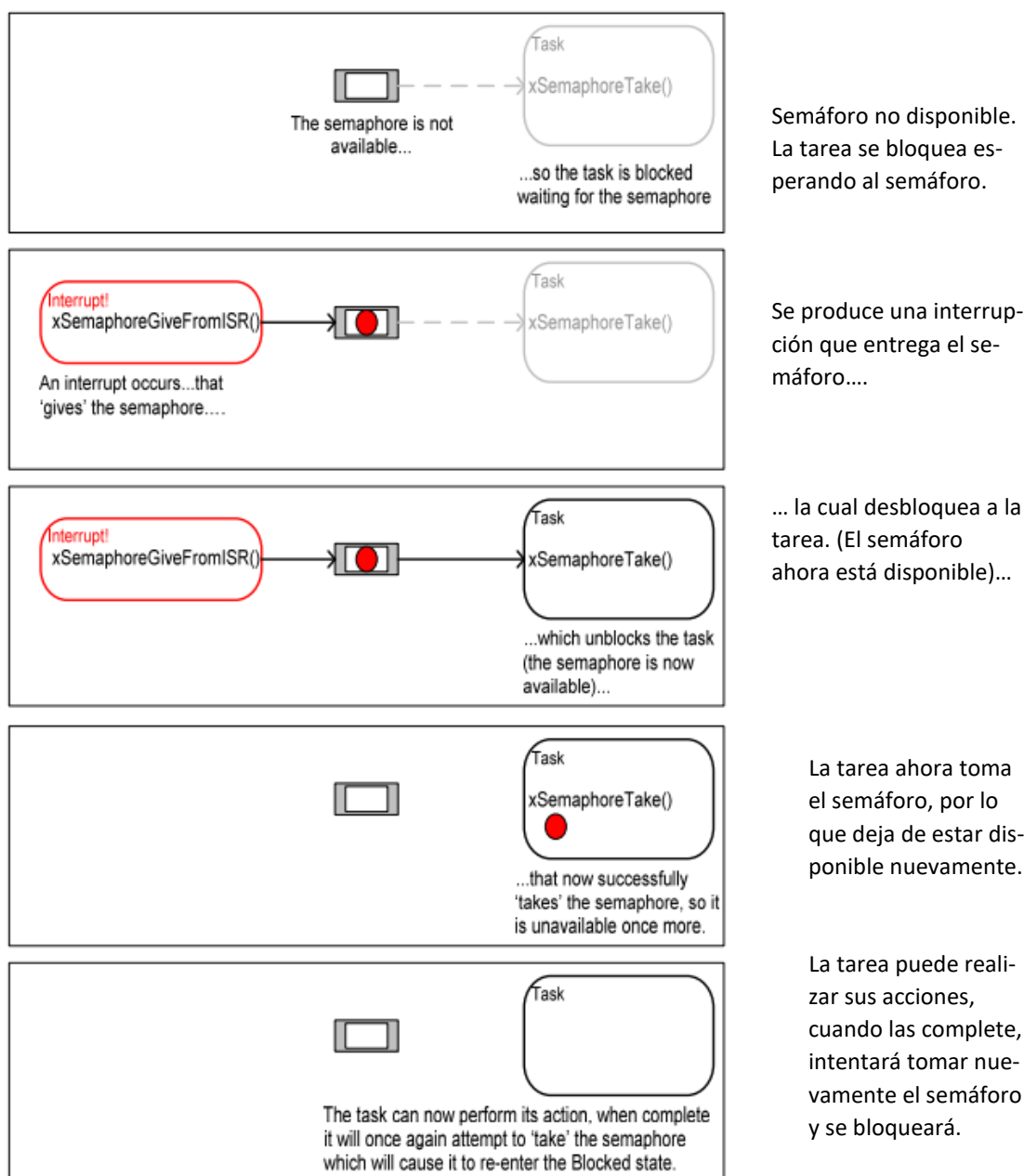


Figura 27 – Usando un semáforo binario para sincronizar una tarea con una interrupción.

Función API `xSemaphoreTake()`:

Tomar un semáforo es sinónimo de obtener o recibir el semáforo. El semáforo solo puede ser tomado si estaba disponible. En la terminología clásica, la función `xSemaphoreTake()` es equivalente a una operación P().

Todos los distintos tipos de semáforos de FreeRTOS excepto los semáforos recursivos pueden ser tomados usando esta función.

Esta función no debe ser usada desde un servicio de interrupción de rutina.

```
portBASE_TYPE xSemaphoreTake( xSemaphoreHandle xSemaphore, portTickType xTicksToWait );
```

Listado 42 – Prototipo de la función API xSemaphoreTake().

Tabla 13 Parámetros y valores de retorno de la función vSemaphoreTake().

Nombre del Parámetro/ Valor de Retorno	Descripción
xSemaphore	Semáforo que se va a tomar. Un semáforo es referido por una variable del tipo xSemaphoreHandle, que debe ser explícitamente creada antes de llamar esta función.
xTicksToWait	Máximo tiempo que la tarea debe permanecer bloqueada esperando por el semáforo si este no está disponible. Si xTicksToWait es 0 retornará inmediatamente si el semáforo no está disponible. El tiempo de bloqueo se especifica en periodos de Tick, y puede usarse la constante portTICK_RATE_MS para convertir este tiempo a milisegundos. Seteando xTicksToWait en portMAX_DELAY causa que la tarea espere indefinidamente a que el semáforo esté disponible, si está seteado en 1 INCLUDE_vTaskSuspend en FreeRTOSConfig.h
Valor de retorno	Hay dos posibles valores de retorno: 1. pdPASS: si se obtuvo el semáforo luego de llamar a esta función. Si se especificó un tiempo de bloqueo, puede que el semáforo se puso disponible en este tiempo, si es que no estaba disponible inicialmente. 2. pdFALSE: El semáforo no estaba disponible. Si se especificó un tiempo de bloqueo, la tarea estuvo en estado bloqueado esperando a que esté disponible, pero expiró el tiempo antes que esto ocurriera.

Función API xSemaphoreGiveFromISR():

Todos los distintos tipos de semáforos de FreeRTOS excepto los semáforos recursivos pueden ser entregados usando esta función.

xSemaphoreGiveFromISR() es una forma especial de la función xSemaphoreGive(), que es usada dentro de servicios de rutinas de interrupción.

```
portBASE_TYPE xSemaphoreGiveFromISR( xSemaphoreHandle xSemaphore,  
                                     portBASE_TYPE *pxHigherPriorityTaskWoken  
                                     );
```

Listado 43 – Prototipo de la función API xSemaphoreGiveFromISR().

Tabla 14 Parámetros y valores de retorno de la función vSemaphoreGiveFromISR().

Nombre del Parámetro/ Valor de retorno	Descripción
xSemaphore	Semáforo siendo entregado. Un semáforo es referenciado por una variable del tipo xSemaphoreHandle y debe ser explícitamente creada antes de ser usada.
pxHigherPriorityTaskWoken	Es posible que un solo semáforo tenga una o más tareas bloqueadas en él, esperando que el semáforo esté disponible. Un llamado a la función xSemaphoreGiveFromISR () puede hacer que el semáforo esté disponible, y así causar que una tarea deje el estado bloqueado. Si el llamado a xSemaphoreGiveFromISR () hace que una tarea deje el estado bloqueado, y esta tarea tiene una prioridad más alta que la actualmente en ejecución , (la tarea que fue interrumpida), a continuación, xSemaphoreGiveFromISR() establecerá internamente * pxHigherPriorityTaskWoken a pdTRUE. Si xSemaphoreGiveFromISR () establece este valor a pdTRUE luego un cambio de contexto se debe realizar antes de salir de la interrupción. Esto asegurará que la interrupción regresa directamente a la tarea disponible con más alta prioridad.
Valor retornado	Hay dos valores de retorno posibles: pdPASS: si el llamado a esta función resultó satisfactorio. pdFAIL: si el semáforo ya estaba disponible, no puede ser retornado nuevamente.

Ejemplo 12. Usando un semáforo binario para sincronizar una tarea con una interrupción:

Este ejemplo usa un semáforo binario para desbloquear una tarea desde adentro de una rutina de servicio de interrupción, sincronizando efectivamente la tarea con la interrupción.

Una tarea periódica es usada para generar una interrupción por software cada 500 milisegundos. Se usa una interrupción por software por conveniencia, porque es más fácil de simular en un entorno DOS. El Listado 44 muestra la implementación de la tarea periódica. Notar que la tarea imprime una cadena antes y después que la interrupción es generada. Esto se hace para permitir que la secuencia de ejecución sea explicada en la salida producida cuando el ejemplo es ejecutado.

```
static void vPeriodicTask( void *pvParameters )
{
    for( ;; )
    {
        /* Esta tarea es usada para simular una interrupción de software cada 500ms */
        vTaskDelay( 500 / portTICK_RATE_MS );
        /* Se imprime un mensaje antes y después de generar la interrupción. */
        vPrintString( "Periodic task - About to generate an interrupt.\r\n" );
        __asm{ int 0x82 } /* Esta línea genera la interrupción. */
        vPrintString( "Periodic task - Interrupt generated.\r\n\r\n\r\n" );
    }
}
```

Listado 44 – Implementación de la tarea que periódicamente genera una interrupción de software para el Ejemplo 12.

El Listado 45 muestra la implementación de la tarea manipuladora (la tarea que es sincronizada con la interrupción mediante el uso de un semáforo binario). Nuevamente un mensaje es impreso en cada iteración de la tarea, para que la secuencia en la cual la tarea y la interrupción se ejecutan sea evidente observando la salida producida cuando el ejemplo es ejecutado.

```
static void vHandlerTask( void *pvParameters )
{
    /* Como en la mayoría de las tareas, se implementa dentro de un lazo infinito. */
    for( ;; )
    {
        /* Se usa el semáforo para esperar por el evento. El semáforo fue creado antes
        que se inicie el scheduler, por lo tanto, antes que esta tarea se ejecute por primera vez.
        La tarea se bloquea indefinidamente, por lo que la función saldrá del estado bloqueado
        recién cuando el semáforo este disponible. Por lo tanto, no hay necesidad de chequear por el
        valor de retorno de la función. */
        xSemaphoreTake( xBinarySemaphore, portMAX_DELAY );

        /* Para llegar aquí, el evento debe haber ocurrido. Aquí se procesa el evento. En este caso,
        simplemente se imprime una cadena. */
        vPrintString( "Handler task - Processing event.\r\n" );
    }
}
```

Listado 45 – Implementación de la tarea manipuladora (tarea sincronizada con la interrupción).

El Listado 46 muestra el controlador de interrupción real. Esto hace muy poco más que "dar" el semáforo para desbloquear la tarea manipuladora. Note cómo se usa el parámetro

pxHigherPriorityTaskWoken. Se fija para pdFALSE antes de llamar xSemaphoreGiveFromISR (), con un cambio de contexto que se realiza si se encuentra en pdTRUE.

La sintaxis de la declaración de la rutina de servicio de interrupción y la macro llamada para forzar un cambio de contexto son específicas al puerto DOS Open Watcom y serán diferentes para los otros puertos. Por favor refiérase a los ejemplos que se incluyen en la aplicación de demostración para encontrar la sintaxis adecuada al puerto que se va a utilizar.

```
static void __interrupt __far vExampleInterruptHandler( void )
{
    static portBASE_TYPE xHigherPriorityTaskWoken;
    xHigherPriorityTaskWoken = pdFALSE;

    /* Entrega el semáforo para desbloquear la tarea. */
    xSemaphoreGiveFromISR( xBinarySemaphore, &xHigherPriorityTaskWoken );

    if( xHigherPriorityTaskWoken == pdTRUE )
    {
        /* Entregando el semáforo se desbloquea una tarea, cuya prioridad es mayor a la
        tarea que se estaba ejecutando. Se fuerza un cambio de contexto para asegurar
        que la interrupción retorna directamente a la tarea desbloqueada (de mayor prioridad).
        Nota: La macro que se debe usar para forzar el cambio de contexto depende del
        puerto. Esta es la macro correcta para usar un puerto DOS Open Watcom. Otros
        puertos pueden requerir una sintaxis diferente. */
        portSWITCH_CONTEXT();
    }
}
```

Listado 46 – Controlador de Interrupción usado en el Ejemplo 12.

La función main() simplemente crea el semáforo binario y las tareas, instala el controlador de interrupción y arranca el scheduler. La implementación se muestra en el Listado 47.


```
int main( void )
{
    /* Antes de usar un semáforo este debe ser explícitamente creado. */
    vSemaphoreCreateBinary( xBinarySemaphore );

    /* Se instala el controlador de interrupción. */
    _dos_setvect( 0x82, vExampleInterruptHandler );

    /* Verifica que el semáforo se creó correctamente. */
    if( xBinarySemaphore != NULL )
    {
        /* Crea la tarea manipuladora. Esta es la tarea que será sincronizada con la interrupción. Es creada con una prioridad mayor para asegurar que correrá inmediatamente después que la interrupción ocurra. En este caso se eligió una prioridad de 3. */
        xTaskCreate( vHandlerTask, "Handler", 1000, NULL, 3, NULL );

        /* Se crea la tarea que generará periódicamente la interrupción de software. Se crea con una prioridad inferior a la tarea manipuladora. */
        xTaskCreate( vPeriodicTask, "Periodic", 1000, NULL, 1, NULL );

        /* Se inicia el scheduler. */
        vTaskStartScheduler();
    }

    /* Si todo anda bien, el main() nunca debería alcanzar este punto, dado que el Scheduler va a estar siempre ejecutando las tareas. Si el main() alcanza este punto, entonces es probable que la memoria disponible no era suficiente para crear la tarea ociosa. El capítulo 5 provee más información sobre el manejo de memoria. */

    for( ;; );
}
```

Listado 47 – Implementación del main() para el Ejemplo 12.

El Ejemplo 12 produce el resultado que se muestra en la figura 28. Como se esperaba, la tarea manipuladora se ejecuta inmediatamente después que se genera la interrupción, por lo que la salida del controlador de tarea divide la salida producida por la tarea periódica. Otras explicaciones se proveen en la Figura 29.

```

C:\WINDOWS\system32\cmd.exe - rtdemo
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.
    
```

Figura 28 – Salida producida cuando el Ejemplo 12 es ejecutado.

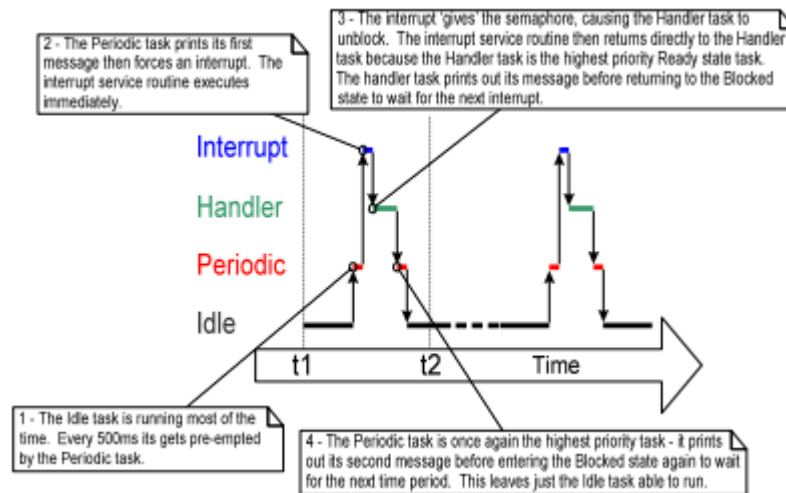


Figura 29 – Secuencia de ejecución cuando es ejecutado el Ejemplo 12.

3.3 Semáforos Counting:

El Ejemplo 12 mostraba un semáforo binario usado para sincronizar una tarea con una interrupción. La secuencia de ejecución era la siguiente:

1. Ocurre una Interrupción.
2. La rutina de servicio de la interrupción es ejecutada, dando el semáforo para desbloquear la tarea manipuladora.
3. La tarea manipuladora es ejecutada tan pronto como la interrupción es completada. Lo primero que hace la tarea manipuladora es tomar el semáforo.
4. La tarea manipuladora procesa el evento antes de intentar volver a tomar el semáforo nuevamente, entrando en estado bloqueado si el semáforo no estaba nuevamente disponible.

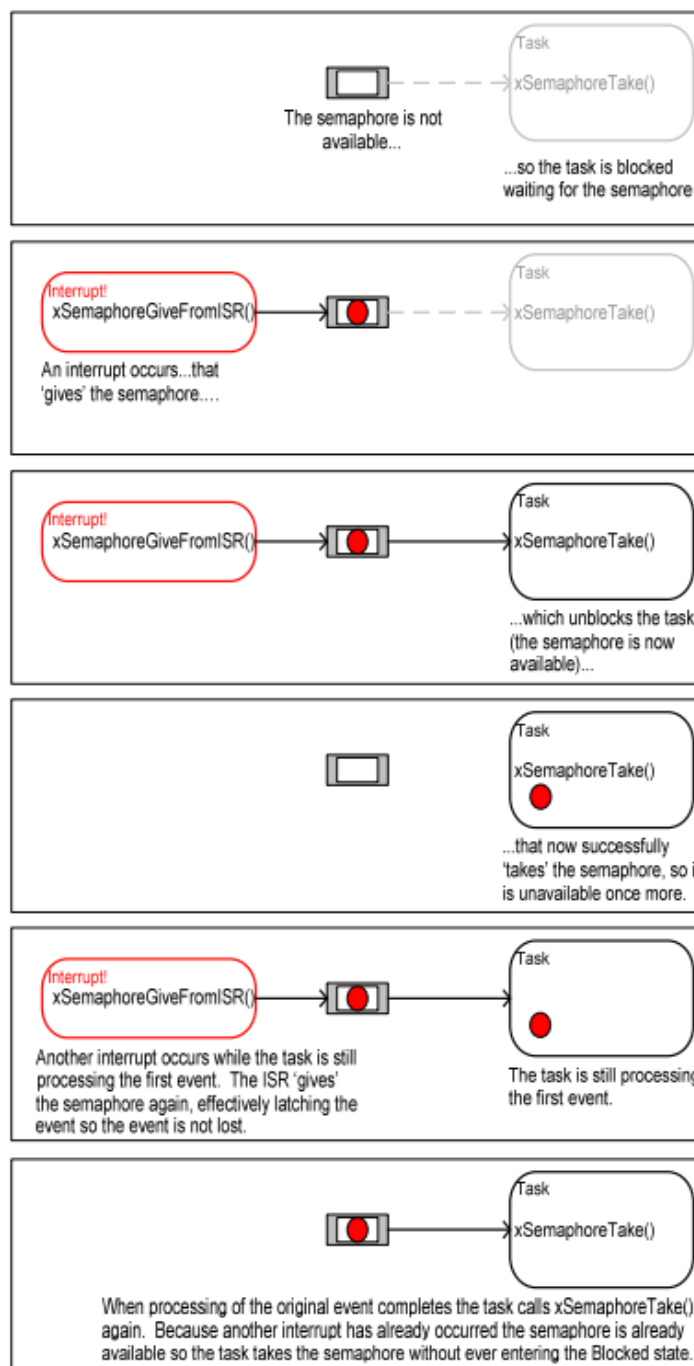
Esta secuencia es adecuada si la interrupción puede ocurrir en una frecuencia relativamente baja. Si otra interrupción ocurre antes que la tarea manipuladora complete su procesamiento, el semáforo binario

estaría nuevamente disponible, permitiendo que se vuelva a procesar el nuevo evento inmediatamente después de terminar de procesar el primero. La tarea manipuladora no entraría en estado bloqueado entre el procesado del primer y segundo evento. Este escenario se muestra en la Figura 30.

La Figura 30 muestra que un semáforo binario puede retener como máximo un evento de interrupción. Si ocurren más eventos, mientras el semáforo está disponible, estos se perderán. Esto se puede evitar usando un semáforo counting en lugar de uno binario.

Habíamos mencionado que un semáforo binario se puede pensar como una cola de un solo elemento. Los semáforos counting, entonces, serán semáforos que pueden tener un largo de más de un elemento. Las tareas no se interesan en los datos que estén almacenados en estas colas, solo se fijan si están vacías o no. Cada vez que un semáforo counting es dado, un espacio de la cola es liberado. El número de ítems en la cola, es la “cuenta” del semáforo.

Figura 30 – Un semáforo binario puede retener sólo un evento.



El semáforo no está disponible. Por lo tanto, la tarea está bloqueada, esperando el semáforo.

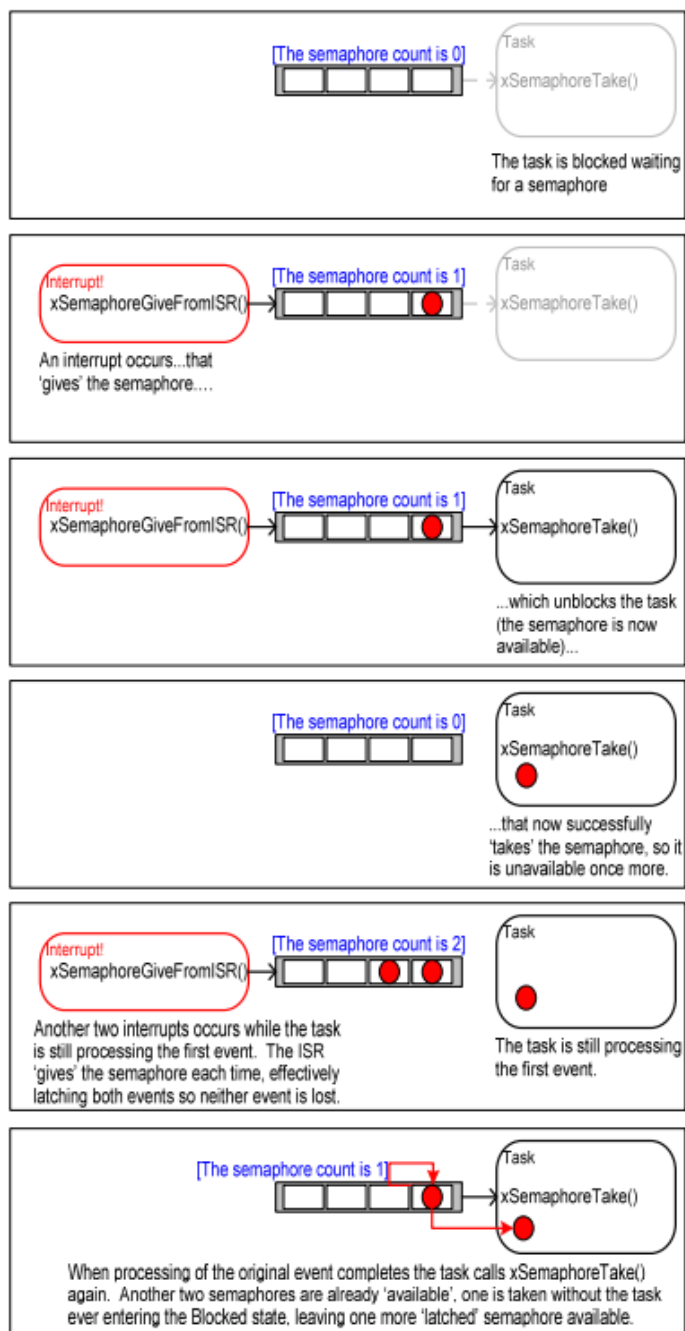
Ocurre una interrupción, que da un semáforo...

... Se desbloquea la tarea (el semáforo ahora está disponible)...

La tarea toma el semáforo, por lo que ahora deja de estar disponible.

Ocurre otra interrupción mientras la tarea está procesando el primer evento. La ISR da el semáforo nuevamente, y queda guardado. La tarea sigue procesando el primer evento.

Cuando termina el procesamiento del primer evento, la tarea vuelve a tomar



el semáforo, que, como se encontraba disponible, comienza a procesarse, sin entrar en estado bloqueado.

La tarea está bloqueada esperando al semáforo.

Ocurre una interrupción, que da un semáforo...

... Se desbloquea la tarea (el semáforo ahora está disponible)...

La tarea toma el semáforo, por lo que ahora deja de estar disponible.

Ocurren dos nuevas interrupciones mientras la tarea está procesando el primer evento. La ISR da dos semáforos, guardando a ambos, por lo que no se pierde ninguno.

Cuando se termina de procesar el primer evento, se toma un semáforo, quedando ahora 1 más disponible. La tarea en este caso continúa procesando el siguiente evento, sin entrar en estado bloqueado.

Figura 31 – Usando un semáforo counting para contar eventos.

Los semáforos counting son típicamente usados para:

1. Contar eventos:

Un evento da un semáforo cada vez que ocurre, causando que la cuenta del semáforo se incremente cada vez que se da un semáforo. La tarea manipuladora toma un semáforo cada vez que procesa un evento, causando que la cuenta se reduzca. La cuenta del semáforo es la diferencia entre la cantidad de eventos que se han producido y la cantidad de eventos que se han procesado. Este mecanismo se muestra en la figura 31.

Los semáforos counting que se usan para contar eventos se crean con un valor inicial de cero.

2. Gestión de Recursos:

La cuenta del semáforo indica el número de recursos disponibles. Para tomar control de un recurso, una tarea debe primero obtener un semáforo, reduciendo el valor de la cuenta de semáforos.

Cuando la cuenta llega a cero, no hay más recursos libres. Cuando una tarea termina de usar un recurso, devuelve el semáforo, incrementando el valor de la cuenta.

Los semáforos counting que son usados para gestión de recursos son creados con valor inicial igual al número de recursos disponibles. El capítulo 4 cubre el uso de semáforos para la gestión de recursos.

Función API xSemaphoreCreateCounting():

Los manipuladores de todos los tipos de semáforos de FreeRTOS son almacenados en una variable del tipo xSemaphoreHandle.

Antes de poder usar un semáforo, este debe ser creado. Para crear un semáforo counting se usa esta función API, cuyo prototipo es:

```
xSemaphoreHandle xSemaphoreCreateCounting( unsignedportBASE_TYPE uxMaxCount,
                                           unsignedportBASE_TYPE uxInitialCount );
```

Listado 48 – Prototipo de la función API xSemaphoreCreateCounting().

Tabla 15 Parámetros y valores de retorno de la función vSemaphoreCreateCounting().

Nombre del Parámetro / Valor de Retorno	Descripción
uxMaxCount	Valor máximo de la cuenta de semáforo. Si lo vemos como una analogía a una cola, se trata del largo de la cola. Es el número máximo de eventos que pueden ser guardados en el semáforo.
uxInitialCount	Valor inicial de la cuenta, luego que el semáforo fue creado. Si el semáforo se usa para contar eventos, se inicia en cero, y si se usa para gestionar recursos, se inicia al valor máximo de la cuenta.
Valor de Retorno	Se retorna NULL si el semáforo no se pudo crear porque no había suficiente memoria disponible. El capítulo 5 provee más información sobre la gestión de memoria. Si se retorna un valor no nulo, indica que el semáforo se creó satisfactoriamente.

Ejemplo 13. Usando un semáforo counting para sincronizar una tarea con una interrupción.

El ejemplo 13 mejora el Ejemplo 12, usando un semáforo counting en lugar de uno binario. El main() es cambiado para incluir un xSemaphoreCreateCounting() en lugar de un vSemaphoreCreateBinary(). Esto se muestra en el Listado 49.

Para simular la ocurrencia de múltiples eventos a una frecuencia elevada, la interrupción de servicio de rutina es cambiada para dar más de un semáforo por cada interrupción. Cada evento es guardado en el valor de la cuenta del semáforo. La rutina de servicio de interrupción modificada se muestra en el Listado 50.

```
static void __interrupt __far vExampleInterruptHandler( void )
{
    static portBASE_TYPE xHigherPriorityTaskWoken;
    xHigherPriorityTaskWoken = pdFALSE;
    /* Se entrega el semáforo varias veces. La primera entrega desbloquea la tarea manipuladora, las
    siguientes entregas se hacen para mostrar cómo la cuenta del semáforo almacena los eventos, para
    permitir que la tarea manipuladora procese el evento sin perder los demás eventos que siguen lle-
    gando.*/
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );

    if( xHigherPriorityTaskWoken == pdTRUE )
    {
        /*Cuando se entregó el semáforo, se desbloqueó la tarea manipuladora, que tiene un ni-
        vel de prioridad mayor que la tarea que estaba corriendo antes. Cambio de contexto para
        un puerto Open Watcom DOS .*/
        portSWITCH_CONTEXT();
    }
}
```

Listado 50 – Implementación de la rutina de servicio de interrupción para el Ejemplo 13.

Todas las demás funciones siguen iguales al Ejemplo 12.

La salida producida cuando el Ejemplo 13 es ejecutado se muestra en la Figura 32. Como puede observarse, la tarea manipuladora procesa los tres eventos (simulados), cada vez que una interrupción es generada. Los eventos fueron guardados en la cuenta del semáforo, permitiendo que la tarea manipuladora los procese en orden.



Figura 32 – Salida producida cuando el Ejemplo 13 es ejecutado.

3.4 Usando colas en una rutina de servicio de interrupción:

`xQueueSendToFrontFromISR()`, `xQueueSendToBackFromISR()` y `xQueueReceiveFromISR()` son versiones de `xQueueSendToFront()`, `xQueueSendToBack()`, y `xQueueReceive()`, respectivamente, y son adecuadas para usarse en un servicio de interrupción de rutina.

Los Semáforos se usan para comunicar eventos. Las colas se usan tanto para comunicar eventos como para transferir datos.

Funciones API `xQueueSendToFrontFromISR()` y `xQueueSendToBackFromISR()`:

`xQueueSendToFrontFromISR()` es equivalente a `xQueueSendToBackFromISR()`.

```
portBASE_TYPE xQueueSendToFrontFromISR( xQueueHandle xQueue,  
                                         void *pvItemToQueue  
                                         portBASE_TYPE *pxHigherPriorityTaskWoken  
                                         );
```

Listado 51 – Prototipo de la función API `xQueueSendToFrontFromISR()`.

```
portBASE_TYPE xQueueSendToBackFromISR( xQueueHandle xQueue,  
                                         void *pvItemToQueue  
                                         portBASE_TYPE *pxHigherPriorityTaskWoken  
                                         );
```

Listado 52 – Prototipo de la Función API `xQueueSendToBackFromISR()`.

Tabla 16 xQueueSendToFrontFromISR() y xQueueSendToBackFromISR(), parámetros y valores de retornos.

Parámetros y Valores de Retorno	Descripción
xQueue	El manipulador de la cola a la cual se enviará el dato (escritura). El manipulador de la cola es retornado de la función xQueueCreate(), al crear la cola.
pvItemToQueue	Puntero al dato que será copiado en la cola. El tamaño de cada ítem que la cola puede guardar es establecido cuando se crea la cola. Esta será la cantidad de bytes que serán copiados desde pvItemToQueue a la cola.
pxHigherPriorityTaskWoken	Es posible que una misma cola tenga más de una tarea bloqueada, esperando a que un dato llegue a la cola. Llamando a una de estas funciones llegara un dato a la cola, haciendo que una tarea deje el estado bloqueado. Si esta tarea que deja el estado bloqueado tiene un nivel de prioridad mayor a la tarea que fue interrumpida, la función API seteará internamente * pxHigherPriorityTaskWoken a pdTRUE. Esto significa que se deberá realizar un cambio de contexto, antes de salir de la interrupción, para asegurar que la interrupción retorna directamente a la función de mayor nivel de prioridad.
Valor retornado	Hay dos posibles valores: 1. pdPass: Dato enviado a la cola satisfactoriamente. 2. errQUEUE_FULL: el dato no se pudo enviar porque la cola estaba llena.

Uso Eficiente de Colas:

La mayoría de las demostraciones y ejemplos de FreeRTOS usan una simple diver de UART que usan colas para pasar caracteres al manipulador de interrupción, y para enviar caracteres desde el manipulador. Cada carácter que es transmitido o recibido pasan a través de una cola. Los drivers de la UART son implementados de esta manera solo porque es un modo conveniente de mostrar cómo usar las colas desde interrupciones.

Pasar caracteres individuales a través de una cola es extremadamente ineficiente (especialmente con altos baud rates), por lo que no es recomendado. Técnicas más eficientes son:

- Ubicar cada caracter recibido en un buffer RAM, luego, usar un semáforo para desbloquear una tarea que procese el buffer luego que el mensaje completo es recibido.
- Interpretar los caracteres recibidos directamente dentro de la rutina de interrupción de servicio, luego, usar una cola para enviar los comandos interpretados y decodificados a la tarea que realizará el procesamiento (similar a lo mostrado en la Figura 23). Esta técnica solo es adecuada si la interpretación de la cadena de datos es suficientemente rápida como para realizarse dentro de la interrupción.

Ejemplo 14. Enviar y recibir datos en una cola desde una interrupción.

Este ejemplo muestra las funciones `xQueueSendToBackFromISR()` y `xQueueReceiveFromISR()` siendo usadas con la misma interrupción. Como antes, por conveniencia es usada una interrupción por software.

Una tarea periódica es creada para enviar cinco números a una cola cada 200 milisegundos. Genera una interrupción por software solo luego que los cinco valores fueron enviados. La implementación de la tarea es mostrada en el Listado 53.

```
static void vIntegerGenerator( void *pvParameters )
{
    portTickType xLastExecutionTime;
    unsigned portLONG ulValueToSend = 0;
    int i;

    /*Inicializa la variable usada por la función vTaskDelayUntil(). */
    xLastExecutionTime = xTaskGetTickCount();

    for( ;; )
    {
        /* Esta es una tarea periodica. La tarea se ejecuta cada 200ms, y se bloquea hasta
        volver a ser ejecutada. */
        vTaskDelayUntil( &xLastExecutionTime, 200 /portTICK_RATE_MS );
        /* Envía 5 veces un número incrementado a la cola. Los valores serán leídos
        desde la cola por la rutina de servicio de interrupción. Esta rutina vacía la cola, por
        lo que se garantiza que esta tarea va a volver a llenarla, por lo que no es necesario
        especificar un tiempo de bloqueo. */
        for( i = 0; i < 5; i++ )
        {
            xQueueSendToBack( xIntegerQueue, &ulValueToSend, 0 );
            ulValueToSend++;
        }

        /* Fuerza una interrupción para que la rutina de servicio pueda leer los valores de
        la cola. */
        vPrintString( "Generator task - About to generate an interrupt.\r\n" );
        __asm{ int 0x82 } /* Esta línea genera la interrupción. */
        vPrintString( "Generator task - Interrupt generated.\r\n\r\n\r\n" );
    }
}
```

Listado 53 – Implementación de la tarea que escribe en la cola en el Ejemplo 14.

La rutina de servicio de interrupción llama repetidamente a la función `xQueueReceiveFromISR()`, hasta que todos los valores escritos en la cola por la tarea periódica hayan sido removidos y la cola quede totalmente vacía. Los últimos dos bits de cada valor recibido son usados como un índice dentro de un vector de cadenas, con un puntero a la cadena con el correspondiente índice, siendo enviada a una cola diferente usando la función `xQueueSendFromISR()`. La implementación de la rutina de servicio de interrupción es mostrada en el Listado 54.

```
static void __interrupt __far vExampleInterruptHandler( void )
{
    static portBASE_TYPE xHigherPriorityTaskWoken;
    static unsigned long ulReceivedNumber;

    /* Las cadenas son declaradas como static const para asegurar que no son asignadas en la pila del
    ISR, y que existan incluso cuando la ISR no se está ejecutando. */
    static const char *pcStrings[] =
    {
        "String 0\r\n",
        "String 1\r\n",
        "String 2\r\n",
        "String 3\r\n"
    };

    xHigherPriorityTaskWoken = pdFALSE;
    /* Repite hasta que la cola se vacíe. */
    while( xQueueReceiveFromISR( xIntegerQueue,
                                &ulReceivedNumber,
                                &xHigherPriorityTaskWoken ) != errQUEUE_EMPTY )
    {
        /* Trunca el valor recibido hasta los últimos dos bits, luego envía un puntero a la cadena
        que corresponde con el valor truncado. */
        ulReceivedNumber &= 0x03;
        xQueueSendToBackFromISR( xStringQueue,
                                &pcStrings[ ulReceivedNumber ],
                                &xHigherPriorityTaskWoken );
    }
    /* Pregunta si al enviar o recibir de la cola desbloquea alguna tarea con prioridad más alta
    que la que estaba ejecutándose. Si esto ocurre, fuerza un cambio de contexto */
    if( xHigherPriorityTaskWoken == pdTRUE )
    {
        /* Nota: La macro que se debe usar para forzar el cambio de contexto depende
        del puerto. Esta es la macro correcta para usar un puerto DOS Open Watcom.
        Otros puertos pueden requerir una sintaxis diferente. */
        /*
        portSWITCH_CONTEXT();
        */
    }
}
```

Listado 54 – La implementación de la rutina de servicio de interrupción usada en el Ejemplo 14.

La tarea que recibe el puntero de carácter desde la rutina de servicio de interrupción solo se bloquea hasta que llegue un mensaje a la cola, imprimiendo cada cadena una vez que la recibe. Su implementación se muestra en el Listado 55.

```
static void vStringPrinter( void *pvParameters )
{
    char *pcString;
    for( ;; )
    {
        /* Se bloquea hasta que lleguen los datos. */
        xQueueReceive( xStringQueue, &pcString, portMAX_DELAY );

        /* Imprime la cadena recibida. */
        vPrintString( pcString );
    }
}
```

Listado 55 – Tarea que imprime la cadena recibida desde la rutina de servicio de interrupción en el Ejemplo 14.

Como es habitual, el main() crea las colas necesarias y las tareas antes de arrancar el scheduler. Su implementación se muestra en el Listado 56.

```
int main( void )
{
    /* Previo a utilizar una cola esta debe ser creada. Se crean las dos colas que se usarán en
    este ejemplo. Una cola puede contener variables del tipo unsigned long, mientras que la
    otra puede contener variables del tipo char*. Ambas tienen un máximo de 10 ítems. Una
    aplicación real debería chequear los valores de retorno para asegurar que ambas colas fue-
    ron creadas. */
    xIntegerQueue = xQueueCreate( 10, sizeof( unsigned long ) );
    xStringQueue = xQueueCreate( 10, sizeof( char * ) );

    /* Instala el manipulador de interrupciones. */
    _dos_setvect( 0x82, vExampleInterruptHandler );

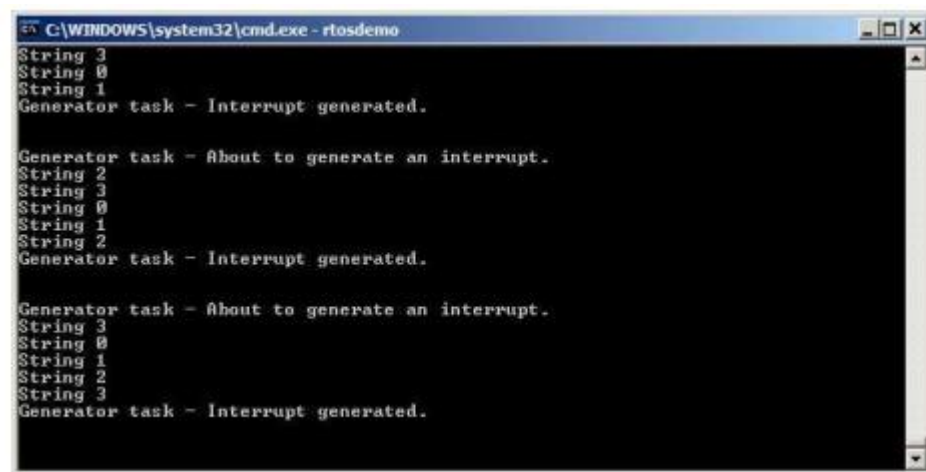
    /* Crea la tarea que usa una cola para pasar enteros a la rutina de interrupción de servicio.
    La tarea se crea con prioridad 1. */
    xTaskCreate( vIntegerGenerator, "IntGen", 1000, NULL, 1, NULL );

    /* Crea la tarea que imprime la cadena enviada desde la rutina de servicio de interrupción.
    Esta tarea se crea con prioridad 2. */
    xTaskCreate( vStringPrinter, "String", 1000, NULL, 2, NULL );
}
```

```
/* Inicia el scheduler para que last areas empiecen a ejecutarse. */  
vTaskStartScheduler();  
  
/*Si todo anda bien, el main() nunca debería alcanzar este punto, dado que el Scheduler va  
a estar siempre ejecutando las tareas. Si el main() alcanza este punto, entonces el proba-  
ble que la memoria disponible no era suficiente para crear la tarea ociosa. El capítulo 5  
provee más información sobre el manejo de memoria. */  
for(;;);  
}
```

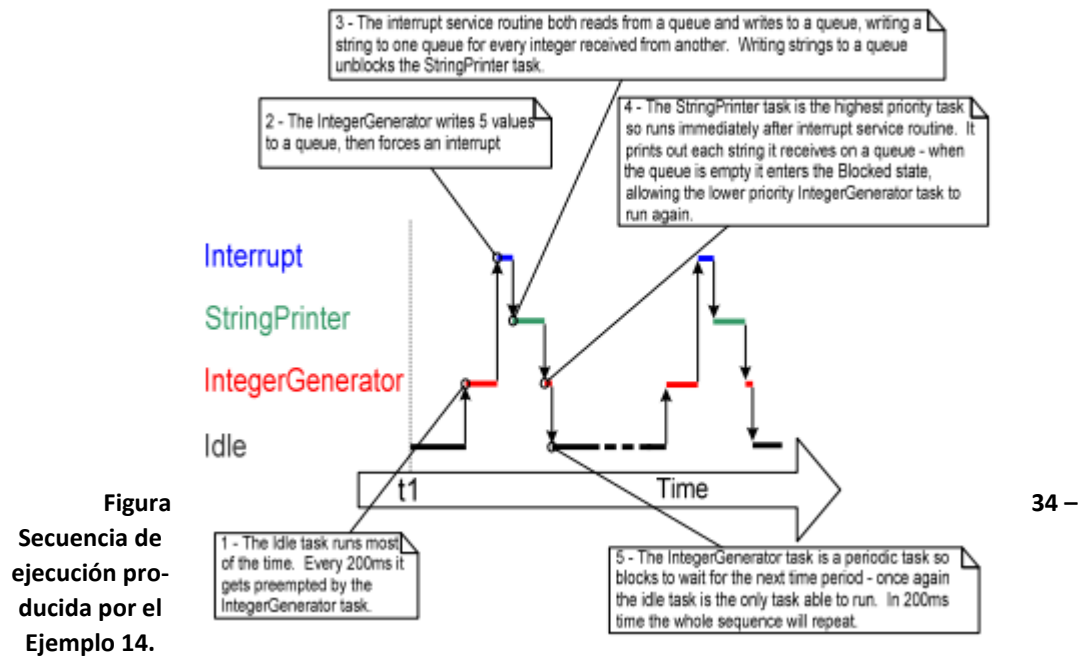
Listado 56 – Función main() para el Ejemplo 14.

La salida producida cuando el Ejemplo 14 se ejecuta es mostrada en la Figura 33. Como puede observarse, La interrupción está recibiendo los 5 enteros y produciendo 5 cadenas como respuesta. Más explicaciones dadas en la Figura 34.



```
C:\WINDOWS\system32\cmd.exe - rtosdemo  
String 3  
String 0  
String 1  
Generator task - Interrupt generated.  
  
Generator task - About to generate an interrupt.  
String 2  
String 3  
String 0  
String 1  
String 2  
Generator task - Interrupt generated.  
  
Generator task - About to generate an interrupt.  
String 3  
String 0  
String 1  
String 2  
String 3  
Generator task - Interrupt generated.
```

Figura 33 – Salida producida cuando el Ejemplo 14 se ejecuta.



- 1- La tarea ociosa corre la mayor parte del tiempo, Cada 200ms es adelantada por la tarea que genera los enteros.
- 2- La tarea generadora de enteros escribe 5 valores a la cola, luego, fuerza una interrupción.
- 3- La rutina de servicio de interrupción lee una cola y escribe en una cola. Escribe una cadena en una cola por cada entero recibido desde la otra cola. Al escribir cadenas en una cola se desbloquea la tarea de impresión.
- 4- La tarea de impresión es la tarea con mayor prioridad, por lo que se ejecuta inmediatamente después de la rutina de servicio. Imprime cada cadena que recibe de la cola, y cuando la cola está vacía, se bloquea, permitiendo que la tarea de menor prioridad (la que genera los enteros) se ejecute nuevamente.
- 5- La tarea generadora de enteros es una tarea periódica, por lo que se bloquea esperando el próximo periodo. Nuevamente la tarea ociosa se ejecutará. Cada 200 milisegundos, se repite toda esta secuencia.

3.5 Anidación de Interrupciones:

Los puertos de FreeRTOS más recientes permiten que las interrupciones se aniden. Estos puertos necesitan que una o las dos constantes detalladas en la Tabla 17 sean definidas en FreeRTOSConfig.h.

Tabla17 Constantes que controlan la anidación de Interrupciones.

Constante	Descripción
configKERNEL_INTERRUPT_PRIORITY	Setea la prioridad de la interrupción tick. Si el puerto no usa la constante configMAX_SYSCALL_INTERRUPT_PRIORITY, entonces cualquier interrupción que utiliza la interrupción Funciones API de seguridad de FreeRTOS también deben ejecutar en esta prioridad.
configMAX_SYSCALL_INTERRUPT_PRIORITY	Setea el valor más elevado de prioridad las interrupciones.

Un modelo de anidación de interrupciones completa se crea mediante el establecimiento de configMAX_SYSCALL_INTERRUPT_PRIORITY a una prioridad mayor que configKERNEL_INTERRUPT_PRIORITY. Esto se demuestra en la Figura 35, que muestra escenario hipotético donde configMAX_SYSCALL_INTERRUPT_PRIORITY ha sido ajustado a tres y configKERNEL_INTERRUPT_PRIORITY se ha fijado a uno. Se muestra un microcontrolador hipotético que tiene siete niveles de prioridad de interrupción diferentes. El valor de siete es sólo un número arbitrario de este ejemplo hipotético y no se pretende que sean representativos de cualquier arquitectura en particular.

Es común que surja la confusión entre las prioridades de trabajo y prioridades de interrupción. La Figura 35 muestra prioridades de interrupción, según la definición de la arquitectura del microcontrolador. Estas son las prioridades controladas por hardware, que las rutinas de interrupción de servicio ejecutan unas relativas a otras. Las tareas no se ejecutan en rutinas de servicio de interrupción, por lo que la prioridad de software asignado a una tarea no está de ninguna manera relacionada con la prioridad de hardware asignada a una fuente de interrupción.

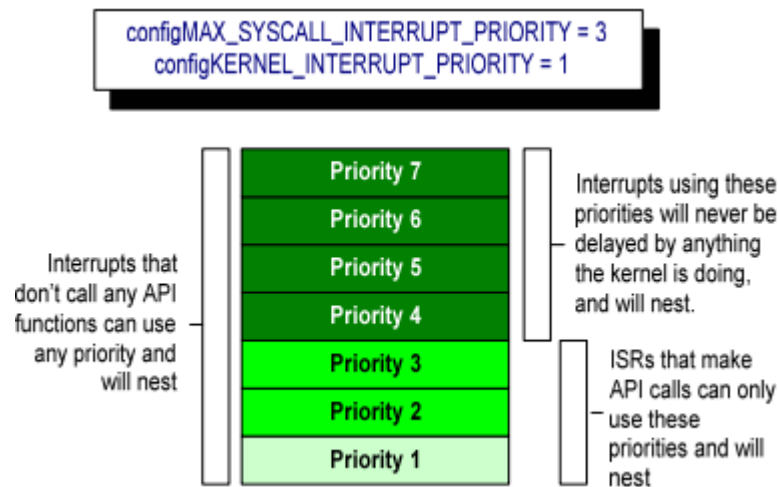


Figura 35 – Constantes que afectan el comportamiento de la anidación de interrupciones.

Referido a la Figura 35:

- Las interrupciones que usan prioridades entre 1 y 3 inclusive, no podrán ejecutarse mientras que el kernel o la aplicación se encuentre dentro de la sección crítica, pero pueden usar las funciones API de interrupciones seguras.
- Las interrupciones que tienen prioridad 4 o mayor no son afectadas por las secciones críticas, por lo tanto, nada que haga el kernel impedirá que estas interrupciones se ejecuten inmediatamente (dentro de las limitaciones del microcontrolador). Funcionalidades típicas que requieren tiempos muy precisos (control de motores por ejemplo) usarán prioridades mayores a configMAX_SYSCALL_INTERRUPT_PRIORITY para asegurar que el scheduler no introduzca variaciones en los tiempos de respuesta de la interrupción.
- Las interrupciones que no hacen uso de llamados a funciones API de FreeRTOS pueden tener cualquier prioridad.

Nota para usuarios de ARM Cortex M3:

El Cortex M3 utiliza números bajos para representar lógicamente niveles de interrupciones de alta prioridad. Esto puede parecer contrario a la intuición y es fácil de olvidar. Si desea asignar una interrupción de baja prioridad, entonces se le debe asignar un alto valor numérico. No asigne una prioridad de 0 (u otro valor numérico bajo) ya que esto puede dar lugar a que la interrupción tenga la más alta prioridad en el sistema, y por lo tanto hacer que su sistema deje de funcionar, si la prioridad es superior a configMAX_SYSCALL_INTERRUPT_PRIORITY.

La prioridad más baja en un núcleo Cortex M3 es 255 aunque diferentes proveedores de Cortex M3 implementan un número diferente de bits de prioridad y suministran funciones de biblioteca que esperan prioridades que deben especificarse en diferentes maneras. Por ejemplo, en el STM32 la prioridad más baja que se puede especificar es 15, y la más alta es 0.