Real Time Operating System "FreeRTOS" Interrupt Management

Sherif Hammad

Using the FreeRTOS Real Time Kernel - a Practical Guide - Cortex M3 Edition (FreeRTOS Tutorial Books)

by Richard Barry

Agenda

- **How Embedded real-time systems have to take actions in response to events that originate from the environment (Interrupt Event Based)**

- **Which FreeRTOS API functions can be used from within an ISR.**

- **How a deferred interrupt scheme can be implemented.**

- **How to create and use binary semaphores and counting semaphores.**

Deferred Interrupt Processing

Binary Semaphores Used for Synchronization

- **A Binary Semaphore can be used to unblock a task each time a particular interrupt occurs**

- **Binary Semaphore synchronizes the task with the interrupt.**

- **This allows the majority of the interrupt event processing to be implemented within the synchronized task**

- **Only a very fast and short portion remaining directly in the ISR.**

- **The interrupt processing is said to have been 'deferred' to a 'handler' task.**

Deferred Interrupt Processing

Binary Semaphores Used for Synchronization

- If the interrupt processing is particularly time critical, then the handler task priority can be set to ensure that the handler task always pre-empts the other tasks in the system.

- The ISR can then be implemented to include a context switch to ensure that the ISR returns directly to the handler task when the ISR itself has completed executing.

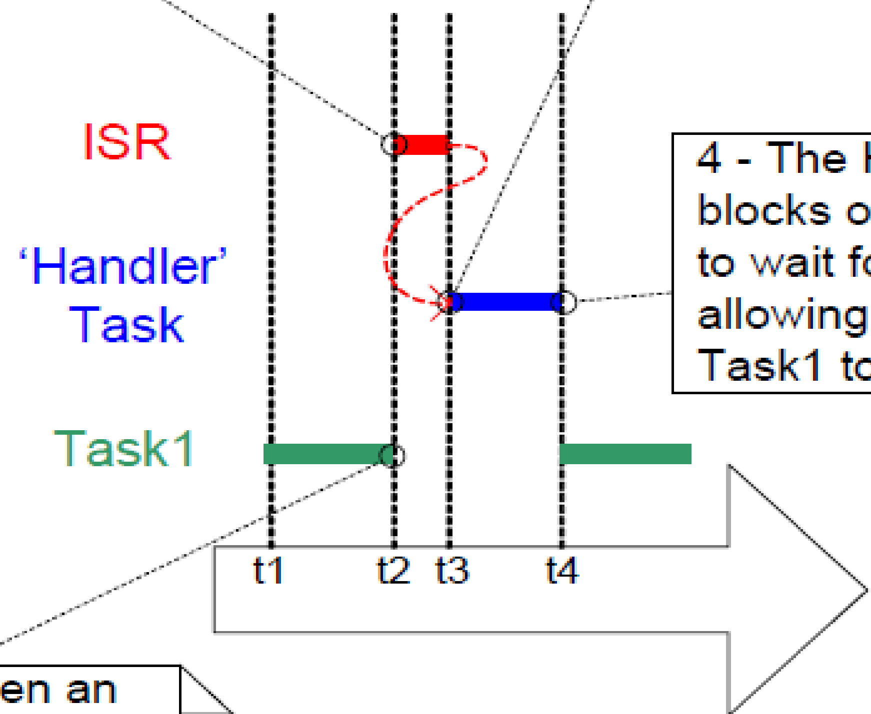
- This has the effect of ensuring that the entire event processing executes contiguously in time, just as if it had all been implemented within the ISR itself.

Deferred Interrupt Processing

Binary Semaphores Used for Synchronization

2 - The ISR executes. The ISR implementation uses a semaphore to unblock the 'Handler Task'.

3 - Because the handler task has the highest priority, and the ISR performs a context switch, the ISR returns directly to the hander task leaving Task1 in the Ready state for now.



4 - The Handler Task blocks on the semaphore to wait for the next event, allowing the lower priority Task1 to run once again.

1 - Task1 is Running when an interrupt occurs.

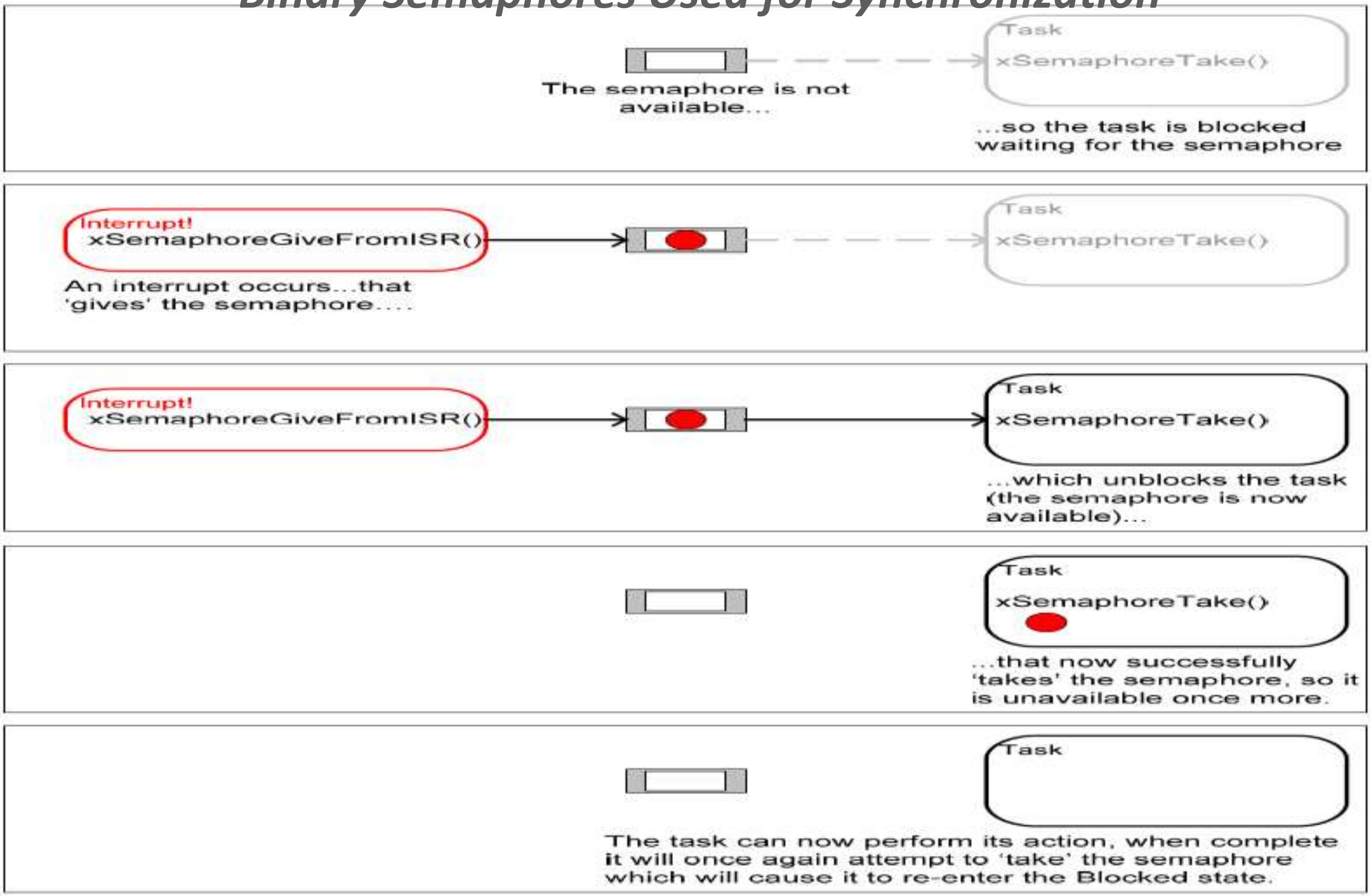
Deferred Interrupt Processing

Binary Semaphores Used for Synchronization

- The handler task uses a blocking 'take' call to a semaphore as a means of entering the Blocked state to wait for the event to occur.
- When the event occurs, the ISR uses a 'give' operation on the same semaphore to unblock the task so that the required event processing can proceed.
- In this interrupt synchronization scenario, the binary semaphore can be considered conceptually as a queue with a length of one. The queue can contain a maximum of one item at any time, so is always either empty or full (hence, binary). By calling xSemaphoreTake()
- The handler task effectively attempts to read from the queue with a block time, causing the task to enter the Blocked state if the queue is empty.
- When the event occurs, the ISR uses the xSemaphoreGiveFromISR() function to place a token (the semaphore) into the queue, making the queue full. This causes the handler task to exit the Blocked state and remove the token, leaving the queue empty once more.
- When the handler task has completed its processing, it once more attempts to read from the queue and, finding the queue empty, re-enters the Blocked state to wait for the next event.

Deferred Interrupt Processing

Binary Semaphores Used for Synchronization



Example 12. Using a binary semaphore to synchronize a task with an interrupt

- **This example uses a binary semaphore to unblock a task from within an interrupt service routine—effectively synchronizing the task with the interrupt.**

- **A simple periodic task is used to generate an interrupt every 500 milliseconds.**

- **A software generated interrupt is used because it allows the time at which the interrupt occurs to be controlled, which in turn allows the sequence of execution to be observed more easily.**

- **The interrupt service routine, which is simply a standard C function.**

- **It does very little other than clear the interrupt and ‘give’ the semaphore to unblock the handler task.**

Example 12. Using a binary semaphore to synchronize a task with an interrupt

- The macro portEND_SWITCHING_ISR() is part of the FreeRTOS Cortex-M3 port and is the ISR safe equivalent of taskYIELD().
- It will force a context switch only if its parameter is not zero (not equal to pdFALSE).
- Note how xHigherPriorityTaskWoken is used. It is initialized to pdFALSE before being passed by reference into xSemaphoreGiveFromISR()
- it will get set to pdTRUE only if xSemaphoreGiveFromISR() causes a task of equal or higher priority than the currently executing task to leave the blocked state. portEND_SWITCHING_ISR() then performs a context switch only if xHigherPriorityTaskWoken equals pdTRUE.

Example 12. Using a binary semaphore to synchronize a task with an interrupt

```
static void vPeriodicTask( void *pvParameters )
{
    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* This task is just used to 'simulate' an interrupt.  This is done by
        periodically generating a software interrupt. */
        vTaskDelay( 500 / portTICK_RATE_MS );

        /* Generate the interrupt, printing a message both before hand and
        afterwards so the sequence of execution is evident from the output. */
        vPrintString( "Periodic task - About to generate an interrupt.\n" );
        mainTRIGGER_INTERRUPT();
        vPrintString( "Periodic task - Interrupt generated.\n\n" );
    }
}
```

Listing 45. Implementation of the task that periodically generates a software interrupt in Example 12

Example 12. Using a binary semaphore to synchronize a task with an interrupt

```
static void vHandlerTask( void *pvParameters )
{
    /* As per most tasks, this task is implemented within an infinite loop.

    Take the semaphore once to start with so the semaphore is empty before the
    infinite loop is entered.  The semaphore was created before the scheduler
    was started so before this task ran for the first time.*/
    xSemaphoreTake( xBinarySemaphore, 0 );

    for( ;; )
    {
        /* Use the semaphore to wait for the event.  The task blocks
        indefinitely meaning this function call will only return once the
        semaphore has been successfully obtained - so there is no need to check
        the returned value. */
        xSemaphoreTake( xBinarySemaphore, portMAX_DELAY );

        /* To get here the event must have occurred.  Process the event (in this
        case we just print out a message). */
        vPrintString( "Handler task - Processing event.\n" );
    }
}
```

Listing 46. The implementation of the handler task (the task that synchronizes with the interrupt) in Example 12

Example 12. Using a binary semaphore to synchronize a task with an interrupt

```
void vSoftwareInterruptHandler( void )
{
portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

    /* 'Give' the semaphore to unblock the task. */
    xSemaphoreGiveFromISR( xBinarySemaphore, &xHigherPriorityTaskWoken );

    /* Clear the software interrupt bit using the interrupt controllers
    Clear Pending register. */
    mainCLEAR_INTERRUPT();

    /* Giving the semaphore may have unblocked a task - if it did and the
    unblocked task has a priority equal to or above the currently executing
    task then xHigherPriorityTaskWoken will have been set to pdTRUE and
    portEND_SWITCHING_ISR() will force a context switch to the newly unblocked
    higher priority task.

    NOTE: The syntax for forcing a context switch within an ISR varies between
    FreeRTOS ports.  The portEND_SWITCHING_ISR() macro is provided as part of
    the Corte M3 port layer for this purpose.  taskYIELD() must never be called
    from an ISR! */
    portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
}
```

Example 12. Using a binary semaphore to synchronize a task with an interrupt

```c
int main( void )
{
    /* Configure both the hardware and the debug interface. */
    vSetupEnvironment();

    /* Before a semaphore is used it must be explicitly created.  In this example
    a binary semaphore is created. */
    vSemaphoreCreateBinary( xBinarySemaphore );

    /* Check the semaphore was created successfully. */
    if( xBinarySemaphore != NULL )
    {
        /* Enable the software interrupt and set its priority. */
        prvSetupSoftwareInterrupt();

        /* Create the 'handler' task.  This is the task that will be synchronized
        with the interrupt.  The handler task is created with a high priority to
        ensure it runs immediately after the interrupt exits.  In this case a
        priority of 3 is chosen. */
        xTaskCreate( vHandlerTask, "Handler", 240, NULL, 3, NULL );

        /* Create the task that will periodically generate a software interrupt.
        This is created with a priority below the handler task to ensure it will
        get preempted each time the handler task exits the Blocked state. */
        xTaskCreate( vPeriodicTask, "Periodic", 240, NULL, 1, NULL );

        /* Start the scheduler so the created tasks start executing. */
        vTaskStartScheduler();
    }

    /* If all is well we will never reach here as the scheduler will now be
    running the tasks.  If we do reach here then it is likely that there was
    insufficient heap memory available for a resource to be created. */
    for( ;; );
}
```

Listing 48. The implementation of main() for Example 12

Console | Problems | Memory | Red Trace Preview

Example12 (Debug) [C/C++ MCU Application] C:\E\Dev\FreeRTOS\DOC\Projects\191-ApplicationNotesAndBook\Source-Code-For-Examples\LPCXp

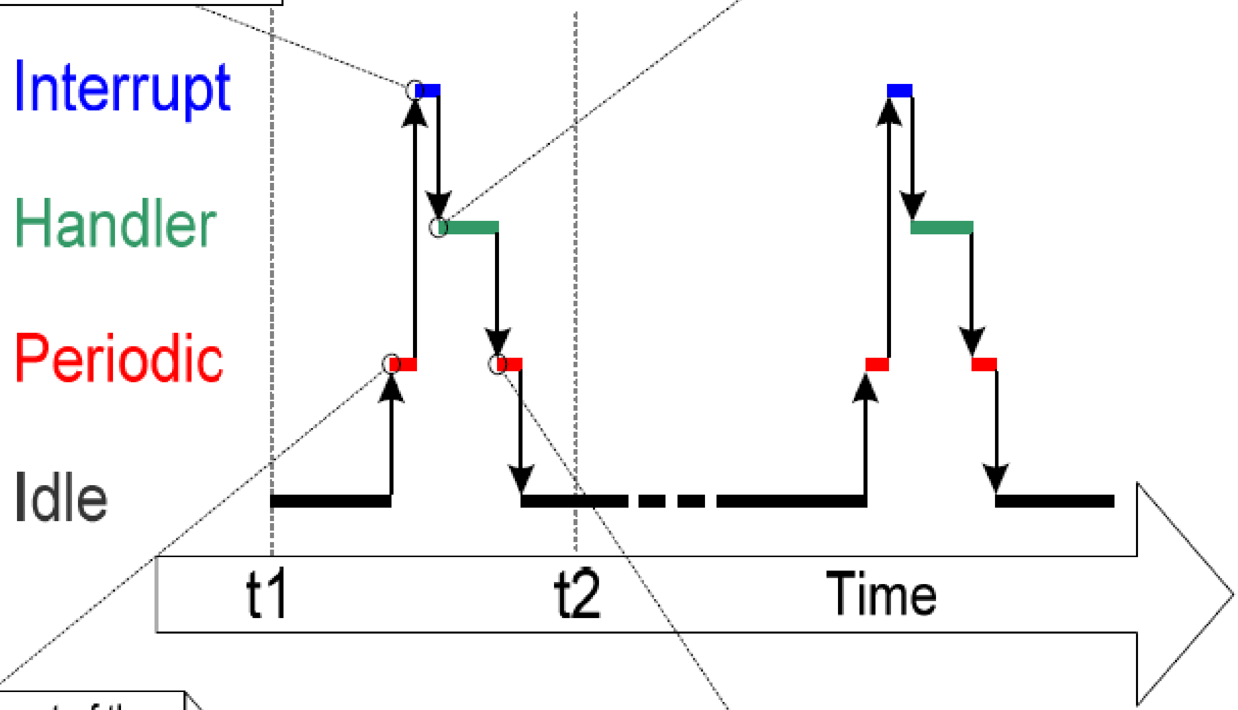
```
Periodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.

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

Example 12. Using a binary semaphore to synchronize a task with an interrupt

2 - The Periodic task prints its first message then forces an interrupt. The interrupt service routine executes immediately.

3 - The interrupt 'gives' the semaphore, causing the Handler task to unblock. The interrupt service routine then returns directly to the Handler task because the Handler task is the highest priority Ready state task. The handler task prints out its message before returning to the Blocked state to wait for the next interrupt.



1 - The Idle task is running most of the time. Every 500ms its gets pre-empted by the Periodic task.

4 - The Periodic task is once again the highest priority task - it prints out its second message before entering the Blocked state again to wait for the next time period. This leaves just the Idle task able to run.