Contents

- Interrupt Management

HAN Embedded Systems Engineering – MIC5

Interrupt Management

Barry, R. (2016). *Mastering the freertos real time kernel. Pre-release 161204 Edition*. Real Time Engineers Ltd.

Chapter 6

Events

Embedded real-time systems have to take actions in response to **events** that originate from the environment.

There are two mechanisms to detect events: **interrupts** and **polling**.

When interrupts are used, **how much processing** should be performed inside the interrupt service routine (ISR), and how much outside? It is normally desirable to keep each ISR as short as possible.

How are events are communicated to the main() function? Shared global memory, abstract data structure (such as a queue), DMA.

Task priority versus interrupt handler priority

A task is a software function that is unrelated to the hardware.

The priority of a task is assigned in software by the application writer, and the scheduler decides which task will be in the Running state.

An interrupt handler is a software function that is controlled by the microcontroller’s hardware.

Tasks will only run when there are no ISRs running.

Using the FreeRTOS API from an ISR

Many FreeRTOS API functions perform actions that are not valid inside an ISR!

Example: calling vTaskDelay() in an ISR

The calling task will be moved into Blocked state, but what task should be placed in Blocked state?

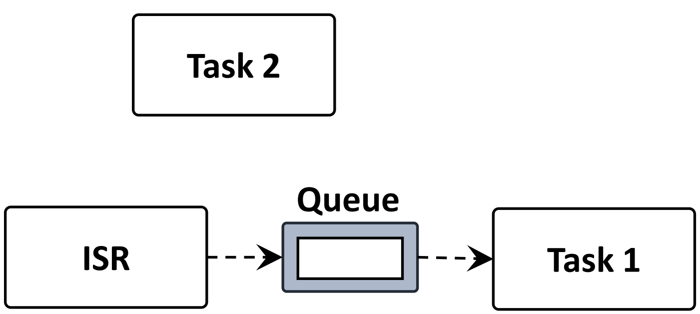
A distinct API has been created in FreeRTOS for functions that are valid for use from ISRs. These function end with *…FromISR()*.

Never call a FreeRTOS API function from an ISR that does not have "…FromISR()" in its name.

The xHigherPriorityTaskWoken Parameter

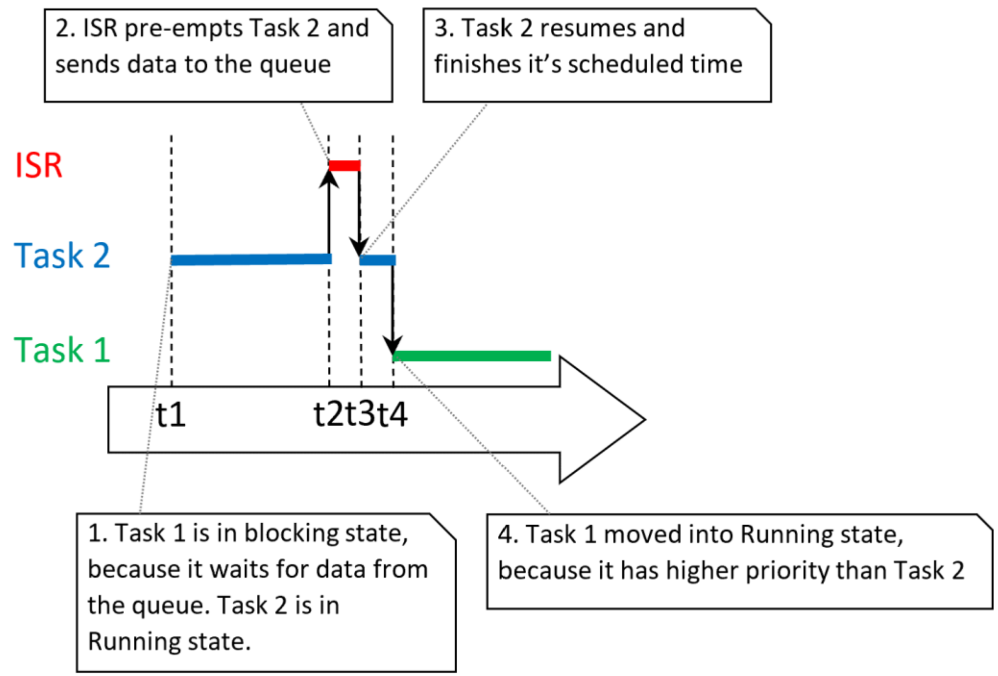
Example:

- Task 1 priority = 5
- Task 2 priority = 2



The ISR receives data and places the data in a queue. Task1 waits in Blocking state for data in the queue.

Normal interrupt handling

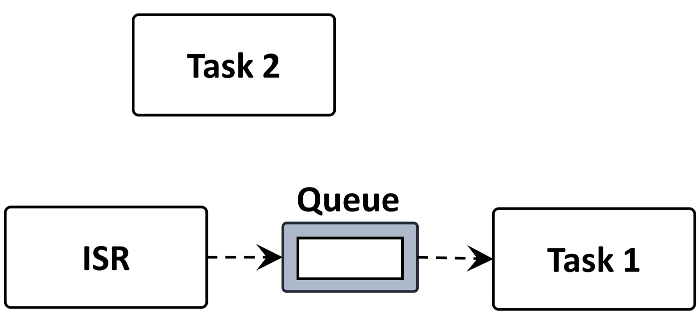


HAN_UNIVERSITY OF APPLIED SCIENCES

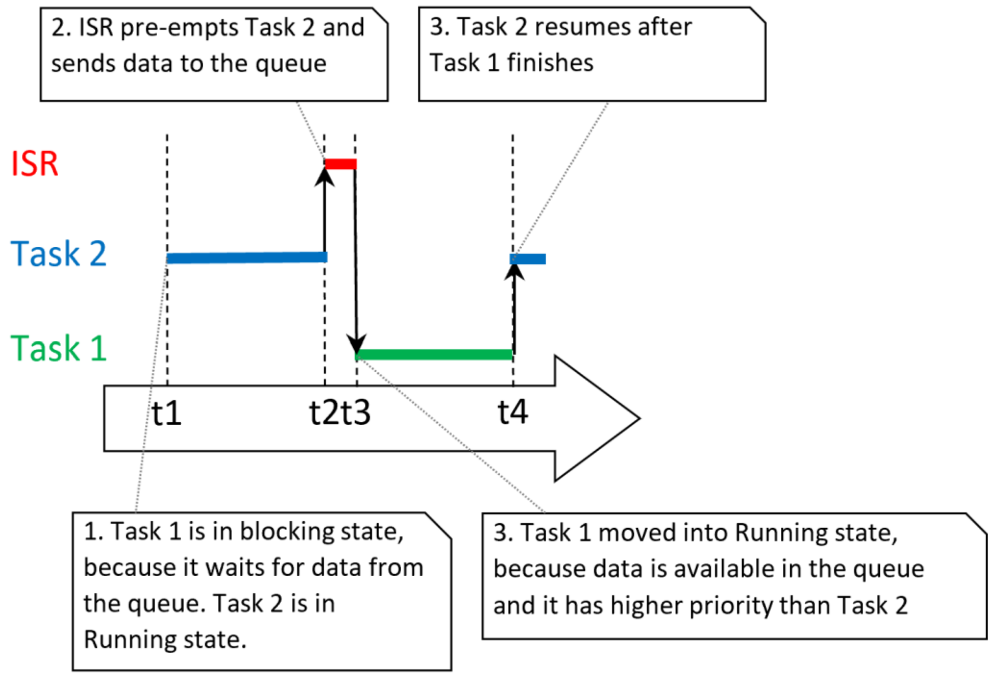
The xHigherPriorityTaskWoken Parameter

Can we move Task 1 into Running state immediately after the ISR has finished?

This is not possible, because an ISR is not aware of the FreeRTOS context!



The ISR receives data and places the data in a queue. Task1 waits in Blocking state for data in the queue.



The xHigherPriorityTaskWoken Parameter

Let's make it aware of the context with the **xHigherPriorityTaskWoken** parameter!

```
void UART0_IRQHandler( void )
{
    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
    char cChar;

    if( UART0->S1 & UART_S1_RDRF_MASK )
    {
        cChar = UART0->D;
        xQueueSendFromISR( xRxedChars,
                           &cChar,
                           &xHigherPriorityTaskWoken );
    }

    portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
}
```

Declare a local variable and set it initially to pdFALSE

The xHigherPriorityTaskWoken Parameter

Let's make it aware of the context with the **xHigherPriorityTaskWoken** parameter!

```
void UART0_IRQHandler( void )
{
    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
    char cChar;

    if( UART0->S1 & UART_S1_RDRF_MASK )
    {
        cChar = UART0->D;
        xQueueSendFromISR( xRxedChars,
                           &cChar,
                           &xHigherPriorityTaskWoken );
    }

    portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
}
```

Call the …FromISR() function. It requires a parameter that is not required in the 'normal' xQueueSend() function. This parameter is set to pdTRUE within the function if <u>another task with a higher priority has woken</u> due to calling this function.

The xHigherPriorityTaskWoken Parameter

Let's make it aware of the context with the **xHigherPriorityTaskWoken** parameter!

```
void UART0_IRQHandler( void )
{
    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
    char cChar;

    if( UART0->S1 & UART_S1_RDRF_MASK )
    {
        cChar = UART0->D;
        xQueueSendFromISR( xRxedChars,
                           &cChar,
                           &xHigherPriorityTaskWoken );
    }

    portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
}
```

An interrupt safe macro is provided that will ensure a context switch (moving Task 1 into Running state instead of Task 2) if the variable has been set to pdTRUE.

Note: portYIELD_FROM_ISR() does the same thing

Interrupt Priority of the Kernel Tick Timer

The FreeRTOS kernel also uses an ISR: a timer that counts the ticks will generate an interrupt as soon as a context switch should take place

For the Cortex-M0 port, this is implemented by the SysTick timer as can be seen by the following define in FreeRTOSConfig.h

```
#define xPortSysTickHandler  SysTick_Handler
```

For the Cortex-M0 port is this priority set (in port.c) to the lowest possible value (192 for the KL25Z), because **the kernel should never interrupt other ISRs**

Interrupt Priority of the Kernel Tick Timer

Other ISRs must therefore run at a priority higher than 192

- 0
- 64
- 128

Interrupt Nesting

Functions in the interrupt safe API (*…FromISR()*) implement a **critical section** to prevent pre-emption. For example:

```
UBaseType_t uxTaskPriorityGetFromISR( const TaskHandle_t xTask )
{
    TCB_t const * pxTCB;
    UBaseType_t uxReturn, uxSavedInterruptState;

    uxSavedInterruptState = portSET_INTERRUPT_MASK_FROM_ISR();
    {
        /* If null is passed in here then it is the priority of
         * the calling task that is being queried. */
        pxTCB = prvGetTCBFromHandle( xTask );
        uxReturn = pxTCB->uxPriority;
    }
    portCLEAR_INTERRUPT_MASK_FROM_ISR( uxSavedInterruptState );

    return uxReturn;
}
```

Saves the current interrupt status (from PRIMASK register) locally and then disables interrupt.

Interrupt Nesting

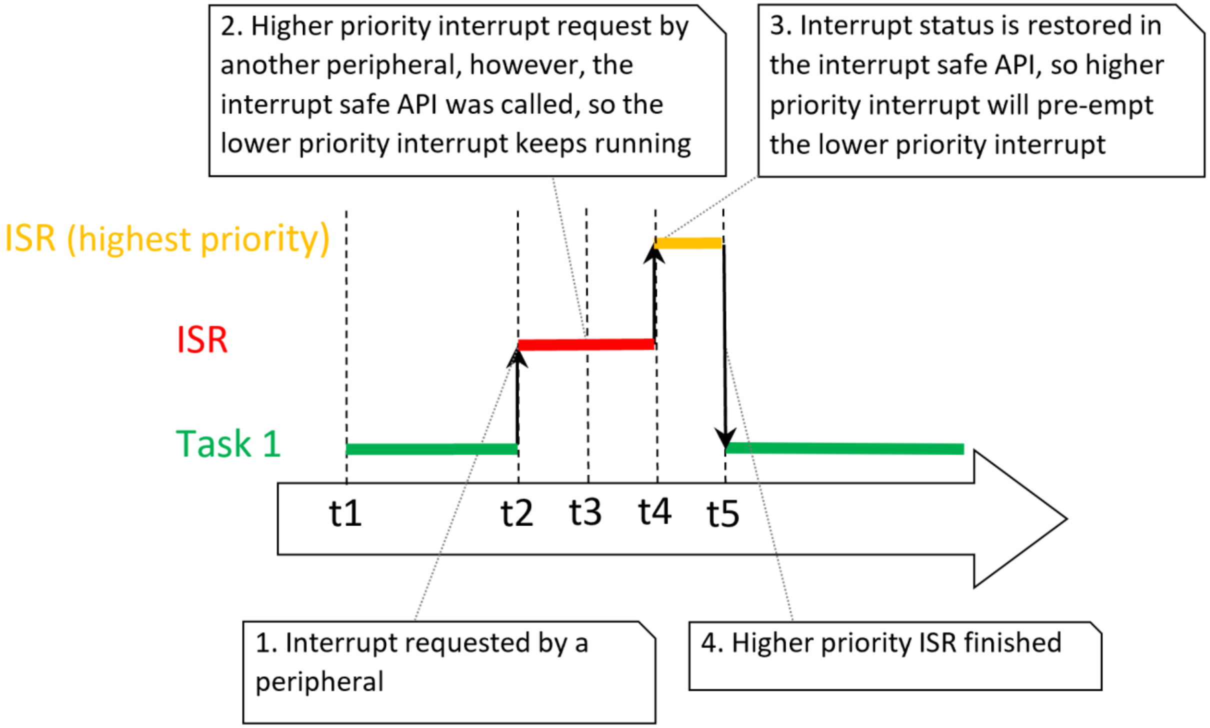
Functions in the interrupt safe API (*…FromISR()*) implement a **critical section** to prevent pre-emption. For example:

```
UBaseType_t uxTaskPriorityGetFromISR( const TaskHandle_t xTask )
{
    TCB_t const * pxTCB;
    UBaseType_t uxReturn, uxSavedInterruptState;

    uxSavedInterruptState = portSET_INTERRUPT_MASK_FROM_ISR();
    {
        /* If null is passed in here then it is the priority of
         * the calling task that is being queried. */
        pxTCB = prvGetTCBFromHandle( xTask );
        uxReturn = pxTCB->uxPriority;
    }
    portCLEAR_INTERRUPT_MASK_FROM_ISR( uxSavedInterruptState );

    return uxReturn;
}
```

Restore interrupt status.

Interrupt Nesting

This means that it is temporarily not possible to pre-empt, even for another higher priority ISR!



Interrupt Nesting

Sidenote

Other Cortex-M cores, such as Cortex-M3, Cortex-M4, etc., have the option to define the minimum priority for exception processing (with a register called BASEPRI).

This opens new options:

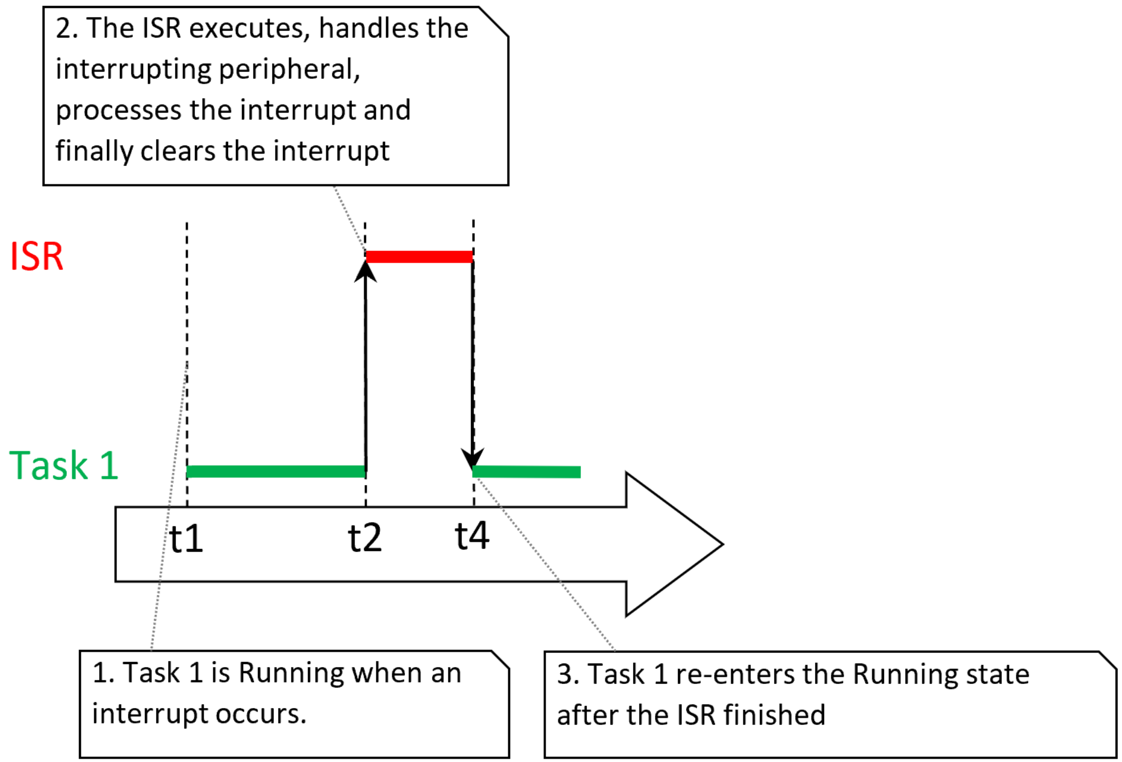
- Disable interrupts up-and-until a specific priority level
- Run high priority interrupt no matter what the kernel is doing, even if it is executing an interrupt safe API function

More detailed information here:

https://www.freertos.org/RTOS-Cortex-M3-M4.html

Deferred Interrupt Processing

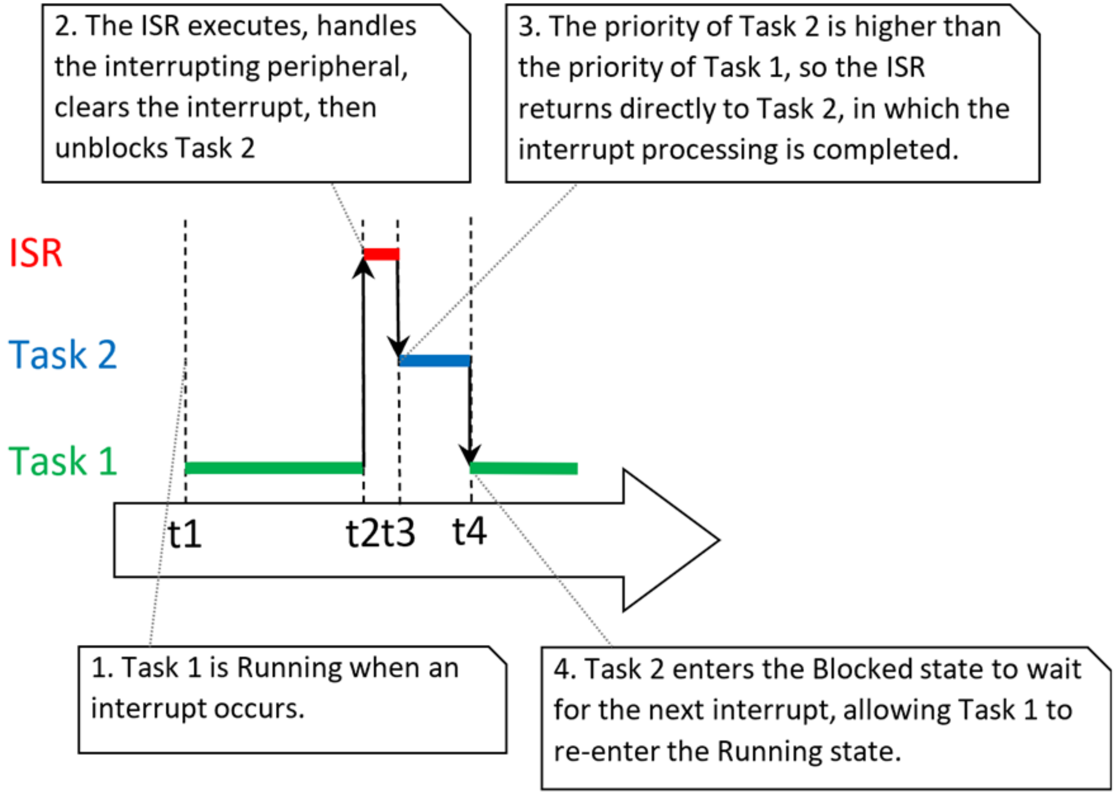
Keep code as short as possible in an ISR, so **don't** do this:



HAN_UNIVERSITY OF APPLIED SCIENCES

Deferred Interrupt Processing

Instead, defer interrupt processing to a task:



Deferred Interrupt Processing

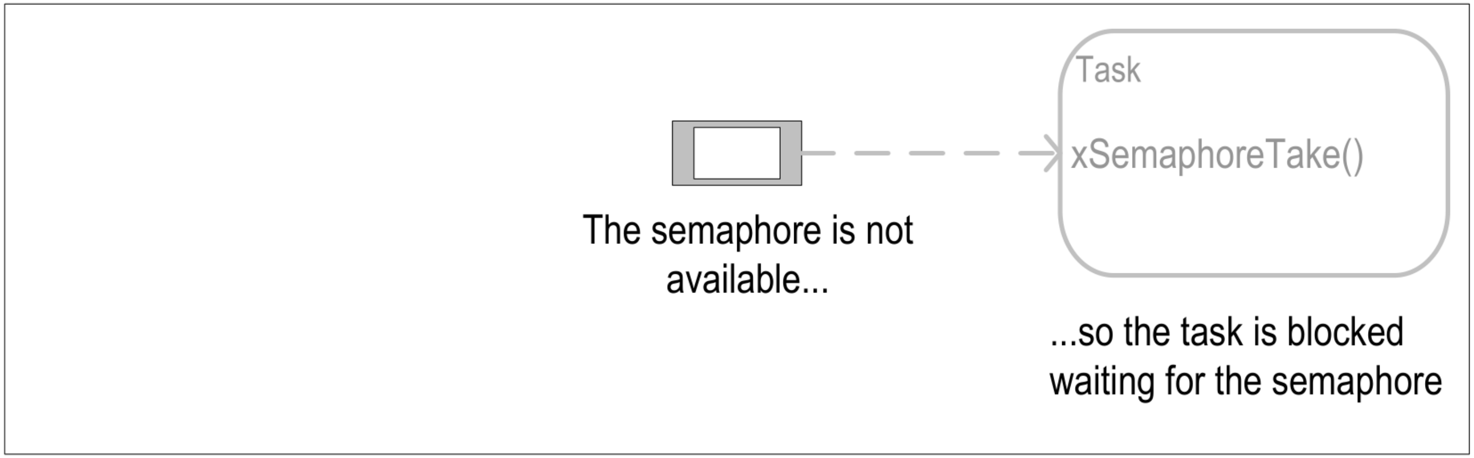
How can we implement this deferred interrupt processing?

- Binary Semaphores
- Counting Semaphores
- Deferring Work to the RTOS Daemon Task
- Queues

Binary Semaphores Used for Synchronization

A binary semaphore is a data structure available in FreeRTOS.

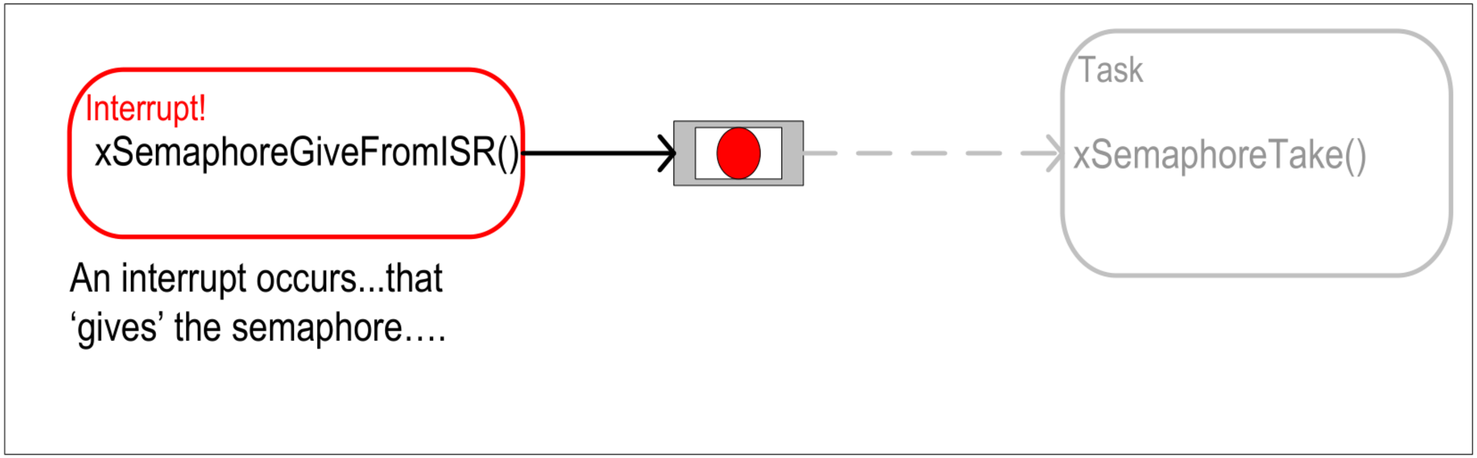
It must be *Given* before it can be *Taken*.



Binary Semaphores Used for Synchronization

A binary semaphore is a data structure available in FreeRTOS.

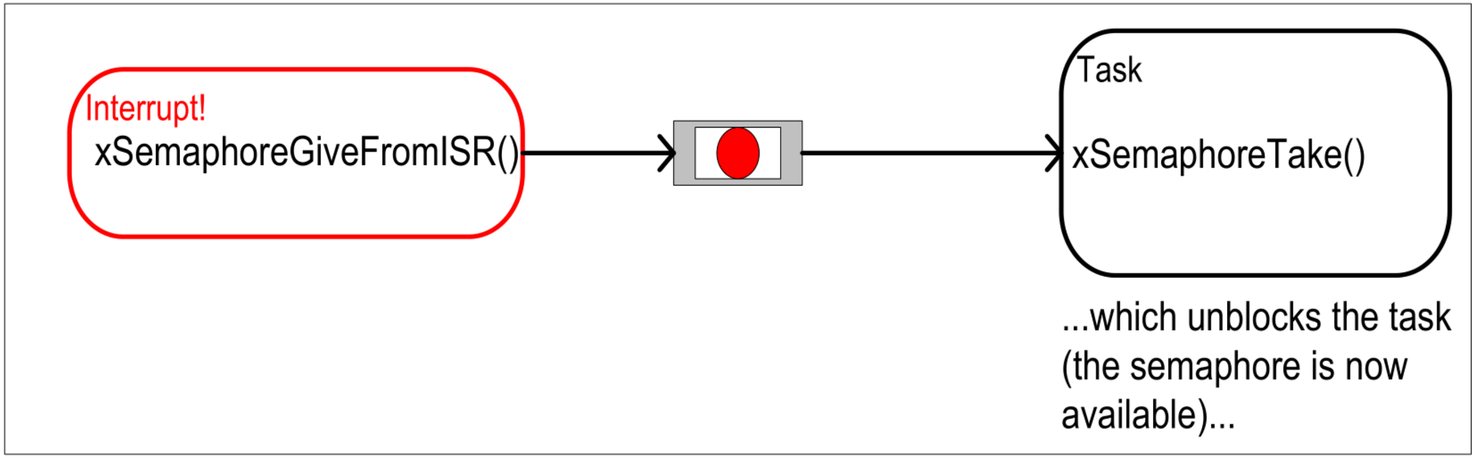
It must be *Given* before it can be *Taken*.



Binary Semaphores Used for Synchronization

A binary semaphore is a data structure available in FreeRTOS.

It must be *Given* before it can be *Taken*.

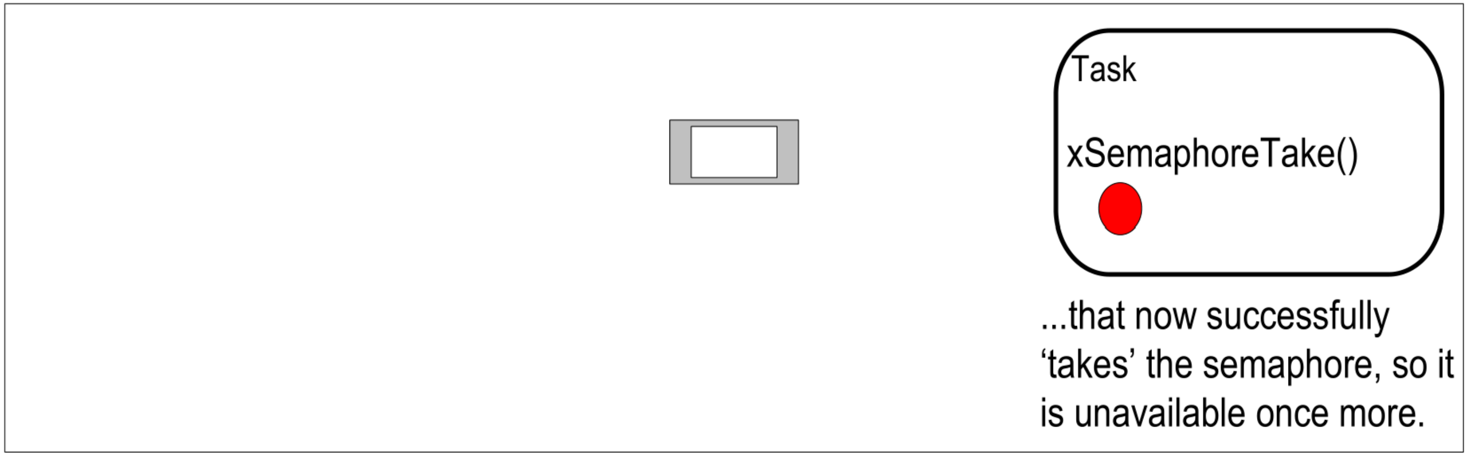


HAN_UNIVERSITY OF APPLIED SCIENCES

Binary Semaphores Used for Synchronization

A binary semaphore is a data structure available in FreeRTOS.

It must be *Given* before it can be *Taken*.

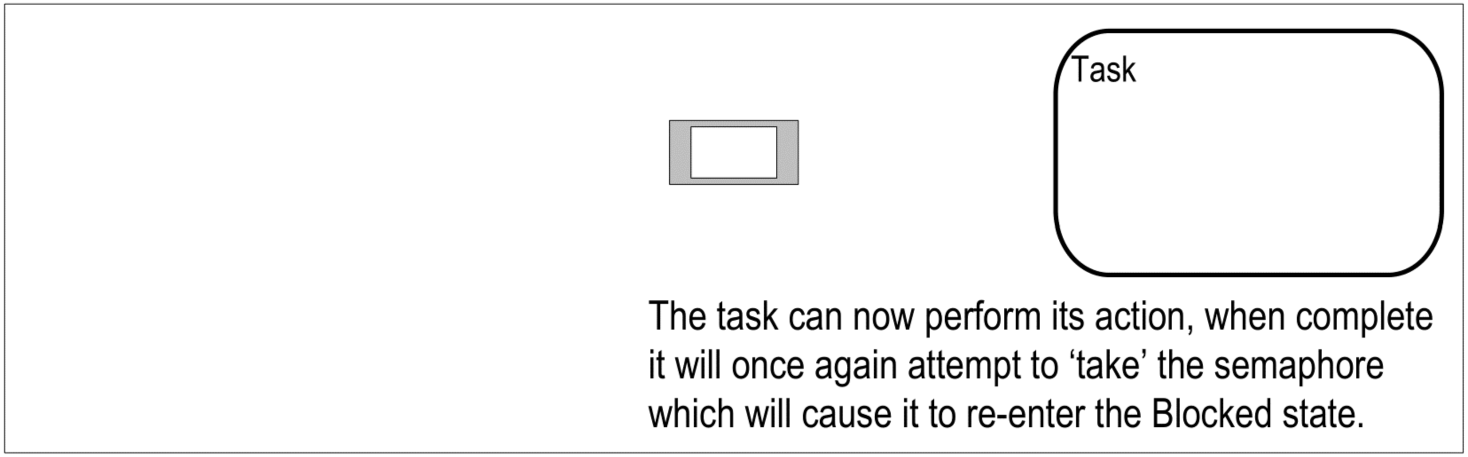


HAN_UNIVERSITY OF APPLIED SCIENCES

Binary Semaphores Used for Synchronization

A binary semaphore is a data structure available in FreeRTOS.

It must be *Given* before it can be *Taken*.



Binary Semaphores Used for Synchronization

Using portYIELD_FROM_ISR() after *giving* the semaphore in the ISR, assures that the deferred task is executed immediately after the ISR finishes

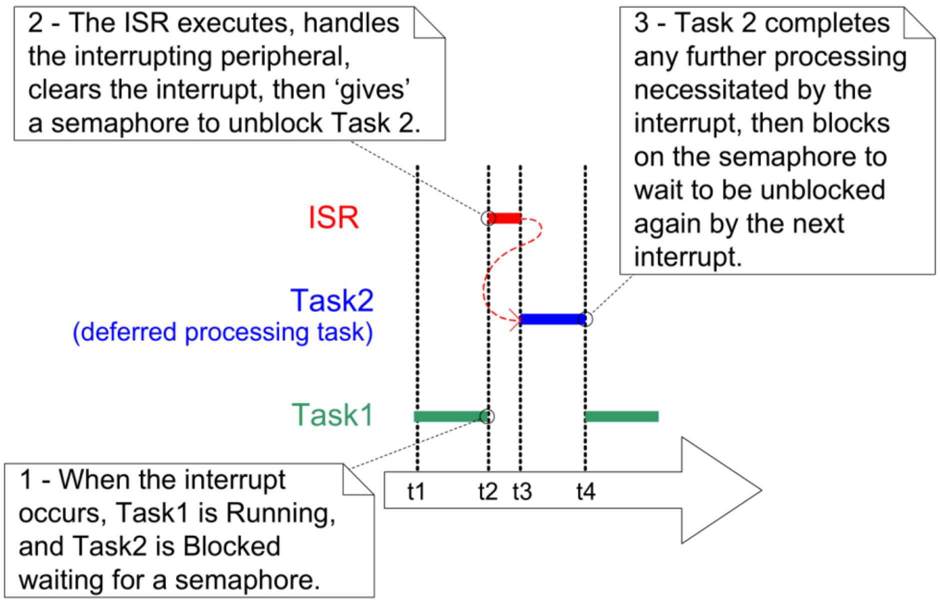


Figure 49. Using a binary semaphore to implement deferred interrupt processing

Binary Semaphores Used for Synchronization

Binary semaphore API functions overview

```
SemaphoreHandle_t xSemaphoreCreateBinary( void );
```

Listing 89. The xSemaphoreCreateBinary() API function prototype

Binary Semaphores Used for Synchronization

Binary semaphore API functions overview

```
BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait );
```

Listing 90. The xSemaphoreTake() API function prototype

```
BaseType_t xSemaphoreGive( SemaphoreHandle_t xSemaphore );
```

Binary Semaphores Used for Synchronization

Binary semaphore API functions overview

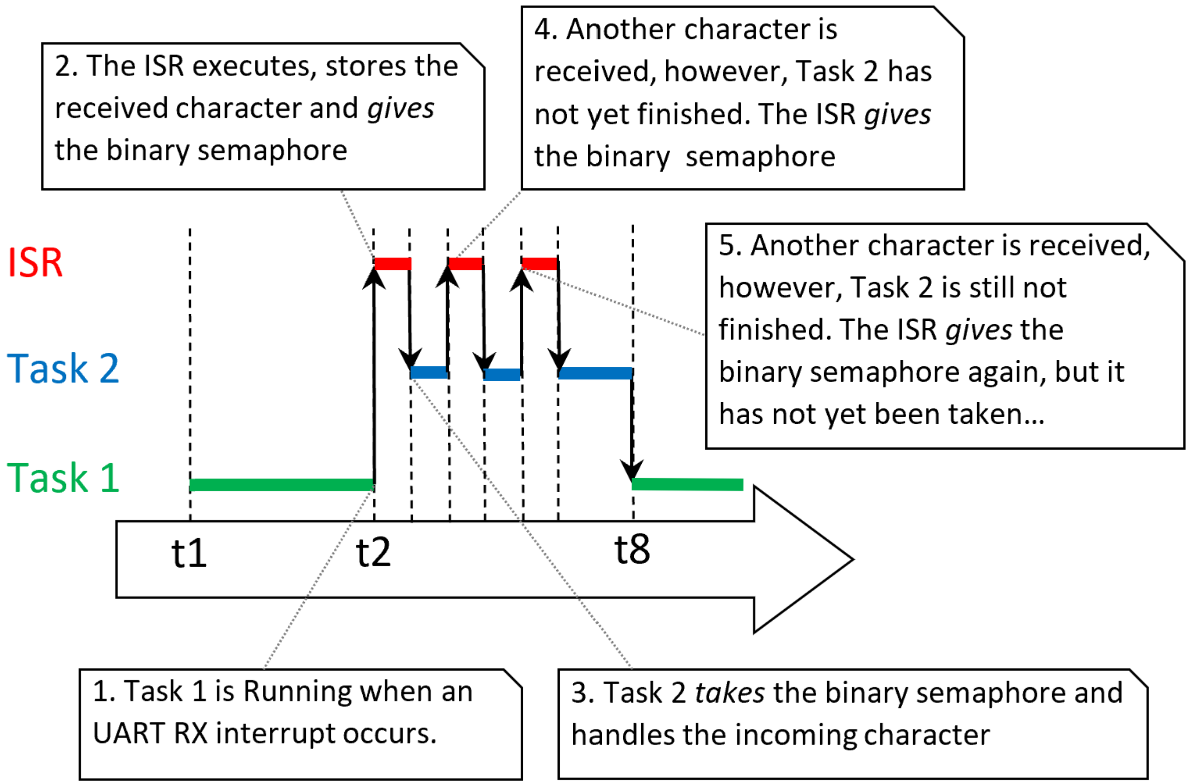
```
BaseType_t xSemaphoreTakeFromISR( SemaphoreHandle_t xSemaphore,
                                  BaseType_t *pxHigherPriorityTaskWoken );
```

```
BaseType_t xSemaphoreGiveFromISR( SemaphoreHandle_t xSemaphore,
                                  BaseType_t *pxHigherPriorityTaskWoken );
```

Listing 91. The xSemaphoreGiveFromISR() API function prototype

Binary Semaphores Used for Synchronization

Don’t use binary semaphores for (relative) high frequency interrupt handling



HAN_UNIVERSITY OF APPLIED SCIENCES

Counting Semaphores

Counts the number of times the semaphore is *given (+1*) and *taken (-1*).

Must be enabled by setting configUSE_COUNTING_SEMAPHORES to 1

Are typically used for:

1. Counting events
   Give a semaphore when an event occurs
   Take a semaphore when an event is processed
2. Resource management
   Take a semaphore when a resource is used
   Give a semaphore when a resource is becomes available

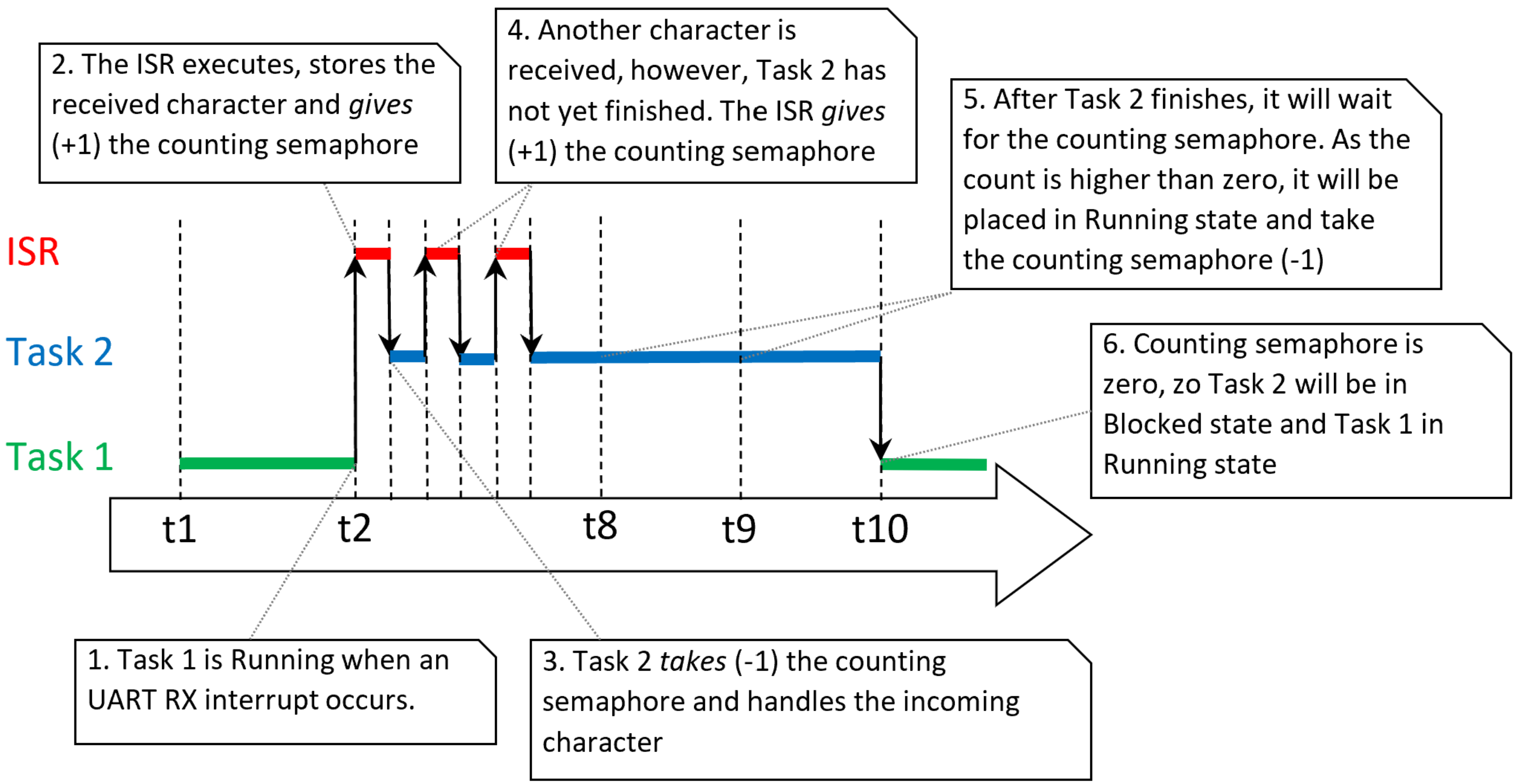
Counting Semaphores

Counting semaphore API functions are equal to the binary semaphore, except for creating.

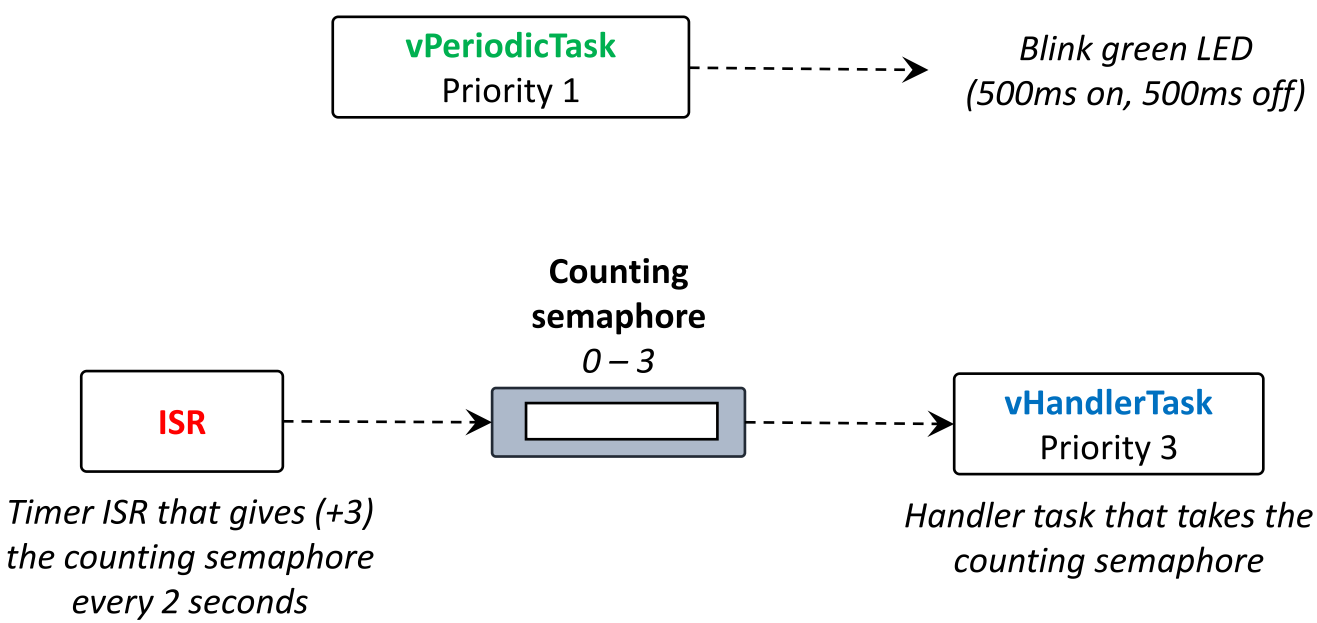
```
SemaphoreHandle_t xSemaphoreCreateCounting( UBaseType_t uxMaxCount,
                                            UBaseType_t uxInitialCount );
```

Listing 97. The xSemaphoreCreateCounting() API function prototype

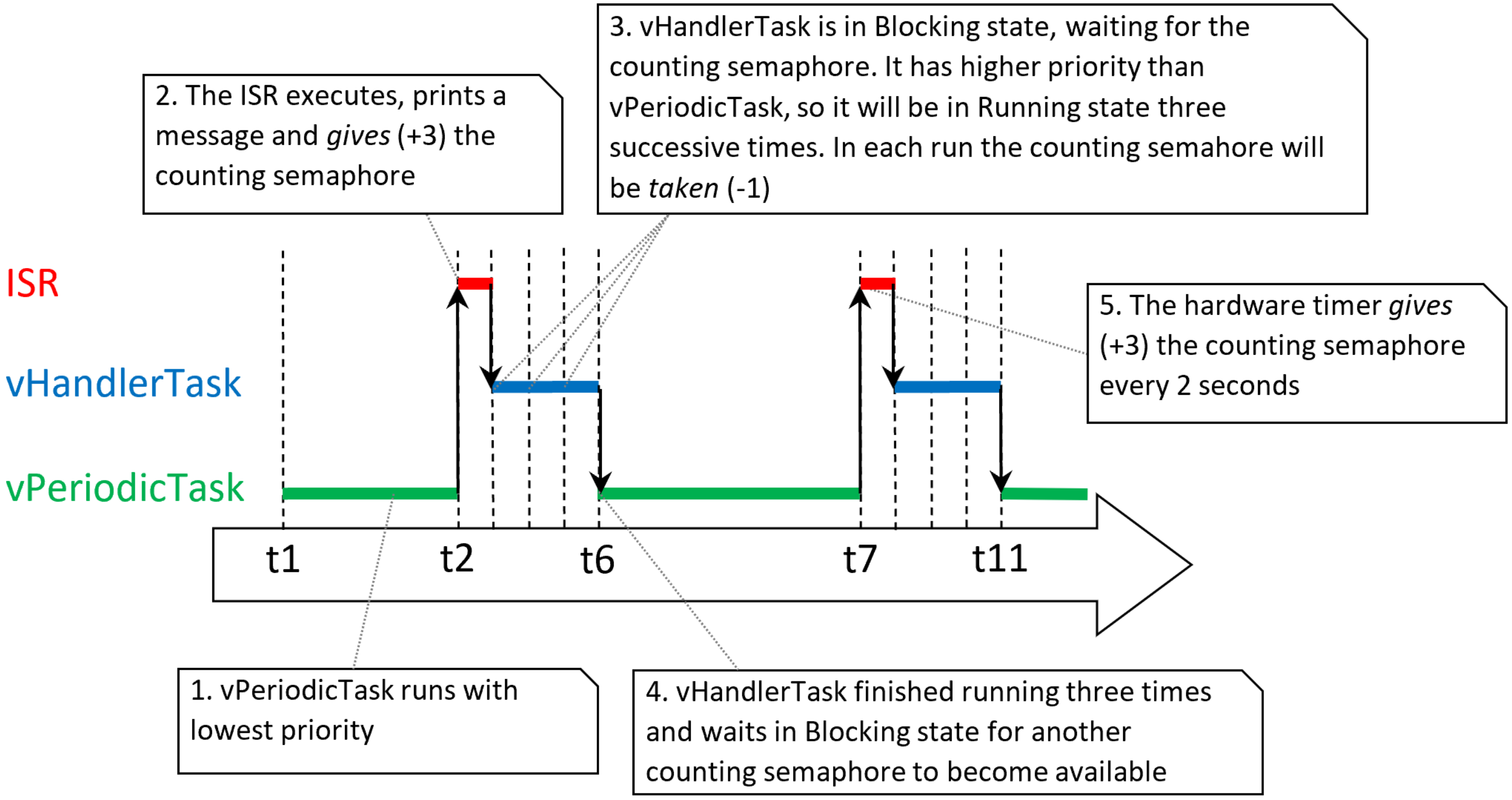
Counting Semaphores



Counting Semaphores Example



Counting Semaphores Example



Counting Semaphores Example

```c
int main(void)
{
    rgb_init();
    xSerialPortInit(921600, 128);

    vSerialPutString("\r\nFRDM-KL25Z FreeRTOS demo Week 4 - Example 01\r\n");
    vSerialPutString("By Hugo Arends\r\n\r\n");

    /* Before a semaphore is used it must be explicitly created. In this example a
    counting semaphore is created. The semaphore is created to have a maximum count
    value of 3, and an initial count value of 0. */
    xCountingSemaphore = xSemaphoreCreateCounting(3, 0);

    /* Check the semaphore was created successfully. */
    if( xCountingSemaphore == NULL )
    {
        /* Error, unable to create the semaphore */
        rgb_red_on(true);
        while(1)
        {;}
    }

    // Semaphore are implemented by using queues that store not data items.
    // We can therefor visualize a sempahore in the kernel aware debugger by
    // registering it as we would do with a queue.
    vQueueAddToRegistry(xCountingSemaphore, "xCountingSemaphore");
```

Counting Semaphores Example

```
    // Create the tasks
    xTaskCreate(vPeriodicTask, "Periodic", configMINIMAL_STACK_SIZE, NULL, 1, NULL );
    xTaskCreate(vHandlerTask,  "Handler",  configMINIMAL_STACK_SIZE, NULL, 3, NULL );

    // Initialize the timer that generates interrupts. An interrupt is generated
    // every 1 ms. Every two seconds, the semaphore will be given - see the ISR
    // TPM1_IRQHandler in timer.c
    tim_init();

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

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks. If main() does reach here then it is likely that
    there was insufficient FreeRTOS heap memory available for the idle task to be
    created. Chapter 2 provides more information on heap memory management. */
    for( ;; );
}
```

Counting Semaphores Example

```c
static void vPeriodicTask( void *pvParameters )
{
    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        // Do not use vTaskDelay(), because that would block this task.
        // Use a for-loop to create a delay instead, which means this task
        // can always be in Running state, except when it is pre-empted by a
        // higher priority task or ISR.

        rgb_green_on(true);
        delay_us(500000);

        rgb_green_on(false);
        delay_us(500000);
    }
}
```

Counting Semaphores Example

```
static void vHandlerTask( void *pvParameters )
{
    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* Use the semaphore to wait for the event. The semaphore was created
        before the scheduler was started, so before this task ran for the first
        time. 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 value returned by xSemaphoreTake(). */
        xSemaphoreTake( xCountingSemaphore, portMAX_DELAY );

        /* To get here the event must have occurred. Process the event (in this
        Case, just print out a message). */
        vSerialPutString( "[Handler task] Processing event\r\n" );
    }
}
```

Counting Semaphores Example

```
void TPM1_IRQHandler(void)
{
    static uint32_t cnt = 0;

    NVIC_ClearPendingIRQ(TPM1_IRQn);

    /* The xHigherPriorityTaskWoken parameter must be initialized to pdFALSE as
    it will get set to pdTRUE inside the interrupt safe API function if a
    context switch is required. */
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    if(TPM1->STATUS & TPM_STATUS_TOF(1))
    {
        // Reset the interrupt flag
        TPM1->STATUS = TPM_STATUS_TOF(1);

        if(++cnt == 2000)
        {
            cnt = 0;

            // In a real application you DO NOT print information in an ISR.
            // This is here now for demonstrating that an interrupt has been generated!
            vSerialPutString( "[ISR handler ] Interrupt generated\r\n" );
```

Counting Semaphores Example

```
            /* 'Give' the semaphore multiple times. The first will unblock the deferred
            interrupt handling task, the following 'gives' are to demonstrate that the
            semaphore latches the events to allow the task to which interrupts are deferred
            to process them in turn, without events getting lost. This simulates multiple
            interrupts being received by the processor, even though in this case the events
            are simulated within a single interrupt occurrence. */
            xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );
            xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );
            xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );

            /* Pass the xHigherPriorityTaskWoken value into portYIELD_FROM_ISR(). If
            xHigherPriorityTaskWoken was set to pdTRUE inside xSemaphoreGiveFromISR()
            then calling portYIELD_FROM_ISR() will request a context switch. If
            xHigherPriorityTaskWoken is still pdFALSE then calling
            portYIELD_FROM_ISR() will have no effect. */
            portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
        }
    }
}
```

Deferring Work to the RTOS Daemon Task

Defer work to the daemon task

This is called *centralized deferred interrupt processing*

Advantages

- Lower resource usage – no separate task required for each deferred interrupt
- Simplified user model – the deferred interrupt handling is a standard C function

Disadvantages

- Less flexibility – priority cannot be set separately
- Less determinism – processing depends on other availability of other commands in the daemon command queue

Deferring Work to the RTOS Daemon Task

```
BaseType_t xTimerPendFunctionCallFromISR( PendedFunction_t xFunctionToPend,
                                          void *pvParameter1,
                                          uint32_t ulParameter2,
                                          BaseType_t *pxHigherPriorityTaskWoken );
```

Listing 100. The xTimerPendFunctionCallFromISR() API function prototype

A pointer to the function that will be executed in the daemon task (in effect, just the function name). The function prototype must conform to

```
void vPendableFunction( void *pvParameter1, uint32_t ulParameter2 );
```

Listing 101. The prototype to which a function passed in the xFunctionToPend parameter of xTimerPendFunctionCallFromISR() must conform

Deferring Work to the RTOS Daemon Task

```
BaseType_t xTimerPendFunctionCallFromISR( PendedFunction_t xFunctionToPend,
                                          void *pvParameter1,
                                          uint32_t ulParameter2,
                                          BaseType_t *pxHigherPriorityTaskWoken );
```

Listing 100. The xTimerPendFunctionCallFromISR() API function prototype

xTimerPendFunctionCallFromISR() writes to the timer command queue. If the RTOS daemon task was in the Blocked state to wait for data to become available on the timer command queue, then writing to the timer command queue will cause the daemon task to leave the Blocked state. If the priority of the daemon task is higher than the priority of the currently executing task (the task that was interrupted), then, internally, xTimerPendFunctionCallFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE.

Deferring Work to the RTOS Daemon Task

```
BaseType_t xTimerPendFunctionCallFromISR( PendedFunction_t xFunctionToPend,
                                          void *pvParameter1,
                                          uint32_t ulParameter2,
                                          BaseType_t *pxHigherPriorityTaskWoken );
```

Listing 100. The xTimerPendFunctionCallFromISR() API function prototype

pdPASS: if the ‘execute function’ command was written to the timer command queue

pdFAIL: if the ‘execute function’ command could not be written to the timer command queue because the timer command queue was already full

HAN_UNIVERSITY OF APPLIED SCIENCES

Using Queues within an Interrupt Service Routine

Binary and counting semaphores are used to communicate events.

Queues are used to communicate events, **and to transfer data**.

Use the interrupt safe queue API functions. These are functionally equivalent to the non-interrupt safe functions discussed last week.

Queues provide an easy and convenient way of passing data from an interrupt to a task, but it is **not efficient** to use a queue if data is arriving at a high frequency. Instead, consider using

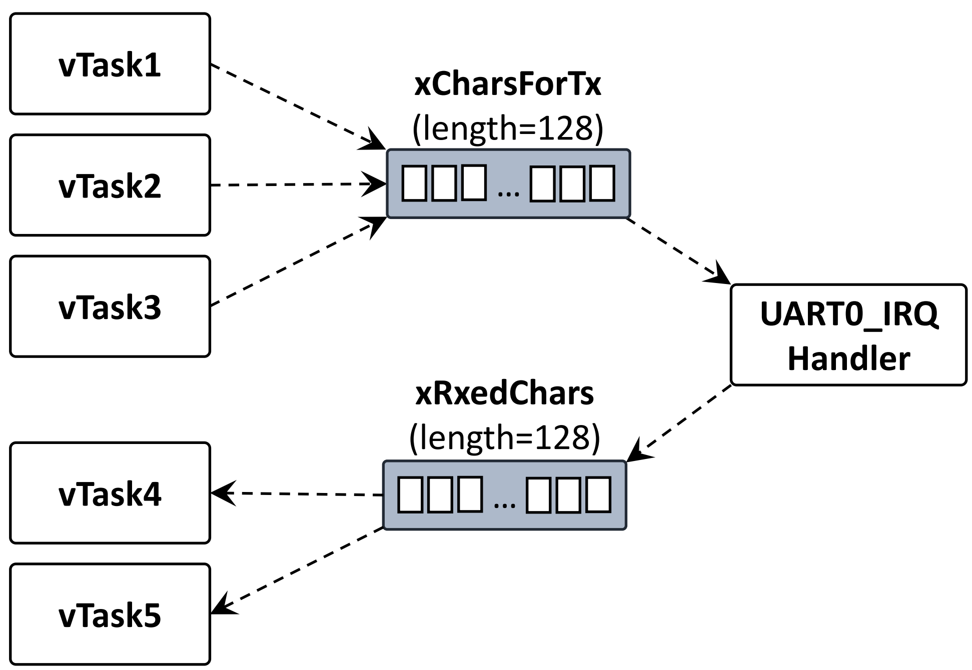
- DMA
- A thread safe RAM buffer
- Direct processing in the ISR

Using Queues within an Interrupt Service Routine

Example – UART communication, see *serial.c* and *serial.h* in any of the provided projects so far

What would be shown in the serial monitor if one task pre-empts the other?
We must prevent one task from being pre-empted by another task when writing to the queue!

The highest priority task will receive characters from the queue



1.
As long as the xCharsForTx queue holds characters, the ISR will send them

2.
Incoming characters are send to the xRxedChars queue

Summary

- Which FreeRTOS API functions can be used from within an interrupt service routine.
- Methods of deferring interrupt processing to a task.
- How to create and use binary semaphores and counting semaphores.
- The differences between binary and counting semaphores.
- How to use a queue to pass data into and out of an interrupt service routine.
- The kernel interrupt priority and interrupt nesting.