**INTERRUPT MANAGEMENT**

**INDEX:**

[**1. Introduction 3**](#_heading=h.v0w1kypuir9o)

[**2. FreeRTOS APIs from an ISR 3**](#_heading=h.nijapf4tcdv)

[2.1 Benefits of Using a Separate Interrupt Safe API 3](#_heading=h.wftdtmy1jag2)

[2.2 Disadvantages of Using a Separate Interrupt Safe API 4](#_heading=h.n99xb3ntq0tr)

[**3. xHigherPriorityTaskWoken Parameter 4**](#_heading=h.rcixif9v5xo2)

[3.1 If the API function was called from a task 5](#_heading=h.20ayieksjds)

[3.2 If the API function was called from an interrupt 5](#_heading=h.fmffrft5zl3k)

[**4. portYIELD\_FROM\_ISR() and portEND\_SWITCHING\_ISR() Macros 6**](#_heading=h.pznbk6bmhsh3)

[**5. Deferred Interrupt Processing 7**](#_heading=h.61ft2rj4si7q)

[**6. Binary Semaphores used for synchronization 9**](#_heading=h.1kuhn6mdtil9)

[**7. Binary semaphore APIs 11**](#_heading=h.c7gzinmh4kml)

[7.1 xSemaphoreCreateBinary() 11](#_heading=h.kg3z95ie7jfd)

[7.2 xSemaphoreTake() 12](#_heading=h.cv0jzhm536kp)

[7.3 xSemaphoreGiveFromISR() 13](#_heading=h.yolhy14d9tmf)

[**8. Using a binary semaphore to synchronize a task with an interrupt 15**](#_heading=h.60q7q7tw6zu4)

[**9. Counting Semaphores 21**](#_heading=h.2qgh451qs9rw)

[**10. Counting Semaphore APIs 23**](#_heading=h.iz1liuqsw20o)

[10.1 xSemaphoreCreateCounting() 23](#_heading=h.3eoc22ed2dhq)

[**11. Using a counting semaphore to synchronize a task with an interrupt 24**](#_heading=h.puizvdceb5tx)

[**12. Deferring Work to the RTOS Daemon Task 26**](#_heading=h.dvoc9e5lz7vc)

[12.1 xTimerPendFunctionCallFromISR() 26](#_heading=h.kiiow5fh9ovy)

[12.2 Example : Centralized deferred interrupt processing 28](#_heading=h.t4f38k738s4d)

[**13. Using Queues within an Interrupt Service Routine 31**](#_heading=h.1vr9yooyy1xd)

[13.1 xQueueSendToFrontFromISR() 31](#_heading=h.1xzsqe7wurdp)

[13.2 xQueueSendToBackFromISR() 32](#_heading=h.m7pe3s2vas77)

[13.3 Example : Sending and receiving on a queue from within an interrupt 33](#_heading=h.y49fwbhyat3p)

[**14. Interrupt Nesting 37**](#_heading=h.cej9easae9ib)

[**15. A Note to ARM Cortex-M1 and ARM GIC Users 40**](#_heading=h.c6vh03psnkta)

[**16. Example Codes 41**](#_heading=h.c9io7vp9z2kk)

[16.1 Use of Binary Semaphores 41](#_heading=h.s81m9c7difqv)

[16.2 Use of xSemaphoreGiveFromISR() 43](#_heading=h.cjga6s72pjpl)

[16.3 Binary semaphore with multiple interrupts 46](#_heading=h.b42foic6mlmv)

[16.4 Use of counting semaphore 49](#_heading=h.bvucaknfmqw4)

[16.5 Deferred Daemon task 52](#_heading=h.riboxbme9ze)

[16.6 Use of queues from ISR 54](#_heading=h.j4rvw1iz3mc5)

# Introduction

It is important to draw a distinction between the priority of a task, and the priority of an interrupt:

∙ A task is a software feature that is unrelated to the hardware on which FreeRTOS is running. The priority of a task is assigned in software by the application writer, and a software algorithm (the scheduler) decides which task will be in the Running state.

∙ Although written in software, an interrupt service routine is a hardware feature because the hardware controls which interrupt service routine will run, and when it will run. Tasks will only run when there are no ISRs running, so the lowest priority interrupt will interrupt the highest priority task, and there is no way for a task to pre-empt an ISR.

All architectures on which FreeRTOS will run are capable of processing interrupts, but details relating to interrupt entry, and interrupt priority assignment, vary between architectures.

# FreeRTOS APIs from an ISR

Often it is necessary to use the functionality provided by a FreeRTOS API function from an interrupt service routine (ISR), but many FreeRTOS API functions perform actions that are not valid inside an ISR—the most notable of which is placing the task that called the API function into the Blocked state; if an API function is called from an ISR, then it is not being called from a task, so there is no calling task that can be placed into the Blocked state.

FreeRTOS solves this problem by providing two versions of some API functions; one version for use from tasks, and one version for use from ISRs. Functions intended for use from ISRs have **“FromISR”** appended to their name

## 2.1 Benefits of Using a Separate Interrupt Safe API

If the same version of an API function could be called from both a task and an ISR then:

1. The API functions would need additional logic to determine if they had been called from a task or an ISR. The additional logic would introduce new paths through the function, making the functions longer, more complex, and harder to test
2. Some API function parameters would be obsolete when the function was called from a task, while others would be obsolete when the function was called from an ISR.
3. Each FreeRTOS port would need to provide a mechanism for determining the execution context (task or ISR).
4. Architectures on which it is not easy to determine the execution context (task or ISR) would require additional, wasteful, more complex to use, and non-standard interrupt entry code that allowed the execution context to be provided by software

## 2.2 Disadvantages of Using a Separate Interrupt Safe API

Having two versions of some API functions allows both tasks and ISRs to be more efficient, but introduces a new problem; sometimes it is necessary to call a function that is not part of the FreeRTOS API, but makes use of the FreeRTOS API, from both a task and an ISR.

This is normally only a problem when integrating third party code, as that is the only time when the software’s design is out of the control of the application writer. If this does become an issue then the problem can be overcome using one of the following techniques:

1. Defer interrupt processing to a task1 , so the API function is only ever called from the context of a task.

2. If you are using a FreeRTOS port that supports interrupt nesting, then use the version of the API function that ends in **“FromISR”,** as that version can be called from tasks and ISRs (the reverse is not true, API functions that do not end in **“FromISR”** must not be called from an ISR).

3. Third party code normally includes an RTOS abstraction layer that can be implemented to test the context from which the function is being called (task or interrupt), and then call the API function that is appropriate for the context.

# 3. xHigherPriorityTaskWoken Parameter

* If a context switch is performed by an interrupt, then the task running when the interrupt exits might be different to the task that was running when the interrupt was entered—the interrupt will have interrupted one task, but returned to a different task.
* Some FreeRTOS API functions can move a task from the Blocked state to the Ready state. This has already been seen with functions such as xQueueSendToBack(), which will unblock a task if there was a task waiting in the Blocked state for data to become available on the subject queue
* If the priority of a task that is unblocked by a FreeRTOS API function is higher than the priority of the task in the Running state then, in accordance with the FreeRTOS scheduling policy, a switch to the higher priority task should occur. When the switch to the higher priority task actually occurs is dependent on the context from which the API function is called:

## 3.1 **If the API function was called from a task**

If configUSE\_PREEMPTION is set to 1 in FreeRTOSConfig.h then the switch to the higher priority task occurs automatically within the API function—so before the API function has exited., where writing to the timer command queue resulted in a switch to the RTOS daemon task before the function that wrote to the command queue had exited.

## 3.2 **If the API function was called from an interrupt**

* A switch to a higher priority task will not occur automatically inside an interrupt. Instead, a variable is set to inform the application writer that a context switch should be performed. Interrupt safe API functions (those that end in “FromISR”) have a pointer parameter called pxHigherPriorityTaskWoken that is used for this purpose.
* If a context switch should be performed, then the interrupt safe API function will set \*pxHigherPriorityTaskWoken to pdTRUE. To be able to detect this has happened, the variable pointed to by pxHigherPriorityTaskWoken must be initialized to pdFALSE before it is used for the first time.
* If the application writer opts not to request a context switch from the ISR, then the higher priority task will remain in the Ready state until the next time the scheduler runs—which in the worst case will be during the next tick interrupt.
* FreeRTOS API functions can only set \*pxHighPriorityTaskWoken to pdTRUE. If an ISR calls more than one FreeRTOS API function, then the same variable can be passed as the pxHigherPriorityTaskWoken parameter in each API function call, and the variable only needs to be initialized to pdFALSE before it is used for the first time.
* There are several reasons why context switches do not occur automatically inside the interrupt safe version of an API function:

1. **Avoiding unnecessary context switches**

An interrupt may execute more than once before it is necessary for a task to perform any processing. For example, consider a scenario where a task processes a string that was received by an interrupt driven UART; it would be wasteful for the UART ISR to switch to the task each time a character was received because the task would only have processing to perform after the complete string had been received.

**2. Control over the execution sequence**

Interrupts can occur sporadically, and at unpredictable times. Expert FreeRTOS users may want to temporarily avoid an unpredictable switch to a different task at specific points in their application—although this can also be achieved using the FreeRTOS scheduler locking mechanism.

1. **Portability**

It is the simplest mechanism that can be used across all FreeRTOS ports.

1. **Efficiency**

Ports that target smaller processor architectures only allow a context switch to be requested at the very end of an ISR, and removing that restriction would require additional and more complex code. It also allows more than one call to a FreeRTOS API function within the same ISR without generating more than one request for a context switch within the same ISR.

1. **Execution in the RTOS tick interrupt**

it is possible to add application code into the RTOS tick interrupt. The result of attempting a context switch inside the tick interrupt is dependent on the FreeRTOS port in use. At best, it will result in an unnecessary call to the scheduler.

Use of the pxHigherPriorityTaskWoken parameter is optional. If it is not required, then set pxHigherPriorityTaskWoken to NULL.

# 4. portYIELD\_FROM\_ISR() and portEND\_SWITCHING\_ISR() Macros

* the macros that are used to request a context switch from an ISR
* taskYIELD() is a macro that can be called in a task to request a context switch.
* portYIELD\_FROM\_ISR() and portEND\_SWITCHING\_ISR() are both interrupt safe versions of taskYIELD(). portYIELD\_FROM\_ISR() and portEND\_SWITCHING\_ISR() are both used in the same way, and do the same thing .
* Some FreeRTOS ports only provide one of the two macros. Newer FreeRTOS ports provide both macros.

**portEND\_SWITCHING\_ISR( xHigherPriorityTaskWoken );**

**portYIELD\_FROM\_ISR( xHigherPriorityTaskWoken );**

* The xHigherPriorityTaskWoken parameter passed out of an interrupt safe API function can be used directly as the parameter in a call to portYIELD\_FROM\_ISR().
* If the portYIELD\_FROM\_ISR() xHigherPriorityTaskWoken parameter is pdFALSE (zero), then a context switch is not requested, and the macro has no effect. If the portYIELD\_FROM\_ISR() xHigherPriorityTaskWoken parameter is not pdFALSE, then a context switch is requested, and the task in the Running state might change. The interrupt will always return to the task in the Running state, even if the task in the Running state changed while the interrupt was executing.
* Most FreeRTOS ports allow portYIELD\_FROM\_ISR() to be called anywhere within an ISR. A few FreeRTOS ports (predominantly those for smaller architectures), only allow portYIELD\_FROM\_ISR() to be called at the very end of an ISR.

# 5. Deferred Interrupt Processing

It is normally considered best practice to keep ISRs as short as possible. Reasons for this include:

∙ Even if tasks have been assigned a very high priority, they will only run if no interrupts are being serviced by the hardware.

∙ ISRs can disrupt both the start time, and the execution time, of a task.

∙ Depending on the architecture on which FreeRTOS is running, it might not be possible to accept any new interrupts, or at least a subset of new interrupts, while an ISR is executing.

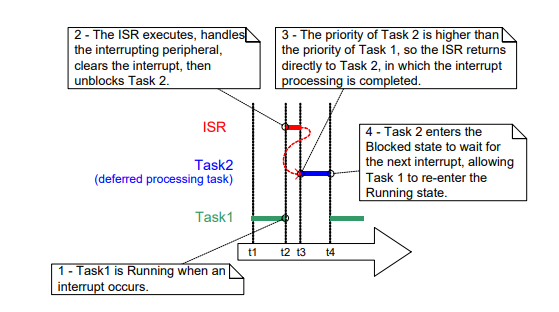
∙ The application writer needs to consider the consequences of, and guard against, resources such as variables, peripherals, and memory buffers being accessed by a task and an ISR at the same time.

∙ Some FreeRTOS ports allow interrupts to nest, but interrupt nesting can increase complexity and reduce predictability. The shorter an interrupt is, the less likely it is to nest.

An interrupt service routine must record the cause of the interrupt, and clear the interrupt. Any other processing necessitated by the interrupt can often be performed in a task, allowing the interrupt service routine to exit as quickly as is practical. This is called ‘deferred interrupt processing’, because the processing necessitated by the interrupt is ‘deferred’ from the ISR to a task.

Deferring interrupt processing to a task also allows the application writer to prioritize the processing relative to other tasks in the application, and use all the FreeRTOS API functions.

If the priority of the task to which interrupt processing is deferred is above the priority of any other task, then the processing will be performed immediately, just as if the processing had been performed in the ISR itself



In above figure interrupt processing starts at time t2, and effectively ends at time t4, but only the period between times t2 and t3 is spent in the ISR. If deferred interrupt processing had not been used then the entire period between times t2 and t4 would have been spent in the ISR.

There is no absolute rule as to when it is best to perform all processing necessitated by an interrupt in the ISR, and when it is best to defer part of the processing to a task. Deferring processing to a task is most useful when:

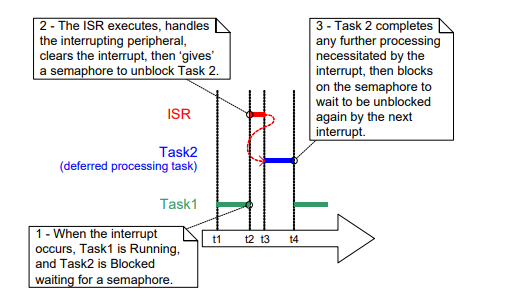
∙ The processing necessitated by the interrupt is not trivial. For example, if the interrupt is just storing the result of an analog to digital conversion, then it is almost certain this is best performed inside the ISR, but if result of the conversion must also be passed through a software filter, then it may be best to execute the filter in a task.

∙ It is convenient for the interrupt processing to perform an action that cannot be performed inside an ISR, such as write to a console, or allocate memory.

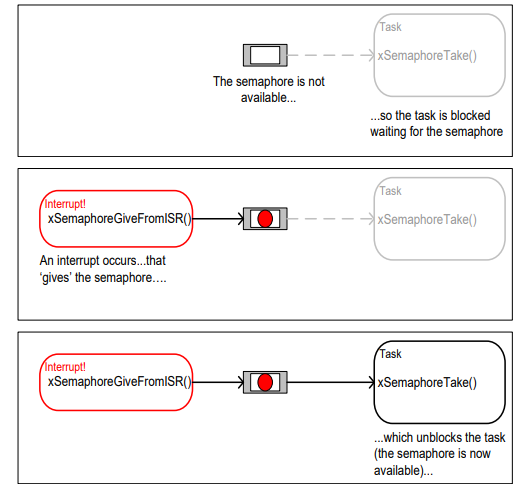
∙ The interrupt processing is not deterministic—meaning it is not known in advance how long the processing will take.

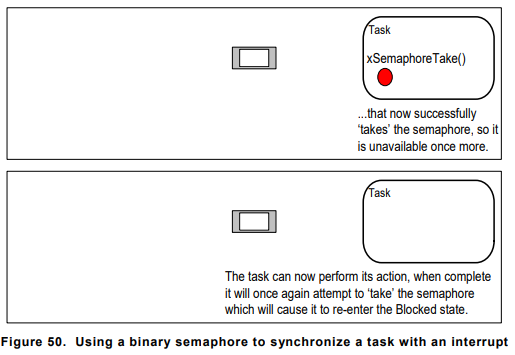
# Binary Semaphores used for synchronization

* The interrupt safe version of the Binary Semaphore API can be used to unblock a task each time a particular interrupt occurs, effectively synchronizing the task with the interrupt. This allows the majority of the interrupt event processing to be implemented within the synchronized task, with only a very fast and short portion remaining directly in the ISR
* If the interrupt processing is particularly time critical, then the priority of the deferred processing task can be set to ensure the task always preempts the other tasks in the system. The ISR can then be implemented to include a call to portYIELD\_FROM\_ISR(), ensuring the ISR returns directly to the task to which interrupt processing is being deferred. This has the effect of ensuring the entire event processing executes contiguously (without a break) in time, just as if it had all been implemented within the ISR itself



* The deferred processing 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 task to which interrupt processing is deferred 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 task to exit the Blocked state and remove the token, leaving the queue empty once more. When the 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.





# Binary semaphore APIs

## 7.1 xSemaphoreCreateBinary()

* Handles to all the various types of FreeRTOS semaphore are stored in a variable of type SemaphoreHandle\_t.
* Before a semaphore can be used, it must be created. To create a binary semaphore, use the xSemaphoreCreateBinary() API function

**SemaphoreHandle\_t xSemaphoreCreateBinary( void );**

* **Returned value :**
* If NULL is returned, then the semaphore cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the semaphore data structures.
* A non-NULL value being returned indicates that the semaphore has been created successfully. The returned value should be stored as the handle to the created semaphore.

## 7.2 xSemaphoreTake()

* ‘Taking’ a semaphore means to ‘obtain’ or ‘receive’ the semaphore. The semaphore can be taken only if it is available.
* All the various types of FreeRTOS semaphore, except recursive mutexes, can be ‘taken’ using the xSemaphoreTake() function.
* xSemaphoreTake() must not be used from an interrupt service routine.

**BaseType\_t xSemaphoreTake( SemaphoreHandle\_t xSemaphore, TickType\_t xTicksToWait );**

| **Parameter Name/ Returned Value** | **Description** |
| --- | --- |
| xSemaphore | * A semaphore is referenced by a variable of type SemaphoreHandle\_t. It must be explicitly created before it can be used. |
| xTicksToWait | * The maximum amount of time the task should remain in the Blocked state to wait for the semaphore if it is not already available. * If xTicksToWait is zero, then xSemaphoreTake() will return immediately if the semaphore is not available. * The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS\_TO\_TICKS() can be used to convert a time specified in milliseconds to a time specified in ticks. * Setting xTicksToWait to portMAX\_DELAY will cause the task to wait indefinitely (without a timeout) if INCLUDE\_vTaskSuspend is set to 1 in FreeRTOSConfig.h. |
| Returned value | There are two possible return values:   1. **pdPASS**   pdPASS is returned only if the call to xSemaphoreTake() was successful in obtaining the semaphore. If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state to wait for the semaphore if it was not immediately available, but the semaphore became available before the block time expired.   1. **pdFALSE**   The semaphore is not available. If a block time was specified (xTicksToWait was not zero), then the calling task will have been placed into the Blocked state to wait for the semaphore to become available, but the block time expired before this happened. |

## 7.3 xSemaphoreGiveFromISR()

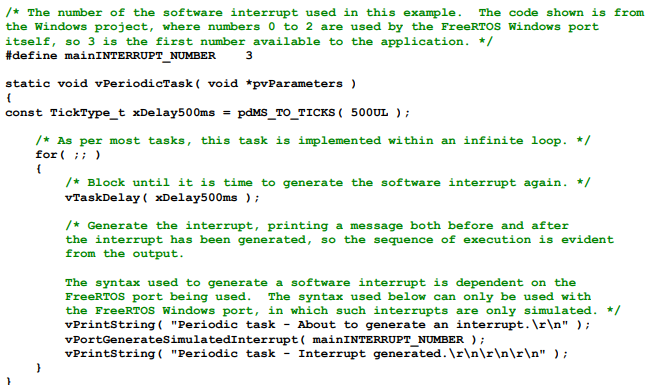
* Binary and counting semaphores can be ‘given’ using the xSemaphoreGiveFromISR() function.
* xSemaphoreGiveFromISR() is the interrupt safe version of xSemaphoreGive(), so has the pxHigherPriorityTaskWoken parameter

**BaseType\_t xSemaphoreGiveFromISR( SemaphoreHandle\_t xSemaphore, BaseType\_t \*pxHigherPriorityTaskWoken );**

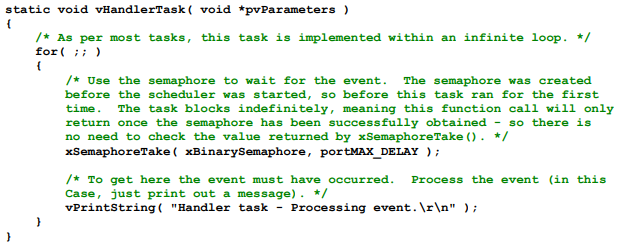
| **Parameter Name/ Returned Value** | **Description** |
| --- | --- |
| xSemaphore | * A semaphore is referenced by a variable of type SemaphoreHandle\_t. It must be explicitly created before it can be used. |
| pxHigherPriorityTaskWoken | * It is possible that a single semaphore will have one or more tasks blocked on it waiting for the semaphore to become available. Calling xSemaphoreGiveFromISR() can make the semaphore available, and so cause a task that was waiting for the semaphore to leave the Blocked state. If calling xSemaphoreGiveFromISR() causes a task to leave the Blocked state, and the unblocked task has a priority higher than the currently executing task (the task that was interrupted), then, internally, xSemaphoreGiveFromISR() will set \*pxHigherPriorityTaskWoken to pdTRUE. * If xSemaphoreGiveFromISR() sets this value to pdTRUE, then normally a context switch should be performed before the interrupt is exited. This will ensure that the interrupt returns directly to the highest priority Ready state task. |
| Returned value | * There are two possible return values:  1. **pdPASS**   pdPASS will be returned only if the call to xSemaphoreGiveFromISR() is successful.   1. **pdFAIL**   If a semaphore is already available, it cannot be given, and xSemaphoreGiveFromISR() will return pdFAIL state to wait for the semaphore to become available, but the block time expired before this happened. |

# 8. Using a binary semaphore to synchronize a task with an interrupt

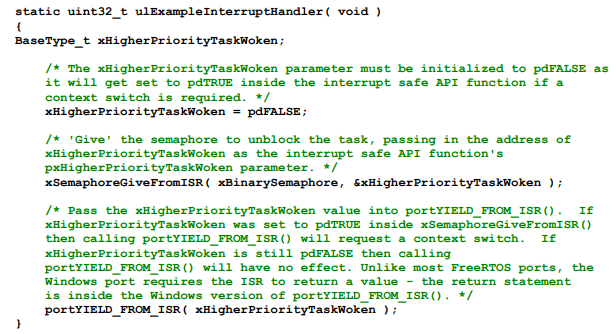
* This example uses a binary semaphore to unblock a task from an interrupt service routine— effectively synchronizing the task with the interrupt.



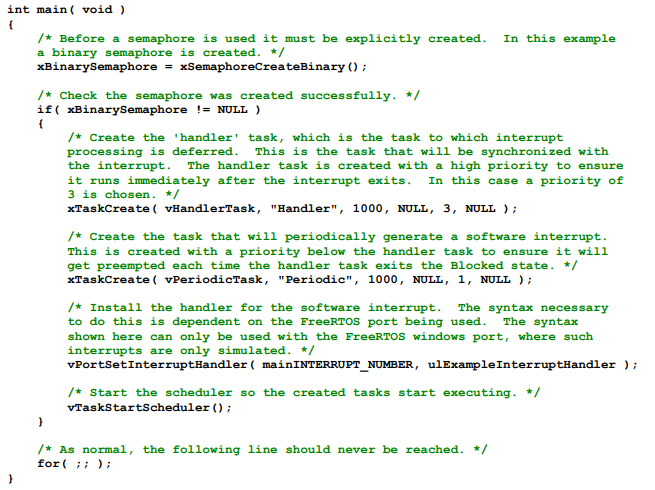
* A simple periodic task is used to generate a software interrupt every 500 milliseconds. A software interrupt is used for convenience because of the complexity of hooking into a real interrupt in some target environments
* Note that the task prints out a string both before and after the interrupt is generated. This allows the sequence of execution to be observed in the output produced when the example is executed



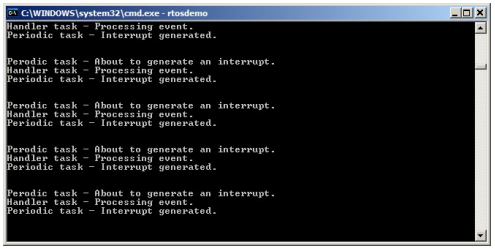
* The implementation of the task to which the interrupt processing is deferred— the task that is synchronized with the software interrupt through the use of a binary semaphore

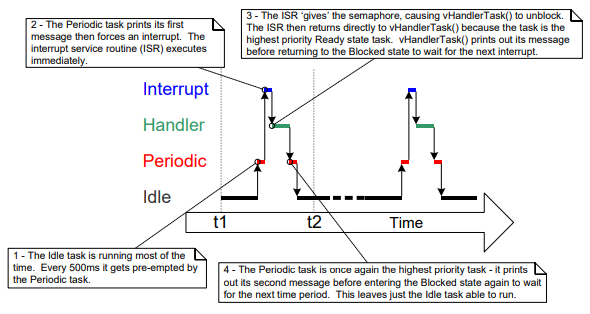


* This does very little other than ‘give’ the semaphore to unblock the task to which interrupt processing is deferred.
* Note how the xHigherPriorityTaskWoken variable is used. It is set to pdFALSE before calling xSemaphoreGiveFromISR(), then used as the parameter when portYIELD\_FROM\_ISR() is called. A context switch will be requested inside the portYIELD\_FROM\_ISR() macro if xHigherPriorityTaskWoken equals pdTRUE.



* As expected, vHandlerTask() enters the Running state as soon as the interrupt is generated, so the output from the task splits the output produced by the periodic task





**Improving the Implementation of the Task Used Above**

The execution sequence was as follows:

1. The interrupt occurred.

2. The ISR executed and ‘gave’ the semaphore to unblock the task.

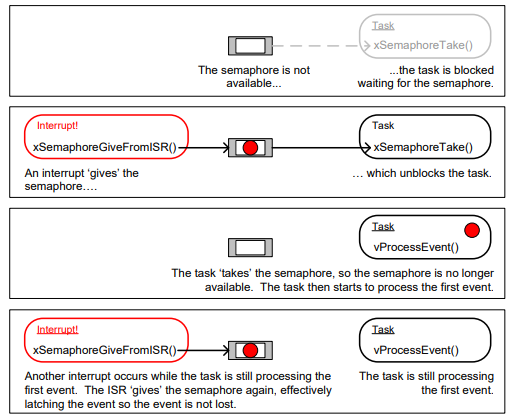
3. The task executed immediately after the ISR, and ‘took’ the semaphore.

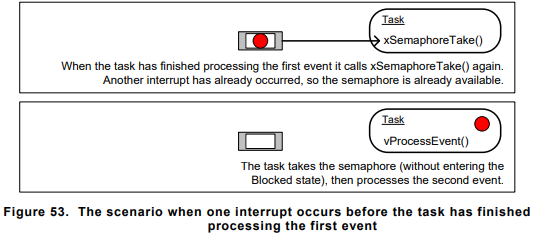
4. The task processed the event, then attempted to ‘take’ the semaphore again—entering the Blocked state because the semaphore was not yet available (another interrupt had not yet occurred).

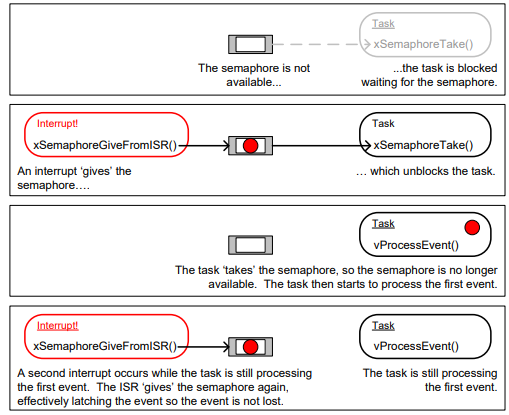
The structure of the task used in above example is adequate only if interrupts occur at a relatively low frequency. To understand why, consider what would happen if a second, and then a third, interrupt had occurred before the task had completed its processing of the first interrupt:

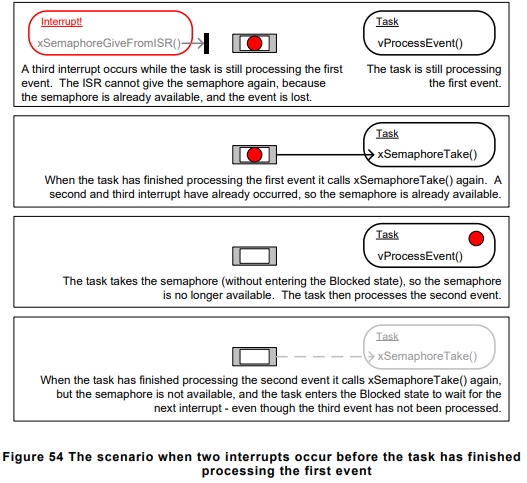
∙ When the second ISR executed the semaphore would be empty, so the ISR would give the semaphore, and the task would process the second event immediately after it had completed processing the first event.

∙ When the third ISR executed, the semaphore would already be available, preventing the ISR giving the semaphore again, so the task would not know the third event had occurred.









* The deferred interrupt handling task used in Example 16, and shown in Listing 93, is structured so that it only processes one event between each call to xSemaphoreTake(). That was adequate for above example, because the interrupts that generated the events were triggered by software, and occurred at a predictable time.
* In real applications, interrupts are generated by hardware, and occur at unpredictable times. Therefore, to minimize the chance of an interrupt being missed, the deferred interrupt handling task must be structured so that it processes all the events that are already available between each call to xSemaphoreTake() .
* The deferred interrupt handling task used in Example had one other weakness; it did not use a time out when it called xSemaphoreTake(). Instead, the task passed portMAX\_DELAY as the xSemaphoreTake() xTicksToWait parameter, which results in the task waiting indefinitely (without a time out) for the semaphore to be available.
* Indefinite timeouts are often used in example code because their use simplifies the structure of the example, and therefore makes the example easier to understand. However, indefinite timeouts are normally bad practice in real applications, because they make it difficult to recover from an error.
* As an example, consider the scenario where a task is waiting for an interrupt to give a semaphore, but an error state in the hardware is preventing the interrupt from being generated:
* If the task is waiting without a time out, it will not know about the error state, and will wait forever.
* If the task is waiting with a time out, then xSemaphoreTake() will return pdFAIL when the time out expires, and the task can then detect and clear the error the next time it executes

# 9. Counting Semaphores

* Just as binary semaphores can be thought of as queues that have a length of one, counting semaphores can be thought of as queues that have a length of more than one.
* Tasks are not interested in the data that is stored in the queue—just the number of items in the queue. **configUSE\_COUNTING\_SEMAPHORES** must be set to 1 in FreeRTOSConfig.h for counting semaphores to be available.
* Each time a counting semaphore is ‘given’, another space in its queue is used. The number of items in the queue is the semaphore’s ‘count’ value.
* Counting semaphores are typically used for two things:

1. **Counting events**

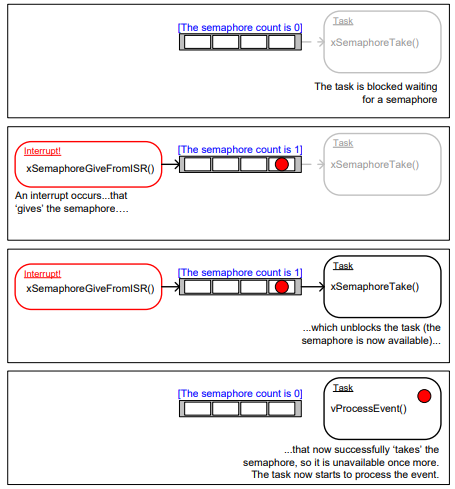
In this scenario, an event handler will ‘give’ a semaphore each time an event occurs— causing the semaphore’s count value to be incremented on each ‘give’. A task will ‘take’ a semaphore each time it processes an event—causing the semaphore’s count value to be decremented on each ‘take’. The count value is the difference between the number of events that have occurred and the number that have been processed.

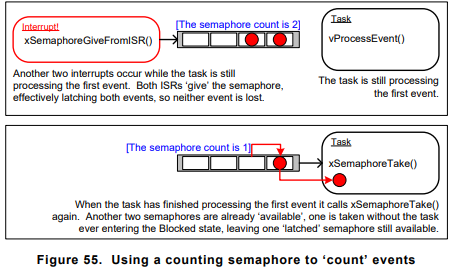
Counting semaphores that are used to count events are created with an initial count value of zero.

1. **Resource management.**

In this scenario, the count value indicates the number of resources available. To obtain control of a resource, a task must first obtain a semaphore—decrementing the semaphore’s count value. When the count value reaches zero, there are no free resources. When a task finishes with the resource, it ‘gives’ the semaphore back— incrementing the semaphore’s count value.

Counting semaphores that are used to manage resources are created so that their initial count value equals the number of resources that are available





# 10. Counting Semaphore APIs

## 10.1 xSemaphoreCreateCounting()

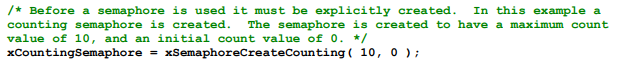
* Before a semaphore can be used, it must be created. To create a counting semaphore, use the xSemaphoreCreateCounting() API function.

**SemaphoreHandle\_t xSemaphoreCreateCounting( UBaseType\_t uxMaxCount, UBaseType\_t uxInitialCount );**

| **Parameter Name/ Returned Value** | **Description** |
| --- | --- |
| uxMaxCount | * The maximum value to which the semaphore will count. To continue the queue analogy, the uxMaxCount value is effectively the length of the queue. * When the semaphore is to be used to count or latch events, uxMaxCount is the maximum number of events that can be latched. * When the semaphore is to be used to manage access to a collection of resources, uxMaxCount should be set to the total number of resources that are available |
| uxInitialCount | * The initial count value of the semaphore after it has been created. * When the semaphore is to be used to count or latch events, uxInitialCount should be set to zero—as, presumably, when the semaphore is created, no events have yet occurred. * When the semaphore is to be used to manage access to a collection of resources, uxInitialCount should be set to equal uxMaxCount—as, presumably, when the semaphore is created, all the resources are available. |
| Returned value | * If NULL is returned, the semaphore cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the semaphore data structures. * A non-NULL value being returned indicates that the semaphore has been created successfully. The returned value should be stored as the handle to the created semaphore. |

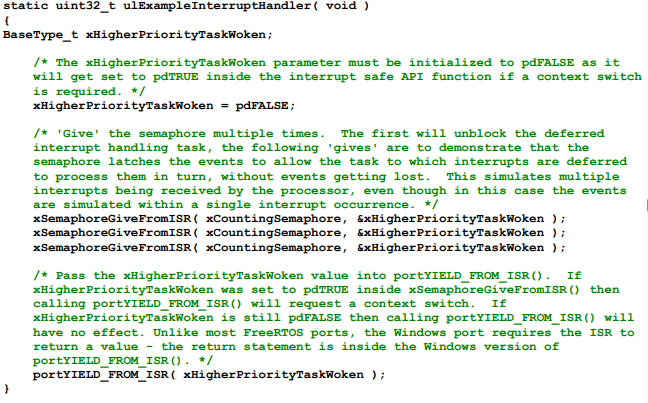
# 11. Using a counting semaphore to synchronize a task with an interrupt

It improves on the binary semaphore Example implementation by using a counting semaphore in place of the binary semaphore. main() is changed to include a call to xSemaphoreCreateCounting() in place of the call to xSemaphoreCreateBinary().

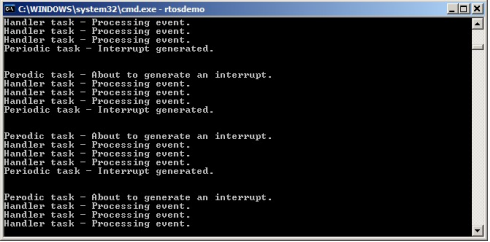


To simulate multiple events occurring at high frequency, the interrupt service routine is changed to ‘give’ the semaphore more than once per interrupt. Each event is latched in the semaphore’s count value.

The modified interrupt service routine is



As can be seen, the task to which interrupt handling is deferred processes all three [simulated] events each time an interrupt is generated. The events are latched into the count value of the semaphore, allowing the task to process them in turn



# 12. Deferring Work to the RTOS Daemon Task

* The deferred interrupt handling examples presented so far have required the application writer to create a task for each interrupt that uses the deferred processing technique.
* It is also possible to use the **xTimerPendFunctionCallFromISR()** API function to defer interrupt processing to the RTOS daemon task—removing the need to create a separate task for each interrupt.
* Deferring interrupt processing to the daemon task is called **‘centralized deferred interrupt processing’**
* The xTimerPendFunctionCall() and xTimerPendFunctionCallFromISR() API functions use the same timer command queue to send an ‘execute function’ command to the daemon task. The function sent to the daemon task is then executed in the context of the daemon task.
* **Advantages of centralized deferred interrupt processing include:**
* **Lower resource usage**

It removes the need to create a separate task for each deferred interrupt.

* **Simplified user model**

The deferred interrupt handling function is a standard C function.

* **Disadvantages of centralized deferred interrupt processing include:**
* **Less flexibility**

It is not possible to set the priority of each deferred interrupt handling task separately. Each deferred interrupt handling function executes at the priority of the daemon task, the priority of the daemon task is set by the configTIMER\_TASK\_PRIORITY compile time configuration constant within FreeRTOSConfig.h.

* **Less determinism**

xTimerPendFunctionCallFromISR() sends a command to the back of the timer command queue. Commands that were already in the timer command queue will be processed by the daemon task before the ‘execute function’ command sent to the queue by xTimerPendFunctionCallFromISR().

## 12.1 xTimerPendFunctionCallFromISR()

* It is the interrupt safe version of xTimerPendFunctionCall().
* Both API functions allow a function provided by the application writer to be executed by, and therefore in the context of, the RTOS daemon task.
* Both the function to be executed, and the value of the function’s input parameters, are sent to the daemon task on the timer command queue.
* When the function actually executes is therefore dependent on the priority of the daemon task relative to other tasks in the application.

**BaseType\_t xTimerPendFunctionCallFromISR( PendedFunction\_t xFunctionToPend, void \*pvParameter1, uint32\_t ulParameter2, BaseType\_t \*pxHigherPriorityTaskWoken );**

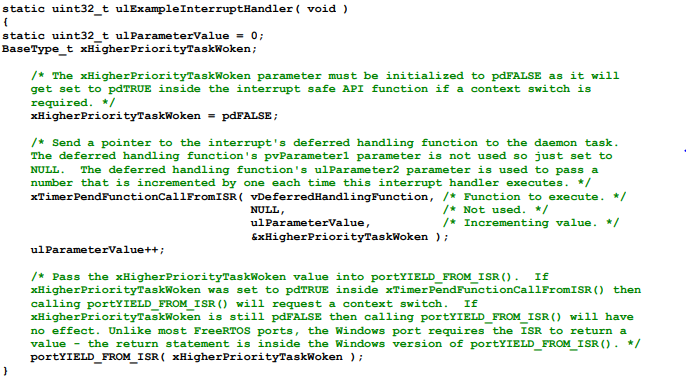
* The prototype to which a function passed in the xFunctionToPend parameter of xTimerPendFunctionCallFromISR() must conform

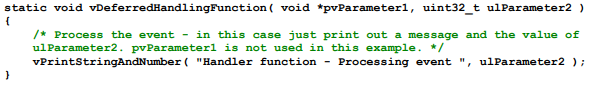
**void vPendableFunction( void \*pvParameter1, uint32\_t ulParameter2 );**

| **Parameter Name/ Returned Value** | **Description** |
| --- | --- |
| xFunctionToPend | * A pointer to the function that will be executed in the daemon task (in effect, just the function name) |
| pvParameter1 | * The value that will be passed into the function that is executed by the daemon task as the function’s pvParameter1 parameter. The parameter has a void \* type to allow it to be used to pass any data type. For example, integer types can be directly cast to a void \*, alternatively the void \* can be used to point to a structure. |
| ulParameter2 | * The value that will be passed into the function that is executed by the daemon task as the function’s ulParameter2 parameter. |
| pxHigherPriorityTaskWoken | * 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. * If xTimerPendFunctionCallFromISR() sets this value to pdTRUE, then a context switch must be performed before the interrupt is exited. This will ensure that the interrupt returns directly to the daemon task, as the daemon task will be the highest priority Ready state task |
| Returned value | * There are two possible return values:  1. **pdPASS**   pdPASS will be returned if the ‘execute function’ command was written to the timer command queue.   1. **pdFAIL**   pdFAIL will be returned if the ‘execute function’ command could not be written to the timer command queue because the timer command queue was already full. |

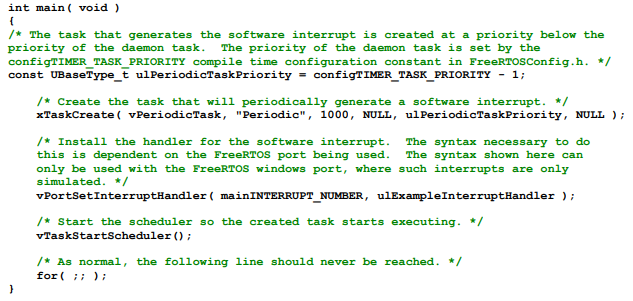
## 12.2 Example : Centralized deferred interrupt processing

* It provides similar functionality to above example , but without using a semaphore, and without creating a task specifically to perform the processing necessitated by the interrupt. Instead, the processing is performed by the RTOS daemon task.
* The interrupt service routine used calls xTimerPendFunctionCallFromISR() to pass a pointer to a function called vDeferredHandlingFunction() to the daemon task. The deferred interrupt processing is performed by the vDeferredHandlingFunction() function. The interrupt service routine increments a variable called ulParameterValue each time it executes.
* ulParameterValue is used as the value of ulParameter2 in the call to xTimerPendFunctionCallFromISR(), so will also be used as the value of ulParameter2 in the call to vDeferredHandlingFunction() when vDeferredHandlingFunction() is executed by the daemon task. The function’s other parameter, pvParameter1, is not used in this example.

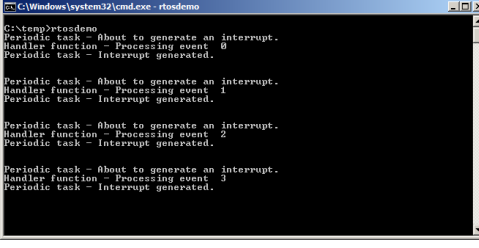


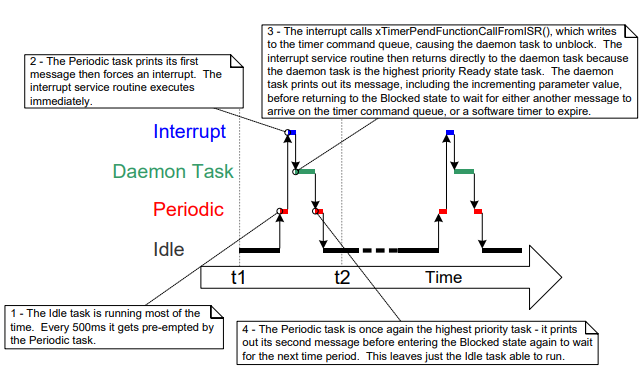


* vPeriodicTask() is the task that periodically generates software interrupts. It is created with a priority below the priority of the daemon task to ensure it is pre-empted by the daemon task as soon as the daemon task leaves the Blocked state.



* The priority of the daemon task is higher than the priority of the task that generates the software interrupt, so vDeferredHandlingFunction() is executed by the daemon task as soon as the interrupt is generated. That results in the message output by vDeferredHandlingFunction() appearing in between the two messages output by the periodic task, just as it did when a semaphore was used to unblock a dedicated deferred interrupt processing task





# 13. 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.
* xQueueSendToFrontFromISR() is the version of xQueueSendToFront() that is safe to use in an interrupt service routine, xQueueSendToBackFromISR() is the version of xQueueSendToBack() that is safe to use in an interrupt service routine, and xQueueReceiveFromISR() is the version of xQueueReceive() that is safe to use in an interrupt service routine.

## 13.1 xQueueSendToFrontFromISR()

BaseType\_t xQueueSendToFrontFromISR( QueueHandle\_t xQueue, void \*pvItemToQueue BaseType\_t \*pxHigherPriorityTaskWoken );

## 13.2 xQueueSendToBackFromISR()

BaseType\_t xQueueSendToBackFromISR( QueueHandle\_t xQueue, void \*pvItemToQueue BaseType\_t \*pxHigherPriorityTaskWoken );

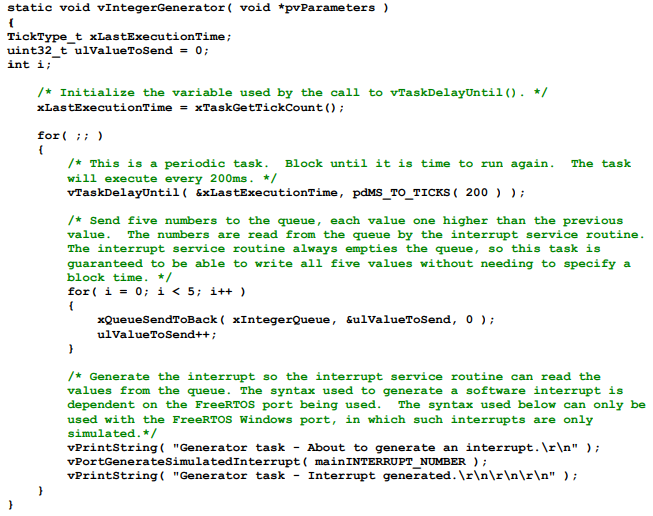
* xQueueSendFromISR() and xQueueSendToBackFromISR() are functionally equivalent.

| **Parameter Name/ Returned Value** | **Description** |
| --- | --- |
| xQueue | * The handle of the queue to which the data is being sent (written). The queue handle will have been returned from the call to xQueueCreate() used to create the queue |
| pvItemToQueue | * A pointer to the data that will be copied into the queue. * The size of each item the queue can hold is set when the queue is created, so this many bytes will be copied from pvItemToQueue into the queue storage area. |
| pxHigherPriorityTaskWoken | * It is possible that a single queue will have one or more tasks blocked on it, waiting for data to become available. Calling xQueueSendToFrontFromISR() or xQueueSendToBackFromISR() can make data available, and so cause such a task to leave the Blocked state. If calling the API function causes a task to leave the Blocked state, and the unblocked task has a priority higher than the currently executing task (the task that was interrupted), then, internally, the API function will set \*pxHigherPriorityTaskWoken to pdTRUE. * If xQueueSendToFrontFromISR() or xQueueSendToBackFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt is exited. This will ensure that the interrupt returns directly to the highest priority Ready state task. |
| Returned value | * There are two possible return values:  1. **pdPASS**   pdPASS is returned only if data has been sent successfully to the queue.   1. **errQUEUE\_FULL**   errQUEUE\_FULL is returned if data cannot be sent to the queue because the queue is already full |

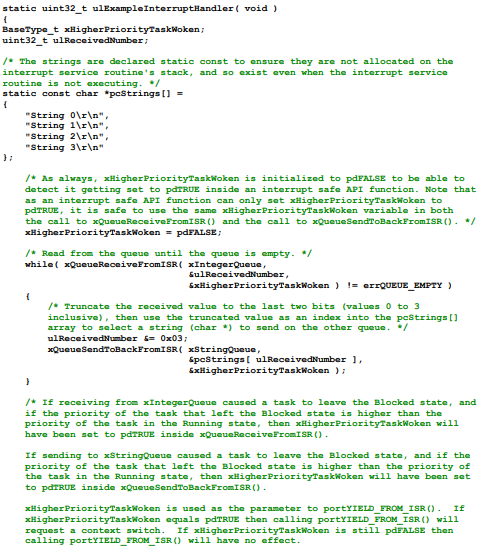
## 13.3 Example : Sending and receiving on a queue from within an interrupt

This example demonstrates xQueueSendToBackFromISR() and xQueueReceiveFromISR() being used within the same interrupt. As before, for convenience the interrupt is generated by software.

A periodic task is created that sends five numbers to a queue every 200 milliseconds. It generates a software interrupt only after all five values have been sent.

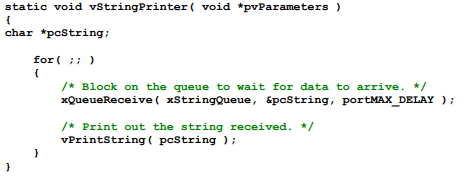


The interrupt service routine calls xQueueReceiveFromISR() repeatedly until all the values written to the queue by the periodic task have been read out, and the queue is left empty. The last two bits of each received value are used as an index into an array of strings. A pointer to the string at the corresponding index position is then sent to a different queue using a call to xQueueSendFromISR().

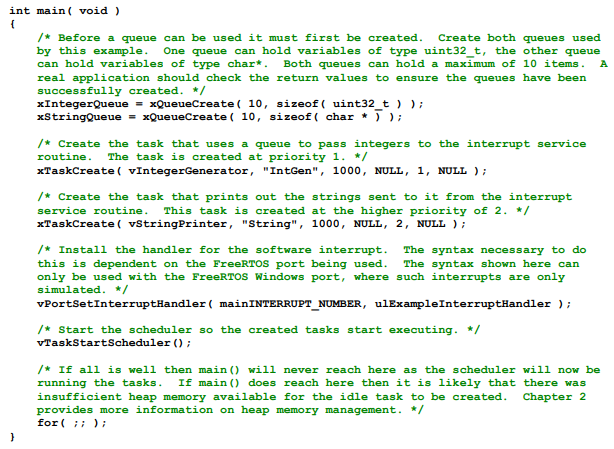


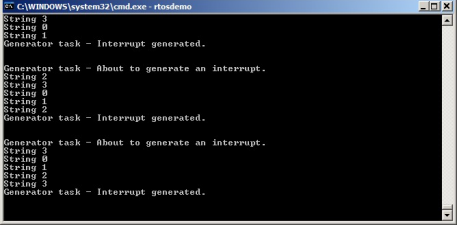


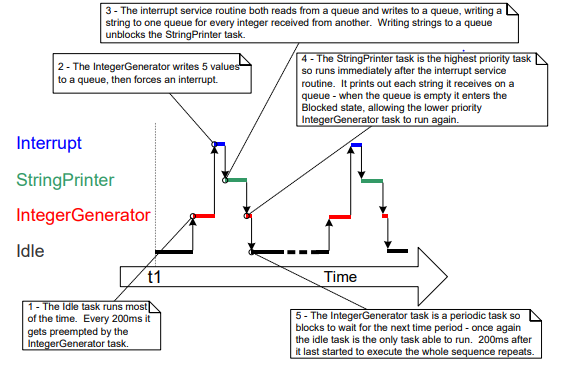
The task that receives the character pointers from the interrupt service routine blocks on the queue until a message arrives, printing out each string as it is received



As normal, main() creates the required queues and tasks before starting the scheduler







# 14. Interrupt Nesting

* It is common for confusion to arise between task priorities and interrupt priorities. This section discusses interrupt priorities, which are the priorities at which interrupt service routines (ISRs) execute relative to each other.
* The priority assigned to a task is in no way related to the priority assigned to an interrupt.
* Hardware decides when an ISR will execute, whereas software decides when a task will execute. An ISR executed in response to a hardware interrupt will interrupt a task, but a task cannot pre-empt an ISR.
* Ports that support interrupt nesting require one or both of the constants to be defined in FreeRTOSConfig.h. configMAX\_SYSCALL\_INTERRUPT\_PRIORITY and configMAX\_API\_CALL\_INTERRUPT\_PRIORITY both define the same property.
* Older FreeRTOS ports use configMAX\_SYSCALL\_INTERRUPT\_PRIORITY, and newer FreeRTOS port use configMAX\_API\_CALL\_INTERRUPT\_PRIORITY.

| **Constant** | **Description** |
| --- | --- |
| configMAX\_SYSCALL\_INTERRUPT\_PRIORITY or configMAX\_API\_CALL\_INTERRUPT\_PRIORITY | * Sets the highest interrupt priority from which interrupt-safe FreeRTOS API functions can be called. |
| configKERNEL\_INTERRUPT\_PRIORITY | * Sets the interrupt priority used by the tick interrupt, and must always be set to the lowest possible interrupt priority. * If the FreeRTOS port in use does not also use the configMAX\_SYSCALL\_INTERRUPT\_PRIORITY constant, then any interrupt that uses interrupt-safe FreeRTOS API functions must also execute at the priority defined by configKERNEL\_INTERRUPT\_PRIORITY |

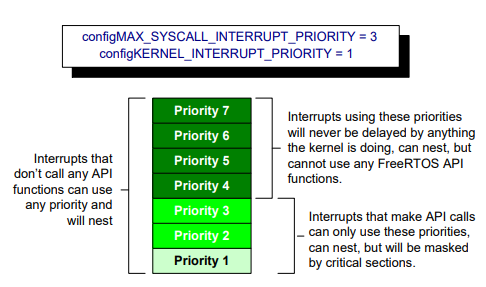
Each interrupt source has a numeric priority, and a logical priority:

**∙ Numeric priority**

The numeric priority is simply the number assigned to the interrupt priority. For example, if an interrupt is assigned a priority of 7, then its numeric priority is 7. Likewise, if an interrupt is assigned a priority of 200, then its numeric priority is 200.

**∙ Logical priority**

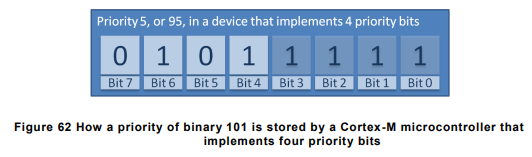
* An interrupt’s logical priority describes that interrupt’s precedence over other interrupts.
* If two interrupts of differing priority occur at the same time, then the processor will execute the ISR for whichever of the two interrupts has the higher logical priority before it executes the ISR for whichever of the two interrupts has the lower logical priority.
* An interrupt can interrupt (nest with) any interrupt that has a lower logical priority, but an interrupt cannot interrupt (nest with) any interrupt that has an equal or higher logical priority.
* The relationship between an interrupt’s numeric priority and logical priority is dependent on the processor architecture; on some processors, the higher the numeric priority assigned to an interrupt the higher that interrupt’s logical priority will be, while on other processor architectures the higher the numeric priority assigned to an interrupt the lower that interrupt’s logical priority will be.
* A full interrupt nesting model is created by setting configMAX\_SYSCALL\_INTERRUPT\_PRIORITY to a higher logical interrupt priority than configKERNEL\_INTERRUPT\_PRIORITY
* The processor has seven unique interrupt priorities.
* Interrupts assigned a numeric priority of 7 have a higher logical priority than interrupts assigned a numeric priority of 1.
* configKERNEL\_INTERRUPT\_PRIORITY is set to one.
* configMAX\_SYSCALL\_INTERRUPT\_PRIORITY is set to three



* Interrupts that use priorities 1 to 3, inclusive, are prevented from executing while the kernel or the application is inside a critical section. ISRs running at these priorities can use interrupt-safe FreeRTOS API functions
* Interrupts that use priority 4, or above, are not affected by critical sections, so nothing the scheduler does will prevent these interrupts from executing immediately—within the limitations of the hardware itself. ISRs executing at these priorities cannot use any FreeRTOS API functions.
* Typically, functionality that requires very strict timing accuracy (motor control, for example) would use a priority above configMAX\_SYSCALL\_INTERRUPT\_PRIORITY to ensure the scheduler does not introduce jitter into the interrupt response time

# 15. A Note to ARM Cortex-M1 and ARM GIC Users

* The ARM Cortex cores, and ARM Generic Interrupt Controllers (GICs), use numerically low priority numbers to represent logically high priority interrupts. This can seem counter-intuitive, and is easy to forget. If you wish to assign an interrupt a logically low priority, then it must be assigned a numerically high value
* The Cortex-M interrupt controller allows a maximum of eight bits to be used to specify each interrupt priority, making 255 the lowest possible priority. Zero is the highest priority. However, Cortex-M microcontrollers normally only implement a subset of the eight possible bits. The number of bits actually implemented is dependent on the microcontroller family.
* When only a subset of the eight possible bits has been implemented, it is only the most significant bits of the byte that can be used—leaving the least significant bits unimplemented. Unimplemented bits can take any value, but it is normal to set them to 1.



* the binary value 101 has been shifted into the most significant four bits because the least significant four bits are not implemented.
* The unimplemented bits have been set to 1. Some library functions expect priority values to be specified after they have been shifted up into the implemented (most significant) bits
* Decimal 95 is binary 101 shifted up by four to make binary 101nnnn (where ‘n’ is an unimplemented bit), and with the unimplemented bits set to 1 to make binary 1011111

# 16. Example Codes

## 16.1 Use of Binary Semaphores

**#include "FreeRTOS.h"**

**#include "task.h"**

**#include "semphr.h"**

**#include <stdio.h>**

**#include <stdarg.h>**

**/\* Declare a binary semaphore handle \*/**

**SemaphoreHandle\_t xBinarySemaphore;**

**void vApplicationIdleHook(void)**

**{**

**}**

**/\* Utility function to print to console \*/**

**void vPrintString(const char \*format, ...) {**

**va\_list args;**

**va\_start(args, format);**

**vprintf(format, args);**

**va\_end(args);**

**fflush(stdout);**

**}**

**/\* Task function that periodically generates a software interrupt \*/**

**void vPeriodicTask(void \*pvParameters) {**

**const TickType\_t xDelay500ms = pdMS\_TO\_TICKS(500UL);**

**for (;;) {**

**/\* Block until it is time to generate the software interrupt again \*/**

**vTaskDelay(xDelay500ms);**

**vPrintString("Periodic task - About to generate an interrupt.\r\n");**

**/\* Simulate an interrupt handling by giving the semaphore \*/**

**if (xSemaphoreGive(xBinarySemaphore) != pdTRUE) {**

**vPrintString("Failed to give semaphore.\r\n");**

**}**

**vPrintString("Periodic task - Interrupt generated.\r\n\r\n\r\n");**

**}**

**}**

**/\* Task function that handles the software interrupt \*/**

**void vHandlerTask(void \*pvParameters) {**

**for (;;) {**

**/\* Wait indefinitely for the semaphore \*/**

**if (xSemaphoreTake(xBinarySemaphore, portMAX\_DELAY) == pdTRUE) {**

**vPrintString("Handler task - Processing event.\r\n");**

**}**

**}**

**}**

**/\* Main function \*/**

**int main(void) {**

**/\* Before a semaphore is used it must be explicitly created \*/**

**xBinarySemaphore = xSemaphoreCreateBinary();**

**/\* Check the semaphore was created successfully \*/**

**if (xBinarySemaphore != NULL) {**

**/\* Create the handler task \*/**

**xTaskCreate(vHandlerTask, "Handler", configMINIMAL\_STACK\_SIZE, NULL, 2, NULL);**

**/\* Create the periodic task \*/**

**xTaskCreate(vPeriodicTask, "Periodic", configMINIMAL\_STACK\_SIZE, NULL, 1, NULL);**

**/\* Start the FreeRTOS scheduler \*/**

**vTaskStartScheduler();**

**}**

**/\* The following line should never be reached \*/**

**for (;;);**

**}**

**Output:**

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.

## 16.2 Use of xSemaphoreGiveFromISR()

**#include "FreeRTOS.h"**

**#include "task.h"**

**#include "semphr.h"**

**#include <stdio.h>**

**#include <stdarg.h>**

**/\* Declare a binary semaphore handle \*/**

**SemaphoreHandle\_t xBinarySemaphore;**

**void vApplicationIdleHook(void)**

**{**

**}**

**/\* Utility function to print to console \*/**

**void vPrintString(const char \*format, ...) {**

**va\_list args;**

**va\_start(args, format);**

**vprintf(format, args);**

**va\_end(args);**

**fflush(stdout);**

**}**

**/\* Task function that handles the software interrupt \*/**

**void vHandlerTask(void \*pvParameters) {**

**for (;;) {**

**/\* Wait indefinitely for the semaphore \*/**

**if (xSemaphoreTake(xBinarySemaphore, portMAX\_DELAY) == pdTRUE) {**

**vPrintString("Handler task - Processing event.\r\n");**

**}**

**}**

**}**

**/\* Function to simulate raising an interrupt \*/**

**void vRaiseInterrupt(void) {**

**BaseType\_t xHigherPriorityTaskWoken = pdFALSE;**

**/\* Print ISR message \*/**

**vPrintString("Simulated ISR: Giving semaphore.\r\n");**

**/\* Give the semaphore to unblock the handler task \*/**

**xSemaphoreGiveFromISR(xBinarySemaphore, &xHigherPriorityTaskWoken);**

**/\* Perform a context switch if necessary \*/**

**portYIELD\_FROM\_ISR(xHigherPriorityTaskWoken);**

**}**

**/\* Task function that periodically generates an interrupt \*/**

**void vPeriodicTask(void \*pvParameters) {**

**const TickType\_t xDelay500ms = pdMS\_TO\_TICKS(500UL);**

**/\* As per most tasks, this task is implemented within an infinite loop. \*/**

**for (;;) {**

**/\* Print message indicating intent to generate an interrupt \*/**

**vPrintString("Periodic task - About to generate an interrupt.\r\n");**

**/\* Simulate raising an interrupt \*/**

**vRaiseInterrupt();**

**/\* Print message indicating interrupt generation \*/**

**vPrintString("Periodic task - Interrupt generated.\r\n");**

**/\* Block until it is time to generate the interrupt again. \*/**

**vTaskDelay(xDelay500ms);**

**}**

**}**

**/\* Main function \*/**

**int main(void) {**

**/\* Before a semaphore is used it must be explicitly created \*/**

**xBinarySemaphore = xSemaphoreCreateBinary();**

**/\* Check the semaphore was created successfully \*/**

**if (xBinarySemaphore != NULL) {**

**/\* Create the handler task \*/**

**xTaskCreate(vHandlerTask, "Handler", configMINIMAL\_STACK\_SIZE, NULL, 2, NULL);**

**/\* Create the periodic task \*/**

**xTaskCreate(vPeriodicTask, "Periodic", configMINIMAL\_STACK\_SIZE, NULL, 1, NULL);**

**/\* Start the FreeRTOS scheduler \*/**

**vTaskStartScheduler();**

**}**

**/\* The following line should never be reached \*/**

**for (;;);**

**}**

**Output:**

Periodic task - About to generate an interrupt.

Simulated ISR: Giving semaphore.

Handler task - Processing event.

Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.

Simulated ISR: Giving semaphore.

Handler task - Processing event.

Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.

Simulated ISR: Giving semaphore.

Handler task - Processing event.

Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.

Simulated ISR: Giving semaphore.

Handler task - Processing event.

Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.

Simulated ISR: Giving semaphore.

Handler task - Processing event.

Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.

Simulated ISR: Giving semaphore.

Handler task - Processing event.

Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.

Simulated ISR: Giving semaphore.

Handler task - Processing event.

Periodic task - Interrupt generated.

## 16.3 Binary semaphore with multiple interrupts

**#include "FreeRTOS.h"**

**#include "task.h"**

**#include "semphr.h"**

**#include <stdio.h>**

**#include <stdarg.h>**

**/\* Declare a binary semaphore handle \*/**

**SemaphoreHandle\_t xBinarySemaphore;**

**volatile uint32\_t ulInterruptCount = 0;**

**static uint32\_t taskCount = 0;**

**void vApplicationIdleHook(void)**

**{**

**}**

**/\* Function prototypes \*/**

**void vMyTask(void \*pvParameters);**

**void vPeriodicInterruptGenerator(void \*pvParameters);**

**/\* Utility function to print to console \*/**

**void vPrintString(const char \*format, ...) {**

**va\_list args;**

**va\_start(args, format);**

**vprintf(format, args);**

**va\_end(args);**

**fflush(stdout);**

**}**

**/\* Task function that waits for the binary semaphore \*/**

**void vMyTask(void \*pvParameters) {**

**for (;;) {**

**/\* Wait for the semaphore to be given by the ISR \*/**

**if (xSemaphoreTake(xBinarySemaphore, portMAX\_DELAY) == pdTRUE) {**

**vPrintString("Task: Processing interrupt event.\r\n");**

**taskCount = ulInterruptCount;**

**/\* Simulate some processing time \*/**

**vTaskDelay(pdMS\_TO\_TICKS(500));**

**vPrintString("Task: Finished processing event.%lu\r\n",taskCount);**

**}**

**}**

**}**

**/\* Simulated ISR Handler \*/**

**void vMyISRHandler(void) {**

**BaseType\_t xHigherPriorityTaskWoken = pdFALSE;**

**/\* Give the semaphore to unblock the task \*/**

**xSemaphoreGiveFromISR(xBinarySemaphore, &xHigherPriorityTaskWoken);**

**/\* Request a context switch if a higher priority task was unblocked \*/**

**portYIELD\_FROM\_ISR(xHigherPriorityTaskWoken);**

**ulInterruptCount++;**

**vPrintString("Simulated ISR: Interrupt number %lu.\r\n", ulInterruptCount);**

**}**

**/\* Task to simulate periodic interrupts \*/**

**void vPeriodicInterruptGenerator(void \*pvParameters) {**

**const TickType\_t xDelay = pdMS\_TO\_TICKS(200); // 200 ms delay to simulate high frequency interrupts**

**for (;;) {**

**vTaskDelay(xDelay);**

**vPrintString("Simulated ISR: Giving semaphore.\r\n");**

**vMyISRHandler();**

**}**

**}**

**/\* Main function \*/**

**int main(void) {**

**/\* Create the binary semaphore \*/**

**xBinarySemaphore = xSemaphoreCreateBinary();**

**/\* Ensure the semaphore was created successfully \*/**

**if (xBinarySemaphore != NULL) {**

**/\* Create the task that will wait for the semaphore \*/**

**xTaskCreate(vMyTask, "MyTask", configMINIMAL\_STACK\_SIZE, NULL, 1, NULL);**

**/\* Create the task that will simulate periodic interrupts \*/**

**xTaskCreate(vPeriodicInterruptGenerator, "InterruptGenerator", configMINIMAL\_STACK\_SIZE, NULL, 2, NULL);**

**/\* Start the scheduler \*/**

**vTaskStartScheduler();**

**}**

**/\* The following line should never be reached \*/**

**for (;;) {**

**}**

**}**

**Output:**

Simulated ISR: Giving semaphore.

Simulated ISR: Interrupt number 1.

Task: Processing interrupt event.

Simulated ISR: Giving semaphore.

Simulated ISR: Interrupt number 2.

Simulated ISR: Giving semaphore.

Simulated ISR: Interrupt number 3.

Task: Finished processing event.1

Task: Processing interrupt event.

Simulated ISR: Giving semaphore.

Simulated ISR: Interrupt number 4.

Simulated ISR: Giving semaphore.

Simulated ISR: Interrupt number 5.

Task: Finished processing event.3

Task: Processing interrupt event.

Simulated ISR: Giving semaphore.

Simulated ISR: Interrupt number 6.

Simulated ISR: Giving semaphore.

Simulated ISR: Interrupt number 7.

Simulated ISR: Giving semaphore.

Simulated ISR: Interrupt number 8.

Task: Finished processing event.5

Task: Processing interrupt event.

Simulated ISR: Giving semaphore.

Simulated ISR: Interrupt number 9.

Simulated ISR: Giving semaphore.

Simulated ISR: Interrupt number 10.

Task: Finished processing event.8

Task: Processing interrupt event.

Simulated ISR: Giving semaphore.

Simulated ISR: Interrupt number 11.

Simulated ISR: Giving semaphore.

Simulated ISR: Interrupt number 12.

Simulated ISR: Giving semaphore.

Simulated ISR: Interrupt number 13.

Task: Finished processing event.10

Task: Processing interrupt event.

Simulated ISR: Giving semaphore.

Simulated ISR: Interrupt number 14.

Simulated ISR: Giving semaphore.

Simulated ISR: Interrupt number 15.

Task: Finished processing event.13

Task: Processing interrupt event.

Simulated ISR: Giving semaphore.

Simulated ISR: Interrupt number 16.

Simulated ISR: Giving semaphore.

Simulated ISR: Interrupt number 17.

Simulated ISR: Giving semaphore.

Simulated ISR: Interrupt number 18.

Task: Finished processing event.15

## 16.4 Use of counting semaphore

**#include "FreeRTOS.h"**

**#include "task.h"**

**#include "semphr.h"**

**#include <stdio.h>**

**#include <stdarg.h>**

**/\* Declare a counting semaphore handle \*/**

**SemaphoreHandle\_t xCountingSemaphore;**

**void vApplicationIdleHook(void)**

**{**

**}**

**/\* Utility function to print to console \*/**

**void vPrintString(const char \*format, ...) {**

**va\_list args;**

**va\_start(args, format);**

**vprintf(format, args);**

**va\_end(args);**

**fflush(stdout);**

**}**

**/\* Task function that handles the software interrupt \*/**

**void vHandlerTask(void \*pvParameters) {**

**for (;;) {**

**/\* Wait indefinitely for the semaphore \*/**

**if (xSemaphoreTake(xCountingSemaphore, portMAX\_DELAY) == pdTRUE) {**

**vPrintString("Handler task - Processing event.\r\n");**

**}**

**}**

**}**

**/\* Function to simulate raising an interrupt \*/**

**void vRaiseInterrupt(void) {**

**BaseType\_t xHigherPriorityTaskWoken = pdFALSE;**

**/\* Print ISR message \*/**

**vPrintString("Simulated ISR: Giving semaphore.\r\n");**

**/\* Give the semaphore to unblock the handler task \*/**

**xSemaphoreGiveFromISR(xCountingSemaphore, &xHigherPriorityTaskWoken);**

**xSemaphoreGiveFromISR(xCountingSemaphore, &xHigherPriorityTaskWoken);**

**xSemaphoreGiveFromISR(xCountingSemaphore, &xHigherPriorityTaskWoken);**

**/\* Perform a context switch if necessary \*/**

**portYIELD\_FROM\_ISR(xHigherPriorityTaskWoken);**

**}**

**/\* Task function that periodically generates an interrupt \*/**

**void vPeriodicTask(void \*pvParameters) {**

**const TickType\_t xDelay500ms = pdMS\_TO\_TICKS(500UL);**

**/\* As per most tasks, this task is implemented within an infinite loop. \*/**

**for (;;) {**

**/\* Print message indicating intent to generate an interrupt \*/**

**vPrintString("Periodic task - About to generate an interrupt.\r\n");**

**/\* Simulate raising an interrupt \*/**

**vRaiseInterrupt();**

**/\* Print message indicating interrupt generation \*/**

**vPrintString("Periodic task - Interrupt generated.\r\n\n");**

**/\* Block until it is time to generate the interrupt again. \*/**

**vTaskDelay(xDelay500ms);**

**}**

**}**

**/\* Main function \*/**

**int main(void) {**

**/\* Before a semaphore is used it must be explicitly created \*/**

**xCountingSemaphore = xSemaphoreCreateCounting(10, 0);**

**/\* Check the semaphore was created successfully \*/**

**if (xCountingSemaphore != NULL) {**

**/\* Create the handler task \*/**

**xTaskCreate(vHandlerTask, "Handler", configMINIMAL\_STACK\_SIZE, NULL, 2, NULL);**

**/\* Create the periodic task \*/**

**xTaskCreate(vPeriodicTask, "Periodic", configMINIMAL\_STACK\_SIZE, NULL, 1, NULL);**

**/\* Start the FreeRTOS scheduler \*/**

**vTaskStartScheduler();**

**}**

**/\* The following line should never be reached \*/**

**for (;;);**

**}**

**Output:**

Periodic task - About to generate an interrupt.

Simulated ISR: Giving semaphore.

Handler task - Processing event.

Handler task - Processing event.

Handler task - Processing event.

Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.

Simulated ISR: Giving semaphore.

Handler task - Processing event.

Handler task - Processing event.

Handler task - Processing event.

Periodic task - Interrupt generated.

## 16.5 Deferred Daemon task

**#include "FreeRTOS.h"**

**#include "task.h"**

**#include "timers.h"**

**#include <stdio.h>**

**#include <stdarg.h>**

**void vApplicationIdleHook(void)**

**{**

**}**

**/\* Utility function to print to console \*/**

**void vPrintString(const char \*format, ...) {**

**va\_list args;**

**va\_start(args, format);**

**vprintf(format, args);**

**va\_end(args);**

**fflush(stdout);**

**}**

**/\* Deferred handling function \*/**

**void vDeferredHandlingFunction(void \*pvParameter1, uint32\_t ulParameter2) {**

**/\* Process the event - in this case just print out a message and the value of ulParameter2 \*/**

**vPrintString("Handler function - Processing event %lu\r\n", ulParameter2);**

**}**

**/\* Example interrupt handler \*/**

**static uint32\_t ulExampleInterruptHandler(void) {**

**static uint32\_t ulParameterValue = 0;**

**BaseType\_t xHigherPriorityTaskWoken = pdFALSE;**

**xTimerPendFunctionCallFromISR(vDeferredHandlingFunction, /\* Function to execute. \*/**

**NULL, /\* Not used. \*/**

**ulParameterValue, /\* Incrementing value. \*/**

**&xHigherPriorityTaskWoken);**

**ulParameterValue++;**

**portYIELD\_FROM\_ISR(xHigherPriorityTaskWoken);**

**}**

**/\* Function to simulate raising an interrupt \*/**

**void vRaiseInterrupt(void) {**

**/\* Print ISR message \*/**

**vPrintString("Simulated ISR: Triggering example interrupt.\r\n");**

**/\* Simulate an interrupt by directly calling the handler \*/**

**ulExampleInterruptHandler();**

**}**

**/\* Task function that periodically generates an interrupt \*/**

**void vPeriodicTask(void \*pvParameters) {**

**const TickType\_t xDelay500ms = pdMS\_TO\_TICKS(500UL);**

**/\* As per most tasks, this task is implemented within an infinite loop. \*/**

**for (;;) {**

**/\* Print message indicating intent to generate an interrupt \*/**

**vPrintString("Periodic task - About to generate an interrupt.\r\n");**

**/\* Simulate raising an interrupt \*/**

**vRaiseInterrupt();**

**/\* Print message indicating interrupt generation \*/**

**vPrintString("Periodic task - Interrupt generated.\r\n");**

**/\* Block until it is time to generate the interrupt again. \*/**

**vTaskDelay(xDelay500ms);**

**}**

**}**

**/\* Main function \*/**

**int main(void) {**

**/\* The priority of the daemon task is set by the configTIMER\_TASK\_PRIORITY compile time configuration constant in FreeRTOSConfig.h. \*/**

**const UBaseType\_t ulPeriodicTaskPriority = configTIMER\_TASK\_PRIORITY - 1;**

**/\* Create the task that will periodically generate a software interrupt. \*/**

**xTaskCreate(vPeriodicTask, "Periodic", configMINIMAL\_STACK\_SIZE, NULL, ulPeriodicTaskPriority, NULL);**

**/\* Start the FreeRTOS scheduler \*/**

**vTaskStartScheduler();**

**/\* The following line should never be reached \*/**

**for (;;);**

**}**

## 16.6 Use of queues from ISR

**#include "FreeRTOS.h"**

**#include "task.h"**

**#include "queue.h"**

**#include "timers.h"**

**#include <stdio.h>**

**#include <stdarg.h>**

**// Define the queue handles globally**

**static QueueHandle\_t xIntegerQueue = NULL;**

**static QueueHandle\_t xStringQueue = NULL;**

**// Print utility function**

**void vPrintString(const char \*format, ...) {**

**va\_list args;**

**va\_start(args, format);**

**vprintf(format, args);**

**va\_end(args);**

**fflush(stdout);**

**}**

**// Idle hook function**

**void vApplicationIdleHook(void) {**

**// Idle hook code here**

**}**

**// Deferred handling function**

**void vDeferredHandlingFunction(void \*pvParameter1, uint32\_t ulParameter2) {**

**vPrintString("Handler function - Processing event %lu\r\n", ulParameter2);**

**}**

**// Example interrupt handler**

**static uint32\_t ulExampleInterruptHandler(void) {**

**BaseType\_t xHigherPriorityTaskWoken;**

**uint32\_t ulReceivedNumber;**

**static const char \*pcStrings[] = {**

**"String 0\r\n",**

**"String 1\r\n",**

**"String 2\r\n",**

**"String 3\r\n"**

**};**

**xHigherPriorityTaskWoken = pdFALSE;**

**// Read from the queue until the queue is empty**

**while (xQueueReceiveFromISR(xIntegerQueue, &ulReceivedNumber, &xHigherPriorityTaskWoken) != errQUEUE\_EMPTY) {**

**ulReceivedNumber &= 0x03;**

**xQueueSendToBackFromISR(xStringQueue, &pcStrings[ulReceivedNumber], &xHigherPriorityTaskWoken);**

**}**

**portYIELD\_FROM\_ISR(xHigherPriorityTaskWoken);**

**return 0; // Return value not used in this context**

**}**

**// Function to simulate raising an interrupt**

**void vRaiseInterrupt(void) {**

**vPrintString("Simulated ISR: Triggering example interrupt.\r\n");**

**ulExampleInterruptHandler();**

**}**

**// Integer generator task**

**static void vIntegerGenerator(void \*pvParameters) {**

**TickType\_t xLastExecutionTime;**

**uint32\_t ulValueToSend = 0;**

**int i;**

**xLastExecutionTime = xTaskGetTickCount();**

**for (;;) {**

**vTaskDelayUntil(&xLastExecutionTime, pdMS\_TO\_TICKS(200));**

**for (i = 0; i < 5; i++) {**

**xQueueSendToBack(xIntegerQueue, &ulValueToSend, 0);**

**ulValueToSend++;**

**}**

**vPrintString("Generator task - About to generate an interrupt.\r\n");**

**vRaiseInterrupt();**

**vPrintString("Generator task - Interrupt generated.\r\n\r\n\r\n");**

**}**

**}**

**// String printer task**

**static void vStringPrinter(void \*pvParameters) {**

**char \*pcString;**

**for (;;) {**

**xQueueReceive(xStringQueue, &pcString, portMAX\_DELAY);**

**vPrintString(pcString);**

**}**

**}**

**int main(void) {**

**xIntegerQueue = xQueueCreate(10, sizeof(uint32\_t));**

**xStringQueue = xQueueCreate(10, sizeof(char \*));**

**if (xIntegerQueue == NULL || xStringQueue == NULL) {**

**vPrintString("Failed to create queues\r\n");**

**}**

**xTaskCreate(vIntegerGenerator, "IntGen", configMINIMAL\_STACK\_SIZE, NULL, 1, NULL);**

**xTaskCreate(vStringPrinter, "String", configMINIMAL\_STACK\_SIZE, NULL, 2, NULL);**

**// Note: vPortSetInterruptHandler is specific to FreeRTOS Windows port.**

**// For actual MCU, ensure the proper interrupt setup.**

**// vPortSetInterruptHandler(mainINTERRUPT\_NUMBER, ulExampleInterruptHandler);**

**vTaskStartScheduler();**

**for (;;);**

**}**

**Output:**

Generator task - About to generate an interrupt.

Simulated ISR: Triggering example interrupt.

String 0

String 1

String 2

String 3

String 0

Generator task - Interrupt generated.

Generator task - About to generate an interrupt.

Simulated ISR: Triggering example interrupt.

String 1

String 2

String 3

String 0

String 1

Generator task - Interrupt generated.

Generator task - About to generate an interrupt.

Simulated ISR: Triggering example interrupt.

String 2

String 3

String 0

String 1

String 2

Generator task - Interrupt generated.

Generator task - About to generate an interrupt.

Simulated ISR: Triggering example interrupt.

String 3

String 0

String 1

String 2

String 3

Generator task - Interrupt generated.

Generator task - About to generate an interrupt.

Simulated ISR: Triggering example interrupt.

String 0

String 1

String 2

String 3

String 0

Generator task - Interrupt generated.

Generator task - About to generate an interrupt.

Simulated ISR: Triggering example interrupt.

String 1

String 2

String 3

String 0

String 1

Generator task - Interrupt generated.

Generator task - About to generate an interrupt.

Simulated ISR: Triggering example interrupt.

String 2

String 3

String 0

String 1

String 2

Generator task - Interrupt generated.

Generator task - About to generate an interrupt.

Simulated ISR: Triggering example interrupt.

String 3

String 0

String 1

String 2

String 3

Generator task - Interrupt generated.