Chapter 6
Interrupt Management

A guide by Chip Heath & Dan Heath

6.1 Introduction

1. How should the event be detected? Interrupts are normally used, but inputs can also be polled

2. 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.

3. How events are communicated to the main (non-ISR) code, and how can this code be
structured to best accommodate processing of potentially asynchronous occurrences?

6.1 Introduction

- 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.

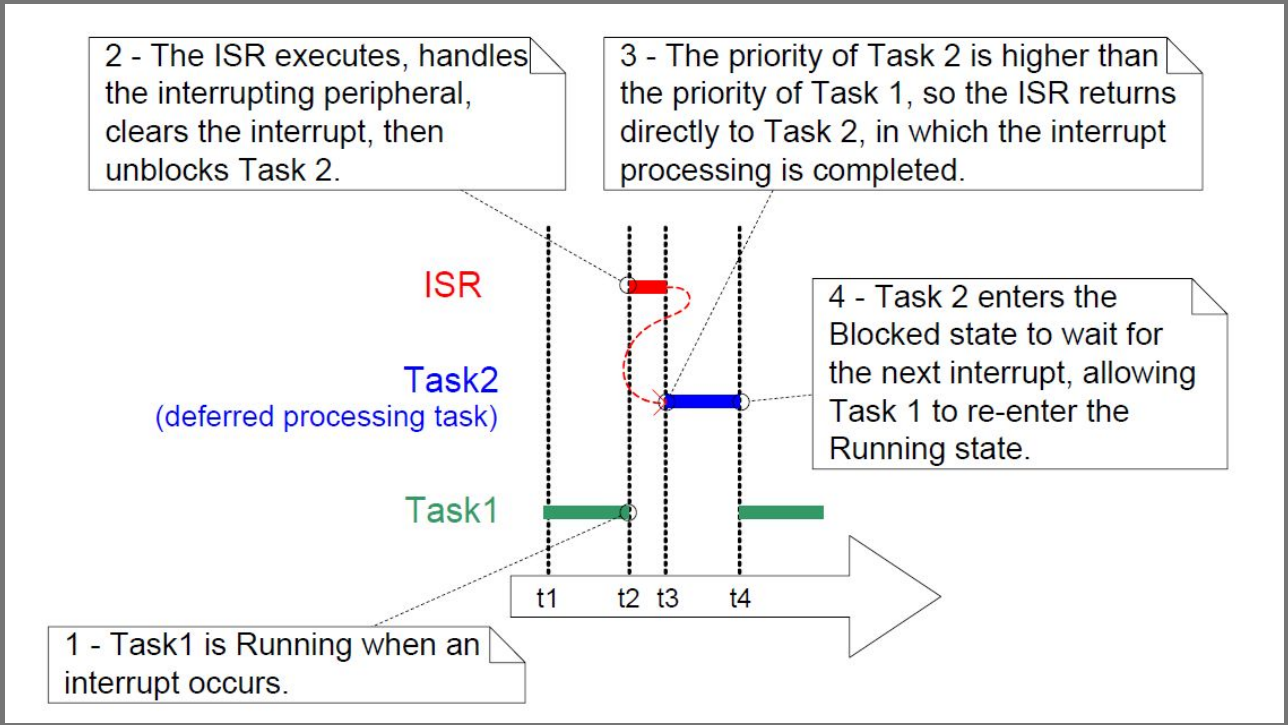
6.1 Scope

- 1:Which FreeRTOS API functions can be used from within an interrupt service routine.
- 2:Methods of deferring interrupt processing to a task.
- 3:How to create and use binary semaphores and counting semaphores.
- 4:The differences between binary and counting semaphores.
- 5:How to use a queue to pass data into and out of an interrupt service routine.
- The interrupt nesting model available with some FreeRTOS ports.
-

6.2 Using the FreeRTOS API from an ISR

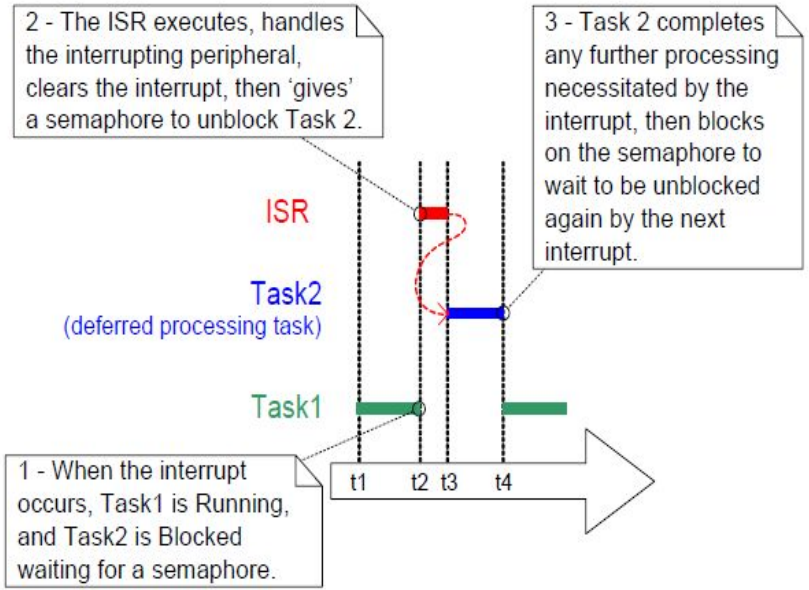
if an API function is called from an ISR, then it is not being called from a task, so there is no calling task that can be placed into the Blocked state. FreeRTOS solves this problem by providing two versions of some API functions; one version for use from tasks, and one version for use from ISRs. Functions intended for use from ISRs have “FromISR” appended to their name.

6.3 Deferred Interrupt Processing



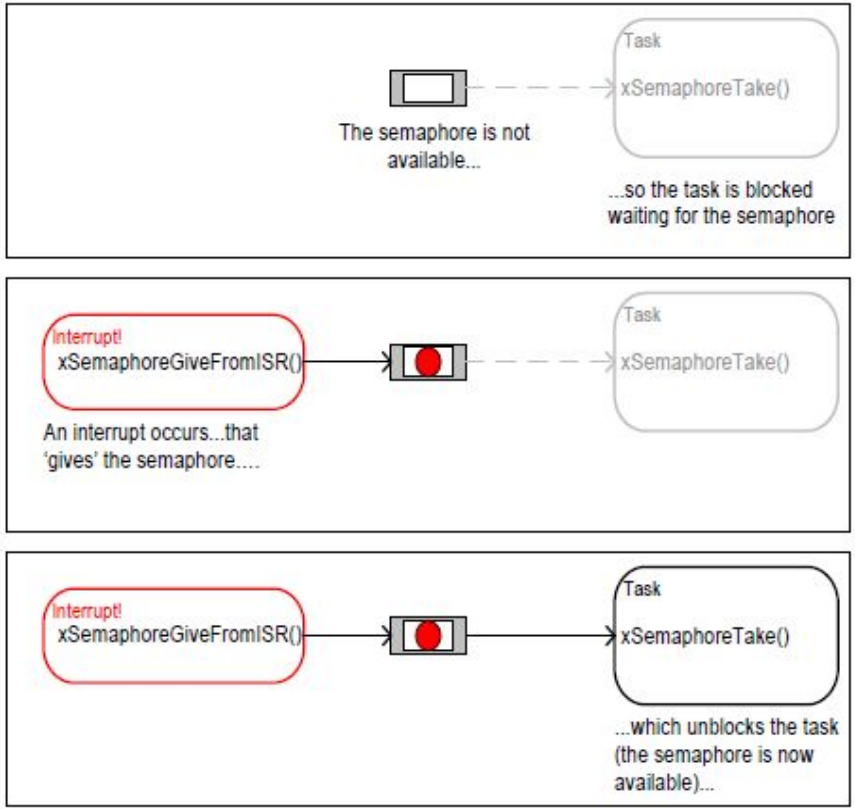
Completing interrupt processing in a high priority task

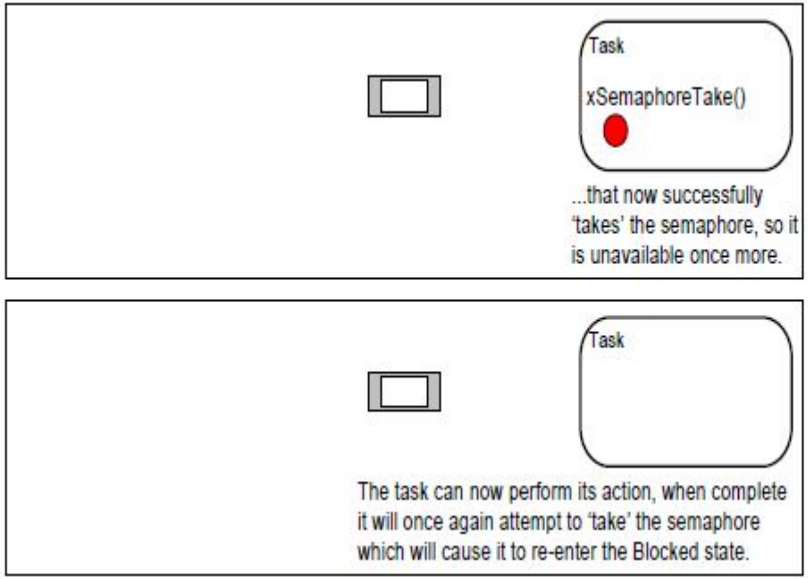
6.4 Binary Semaphores Used for Synchronization



Using a binary semaphore to implement deferred interrupt processing

6.4 Binary Semaphores Used for Synchronization





Using a binary semaphore to synchronize a task with an interrupt

6.4 Binary Semaphores Used for Synchronization

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.

The xSemaphoreTake() API Function

BaseType_t xSemaphoreTake(SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait);

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.

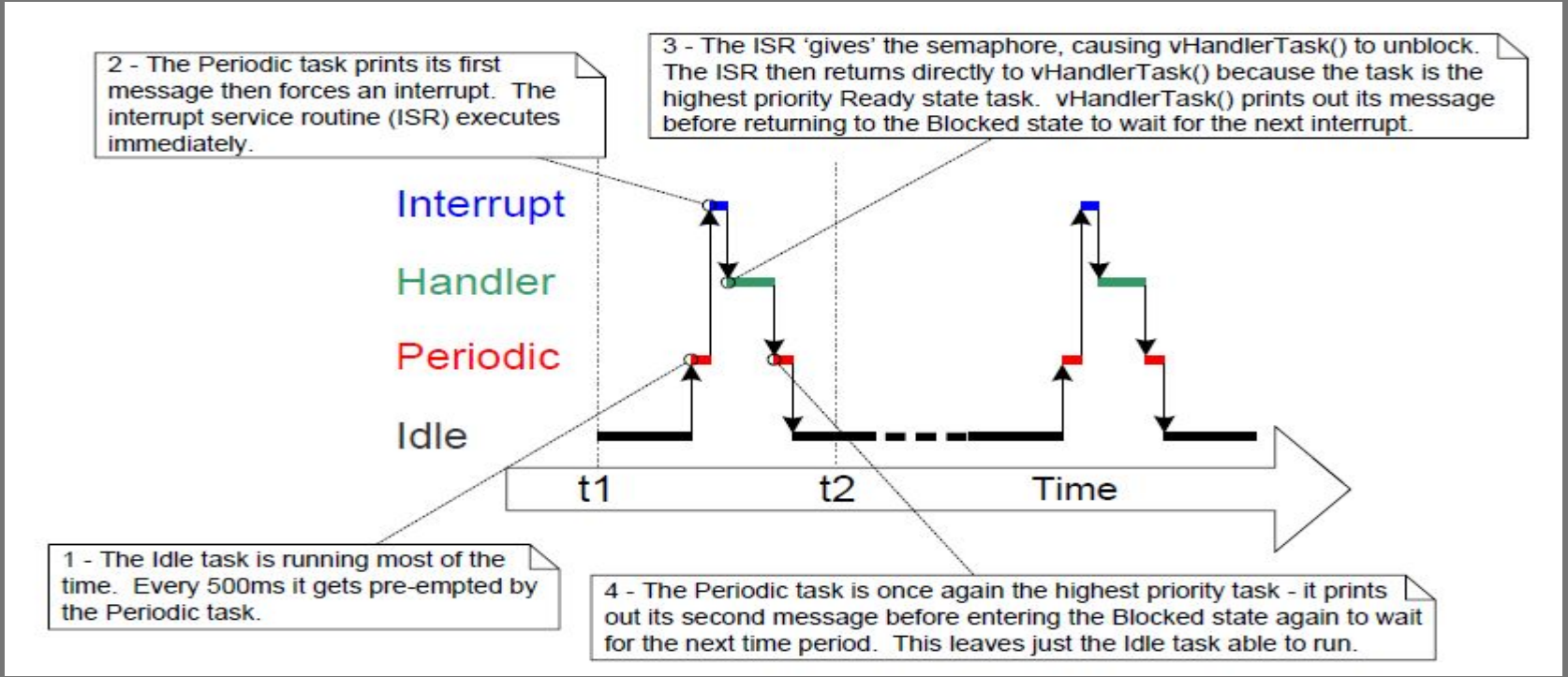
6.4 Binary Semaphores Used for Synchronization

The xSemaphoreGiveFromISR() API Function

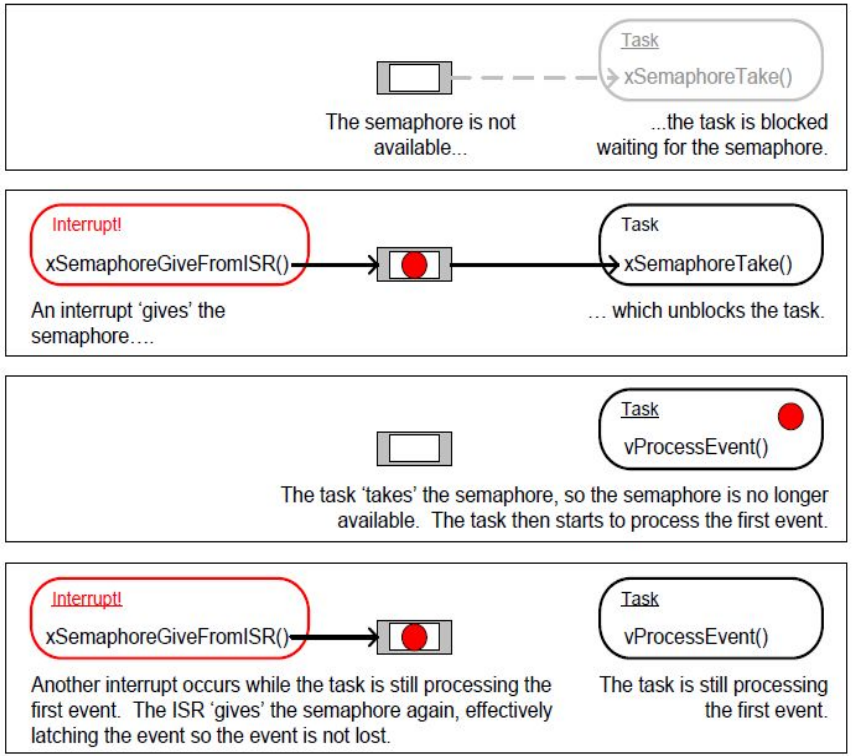
```
BaseType_t xSemaphoreGiveFromISR(
SemaphoreHandle_t xSemaphore,
BaseType_t *pxHigherPriorityTaskWoken );
```

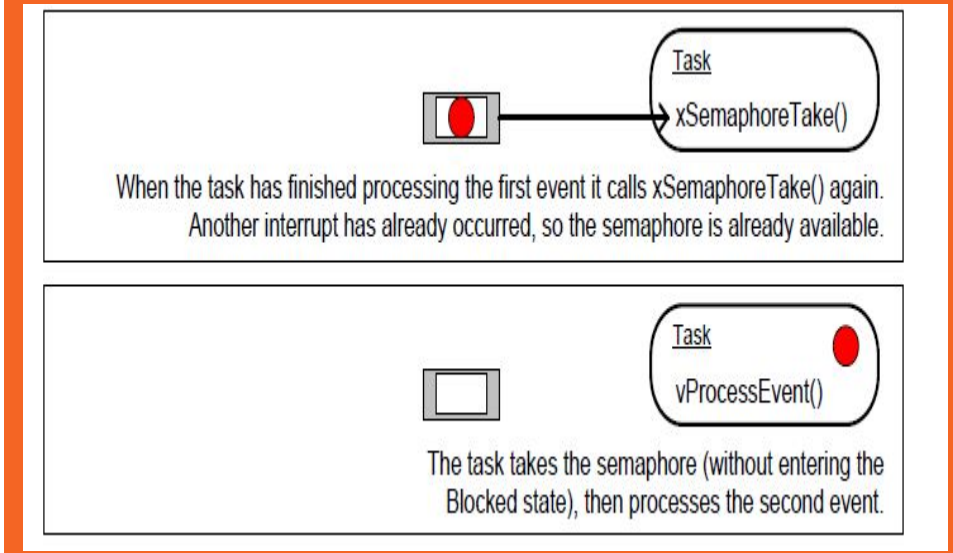
Calling xSemaphoreGiveFromISR() can make the semaphore available, and so cause a task that was waiting for the semaphore to leave the Blocked state. If calling xSemaphoreGiveFromISR() causes a task to leave the Blocked state, and the unblocked task has a priority higher than the currently executing task (the task that was interrupted), then, internally, xSemaphoreGiveFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE.

6.4 Binary Semaphores Used for Synchronization



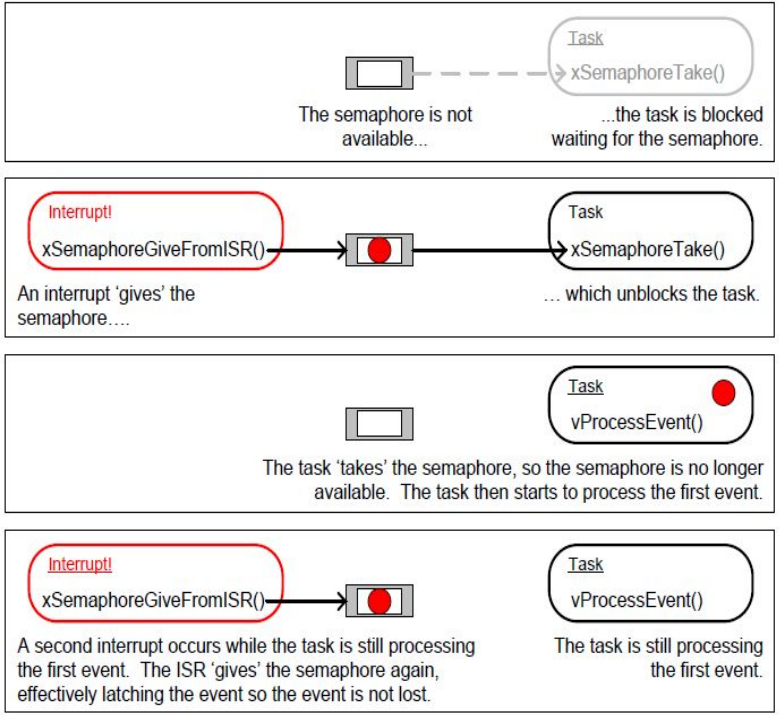
6.4 Binary Semaphores Used for Synchronization

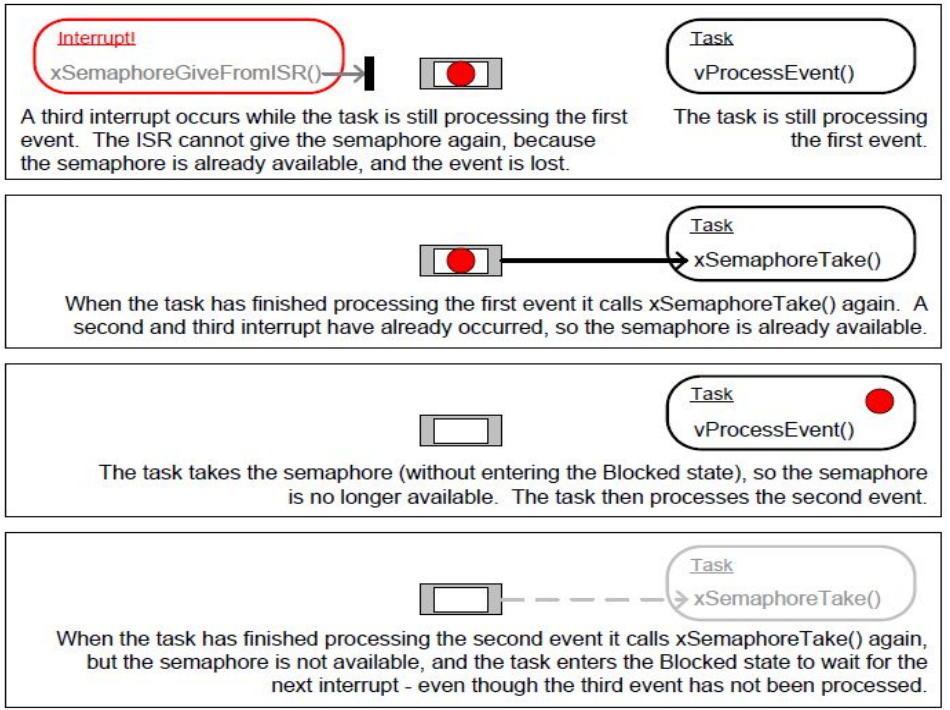




The scenario when one interrupt occurs before the task has finished processing the first event

6.4 Binary Semaphores Used for Synchronization





The scenario when two interrupts occur before the task has finished processing the first event

6.4 Binary Semaphores Used for Synchronization

EXAMPLE 1

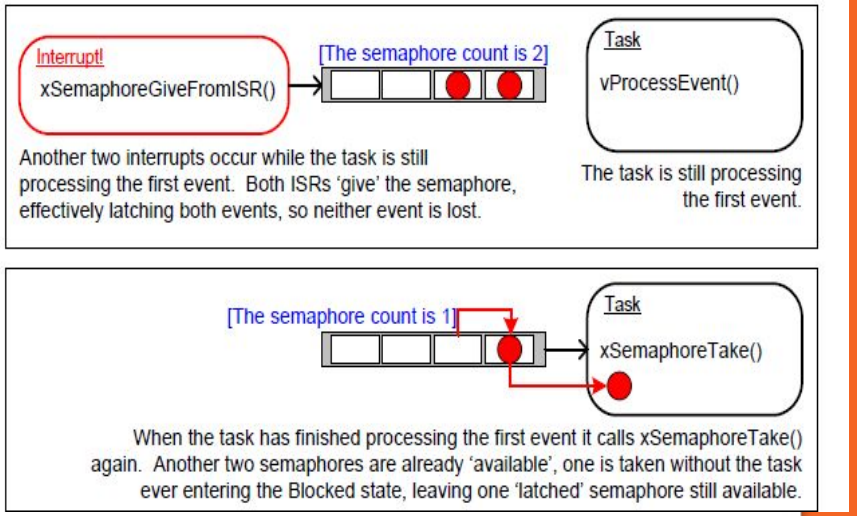
6.5 Counting Semaphores

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.

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.

2. 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

6.5 Counting Semaphores



Using a counting semaphore to 'count' events

6.5 Counting Semaphores

The xSemaphoreCreateCounting() API Function

SemaphoreHandle_t xSemaphoreCreateCounting(UBaseType_t uxMaxCount, UBaseType_t uxInitialCount);

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.

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.

EXAMPLE 2

6.6 Deferring Work to the RTOS Daemon Task

The xTimerPendFunctionCallFromISR() API Function

BaseType_t xTimerPendFunctionCallFromISR(PendedFunction_t xFunctionToPend, void*pvParameter1,

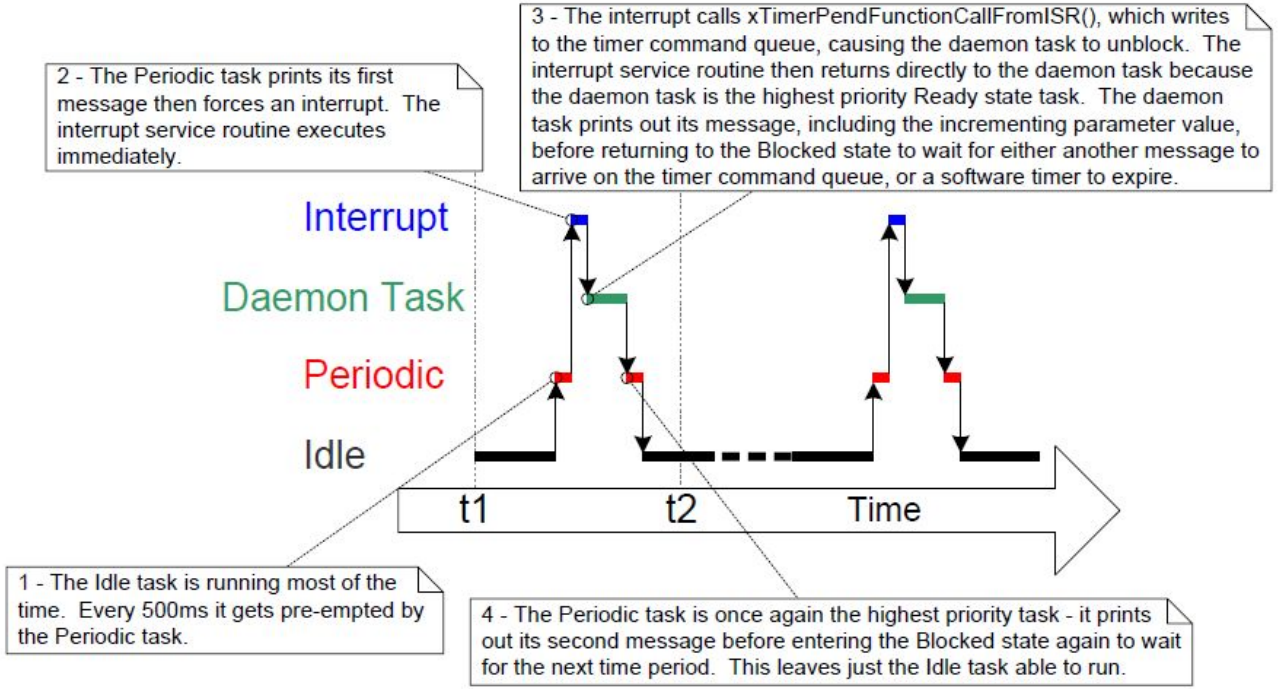
uint32_t ulParameter2,BaseType_t *pxHigherPriorityTaskWoken);

P1 : A pointer to the function that will be executed in the daemon task

P2 : The value that will be passed into the function that is executed by the daemon task as the function's 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 as structure.

P3 : The value that will be passed into the function that is executed by the daemon task as the function's parameter.

6.6 Deferring Work to the RTOS Daemon Task



6.6 Deferring Work to the RTOS Daemon Task

EXAMPLE 3

6.7 Using Queues within an Interrupt Service Routine

The xQueueSendToFrontFromISR()

```
BaseType_t xQueueSendToFrontFromISR(
QueueHandle_t xQueue,void *pvItemToQueue
BaseType_t *pxHigherPriorityTaskWoken);
```

The xQueueSendToBackFromISR()

```
BaseType_t xQueueSendToBackFromISR(
QueueHandle_t xQueue,void *pvItemToQueue
BaseType_t *pxHigherPriorityTaskWoken);
```

6.7 Using Queues within an Interrupt Service Routine

EXAMPLE 4