# FreeRTOS and introduction to Linux embedded

Livio Tenze

ltenze@units.it

April 22, 2026

# Interrupts

Embedded real-time systems have to take actions in response to **events that originate from the environment**. For example, a packet arriving on an Ethernet peripheral (the event) might require passing to a TCP/IP stack for processing (the action).
In each case, a judgment has to be made as to the best event processing implementation strategy:

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

# Why ISR short?

- 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 (add 'jitter' to) 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.

# Interrupts

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

**API**

Note that only API functions and macros ending in 'FromISR' or 'FROM_ISR' should be used within an interrupt service routine.

# Deferred interrupt techniques

- Binary semaphores
- Counting semaphores
- Queues
- Use daemon task (xTimerPendFunctionCallFromISR())
- Notification

# Deferred interrupt processing

A **Binary Semaphore** 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. **The interrupt processing is said to have been 'deferred' to a 'handler' task.**

# Deferred interrupt processing

*If the interrupt processing is particularly time critical*, then the handler task priority can be set to ensure that the handler task always pre-empts the other tasks in the system. **The ISR can then be implemented to include a context switch to ensure that the ISR returns directly to the handler task when the ISR itself has completed executing**. This has the effect of ensuring that the entire event processing executes contiguously in time, just as if it had all been implemented within the ISR itself.

# Deferred interrupt processing

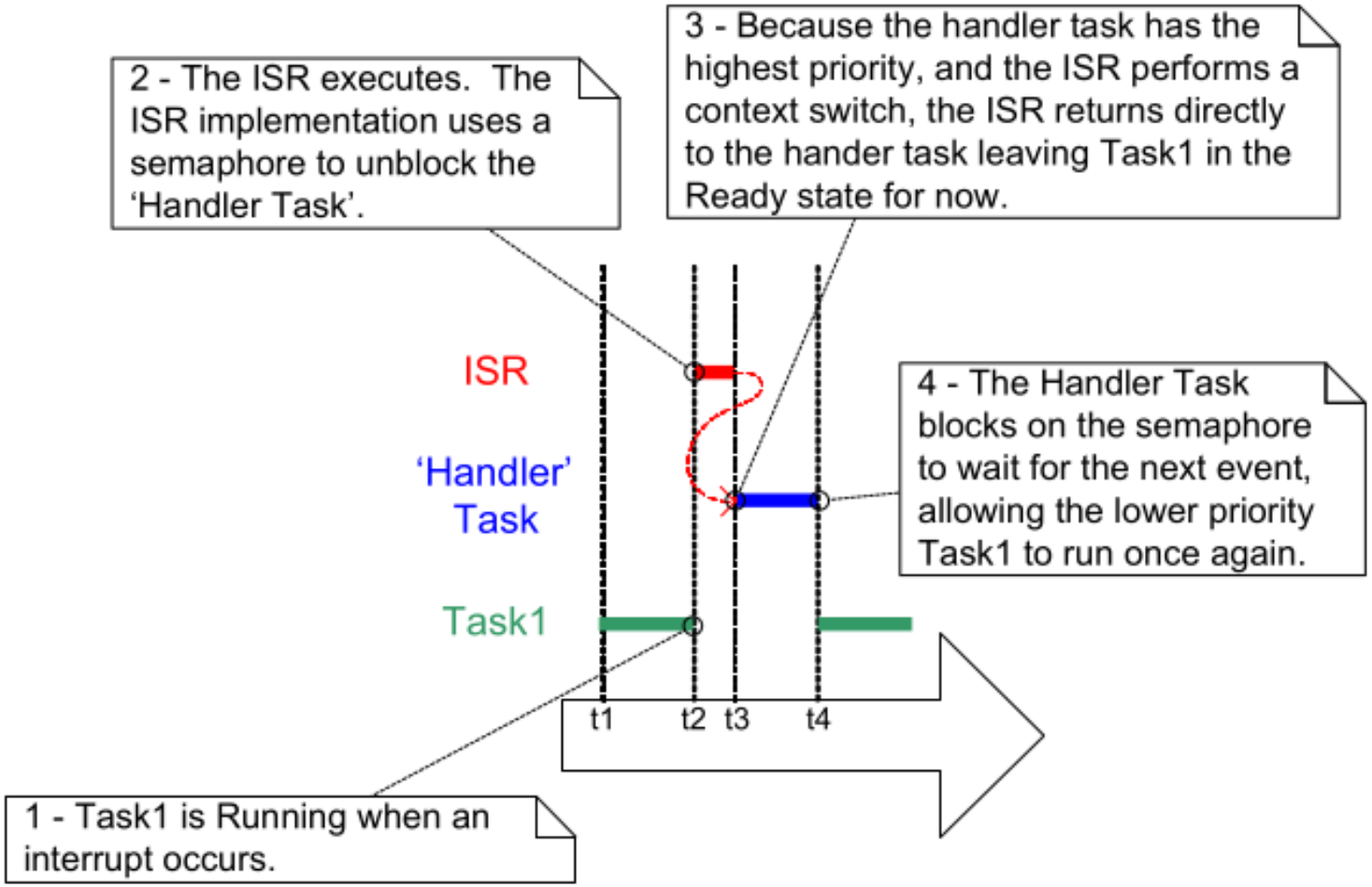



Figure: Deferred interrupt

# Semaphore

The handler task uses a blocking ‘take’ call to a semaphore as a means of entering the Blocked state to wait for the event to occur. When the event occurs, the ISR uses a ‘give’ operation on the same semaphore to unblock the task so that the required event processing can proceed.

**semaphore**

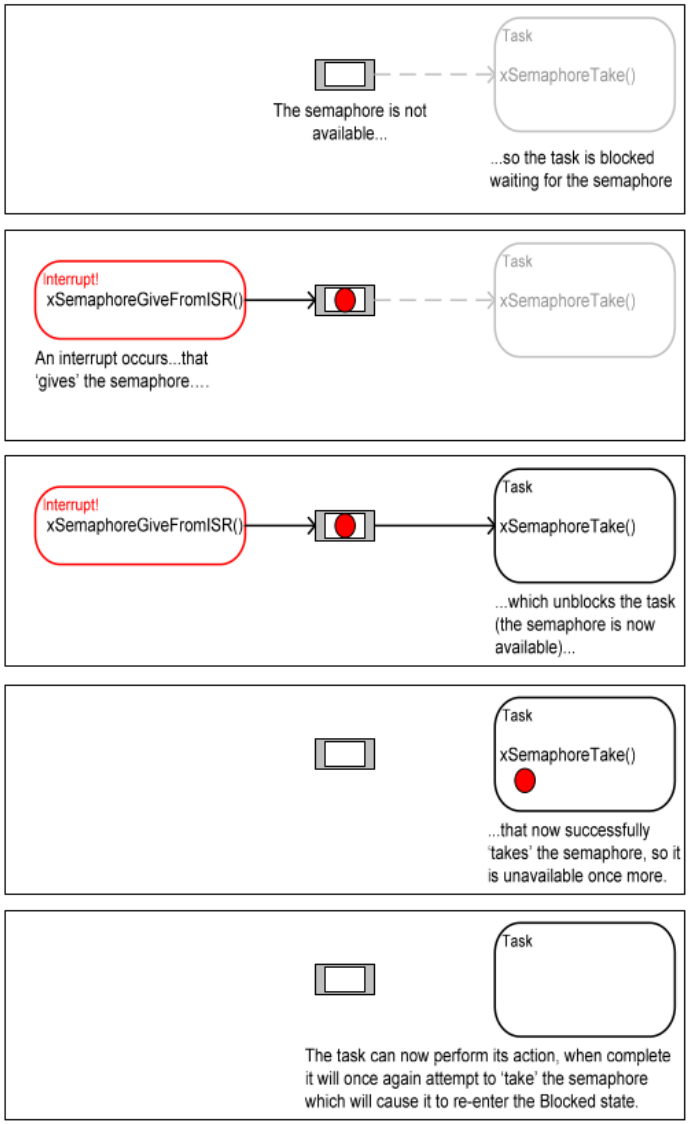
The binary semaphore can be considered conceptually as a queue with a length of one. By calling xSemaphoreTake(), the handler task effectively attempts to read from the queue with a block time, causing the task to enter the Blocked state if the queue is empty.

**From ISR**

When the event occurs, the ISR uses the xSemaphoreGiveFromISR() function to place a token (the semaphore) into the queue, making the queue full.

This causes the handler task to exit the Blocked state and remove the token, leaving the queue empty once more. When the handler task has completed its processing, it once more attempts to read from the queue and, finding the queue empty, re-enters the Blocked state to wait for the next event.

# Semaphore


Task
xSemaphoreTake()
The semaphore is not available...
...so the task is blocked waiting for the semaphore
Interrupt!
xSemaphoreGiveFromISR()
Task
xSemaphoreTake()
An interrupt occurs...that 'gives' the semaphore....
Interrupt!
xSemaphoreGiveFromISR()
Task
xSemaphoreTake()
...which unblocks the task (the semaphore is now available)...
Task
xSemaphoreTake()
...that now successfully 'takes' the semaphore, so it is unavailable once more.
Task
The task can now perform its action, when complete it will once again attempt to 'take' the semaphore which will cause it to re-enter the Blocked state.

# FreeRTOS interrupt handlers

The Cortex-M3 architecture and FreeRTOS port both permit ISRs to be written entirely in C, even when the ISR wants to cause a context switch.

# 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);
```

SemaphoreHandle_t The semaphore being created.

Returns NULL if the semaphore could not be created because there was insufficient FreeRTOS heap available.

# xSemaphoreTake()

'Taking' a semaphore means to 'obtain' or 'receive' the semaphore. The semaphore can be taken only if it is available (=P()).

**ISR**

xSemaphoreTake() must not be used from an interrupt service routine.

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

# xSemaphoreTake()

xSemaphore The semaphore being 'taken'.

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 (0=immediately, portMAX_DELAY=forever).

return pdPASS is returned only if the call to xSemaphoreTake() was successful in obtaining the semaphore. pdFALSE=The semaphore is not available.

# xSemaphoreGiveFromISR()

xSemaphoreGiveFromISR() is a special form of xSemaphoreGive() that is specifically for use within an interrupt service routine.

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

xSemaphore The semaphore being 'given'.

# xSemaphoreGiveFromISR()

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 such a task 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 or equal to the currently executing task (the task that was interrupted), then, internally, xSemaphoreGiveFromISR() will set pxHigherPriorityTaskWoken to pdTRUE.

# xSemaphoreGiveFromISR()

pxHigherPriorityTaskWoken If xSemaphoreGiveFromISR() 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.

return pdPASS will be returned only if the call to xSemaphoreGiveFromISR() is successful. pdFAIL if a semaphore is already available, it cannot be given

# Example 12

This example uses a binary semaphore to unblock a task from within an interrupt service routine—effectively synchronizing the task with the interrupt.
A simple periodic task is used to generate an interrupt every 500 milliseconds. In this case, a software generated interrupt is used because it allows the time at which the interrupt occurs to be controlled, which in turn allows the sequence of execution to be observed more easily. Next slides show the implementation of the periodic task. mainTRIGGER_INTERRUPT() simply sets a bit in the interrupt controller's Set Pending register.

# Example 12: periodic task

```
static void vPeriodicTask( void *pvParameters )
{
    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* This task is just used to 'simulate' an interrupt. This is done by
        periodically generating a software interrupt. */
        vTaskDelay( 500 / portTICK_RATE_MS );
        /* Generate the interrupt, printing a message both before hand and
        afterwards so the sequence of execution is evident from the output. */
        vPrintString( "Periodic task - About to generate an interrupt.\n" );
        mainTRIGGER_INTERRUPT();
        vPrintString( "Periodic task - Interrupt generated.\n\n" );
    }
}
```

# Example 12: handler task

```
static void vHandlerTask( void *pvParameters )
{
    /* As per most tasks, this task is implemented wit
    hin an infinite loop.
    Take the semaphore once to start with so the semaphore is empty before the
    infinite loop is entered. The semaphore was created before the scheduler
    was started so before this task ran for the first time.*/
    xSemaphoreTake( xBinarySemaphore, 0 );
    for( ;; )
    {
        /* Use the semaphore to wait for the event. The task blocks
        indefinitely meaning this function call will only return once the
        semaphore has been successfully obtained - so there is no need to check
        the returned value. */
        xSemaphoreTake( xBinarySemaphore, portMAX_DELAY );
        /* To get here the event must have occurred. Process the event (in this
        case we just print out a message). */
        vPrintString( "Handler task - Processing event.\n" );
    }
}
```

# Example 12: portEND_SWITCHING_ISR

The macro portEND_SWITCHING_ISR() is part of the FreeRTOS Cortex-M3 port and is the ISR safe equivalent of taskYIELD(). It will force a context switch only if its parameter is not zero (not equal to pdFALSE).

Use portYIELD_FROM_ISR(xHigherPriorityTaskWoken) to simplify the code and to use the latest coding style.

## Where to use

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.

# Switch to the higher priority task

When the switch to the higher priority task actually occurs is dependent on the context from which the API function is called:

- **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, in other words, before the API function has exited.
- **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.

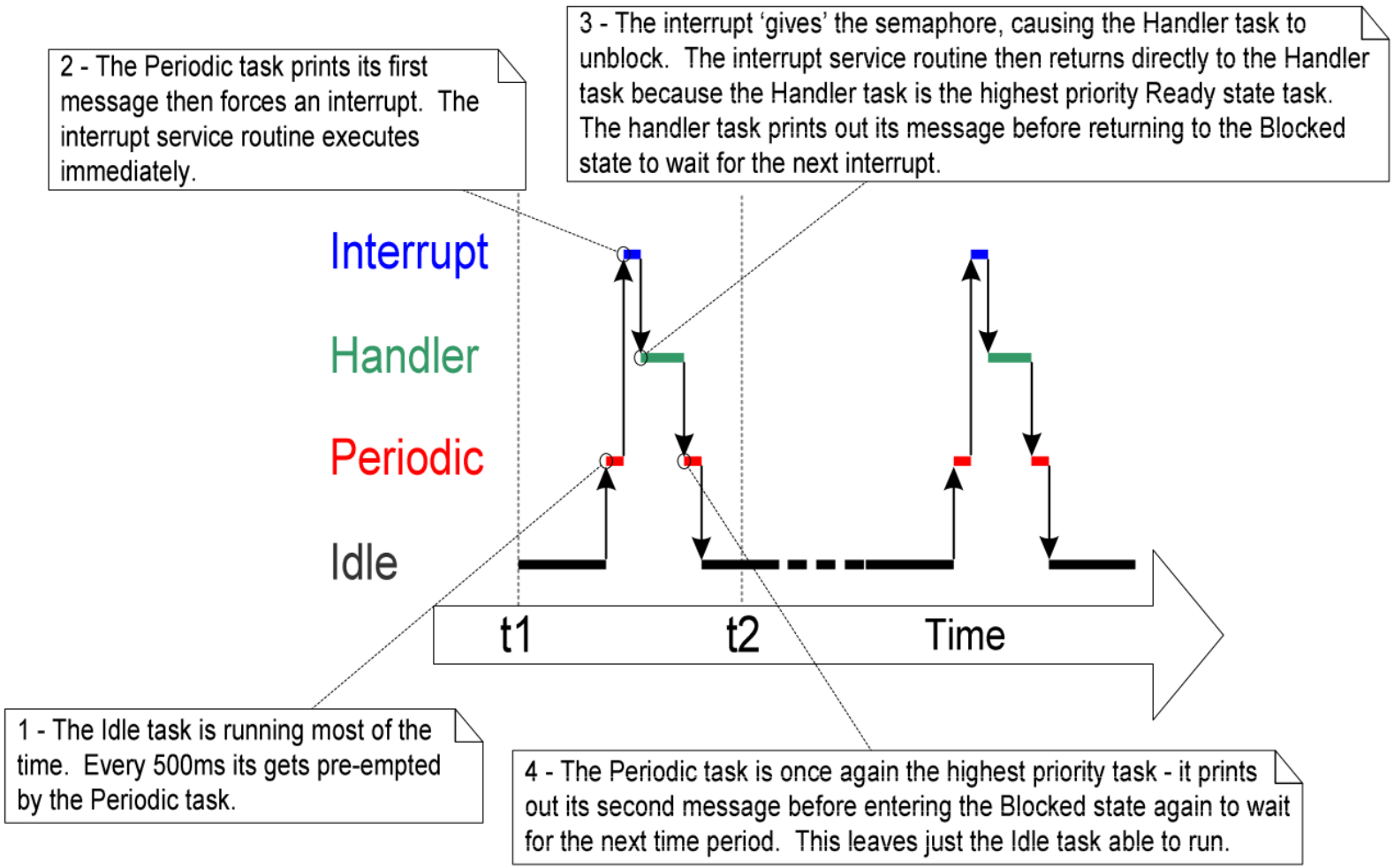
# Example 12: interrupt handler

```
void vSoftwareInterruptHandler( void )
{
    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
    /* 'Give' the semaphore to unblock the task. */
    xSemaphoreGiveFromISR( xBinarySemaphore, &xHigherPriorityTaskWoken );
    /* Clear the software interrupt bit using the interrupt controllers
    Clear Pending register. */
    mainCLEAR_INTERRUPT();
    /* Giving the semaphore may have unblocked a task - if it did and the
    unblocked task has a priority equal to or above the currently executing
    task then xHigherPriorityTaskWoken will have been set to pdTRUE and
    portEND_SWITCHING_ISR() will force a context switch to the newly unblocked
    higher priority task.
    NOTE: The syntax for forcing a context switch within an ISR varies between
    FreeRTOS ports. The portEND_SWITCHING_ISR() macro is provided as part of
    the Corte M3 port layer for this purpose. taskYIELD() must never be called
    from an ISR! */
    portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
}
```

# Example 12: main

```
int main( void )
{
    /* Configure both the hardware and the debug interface. */
    vSetupEnvironment();
    /* Before a semaphore is used it must be explicitly created.
    a binary semaphore is created. */
    vSemaphoreCreateBinary( xBinarySemaphore );
    /* Check the semaphore was created successfully. */
    if( xBinarySemaphore != NULL )
    {
        /* Enable the software interrupt and set its priority. */
        prvSetupSoftwareInterrupt();
        xTaskCreate( vHandlerTask, "Handler", 240, NULL, 3, NULL );
        xTaskCreate( vPeriodicTask, "Periodic", 240, NULL, 1, NULL );
        /* Start the scheduler so the created tasks start executing. */
        vTaskStartScheduler();
    }
    /* If all is well we will never reach here as the scheduler will now be
    running the tasks. If we do reach here then it is likely that there was
    insufficient heap memory available for a resource to be created. */
    for( ;; );
}
```

# Example 12: timing diagram



# Counting semaphore

Example 12 demonstrates a binary semaphore:

- An interrupt occurs.
- The interrupt service routine executes, 'giving' the semaphore to unblock the Handler task.
- The Handler task executes as soon as the interrupt completes. The first thing the Handler task does is 'take' the semaphore.
- The Handler task processes the event before attempting to 'take' the semaphore again—entering the Blocked state if the semaphore is not immediately available.

# Counting semaphore

**This sequence is perfectly adequate if interrupts can occur only at a relatively low frequency**. If another interrupt occurs before the Handler task has completed its processing of the first interrupt, then the binary semaphore will effectively latch the event, allowing the Handler task to process the new event immediately after it has completed processing the original event.

# Counting semaphore

**This sequence is perfectly adequate if interrupts can occur only at a relatively low frequency**. If another interrupt occurs before the Handler task has completed its processing of the first interrupt, then the binary semaphore will effectively latch the event, allowing the Handler task to process the new event immediately after it has completed processing the original event. What does it happen if the interrupt is faster?

# Counting semaphore

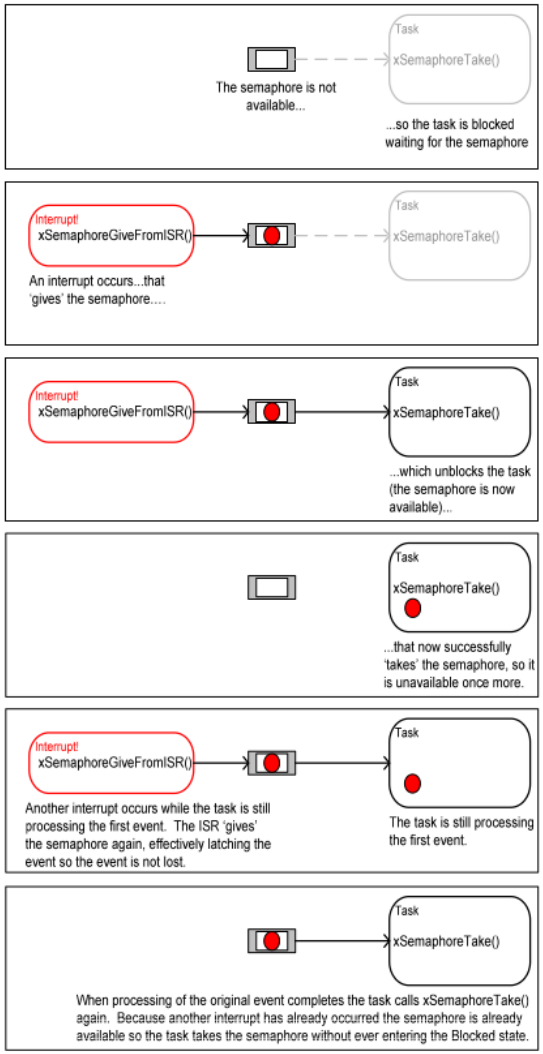



Figure: Low frequency

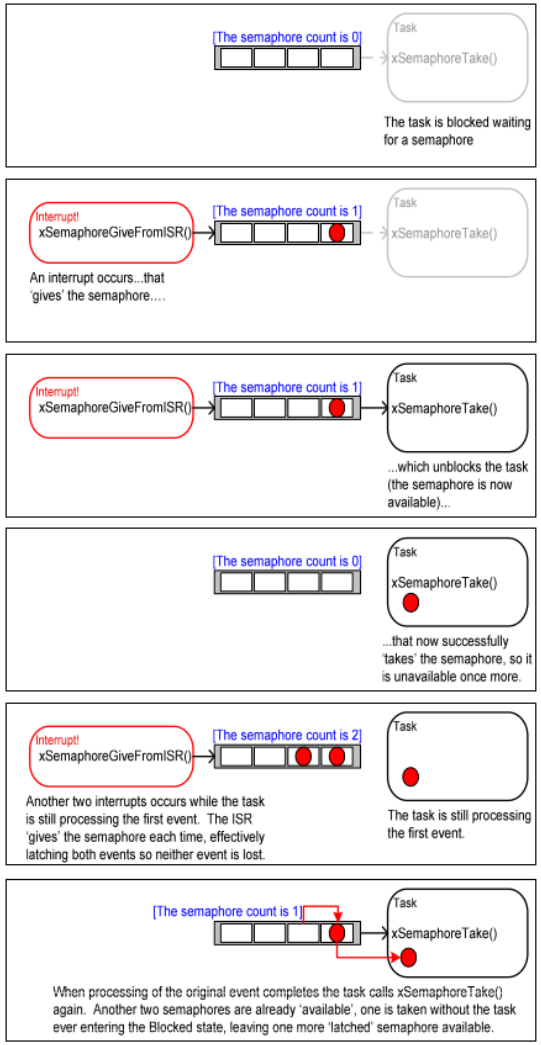



Figure: High frequency

# Counting semaphore

A binary semaphore can latch, at most, one interrupt event. Any subsequent events, occurring before the latched event has been processed, will be lost.

# Counting semaphore: solution

Counting semaphores are typically used for two things:

- Counting events
- Resource management

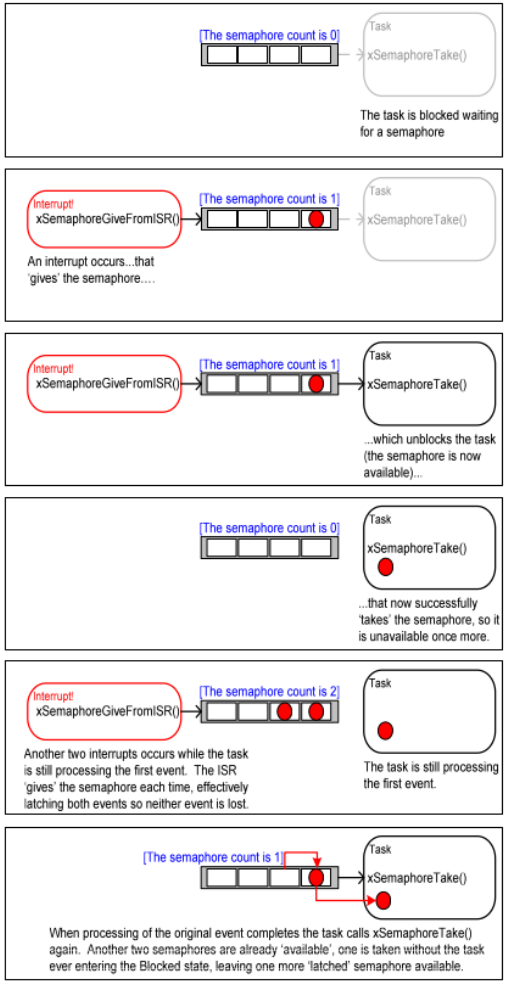
# Counting events

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 handler task will 'take' a semaphore each time it processes an event—causing the semaphore's count value to be decremented on each take.

# Resource management

In this usage 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 semaphore: solution



# xSemaphoreCreateCounting()

```
SemaphoreHandle_t xSemaphoreCreateCounting(unsigned portBASE_TYPE uxMaxCount,
                                           unsigned portBASE_TYPE uxInitialCount);
```

uxMaxCount The maximum value the semaphore will count to. To continue the queueanalogy, the uxMaxCount value is effectively the length of the queue.

uxInitialCount The initial count value of the semaphore after it has been created.

return If NULL is returned, the semaphore cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the semaphore data structures.

# Example 13: main

It improves on the Example 12 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().

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

# Example 13: interrupt handler

```
void vSoftwareInterruptHandler( void )
{
    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
    /* 'Give' the semaphore multiple times. The first will unblock the handler
    task, the following 'gives' are to demonstrate that the semaphore latches
    the events to allow the handler task to process them in turn without any
    events getting lost. This simulates multiple interrupts being taken 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 );
    /* Clear the software interrupt bit using the interrupt controllers Clear
    Pending register. */
    mainCLEAR_INTERRUPT();
    /* Giving the semaphore may have unblocked a task - if it did and the
    unblocked task has a priority equal to or above the currently executing
    task then xHigherPriorityTaskWoken will have been set to pdTRUE and
    portEND_SWITCHING_ISR() will force a context switch to the newly unblocked
    higher priority task.
    NOTE: The syntax for forcing a context switch within an ISR varies between
    FreeRTOS ports. The portEND_SWITCHING_ISR() macro is provided as part of
    the Cortex-M3 port layer for this purpose. taskYIELD() must never be called
    from an ISR! */
    portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
}
```

# Using queue within ISR

**xQueueSendToFrontFromISR()**, **xQueueSendToBackFromISR()** and **xQueueReceiveFromISR()** are versions of xQueueSendToFront(), xQueueSendToBack() and xQueueReceive(), respectively, that are safe to use within an interrupt service routine.

**Queue vs semaphore**

Semaphores are used to communicate events. Queues are used to communicate events and to transfer data.

# xQueueSendToFrontFromISR() and xQueueSendToBackFromISR()

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

xQueue The handle of the queue to which the data is being sent.

pvItemToQueue A pointer to the data to be copied into the queue.

# Using queue within ISR

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 equal to or higher than the currently executing task (the task that was interrupted), then, internally, the API function will set *pxHigherPriorityTaskWoken to pdTRUE.

# Using queue within ISR

return pdPASS is returned only if data has been sent successfully to the queue.
errQUEUE_FULL is returned if data cannot be sent to the queue because the queue is already full.

# Efficient queue usage

Most of the demo applications in the FreeRTOS download include a simple UART driver that uses queues to pass characters into the transmit interrupt handler and out of the receive interrupt handler. Every character that is transmitted or received gets passed individually through a queue. The UART drivers are implemented in this manner purely as a convenient way of demonstrating queues being used from within interrupts. **Passing individual characters through a queue is extremely inefficient (especially at high baud rates) and is not recommended for production code.**

# Efficient queue usage

More efficient techniques include:

- Placing each received character in a simple RAM buffer, then using a semaphore to unblock a task to process the buffer.
- Interpreting the received characters directly within the interrupt service routine, then using a queue to send the interpreted and decoded commands to a task for processing. This technique is suitable only if interpreting the data stream is quick enough to be performed entirely from within an interrupt.

# Example 14: generator

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.

```
static void vIntegerGenerator( void *pvParameters )
{
    portTickType xLastExecutionTime;
    unsigned long ulValueToSend = 0;
    int i;
    xLastExecutionTime = xTaskGetTickCount();
    for( ;; )
    {
        /* This is a periodic task. Block until it is time to run again.
        The task will execute every 200ms. */
        vTaskDelayUntil( &xLastExecutionTime, 200 / portTICK_RATE_MS );
        for( i = 0; i < 5; i++ )
        {
            xQueueSendToBack( xIntegerQueue, &ulValueToSend, 0 );
            ulValueToSend++;
        }
        /* Force an interrupt so the interrupt service routine can read the
        values from the queue. */
        vPrintString( "Generator task - About to generate an interrupt.\n" );
        mainTRIGGER_INTERRUPT();
        vPrintString( "Generator task - Interrupt generated.\n\n" );
    }
}
```

# Example 14: interrupt handler

```
void vSoftwareInterruptHandler( void )
{
    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
    static unsigned long ulReceivedNumber;
    /* The strings are declared static const to ensure they are not allocated to the
    interrupt service routine stack, and exist even when the interrupt service routine
    is not executing. */
    static const char *pcStrings[] =
    {
        "String 0\n", "String 1\n",
        "String 2\n", "String 3\n"
    };
    while( xQueueReceiveFromISR(xIntegerQueue,
                                &ulReceivedNumber,
                                &xHigherPriorityTaskWoken ) != errQUEUE_EMPTY )
    {
    /* Truncate the received value to the last two bits (values 0 to 3 inc.),
    then send the string that corresponds to the truncated value to the other
    queue. */
        ulReceivedNumber &= 0x03;
        xQueueSendToBackFromISR(xStringQueue,
                                &pcStrings[ ulReceivedNumber ],
                                &xHigherPriorityTaskWoken );
    }
    /* Clear the software interrupt bit using the interrupt controllers Clear
    Pending register. */
    mainCLEAR_INTERRUPT();
    portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
}
```

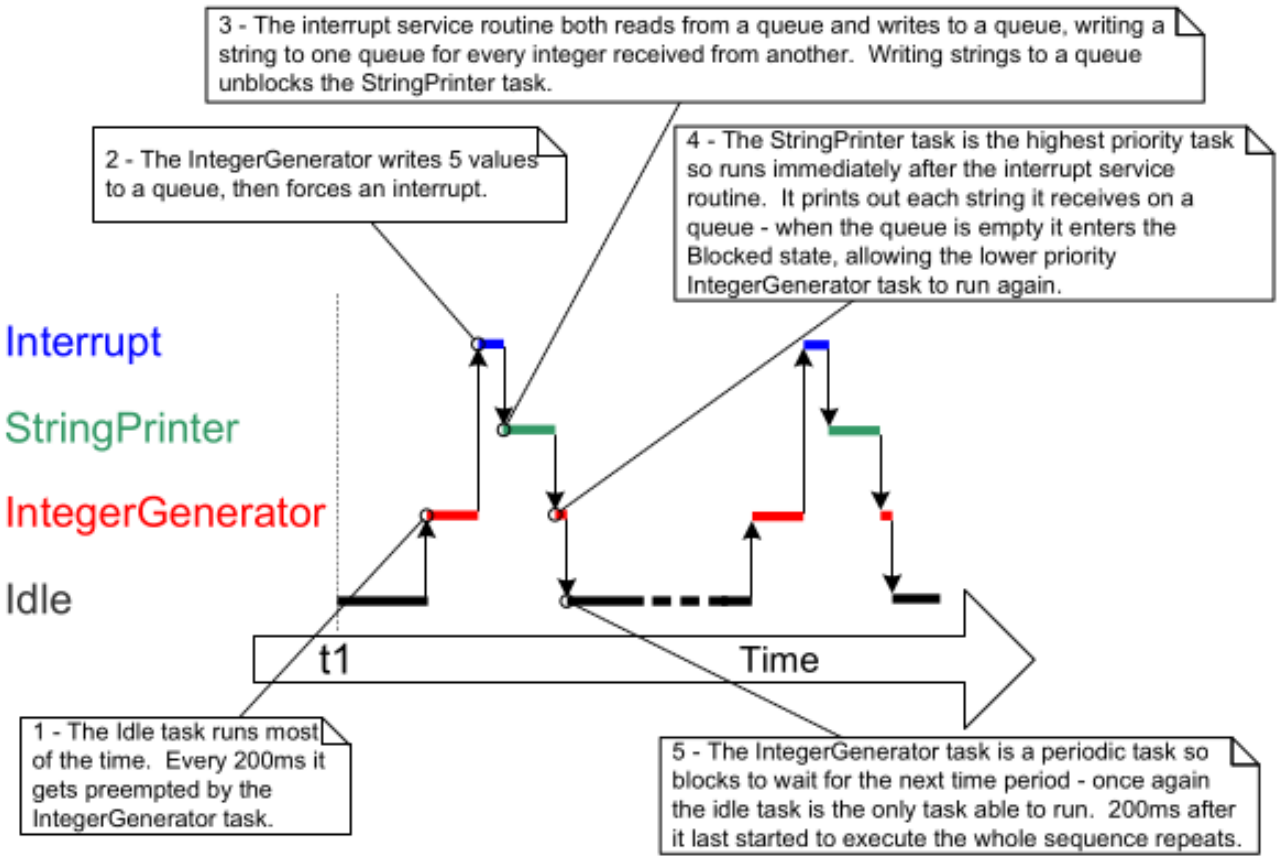
# Example 14: printer task

```
static void vStringPrinter( void *pvParameters )
{
    char *pcString;
    for( ;; )
    {
        /* Block on the queue to wait for data to arrive. */
        xQueueReceive( xStringQueue, &pcString, portMAX_DELAY );
        /* Print out the string received. */
        vPrintString( pcString );
    }
}
```

# Example 14: main

```
int main( void )
{
    /* Before a queue can be used it must first be created. Create both queues
    used by this example. One queue can hold variables of type unsigned long,
    the other queue can hold variables of type char*. Both queues can hold a
    maximum of 10 items. A real application should check the return values to
    ensure the queues have been successfully created. */
    xIntegerQueue = xQueueCreate( 10, sizeof( unsigned long ) );
    xStringQueue = xQueueCreate( 10, sizeof( char * ) );
    /* Enable the software interrupt and set its priority. */
    prvSetupSoftwareInterrupt();
    /* Create the task that uses a queue to pass integers to the interrupt service
    routine. The task is created at priority 1. */
    xTaskCreate( vIntegerGenerator, "IntGen", 240, NULL, 1, NULL );
    /* Create the task that prints out the strings sent to it from the interrupt
    service routine. This task is created at the higher priority of 2. */
    xTaskCreate( vStringPrinter, "String", 240, NULL, 2, NULL );
    /* Start the scheduler so the created 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 heap memory available for the idle task to be created.
    */
    for( ;; );
}
```

# Example 14: timing



# Interrupt nesting

It is common for confusion to arise between task priorities and interrupt priorities.

- Interrupt priorities 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.**

# Interrupt nesting

Ports that support interrupt nesting require one or both of the constants detailed below to be defined in FreeRTOSConfig.h.

- configMAX_SYSCALL_INTERRUPT_PRIORITY and 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 (Tick and PendSV).

# Interrupt nesting

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.

# Interrupt nesting

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.**

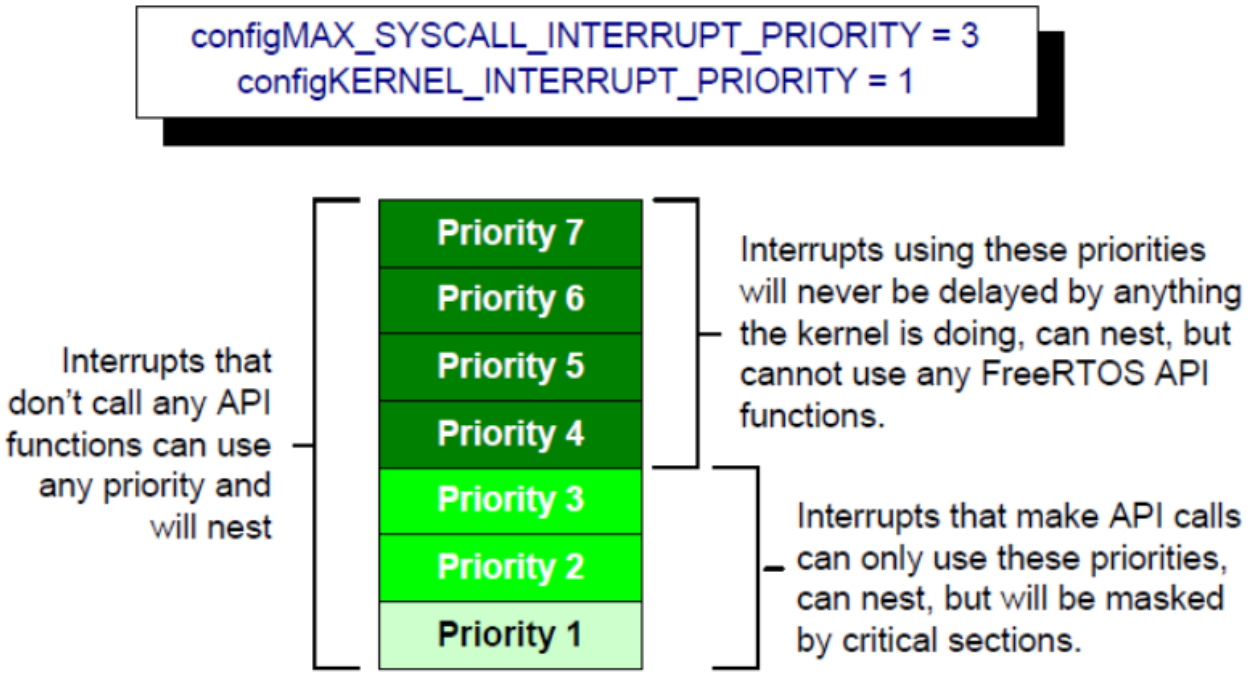
`https://developer.arm.com/documentation/ihi0048/b/Interrupt-Handling-and-Prioritization/Interrupt-prioritization/Priority-grouping?lang=en`

# Interrupt nesting

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.

Check the logical and numeric priority in STMCubeIDE FreeRTOS.

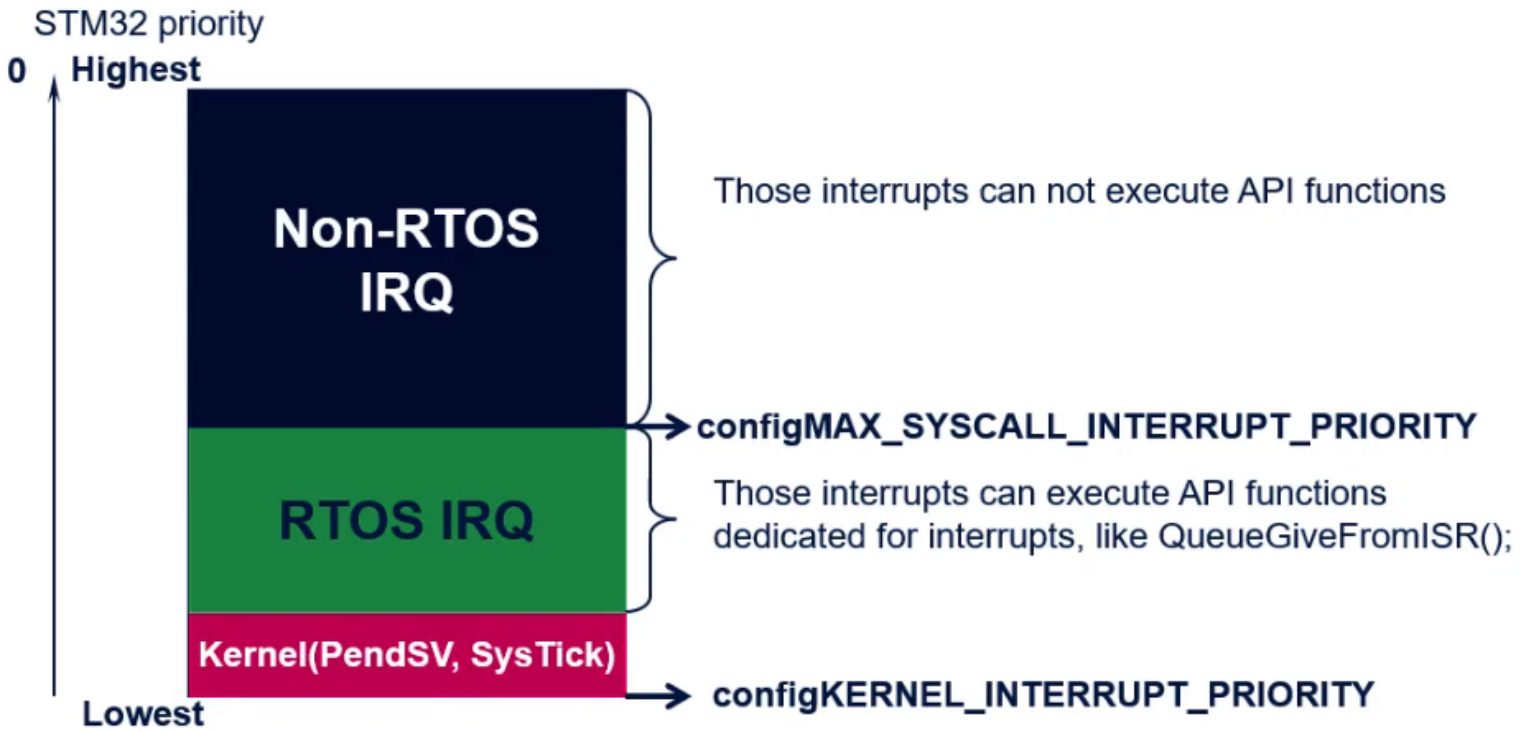
# Interrupt nesting



# Interrupt nesting

- 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.

# Interrupt nesting in STM32



# Resource access and conflicts

In a multitasking system, there is potential for conflict if one task starts to access a resource, but does not complete its access before being transitioned out of the Running state. If the task leaves the resource in an inconsistent state, then access to the same resource by any other task or interrupt could result in data corruption or other similar error.

## Remarks

The same behaviour you can find in the posix pthread environment, where you can have a concurrent access to a shared resources.

# Resource management: examples

- Accessing peripherals
- Read, Modify, Write Operations
- Non atomic access to variables
- **Function reentrancy**: a function is reentrant if it is safe to call the function from more than one task, or from both tasks and interrupts.

# Resource management: reentrant

```
/* A parameter is passed into the function. This will either be
passed on the stack or in a CPU register. Either way is safe as
each task maintains its own stack and its own set of register
values. */
long lAddOneHundered( long lVar1 )
{
    /* This function scope variable will also be allocated to the stack
    or a register, depending on the compiler and optimization level. Each
    task or interrupt that calls this function will have its own copy
    of lVar2. */
    long lVar2;
    lVar2 = lVar1 + 100;
    /* Most likely the return value will be placed in a CPU register,
    although it too could be placed on the stack. */
    return lVar2;
}
```

# Resource management: non reentrant

```
/* In this case lVar1 is a global variable so every task that calls
the function will be accessing the same single copy of the variable. */
long lVar1;
long lNonsenseFunction( void )
{
    /* This variable is static so is not allocated on the stack. Each task
    that calls the function will be accessing the same single copy of the
    variable. */
    static long lState = 0;
    long lReturn;
    switch( lState )
    {
        case 0 : lReturn = lVar1 + 10;
        lState = 1;
        break;
        case 1 : lReturn = lVar1 + 20;
        lState = 0;
        break;
    }
}
```

# Mutual Exclusion

Access to a resource that is shared between tasks, or between tasks and interrupts, must be managed using a 'mutual exclusion' technique, to ensure that data consistency is maintained at all times. The goal is to ensure that, once a task starts to access a shared resource, the same task has exclusive access until the resource has been returned to a consistent state.

FreeRTOS provides several features that can be used to implement mutual exclusion, but the best mutual exclusion method is to (whenever possible) **design the application in such a way that resources are not shared and each resource is accessed only from a single task**.

# Management of mutual exclusion

- Critical sections
- Suspending the scheduler
- Mutexes
- Gatekeeper

# Critical sections

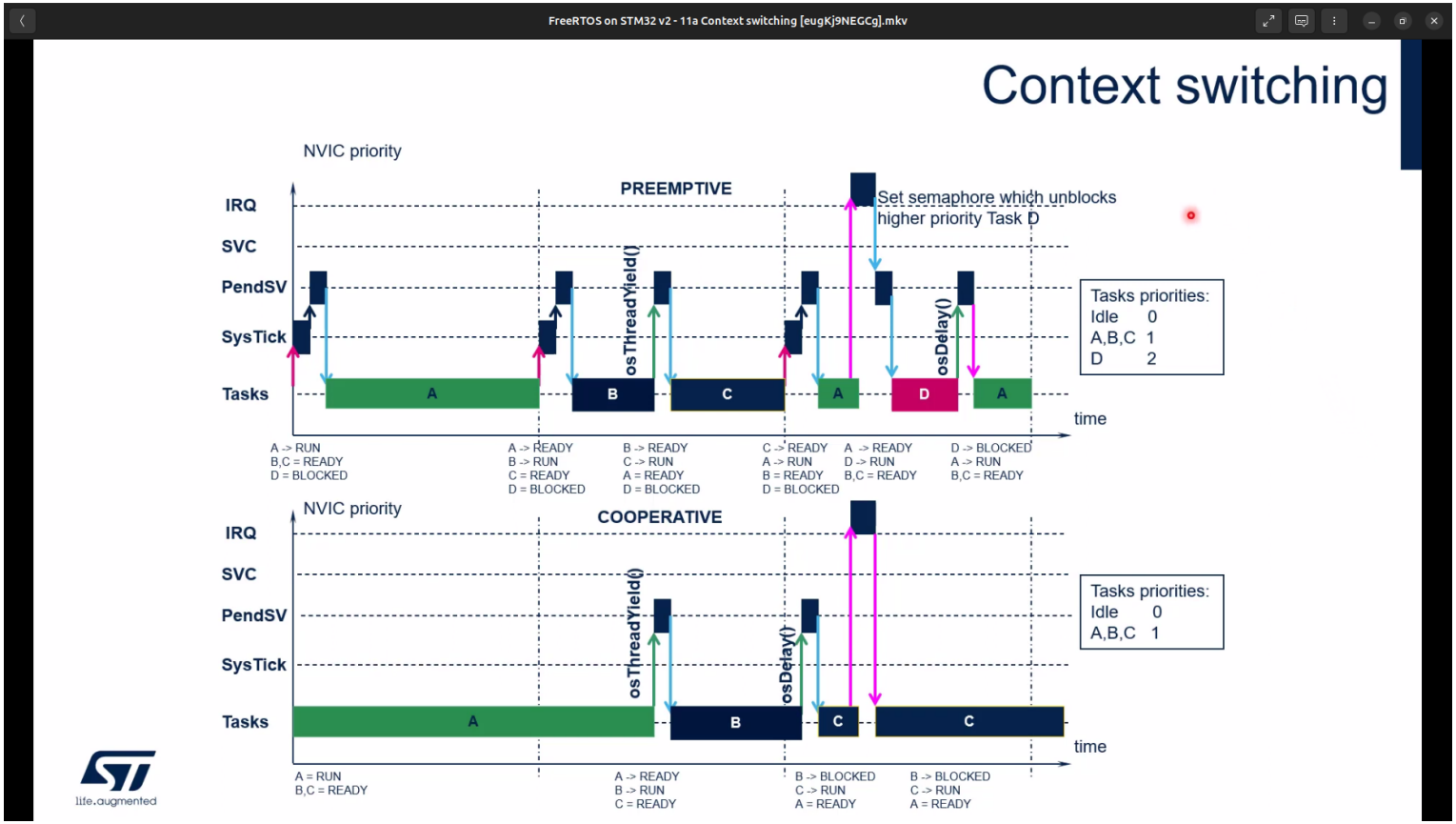
Basic critical sections are regions of code that are surrounded by calls to the macros taskENTER_CRITICAL() and taskEXIT_CRITICAL(), respectively. Critical sections are also known as critical regions.

```
/* Ensure access to the GlobalVar variable cannot be interrupted by
placing it within a critical section. Enter the critical section. */
taskENTER_CRITICAL();
/* A switch to another task cannot occur between the call to
taskENTER_CRITICAL() and the call to taskEXIT_CRITICAL(). Interrupts
may still execute, but only interrupts whose priority is above the
value assigned to the configMAX_SYSCALL_INTERRUPT_PRIORITY constant
- and those interrupts are not permitted to call FreeRTOS API
functions. */
GlobalVar |= 0x01;
/* Access to GlobalVar is complete so the critical section can be exited. */
taskEXIT_CRITICAL();
```

# Critical sections

Critical sections implemented in this way are a very crude method of providing mutual exclusion. **They work by disabling interrupts up to the interrupt priority set by configMAX_SYSCALL_INTERRUPT_PRIORITY.** Pre-emptive context switches can occur only from within an interrupt, so, as long as interrupts remain disabled, the task that called taskENTER_CRITICAL() is guaranteed to remain in the Running state until the critical section is exited.

# Context switching and interrupts



# Critical sections

**It is safe** for critical sections to become nested, because the kernel keeps a count of the nesting depth. The critical section will be exited only when the nesting depth returns to zero— which is when one call to taskEXIT_CRITICAL() has been executed for every preceding call to taskENTER_CRITICAL().

# Suspending the scheduler

Critical sections can also be created by suspending the scheduler. Suspending the scheduler is sometimes also known as 'locking' the scheduler.

```
void vTaskSuspendAll( void );
```

A critical section that is too long to be implemented by simply disabling interrupts can, instead, be implemented by suspending the scheduler. **However, resuming (or 'un-suspending') the scheduler can be a relatively long operation**, so consideration must be given to which is the best method to use in each case.

# Suspending the scheduler

```
portBASE_TYPE xTaskResumeAll( void );
```

return Context switches that are requested while the scheduler is suspended are held pending and performed only as the scheduler is being resumed. A previously pending context switch being performed before xTaskResumeAll() returns results in the function returning pdTRUE. In all other cases, xTaskResumeAll() returns pdFALSE.

It is safe for calls to vTaskSuspendAll() and xTaskResumeAll() to become nested, because the kernel keeps a count of the nesting depth.

# Mutexes

A Mutex is a special type of binary semaphore that is used to control access to a resource that is shared between two or more tasks. The word MUTEX originates from ‘MUTual EXclusion’.

# Mutexes

A Mutex is a special type of binary semaphore that is used to control access to a resource that is shared between two or more tasks. The word MUTEX originates from 'MUTual EXclusion'. When used in a mutual exclusion scenario, the mutex can be thought of as a token that is associated with the resource being shared. For a task to access the resource legitimately, it must first successfully **'take'** the token (be the token holder). When the token holder has finished with the resource, it must **'give'** the token back. Only when the token has been returned can another task successfully take the token and then safely access the same shared resource.

# Mutexes

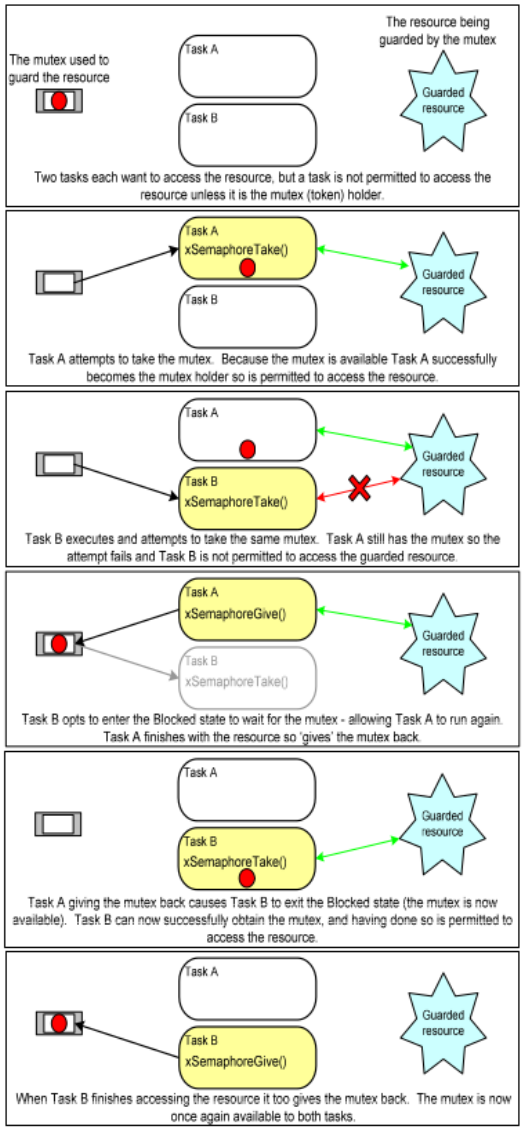
The primary difference is what happens to the semaphore after it has been obtained:

- **A semaphore that is used for mutual exclusion must always be returned (ownership!).**
- A semaphore that is used for synchronization is normally discarded and not returned.

**Interrupts?**

Mutexes should not be used from an interrupt because: They include a priority inheritance mechanism which only makes sense if the mutex is given and taken from a task, not an interrupt. An interrupt cannot block to wait for a resource that is guarded by a mutex to become available.

# Mutual exclusion with mutex



# xSemaphoreCreateMutex()

A mutex is a type of semaphore. Handles to all the various types of FreeRTOS semaphore are stored in a variable of type SemaphoreHandle_t.

```
SemaphoreHandle_t xSemaphoreCreateMutex( void );
```

return If NULL is returned, then the mutex could not be created because there is insufficient heap memory available for FreeRTOS to allocate the mutex data structures.

# Example 15: print function

```
static void prvNewPrintString( const char *pcString )
{
    static char cBuffer[ mainMAX_MSG_LEN ];
    /* The mutex is created before the scheduler is started so already
    exists by the time this task first executes.
    Attempt to take the mutex, blocking indefinitely to wait for the mutex if
    it is not available straight away. The call to xSemaphoreTake() will only
    return when the mutex has been successfully obtained so there is no need to
    check the function return value. If any other delay period was used then
    the code must check that xSemaphoreTake() returns pdTRUE before accessing
    the shared resource (which in this case is standard out). */
    xSemaphoreTake( xMutex, portMAX_DELAY );
    {
        /* The following line will only execute once the mutex has been
        successfully obtained. Standard out can be accessed freely now as
        only one task can have the mutex at any one time. */
        sprintf( cBuffer, "%s", pcString );
        consoleprint( cBuffer );
        /* The mutex MUST be given back! */
    }
    xSemaphoreGive( xMutex );
}
```

# Example 15: print task

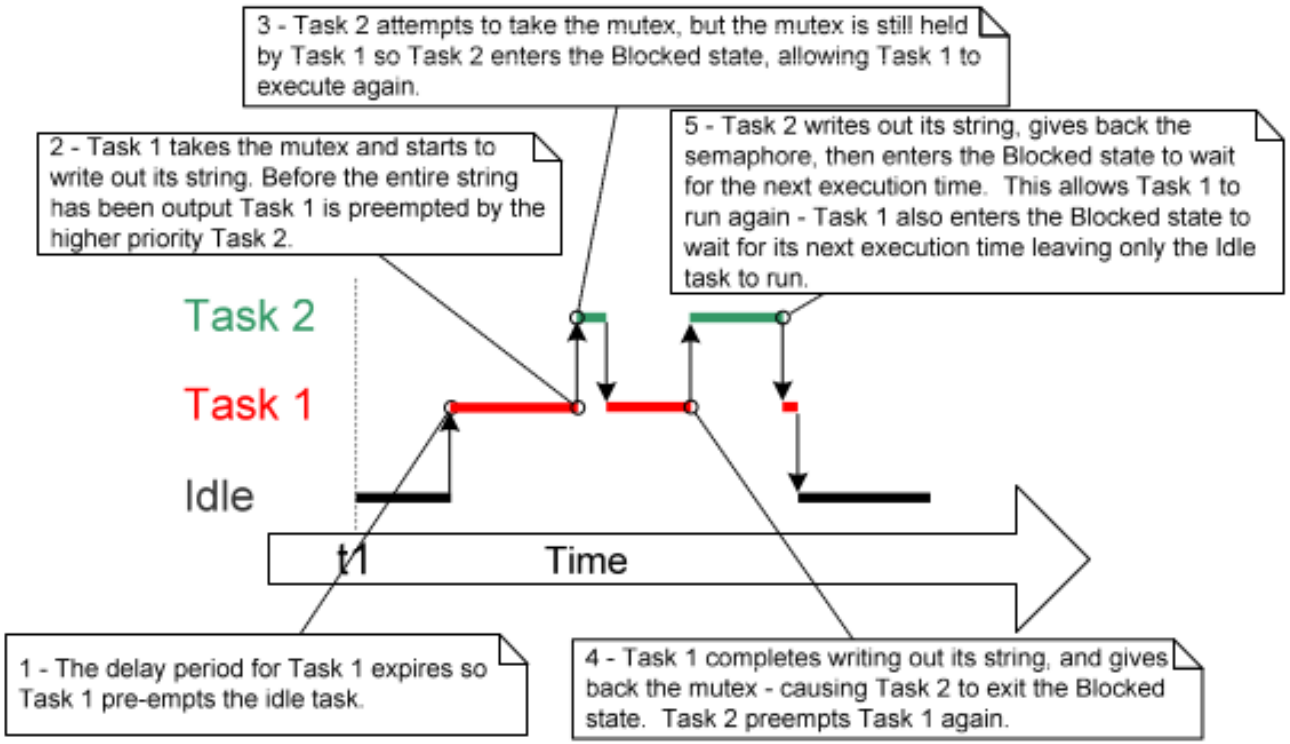
```
static void prvPrintTask( void *pvParameters )
{
    char *pcStringToPrint;
    /* Two instances of this task are created so the string the task will send
    to prvNewPrintString() is passed into the task using the task parameter.
    Cast this to the required type. */
    pcStringToPrint = ( char * ) pvParameters;
    for( ;; )
    {
        /* Print out the string using the newly defined function. */
        prvNewPrintString( pcStringToPrint );
        /* Wait a pseudo random time. Note that rand() is not necessarily
        reentrant, but in this case it does not really matter as the code does
        not care what value is returned. In a more secure application a version
        of rand() that is known to be reentrant should be used - or calls to
        rand() should be protected using a critical section. */
        vTaskDelay( ( rand() & 0x1FF ) );
    }
}
```

# Example 15: main

```
int main( void )
{
    /* Before a semaphore is used it must be explicitly created.
    a mutex type semaphore is created. */
    xMutex = xSemaphoreCreateMutex();
    /* The tasks are going to use a pseudo random delay, seed the random number
    generator. */
    srand( 567 );
    /* Only create the tasks if the semaphore was created successfully. */
    if( xMutex != NULL )
    {
        /* Create two instances of the tasks that write to stdout. The string
        they write is passed in as the task parameter. The tasks are created
        at different priorities so some pre-emption will occur. */
        xTaskCreate( prvPrintTask, "Print1", 240,
        "Task 1 ****************************************\n", 1, NULL );
        xTaskCreate( prvPrintTask, "Print2", 240,
        "Task 2 ----------------------------------------\n", 2, NULL );
        /* Start the scheduler so the created 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 heap memory available for the idle task to be created.
    */
    for( ;; );
}
```

# Example 15: comments and timing

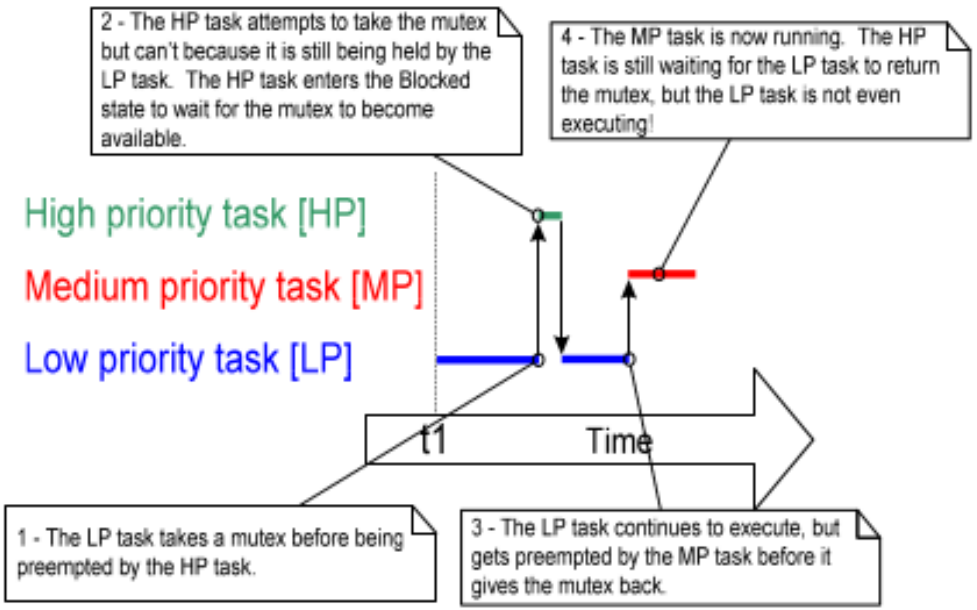
The two instances of prvPrintTask() are created at different priorities, so the lower priority task will sometimes be pre-empted by the higher priority task.



# Possible problems

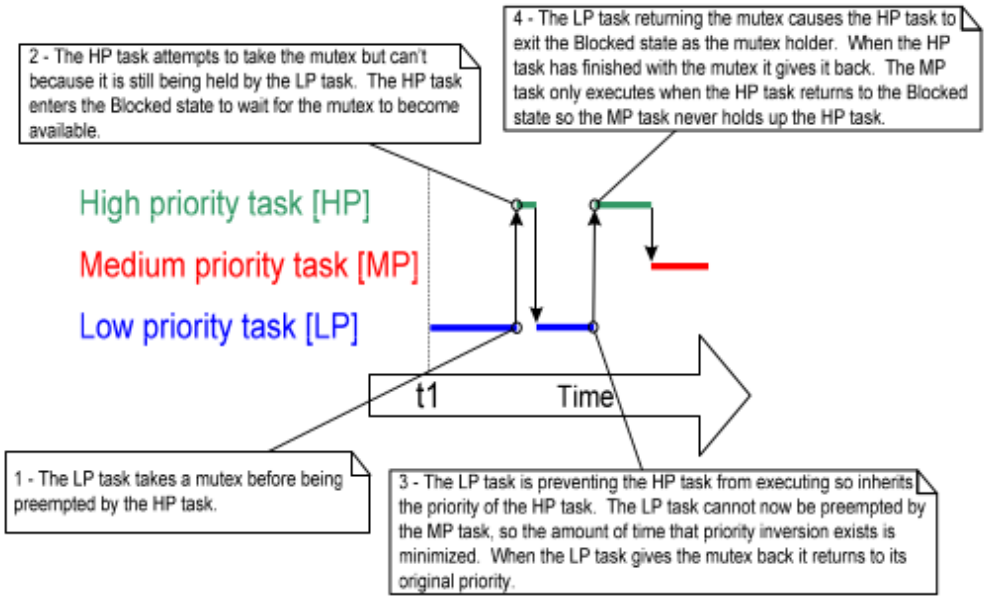
- Priority inversion
- Priority inheritance
- Deadlock

# Priority inversion




The figure demonstrates one of the potential pitfalls of using a mutex to provide mutual exclusion. The possible sequence of execution depicted shows the higher priority Task 2 having to wait for the lower priority Task 1 to give up control of the mutex. A higher priority task being delayed by a lower priority task in this manner is called 'priority inversion'.

# Priority inheritance




FreeRTOS mutexes and binary semaphores are very similar—the difference being that mutexes include a basic 'priority inheritance' mechanism, whereas binary semaphores do not.

The low priority task that holds the mutex 'inherits' the priority of the task waiting for the mutex.

# Deadlock

'Deadlock' is another potential pitfall that can occur when using mutexes for mutual exclusion.

- Task A executes and successfully takes mutex X.
- Task A is pre-empted by Task B.
- Task B successfully takes mutex Y before attempting to also take mutex X—but mutex X is held by Task A, so is not available to Task B. Task B opts to enter the Blocked state to wait for mutex X to be released.
- Task A continues executing. It attempts to take mutex Y—but mutex Y is held by Task B, so is not available to Task A. Task A opts to enter the Blocked state to wait for mutex Y to be released.

# Gatekeeper task

Gatekeeper tasks provide a clean method of implementing mutual exclusion without the risk of priority inversion or deadlock.
A gatekeeper task is a task that has sole ownership of a resource.
**Only the gatekeeper task is allowed to access the resource directly**—any other task requiring access to the resource can do so only indirectly by using the services of the gatekeeper.

# Example 16

Example 16 provides an alternative implementation for vPrintString(). This time, a gatekeeper task is used to manage access to standard out. When a task wants to write a message to the terminal, it does not call a print function directly but, instead, sends the message to the gatekeeper.

**The gatekeeper task uses a FreeRTOS queue to serialize access to the terminal. The internal implementation of the task does not have to consider mutual exclusion because it is the only task permitted to access the terminal directly.**

The gatekeeper task spends most of its time in the Blocked state, waiting for messages to arrive on the queue.

A tick hook (or tick callback) is a function that is called by the kernel during each tick interrupt.

# Example 16

```
static void prvStdioGatekeeperTask( void *pvParameters )
{
    char *pcMessageToPrint;
    static char cBuffer[ mainMAX_MSG_LEN ];
    /* This is the only task that is allowed to write to the terminal output.
    Any other task wanting to write a string to the output does not access the
    terminal directly, but instead sends the string to this task. As only this
    task accesses standard out there are no mutual exclusion or serialization
    issues to consider within the implementation of the task itself. */
    for( ;; )
    {
    /* Wait for a message to arrive. An indefinite block time is specified
    so there is no need to check the return value - the function will only
    return when a message has been successfully received. */
        xQueueReceive( xPrintQueue, &pcMessageToPrint, portMAX_DELAY );
    /* Output the received string. */
        sprintf( cBuffer, "%s", pcMessageToPrint );
        consoleprint( cBuffer );
    /* Now go back to wait for the next message. */
    }
}
```

The task that prints out the message is similar to that used in Example 15, except that, here, the string is sent on the queue to the gatekeeper task, rather than written out directly.

# Example 16

```
static void prvPrintTask( void *pvParameters )
{
    int iIndexToString;
    /* Two instances of this task are created. The task parameter is used to pass an
    index into an array of strings into the task. Cast this to the required type. */
    iIndexToString = ( int ) pvParameters;
    for( ;; )
    {
    /* Print out the string, not directly but instead by passing a pointer to
    the string to the gatekeeper task via a queue. The queue is created before
    the scheduler is started so will already exist by the time this task executes
    for the first time. A block time is not specified because there should
    always be space in the queue. */
        xQueueSendToBack( xPrintQueue, &( pcStringsToPrint[ iIndexToString ] ), 0 );
    /* Wait a pseudo random time. Note that rand() is not necessarily
    reentrant, but in this case it does not really matter as the code does
    not care what value is returned. In a more secure application a version
    of rand() that is known to be reentrant should be used - or calls to
    rand() should be protected using a critical section. */
        vTaskDelay( ( rand() & 0x1FF ) );
    }
}
```

# Example 16

```
void vApplicationTickHook( void )
{
    static int iCount = 0;
    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
    /* Print out a message every 200 ticks. The message is not written out
    directly, but sent to the gatekeeper task. */
    iCount++;
    if( iCount >= 200 )
    {
        /* In this case the last parameter (xHigherPriorityTaskWoken) is not
        actually used but must still be supplied. */
        xQueueSendToFrontFromISR( xPrintQueue,
        &( pcStringsToPrint[ 2 ] ),
        &xHigherPriorityTaskWoken );
        /* Reset the count ready to print out the string again in 200 ticks
        time. */
        iCount = 0;
    }
}
```

Tick hook functions execute within the context of the tick interrupt, and so must be kept very short, must use only a moderate amount of stack space, and must not call any FreeRTOS API function whose name does not end with 'FromISR()'.

# Example 16

```
/* Define the strings that the tasks and interrupt will print out via the
gatekeeper. */
static char *pcStringsToPrint[] =
    {
    "Task 1 ****************************************************\n",
    "Task 2 ----------------------------------------------------\n",
    "Message printed from the tick hook interrupt ##############\n"
    };
/*-----------------------------------------------------------*/
/* Declare a variable of type xQueueHandle. This is used to send messages from
the print tasks and the tick interrupt to the gatekeeper task. */
xQueueHandle xPrintQueue;
/*-----------------------------------------------------------*/
```

# Example 16

```
int main( void )
{
    /* Before a queue is used it must be explicitly created.
    to hold a maximum of 5 character pointers. */
    xPrintQueue = xQueueCreate( 5, sizeof( char * ) );
    /* The tasks are going to use a pseudo random delay, seed the random number
    generator. */
    srand( 567 );
    /* Check the queue was created successfully. */
    if( xPrintQueue != NULL )
    {
    /* Create two instances of the tasks that send messages to the gatekeeper.
    The index to the string the task uses is passed to the task via the task
    parameter (the 4th parameter to xTaskCreate()). The tasks are created at
    different priorities so the higher priority task will occasionally preempt
    the lower priority task. */
        xTaskCreate( prvPrintTask, "Print1", 240, ( void * ) 0, 1, NULL );
        xTaskCreate( prvPrintTask, "Print2", 240, ( void * ) 1, 2, NULL );
    /* Create the gatekeeper task. This is the only task that is permitted
    to directly access standard out. */
        xTaskCreate( prvStdioGatekeeperTask, "Gatekeeper", 240, NULL, 0, NULL );
    /* Start the scheduler so the created 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 heap memory available for the idle task to be created.
    Chapter 5 provides more information on memory management. */
    for( ;; );
}
```

# Example 16: final remarks

The gatekeeper task is assigned a lower priority than the print tasks—so messages sent to the gatekeeper remain in the queue until both print tasks are in the Blocked state. In some situations, it would be appropriate to assign the gatekeeper a higher priority, so that messages get processed sooner—but doing so would be at the cost of the gatekeeper delaying lower priority tasks, until it had completed accessing the protected resource.

# Switching Between Static and Dynamic Memory Allocation

The RAM required to hold these objects can be allocated statically at compile-time or dynamically at run time.

- **Dynamic allocation reduces design and planning effort**, simplifies the API, and minimizes the RAM footprint.
- **Static allocation is more deterministic**, removes the need to handle memory allocation failures, and removes the risk of heap fragmentation (where the heap has enough free memory but not in one usable contiguous block).

# Dynamic allocation

Dynamic memory allocation is a C programming concept, not a concept specific to either FreeRTOS or multitasking.
The general-purpose C library malloc() and free() functions may not be suitable for one or more of the following reasons:

- They are not always available on small embedded systems.
- Their implementation can be relatively large, taking up valuable code space.
- They are rarely thread-safe.
- They are not deterministic; the amount of time taken to execute the functions will differ from call to call.

# Dynamic allocation

- They can suffer from **fragmentation** (where the heap has enough free memory but not in one usable contiguous block).
- They can complicate the linker configuration.
- They can be the source of difficult to debug errors if **the heap space is allowed to grow into memory used by other variables**.

# Dynamic allocation

Early versions of FreeRTOS used a memory pools allocation scheme, where pools of different size memory blocks are pre-allocated at compile-time, then returned by the memory allocation functions.
Although block allocation is common in real-time systems, it was removed from FreeRTOS because its inefficient use of RAM in really small embedded systems led to many support requests.
FreeRTOS now treats memory allocation as part of the **portable layer** (instead of part of the core codebase).

# Dynamic allocation API

- When FreeRTOS requires RAM it calls **pvPortMalloc()** instead of malloc().
- Likewise, when FreeRTOS frees previously allocated RAM it calls **vPortFree()** instead of free().

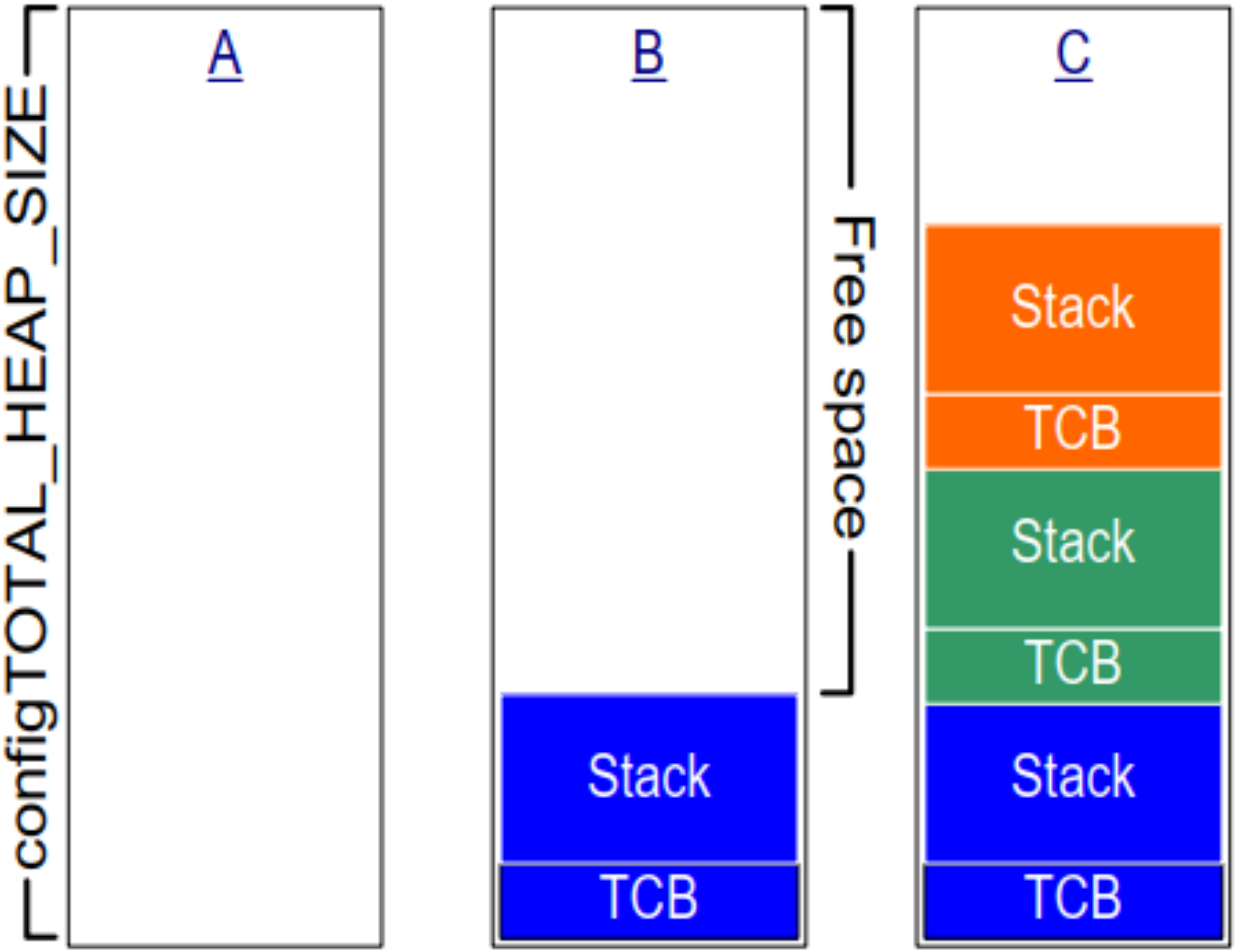
FreeRTOS comes with five example implementations of pvPortMalloc() and vPortFree(). FreeRTOS applications can use one of the example implementations or provide their own. The five examples are defined in the `heap_1.c`, `heap_2.c`, `heap_3.c`, `heap_4.c` and `heap_5.c`

# Heap 1

heap_1.c implements a very basic version of pvPortMalloc(), and **does not implement vPortFree()**.
Heap_1's implementation of pvPortMalloc() simply subdivides a simple uint8_t array called the FreeRTOS heap into smaller blocks each time it's called.

Critical systems often prohibit dynamic memory allocation because of the uncertainties associated with non-determinism, memory fragmentation, and failed allocations. Heap_1 is always deterministic and cannot fragment memory.

# Heap 1


configTOTAL_HEAP_SIZE
A
B
Free space
C
Stack
TCB
Stack
TCB
Stack
TCB
Stack
TCB

# Heap 2

heap_2 is superseded by heap_4, which includes enhanced functionality.
heap_2.c also works by subdividing an array dimensioned by the configTOTAL_HEAP_SIZE constant. It uses a best-fit algorithm to allocate memory, and, unlike heap_1, **it does implement vPortFree()**.
Again, *implementing the heap as a statically allocated array makes FreeRTOS appear to consume a lot of RAM* because the heap becomes part of the FreeRTOS data.

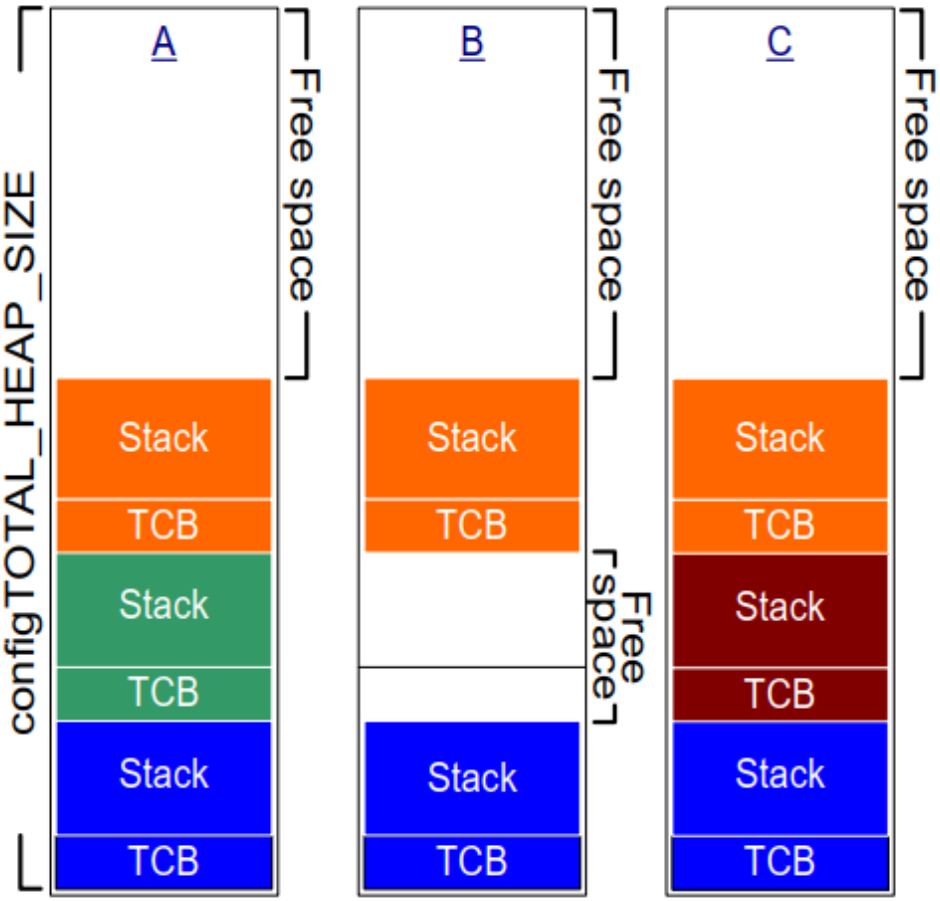
# Heap 2: best fit

The **best-fit algorithm** ensures that pvPortMalloc() uses the free block of memory that is closest in size to the number of bytes requested. For example, consider the scenario where:

- The heap contains three blocks of free memory that are 5 bytes, 25 bytes, and 100 bytes, respectively.
- pvPortMalloc() requests 20 bytes of RAM.

The smallest free block of RAM into which the requested number of bytes fits is the 25-byte block, so pvPortMalloc() splits the 25-byte block into one block of 20 bytes and one block of 5 bytes before returning a pointer to the 20-byte block. The new 5-byte block remains available for future calls to pvPortMalloc().

# Heap 2


configTOTAL_HEAP_SIZE
A
B
C
Free space
Free space
Free space
Free space
Stack
TCB
Stack
TCB
Stack
TCB
Stack
TCB
Stack
TCB
Stack
TCB
Stack
TCB

# Heap 3

heap_3.c **uses the standard library malloc() and free() functions**, so the linker configuration defines the heap size, and the configTOTAL_HEAP_SIZE constant is not used.
heap_3 makes malloc() and free() thread-safe by temporarily suspending the FreeRTOS scheduler for the duration of their execution.

# Heap 4

Like heap_1 and heap_2, heap_4 works by subdividing an array into smaller blocks. As before, the array is statically allocated and dimensioned by configTOTAL_HEAP_SIZE. heap_4 uses a first-fit algorithm to allocate memory. Unlike heap_2, it combines (coalesces) adjacent free blocks of memory into a single larger block, which **minimizes the risk of memory fragmentation**.

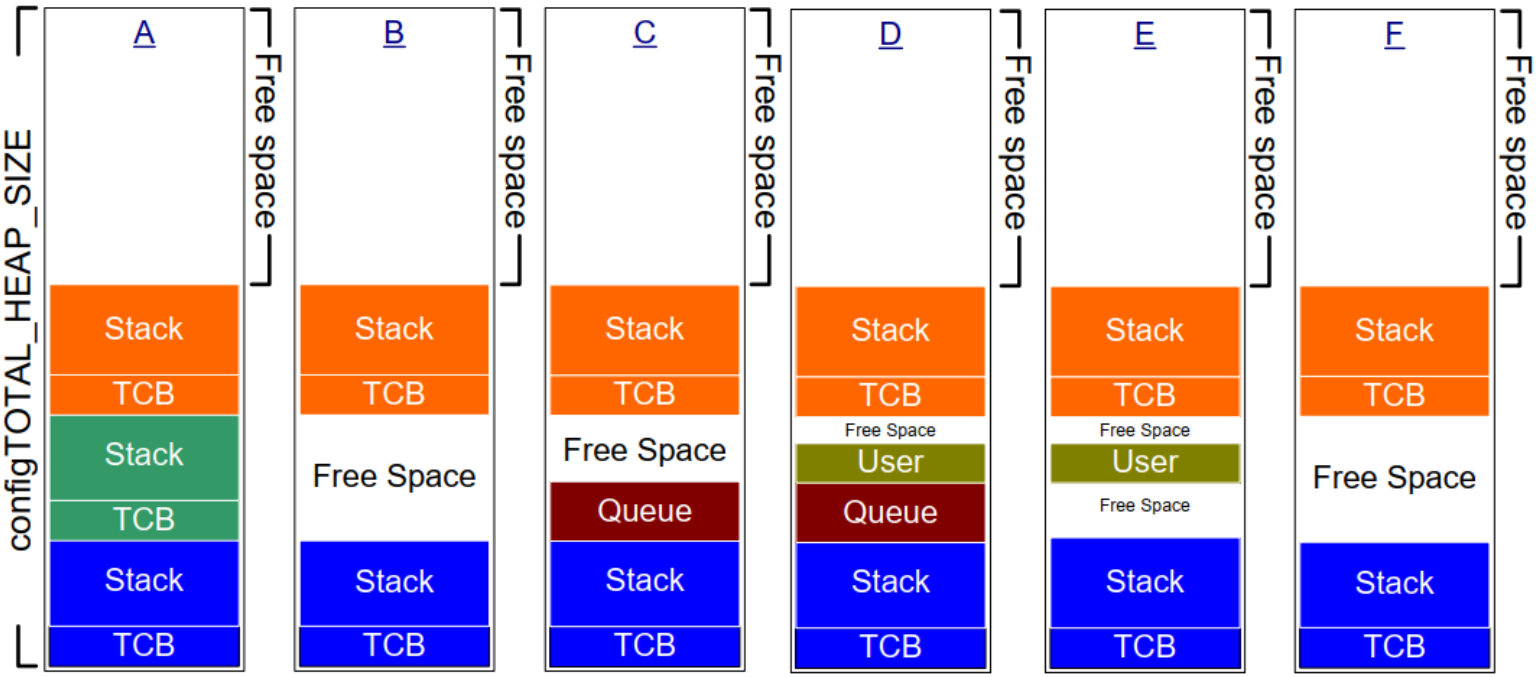
# Heap 4

The **first fit algorithm** ensures pvPortMalloc() uses the first free block of memory that is large enough to hold the number of bytes requested. For example, consider the scenario where:

- The heap contains three blocks of free memory that, in the order in which they appear in the array, are 5 bytes, 200 bytes, and 100 bytes, respectively.
- pvPortMalloc() requests 20 bytes of RAM.

The first free block of RAM that the requested number of bytes fits is the 200-byte block, so pvPortMalloc() splits the 200-byte block into one block of 20 bytes and one of 180 bytes, before returning a pointer to the 20-byte block.
The new 180-byte block remains available to future calls to pvPortMalloc().

# Heap 4


configTOTAL_HEAP_SIZE
A
B
C
D
E
F
Free space
Stack
TCB
Free Space
User
Queue

# Heap 5

heap_5 uses the same allocation algorithm as heap_4. Unlike heap_4, which is limited to allocating memory from a single array, heap_5 can combine memory from multiple separated memory spaces into a single heap.
heap_5 is useful when the RAM provided by the system on which FreeRTOS is running does not appear as a single contiguous (without space) block in the system's memory map.

# Software timers

- Software timers are used to schedule the execution of a function at a set time in the future, or periodically with a fixed frequency. **The function executed by the software timer is called the software timer's callback function**.
- Software timers are implemented by, and are under the control of, the FreeRTOS kernel. **They do not require hardware support**, and are not related to hardware timers or hardware counters.
- Note that, in line with the FreeRTOS philosophy of using innovative design to ensure maximum efficiency, software timers do not use any processing time unless a software timer callback function is actually executing.

Software timer functionality is optional.

# Software timers callback

Software timer callback functions are implemented as C functions. The only thing special about them is their prototype, which must return void, and take a handle to a software timer as its only parameter.

```
void ATimerCallback( TimerHandle_t xTimer );
```

Software timer callback functions execute from start to finish, and exit in the normal way. They should be kept short, and **must not enter the Blocked state.**

# Software timers callback

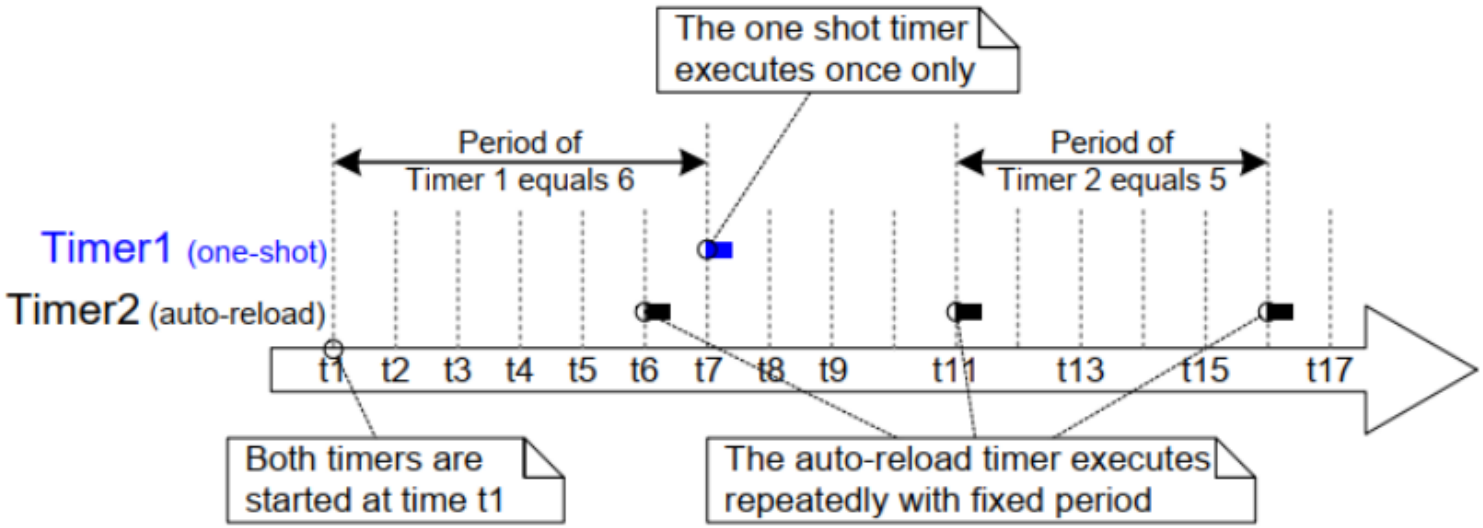
As will be seen, software timer callback functions **execute in the context of a task that is created automatically when the FreeRTOS scheduler is started**. Therefore, it is essential that software timer callback functions never call FreeRTOS API functions that will result in the calling task entering the Blocked state. It is ok to call functions such as xQueueReceive(), but only if the function's xTicksToWait parameter (which specifies the function's block time) is set to 0. It is not ok to call functions such as vTaskDelay(), as calling vTaskDelay() will always place the calling task into the Blocked state.

# Attributes of software timers

**A software timer's 'period' is the time between the software timer being started, and the software timer's callback function executing.** Two types of software timer:

- One-shot timers
- Auto-reload timers

# Attributes of software timers

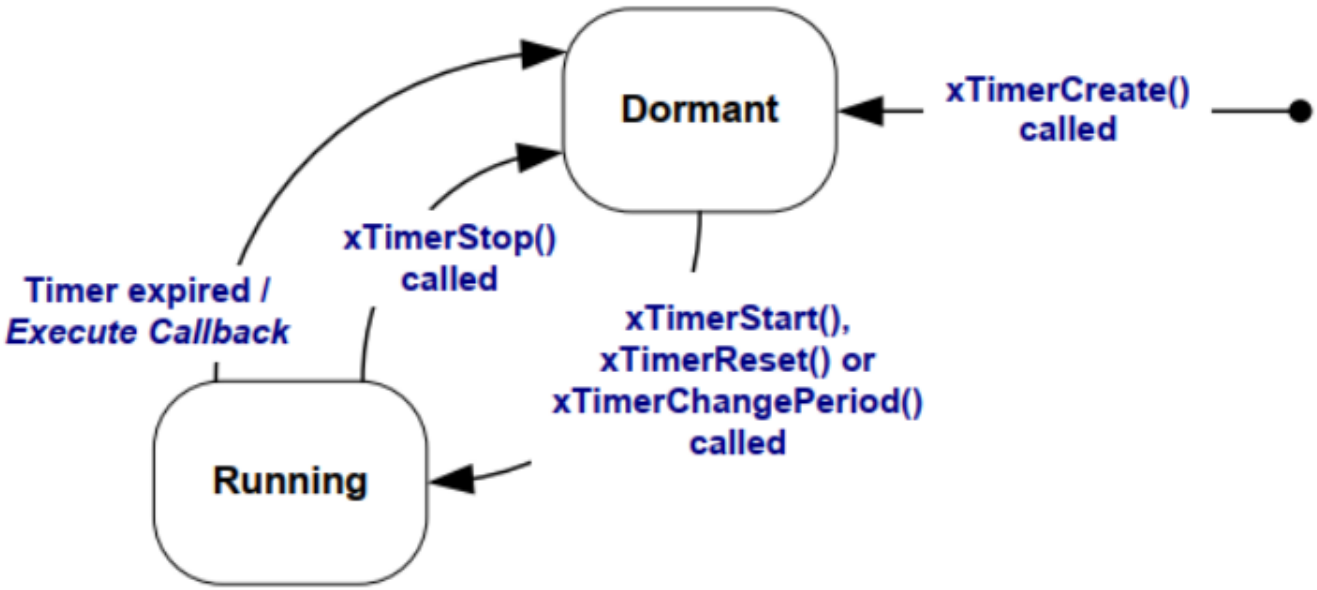


# Software timer states

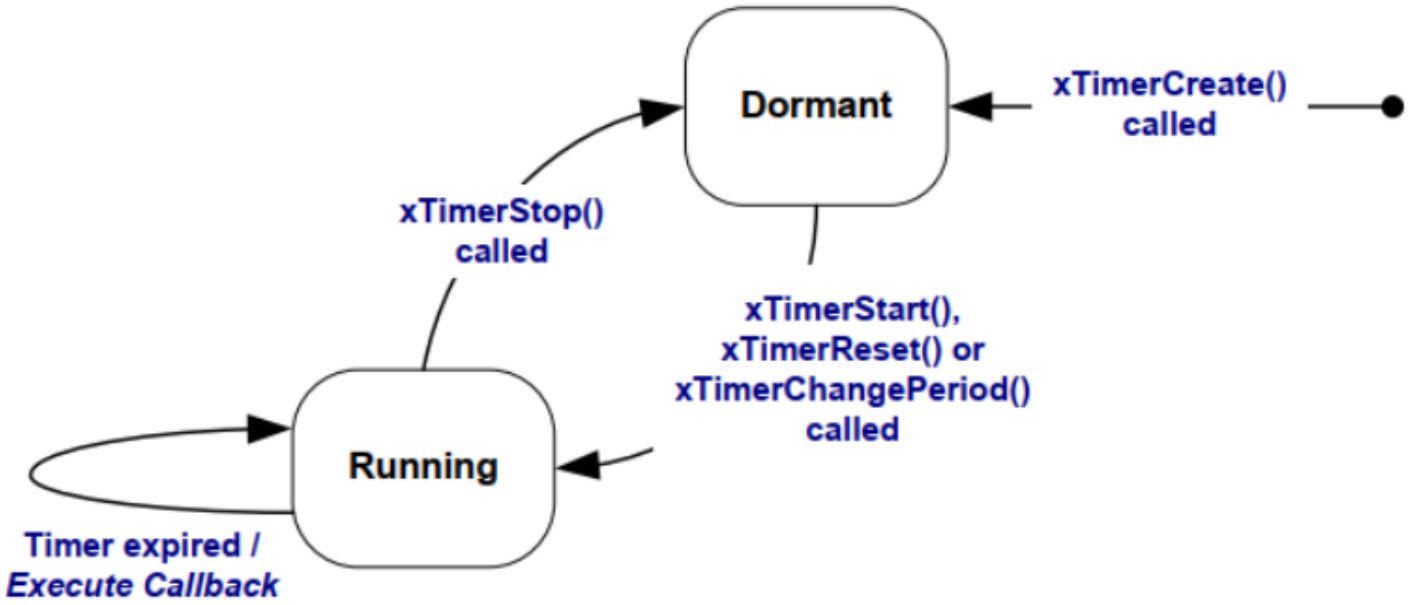
A software timer can be in one of the following two states:

- Dormant: A Dormant software timer exists, and can be referenced by its handle, but is not running, so its callback functions will not execute.
- Running: A Running software timer will execute its callback function after a time equal to its period has elapsed since the software timer entered the Running state, or since the software timer was last reset.

# One-shot


Dormant
xTimerCreate()
called
xTimerStop()
called
Timer expired /
Execute Callback
xTimerStart(),
xTimerReset() or
xTimerChangePeriod()
called
Running

# Autoreload

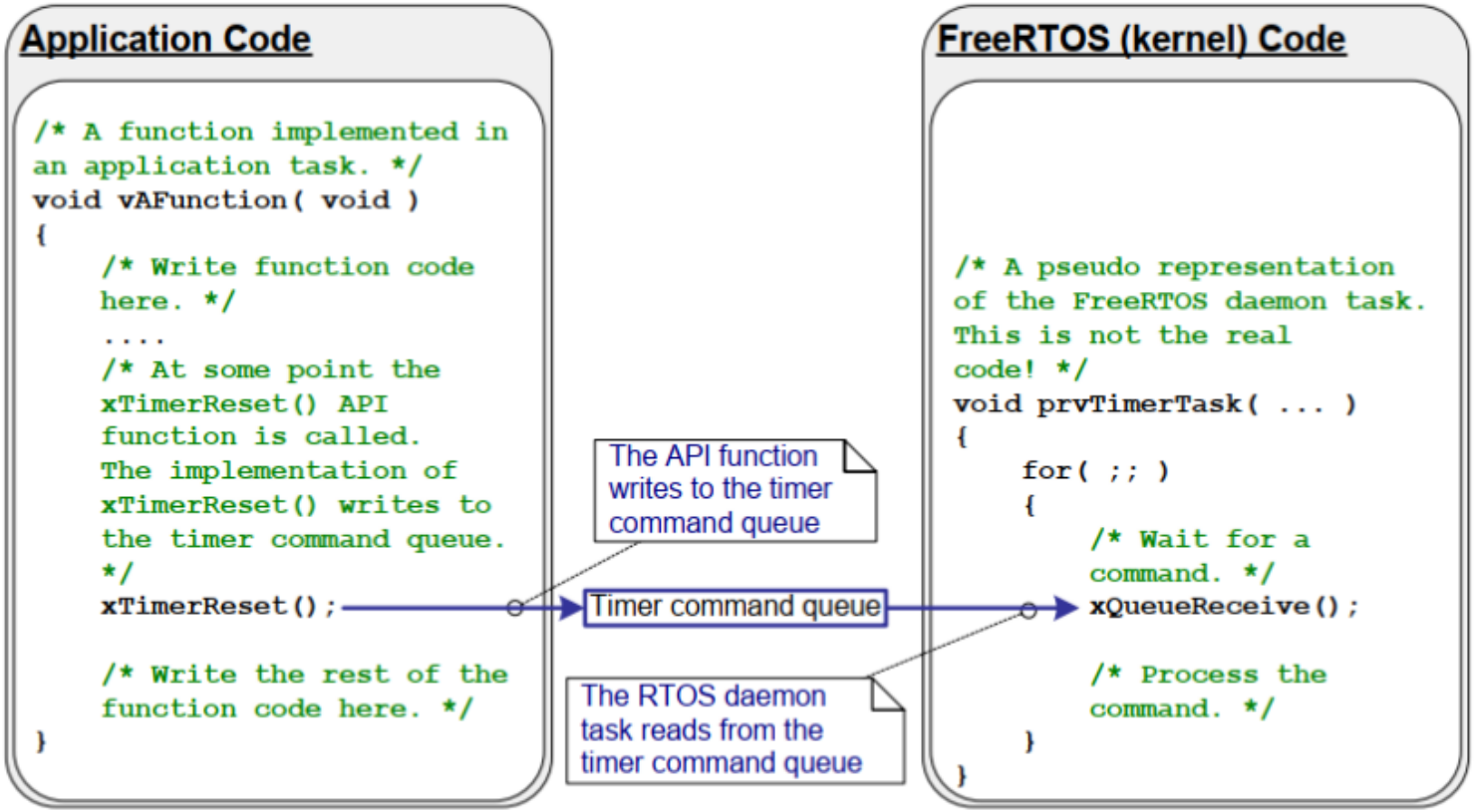

Dormant
xTimerCreate()
called
xTimerStop()
called
xTimerStart(),
xTimerReset() or
xTimerChangePeriod()
called
Running
Timer expired /
Execute Callback

# Software timer context

- All software timer callback functions execute in the context of the same RTOS daemon (or 'timer service') task. Now the task is used for other purposes too, so it is known by the more generic name of the 'RTOS daemon task'.
- The **daemon task** is a standard FreeRTOS task that is created automatically when the scheduler is started. Its priority and stack size are set by the configTIMER_TASK_PRIORITY and configTIMER_TASK_STACK_DEPTH compile time configuration constants respectively.
- Software timer callback functions must not call FreeRTOS API functions that will result in the calling task entering the Blocked state, as to do so will result in the daemon task entering the Blocked state.

# The Timer Command Queue

Software timer API functions send commands from the calling task to the daemon task on a queue called the '**timer command queue**'.



# The Timer Command Queue

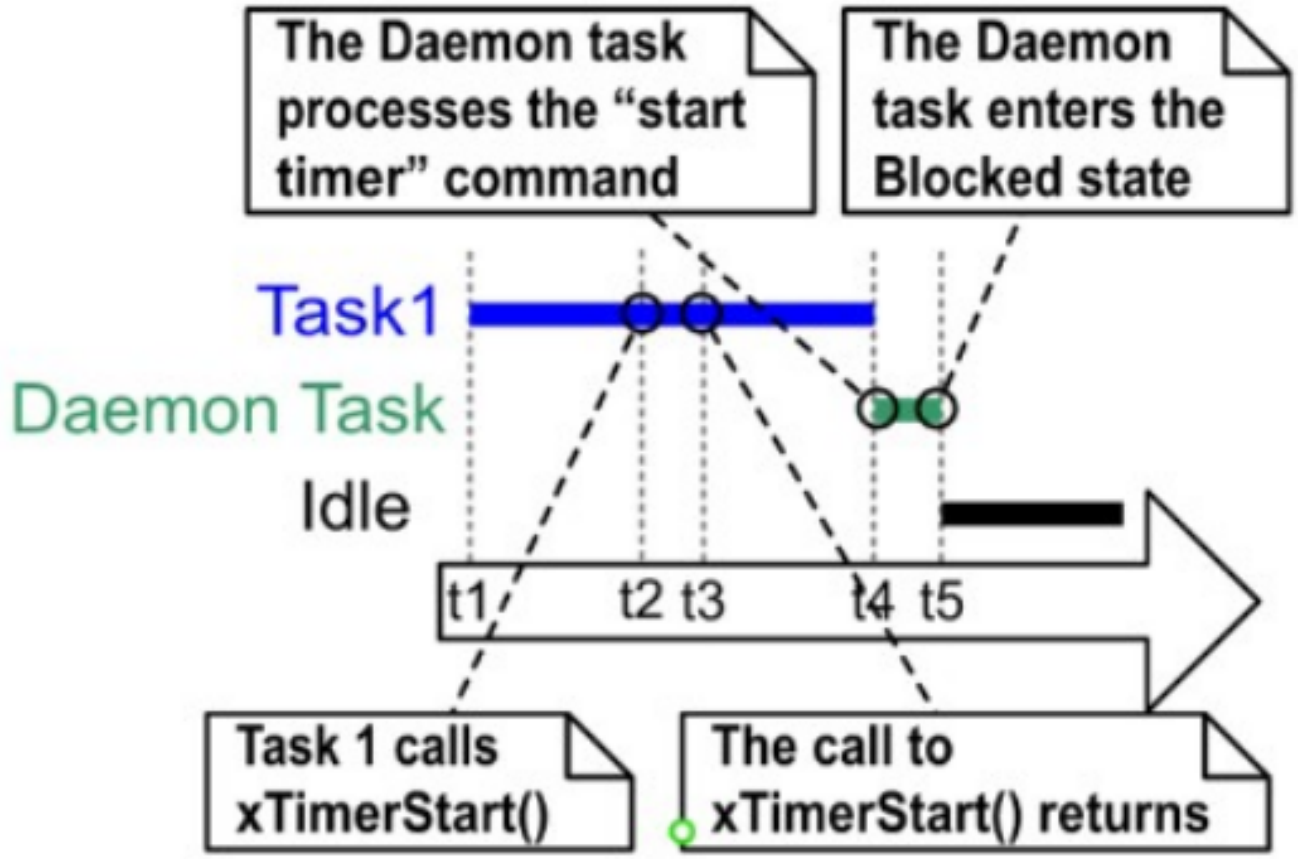
The timer command queue is a standard FreeRTOS queue that is created automatically when the scheduler is started.

# The Timer Command Queue

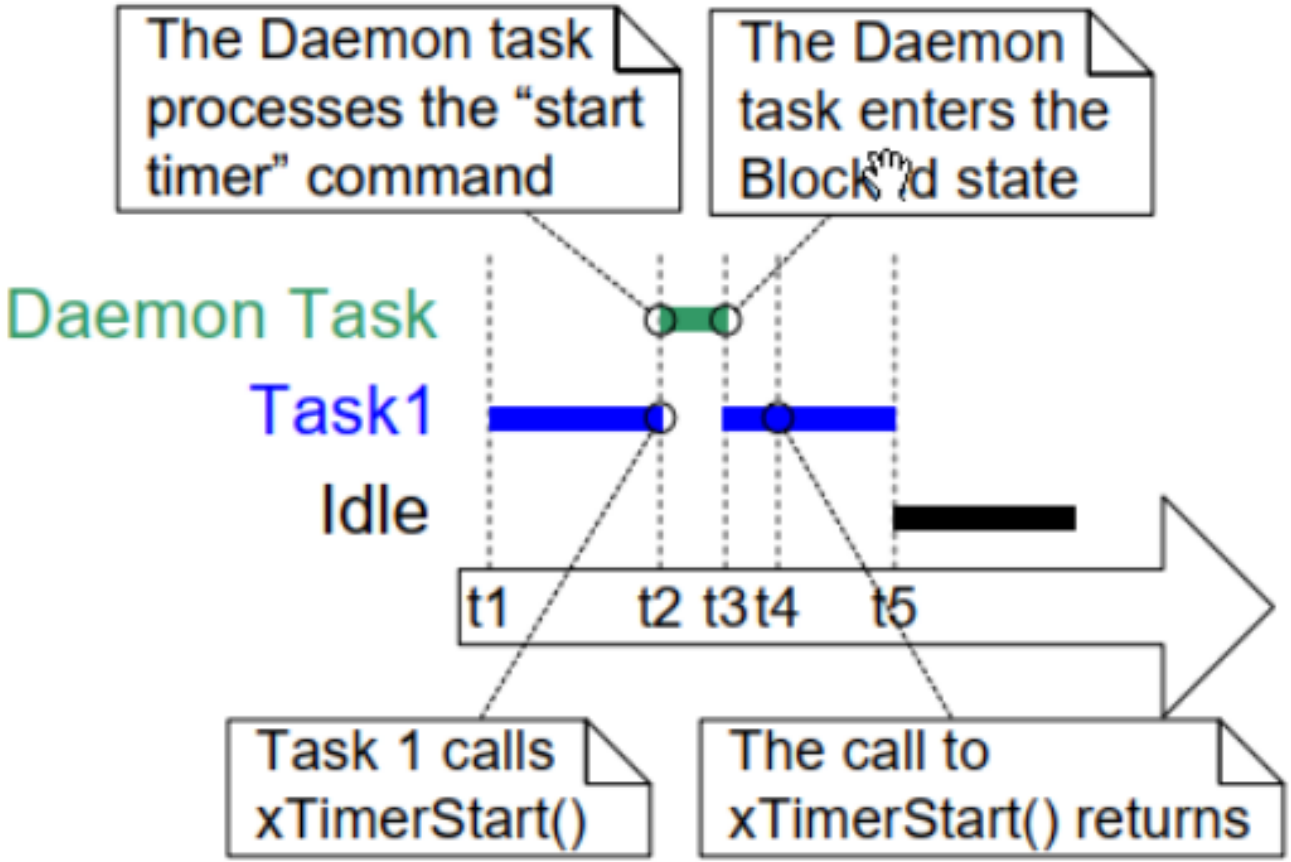
The timer command queue is a standard FreeRTOS queue that is created automatically when the scheduler is started.
The daemon task is scheduled like any other FreeRTOS task; **it will only process commands, or execute timer callback functions, when it is the highest priority task that is able to run**.

The time at which the software timer being started will expire is calculated from the time the 'start a timer' command was sent to the timer command queue—it is not calculated from the time the daemon task received the 'start a timer' command from the timer command queue.

# Daemon task priority: low



# Daemon task priority: high



# xTimerCreate()

- Software timers are referenced by variables of type TimerHandle_t. Software timers are created in the Dormant state.
- xTimerCreateStatic() function allocates the memory required to create a timer statically at compile time: a software timer must be explicitly created before it can be used.

# xTimerCreate()

```
TimerHandle_t xTimerCreate( const char *const pcTimerName,
                            const TickType_t xTimerPeriodInTicks,
                            const BaseType_t xAutoReload,
                            void * const pvTimerID,
                            TimerCallbackFunction_t pxCallbackFunction );
```

pcTimerName A descriptive name for the timer.

xTimerPeriodInTicks The timer's period specified in ticks. Cannot be 0.

xAutoReload **Set xAutoReload to pdFALSE to create a one-shot timer**.

pvTimerID Each software timer has an ID value. The ID is a void pointer.

# xTimerCreate()

pxCallbackFunction Software timer callback functions are simply C functions that conform to the prototype.

return If a non-NULL value is returned it indicates that the software timer has been created successfully. The returned value is the handle of the created timer.

# xTimerStart()

- xTimerStart() is used to start a software timer that is in the Dormant state, or reset (re-start) a software timer that is in the Running state.
- xTimerStop() is used to stop a software timer that is in the Running state. Stopping a software timer is the same as transitioning the timer into the Dormant state.
- xTimerStart() can be called before the scheduler is started, but **when this is done, the software timer will not actually start until the time at which the scheduler starts**.

The interrupt-safe version xTimerStartFromISR() should be used in ISR.

# xTimerStart()

```
BaseType_t xTimerStart( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

xTimer The handle of the software timer being started or reset.

xTicksToWait xTimerStart() uses the timer command queue to send the 'start a timer' command to the daemon task. xTicksToWait specifies the maximum amount of time the calling task should remain in the Blocked state to wait for space to become available on the timer command queue, should the queue already be full. xTimerStart() will return immediately if xTicksToWait is zero and the timer command queue is already full.

# xTimerStart()

return pdFAIL will be returned if the 'start a timer' command could not be written to the timer command queue because the queue was already full.

# Example 14b: main

```
/* The periods assigned to the one-shot and auto-reload timers are 3.333
second and half a second respectively. */
#define mainONE_SHOT_TIMER_PERIOD pdMS_TO_TICKS( 3333 )
#define mainAUTO_RELOAD_TIMER_PERIOD pdMS_TO_TICKS( 500 )
int main( void )
{
    TimerHandle_t xAutoReloadTimer, xOneShotTimer;
    BaseType_t xTimer1Started, xTimer2Started;
    /* Create the one shot timer, storing the handle to the created timer in
    xOneShotTimer. */
    xOneShotTimer = xTimerCreate(
    /* Text name for the software timer - not used by FreeRTOS. */
                                 "OneShot",
    /* The software timer's period in ticks. */
                                 mainONE_SHOT_TIMER_PERIOD,
    /* Setting uxAutoRealod to pdFALSE creates a one-shot software timer. */
                                 pdFALSE,
    /* This example does not use the timer id. */
                                 0,
    /* Callback function to be used by the software timer being created. */
                                 prvOneShotTimerCallback );
    /* Create the auto-reload timer, storing the handle to the created timer
    in xAutoReloadTimer. */
```

# Example 14b: main

```
    xAutoReloadTimer = xTimerCreate(
    /* Text name for the software timer - not used by FreeRTOS. */
                                    "AutoReload",
    /* The software timer's period in ticks. */
                                    mainAUTO_RELOAD_TIMER_PERIOD,
    /* Setting uxAutoRealod to pdTRUE creates an auto-reload timer. */
                                    pdTRUE,
    /* This example does not use the timer id. */
                                    0,
    /* Callback function to be used by the software timer being created. */
                                    prvAutoReloadTimerCallback );
    /* Check the software timers were created. */
    if( ( xOneShotTimer != NULL ) && ( xAutoReloadTimer != NULL ) )
    {
        /* Start the software timers, using a block time of 0 (no block time).
        The scheduler has not been started yet so any block time specified
        here would be ignored anyway. */
        xTimer1Started = xTimerStart( xOneShotTimer, 0 );
        xTimer2Started = xTimerStart( xAutoReloadTimer, 0 );
```

# Example 14b: main

```
        /* The implementation of xTimerStart() uses the timer command queue,
        and xTimerStart() will fail if the timer command queue gets full.
        The timer service task does not get created until the scheduler is
        started, so all commands sent to the command queue will stay in the
        queue until after the scheduler has been started. Check both calls
        to xTimerStart() passed. */
        if( ( xTimer1Started == pdPASS )  ( xTimer2Started == pdPASS ) )
        {
            /* Start the scheduler. */
            vTaskStartScheduler();
        }
    }
    /* As always, this line should not be reached. */
    for( ;; );
}
```

# Example 14b: prvOneShotTimerCallback

```
static void prvOneShotTimerCallback( TimerHandle_t xTimer )
{
    TickType_t xTimeNow;
    /* Obtain the current tick count. */
    xTimeNow = xTaskGetTickCount();
    /* Output a string to show the time at which the callback was executed. */
    vPrintStringAndNumber( "One-shot timer callback executing", xTimeNow );
    /* File scope variable. */
    ulCallCount++;
}
```

# Example 14b: prvAutoReloadTimerCallback

```
static void prvAutoReloadTimerCallback( TimerHandle_t xTimer )
{
    TickType_t xTimeNow;
    /* Obtain the current tick count. */
    xTimeNow = xTaskGetTickCount();
    /* Output a string to show the time at which the callback was executed. */
    vPrintStringAndNumber( "Auto-reload timer callback executing", xTimeNow);
    ulCallCount++;
}
```

The version provided in docker is a bit different...

# Other software timer APIs

```
void vTimerSetTimerID( const TimerHandle_t xTimer, void *pvNewID );
void *pvTimerGetTimerID( const TimerHandle_t xTimer );
BaseType_t xTimerChangePeriod( TimerHandle_t xTimer,
                               TickType_t xNewPeriod,
                               TickType_t xTicksToWait );
BaseType_t xTimerReset( TimerHandle_t xTimer, TickType_t xTicksToWait );
BaseType_t xTimerDelete( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

# Event group vs other sync methods

It has already been noted that real-time embedded systems have to take actions in response to events.

# Event group vs other sync methods

It has already been noted that real-time embedded systems have to take actions in response to events. Examples of such features include semaphores and queues, both of which have the following properties:

- They allow a task to wait in the Blocked state for a single event to occur.
- They unblock a single task when the event occurs. **The task that is unblocked is the highest priority task** that was waiting for the event.

# Event group

Event groups are another feature of FreeRTOS that allow events to be communicated to tasks. Unlike queues and semaphores:

- Event groups allow a task to wait in the Blocked state **for a combination of one of more events to occur**.
- Event groups **unblock all the tasks that were waiting for the same event**, or combination of events, when the event occurs.

# Event group

Event groups also provide the opportunity to **reduce the RAM** used by an application as, often, it is possible to replace many binary semaphores with a single event group.

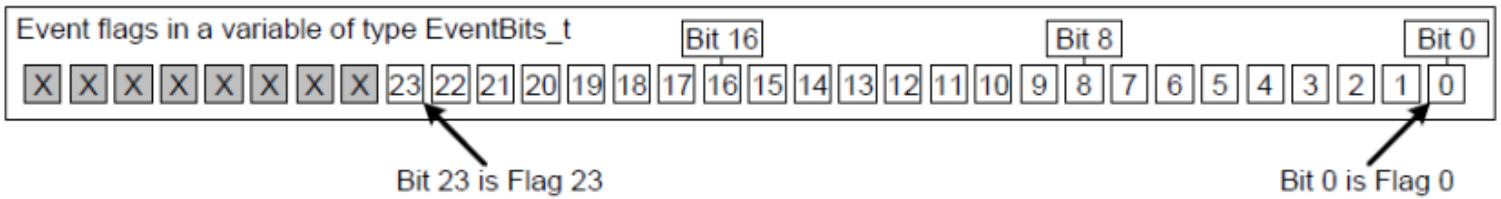
# Event group

Event groups also provide the opportunity to **reduce the RAM** used by an application as, often, it is possible to replace many binary semaphores with a single event group.

Event group functionality is optional. To include event group functionality, build the FreeRTOS source file event_groups.c as part of your project.

# Event Groups, Event Flags and Event Bits

- An event 'flag' is a Boolean (1 or 0) value used to indicate if an event has occurred or not.
- An event 'group' is a set of event flags.




The state of each event flag is represented by a single bit in a variable of type EventBits_t. For that reason, event flags are also known as event 'bits'

# Event group example

It is up to the application writer to assign a meaning to individual bits within an event group. For example, the application writer might create an event group, then:

- Define bit 0 within the event group to mean "a message has been received from the network".
- Define bit 1 within the event group to mean "a message is ready to be sent onto the network".
- Define bit 2 within the event group to mean "abort the current network connection".

# Multiple tasks and compilation

Event groups are objects in their own right that can be accessed by any task or ISR that knows of their existence.
Any number of tasks can set bits in the same event group, and any number of tasks can read bits from the same event group.

The number of event bits in an event group is dependent on the configTICK_TYPE_WIDTH_IN_BITS compile time configuration constant in FreeRTOSConfig.h.

# xEventGroupCreate()

FreeRTOS also includes the xEventGroupCreateStatic() function, which allocates the memory required to create an event group statically at compile time.

```
EventGroupHandle_t xEventGroupCreate( void );
```

return If NULL is returned, then the event group cannot be created because there is insufficient heap memory available.
**The returned value should be stored as the handle to the created event group.**

# xEventGroupSetBits()

The xEventGroupSetBits() API function sets one or more bits in an event group, and is typically used to notify a task that the events represented by the bit, or bits, being set has occurred.

```
EventBits_t xEventGroupSetBits(EventGroupHandle_t xEventGroup,
                               const EventBits_t uxBitsToSet);
```

xEventGroup The handle of the event group in which bits are being set.

uxBitsToSet A bit mask that specifies the event bit, or event bits, to set to 1 in the event group. The value of the event group is updated by bitwise ORing the event group's existing value with the value passed in uxBitsToSet.

# xEventGroupSetBits()

return The value of the event group at the time the call to xEventGroupSetBits() returned. **Note that the value returned will not necessarily have the bits specified by uxBitsToSet set, because the bits may have been cleared again by a different task.**

# xEventGroupSetBitsFroISR()

Giving a semaphore is a deterministic operation because it is known in advance that giving a semaphore can result in at most one task leaving the Blocked state. When bits are set in an event group **it is not known in advance how many tasks will leave the Blocked state, so setting bits in an event group is not a deterministic operation**.

```
BaseType_t xEventGroupSetBitsFromISR( EventGroupHandle_t xEventGroup,
                                      const EventBits_t uxBitsToSet,
                                      BaseType_t *pxHigherPriorityTaskWoken );
```

pxHigherPriorityTaskWoken does not set the event bits directly inside the interrupt service routine, but instead defers the action to the RTOS daemon task by sending a command on the timer command queue.

# xEventGroupSetBitsFroISR()

pxHigherPriorityTaskWoken **xEventGroupSetBitsFromISR() does not set the event bits directly inside the interrupt service routine, but instead defers the action to the RTOS daemon task by sending a command on the timer command queue.** If the 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, xEventGroupSetBitsFromISR() will set pxHigherPriorityTaskWoken to pdTRUE.

# xEventGroupSetBitsFromISR()

return pdPASS will be returned only if data was successfully sent to the timer command queue.
pdFALSE will be returned if the 'set bits' command could not be written to the timer command queue because the queue was already full.

# xEventGroupWaitBits()

The xEventGroupWaitBits() API function allows a task to read the value of an event group, and optionally wait in the Blocked state for one or more event bits in the event group to become set, if the event bits are not already set.

```
EventBits_t xEventGroupWaitBits( EventGroupHandle_t xEventGroup,
                                 const EventBits_t uxBitsToWaitFor,
                                 const BaseType_t xClearOnExit,
                                 const BaseType_t xWaitForAllBits,
                                 TickType_t xTicksToWait );
```

uxBitsToWaitFor specifies which event bits in the event group to test

xWaitForAllBits specifies whether to use a bitwise OR test, or a bitwise AND test.

# xEventGroupWaitBits()

xClearOnExit If xClearOnExit is set to pdTRUE, then the **testing and clearing of event bits appears to the calling task to be an atomic operation** (uninterruptable by other tasks or interrupts).

## Clear bits

Event bits can be cleared using the xEventGroupClearBits() API function, but using that function to manually clear event bits will lead to race conditions in the application code if:

- There is more than one task using the same event group.
- Bits are set in the event group by a different task, or by an interrupt service routine.

# xEventGroupWaitBits()

xTicksToWait The maximum amount of time the task should remain in the Blocked state to wait for its unblock condition to be met.

return If xEventGroupWaitBits() returned because the calling task's unblock condition was met, then the returned value is the value of the event group at the time the calling task's unblock condition was met. If xEventGroupWaitBits() returned because the block time specified by the xTicksToWait parameter expired, then the returned value is the value of the event group at the time the block time expired.

# xEventGroupGetStaticBuffer()

The xEventGroupGetStaticBuffer() API function provides a method to retrieve a pointer to a buffer of a statically created event group. It is the same buffer that is supplied at the time of creation of the event group.

```
BaseType_t xEventGroupGetStaticBuffer( EventGroupHandle_t xEventGroup,
                                       StaticEventGroup_t ** ppxEventGroupBuffer );
```

xEventGroup The event group for which to retrieve the buffer.

ppxEventGroupBuffer Used to return a pointer to the event groups's data structure buffer.

return pdTRUE will be returned if the buffer was successfully retrieved, pdFALSE otherwise.

# Example 22: vEventBitSettingTask

```
static void vEventBitSettingTask( void *pvParameters )
{
    const TickType_t xDelay200ms = pdMS_TO_TICKS( 200UL ), xDontBlock = 0;
    for( ;; )
    {
        /* Delay for a short while before starting the next loop. */
        vTaskDelay( xDelay200ms );
        /* Print out a message to say event bit 0 is about to be set by the
        task, then set event bit 0. */
        vPrintString( "Bit setting task -\t about to set bit 0.\r\n" );
        xEventGroupSetBits( xEventGroup, mainFIRST_TASK_BIT );
        /* Delay for a short while before setting the other bit. */
        vTaskDelay( xDelay200ms );
        /* Print out a message to say event bit 1 is about to be set by the
        task, then set event bit 1. */
        vPrintString( "Bit setting task -\t about to set bit 1.\r\n" );
        xEventGroupSetBits( xEventGroup, mainSECOND_TASK_BIT );
    }
}
```

# Example 22: ulEventBitSettingISR

```
static uint32_t ulEventBitSettingISR( void )
{
    /* The string is not printed within the interrupt service routine, but is
    instead sent to the RTOS daemon task for printing. It is therefore
    declared static to ensure the compiler does not allocate the string on
    the stack of the ISR, as the ISR's stack frame will not exist when the
    string is printed from the daemon task. */
    static const char *pcString = "Bit setting ISR -\t about to set bit 2.\r\n";
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    /* Print out a message to say bit 2 is about to be set. Messages cannot
    be printed from an ISR, so defer the actual output to the RTOS daemon
    task by pending a function call to run in the context of the RTOS
    daemon task. */
    xTimerPendFunctionCallFromISR( vPrintStringFromDaemonTask,
                                   ( void * ) pcString,
                                   0,
                                   &xHigherPriorityTaskWoken );
    /* Set bit 2 in the event group. */
    xEventGroupSetBitsFromISR( xEventGroup,
                               mainISR_BIT,
                               &xHigherPriorityTaskWoken );
```

# Example 22: ulEventBitSettingISR

```
    /* xTimerPendFunctionCallFromISR() and xEventGroupSetBitsFromISR() both
    write to the timer command queue, and both used the same
    xHigherPriorityTaskWoken variable. If writing to the timer command
    queue resulted in the RTOS daemon task leaving the Blocked state, and
    if the priority of the RTOS daemon task is higher than the priority of
    the currently executing task (the task this interrupt interrupted) then
    xHigherPriorityTaskWoken will have been set to pdTRUE.
    xHigherPriorityTaskWoken is used as the parameter to
    portYIELD_FROM_ISR(). If xHigherPriorityTaskWoken equals pdTRUE, then
    calling portYIELD_FROM_ISR() will request a context switch. If
    xHigherPriorityTaskWoken is still pdFALSE, then calling
    portYIELD_FROM_ISR() will have no effect.
    The implementation of portYIELD_FROM_ISR() used by the Windows port
    includes a return statement, which is why this function does not
    explicitly return a value. */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

# Example 22: vEventBitReadingTask

```
static void vEventBitReadingTask( void *pvParameters )
{
    EventBits_t xEventGroupValue;
    const EventBits_t xBitsToWaitFor = ( mainFIRST_TASK_BIT |
    mainSECOND_TASK_BIT |
    mainISR_BIT );
    for( ;; )
    {
        /* Block to wait for event bits to become set within the event
        group. */
        xEventGroupValue = xEventGroupWaitBits( /* The event group to read */
                                                xEventGroup,
                                                /* Bits to test */
                                                xBitsToWaitFor,
                                                /* Clear bits on exit if the
                                                unblock condition is met */
                                                pdTRUE,
                                                /* Don't wait for all bits. This
                                                parameter is set to pdTRUE for
                                                the second execution. */
                                                pdFALSE,
                                                /* Don't time out. */
                                                portMAX_DELAY );
        /* Print a message for each bit that was set. */
        if( ( xEventGroupValue & mainFIRST_TASK_BIT ) != 0 )
        {
```

# Example 22: vEventBitReadingTask

```
            vPrintString( "Bit reading task -\t Event bit 0 was set\r\n" );
        }
        if( ( xEventGroupValue & mainSECOND_TASK_BIT ) != 0 )
        {
            vPrintString( "Bit reading task -\t Event bit 1 was set\r\n" );
        }
        if( ( xEventGroupValue & mainISR_BIT ) != 0 )
        {
            vPrintString( "Bit reading task -\t Event bit 2 was set\r\n" );
        }
    }
}
```

# Example 22: main

```
int main( void )
{
    /* Before an event group can be used it must first be created. */
    xEventGroup = xEventGroupCreate();
    /* Create the task that sets event bits in the event group. */
    xTaskCreate( vEventBitSettingTask, "Bit Setter", 1000, NULL, 1, NULL );
    /* Create the task that waits for event bits to get set in the event
    group. */
    xTaskCreate( vEventBitReadingTask, "Bit Reader", 1000, NULL, 2, NULL );
    /* Create the task that is used to periodically generate a software
    interrupt. */
    xTaskCreate( vInterruptGenerator, "Int Gen", 1000, NULL, 3, NULL );
    /* Install the handler for the software interrupt. The syntax necessary
    to do this is dependent on the FreeRTOS port being used. The syntax
    shown here can only be used with the FreeRTOS Windows port, where such
    interrupts are only simulated. */
    vPortSetInterruptHandler( mainINTERRUPT_NUMBER, ulEventBitSettingISR );
    /* Start the scheduler so the created tasks start executing. */
    vTaskStartScheduler();
    /* The following line should never be reached. */
    for( ;; );
    return 0;
}
```

# Task synchronization

Sometimes the design of an application requires two or more tasks to synchronize with each other.

# Task synchronization

Sometimes the design of an application requires two or more tasks to synchronize with each other.
For example, consider a design where Task A receives an event, then delegates some of the processing necessitated by the event to three other tasks: Task B, Task C and Task D.

# Task synchronization

Sometimes the design of an application requires two or more tasks to synchronize with each other.
For example, consider a design where Task A receives an event, then delegates some of the processing necessitated by the event to three other tasks: Task B, Task C and Task D.
If Task A cannot receive another event until tasks B, C and D have all completed processing the previous event, then all four tasks will need to synchronize with each other. Each task's synchronization point will be after that task has completed its processing, and cannot proceed further until each of the other tasks have done the same.

# Task synchronization

Sometimes the design of an application requires two or more tasks to synchronize with each other.
For example, consider a design where Task A receives an event, then delegates some of the processing necessitated by the event to three other tasks: Task B, Task C and Task D.
If Task A cannot receive another event until tasks B, C and D have all completed processing the previous event, then all four tasks will need to synchronize with each other. Each task's synchronization point will be after that task has completed its processing, and cannot proceed further until each of the other tasks have done the same.
Task A can only receive another event after all four tasks have reached their synchronization point.

# Task synchronization

An event group can be used to create a synchronization point:

- Each task that must participate in the synchronization is assigned a unique event bit within the event group.
- Each task sets its own event bit when it reaches the synchronization point.
- Having set its own event bit, each task blocks on the event group to wait for the event bits that represent all the other synchronizing tasks to also become set.

However, the **xEventGroupSetBits() and xEventGroupWaitBits() API functions cannot be used in this scenario**. If they were used, then the setting of a bit (to indicate a task had reached its synchronization point) and the testing of bits (to determine if the other synchronizing tasks had reached their synchronization point) **would be performed as two separate operations**.

# Task synchronization

However, the **xEventGroupSetBits() and xEventGroupWaitBits() API functions cannot be used in this scenario**. If they were used, then the setting of a bit (to indicate a task had reached its synchronization point) and the testing of bits (to determine if the other synchronizing tasks had reached their synchronization point) **would be performed as two separate operations**.

The xEventGroupSync() API function is provided for that purpose.

# xEventGroupSync()

xEventGroupSync() is provided to allow two or more tasks to use an event group to synchronize with each other. The function allows a task to set one or more event bits in an event group, then wait for a combination of event bits to become set in the same event group, as a single uninterruptable operation.

```
EventBits_t xEventGroupSync(EventGroupHandle_t xEventGroup,
                            const EventBits_t uxBitsToSet,
                            const EventBits_t uxBitsToWaitFor,
                            TickType_t xTicksToWait);
```

# Example 23: vSyncingTask

```
static void vSyncingTask( void *pvParameters )
{
    const TickType_t xMaxDelay = pdMS_TO_TICKS( 4000UL );
    const TickType_t xMinDelay = pdMS_TO_TICKS( 200UL );
    TickType_t xDelayTime;
    EventBits_t uxThisTasksSyncBit;
    const EventBits_t uxAllSyncBits = ( mainFIRST_TASK_BIT |
                                        mainSECOND_TASK_BIT |
                                        mainTHIRD_TASK_BIT );
    /* Three instances of this task are created - each task uses a different
    event bit in the synchronization. The event bit to use is passed into
    each task instance using the task parameter. Store it in the
    uxThisTasksSyncBit variable. */
    uxThisTasksSyncBit = ( EventBits_t ) pvParameters;
    for( ;; )
    {
        /* Simulate this task taking some time to perform an action by delaying
        for a pseudo random time. This prevents all three instances of this
        task reaching the synchronization point at the same time, and so
        allows the example's behavior to be observed more easily. */
        xDelayTime = ( rand() % xMaxDelay ) + xMinDelay;
        vTaskDelay( xDelayTime );
```

# Example 23: vSyncingTask

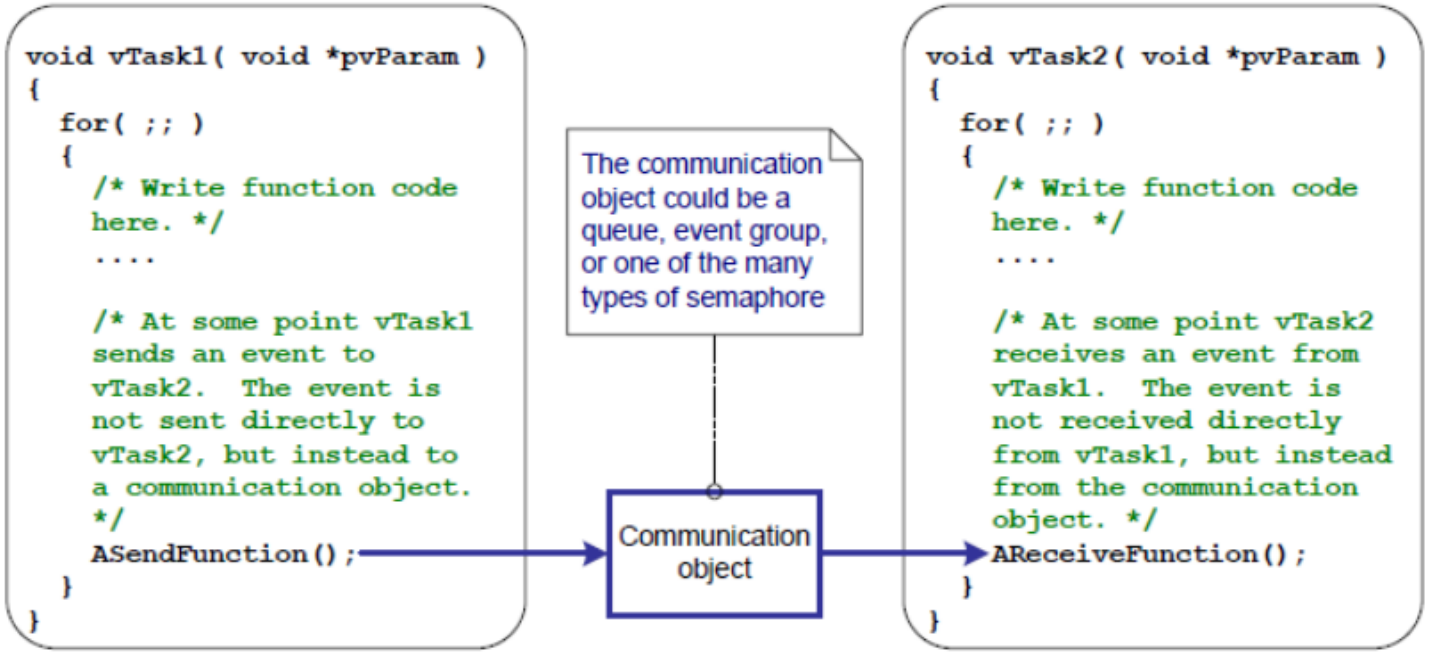
```
        /* Print out a message to show this task has reached its synchronization
        point. pcTaskGetTaskName() is an API function that returns the name
        assigned to the task when the task was created. */
        vPrintTwoStrings( pcTaskGetTaskName( NULL ), "reached sync point" );
        /* Wait for all the tasks to have reached their respective
        synchronization points. */
        xEventGroupSync(/* The event group used to synchronize. */
                        xEventGroup,
                        /* The bit set by this task to indicate it has reached
                        the synchronization point. */
                        uxThisTasksSyncBit,
                        /* The bits to wait for, one bit for each task taking
                        part in the synchronization. */
                        uxAllSyncBits,
                        /* Wait indefinitely for all three tasks to reach the
                        synchronization point. */
                        portMAX_DELAY );
        /* Print out a message to show this task has passed its synchronization
        point. As an indefinite delay was used the following line will only
        be executed after all the tasks reached their respective
        synchronization points. */
        vPrintTwoStrings( pcTaskGetTaskName( NULL ), "exited sync point" );
    }
}
```

# Example 23: main

```
#define mainFIRST_TASK_BIT ( 1UL << 0UL ) /* Event bit 0, set by the 1st task */
#define mainSECOND_TASK_BIT( 1UL << 1UL ) /* Event bit 1, set by the 2nd task */
#define mainTHIRD_TASK_BIT ( 1UL << 2UL ) /* Event bit 2, set by the 3rd task */
/* Declare the event group used to synchronize the three tasks. */
EventGroupHandle_t xEventGroup;
int main( void )
{
    /* Before an event group can be used it must first be created. */
    xEventGroup = xEventGroupCreate();
    /* Create three instances of the task. Each task is given a different
    name, which is later printed out to give a visual indication of which
    task is executing. The event bit to use when the task reaches its
    synchronization point is passed into the task using the task parameter. */
    xTaskCreate( vSyncingTask, "Task 1", 1000, mainFIRST_TASK_BIT, 1, NULL );
    xTaskCreate( vSyncingTask, "Task 2", 1000, mainSECOND_TASK_BIT, 1, NULL );
    xTaskCreate( vSyncingTask, "Task 3", 1000, mainTHIRD_TASK_BIT, 1, NULL );
    /* Start the scheduler so the created tasks start executing. */
    vTaskStartScheduler();
    /* As always, the following line should never be reached. */
    for( ;; );
    return 0;
}
```

# Task notification: problem

Task notifications are an **efficient** mechanism allowing one task to **directly notify** another task.
The methods described so far have required the creation of a communication object. Examples of communication objects include queues, event groups, and various different types of semaphore.

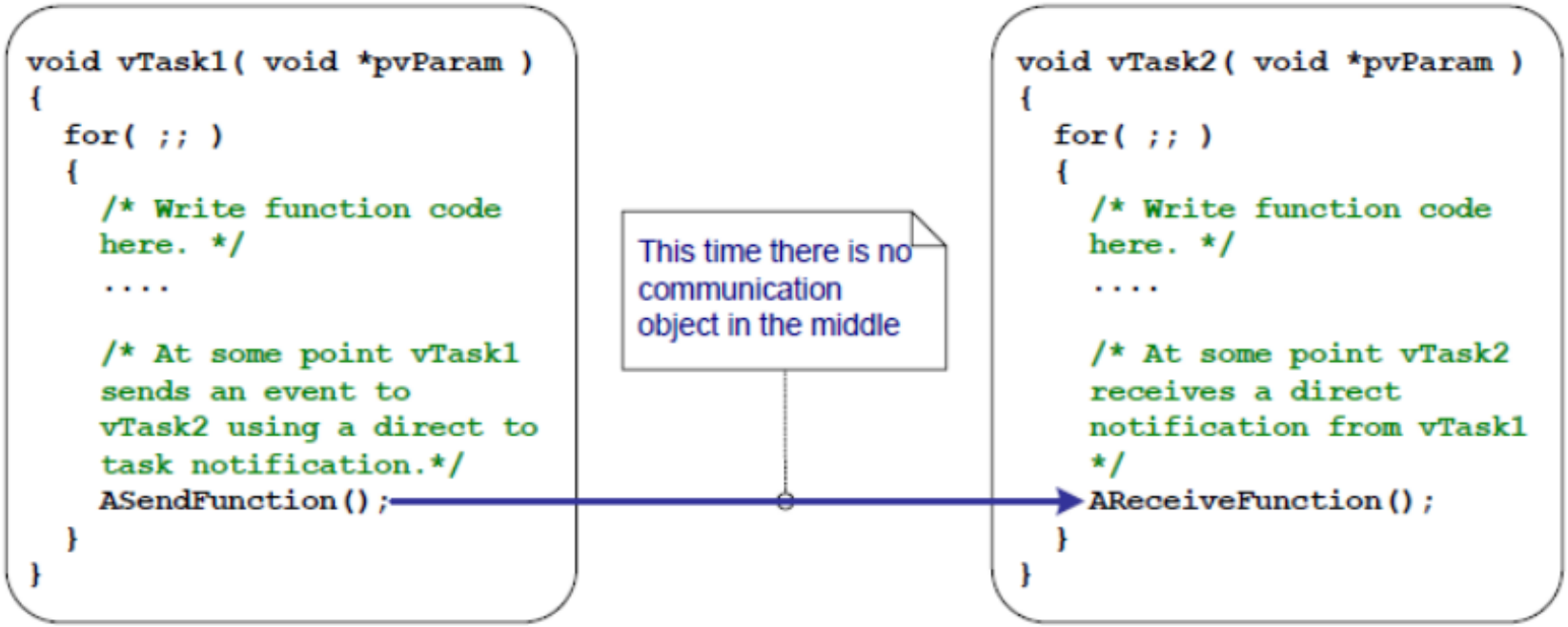


# Task notification: problem

When a communication object is used, events and data are not sent directly to a receiving task, or a receiving ISR, but are instead sent to the communication object.

# Task notification: solution

'Task Notifications' allow tasks to interact with other tasks, and to synchronize with ISRs, without the need for a separate communication object. By using a task notification, a task or ISR can send an event directly to the receiving task



# Task Notifications: Benefits

- Using a task notification to send an event or data to a task is **significantly faster** than equivalent operation.
- Likewise, using a task notification to send an event or data to a task requires **significantly less RAM** than using a queue, semaphore or event group.
- The RAM cost for task notifications is configTASK_NOTIFICATION_ARRAY_ENTRIES * 5 bytes per task.

When configUSE_TASK_NOTIFICATIONS is set to 1, each task has at least one 'Notification State', which can be either 'Pending' or 'Not-Pending', and a 'Notification Value', which is a 32-bit unsigned integer. When a task receives a notification, its notification state is set to pending. When a task reads its notification value, its notification state is set to not-pending.

# Task Notifications: Limitations

Task Notification **cannot be used**:

- Sending an event or data to an ISR
- Enabling more than one receiving task
- Buffering multiple data items
- Broadcasting to more than one task
- Waiting in the blocked state for a send to complete

# Task Notifications: how to use

Task notifications are a very powerful feature that **can often be used in place of a binary semaphore, a counting semaphore, an event group, and sometimes even a queue**. This wide range of usage scenarios can be achieved by using the xTaskNotify() API function to send a task notification, and the xTaskNotifyWait() API function to receive a task notification.

# Task Notifications: how to use

However, in the majority of cases, the full flexibility provided by the xTaskNotify() and xTaskNotifyWait() API functions is not required, and simpler functions would suffice. Therefore, the **xTaskNotifyGive()** API function is provided as a simpler but less flexible alternative to xTaskNotify(), and the **ulTaskNotifyTake()** API function is provided as a simpler but less flexible alternative to xTaskNotifyWait().

# Task Notifications: how to use

The task notification system is not limited to a single notification event. The configuration parameter **configTASK_NOTIFICATION_ARRAY_ENTRIES is set to 1 by default**. If it is set to a value greater than 1, an array of notifications are created inside each task. This allows notifications to be managed by index.
Every task notification api function has an indexed version. Using the non-indexed version will result in accessing notification[0] (the first one in the array).
The indexed version of each API function is identified by the suffix Indexed so the function xTaskNotify becomes xTaskNotifyIndexed.

# Task Notifications: how to use

Note: The FromISR functions do not exist for receiving notifications because a notification is always sent to a task and interrupts are not associated with any task.

# xTaskNotifyGive()

xTaskNotifyGive() sends a notification directly to a task, and increments (adds one to) the receiving task's notification value. Calling xTaskNotifyGive() will set the receiving task's notification state to pending, if it was not already pending.

```
BaseType_t xTaskNotifyGive( TaskHandle_t xTaskToNotify );
BaseType_t xTaskNotifyGiveIndexed( xTaskHandle_t xTaskToNotify,
                                   UBaseType_t uxIndexToNotify );
```

xTaskToNotify The handle of the task to which the notification is being sent.

uxIndexToNotify The index into the array

return pdPASS is the only possible return value.

# xTaskNotifyGiveFromISR()

vTaskNotifyGiveFromISR() is a version of xTaskNotifyGive() that can be used in an interrupt service routine.

```
void vTaskNotifyGiveFromISR( TaskHandle_t xTaskToNotify,
                             BaseType_t *pxHigherPriorityTaskWoken );
```

pxHigherPriorityTaskWoken If the task to which the notification is being sent is waiting in the Blocked state to receive a notification, then sending the notification will cause the task to leave the Blocked state. If calling vTaskNotifyGiveFromISR() causes a task to leave the Blocked state, and the unblocked task has a priority higher than the priority of the currently executing task (the task that was interrupted), then, internally, vTaskNotifyGiveFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE.

# ulTaskNotifyTake()

ulTaskNotifyTake() allows a task to wait in the Blocked state for its notification value to be greater than zero, and either decrements (subtracts one from) or clears the task's notification value before it returns.

**The ulTaskNotifyTake() API function is provided to allow a task notification to be used as a lighter weight and faster alternative to a binary or counting semaphore.**

```
uint32_t ulTaskNotifyTake( BaseType_t xClearCountOnExit,
                           TickType_t xTicksToWait );
```

# ulTaskNotifyTake()

xClearCountOnExit If xClearCountOnExit is set to pdTRUE, then the calling task's notification value will be cleared to zero before the call to ulTaskNotifyTake() returns. If xClearCountOnExit is set to pdFALSE, and the calling task's notification value is greater than zero, then the calling task's notification value will be decremented before the call to ulTaskNotifyTake() returns.

xTicksToWait The maximum amount of time the calling task should remain in the Blocked state to wait for its notification value to be greater than zero.

# ulTaskNotifyTake()

return The returned value is the calling task's notification value before it was either cleared to zero or decremented, as specified by the value of the xClearCountOnExit parameter.
If a block time was specified (xTicksToWait was not zero), and the return value is not zero, then it is possible the calling task was placed into the Blocked state to wait for its notification value to be greater than zero, and its notification value was updated before the block time expired.

# Example 24: vHandlerTask

```
/* The rate at which the periodic task generates software interrupts. */
const TickType_t xInterruptFrequency = pdMS_TO_TICKS( 500UL );
static void vHandlerTask( void *pvParameters )
{
/* xMaxExpectedBlockTime is set to be a little longer than the maximum
expected time between events. */
const TickType_t xMaxExpectedBlockTime = xInterruptFrequency +
pdMS_TO_TICKS( 10 );
uint32_t ulEventsToProcess;
/* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* Wait to receive a notification sent directly to this task from the
        interrupt service routine. */
        ulEventsToProcess = ulTaskNotifyTake( pdTRUE, xMaxExpectedBlockTime );
        if( ulEventsToProcess != 0 )
        {
            /* To get here at least one event must have occurred. Loop here
            until all the pending events have been processed (in this case,
            just print out a message for each event). */
            while( ulEventsToProcess > 0 )
            {
                vPrintString( "Handler task - Processing event.\r\n" );
                ulEventsToProcess--;
            }
        }
```

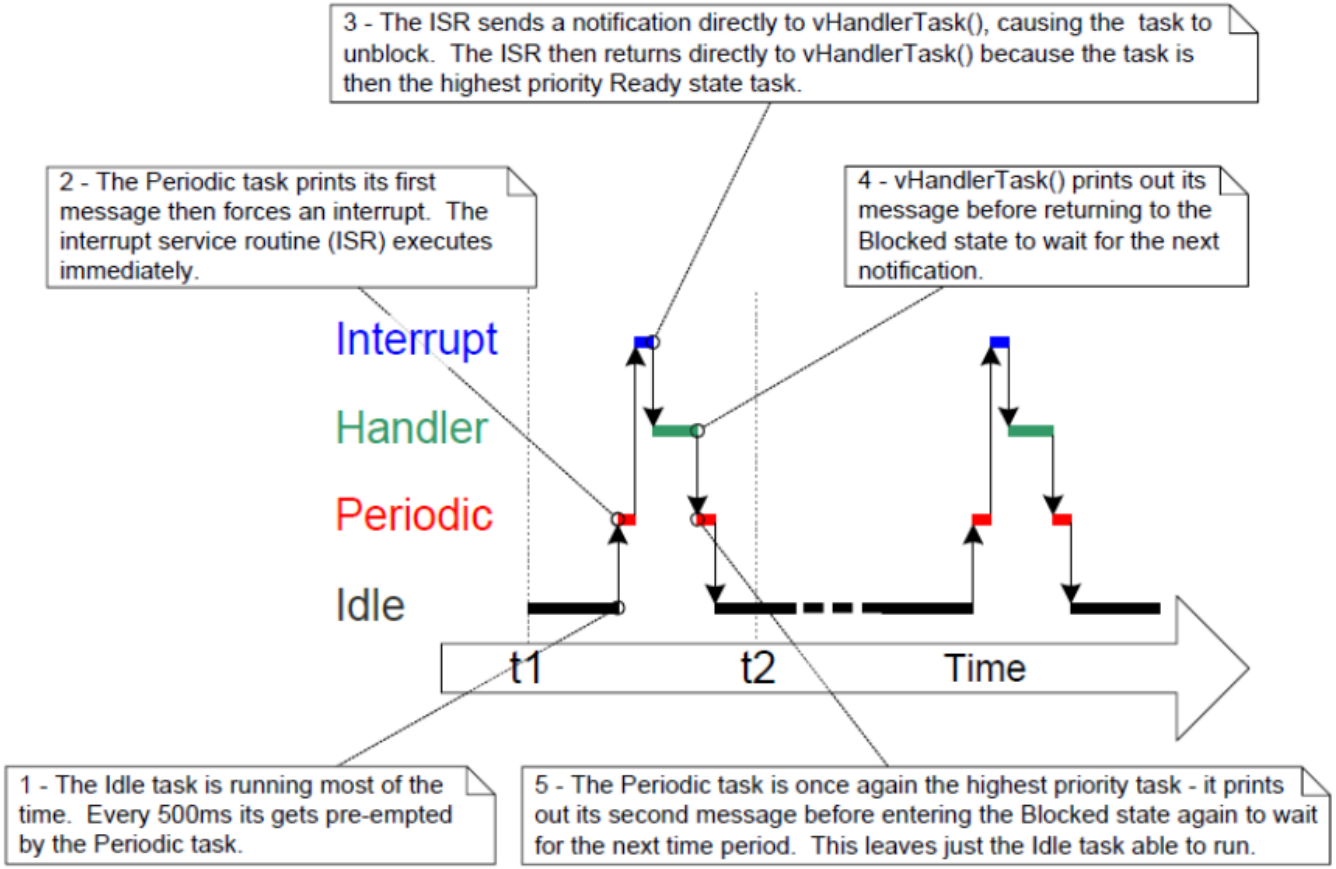
# Example 24: vHandlerTask

```
else
        {
            /* If this part of the function is reached then an interrupt did
            not arrive within the expected time, and (in a real application)
            it may be necessary to perform some error recovery operations. */
        }
    }
}
```

# Example 24: ulExampleInterruptHandler

```
static uint32_t ulExampleInterruptHandler( void )
{
    BaseType_t xHigherPriorityTaskWoken;
    /* 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. */
    xHigherPriorityTaskWoken = pdFALSE;
    /* Send a notification directly to the task to which interrupt processing
    is being deferred. */
    vTaskNotifyGiveFromISR( /* The handle of the task to which the notification
                            is being sent. The handle was saved when the task
                            was created. */
                            xHandlerTask,
                            /* xHigherPriorityTaskWoken is used in the usual
                            way. */
                            &xHigherPriorityTaskWoken );
    /* Pass the xHigherPriorityTaskWoken value into portYIELD_FROM_ISR(). If
    xHigherPriorityTaskWoken was set to pdTRUE inside vTaskNotifyGiveFromISR()
    then calling portYIELD_FROM_ISR() will request a context switch. If
    xHigherPriorityTaskWoken is still pdFALSE then calling
    portYIELD_FROM_ISR() will have no effect. The implementation of
    portYIELD_FROM_ISR() used by the Windows port includes a return statement,
    which is why this function does not explicitly return a value. */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

# Example 24: timing



# Troubleshooting: scope

These slides highlight the most common issues encountered by users who are new to FreeRTOS. First, they focuse on three issues that have proven to be the most frequent source of support requests over the years:

- incorrect interrupt priority assignment
- stack overflow
- inappropriate use of printf()

Using configASSERT() improves productivity by immediately trapping and identifying many of the most common sources of error. It is strongly advised to have configASSERT() defined while developing or debugging a FreeRTOS application.

# Interrupt priority

If the FreeRTOS port in use supports interrupt nesting, and the service routine for an interrupt makes use of the FreeRTOS API, then **it is essential the interrupt's priority is set at or below configMAX_SYSCALL_INTERRUPT_PRIORITY**.

- Interrupt priorities default to having the highest possible priority, which is the case on some ARM Cortex processors, and possibly others. On such processors, the priority of an interrupt that uses the FreeRTOS API cannot be left uninitialized.

# Interrupt priority

- Numerically high priority numbers represent logically low interrupt priorities, which may seem counterintuitive, and therefore cause confusion. Again this is the case on ARM Cortex processors, and possibly others.
- For example, on such a processor an interrupt that is executing at priority 5 can itself be interrupted by an interrupt that has a priority of 4. Therefore, if configMAX_SYSCALL_INTERRUPT_PRIORITY is set to 5, any interrupt that uses the FreeRTOS API can only be assigned a priority numerically higher than or equal to 5. In that case, interrupt priorities of 5 or 6 would be valid, but an interrupt priority of 3 is definitely invalid.

# Interrupt priority

- Different library implementations expect the priority of an interrupt to be specified in a different way. Again, this is particularly relevant to libraries that target ARM Cortex processors, where interrupt priorities are bit shifted before being written to the hardware registers.
- Different implementations of the same architecture implement a different number of interrupt priority bits. For example, a Cortex-M processor from one manufacturer may implement 3 priority bits, while a Cortex-M processor from another manufacturer may implement 4 priority bits.

# Interrupt priority

- The bits that define the priority of an interrupt can be split between bits that define a pre-emption priority, and bits that define a sub-priority. Ensure all the bits are assigned to specifying a pre-emption priority, so that sub-priorities are not used.

# Stack overflow

Stack overflow is the second most common source of support requests. FreeRTOS provides several features to assist trapping and debugging stack related issues.
Each task maintains its own stack, the total size of which is specified when the task is created:
**uxTaskGetStackHighWaterMark() is used to query how close a task has come to overflowing the stack space allocated to it**.

```
UBaseType_t uxTaskGetStackHighWaterMark( TaskHandle_t xTask );
```

returns the **minimum amount of remaining stack space** that has been available since the task started executing.

# Stack overflow

FreeRTOS includes three optional run time stack checking mechanisms. These are controlled by the configCHECK_FOR_STACK_OVERFLOW compile time configuration constant within FreeRTOSConfig.h.

Both methods increase the time it takes to perform a context switch.

The stack overflow hook (or stack overflow callback) is a function that is called by the kernel when it detects a stack overflow.

```
void vApplicationStackOverflowHook(TaskHandle_t *pxTask,
                                   signed char *pcTaskName);
```

# Stack overflow

- Set configCHECK_FOR_STACK_OVERFLOW to either 1, 2 or 3.
- Provide the implementation of the hook function

The stack overflow hook is provided to make trapping and debugging stack errors easier, but there is no real way to recover from a stack overflow when it occurs. **The function's parameters pass the handle and name of the task that has overflowed** its stack into the hook function. The stack overflow hook gets called from the context of an interrupt.

# Stack overflow: method 1

A task's entire execution context is saved onto its stack each time it gets swapped out. It is likely that this will be the time at which stack usage reaches its peak. The kernel checks that the stack pointer remains within the valid stack space after the context has been saved. The stack overflow hook is called if the stack pointer is found to be outside its valid range.
Method 1 is quick to execute, but can miss stack overflows that occur between context switches.

# Stack overflow: method 2

Method 2 performs additional checks to those already described for method 1.
When a task is created, its stack is filled with a known pattern. Method 2 tests the last valid 20 bytes of the task stack space to verify that this pattern has not been overwritten. The stack overflow hook function is called if any of the 20 bytes have changed from their expected values.
Method 2 is not as quick to execute as method 1, but is still relatively fast, as only 20 bytes are tested. Most likely, it will catch all stack overflows; however, it is possible (but highly improbable) that some overflows will be missed.

# Stack overflow: method 3

This method is available only for selected ports. When available, this method enables ISR stack checking. When an ISR stack overflow is detected, an assert is triggered. Note that the stack overflow hook function is not called in this case because it is specific to a task stack and not the ISR stack.

# Misuse of printf()

Logging via printf() is a common source of error, and, unaware of this, it is common for application developers to then add further calls to printf() to aid debugging, and in-so-doing, exacerbate the problem.

Many cross compiler vendors will provide a printf() implementation that is suitable for use in small embedded systems. Even when that is the case, the implementation may not be thread safe, probably won't be suitable for use inside an interrupt service routine, and depending on where the output is directed, take a relatively long time to execute.

# Misuse of printf()

Particular care must be taken if a printf() implementation that is specifically designed for small embedded systems is not available, and a generic printf() implementation is used instead, as:

- Just including a call to printf() or sprintf() can massively increase the size of the application's executable.
- printf() and sprintf() may call malloc(), which might be invalid if a memory allocation scheme otherthan heap_3 is in use.
- printf() and sprintf() may require a stack that is many times bigger than would otherwise be required.

# Other sources of errors

- Adding a simple task to a demo causes the demo to crash
- Using an API function within an interrupt causes the application to crash
- Sometimes the application crashes within an interrupt service routine
- Interrupts are unexpectedly left disabled, or critical sections do not nest correctly
- The application crashes even before the scheduler is started
- Calling API functions while the scheduler is suspended, or from inside a critical section, causes the application to crash

# Adding a simple task to a demo causes the demo to crash

Creating a task requires memory to be obtained from the heap. Many of the demo application projects dimension the heap to be exactly big enough to create the demo tasks—so, after the tasks are created, there will be insufficient heap remaining for any further tasks, queues, event groups, or semaphores to be added.

To be able to add more tasks, you must either increase the heap size, or remove some of the existing demo tasks.

# Using an API function within an interrupt causes the application to crash

Do not use API functions within interrupt service routines, unless the name of the API function ends with '...FromISR()'. In particular, do not create a critical section within an interrupt unless using the interrupt safe macros.

In FreeRTOS ports that support interrupt nesting, do not use any API functions in an interrupt that has been assigned an interrupt priority above configMAX_SYSCALL_INTERRUPT_PRIORITY.

# Sometimes the application crashes within an interrupt service routine

The first thing to check is that the interrupt is not causing a stack overflow. Some ports only check for stack overflow within tasks, and not within interrupts.
The way interrupts are defined and used differs between ports and between compilers. Therefore, the second thing to check is that the syntax, macros, and calling conventions used in the interrupt service routine are exactly as described on the documentation page provided for the port being used, and exactly as demonstrated in the demo application provided with the port.

# Interrupts are unexpectedly left disabled, or critical sections do not nest correctly

If a FreeRTOS API function is called before the scheduler has been started then interrupts will deliberately be left disabled, and not re-enabled again until the first task starts to execute. This is done to protect the system from crashes caused by interrupts that attempt to use FreeRTOS API functions during system initialization, before the scheduler has been started, and while the scheduler may be in an inconsistent state.

Do not alter the microcontroller interrupt enable bits or priority flags using any method other than calls to taskENTER_CRITICAL() and taskEXIT_CRITICAL().

# The application crashes even before the scheduler is started

An interrupt service routine that could potentially cause a context switch must not be permitted to execute before the scheduler has been started. The same applies to any interrupt service routine that attempts to send to or receive from a FreeRTOS object, such as a queue or semaphore. A context switch cannot occur until after the scheduler has started.

Many API functions cannot be called until after the scheduler has been started. It is best to restrict API usage to the creation of objects such as tasks, queues, and semaphores, rather than the use of these objects, until after vTaskStartScheduler() has been called.

# Calling API functions while the scheduler is suspended, or from inside a critical section, causes the application to crash

The scheduler is suspended by calling vTaskSuspendAll() and resumed (unsuspended) by calling xTaskResumeAll(). A critical section is entered by calling askENTER_CRITICAL(), and exited by calling taskEXIT_CRITICAL().

Do not call API functions while the scheduler is suspended, or from inside a critical section.