Mastering the Free_Rtos_Real time kernel

* FreeRTOS can be thought of as a library that provide multi-tasking capabilities to what would otherwise a bare metal application.

FreeRTOS
- Source
  - task.c
  - list.c
  - queue.c
  - timer.c
  - event_group.c
  - croutine.c

Include path

1. FreeRTOS/Source/include
2. FreeRTOS/Source/portable
3. FreeRTOSConfig.h

Demo project

(FreeRTOS.org)

FreeRTOS/Demo directory

Data type and coding style guide

Each port of FreeRTOS has a unique portmacrah header file that contains defination for two port specific data types:

TickType_t and BaseType_t

→ FreeRTOS configure a periodic interrupt called the tick interrupt.

→ The number of tick interrupt that have occured since the FreeRTOS application started in called the tick Count.

→ The time between two tick interrupt is called tick period.

→ TickType_t is the data type used to hold the tick count value, and to specify time.

→ BaseType_t is generally used for return type that can take only a very limited range of values, and for pdTrue / pdFalse type Booleans.

Variable names

c → char
s → int16 (short)
l → int32 (long)
x → BaseType_t
u → unsigned
p → pointer
uc → uint8_t
pc → pointer to char

Function Names

- vTaskPrioritySet() returns a void and is defined within task.c
- xQueueReceive() returns a variable of type BaseType_t and is defined within queue.c
- pvTimerGetTimerID() return a pointer to void and is defined within timer.c.

Macro Names

Most macro written in upper case, and prefixed with lower case letters that indicate where the macro is defined.

eg - port (for example, portMAX_DELAY)
→ (portable.h)

pdTRUE → 1 pdFALSE → 0
pdPASS → 1 pdFAIL → 0

→ configCPU_CLOCK_Hz

120 MHz → 1000 → 1ms

(configTICK_RATE_Hz)

120 MHz → 4000 → 250us

$\frac{1}{4000}$ (0.00025 $25 \times 10^{-5} \times \frac{10^1}{10^{-1}}$: 250×10^{-6} : 250us)

→ Define the frequency of RTOS tick interrupt in Hertz. (how many tick interrupt occurs per second)

Reload Value = $\frac{\text{configCPU\_CLOCK\_Hz}}{\text{configTICK\_RATE\_Hz}}$

eg — : $\frac{48\,MHz}{1000} = 48000$

→ Hardware timer will count up to 48000 before generating an interrupt.

→ Mean tick_interrupt occur every $^1/_{1000}$ sec or 1ms.

To get 1us tick → $\frac{1}{10^6} = 10^{-6}$

Task Management

Task Functions

void ATaskFunction (void *pvParameters)

→ It has an entry point, will normally run forever within an infinite loop, and will not exit

→ A single task function defination can be used to create any number of tasks — each created task being a seperate execution instance, with its own stack and its own copy of any automatic (stack) variable defined within the task itself.

→ If a task is no longer required, it should instead be explicitly deleted.

```
void ATaskFunction( void *pvParameters )
{
/* Variables can be declared just as per a normal function.  Each instance of a task
created using this example function will have its own copy of the lVariableExample
variable.  This would not be true if the variable was declared static - in which case
only one copy of the variable would exist, and this copy would be shared by each
created instance of the task. (The prefixes added to variable names are described in
section 1.5, Data Types and Coding Style Guide.) */
int32_t lVariableExample = 0;

    /* A task will normally be implemented as an infinite loop. */
    for( ;; )
    {
        /* The code to implement the task functionality will go here. */
    }

    /* Should the task implementation ever break out of the above loop, then the task
    must be deleted before reaching the end of its implementing function.  The NULL
    parameter passed to the vTaskDelete() API function indicates that the task to be
    deleted is the calling (this) task.  The convention used to name API functions is
    described in section 0, Projects that use a FreeRTOS version older than V9.0.0
must build one of the heap_n.c files.  From FreeRTOS V9.0.0 a heap_n.c file is only
required if configSUPPORT_DYNAMIC_ALLOCATION is set to 1 in FreeRTOSConfig.h or if
configSUPPORT_DYNAMIC_ALLOCATION is left undefined.  Refer to Chapter 2, Heap Memory
Management, for more information.
Data Types and Coding Style Guide. */
    vTaskDelete( NULL );
}
```

Listing 12. The structure of a typical task function

* FreeRTOS queues, binary semaphore, counting semaphores, mutexes, recursive mutexes, event group and direct to task notification can all be used to create synchronization event.

⇒ A task transitioned from the Not Running state to running state to have been 'switched in' or 'swapped in'.

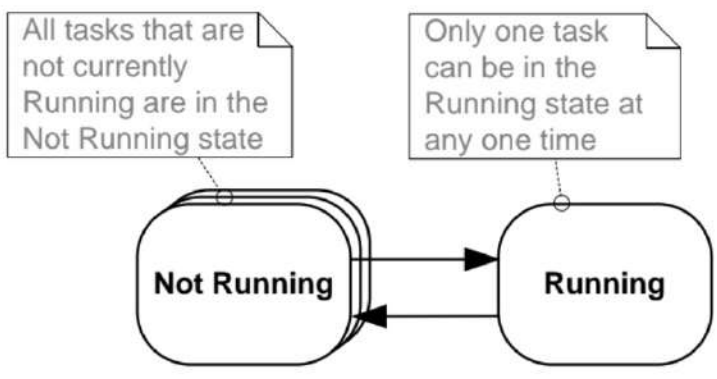


Figure 9. Top level task states and transitions

Creating a task

→ Task are created using the FreeRTOS xTaskCreate() API function.

→ xTaskCreateStatic() function, which allocate the memory required to create a task statically

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        uint16_t usStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *pxCreatedTask );
```

Listing 13. The xTaskCreate() API function prototype

Example 1

→ Two simple task created

→ Task simply printout a string periodically, using a crude null loop to create period delay.

→ Task have same priority.

```
void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\r\n";
volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

Listing 14. Implementation of the first task used in Example 1

```
void vTask2( void *pvParameters )
{
const char *pcTaskName = "Task 2 is running\r\n";
volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

Listing 15. Implementation of the second task used in Example 1

```
int main( void )
{
    /* Create one of the two tasks.  Note that a real application should check
    the return value of the xTaskCreate() call to ensure the task was created
    successfully. */
    xTaskCreate(    vTask1,  /* Pointer to the function that implements the task. */
                    "Task 1",/* Text name for the task.  This is to facilitate
                                debugging only. */
                    1000,    /* Stack depth - small microcontrollers will use much
                                less stack than this. */
                    NULL,    /* This example does not use the task parameter. */
                    1,       /* This task will run at priority 1. */
                    NULL );  /* This example does not use the task handle. */

    /* Create the other task in exactly the same way and at the same priority. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

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

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

Listing 16. Starting the Example 1 tasks

⇒ Output of tasks

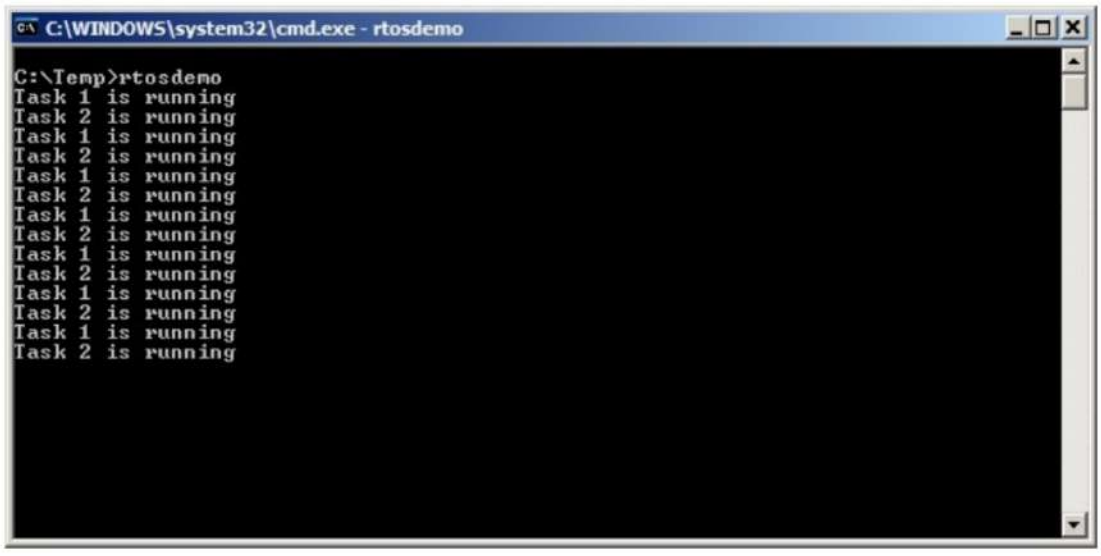


Figure 10. The output produced when Example 1 is executed[1]

→ In reality, both task are rapidly entering and exiting the Running state. Both task are running at the same priority, and so share time on the same processor core.

→ Only one task can exist in Running state at any one time. So, as one task enters the Running state, the other enter the Not-running state

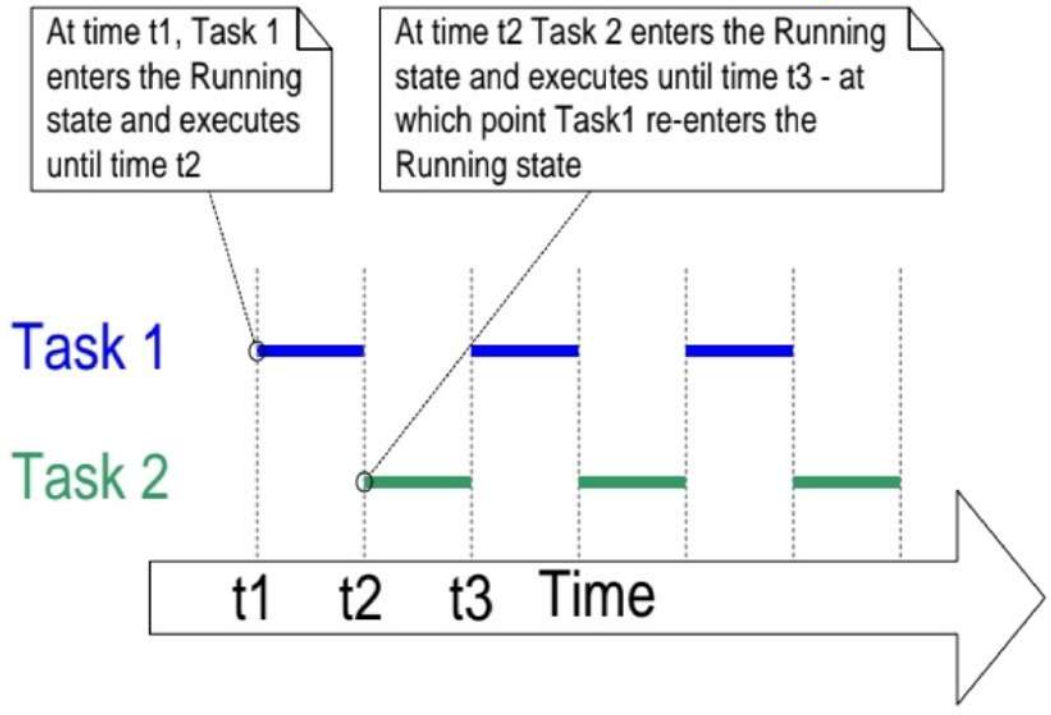


Figure 11. The actual execution pattern of the two Example 1 tasks

- It is also possible to create a task from within another task. Task 2 could have been created from within Task 1.

```
void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\r\n";
volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* If this task code is executing then the scheduler must already have
    been started.  Create the other task before entering the infinite loop. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

Listing 17. Creating a task from within another task after the scheduler has started

Example 2

→ The single task function (vTaskFunction) replace the vTask1 and vTask2 (used in example 1)

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;
volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* The string to print out is passed in via the parameter.  Cast this to a
    character pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later exercises will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

Listing 18. The single task function used to create two tasks in Example 2

```
/* Define the strings that will be passed in as the task parameters.  These are
defined const and not on the stack to ensure they remain valid when the tasks are
executing. */
static const char *pcTextForTask1 = "Task 1 is running\r\n";
static const char *pcTextForTask2 = "Task 2 is running\r\n";

int main( void )
{
    /* Create one of the two tasks. */
    xTaskCreate(    vTaskFunction,          /* Pointer to the function that
                                               implements the task. */
                    "Task 1",               /* Text name for the task.  This is to
                                               facilitate debugging only. */
                    1000,                   /* Stack depth - small microcontrollers
                                               will use much less stack than this. */
                    (void*)pcTextForTask1,  /* Pass the text to be printed into the
                                               task using the task parameter. */
                    1,                      /* This task will run at priority 1. */
                    NULL );                 /* The task handle is not used in this
                                               example. */

    /* Create the other task in exactly the same way.  Note this time that multiple
    tasks are being created from the SAME task implementation (vTaskFunction).  Only
    the value passed in the parameter is different.  Two instances of the same
    task are being created. */
    xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 1, NULL );

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

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

Listing 19. The main() function for Example 2.

⇒ Task Priorities

→ The uxPriority parameter of the xTaskCreate() API function assign an initial priority to the task being created.

→ The priority can be changed after the scheduler has been started by using the vTaskPrioritySet() API function.

→ The range of available priorities is 0 to (configMAX_PRIORITES−1)

⇒ Time measurement and Tick interrupt

→ To be able to select the next task to run, the scheduler itself must execute at the end of each time slice.

→ The length of the time slice is effectively set by the tick interrupt frequency, configTICK_RATE_HZ in FreeRTOSConfig.h.

→ If configTICK_RATE_HZ is set to 100(Hz), then the time slice will be 10 millisecond.

→ The time between two tick interrupt is called the 'tick period'.

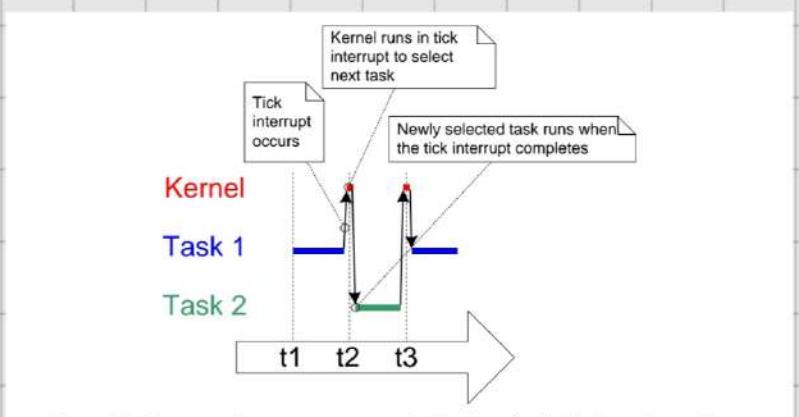


Figure 12. The execution sequence expanded to show the tick interrupt executing

→ The pdMS_TO_TICKS() macro convert a time specified in milliseconds into a time specified in ticks.
→ The pdMS_TO_TICKS() cannot be used if the tick frequency is above 1kHz

```
/* pdMS_TO_TICKS() takes a time in milliseconds as its only parameter, and evaluates
to the equivalent time in tick periods. This example shows xTimeInTicks being set to
the number of tick periods that are equivalent to 200 milliseconds. */
TickType_t xTimeInTicks = pdMS_TO_TICKS( 200 );
```

Listing 20. Using the pdMS_TO_TICKS() macro to convert 200 milliseconds into an equivalent time in tick periods

Example 3 (Experimenting with priorities)

→ Two task created with different priorities (using Task function)

```
/* Define the strings that will be passed in as the task parameters. These are
defined const and not on the stack to ensure they remain valid when the tasks are
executing. */
static const char *pcTextForTask1 = "Task 1 is running\r\n";
static const char *pcTextForTask2 = "Task 2 is running\r\n";

int main( void )
{
    /* Create the first task at priority 1. The priority is the second to last
    parameter. */
    xTaskCreate( vTaskFunction, "Task 1", 1000, (void*)pcTextForTask1, 1, NULL );

    /* Create the second task at priority 2, which is higher than a priority of 1.
    The priority is the second to last parameter. */
    xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 2, NULL );

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

    /* Will not reach here. */
    return 0;
}
```

Listing 21. Creating two tasks at different priorities

→ The scheduler will always select the highest priority task that is able to run.
→ Task 1 never enters the running state, Task 2 is the only task to ever enter the running state.

→ Task 2 is always able to run because it never has to wait.

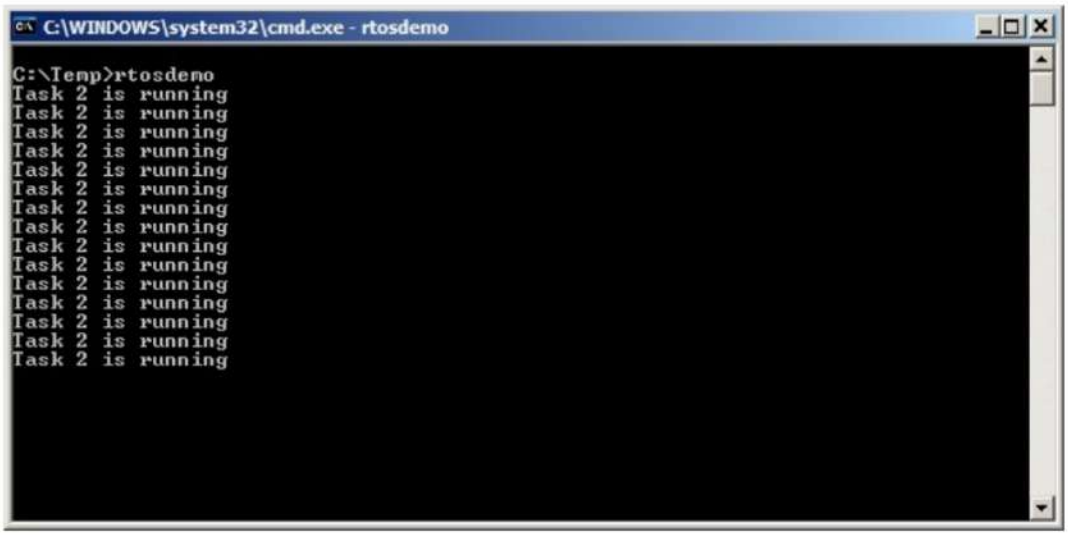


Figure 13. Running both tasks at different priorities

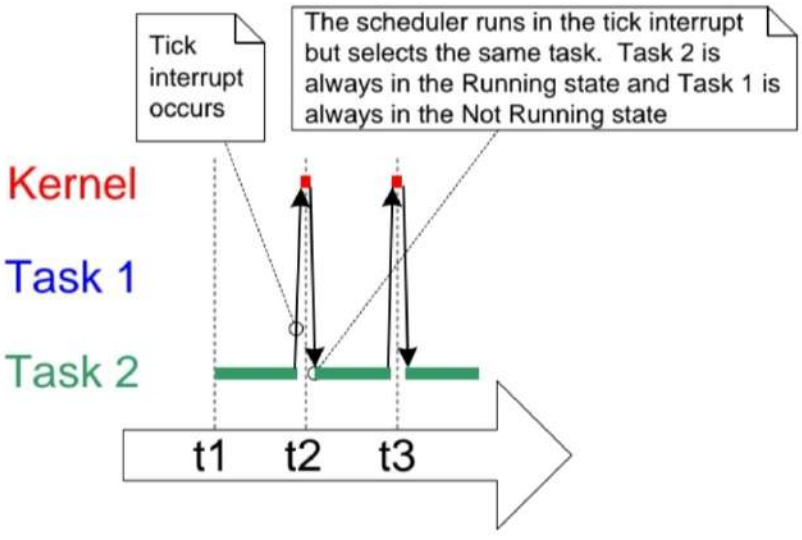


Figure 14. The execution pattern when one task has a higher priority than the other

An Event-driven task — An event-driven task has work (processing) to perform only after the occurrence of the event that trigger it, and is not able to enter the Running state before that event has occured.

The Blocked state — A Task that is waiting for an event is said to be in blocked state, which is sub-state of Not running state.

Task can enter the Blocked state to wait for two different types of event.

1. Temporal (time related) events — A task may enter the Blocked state to wait for 10 millisecond to pass.
2. Synchronization events — A task may enter the blocked state to wait for data to arrive on a queue. Synchronization events cover a broad range of event types.
   → FreeRTOS queue, binary semaphores, counting semaphores, mutexes, recursive mutexes, event group and direct to task notification can all be used to create synchronization event.

3. The suspended State — Tasks in the suspended state are not available to the scheduler. The only way into the suspended state is through a call to the vTaskSuspend() API function, the only way out being through a call to the vTaskResume() or

xTaskResumeFromISR() API function.

4. Task that are in the Not Running state but are not Blocked or suspended are said to be in the Ready state.

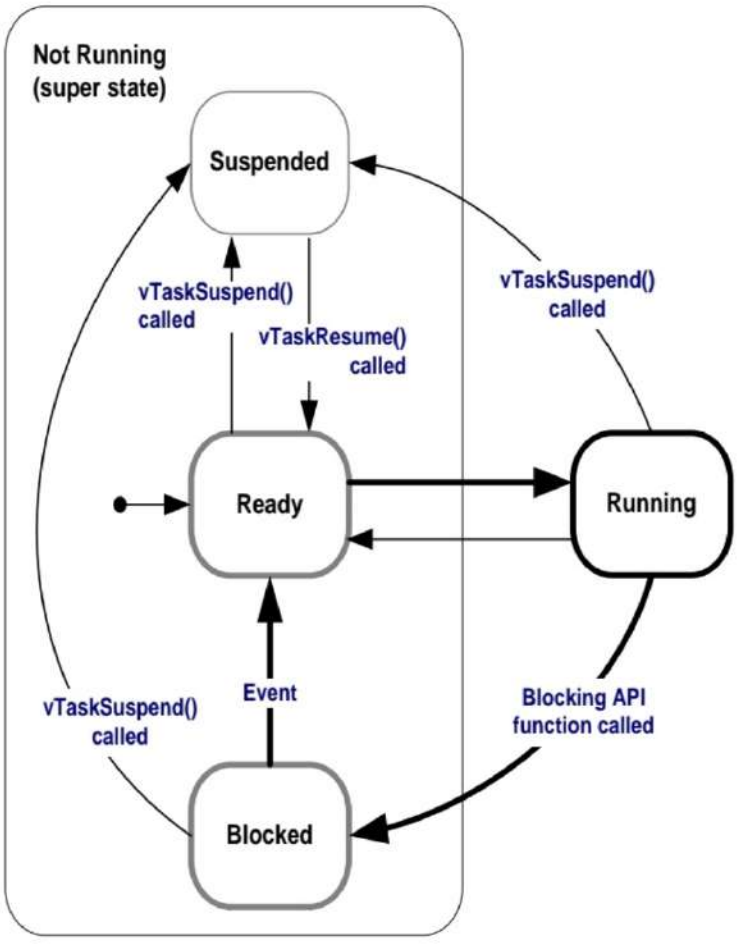


Figure 18. Bold lines indicate the state transitions performed by the tasks in Example 4

vTaskDelay() places the calling task into the Blocked for a fixed number of tick interrupt.

```
void vTaskDelayUntil( TickType_t * pxPreviousWakeTime, TickType_t xTimeIncrement );
```

Listing 24. vTaskDelayUntil() API function prototype

Example 4

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;
const TickType_t xDelay250ms = pdMS_TO_TICKS( 250 );

    /* The string to print out is passed in via the parameter.  Cast this to a
    character pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period.  This time a call to vTaskDelay() is used which places
        the task into the Blocked state until the delay period has expired.  The
        parameter takes a time specified in 'ticks', and the pdMS_TO_TICKS() macro
        is used (where the xDelay250ms constant is declared) to convert 250
        milliseconds into an equivalent time in ticks. */
        vTaskDelay( xDelay250ms );
    }
}
```

Listing 23. The source code for the example task after the null loop delay has been replaced by a call to vTaskDelay()

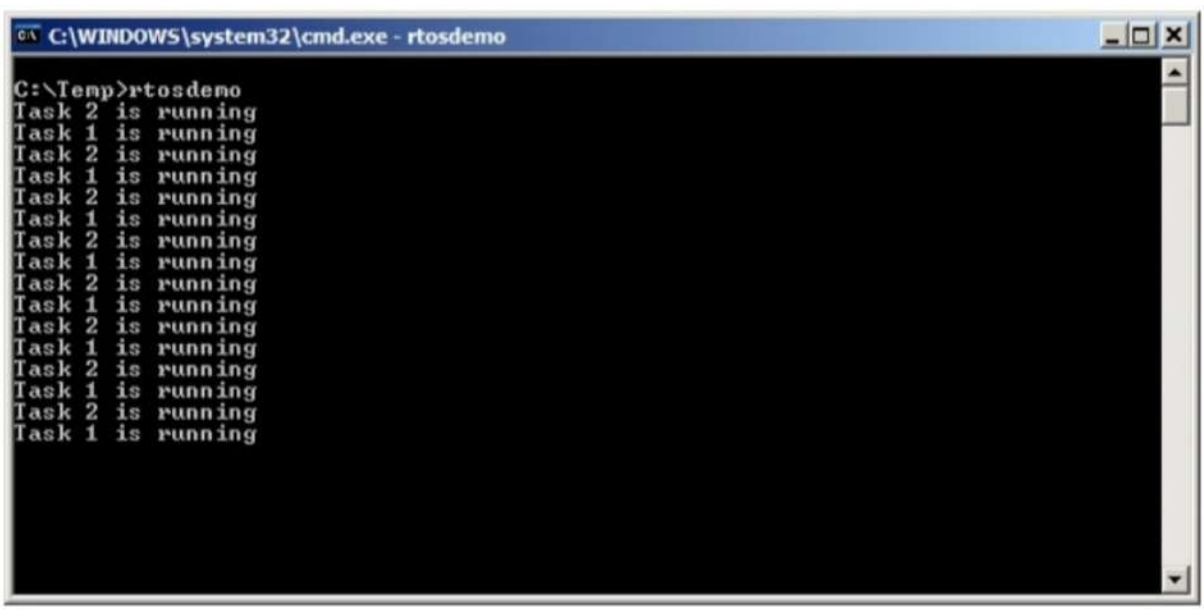


Figure 16. The output produced when Example 4 is executed

→ The idle task is created automatically when the scheduler is started, to ensure there is always at least one task that is able to run. (at least one task in ready state)

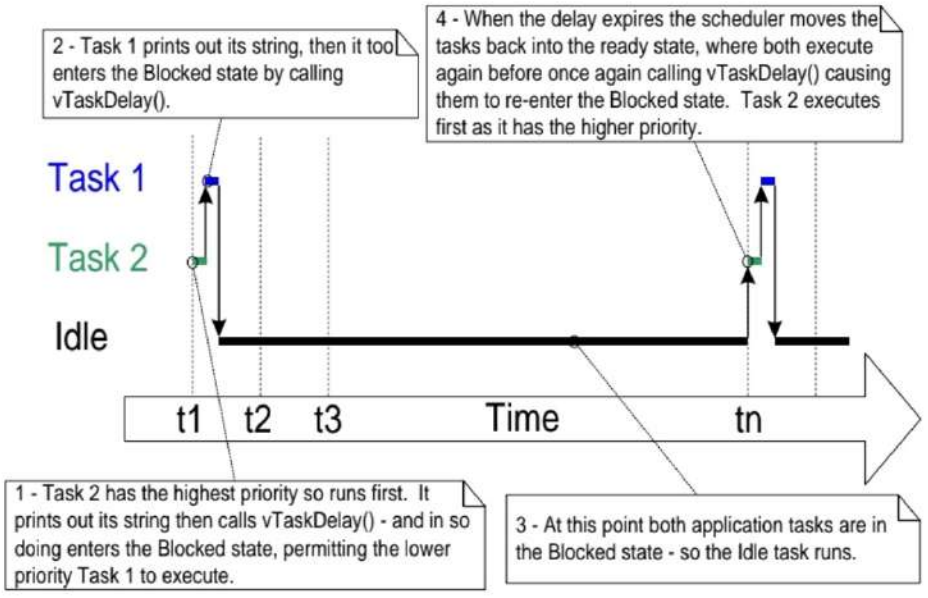


Figure 17. The execution sequence when the tasks use vTaskDelay() in place of the NULL loop

vTaskDelayUntil() — API function that should be used when a fixed execution period in required.

→ The two task created in Example 4 are periodic task, but using vTaskDelay() doesn't guarantee that the frequency at which they run in fixed, as the time at which the task leave the blocked state is relative to when they call vTaskDelay()

Example-5

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;
TickType_t xLastWakeTime;

    /* The string to print out is passed in via the parameter.  Cast this to a
    character pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* The xLastWakeTime variable needs to be initialized with the current tick
    count.  Note that this is the only time the variable is written to explicitly.
    After this xLastWakeTime is automatically updated within vTaskDelayUntil(). */
    xLastWakeTime = xTaskGetTickCount();

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* This task should execute every 250 milliseconds exactly.  As per
        the vTaskDelay() function, time is measured in ticks, and the
        pdMS_TO_TICKS() macro is used to convert milliseconds into ticks.
        xLastWakeTime is automatically updated within vTaskDelayUntil(), so is not
        explicitly updated by the task. */
        vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 250 ) );
    }
}
```

Listing 25. The implementation of the example task using vTaskDelayUntil()

→ Changing the priority of a Task

→ The vTaskPrioritySet() API function can be used to change the priority of any task after the scheduler has been started.

→ The uxTaskPriorityGet() API function can be used to query the priority of a task.

→ The uxTaskPriorityGet() API function in available only when INCLUDE_uxTaskPriorityGet in set to 1 in FreeRTOSconfig.h

```
void vTaskPrioritySet( TaskHandle_t pxTask, UBaseType_t uxNewPriority );
```

Listing 31. The vTaskPrioritySet() API function prototype

Example-8

```
void vTask1( void *pvParameters )
{
UBaseType_t uxPriority;

    /* This task will always run before Task 2 as it is created with the higher
    priority.  Neither Task 1 nor Task 2 ever block so both will always be in
    either the Running or the Ready state.

    Query the priority at which this task is running - passing in NULL means
    "return the calling task's priority". */
    uxPriority = uxTaskPriorityGet( NULL );

    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Task 1 is running\r\n" );

        /* Setting the Task 2 priority above the Task 1 priority will cause
        Task 2 to immediately start running (as then Task 2 will have the higher
        priority of the two created tasks).  Note the use of the handle to task
        2 (xTask2Handle) in the call to vTaskPrioritySet().  Listing 35 shows how
        the handle was obtained. */
        vPrintString( "About to raise the Task 2 priority\r\n" );
        vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );

        /* Task 1 will only run when it has a priority higher than Task 2.
        Therefore, for this task to reach this point, Task 2 must already have
        executed and set its priority back down to below the priority of this
        task. */
    }
}
```

Listing 33. The implementation of Task 1 in Example 8

```
void vTask2( void *pvParameters )
{
UBaseType_t uxPriority;

    /* Task 1 will always run before this task as Task 1 is created with the
    higher priority.  Neither Task 1 nor Task 2 ever block so will always be
    in either the Running or the Ready state.

    Query the priority at which this task is running - passing in NULL means
    "return the calling task's priority". */
    uxPriority = uxTaskPriorityGet( NULL );

    for( ;; )
    {
        /* For this task to reach this point Task 1 must have already run and
        set the priority of this task higher than its own.

        Print out the name of this task. */
        vPrintString( "Task 2 is running\r\n" );

        /* Set the priority of this task back down to its original value.
        Passing in NULL as the task handle means "change the priority of the
        calling task".  Setting the priority below that of Task 1 will cause
        Task 1 to immediately start running again – pre-empting this task. */
        vPrintString( "About to lower the Task 2 priority\r\n" );
        vTaskPrioritySet( NULL, ( uxPriority - 2 ) );
    }
}
```

Listing 34. The implementation of Task 2 in Example 8

```
/* Declare a variable that is used to hold the handle of Task 2. */
TaskHandle_t xTask2Handle = NULL;

int main( void )
{
    /* Create the first task at priority 2.  The task parameter is not used
    and set to NULL.  The task handle is also not used so is also set to NULL. */
    xTaskCreate( vTask1, "Task 1", 1000, NULL, 2, NULL );
    /* The task is created at priority 2 ______^. */

    /* Create the second task at priority 1 - which is lower than the priority
    given to Task 1.  Again the task parameter is not used so is set to NULL -
    BUT this time the task handle is required so the address of xTask2Handle
    is passed in the last parameter. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, &xTask2Handle );
    /* The task handle is the last parameter _____^^^^^^^^^^^^^ */

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

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

Listing 35. The implementation of main() for Example 8

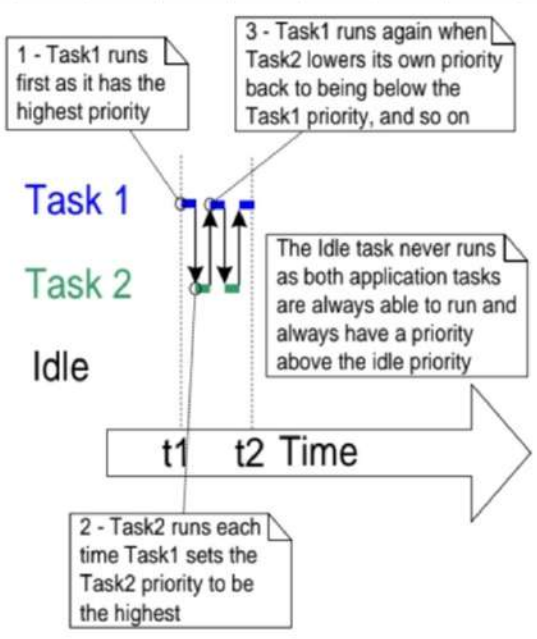


Figure 22. The sequence of task execution when running Example 8

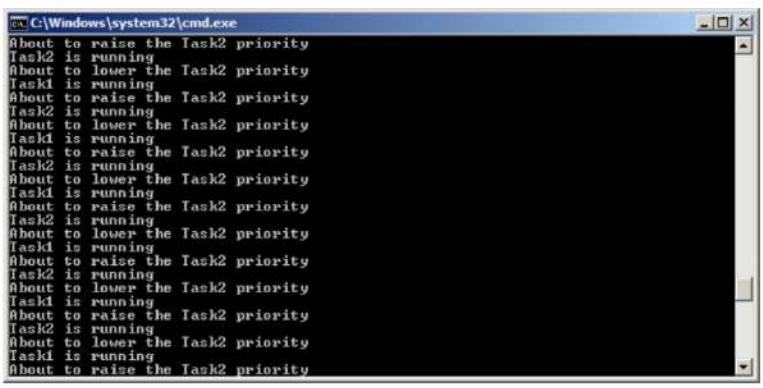


Figure 23. The output produced when Example 8 is executed

→ Deleting a task

→ The vTaskDelete() API function is used to delete itself or any other task.

→ This API is available only when INCLUDE_vTaskDelete is set to 1 in FreeRTOSConfig.h.

Example 9

```
int main( void )
{
    /* Create the first task at priority 1.  The task parameter is not used
    so is set to NULL.  The task handle is also not used so likewise is set
    to NULL. */
    xTaskCreate( vTask1, "Task 1", 1000, NULL, 1, NULL );
    /* The task is created at priority 1 ______^. */

    /* Start the scheduler so the task starts executing. */
    vTaskStartScheduler();

    /* main() should never reach here as the scheduler has been started. */
    for( ;; );
}
```

Listing 37. The implementation of main() for Example 9

```
TaskHandle_t xTask2Handle = NULL;

void vTask1( void *pvParameters )
{
const TickType_t xDelay100ms = pdMS_TO_TICKS( 100UL );

    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Task 1 is running\r\n" );

        /* Create task 2 at a higher priority.  Again the task parameter is not
        used so is set to NULL - BUT this time the task handle is required so
        the address of xTask2Handle is passed as the last parameter. */
        xTaskCreate( vTask2, "Task 2", 1000, NULL, 2, &xTask2Handle );
        /* The task handle is the last parameter _____^^^^^^^^^^^^ */

        /* Task 2 has/had the higher priority, so for Task 1 to reach here Task 2
        must have already executed and deleted itself.  Delay for 100
        milliseconds. */
        vTaskDelay( xDelay100ms );
    }
}
```

Listing 38. The implementation of Task 1 for Example 9

```
void vTask2( void *pvParameters )
{
    /* Task 2 does nothing but delete itself.  To do this it could call vTaskDelete()
    using NULL as the parameter, but instead, and purely for demonstration purposes,
    it calls vTaskDelete() passing its own task handle. */
    vPrintString( "Task 2 is running and about to delete itself\r\n" );
    vTaskDelete( xTask2Handle );
}
```

Listing 39. The implementation of Task 2 for Example 9

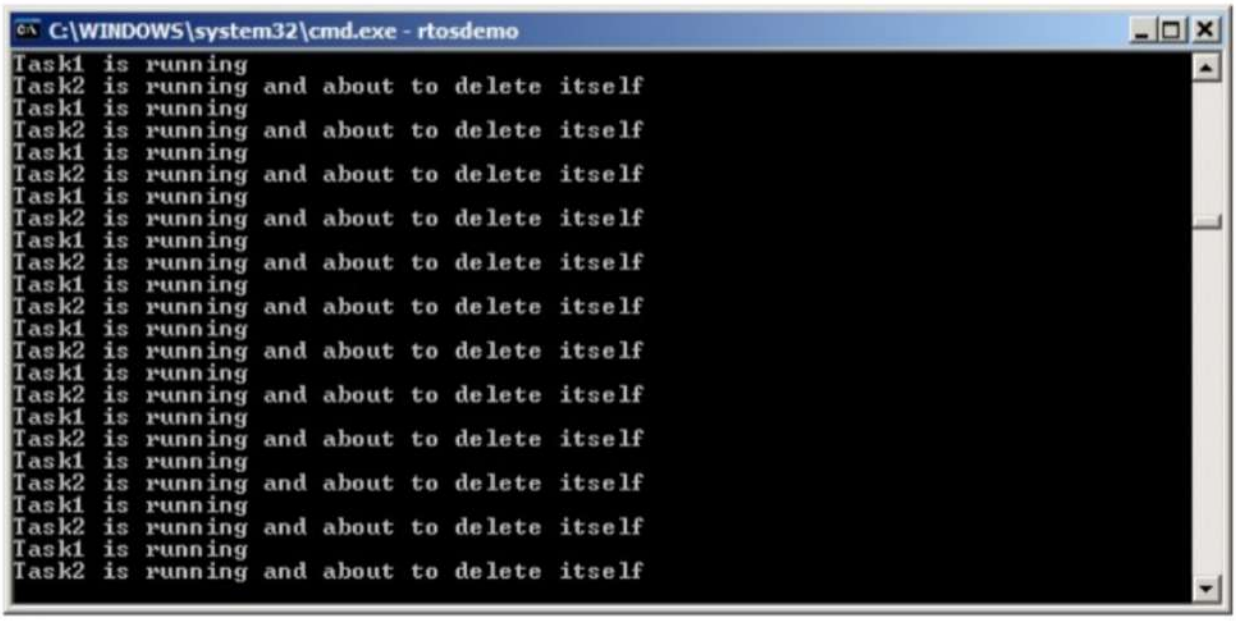


Figure 24. The output produced when Example 9 is executed

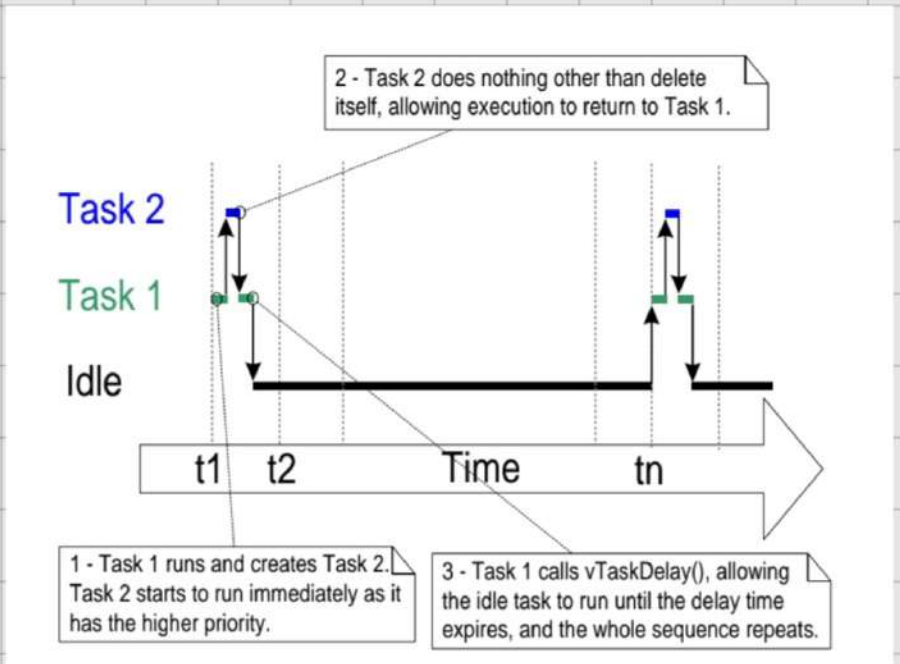


Figure 25. The execution sequence for example 9

Pre-emptive scheduling

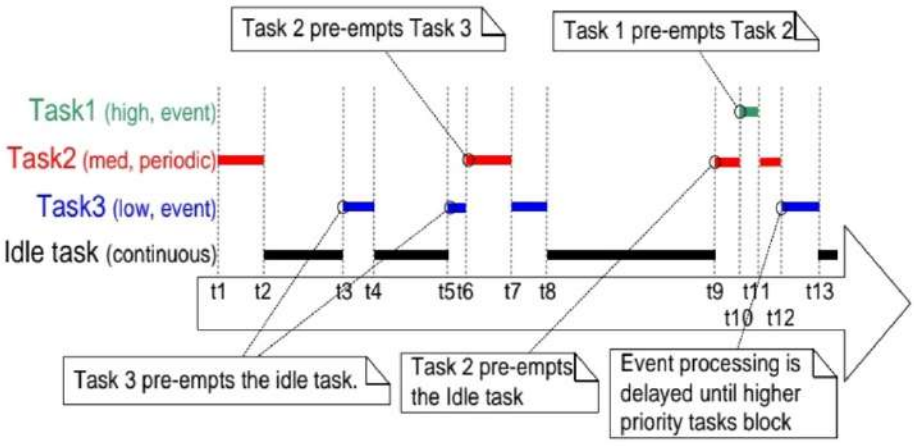


Figure 26. Execution pattern highlighting task prioritization and pre-emption in a hypothetical application in which each task has been assigned a unique priority

Co-operative scheduling

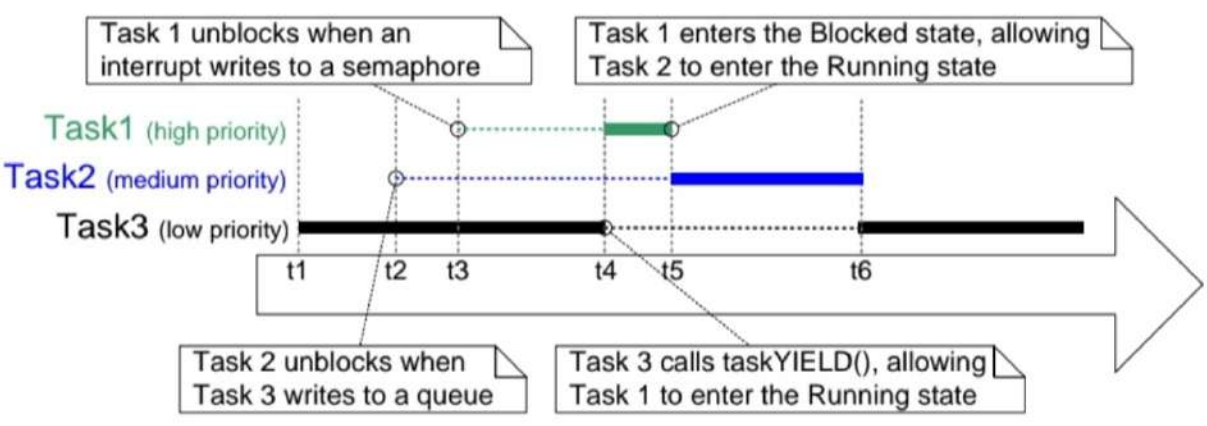


Figure 30 Execution pattern demonstrating the behavior of the co-operative scheduler

Interrupt management

* Aim of this chapter

⇒ which freeRTOS API can be used from ISR
⇒ Creation of binary and counting semaphores.
⇒ How to use queue to pass data into and out of an interrupt.
⇒ Interrupt nesting model.

* Function intended for use from ISRs have "FromISR" append to their name.

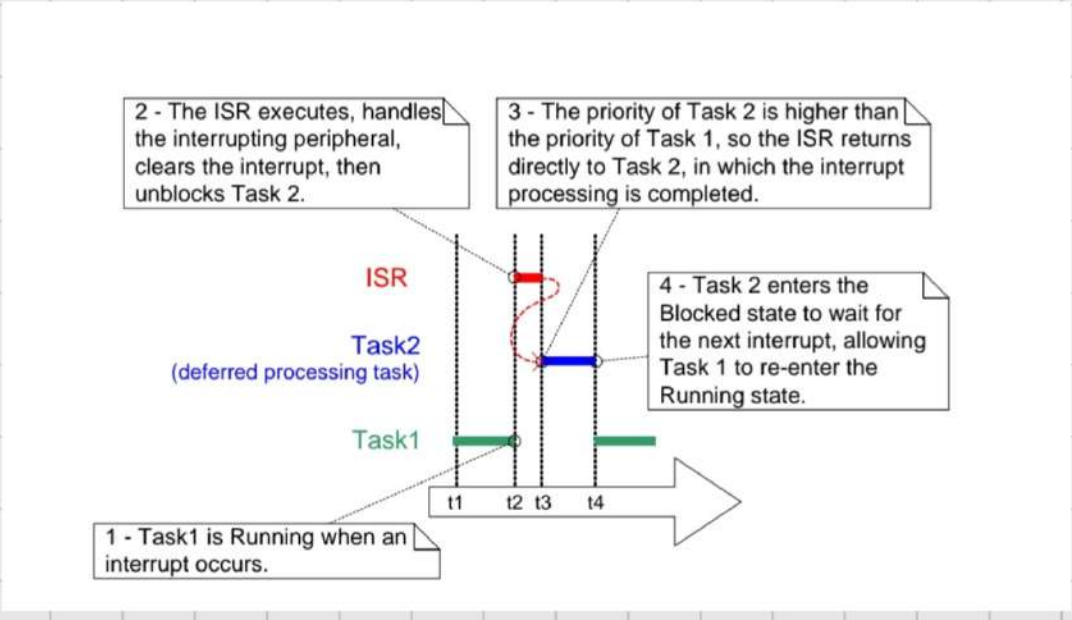
* API function that do not end in "FromISR" must not be called from an ISR

* If a context switch is performed by an interrupt then the task running when the interrupt exists might be different to the task that was running when the interrupt was entered — the interrupt will have interrupted one task, but returned to a different task.

* taskYIELD() is a macro that can be called in a task to request a context switch.
portYIELD_FROM_ISR() and portEND_SWITCHING_ISR() are both interrupt safe version of taskYIELD()

Deffered interrupt processing

⇒ Deffered interrupt processing to a task also allow the application writer to prioritize the processing relative to other task in the application and use all the freeRTOS API function.



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

Binary Semaphore Used for Synchronization

→ Binary semaphore is used to 'defer' interrupt processing to a task.

→ The interrupt safe version of the Binary semaphore API can be used to unblock a task each time a particular interrupt occurs, effectively synchronizing the task with interrupt.

→ The figure describe how the execution of deffered processing task can be controlled using a semaphore.

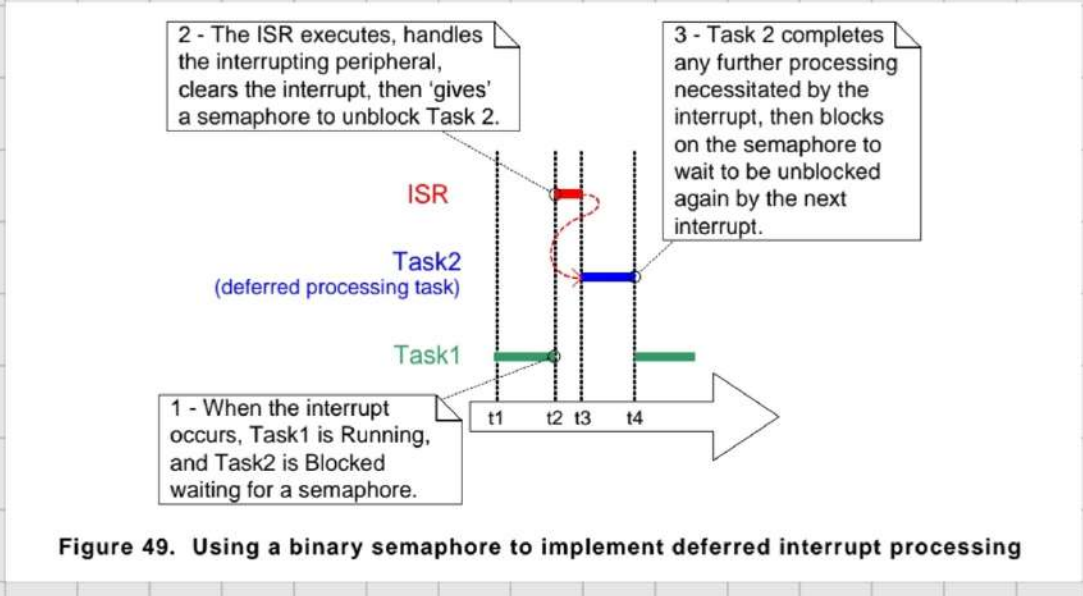


Figure 49. Using a binary semaphore to implement deferred interrupt processing

→ In interrupt synchronization scenario, the binary semaphore can be considered conceptually as a queue with length of one.

Binary semaphore to synchronise task with interrupt

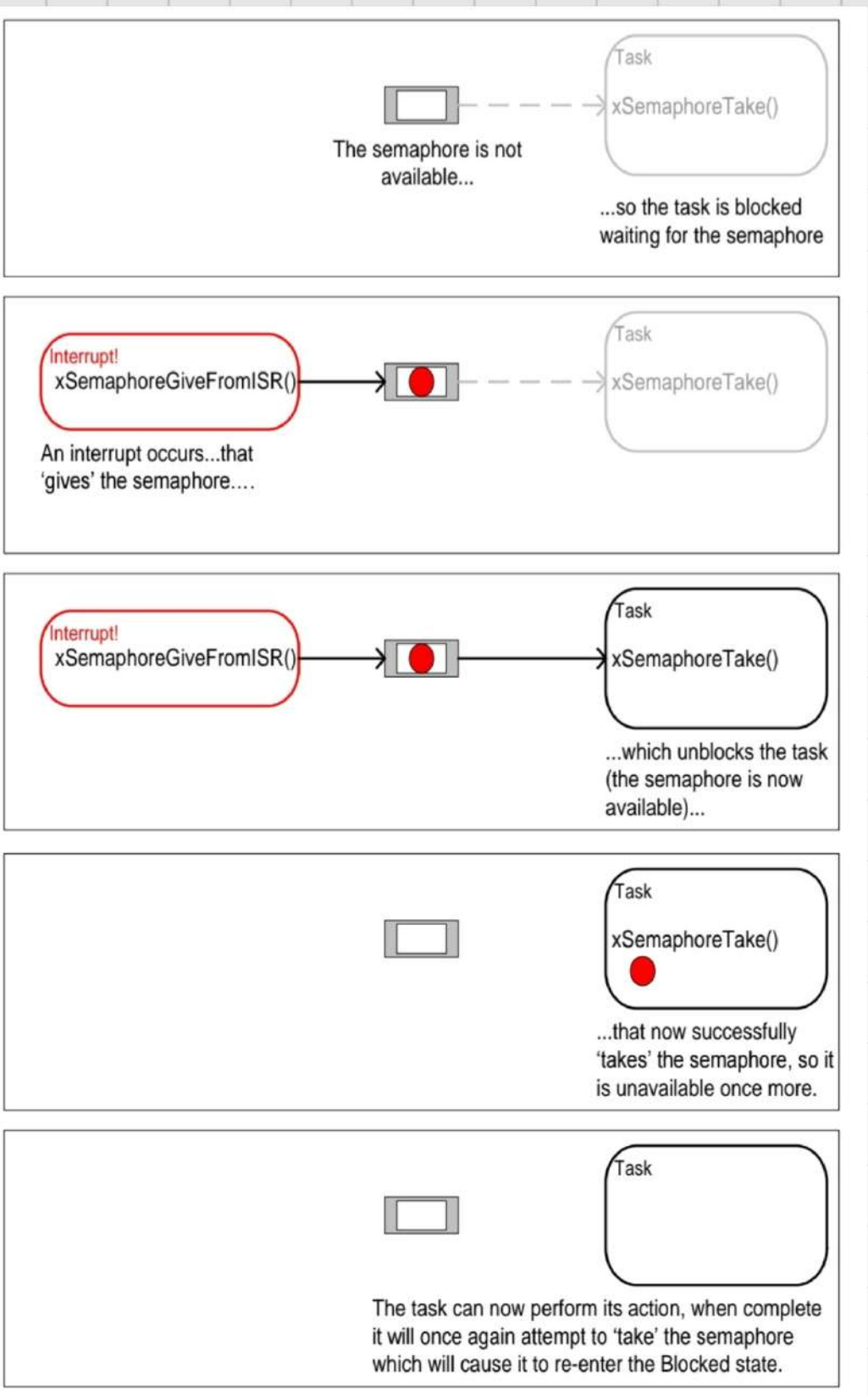


Figure 50. Using a binary semaphore to synchronize a task with an interrupt

→ By calling xSemaphoreTake(), the task to which interrupt processing is deferred effectively attempt to read from the queue with block time, causing the task to enter the Blocked

state, if the queue is empty.

→ when the event occur, the ISR uses the xSemaphoreGiveFromISR() function to place a token (the semaphore) into the queue,

→ This causes the task to exit the blocked state and remove the token, leaving the queue empty once more.

→ when the task has completed its processing, it once more attempt to read from the queue and finding the queue empty, re-enter the blocked state to wait for the next event.

xSemaphoreCreateBinary() → To create binary semaphore.

Example of using binary semaphore to synchronise a task with interrupt

```
/* The number of the software interrupt used in this example.  The code shown is from
the Windows project, where numbers 0 to 2 are used by the FreeRTOS Windows port
itself, so 3 is the first number available to the application. */
#define mainINTERRUPT_NUMBER    3

static void vPeriodicTask( void *pvParameters )
{
const TickType_t xDelay500ms = pdMS_TO_TICKS( 500UL );

    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* Block until it is time to generate the software interrupt again. */
        vTaskDelay( xDelay500ms );

        /* Generate the interrupt, printing a message both before and after
        the interrupt has been generated, so the sequence of execution is evident
        from the output.

        The syntax used to generate a software interrupt is dependent on the
        FreeRTOS port being used.  The syntax used below can only be used with
        the FreeRTOS Windows port, in which such interrupts are only simulated. */
        vPrintString( "Periodic task - About to generate an interrupt.\r\n" );
        vPortGenerateSimulatedInterrupt( mainINTERRUPT_NUMBER );
        vPrintString( "Periodic task - Interrupt generated.\r\n\r\n\r\n" );
    }
}
```

Listing 92. Implementation of the task that periodically generates a software interrupt in Example 16

```
static void vHandlerTask( void *pvParameters )
{
    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* Use the semaphore to wait for the event.  The semaphore was created
        before the scheduler was started, so before this task ran for the first
        time.  The task blocks indefinitely, meaning this function call will only
        return once the semaphore has been successfully obtained - so there is
        no need to check the value returned by xSemaphoreTake(). */
        xSemaphoreTake( xBinarySemaphore, portMAX_DELAY );

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

Listing 93. The implementation of the task to which the interrupt processing is deferred (the task that synchronizes with the interrupt) in Example 16

```
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;

    /* 'Give' the semaphore to unblock the task, passing in the address of
    xHigherPriorityTaskWoken as the interrupt safe API function's
    pxHigherPriorityTaskWoken parameter. */
    xSemaphoreGiveFromISR( xBinarySemaphore, &xHigherPriorityTaskWoken );

    /* Pass the xHigherPriorityTaskWoken value into portYIELD_FROM_ISR().  If
    xHigherPriorityTaskWoken was set to pdTRUE inside xSemaphoreGiveFromISR()
    then calling portYIELD_FROM_ISR() will request a context switch.  If
    xHigherPriorityTaskWoken is still pdFALSE then calling
    portYIELD_FROM_ISR() will have no effect. Unlike most FreeRTOS ports, the
    Windows port requires the ISR to return a value - the return statement
    is inside the Windows version of portYIELD_FROM_ISR(). */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Listing 94. The ISR for the software interrupt used in Example 16

```
int main( void )
{
    /* Before a semaphore is used it must be explicitly created.  In this example
    a binary semaphore is created. */
    xBinarySemaphore = xSemaphoreCreateBinary();

    /* Check the semaphore was created successfully. */
    if( xBinarySemaphore != NULL )
    {
        /* Create the 'handler' task, which is the task to which interrupt
        processing is deferred.  This is the task that will be synchronized with
        the interrupt.  The handler task is created with a high priority to ensure
        it runs immediately after the interrupt exits.  In this case a priority of
        3 is chosen. */
        xTaskCreate( vHandlerTask, "Handler", 1000, NULL, 3, NULL );

        /* Create the task that will periodically generate a software interrupt.
        This is created with a priority below the handler task to ensure it will
        get preempted each time the handler task exits the Blocked state. */
        xTaskCreate( vPeriodicTask, "Periodic", 1000, NULL, 1, 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, ulExampleInterruptHandler );

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

    /* As normal, the following line should never be reached. */
    for( ;; );
}
```

Listing 95. The implementation of main() for Example 16

→ Listing 92 shows the implementation of periodic task.

→ Listing 93 shows the implementation of task to which interrupt processing is deferred — the task that is synchronised with software interrupt through the use of binary semaphore.

→ Listing 94 shows the ISR

→ The main() function create binary semaphore, create the tasks, install the interrupt handler, and start the scheduler. shown in listing 95

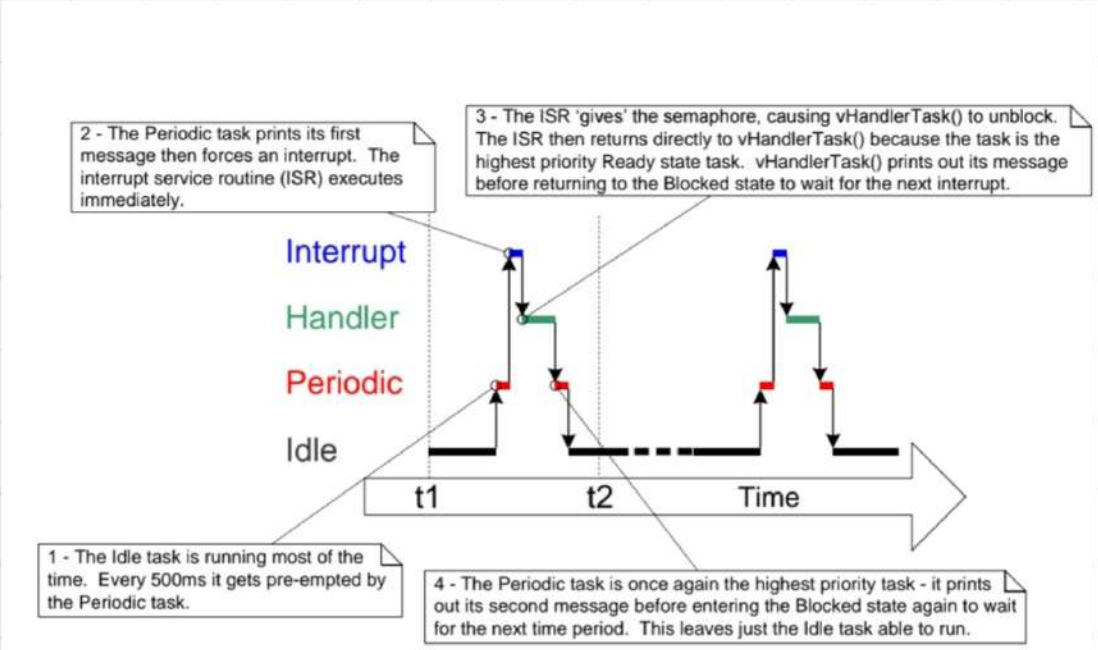


Figure 52. The sequence of execution when Example 16 is executed

⇒ what would happen if a second, and then a third, interrupt had occured before the task had completed it's processing of first interrupt.

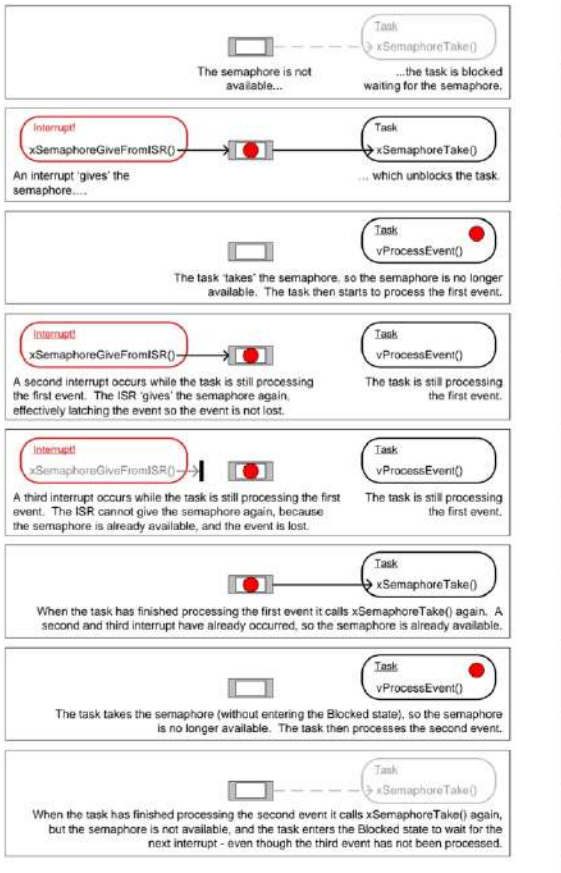


Figure 54 The scenario when two interrupts occur before the task has finished processing the first event

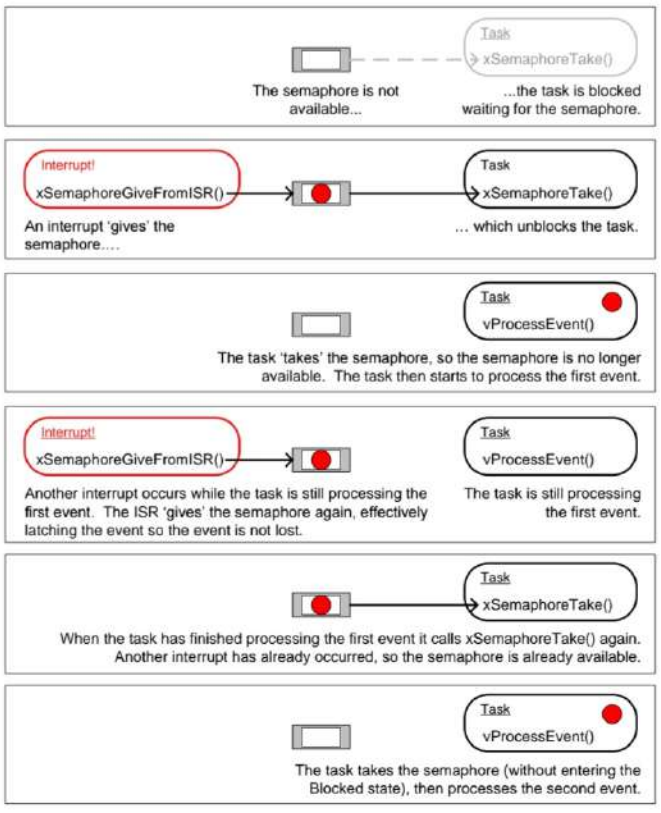


Figure 53. The scenario when one interrupt occurs before the task has finished processing the first event

Counting semaphores

Counting semaphore can be thought of as queue that have length of more than one.

① Counting events —

- An event handler will "give" a semaphore each time an event occur, semaphore's count value to be incremented on each 'give'.
- A task will 'take' a semaphore each time it process an event causing semaphore value to be decremented

② Resource management —

The count value indicate the number of resource available.

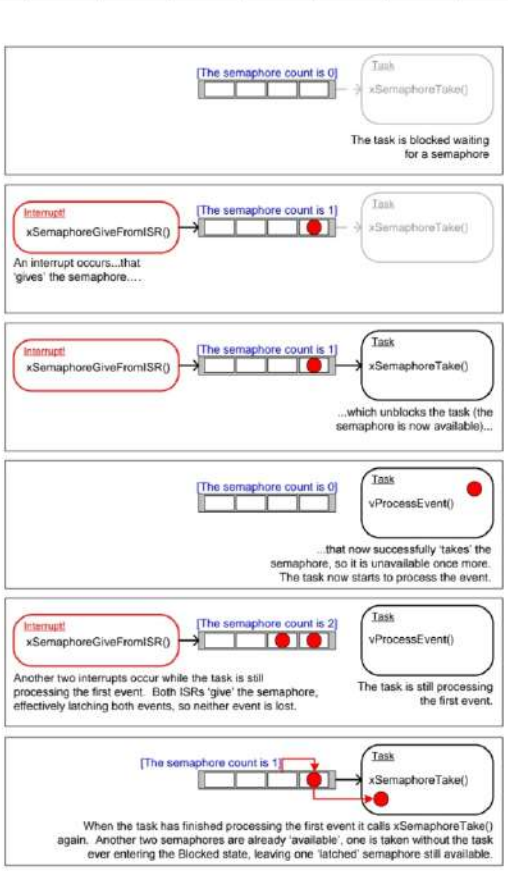


Figure 55. Using a counting semaphore to 'count' events

```
/* Before a semaphore is used it must be explicitly created.  In this example a
counting semaphore is created.  The semaphore is created to have a maximum count
value of 10, and an initial count value of 0. */
xCountingSemaphore = xSemaphoreCreateCounting( 10, 0 );
```

Listing 98. The call to xSemaphoreCreateCounting() used to create the counting semaphore in Example 17

```
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;

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

    /* Pass the xHigherPriorityTaskWoken value into portYIELD_FROM_ISR().  If
    xHigherPriorityTaskWoken was set to pdTRUE inside xSemaphoreGiveFromISR() then
    calling portYIELD_FROM_ISR() will request a context switch.  If
    xHigherPriorityTaskWoken is still pdFALSE then calling portYIELD_FROM_ISR() will
    have no effect. Unlike most FreeRTOS ports, the Windows port requires the ISR to
    return a value - the return statement is inside the Windows version of
    portYIELD_FROM_ISR(). */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Listing 99. The implementation of the interrupt service routine used by Example 17

Deffering work to the RTOS Daemon Task

→ It in also possible to use the xTimerPendFunctionCall FromISR() API function to defer processing to the RTOS daemon task — removing the need to create a seperate task for each interrupt. Deffering interrupt processing to the daemon task in called 'centralized deffered interrupt processing'.

→ The priority of the daemon task in set by the configTIMER_TASK_PRIDRITY compile time configurat constant within FreeRTOSConfig.h

Example for centralized deffered interrupt processing

→ Same functionality but without using semaphore and without creating a task specifially to perform the processing necessiated by the interrupt.

```
int main( void )
{
/* The task that generates the software interrupt is created at a priority below the
priority of the daemon task.  The priority of the daemon task is set by the
configTIMER_TASK_PRIORITY compile time configuration constant in FreeRTOSConfig.h. */
const UBaseType_t ulPeriodicTaskPriority = configTIMER_TASK_PRIORITY - 1;

    /* Create the task that will periodically generate a software interrupt. */
    xTaskCreate( vPeriodicTask, "Periodic", 1000, NULL, ulPeriodicTaskPriority, 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, ulExampleInterruptHandler );

    /* Start the scheduler so the created task starts executing. */
    vTaskStartScheduler();

    /* As normal, the following line should never be reached. */
    for( ;; );
}
```

Listing 104. The implementation of main() for Example 18

```
static uint32_t ulExampleInterruptHandler( void )
{
static uint32_t ulParameterValue = 0;
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 pointer to the interrupt's deferred handling function to the daemon task.
    The deferred handling function's pvParameter1 parameter is not used so just set to
    NULL.  The deferred handling function's ulParameter2 parameter is used to pass a
    number that is incremented by one each time this interrupt handler executes. */
    xTimerPendFunctionCallFromISR( vDeferredHandlingFunction, /* Function to execute. */
                                   NULL,                      /* Not used. */
                                   ulParameterValue,          /* Incrementing value. */
                                   &xHigherPriorityTaskWoken );
    ulParameterValue++;

    /* Pass the xHigherPriorityTaskWoken value into portYIELD_FROM_ISR().  If
    xHigherPriorityTaskWoken was set to pdTRUE inside xTimerPendFunctionCallFromISR() then
    calling portYIELD_FROM_ISR() will request a context switch.  If
    xHigherPriorityTaskWoken is still pdFALSE then calling portYIELD_FROM_ISR() will have
    no effect. Unlike most FreeRTOS ports, the Windows port requires the ISR to return a
    value - the return statement is inside the Windows version of portYIELD_FROM_ISR(). */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Listing 102. The software interrupt handler used in Example 18

```
static void vDeferredHandlingFunction( void *pvParameter1, uint32_t ulParameter2 )
{
    /* Process the event - in this case just print out a message and the value of
    ulParameter2. pvParameter1 is not used in this example. */
    vPrintStringAndNumber( "Handler function - Processing event ", ulParameter2 );
}
```

Listing 103. The function that performs the processing necessitated by the interrupt in Example 18.

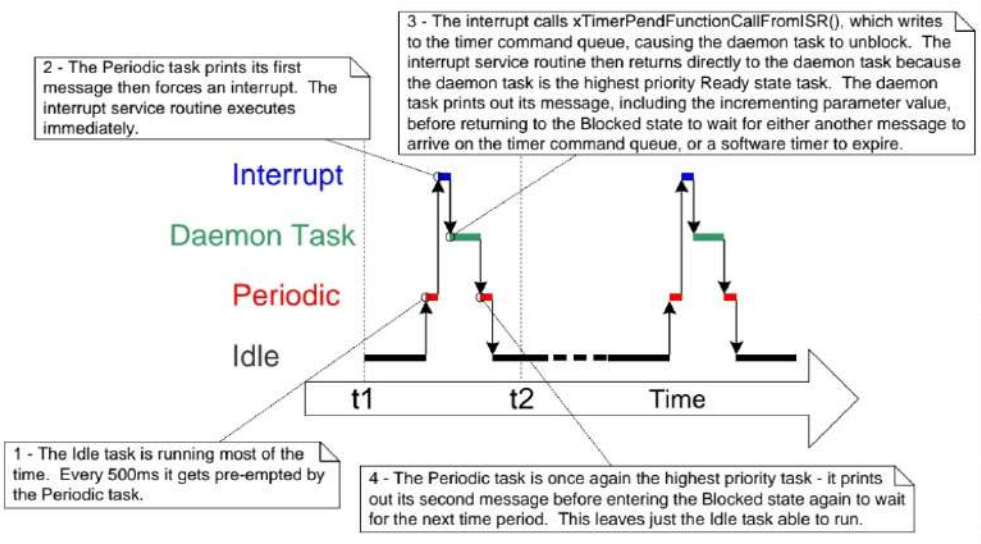


Figure 58 The sequence of execution when Example 18 is executed

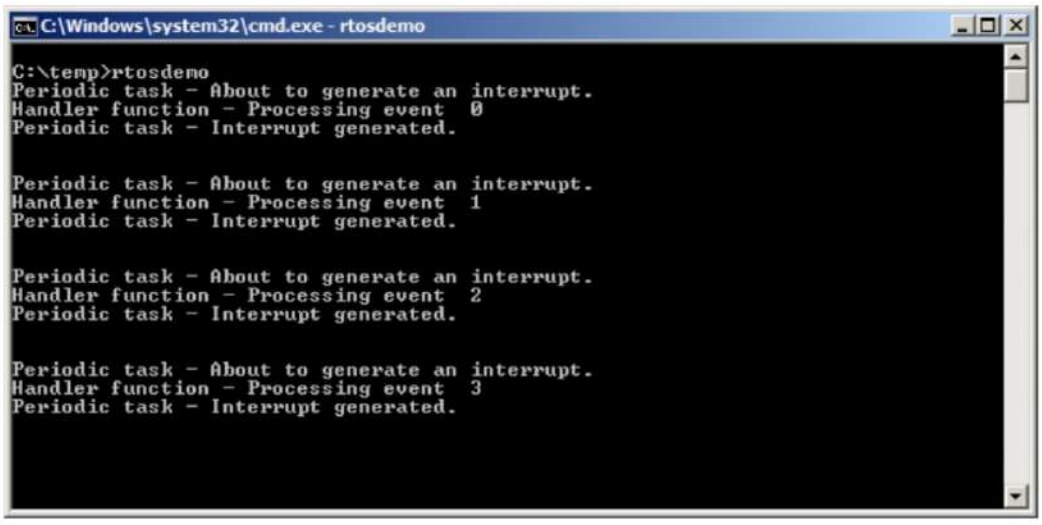


Figure 57. The output produced when Example 18 is executed

Using queues within an Interrupt service Routine