USING THE FreeRTOS REAL TIME KERNEL

A Practical Guide

Richard Barry

This page intentionally left blank

CONTENTS

Designed For Microcontrollers;

Designed For Microcontrollers;

Designed For Microcontrollers;

LIST OF FIGURES

LIST OF CODE LISTINGS

```
Designed For Microcontrollers;
```

Designed For Microcontrollers;

LIST OF TABLES

LIST OF NOTATION

FAQ   Frequently Asked Question

FIFO   First In First Out

HMI   Human Machine Interface

ISR   Interrupt Service Routine

LCD   Liquid Crystal Display

RTOS   Real Time Operating System

TCB   Task Control Block

UART   Universal Asynchronous Receiver / Transmitter

CHAPTER 1

TASK MANAGEMENT

1.1 CHAPTER INTRODUCTION AND SCOPE

[The appendixes also provide practical information specific to using the FreeRTOS source code.]

An Introduction to Multi Tasking in Small Embedded Systems

Different multi tasking systems have different objectives. Taking workstations and desktops as an example:

- In the 'old days' processors were expensive so multitasking was used as a means to allow lots of users access to a single processor. The scheduling algorithms used in these types of system were designed with the objective of allowing each user a 'fair share' of processing time.

- In more recent times processing power has become less expensive so each user can have exclusive access to one or more processors. The scheduling algorithms in these types of system are designed to allow users to run multiple applications simultaneously without the computer becoming unresponsive. For example a user may run a word processor, a spreadsheet, an email client and a WEB browser all at the same time and would expect each application to respond adequately to input at all time.

Input processing on a desktop computer can be classified as 'soft real time'. To ensure the best user experience the computer should respond to each input within a preferred time limit – but a response falling outside of this limit will not render the computer useless. For example, key presses must be visibly registered within a certain time of the key being pressed. Registering a key press outside of this time could result in the system seeming unresponsive, but not unusable.

Multi tasking in a real time embedded system is conceptually similar to multi tasking in a desktop system to the point that it describes multiple threads of execution using a single processor. However the objectives of real time embedded systems are likely to be quite different to that of desktops – especially when the embedded system is expected to provide 'hard real time' behavior.

Hard real time functions **must** complete within a given time limit – failure to do so will result in absolute failure of the system. The airbag triggering mechanism in a car is an example of a hard real time function. The airbag must deploy within a given time limit of an impact. A response falling outside of this time limit can result in the driver sustaining injuries that would otherwise have been avoided.

Most embedded systems implement a mix of both hard and soft real time requirements.

A Note About Terminology

In FreeRTOS each thread of execution is called a 'task'. There is no absolute agreed consensus on terminology within the embedded community, but I prefer 'task' to 'thread' as thread can have a more specific meaning depending on your previous experience.

Scope

This chapter aims to give readers a good understanding of:

- How FreeRTOS allocates processing time to each task within an application.

- How FreeRTOS chooses which task should execute at any given time.

- How the relative priority of each task affects system behavior.

- The states that a task can exist in.

In addition readers will hopefully gain a good understanding of:

- How to implement tasks.

- How to create one or more instances of a task.

- How to use the task parameter.

- How to change the priority of a task that has already been created.

- How to delete a task.

- How to implement periodic processing.

- When the idle task will execute and how it can be used.

The concepts presented in this chapter are fundamental to understanding how to use FreeRTOS and how FreeRTOS applications behave – this is therefore the most detailed chapter in the book.

1.2   TASK FUNCTIONS

Tasks are implemented as C functions. <mark>The only thing special about them is their prototype, which must return void and take a void pointer parameter</mark>. The prototype is demonstrated by Listing 1.

```
void ATaskFunction( void *pvParameters );
```

Listing 1 The task function prototype

Each task is a small program in its own right. It has an entry point, will normally run forever within an infinite loop, and will not exit. The structure of a typical task is shown in Listing 2.

FreeRTOS tasks must **not** be allowed to return from their implementing function in any way – they must not contain a 'return' statement and must not be allowed to execute past the end of the function. If a task is no longer required it should instead be explicitly deleted. This is also demonstrated in Listing 2.

A single task function definition can be used to create any number of tasks – each created task being a separate execution instance with its own stack and its own copy of any automatic (stack) variables defined within the task itself.

```
void ATaskFunction( void *pvParameters )
{
/* Variables can be declared just as per a normal function.  Each instance
of a task created using this function will have its own copy of the
iVariableExample 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. */
int iVariableExample = 0;

    /* A task will normally be implemented as in 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 this function.
    The NULL parameter passed to the vTaskDelete() function indicates that
    the task to be deleted is the calling (this) task. */
    vTaskDelete( NULL );
}
```

Listing 2 The structure of a typical task function

1.3   TOP LEVEL TASK STATES

An application can consist of many tasks. If the microcontroller running the application only contains a single core then only one task can actually be executing at any given time. This implies that a task can exist in one of two states, Running and Not Running. We will consider this simplistic model first - but keep in mind that this is an over simplification as later we will see the Not Running state actually contains a number of sub-states.

When a task is in the Running state the processor is actually executing its code. When a task is in the Not Running state the task is dormant, its status having been saved ready for it to resume execution the next time the scheduler decides it should enter the Running state. When a task resumes execution it does so from exactly the instruction it was about to execute before it last left the Running state.
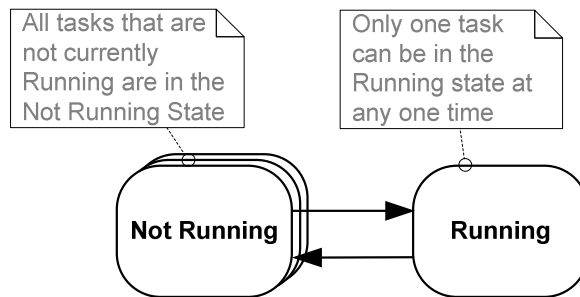
Figure 1 Top level task states and transitions.

A task transitioned from the Not Running to the Running state is said to have been "switched in" or "swapped in". Conversely, a task transitioned from the Running state to the Not Running state is said to have been "switched out" or "swapped out". The FreeRTOS scheduler is the only entity that can switch a task in and out.

1.4   CREATING TASKS

xTaskCreate() API Function

Tasks are created using the FreeRTOS xTaskCreate() API function. This is probably the most complex of all the API functions so it is unfortunate that it is the first encountered, but tasks must be mastered first as they are the most fundamental component of a multitasking system. All the examples that accompany this book make use of the xTaskCreate() function so there are plenty of examples to reference.

APPENDIX 5: describes the data types and naming conventions used.

```
portBASE_TYPE xTaskCreate( pdTASK_CODE pvTaskCode,
                           const signed portCHAR * const pcName,
                           unsigned portSHORT usStackDepth,
                           void *pvParameters,
                           unsigned portBASE_TYPE uxPriority,
                           xTaskHandle *pxCreatedTask
                         );
```

Listing 3 The xTaskCreate() API function prototype

Table 1 xTaskCreate() parameters and return value

Parameter Name/Returned Value	Description
pvTaskCode	Tasks are just C functions that never exit, and as such are normally implemented as an infinite loop. The pvTaskCode parameter is simply a pointer to the function (in effect just the function name) that implements the task.
pcName	A descriptive name for the task. This is not used by FreeRTOS in any way. It is included purely as a debugging aid. Identifying a task by a human readable name is much simpler than attempting to do the same from its handle. The application defined constant configMAX_TASK_NAME_LEN defines the maximum length a task name can take – including the NULL terminator. Supplying a string longer than this maximum will simply result in the string being silently truncated.

Table 1 xTaskCreate() parameters and return value

Parameter Name/Returned Value	Description
usStackDepth	Each task has its own unique state that is allocated by the kernel to the task when the task is created. The usStackDepth value tells the kernel how big to make the stack. The value specifies the number of words the stack can hold, not the number of bytes. For example, if the stack is 32 bits wide and usStackDepth is passed in as 100, then 400 bytes of stack space will be allocated (100 * 4bytes). The stack depth multiplied by the stack width must not exceed the maximum value that can be contained in a variable of type size_t. The size of the stack used by the idle task is defined by the application defined constant configMINIMAL_STACK_SIZE . The value assigned to this constant in the FreeRTOS demo application for the microcontroller architecture being used is the minimum recommended for any task. If your task uses a lot of stack space then you will need to assign a larger value. There is no easy way of determining the stack space required by a task. It is possible to calculate, but most users will simply assign what they think is a reasonable value, then use the features provided by FreeRTOS to ensure both that the space allocated is indeed adequate, and that RAM is not being unnecessarily wasted. CHAPTER 6 contains information on how to query the stack space being used by a task.
pvParameters	Task functions accept a parameter of type pointer to void (void*). The value assigned to pvParameters will be the value passed into the task. Some examples within this document demonstrate how the parameter can be used.
uxPriority	Defines the priority at which the task will execute. Priorities can be assigned from 0, which is the lowest priority, to (configMAX_PRIORITIES – 1), which is the highest priority. configMAX_PRIORITIES is a user defined constant. There is no upper limit on the number of priorities that can be available (other than the limit of the data types used and the RAM available in your microcontroller), but you should use the lowest number of priorities actually required in order to avoid wasting RAM. Passing a uxPriority value above (configMAX_PRIORITIES – 1) will result in the actual priority assigned to the task being silently capped to the maximum legitimate value.

Table 1 xTaskCreate() parameters and return value

Parameter Name/Returned Value	Description
pxCreatedTask	pxCreatedTask can be used to pass out a handle to the task being created. This handle can then be used to reference the task within API calls that, for example, change the task priority or delete the task. If your application has no use for the task handle then pxCreatedTask can be set to NULL.

Returned value	There are two possible return values:
	
	    1. pdTRUE
	
	This indicates that the task was created successfully.
	
	    2. errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY
	
	This indicates that the task could not be created because there was insufficient heap memory available for FreeRTOS to allocate enough RAM to hold the task data structures and stack.
	
	CHAPTER 5 provides more information on memory management.

Example 1. Creating Tasks

APPENDIX 1: contains information on the tools required to build the example projects.

This example demonstrates the steps necessary to create two simple tasks then start the tasks executing. The tasks just periodically print out a string, using a crude null loop to create the period delay. Both tasks are created at the same priority and are identical other than the string they print out – see Listing 4 and Listing 5 for their respective implementations.

```
void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\r\n";
volatile unsigned long ul;

    /* 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 4 Implementation of the first task used in Example 1

```
void vTask2( void *pvParameters )
{
const char *pcTaskName = "Task 2 is running\r\n";
volatile unsigned long ul;

    /* 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 5 Implementation of the second task used in Example 1

The main() function simply creates the tasks before starting the scheduler – see Listing 6 for its implementation.

```
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 - most small microcontrollers will use much
                             less stack than this. */
                    NULL,    /* We are not using the task parameter. */
                    1,       /* This task will run at priority 1. */
                    NULL );  /* We are not going to 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 5 provides more information on memory management. */
    for( ;; );
}
```

Listing 6 Starting the Example 1 tasks

Executing the example produces the output shown in Figure 2.
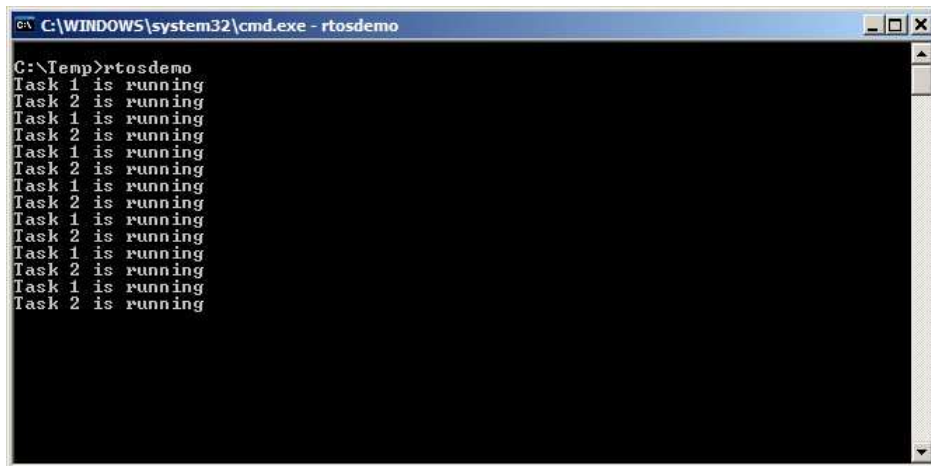
Figure 2 The output produced when Example 1 is executed

Figure 2 shows the two tasks appearing to execute simultaneously, but both tasks are executing on the same processor so this cannot actually be the case. In reality both tasks are rapidly entering and exiting the Running state. Both tasks are running at the same priority so share time on the single processor. Their actual execution pattern is shown in Figure 3.

The arrow along the bottom of Figure 3 shows the passing of time from time t1 onwards. The colored lines show which task is executing at each point in time – for example Task1 is executing between time t1 and time t2.

Only one task can exist in the Running state at any one time so as one task enters the Running state (the task is switched in) the other enters the Not Running state (the task is switched out).
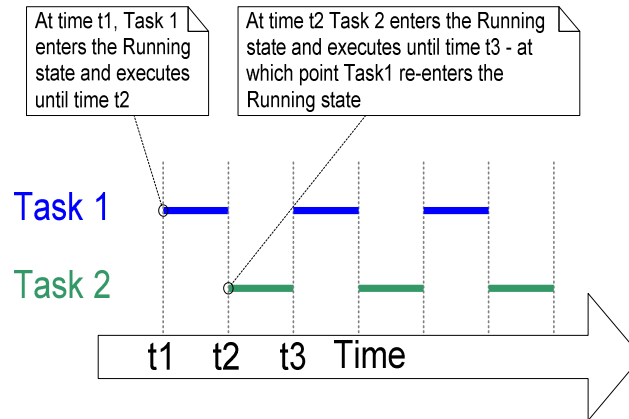
Figure 3 The actual execution pattern of the two Example 1 tasks

Example 1 created both tasks from within main() prior to starting the scheduler. It is also possible to create a task from within another task. We could have created Task1 from main(), and then created Task2 from within Task1. Were we to do this our Task1 function would change as shown by Listing 7. Task2 would not get created until after the scheduler had been started but the output produced by the example would be the same.

```
void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\r\n";
volatile unsigned long ul;

    /* If this task code is executing then the scheduler must already have
    been started.  Create the other task before we enter 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 7 Creating a task from within another task – after the scheduler has started

Example 2. Using the Task Parameter

The two tasks created in Example 1 were nearly identical, the only difference between them was the text string they printed out. This duplication can be removed by instead creating two instances of a single task implementation. The task parameter can then be used to pass into each task the string that that instance should print out.

Listing 8 contains the code of the single task function (vTaskFunction) used by Example 2. This single function replaces the two task functions (vTask1 and vTask2) used in Example 1. Note how the task parameter is cast to a char * to obtain the string the task should print out.

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;
volatile unsigned long ul;

    /* 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 8 The single task function used to create two tasks in Example 2

Even though there is now only one task implementation (vTaskFunction), more than one instance of the defined task can be created. Each created instance will execute independently under the control of the FreeRTOS scheduler.

The pvParameters parameter to the xTaskCreate() function is used to pass in the text string as shown in Listing 9.

```c
/* 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\t\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 - most 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 );                 /* We are not using the task handle. */

    /* 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 our 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( ;; );
}
```

Listing 9 The main() function for Example 2.

The output from Example 2 is exactly as per that shown for example 1 in Figure 2.

1.5   TASK PRIORITIES

The uxPriority parameter of the xTaskCreate() API function assigns 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 maximum number of priorities available is set by the application defined configMAX_PRIORITIES compile time configuration constant within FreeRTOSConfig.h.   FreeRTOS itself does not limit the maximum value this constant can take, but the higher the configMAX_PRIORITIES value the more RAM the kernel will consume, so it is always advisable to keep the value set at the minimum necessary.

FreeRTOS does not impose any restrictions on how priorities can be assigned to tasks.   Any number of tasks can share the same priority – ensuring maximum design flexibility.   You can assign a unique priority to every task if this is desirable (as required by some schedule-ability algorithms) but this restriction is not enforced in any way.

Low numeric priority values denote low priority tasks, with priority 0 being the lowest priority possible. The range of available priorities is therefore 0 to (configMAX_PRIORITIES – 1).

The scheduler will always ensure that the highest priority task that is able to run is the task selected to enter the Running state.   Where more than one task of the same priority is able to run the scheduler will transition each task into and out of the Running state in turn.   This is the behavior observed in the examples so far, where both test tasks were created at the same priority and both were always able to run.   Each such task executes for a "time slice", it enters the Running state at the start of the time slice and exits the Running state at the end of the time slice.   In Figure 3 the time between t1 and t2 equals a single time slice.

To be able to select the next task to run the scheduler itself has to execute at the end of each time slice.   A periodic interrupt called the tick interrupt is used for this purpose.   The length of the time slice is effectively set by the tick interrupt frequency which is configured by the configTICK_RATE_HZ compile time configuration constant in FreeRTOSConfig.h.   For example, if configTICK_RATE_HZ is set to 100 (Hz) then the time slice will be 10ms.   Figure 3 can be expanded to show the execution of the scheduler itself in the sequence of execution.   This is shown in Figure 4.

Note that FreeRTOS API calls always specify time in tick interrupts (commonly referred to as just 'ticks').   The portTICK_RATE_MS constant is provided to allow time delays to be converted from the number of tick interrupts into milliseconds.   The resolution available depends on the tick frequency.

The 'tick count' value is the total number of tick interrupts that have occurred since the scheduler was started; assuming the tick count has not overflowed.   User applications do not need to consider overflows when specifying delay periods as time consistency is managed internally by the kernel.
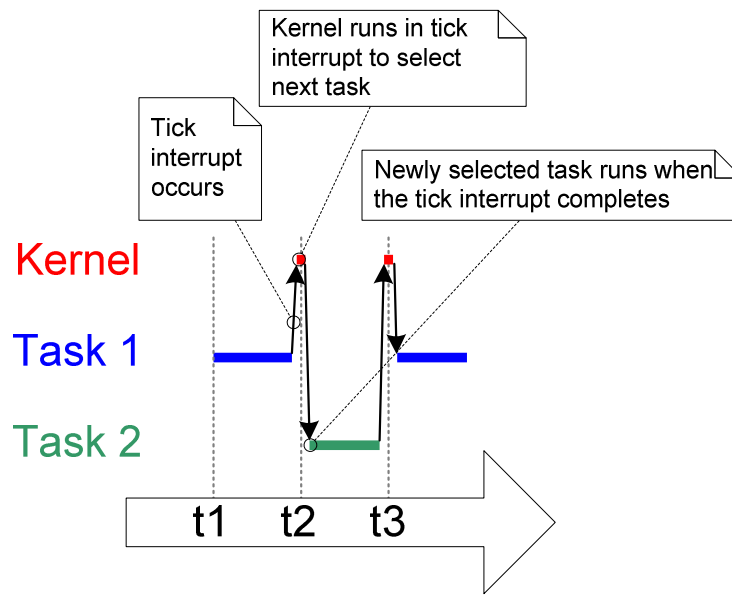
Figure 4 The execution sequence expanded to show the tick interrupt executing.

In Figure 4 the red line shows when the kernel itself is running. The black arrows show the sequence of execution from task to interrupt, then from interrupt back to a different task.

Example 3.   Experimenting with priorities

The scheduler will always ensure that the highest priority task that is able to run is the task selected to enter the Running state. In our examples so far two tasks have been created at the same priority, so both entered and exited the Running state in turn. This example looks at what happens when we change the priority of one of the two tasks created in Example 2. This time the first task will be created at priority 1, and the second at priority 2. The code to create the tasks is shown in Listing 10. The single function that implements both tasks has not changed, it still just periodically prints out a string using a null loop to create a delay.

```
/* 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\t\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. */
    xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 2, NULL );

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

    return 0;
}
```

Listing 10 Creating two tasks at different priorities

The output produced by Example 3 is shown in Figure 5.

The scheduler will always select the highest priority task that is able to run. Task 2 has a higher priority than Task 1 and is always able to run; therefore Task 2 is the only task to ever enter the Running state. As Task 1 never enters the Running state it never prints out its string. Task 1 is said to be 'starved' of processing time by Task 2.
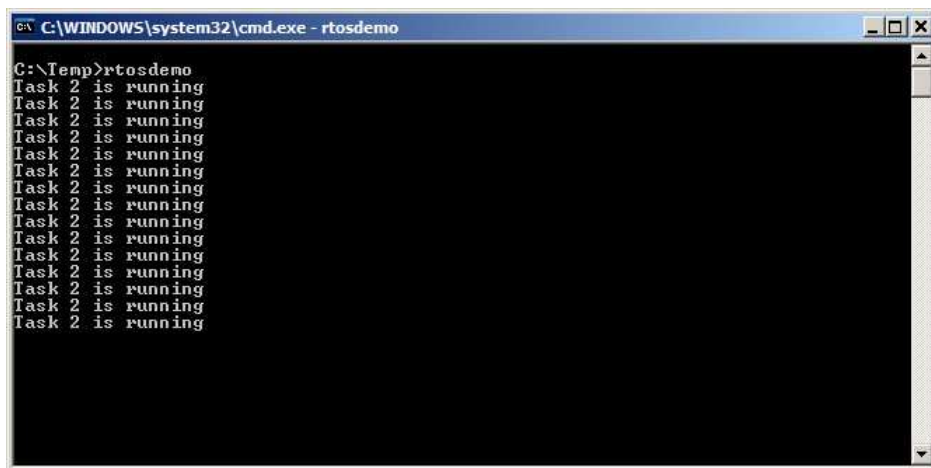
Figure 5 Running both test tasks at different priorities

Task 2 is always able to run because it never has to wait for anything – it is either spinning around a null loop or printing to the terminal.

Figure 6 shows the execution sequence for Example 3.
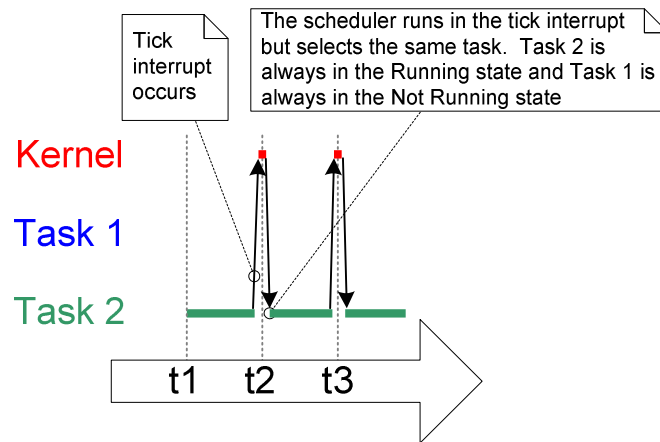
Figure 6 The execution pattern when one task has a higher priority than the other

1.6   EXPANDING THE 'NOT RUNNING' STATE

So far the in the examples presented each created tasks has always had processing it wants to perform and never needed to wait for anything – as they never have to wait for anything they are always able to enter the Running state. This type of 'continuous processing' task has limited usefulness because they can only be created at the very lowest priority. If they run at any other priority they will prevent tasks of lower priority ever running at all.

To make our tasks actually useful we need a way of allowing them to be event driven. An event driven task only has work (processing) to perform after the occurrence of the event that triggers it, and is not *able* to enter the Running state before the event has occurred. The scheduler always selects the highest priority task that is *able* to run. High priority tasks not being able to run means the scheduler cannot select them and must instead select a lower priority task that is able to run. Using event driven tasks therefore means that tasks can be created at lots of different priorities without the highest priority tasks starving all the lower priority tasks of processing time.

The Blocked State

A task that is waiting for an event is said to be in the 'Blocked' state, which is a sub-state of the Not Running state.

Tasks can enter the Blocked state to wait for two different types of event:

1. Temporal (time related) events – the event being either a delay period expiring or an absolute time being reached. For example a task may enter the Blocked state to wait for 10 milliseconds to pass.

2. Synchronization events – where the events originate from another task or interrupt. For example, 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 queues, binary semaphores, counting semaphores, recursive semaphores and mutexes can all be used to create synchronization events. CHAPTER 2 and CHAPTER 3 cover these in more detail.

It is possible for a task to block on a synchronization event with a timeout, effectively blocking on both types of event simultaneously. For example, a task may choose to wait for a maximum of 10 milliseconds for data to arrive on a queue. The task will leave the Blocked state if either data arrives within 10 milliseconds, or 10 milliseconds pass with no data arriving.

The Suspended State

'Suspended' is also a sub-state of Not Running. 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, and the only way out through a call to the vTaskResume() or xTaskResumeFromISR() API functions. Most applications don't use the Suspended state.

The Ready State

Tasks that are in the Not Running but are not Blocked or Suspended are said to be in the Ready state. They are able to run, and therefore 'ready' to run, but not currently in the Running state.

Completing the State Transition Diagram

Figure 7 expands on the previous over simplified state diagram to include all the Not Running sub states described in this section. The tasks created in the examples so far have not used either the Blocked or Suspended states so have only transitioned between the Ready and the Running state – highlighted by the bold lines in Figure 7.
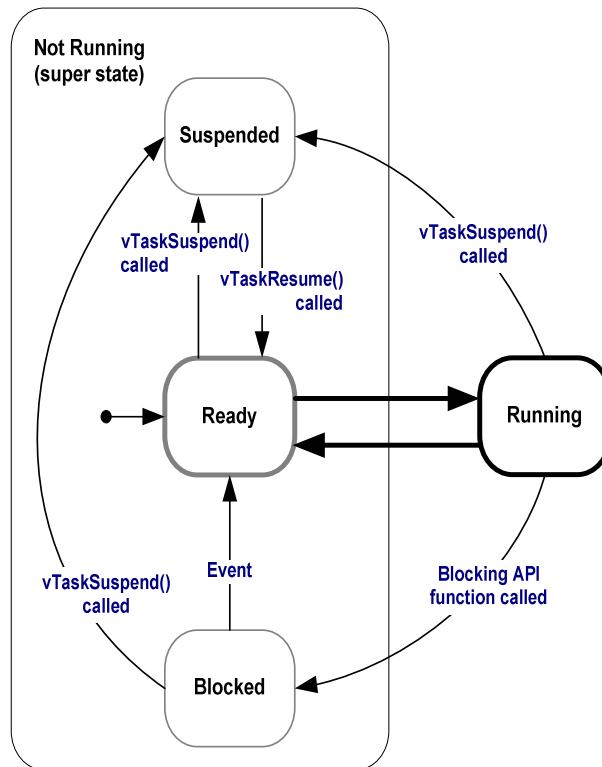
Figure 7 Full task state machine

Example 4. Using the Blocked state to create a delay

All the tasks created in the examples presented so far have been 'periodic' – they have delayed for a period, printed out their string, before delaying once more, and so on. The delay has been generated very crudely using a null loop – the task effectively polled an incrementing loop counter until it reached a fixed value. Example 3 clearly demonstrated the disadvantage of this method. While executing the null loop the task remained in the Ready state, 'starving' the other task of any processing time.

There are several other disadvantages to any form of polling, not least of which is its inefficiency. While polling the task does not really have any work to do, but it still uses maximum processing time and so wastes processor cycles. Example 4 corrects this behavior by replacing the polling null loop with a call to the vTaskDelay() API function, the prototype for which is shown in Listing 11. The new task definition is shown in Listing 12.

vTaskDelay() places the calling task into the Blocked state for a fixed number of tick interrupts. While in the Blocked state the task will not use any processing time at all, so processing time is only consumed when there is genuinely work to be done.

```
void vTaskDelay( portTickType xTicksToDelay );
```

Listing 11 The vTaskDelay() API function prototype

Table 2 vTaskDelay() parameters

Parameter Name	Description
xTicksToDelay	The number of tick interrupts that the calling task should remain in the Blocked state before being transitioned back into the Ready state.

For example, if a task called vTaskDelay(100) while the tick count was 10,000, then it would immediately enter the Blocked state and remain there until the tick count reached 10,100.

The constant portTICK_RATE_MS can be used to convert milliseconds into ticks. |

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

    /* 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 delay period is specified in 'ticks', but the constant
        portTICK_RATE_MS can be used to convert this to a more user friendly value
        in milliseconds.  In this case a period of 250 milliseconds is being
        specified. */
        vTaskDelay( 250 / portTICK_RATE_MS );
    }
}
```

Listing 12 The source code for the example task after the null loop delay has been replaced by a call to vTaskDelay()

Even though the two tasks are still being created at different priorities both will now run. The output of Example 4 shown in Figure 8 confirms the expected behavior.
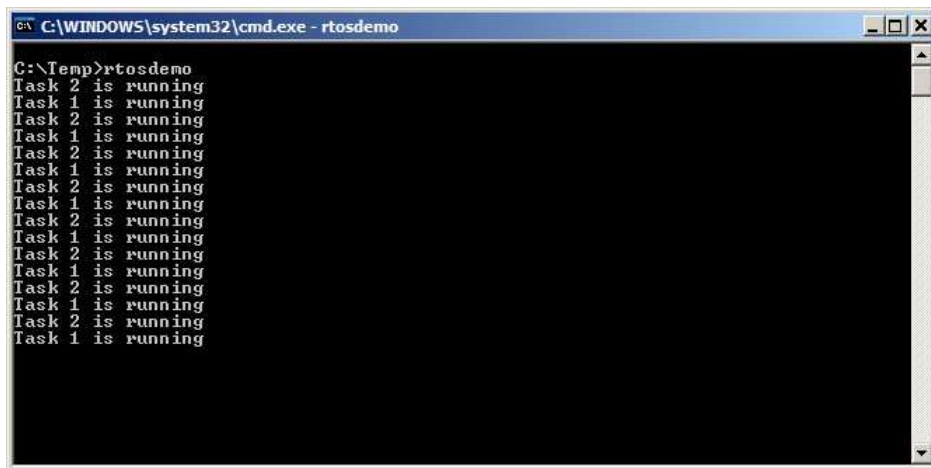
Figure 8 The output produced when Example 4 is executed

The execution sequence shown in Figure 9 explains why both tasks run even though they are created at different priorities. The execution of the kernel itself is omitted for simplicity.

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 the Ready state). Section 1.7 describes the Idle task in more detail.

2 - Task 1 prints out its string, then it too enters the Blocked state by calling vTaskDelay().

4 - When the delay expires the scheduler moves the tasks back into the ready state, where both execute again before once again calling vTaskDelay() causing them to re-enter the Blocked state. Task 2 executes first as it has the higher priority.

1 - Task 2 has the highest priority so runs first. It prints out its string then calls vTaskDelay() - and in so doing enters the Blocked state, permitting the lower priority Task 1 to execute.

3 - At this point both application tasks are in the Blocked state - so the Idle task runs.

Figure 9 The execution sequence when the tasks use vTaskDelay() in place of the NULL loop

Only the implementation of our two tasks has changed, not their functionality. Comparing Figure 9 with Figure 4 clearly demonstrates that this functionality is being achieved in a much more efficient manner.

Figure 4 shows the execution pattern when the tasks were using a null loop to create a delay – so were always able to run and used a lot of processor time as a result. Figure 9 shows the execution pattern when the tasks enter the Blocked state for the entirety of their delay period, so only utilize processor time when they actually have work that needs to be performed (in this case simply a message being printed out).

In the Figure 9 scenario each time the tasks leave the Blocked state they only execute for a fraction of a tick period before re-entering the blocked state. Most of the time there are no application tasks that are able to run (there are no application tasks in the Ready state) and therefore no application tasks that can be selected to enter the Running state. While this is the case the idle task will run. The amount of processing time the idle task gets is a measure of the spare processing capacity in the system.

The bold lines in Figure 10 show the transitions performed by the tasks in Example 4, with each now transitioning through the Blocked state before being returned to the Ready state.
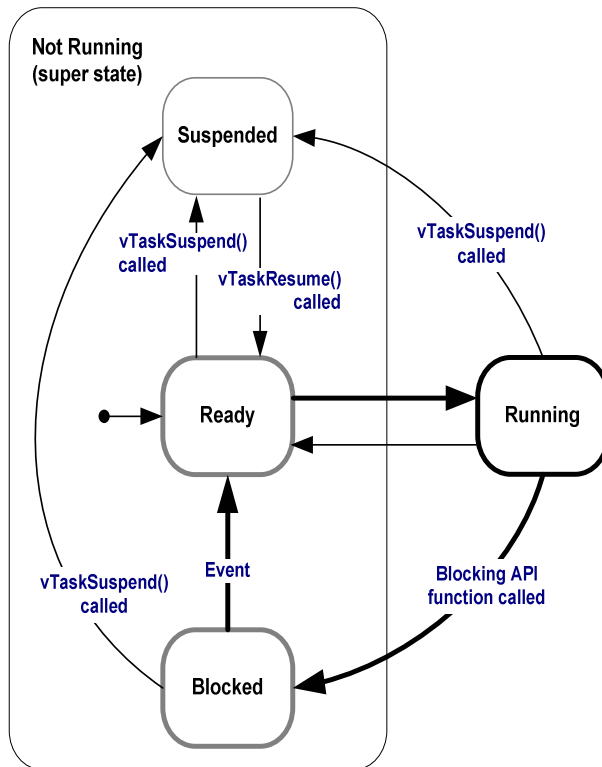
Figure 10 Bold lines indicate the state transitions performed by the tasks in Example 4

vTaskDelayUntil() API function

vTaskDelayUntil() is similar to vTaskDelay(). As just demonstrated, the vTaskDelay() parameter specifies the number of tick interrupts that should occur between a task calling vTaskDelay() and the same task once again transitioning out of the Blocked state. The amount of time the task remains in the blocked state is specified by the vTaskDelay() parameter but the actual time at which the task leaves the blocked state is relative to the time at which vTaskDelay() was called. The parameters to vTaskDelayUntil() instead specify the exact tick count value at which the calling task should be moved from the Blocked state into the Ready state. vTaskDelayUntil() is the API function that should be used when a fixed execution period is required (where you want your task to execute periodically with a fixed frequency) as the time at which the calling task is unblocked is absolute, rather than relative to when the function was called (as is the case with vTaskDelay()).

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

Listing 13 vTaskDelayUntil() API function prototype

Table 3 vTaskDelayUntil() parameters

Parameter Name	Description
pxPreviousWakeTime	This parameter is named on the assumption that vTaskDelayUntil() is being used to implement a task that executes periodically and with a fixed frequency. In this case pxPreviousWakeTime holds the time at which the task last left the Blocked state (was 'woken' up). This time is used as a reference point to calculate the time at which the task should next leave the Blocked state. The variable pointed to by pxPreviousWakeTime is updated automatically within the vTaskDelayUntil() function and would not normally be modified by the application code other than when the variable is first initialized. Listing 14 demonstrates how the initialization is performed.
xTimeIncrement	This parameter is also named on the assumption that vTaskDelayUntil() is being used to implement a task that executes periodically and with a fixed frequency – the frequency being set by the xTimeIncrement value. xTimeIncrement is specified in 'ticks'. The constant portTICK_RATE_MS can be used to convert milliseconds into ticks.

Example 5.   Converting the example tasks to use vTaskDelayUntil()

The two tasks created in Example 4 are periodic tasks, but using vTaskDelay() will not guarantee that the frequency at which they run will be fixed as the time at which the tasks leave the Blocked state is relative to when they call vTaskDelay(). Converting the tasks to use vTaskDelayUntil() in place of vTaskDelay() will solve this potential problem.

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;
portTickType 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 updated automatically internally 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 exactly every 250 milliseconds.  As per
        the vTaskDelay() function, time is measured in ticks, and the
        portTICK_RATE_MS constant is used to convert milliseconds into ticks.
        xLastWakeTime is automatically updated within vTaskDelayUntil() so is not
        explicitly updated by the task. */
        vTaskDelayUntil( &xLastWakeTime, ( 250 / portTICK_RATE_MS ) );
    }
}
```

Listing 14 The implementation of the example task using vTaskDelayUntil()

The output produced by Example 5 if exactly as per that shown in Figure 8 for Example 4.

Example 6.   Combining blocking and non-blocking tasks

Previous examples have examined the behavior of both polling and blocking tasks in isolation. This example re-enforces the stated expected system behavior by demonstrating an execution sequence when the two schemes are combined, as follows:

- Two tasks are created at priority 1. These do nothing other than continuously print out a string.

  These tasks never make any API function calls that could cause them to enter the Blocked state so are always in either the Ready or the Running state. Tasks of this nature are called 'continuous processing' tasks as they always have work to do, albeit rather trivial work in this case. The source for the continuous processing tasks is shown in Listing 15.

- A third task is then created at priority 2, so above the priority of the other two tasks. The third task also just prints out a string, but this time periodically so uses the vTaskDelayUntil() API call to place itself into the Blocked state between each print iteration.

  The source for the periodic task is shown in Listing 16.

```
void vContinuousProcessingTask( void *pvParameters )
{
char *pcTaskName;

    /* 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.  This task just does this repeatedly
        without ever blocking or delaying. */
        vPrintString( pcTaskName );
    }
}
```

Listing 15 The continuous processing task used in Example 6.

```
void vPeriodicTask( void *pvParameters )
{
portTickType xLastWakeTime;

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

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Periodic task is running\r\n" );

        /* The task should execute every 10 milliseconds exactly. */
        vTaskDelayUntil( &xLastWakeTime, ( 10 / portTICK_RATE_MS ) );
    }
}
```

Listing 16 The periodic task used in Example 6.

Figure 11 shows the output produced by Example 6, with an explanation of the observed behavior given by the execution sequence shown in Figure 12.

Figure 11 The output produced when Example 6 is executed[1].

4 - At time t5 the tick interrupt finds that the Periodic task block period has expired so moved the Periodic task into the Ready state. The Pericod task is the highest priority task so immediately then enters the Running state where it prints out its string exactly once before calling vTaskDelayUntil() to return to the Blocked state.

1 - Continuous task 1 runs for a complete tick period (time slice between times t1 and t2) - during which time it could print out its string many times.

5 - The Periodic task entering the Blocked state means the scheduler has again to choose a task to enter the Running state - in this case Continuous 1 is chosen and it runs up to the next tick interrupt - during which time it could print out its string many times.

The Idle task never enters the Running state as there are always higher priority task that are able to do so.

2 - The tick interrupt occurs during which the scheduler selects a new task to run. As both Continuous tasks have the same priority and both are always able to run the scheduler shares processing time between the two - so Continuous 2 enters the Running state where it remains for the entire tick period - during which time it could print out its string many times.

3 - At time t3 the tick interrupt runs again, cause a switch back to Continuous 1, and so it goes on.

Figure 12 The execution pattern of Example 6

---

[1] This output was produced using the DOSBox simulator to slow the execution down to a level where the entire behaviour could be observed in a single screen shot.

1.7 THE IDLE TASK AND THE IDLE TASK HOOK

The tasks created in Example 4 spend most of their time in the Blocked state. While in this state they are not able to run and cannot be selected by the scheduler.

The processor always needs something to execute – there must always be at least one task that can enter the Running state. To ensure this is the case an Idle task is automatically created by the scheduler when vTaskStartScheduler() is called. The idle task does very little more than sit in a loop – so like the tasks in the original examples it is always able to run.

The idle task has the lowest possible priority (priority 0) to ensure it never prevents a higher priority application task from entering the Running state – although there is nothing to prevent application designers creating task at, and therefore sharing, the idle task priority if desired.

Running at the lowest priority ensures the Idle task will be immediately transitioned out of the Running state as soon as a higher priority task enters the Ready state. This can be seen at time *tn* in Figure 9, where the Idle task is immediately swapped out to allow Task 2 to execute at the instant Task 2 leaves the Blocked state. Task 2 is said to have *pre-empted* the idle task. Pre-emption occurs automatically, and without the knowledge of the task being pre-empted.

Idle Task Hook Functions

It is possible to add application specific functionality directly into the idle task through the use of an idle hook (or call-back) function – a function that is automatically called by the idle task once per iteration of the idle task loop.

Common uses for the Idle task hook include:

- Executing low priority, background or continuous processing.

- Measuring the amount of spare processing capacity (the idle task will only run when all the other tasks have no work to perform, so measuring the amount of processing time allocated to the idle task provides a clear indication of how much processing time is spare).

- Placing the processor into a low power mode – providing an automatic method of saving power whenever there is no application processing to be performed.

Limitations on the Implementation of Idle Task Hook Functions

Idle task hook functions must adhere to the following rules:

1. They must never attempt to block or suspend. The Idle task will only execute when no other tasks are able to do so (unless application tasks are sharing the idle priority). Blocking the idle task in any way could therefore cause a scenario where no tasks are available to enter the Running state!

2. If the application makes use of the vTaskDelete() API function then the Idle task hook must always return to its caller within a reasonable time period. This is because the Idle task is responsible for cleaning up kernel resources after a task has been deleted. If the idle task remains permanently in the Idle hook function then this clean up cannot occur.

Idle task hook functions must have the name and prototype shown by Listing 17.

```
void vApplicationIdleHook( void );
```

Listing 17 The idle task hook function name and prototype.

Example 7. Defining an Idle Task Hook Function

The use of blocking vTaskDelay() API calls in Example 4 created a lot of idle time – time when the Idle task was executing because both application tasks were in the Blocked state. This example makes use of this idle time through the addition of an Idle hook function, the source for which is shown in Listing 18.

```
/* Declare a variable that will be incremented by the hook function. */
unsigned long ulIdleCycleCount = 0UL;

/* Idle hook functions MUST be called vApplicationIdleHook(), take no parameters,
and return void. */
void vApplicationIdleHook( void )
{
    /* This hook function does nothing but increment a counter. */
    ulIdleCycleCount++;
}
```

Listing 18 A very simple Idle hook function

configUSE_IDLE_HOOK must be set to 1 within FreeRTOSConfig.h for the idle hook function to get called.

The function that implements the created tasks is modified slightly to print out the ulIdleCycleCount value, as shown in Listing 19.

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

    /* 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 AND the number of times ulIdleCycleCount
        has been incremented. */
        vPrintStringAndNumber( pcTaskName, ulIdleCycleCount );

        /* Delay for a period for 250 milliseconds. */
        vTaskDelay( 250 / portTICK_RATE_MS );
    }
}
```

Listing 19 The source code for the example task now prints out the ulIdleCycleCount value

The output produced by Example 7 is shown in Figure 13 and shows that (on my computer) the idle task hook function is called (very) approximately 4.5 million times between each iteration of the application tasks.
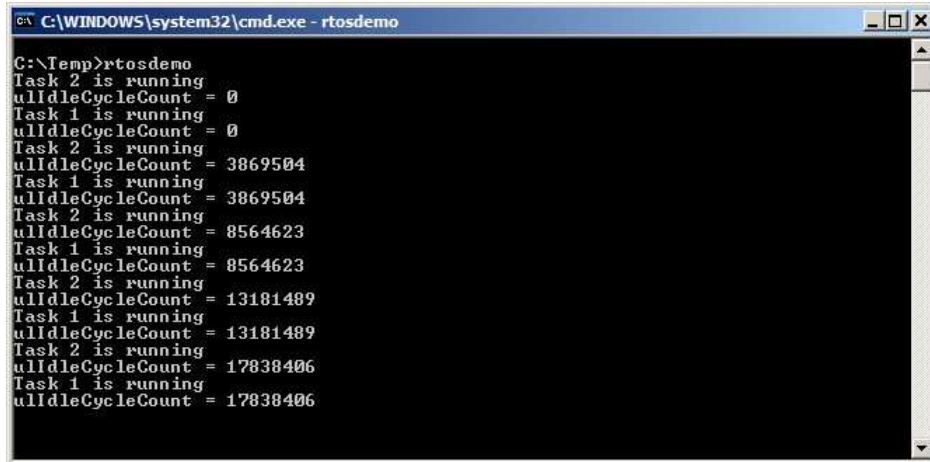
Figure 13 The output produced when Example 7 is executed

1.8 CHANGING THE PRIORITY OF A TASK

vTaskPrioritySet() API function

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

```
void vTaskPrioritySet( xTaskHandle pxTask, unsigned portBASE_TYPE uxNewPriority );
```

Listing 20 The vTaskPrioritySet() API function prototype

Table 4 vTaskPrioritySet() parameters

Parameter Name	Description
pxTask	The handle of the task whose priority is being modified (the subject task) – see the pxCreatedTask parameter of the xTaskCreate() API function for information on obtaining handles to tasks. A task can change its own priority by passing NULL in place of a valid task handle.
uxNewPriority	The priority to which the subject task is to be set. This will be automatically capped to the maximum available priority of (configMAX_PRIORITIES – 1), where configMAX_PRIORITIES is a compile time option set in the FreeRTOSConfig.h header file.

uxTaskPriorityGet() API function

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

```
unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle pxTask );
```

Listing 21 The uxTaskPriorityGet() API function prototype

Table 5 uxTaskPriorityGet() parameters and return value

Parameter Name/Return Value	Description
pxTask	The handle of the task whose priority is being queried (the subject task) – see the pxCreatedTask parameter of the xTaskCreate() API function for information on obtaining handles to tasks. A task can query its own priority by passing NULL in place of a valid task handle.
Returned value	The priority currently assigned to the task being queried.

Example 8. Changing task priorities

The scheduler will always select the highest Ready state task as the task to enter the Running state. Example 8 demonstrates this by using the vTaskPrioritySet() API function to change the priority of two tasks relative to each other.

Two tasks are created at two different priorities. Neither task make any API function calls that could cause them to enter the Blocked state so both are always in either the Ready or the Running state – as such the task with the highest relative priority will always be the task selected by the scheduler to be in the Running state.

Example 8 behaves as follows:

- Task1 (Listing 22) is created with the highest priority so is guaranteed to run first. Task1 prints out a couple of strings before raising the priority of Task2 (Listing 23) to above its own priority.

- Task2 starts to run (enters the Running state) as soon as it has the highest relative priority. Only one task can be in the Running state at any one time so when Task2 is in the Running state Task1 is in the Ready state.

- Task2 prints out a message before setting its own priority back down to below that of Task1.

- Task2 setting its priority back down means Task1 is once again the highest priority task, so Task1 re-enters the Running state, forcing Task2 back into the Ready state.

```
void vTask1( void *pvParameters )
{
unsigned portBASE_TYPE uxPriority;

    /* This task will always run before Task2 as it is created with the higher
    priority.  Neither Task1 nor Task2 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 my priority". */
    uxPriority = uxTaskPriorityGet( NULL );

    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Task1 is running\r\n" );

        /* Setting the Task2 priority above the Task1 priority will cause
        Task2 to immediately start running (as then Task2 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 24 shows how
        the handle was obtained. */
        vPrintString( "About to raise the Task2 priority\r\n" );
        vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );

        /* Task1 will only run when it has a priority higher than Task2.
        Therefore, for this task to reach this point Task2 must already have
        executed and set its priority back down to below the priority of this
        task. */
    }
}
```

Listing 22 The implementation of Task1 in Example 8

Designed For Microcontrollers;

```
void vTask2( void *pvParameters )
{
unsigned portBASE_TYPE uxPriority;

    /* Task1 will always run before this task as Task1 is created with the
    higher priority.  Neither Task1 nor Task2 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 my priority". */
    uxPriority = uxTaskPriorityGet( NULL );

    for( ;; )
    {
        /* For this task to reach this point Task1 must have already run and
        set the priority of this task higher than its own.

        Print out the name of this task. */
        vPrintString( "Task2 is running\r\n" );

        /* Set our priority back down to its original value.  Passing in NULL
        as the task handle means "change my priority".  Setting the
        priority below that of Task1 will cause Task1 to immediately start
        running again – pre-empting this task. */
        vPrintString( "About to lower the Task2 priority\r\n" );
        vTaskPrioritySet( NULL, ( uxPriority - 2 ) );
    }
}
```

Listing 23 The implementation of Task2 in Example 8

Each task can both query and set its own priority without the use of a valid task handle – NULL is just used in its place. A task handle is only required when a task wishes to reference a task other than itself as when Task1 changes the priority of Task2. To allow Task1 to do this the Task2 handle is obtained and saved when Task2 is created as highlighted in the comments within Listing 24.

```
/* Declare a variable that is used to hold the handle of Task2. */
xTaskHandle xTask2Handle;

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 Task1.  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 that
    there was insufficient heap memory available for the idle task to be created.
    CHAPTER 5 provides more information on memory management. */
    for( ;; );
}
```

Listing 24 The implementation of main() for Example 8

Figure 14 demonstrates the sequence in which the Example 8 tasks execute, with the resultant output shown in Figure 15.
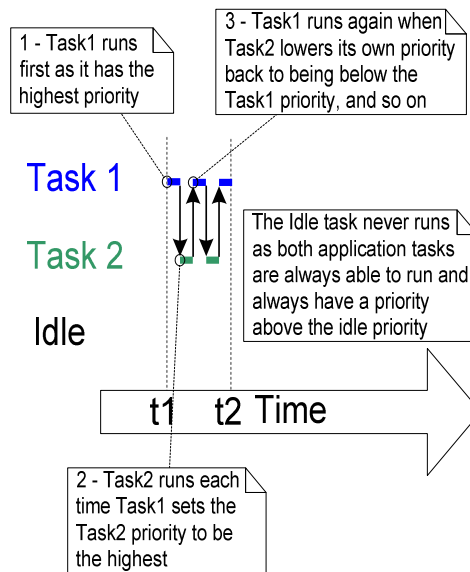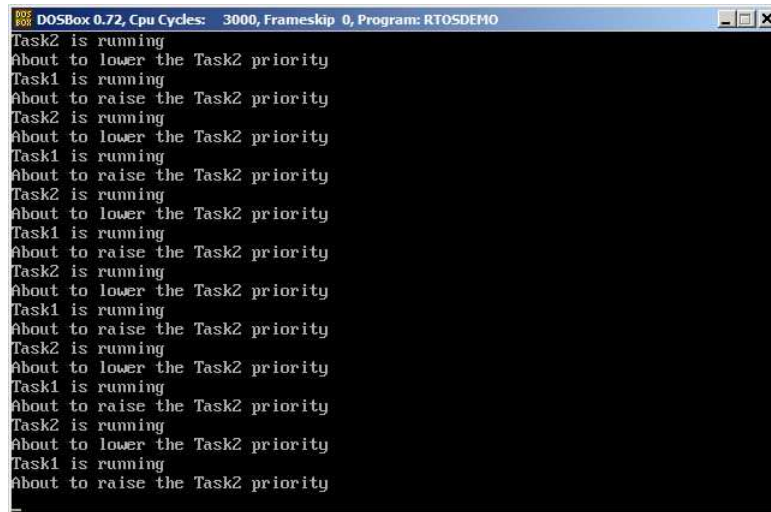
Figure 14 The sequence of task execution when running Example 8

Figure 15 The output produced when Example 8 is executed

1.9   DELETING A TASK

vTaskDelete() API function

A task can use the vTaskDelete() API function to delete itself or any other task.

Deleted tasks no longer exist and cannot enter the Running state again.

It is the responsibility of the idle task to free memory that was allocated to tasks that have since been deleted. It is therefore important that applications that make use of the vTaskDelete() API function do not completely starve the idle task of all processing time.

Note also that only memory that is allocated to a task by the kernel itself will be automatically freed when the task is deleted. Any memory or other resource that the implementation of the task allocates itself must be freed explicitly.

```
void vTaskDelete( xTaskHandle pxTaskToDelete );
```

Listing 25 The vTaskDelete() API function prototype

Table 6 vTaskDelete() parameters

Parameter Name/Return Value	Description
pxTaskToDelete	The handle of the task that is to be deleted (the subject task) – see the pxCreatedTask parameter of the xTaskCreate() API function for information on obtaining handles to tasks. A task can delete itself by passing NULL in place of a valid task handle.

Example 9. Deleting tasks

This is a very simple example that behaves as follows:

- Task 1 is created by main() with priority 1. When it runs it creates Task 2 at priority 2. Task 2 is now the highest priority task so will start to execute immediately. The source for main() is shown in Listing 26, and for Task 1 in Listing 27.

- Task 2 does nothing but delete itself. It could delete itself by passing NULL to vTaskDelete(), but purely for demonstration purposes it instead uses its own task handle. The source for Task2 is shown in Listing 28.

- When task 2 has been deleted Task 1 will again be the highest priority task so continue executing – at which point it calls vTaskDelay() to block for a short period.

- The Idle task will execute while Task 1 is in the blocked state and free the memory that was allocated to the now deleted Task 2.

- When Task 1 leaves the blocked state it will once again be the highest priority Ready state task and so pre-empt the Idle task. When it enters the Running state it simply creates Task 2 again, and so it goes on.

```c
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 tasks start executing. */
    vTaskStartScheduler();

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

Listing 26 The implementation of main() for Example 9

```
void vTask1( void *pvParameters )
{
const portTickType xDelay100ms = 100 / portTICK_RATE_MS;

    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Task1 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 _____^^^^^^^^^^^^^ */

        /* Task2 has/had the higher priority, so for Task1 to reach here Task2
        must have already executed and deleted itself.  Delay for 100
        milliseconds. */
        vTaskDelay( xDelay100ms );
    }
}
```

Listing 27 The implementation of Task 1 for Example 9

```
void vTask2( void *pvParameters )
{
    /* Task2 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
    instead calls vTaskDelete() passing its own task handle. */
    vPrintString( "Task2 is running and about to delete itself\r\n" );
    vTaskDelete( xTask2Handle );
}
```

Listing 28 The implementation of Task 2 for Example 9
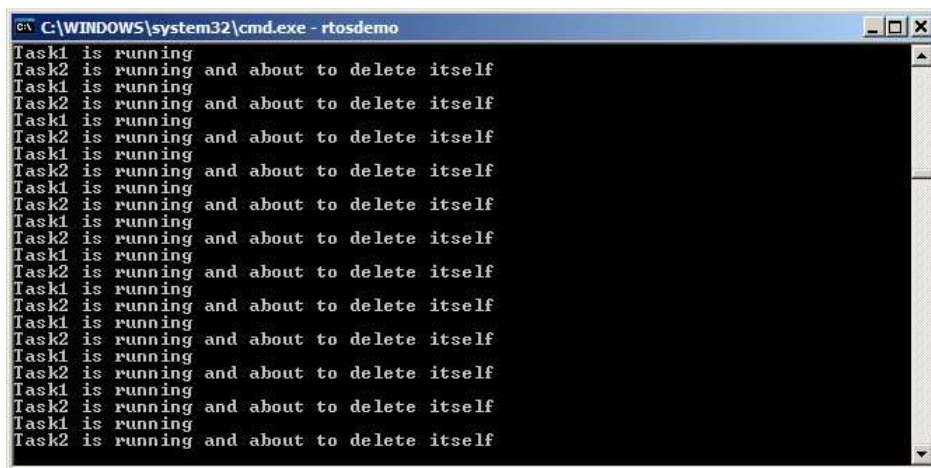
Figure 16 The output produced when Example 9 is executed

2 - Task 2 does nothing other than delete itself, allowing execution to return to Task 1.

1 - Task 1 runs and creates Task 2. Task 2 starts to run immediately as it has the higher priority.

3 - Task 1 calls vTaskDelay(), allowing the idle task to run until the delay time expires, and the whole sequence repeats.
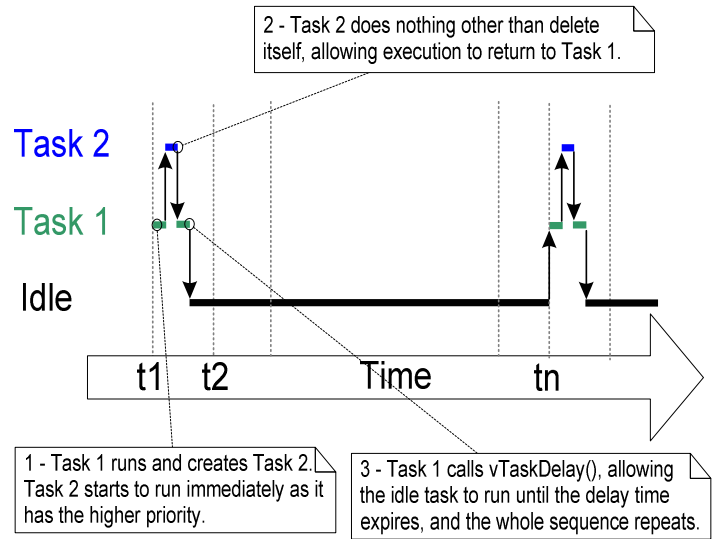
Figure 17 The execution sequence for example 9

1.10 THE SCHEDULING ALGORITHM – A SUMMARY

Prioritized Preemptive Scheduling

The examples in this chapter have illustrated how and when FreeRTOS selects which task should be in the Running state:

- Each task is assigned a priority.

- Each task can exist in one of several states.

- Only one task can exist in the Running state at any one time.

- The scheduler will always select the highest priority Ready state task to enter the Running state.

This type of scheme is called 'Fixed Priority Preemptive Scheduling'. 'Fixed Priority' because each task is assigned a priority that is not altered by the kernel itself (only tasks can change priorities). 'Preemptive' because a task entering the Ready state or having its priority altered will always pre-empt the Running state task if the Running state task has a lower priority.

Tasks can wait in the Blocked state for an event and will be automatically moved back to the Ready state when the event occurs. Temporal events occur at a particular time, for example when a block time expires. They are generally used to implement periodic or timeout behavior. Synchronization events occur when a task or interrupt service routine sends information to queue or one of the many types of semaphore. They are generally used to signal asynchronous activity, such as data arriving at a peripheral.

Figure 18 demonstrates all this behavior by illustrating the execution pattern of a hypothetical application.
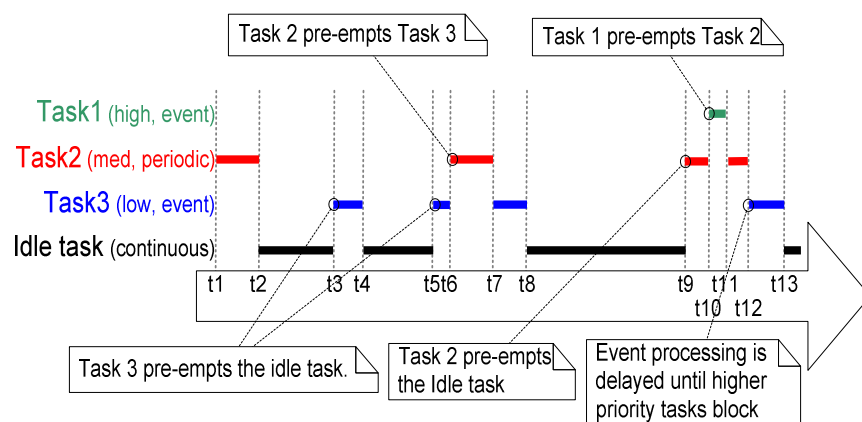
Figure 18 Execution pattern with pre-emption points highlighted

Designed For Microcontrollers;

Referring to Figure 18:

1. Idle Task

The idle task is running at the lowest priority so gets pre-empted every time a higher priority task enters the Ready state – for example at times t3, t5 and t9.

2. Task 3

Task 3 is an event driven task that executes with a relatively low priority but above the Idle task priority. It spends most of its time in the Blocked state waiting for the event of interest, transitioning from the Blocked state to the Ready state each time the event occurs. All FreeRTOS inter-task communication mechanisms (queues, semaphores, etc) can be used to signal events and unblock tasks in this way.

Events occur at times t3, t5 and also somewhere between t9 and t12. The events occurring at times t3 and t5 are processed immediately as at these times Task 3 is the highest priority task that is able to run. The event that occurs somewhere between times t9 and t12 is not processed until t12 because until then the higher priority tasks Task 1 and Task 2 are still executing. It is only at time t12 that both Task 1 and Task 2 are in the Blocked state making Task 3 the highest priority Ready state task.

3. Task 2

Task 2 is a periodic task that executes at a priority above the priority of Task 3 but below the priority of Task 1. The period interval means Task 2 wants to execute at times t1, t6 and t9.

At time t6 Task 3 is in the Running state, but Task 2 has the higher relative priority so pre-empts Task 3 and starts executing immediately. Task 2 completes its processing and re-enters the Blocked state at time t7, at which point Task 3 can re-enter the Running state to complete its processing. Task 3 itself Blocks at time t8.

4. Task 1

Task 1 is also an event driven task. It executes with the highest priority of all so can pre-empt any other task in the system. The only Task 1 event shown occurs at time t10, at which time Task 1 pre-empts Task 2. Task 2 can only complete its processing after Task 1 has re-entered the Blocked at time t11.

Selecting Task Priorities

Figure 18 shows how fundamental priority assignment is to the way an application behaves.

As a general rule tasks that implement hard real time functions are assigned priorities above those that implement soft real time functions. However other characteristics, such as execution times and processor utilization must also be taken into account to ensure the entire application will never miss a hard real time deadline.

Rate Monotonic Scheduling (RMS) is a common priority assignment technique that dictates a unique priority be assigned to each task in accordance with the tasks periodic execution rate. The highest priority is assigned to the task that has the highest frequency of periodic execution. The lowest priority is assigned to the task with the lowest frequency of periodic execution. Assigning priorities in this way has been shown to maximize the 'schedulability' of the entire application, but run time variations and the fact that not all tasks are in any way periodic makes absolute calculations a complex process.

Co-operative Scheduling

This book focuses on preemptive scheduling. FreeRTOS can also optionally use co-operative scheduling.

When a pure co-operative scheduler is used a context switch will only occur when either the Running state task enters the Blocked state or the Running state task explicitly calls taskYIELD(). Tasks will never be pre-empted and tasks of equal priority will not automatically share processing time. Co-operative scheduling in this manner is simpler but can potentially result in a less responsive system.

A hybrid scheme is also possible where interrupt service routines are used to explicitly cause a context switch. This allows synchronization events to cause pre-emption, but not temporal events. The result is a preemptive system without timeslicing. This can be desirable because of its efficiency gains and is a common scheduler configuration.

CHAPTER 2

QUEUE MANAGEMENT

2.1   CHAPTER INTRODUCTION AND SCOPE

Applications that use FreeRTOS are structured as a set of independent tasks – each task is effectively a mini program in its own right. It is likely that this collection of autonomous tasks will have to communicate with each other in order that collectively they can provide useful system functionality. Queues are the underlying primitive used by all FreeRTOS communication and synchronization mechanisms.

Scope

This chapter aims to give readers a good understanding of:

- How to create a queue.

- How a queue manages the data it contains.

- How to send data to a queue.

- How to receive data from a queue.

- What it means to block on a queue.

- The effect task priorities have when writing to and reading from a queue.

Only task to task communication is covered in this chapter. Task to interrupt and interrupt to task communication is covered in CHAPTER 3.

2.2   CHARACTERISTICS OF A QUEUE

Data Storage

A queue can hold a finite number of fixed size data items. The maximum number of items a queue can hold is called its 'length'. Both the length and the size of each data item are set when the queue is created.

Normally queues are used as First In First Out (FIFO) buffers where data is written to the end (tail) of the queue and removed from the front (head) of the queue. It is also possible to write to the front of a queue.

Writing data to a queue causes a byte for byte *copy* of the data to be stored in the queue itself. Reading data from a queue causes the *copy* of the data to be removed from the queue. Figure 19 demonstrates data being written to and read from a queue, and the effect on the data stored in the queue of each operation.

Access by Multiple Tasks

Queues are objects in their own right that are not owned by or assigned to any particular task. Any number of tasks can write to the same queue and any number of tasks can read from the same queue. A queue having multiple writers is very common while a queue having multiple readers is quite rare.

Blocking on Queue Reads

When a task attempts to read from a queue it can optionally specify a 'block' time. This is the time the task should be kept in the Blocked state to wait for data to be available from the queue should the queue already be empty. A task that is in the Blocked state waiting for data to become available from a queue is automatically moved to the Ready state when another task or interrupt places data into the queue. The task will also be automatically moved from the Blocked state to the Ready state if the specified block time expires before data becomes available.

Queues can have multiple readers so it is possible that a single queue will have more than one task blocked on it waiting for data. When this is the case only one task will be unblocked when data becomes available. The task that is unblocked will always be the highest priority task that was waiting for data. If the blocked tasks have equal priority then it will be the task that has been waiting for data the longest that is unblocked.

Blocking on Queue Writes

Just as when reading from a queue, a task can optionally specify a block time when writing to a queue. In this case the block time is the maximum time the task should be held in the Blocked state to wait for space to become available on the queue should the queue already be full.

Queues can have multiple writers so it is possible that a full queue will have more than one task blocked on it waiting to complete a send operation. When this is the case only one task will be unblocked when space on the queue becomes available. The task that is unblocked will always be the highest priority task that was waiting for space. If the blocked tasks have equal priority then it will be the task that has been waiting for space the longest that is unblocked.
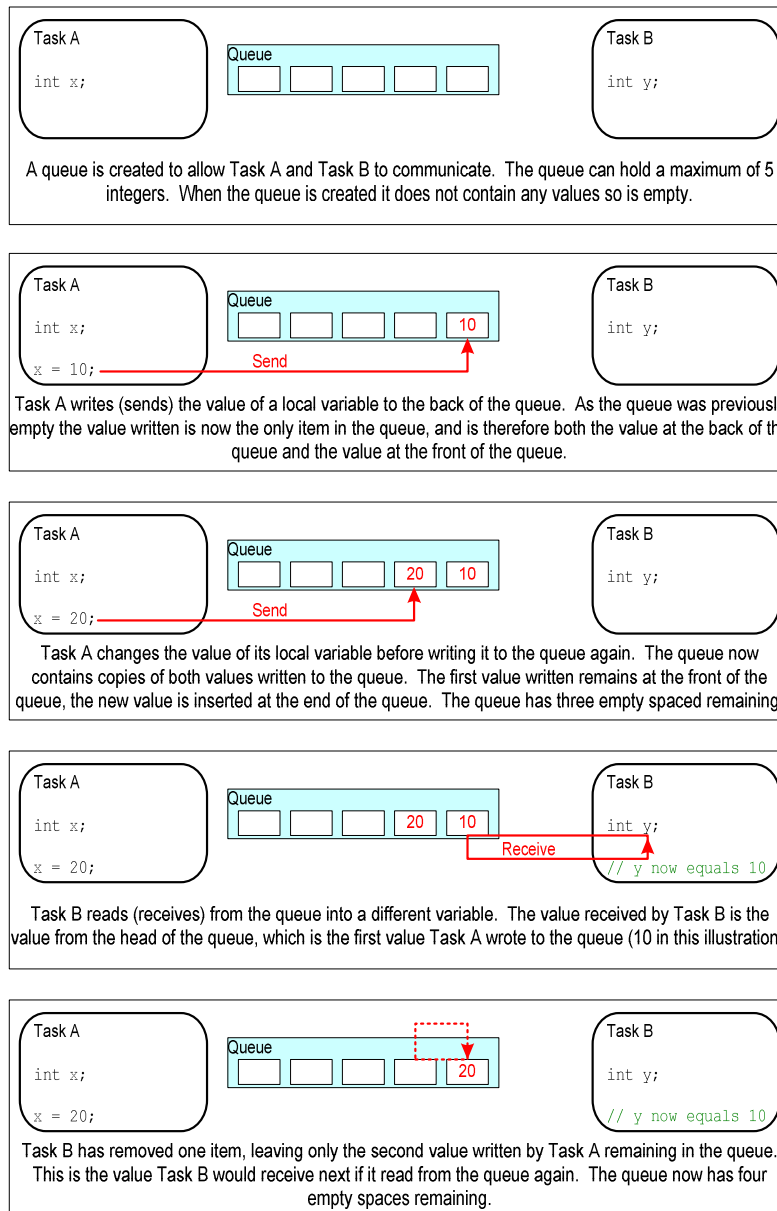
Figure 19 An example sequence of writes and reads to/from a queue

2.3   USING A QUEUE

xQueueCreate() API Function

A queue must be explicitly created before it can be used.

Queues are referenced using variables of type xQueueHandle. xQueueCreate() is used to create a queue and returns an xQueueHandle to reference the queue it creates.

FreeRTOS allocates RAM from the FreeRTOS heap when a queue is created. The RAM is used to hold both the queue data structures and the items that are contained in the queue.   xQueueCreate() will return NULL if there is insufficient heap RAM available for the queue to be created. CHAPTER 5 provides more information on heap memory management.

```
xQueueHandle xQueueCreate( unsigned portBASE_TYPE uxQueueLength,
                           unsigned portBASE_TYPE uxItemSize
                         );
```

Listing 29 The xQueueCreate() API function prototype

Table 7 xQueueCreate() parameters and return value

Parameter Name	Description
uxQueueLength	The maximum number of items that the queue being created can hold at any one time.
uxItemSize	The size in bytes of each data item that can be stored in the queue.
Return Value	If NULL is returned then the queue could not be created because there was insufficient heap memory available for FreeRTOS to allocate the queue data structures and storage area.
	A non-NULL value being returned indicates that the queue was created successfully. The returned value should be stored as the handle to the created queue.

xQueueSendToBack() and xQueueSendToFront() API Functions

As might be expected, xQueueSendToBack() is used to send data to the back (tail) of a queue, and xQueueSendToFront() is used to send data to the front (head) of a queue.

xQueueSend() is equivalent to and exactly the same as xQueueSendToBack().

Never call xQueueSendToFront() or xQueueSendToBack() from an interrupt service routine. The interrupt safe versions xQueueSendToFrontFromISR() and xQueueSendToBackFromISR() should be used in their place. These are described in CHAPTER 3.

```
portBASE_TYPE xQueueSendToFront(   xQueueHandle xQueue,
                                   const void * pvItemToQueue,
                                   portTickType xTicksToWait
                                );
```

Listing 30 The xQueueSendToFront() API function prototype

```
portBASE_TYPE xQueueSendToBack(  xQueueHandle xQueue,
                                  const void * pvItemToQueue,
                                  portTickType xTicksToWait
                                );
```

Listing 31 The xQueueSendToBack() API function prototype

Table 8 xQueueSendToFront() and xQueueSendToBack() function parameters and return value

Parameter Name/Returned Value	Description
xQueue	The handle of the queue to which the data is being sent (written). The queue handle will have been returned from the call to xQueueCreate() used to create the queue.
pvItemToQueue	A pointer to the data that will be copied into the queue. The size of each item that the queue can hold is set when the queue is created, so this many bytes will be copied from pvItemToQueue into the queue storage area.

Table 8 xQueueSendToFront() and xQueueSendToBack() function parameters and return value

Parameter Name/Returned Value	Description
xTicksToWait	The maximum amount of time the task should remain in the Blocked state to wait for space to become available on the queue, should the queue already be full.

Both xQueueSendToFront() and xQueueSendToBack() will return immediately if xTicksToWait is 0 and the queue is already full.

The block time is specified in tick periods so the absolute time is represents is dependent on the tick frequency. The constant portTICK_RATE_MS can be used to convert a time specified in milliseconds to a time specified in ticks.

Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out) provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h. |
| Returned value | There are two possible return values:

    1. pdPASS

pdPASS will only be returned if data was successfully sent to the queue.

If a block time was specified (xTicksToWait was not zero) then it is possible that the calling task was placed in the Blocked state to wait for space to become available in the queue before the function returned – but data was successfully written to the queue before the block time expired.

    2. errQUEUE_FULL

errQUEUE_FULL will be returned if data could not be written to the queue because the queue was already full.

If a block time was specified (xTicksToWait was not zero) then the calling task will have been placed into the Blocked state to wait for another task or interrupt to make room in the queue, but the specified block time expired before this happened. |

xQueueReceive() and xQueuePeek() API Functions

xQueueReceive() is used to receive (read) an item from a queue. The item that is received is removed from the queue.

xQueuePeek() is used to receive an item from a queue *without* the item being removed from the queue. xQueuePeek() will receive the item from the head of the queue without modifying the data that is stored in the queue, or the order in which data is stored in the queue.

Never call xQueueReceive() or xQueuePeek() from an interrupt service routine. The interrupt safe xQueueReceiveFromISR() API function is described in CHAPTER 3.

```
portBASE_TYPE xQueueReceive(
                        xQueueHandle xQueue,
                        const void * pvBuffer,
                        portTickType xTicksToWait
                    );
```

Figure 20 The xQueueReceive() API function prototype

```
portBASE_TYPE xQueuePeek(
                        xQueueHandle xQueue,
                        const void * pvBuffer,
                        portTickType xTicksToWait
                    );
```

Listing 32 The xQueuePeek() API function prototype

Table 9 xQueueReceive() and xQueuePeek() function parameters and return values

Parameter Name/Returned value	Description
xQueue	The handle of the queue from which the data is being received (read). The queue handle will have been returned from the call to xQueueCreate() used to create the queue.
pvBuffer	A pointer to the memory into which the received data will be copied. The size of each data item that the queue holds is set when the queue is created. The memory pointed to by pvBuffer must be at least large enough to hold that many bytes.

Table 9 xQueueReceive() and xQueuePeek() function parameters and return values

Parameter Name/Returned value	Description
xTicksToWait	The maximum amount of time the task should remain in the Blocked state to wait for data to become available on the queue, should the queue already be empty. If xTicksToWait is zero then both xQueueReceive() and xQueuePeek() will return immediately if the queue is already empty. The block time is specified in tick periods so the absolute time it represents is dependent on the tick frequency. The constant portTICK_RATE_MS can be used to convert a time specified in milliseconds to a time specified in ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out) provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.
Returned value	There are two possible return values: 1. pdPASS pdPASS will only be returned if data was successfully read from the queue. If a block time was specified (xTicksToWait was not 0) then it is possible that the calling task was placed in the Blocked state to wait for data to become available on the queue, but data was successfully read from the queue before the block time expired. 2. errQUEUE_EMPTY errQUEUE_EMPTY is returned when data could not be read from the queue because the queue was already empty. If a block time was specified (xTicksToWait was not zero) then the calling task will have been placed into the Blocked state to wait for another task or interrupt to send data to the queue, but the block time expired before this happened.

uxQueueMessagesWaiting() API Function

uxQueueMessagesWaiting() is used to query the number of items that are currently in a queue.

Never call uxQueueMessagesWaiting() from an interrupt service routine. The interrupt safe uxQueueMessagesWaitingFromISR() should be used in its place.

```
unsigned portBASE_TYPE uxQueueMessagesWaiting( xQueueHandle xQueue );
```

Listing 33 The uxQueueMessagesWaiting() API function prototype

Table 10 uxQueueMessagesWaiting() function parameters and return value

Parameter Name/Returned Value	Description
xQueue	The handle of the queue being queried. The queue handle will have been returned from the call to xQueueCreate() used to create the queue.
Returned value	The number of items that the queue being queried is currently holding. If 0 is returned then the queue is empty.

Example 10. Blocking When Receiving From a Queue

This example demonstrates a queue being created, data being sent to the queue from multiple tasks, and data being received from the queue. The queue is created to hold data items of type long. The tasks that send to the queue do not specify a block time, while the task that receives from the queue does.

The priority of the tasks that send to the queue is lower than the priority of the task that receives from the queue. This means the queue should never contain more than one item because as soon as data is sent to the queue the receiving task will unblock, pre-empt the sending task, and remove the data – leaving the queue empty once again.

Listing 34 shows the implementation of the task that writes to the queue. Two instances of this task are created, one that continuously writes the value 100 to the queue, and another that continuously writes the value 200 to the same queue. The task parameter is used to pass these values into each task instance.

```
static void vSenderTask( void *pvParameters )
{
long lValueToSend;
portBASE_TYPE xStatus;

    /* Two instances of this task are created so the value that is sent to the
    queue is passed in via the task parameter - this way each instance can use
    a different value.  The queue was created to hold values of type long,
    so cast the parameter to the required type. */
    lValueToSend = ( long ) pvParameters;

    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* Send the value to the queue.

        The first parameter is the queue to which data is being sent.  The
        queue was created before the scheduler was started, so before this task
        started to execute.

        The second parameter is the address of the data to be sent, in this case
        the address of lValueToSend.

        The third parameter is the Block time – the time the task should be kept
        in the Blocked state to wait for space to become available on the queue
        should the queue already be full.  In this case a block time is not
        specified because the queue should never contain more than one item and
        therefore never be full. */
        xStatus = xQueueSendToBack( xQueue, &lValueToSend, 0 );

        if( xStatus != pdPASS )
        {
            /* The send operation could not complete because the queue was full -
            this must be an error as the queue should never contain more than
            one item! */
            vPrintString( "Could not send to the queue.\r\n" );
        }

        /* Allow the other sender task to execute.  taskYIELD() informs the
        scheduler that a switch to another task should occur now rather than
        keeping this task in the Running state until the end of the current time
        slice. */
        taskYIELD();
    }
}
```

Listing 34 Implementation of the sending task used in Example 10.

Listing 35 shows the implementation of the task that receives data from the queue. The receiving task specifies a block time of 100 milliseconds so will enter the Blocked state to wait for data to become available. It will leave the Blocked state when either data is available on the queue, or 100 milliseconds passes without data becoming available. In this example the 100 milliseconds timeout should never expire as there are two tasks that are continuously writing to the queue.

```
static void vReceiverTask( void *pvParameters )
{
/* Declare the variable that will hold the values received from the queue. */
long lReceivedValue;
portBASE_TYPE xStatus;
const portTickType xTicksToWait = 100 / portTICK_RATE_MS;

    /* This task is also defined within an infinite loop. */
    for( ;; )
    {
        /* This call should always find the queue empty because this task will
        immediately remove any data that is written to the queue. */
        if( uxQueueMessagesWaiting( xQueue ) != 0 )
        {
            vPrintString( "Queue should have been empty!\r\n" );
        }

        /* Receive data from the queue.

        The first parameter is the queue from which data is to be received.  The
        queue is created before the scheduler is started, and therefore before this
        task runs for the first time.

        The second parameter is the buffer into which the received data will be
        placed.  In this case the buffer is simply the address of a variable that
        has the required size to hold the received data.

        The last parameter is the block time – the maximum amount of time that the
        task should remain in the Blocked state to wait for data to be available
        should the queue already be empty.  In this case the constant
        portTICK_RATE_MS is used to convert 100 milliseconds to a time specified in
        ticks. */
        xStatus = xQueueReceive( xQueue, &lReceivedValue, xTicksToWait );

        if( xStatus == pdPASS )
        {
            /* Data was successfully received from the queue, print out the received
            value. */
            vPrintStringAndNumber( "Received = ", lReceivedValue );
        }
        else
        {
            /* Data was not received from the queue even after waiting for 100ms.
            This must be an error as the sending tasks are free running and will be
            continuously writing to the queue. */
            vPrintString( "Could not receive from the queue.\r\n" );
        }
    }
}
```

Listing 35 Implementation of the receiver task for Example 10.

Listing 36 contains the definition of the main() function. This simply creates the queue and the three tasks before starting the scheduler. The queue is created to hold a maximum of 5 long values even

though the priorities of the tasks are set so the queue will never actually contain more than one item at a time.

```c
/* Declare a variable of type xQueueHandle.  This is used to store the reference
to the queue that is accessed by all three tasks. */
xQueueHandle xQueue;


int main( void )
{
    /* The queue is created to hold a maximum of 5 values, each of which is
    large enough to hold a variable of type long. */
    xQueue = xQueueCreate( 5, sizeof( long ) );

    if( xQueue != NULL )
    {
        /* Create two instances of the task that will send to the queue.  The task
        parameter is used to pass the value that the task will write to the queue,
        so one task will continuously write 100 to the queue while the other task
        will continuously write 200 to the queue.  Both tasks are created at
        priority 1. */
        xTaskCreate( vSenderTask, "Sender1", 1000, ( void * ) 100, 1, NULL );
        xTaskCreate( vSenderTask, "Sender2", 1000, ( void * ) 200, 1, NULL );

        /* Create the task that will read from the queue.  The task is created with
        priority 2, so above the priority of the sender tasks. */
        xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 2, NULL );

        /* Start the scheduler so the created tasks start executing. */
        vTaskStartScheduler();
    }
    else
    {
        /* The queue could not be created. */
    }

    /* 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( ;; );
}
```

Listing 36 The implementation of main()Example 10

The tasks that send to the queue call taskYIELD() on each iteration of their infinite loop. taskYIELD() informs the scheduler that a switch to another task should occur now rather than   keeping the executing task in the Running state until the end of the current time slice. A task that calls taskYIELD() is in effect volunteering to be removed from the Running state. As both tasks that send to the queue have an identical priority each time one calls taskYIELD() the other starts executing – the task that calls taskYIELD() is moved to the Ready state as the other sending task is moved to the Running state. This causes the two sending tasks to send data to the queue in turn. The output produced by Example 10 is shown in Figure 21.
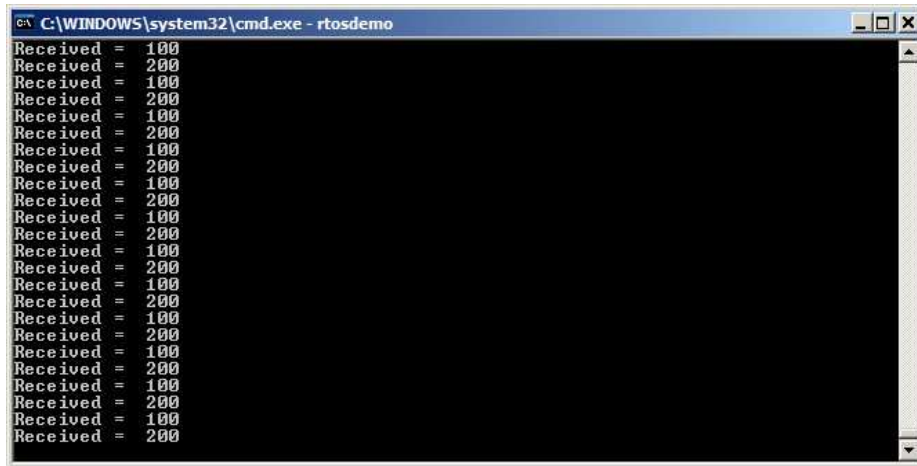
Figure 21 The output produced when Example 10 is executed

Figure 22 demonstrate the sequence of execution.

1 - The Receiver task runs first because it has the highest priority. It attempts to read from the queue. The queue is empty so the Receiver enters the Blocked state to wait for data to become available. Once the Receiver is blocked Sender 2 can run.

3 - The Receiver task empties the queue then enters the Blocked state again, allowing Sender 2 to execute once more. Sender 2 immediately Yields to Sender 1.

2 - Sender two writes to the queue, causing the Receiver to exit the Blocked state. The Receiver has the highest priority so pre-empts Sender 2.

4 - Sender 1 writes to the queue, causing the Receiver to exit the Blocked state and pre-empt Sender 1 - and so it goes on ……..

Figure 22 The sequence of execution produced by Example 10

Using Queues to Transfer Compound Types

It is common for a task to receive data from multiple sources on a single queue. Often the receiver of the data needs to know where the data came from so it can determine how it should be processed. A simple way of achieving this is to use the queue to transfer structures where both the value of the data and the source of the data are contained in the structure fields. This scheme is demonstrated in Figure 23.

```
typedef struct
{
        int iValue;
        int iMeaning;
} xData;
```
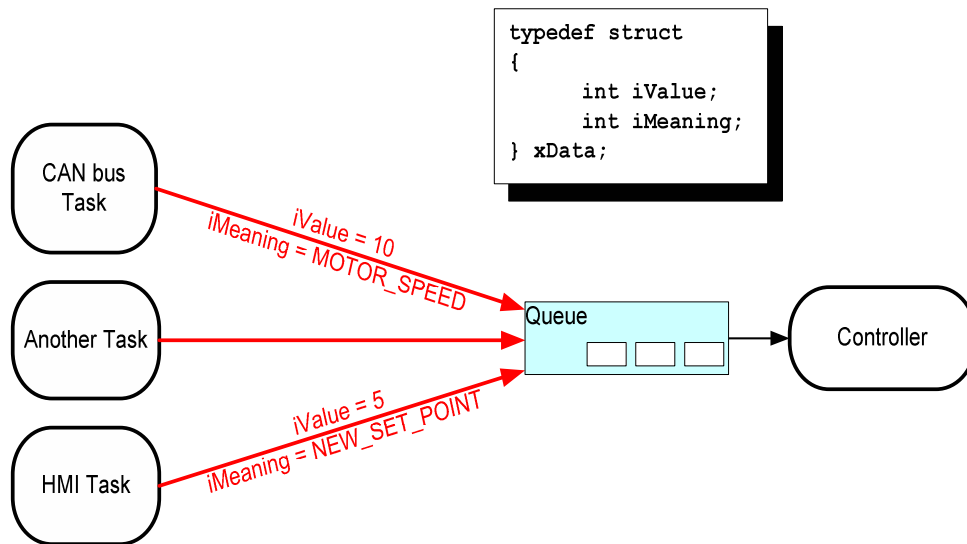
Figure 23 An example scenario where structures are sent on a queue

Referring to Figure 23:

- A queue is created that holds structures of type xData. The structure members allow both a data value and a code indicating what the data means to be sent to the queue in one message.

- A central Controller task is used to perform the primary system function. This has to react to inputs and changes to the system state communicated to it on the queue.

- A CAN bus task is used to encapsulate the CAN bus interfacing functionality. When the CAN bus task has received and decoded a message it sends the already decoded message to the Controller task in an xData structure. The iMeaning member of the transferred structure is used to let the controlling task know what the data is – in the depicted case it is a motor speed. The iValue member of the transferred structure is used to let the Controlling task know the actual motor speed value.

- A Human Machine Interface (HMI) task is used to encapsulate all the HMI functionality. The machine operator can probably input commands and queries values in a number of ways that have to be detected and interpreted within the HMI task. When a new command is input the HMI task sends the command to the Controller task in an xData structure. The iMeaning member of the transferred structure is used to let the controlling task know what the data is – in the depicted case it is a new set point value. The iValue member of the transferred structure is used to let the Controlling task know the actual set point value.

Example 11. Blocking When Sending to a Queue / Sending Structures on a Queue

Example 11 is similar to Example 10, but the task priorities are reversed so the receiving task has a lower priority than the sending tasks. Also the queue is used to pass structures between the tasks rather than simple long integers.

Listing 37 shows the definition of the structure used by Example 11.

```
/* Define the structure type that will be passed on the queue. */
typedef struct
{
      unsigned char ucValue;
      unsigned char ucSource;
} xData;

/* Declare two variables of type xData that will be passed on the queue. */
static const xData xStructsToSend[ 2 ] =
{
      { 100, mainSENDER_1 }, /* Used by Sender1. */
      { 200, mainSENDER_2 }  /* Used by Sender2. */

};
```

Listing 37 The definition of the structure that is to be passed on a queue, plus the declaration of two variables for use by the example.

In Example 10 the receiving task had the highest priority so the queue never contained more than one item. This was because the receiving task pre-empted the sending tasks as soon as data was placed into the queue. In Example 11 the sending tasks have the higher priority so the queue will normally be full. This is because as soon as the receiving task removes an item from the queue it will be pre-empted by one of the sending tasks which will then immediately re-fill the queue. The sending task will then re-enter the Blocked state to wait for space to become available on the queue again.

Listing 38 shows the implementation of the sending task. The sending task specifies a block time of 100 milliseconds so will enter the Blocked state to wait for space to become available each time the queue becomes full. It will leave the Blocked state when either space is available on the queue or 100 milliseconds passes without space becoming available. In this example the 100 milliseconds timeout should never expire as the receiving task is continuously making space by removing items from the queue.

```
static void vSenderTask( void *pvParameters )
{
portBASE_TYPE xStatus;
const portTickType xTicksToWait = 100 / portTICK_RATE_MS;

    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* Send to the queue.

        The second parameter is the address of the structure being sent.  The
        address is passed in as the task parameter so pvParameters is used
        directly.

        The third parameter is the Block time - the time the task should be kept
        in the Blocked state to wait for space to become available on the queue
        if the queue is already full.  A block time is specified because the
        sending tasks have a higher priority than the receiving task so the queue
        is expected to become full.  The receiving task will on execute and remove
        items from the queue when both sending tasks are in the Blocked state. */
        xStatus = xQueueSendToBack( xQueue, pvParameters, xTicksToWait );

        if( xStatus != pdPASS )
        {
            /* The send operation could not complete, even after waiting for 100ms.
            This must be an error as the receiving task should make space in the
            queue as soon as both sending tasks are in the Blocked state. */
            vPrintString( "Could not send to the queue.\r\n" );
        }

        /* Allow the other sender task to execute. */
        taskYIELD();
    }
}
```

Listing 38 The implementation of the sending task for Example 11.

The receiving task has the lowest priority so will only run when both sending tasks are in the Blocked state. The sending tasks will only enter the Blocked state when the queue is full, so the receiving task will only execute when the queue is already full. It therefore always expects to receive data even without having to specify a block time.

The implementation of the receiving task is shown in Listing 39.

```
static void vReceiverTask( void *pvParameters )
{
/* Declare the structure that will hold the values received from the queue. */
xData xReceivedStructure;
portBASE_TYPE xStatus;

    /* This task is also defined within an infinite loop. */
    for( ;; )
    {
        /* Because it has the lowest priority this task will only run when the
        sending tasks are in the Blocked state.  The sending tasks will only enter
        the Blocked state when the queue is full so this task always expects the
        number of items in the queue to be equal to the queue length – 3 in this
        case. */
        if( uxQueueMessagesWaiting( xQueue ) != 3 )
        {
            vPrintString( "Queue should have been full!\r\n" );
        }

        /* Receive from the queue.

        The second parameter is the buffer into which the received data will be
        placed.  In this case the buffer is simply the address of a variable that
        has the required size to hold the received structure.

        The last parameter is the block time - the maximum amount of time that the
        task will remain in the Blocked state to wait for data to be available
        if the queue is already empty.  In this case a block time is not necessary
        because this task will only run when the queue is full. */
        xStatus = xQueueReceive( xQueue, &xReceivedStructure, 0 );

        if( xStatus == pdPASS )
        {
            /* Data was successfully received from the queue, print out the received
            value and the source of the value. */
            if( xReceivedStructure.ucSource == mainSENDER_1 )
            {
                vPrintStringAndNumber( "From Sender 1 = ", xReceivedStructure.ucValue );
            }
            else
            {
                vPrintStringAndNumber( "From Sender 2 = ", xReceivedStructure.ucValue );
            }
        }
        else
        {
            /* Nothing was received from the queue.  This must be an error
            as this task should only run when the queue is full. */
            vPrintString( "Could not receive from the queue.\r\n" );
        }
    }
}
```

Listing 39 The definition of the receiving task for Example 11

main() changes only slightly from the previous example. The queue is created to hold three xData structures and the priorities of the sending and receiving tasks are reversed. The implementation of main() is shown in Listing 40.

```
int main( void )
{
    /* The queue is created to hold a maximum of 3 structures of type xData. */
    xQueue = xQueueCreate( 3, sizeof( xData ) );

    if( xQueue != NULL )
    {
        /* Create two instances of the task that will write to the queue.  The
        parameter is used to pass the structure that the task will write to the
        queue, so one task will continuously send xStructsToSend[ 0 ] to the queue
        while the other task will continuously send xStructsToSend[ 1 ].  Both
        tasks are created at priority 2 which is above the priority of the receiver. */
        xTaskCreate( vSenderTask, "Sender1", 1000, &( xStructsToSend[ 0 ] ), 2, NULL );
        xTaskCreate( vSenderTask, "Sender2", 1000, &( xStructsToSend[ 1 ] ), 2, NULL );

        /* Create the task that will read from the queue.  The task is created with
        priority 1, so below the priority of the sender tasks. */
        xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 1, NULL );

        /* Start the scheduler so the created tasks start executing. */
        vTaskStartScheduler();
    }
    else
    {
        /* The queue could not be created. */
    }

    /* 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( ;; );
}
```

Listing 40 The implementation of main() for Example 11

As in Example 10 the tasks that send to the queue yield on each iteration of their infinite loop so take it in turns to send data to the queue. The output produced by Example 11 is shown in Figure 24.
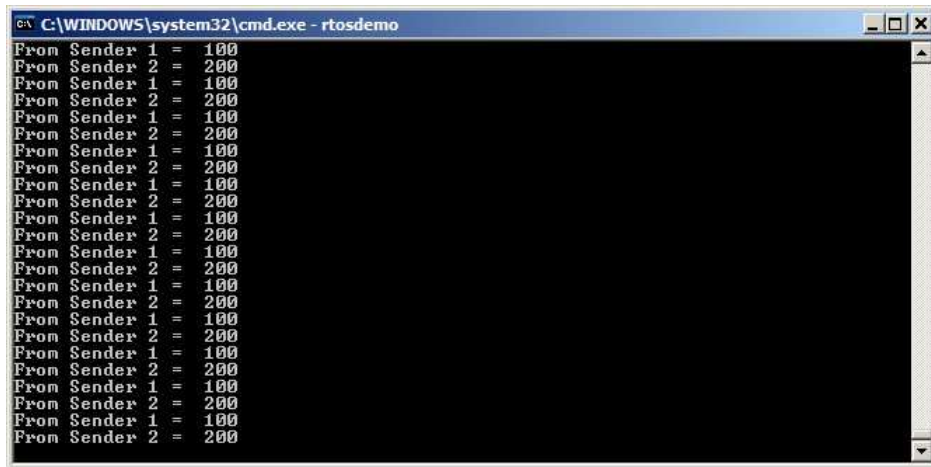
Figure 24 The output produced by Example 11

Figure 25 demonstrates the sequence of execution that results from having the priority of the sending tasks above that of the receiving task. Further explanation of Figure 25 is provided within Table 12.
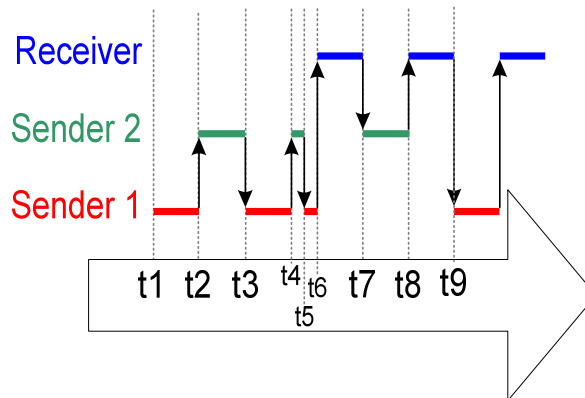
Figure 25 The sequence of execution produced by Example 11

Table 11 Key to Figure 25

Time	Description
t1	Task Sender 1 executes and sends data to the queue.
t2	Sender 1 yields to Sender 2. Sender 2 writes data to the queue.
t3	Sender 2 yields back to Sender 1. Sender 1 writes data to the queue making the queue full.
t4	Sender 1 yields to Sender 2.
t5	Sender 2 attempts to write data to the queue. Because the queue is already full Sender 2 enters the Blocked state to wait for space to become available, allowing Sender 1 to execute once more.
t6	Sender 1 attempts to write data to the queue. Because the queue is already full Sender 1 also enters the Blocked state to wait for space to become available. Now both Sender 1 and Sender 2 are in the Blocked state, so the lower priority Receiver task can execute.
t7	The receiver task removes an item from the queue. As soon as there is space on the queue Sender 2 leaves the Blocked state and, as the higher priority task, pre-empts the Receiver task. Sender 2 writes to the queue, filling the space just created by the receiver task. The queue is now full again. Sender 2 calls taskYIELD() but Sender 1 is still in the Blocked state so Sender 2 is reselected as the Running state task and continues to execute.
t8	Sender 2 attempts to write to the queue. The queue is already full so Sender 2 enters the Blocked state. Once again both Sender 1 and Sender 2 are in the Blocked state so the Receiver task can execute.
t9	The Receiver task removes an item from the queue. As soon as there is space on the queue Sender 1 leaves the Blocked state and, as the higher priority task, pre-empts the Receiver task. Sender 1 writes to the queue, filling the space just created by the receiver task. The queue is now full again. Sender 1 calls taskYIELD() but Sender 2 is still in the Blocked state so Sender 1 is reselected as the Running state task and continues to execute. Sender 1 attempts to write to the queue but the queue is full so Sender 1 enters the Blocked state.
	Both Sender 1 and Sender 2 are again in the Blocked state, allowing the lower priority Receiver task to execute.

2.4 WORKING WITH LARGE DATA

If the size of the data being stored in the queue is large then it is preferable to use the queue to transfer pointers to the data rather than copy the data itself into and out of the queue byte by byte. Transferring pointers is more efficient in both processing time and the amount of RAM required to create the queue. However, when queuing pointers extreme care must be taken to ensure that:

1. The owner of the RAM being pointed to is clearly defined.

When sharing memory between tasks via a pointer it is essential to ensure that both tasks do not modify the memory contents simultaneously, or take any other action that could cause the memory contents to be invalid or inconsistent. Ideally only the sending task should be permitted to access the memory until a pointer to the memory has been queued, and only the receiving task should be permitted to access the memory after the pointer has been received from the queue.

2. The RAM being pointed to remains valid

If the memory being pointed to was allocated dynamically then exactly one task should be responsible for freeing the memory. No tasks should attempt to access the memory after it has been freed.

A pointer should never be used to access data that has been allocated on a task stack. The data will not be valid after the stack frame has changed.

CHAPTER 3

INTERRUPT MANAGEMENT

3.1   CHAPTER INTRODUCTION AND SCOPE

Events

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).   Non-trivial systems will have to service events that originate from multiple sources, all of which will have different processing overhead and response time requirements.   In each case a judgment has to be made as to the best event processing implementation strategy:

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

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

3. How 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?

FreeRTOS does not impose any specific event processing strategy on the application designer, but does provide features that will allow the chosen strategy to be implemented in a simple and maintainable way.

It should be noted that only API functions and macros that end in 'FromISR' or 'FROM_ISR' should ever be used within an interrupt service routine.

Scope

This chapter aims to give readers a good understanding of:

- Which FreeRTOS API functions can be used from within an interrupt service routine.

- How a deferred interrupt scheme can be implemented.

- How to create and use binary semaphores and counting semaphores.

- The differences between binary and counting semaphores.

- How to use a queue to pass data into and out of an interrupt service routine.

- The interrupt nesting model available with some FreeRTOS ports.

3.2   DEFERRED INTERRUPT PROCESSING

Binary Semaphores used for Synchronization

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.

If the interrupt processing is particularly time critical then the handler task priority can be set to ensure the handler task always pre-empts the other tasks in the system. It will then be the task that the ISR returns to when the ISR itself has completed executing. This has the effect of ensuring the entire event processing executes contiguously in time, just as if it had all been implemented within the ISR itself. This scheme is demonstrated in Figure 26.
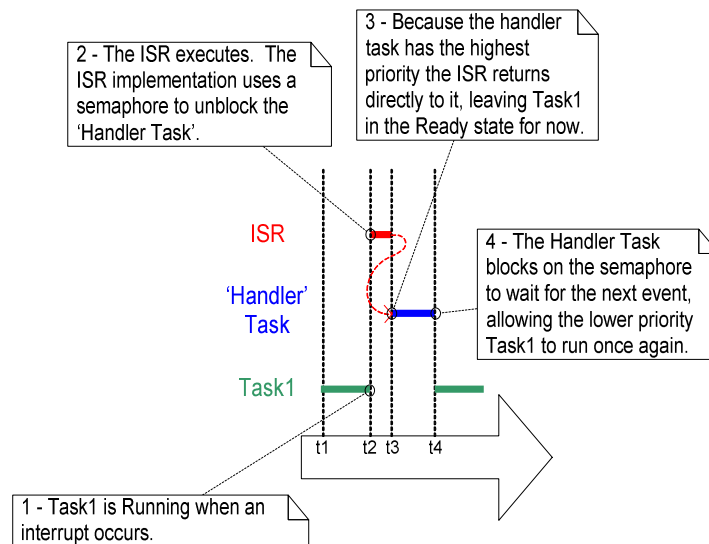
Figure 26 The interrupt interrupts one task, but returns to another.

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 the required event processing can proceed.

'Taking' and 'Giving' a semaphore are concepts that have several different meanings depending on their usage scenario. In classic semaphore terminology Taking a semaphore is equivalent to a P() operation and Giving a semaphore is equivalent to a V() operation.

In this interrupt synchronization scenario the semaphore can be conceptually thought of as a queue that has a length of one. The queue can contain a maximum of one item at any time so is always either empty or full (hence binary). By calling xSemaphoreTake() the 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. When the event occurs the ISR simply uses the xSemaphoreGiveFromISR() function to place a token (the semaphore) into the queue, making the queue full. This causing the handler task to exit the Blocked state and remove the token, leaving the queue empty once more. Once 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. This sequence is demonstrated in Figure 27.

Figure 27 shows the interrupt 'giving' the semaphore even though it has not first 'taken' it, and the task 'taking' the semaphore but never giving it back. This is why the scenario is described as conceptually being similar to writing to and reading from a queue. It often causes confusion as it does not follow the same rules as other semaphore usage scenarios where a task that takes a semaphore must always give it back – such as the scenario described in CHAPTER 4.

vSemaphoreCreateBinary() API Function

Handles to all the various types of FreeRTOS semaphore are stored in a variable of type xSemaphoreHandle.

Before a semaphore can actually be used it must first be created. To create a binary semaphore use the vSemaphoreCreateBinary() API function[2].

```
void vSemaphoreCreateBinary( xSemaphoreHandle xSemaphore );
```

Listing 41 The vSemaphoreCreateBinary() API function prototype

Table 12 vSemaphoreCreateBinary() parameters

Parameter Name	Description
xSemaphore	The semaphore being created. Note that vSemaphoreCreateBinary() is actually implemented as a macro so the semaphore variable should be passed in directly rather than by reference. The examples in this chapter include calls to vSemaphoreCreateBinary() that can be used as a reference and copied.

---

[2] The Semaphore API is actually implemented as a set of macros, not functions. Throughout this book they are referred to as functions for simplicity.
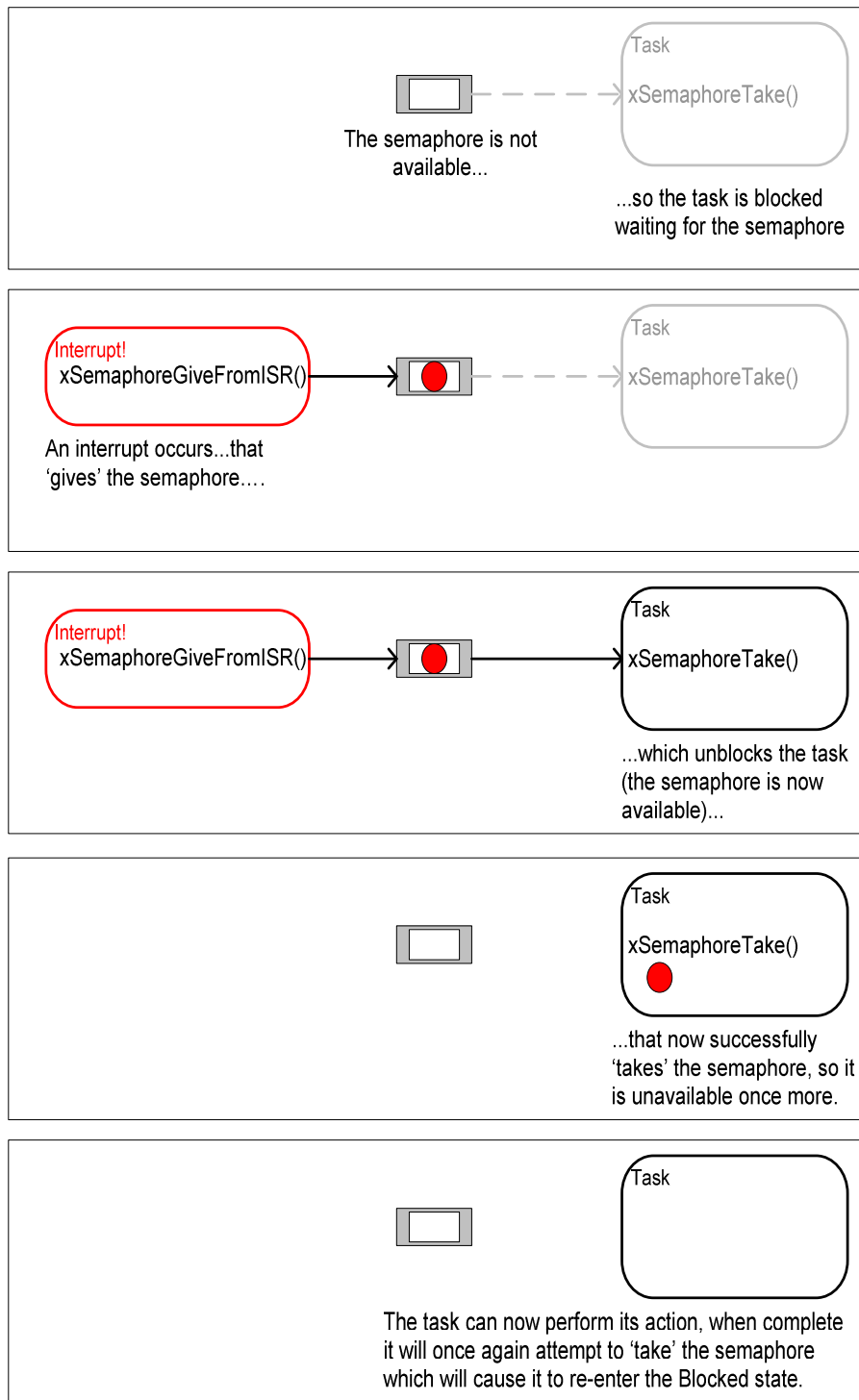
Figure 27 Using a binary semaphore to synchronize a task with an interrupt

xSemaphoreTake() API Function

'Taking' a semaphore means to 'obtain' or 'receive' the semaphore. The semaphore can only be taken if it is available. In classic semaphore terminology xSemaphoreTake() is equivalent to a P() operation.

All the various types of FreeRTOS semaphore *except* recursive semaphores can be 'taken' using the xSemaphoreTake() function.

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

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

Listing 42 The xSemaphoreTake() API function prototype

Table 13 xSemaphoreTake() parameters and return value

Parameter Name / Returned Value	Description
xSemaphore	The semaphore being 'taken'. A semaphore is referenced by a variable of type xSemaphoreHandle and must be explicitly created before it can be used.
xTicksToWait	The maximum amount of time the task should remain in the Blocked state to wait for the semaphore if it is not already available. If xTicksToWait is 0 then xSemaphoreTake() will return immediately if the semaphore is not available. The block time is specified in tick periods so the absolute time it represents is dependent on the tick frequency. The constant portTICK_RATE_MS can be used to convert a time specified in milliseconds to a time specified in ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without a timeout) if INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.
Returned value	There are two possible return values: 1. pdPASS pdPASS will only be returned if the call to xSemaphoreTake() was successful in obtaining the semaphore. If a block time was specified (xTicksToWait was not 0) then it is possible that the calling task was placed in the Blocked state to wait for the semaphore if it was not immediately available, but the semaphore became available before the block time expired. 2. pdFALSE The semaphore was not available. If a block time was specified (xTicksToWait was not zero) then the calling task will have been placed into the Blocked state to wait for the semaphore to become available, but the block time expired before this happened.

xSemaphoreGiveFromISR() API Function

All the various types of FreeRTOS semaphore *except* recursive semaphores can be 'given' using the xSemaphoreGiveFromISR() function.

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

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

Listing 43 The xSemaphoreGiveFromISR() API function prototype

Table 14 xSemaphoreGiveFromISR() parameters and return value

Parameter Name / Returned Value	Description
xSemaphore	The semaphore being 'given'. A semaphore is referenced by a variable of type xSemaphoreHandle and must be explicitly created before being used.
pxHigherPriorityTaskWoken	It is possible that a single semaphore will have one or more tasks blocked on it waiting for the semaphore to become available. Calling xSemaphoreGiveFromISR() can make the semaphore available, and so cause 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 the currently executing task (the task that was interrupted), then xSemaphoreGiveFromISR() will internally set *pxHigherPriorityTaskWoken to pdTRUE. If xSemaphoreGiveFromISR() sets this value to pdTRUE then a context switch should be performed before the interrupt is exited. This will ensure the interrupt returns directly to the highest priority Ready state task.
Returned value	There are two possible return values: 3. pdPASS pdPASS will only be returned if the call to xSemaphoreGiveFromISR() was successful. 4. pdFAIL If a semaphore is already available, it cannot be given, and xSemaphoreGiveFromISR() will return pdFAIL.

Example 12. Using a Binary Semaphore to Synchronize a Task with an Interrupt

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 a software interrupt every 500 milliseconds. A software interrupt is used for convenience because of the difficulties in hooking into a real IRQ in a simulated DOS environment. Listing 44 shows the implementation of the periodic task. Note that the task prints

out a string both before and after the interrupt is generated. This is to allow the sequence of execution to be explicit in output produced when the example is executed.

```c
static void vPeriodicTask( void *pvParameters )
{
    for( ;; )
    {
        /* This task is just used to 'simulate' an interrupt by generating a
        software interrupt every 500ms. */
        vTaskDelay( 500 / portTICK_RATE_MS );

        /* Generate the interrupt, printing a message both before and after
        so the sequence of execution is evident from the output produced when
        the example is executed. */
        vPrintString( "Periodic task - About to generate an interrupt.\r\n" );
        __asm{ int 0x82 } /* This line generates the interrupt. */
        vPrintString( "Periodic task - Interrupt generated.\r\n\r\n\r\n" );
    }
}
```

Listing 44 Implementation of the task that periodically generates a software interrupt in Example 12

Listing 45 shows the implementation of the handler task – the task that is synchronized with the software interrupt through the use of a binary semaphore. Again a message is printed out on each iteration of the task so the sequence in which the task and the interrupt execute is evident from the output produced when the example is executed.

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

        /* To get here the event must have occurred.  Process the event.  In this
        case processing is simply a matter of printing out a message. */
        vPrintString( "Handler task - Processing event.\r\n" );
    }
}
```

Listing 45 The implementation of the handler task (the task that synchronizes with the interrupt) in Example 12

Listing 46 shows the actual interrupt handler. This does very little other than 'give' the semaphore to unblock the handler task. Note how the pxHigherPriorityTaskWoken parameter is used. It is set to pdFALSE before calling xSemaphoreGiveFromISR(), with a context switch being performed if it is subsequently found to equal pdTRUE.

The syntax of the interrupt service routine declaration and the macro called to force a context switch are both specific to the Open Watcom DOS port and will be different for other ports. Please refer to the examples that are included in the demo application for the port being used to find the actual syntax required.

```
static void __interrupt __far vExampleInterruptHandler( void )
{
static portBASE_TYPE xHigherPriorityTaskWoken;

    xHigherPriorityTaskWoken = pdFALSE;

    /* 'Give' the semaphore to unblock the task. */
    xSemaphoreGiveFromISR( xBinarySemaphore, &xHigherPriorityTaskWoken );

    if( xHigherPriorityTaskWoken == pdTRUE )
    {
        /* Giving the semaphore unblocked a task, and the priority of the
        unblocked task is higher than the currently running task - force
        a context switch to ensure that the interrupt returns directly to
        the unblocked (higher priority) task.

        NOTE: The actual macro to use to force a context switch from an
        ISR is dependent on the port.  This is the correct macro for the
        Open Watcom DOS port.  Other ports may require different syntax.
        Refer to the examples provided for the port being used to determine
        the syntax required. */
        portSWITCH_CONTEXT();
    }
}
```

Listing 46 The software interrupt handler used in Example 12

The main() function simply creates the binary semaphore and the tasks, installs the interrupt handler, and starts the scheduler. The implementation is shown in Listing 47.

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

    /* Install the interrupt handler. */
    _dos_setvect( 0x82, vExampleInterruptHandler );

    /* Check the semaphore was created successfully. */
    if( xBinarySemaphore != NULL )
    {
        /* Create the 'handler' task.  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 pre-empted each time the handler task exits the Blocked state. */
        xTaskCreate( vPeriodicTask, "Periodic", 1000, NULL, 1, 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( ;; );
}
```

Listing 47 The implementation of main() for Example 12

Example 12 produces the output shown in Figure 28. As expected the handler task executes immediately that the interrupt is generated, so the output from the handler task splits the output produced by the periodic task. Further explanation is provided in Figure 29.

Figure 28 The output produced when Example 12 is executed

2 - The Periodic task prints its first message then forces an interrupt. The interrupt service routine executes immediately.

3 - The interrupt 'gives' the semaphore, causing the Handler task to unblock. The interrupt service routine then returns directly to the Handler task because the Handler task is the highest priority Ready state task. The handler task prints out its message before returning to the Blocked state to wait for the next interrupt.

Interrupt

Handler

Periodic

Idle

t1   t2   Time

1 - The Idle task is running most of the time. Every 500ms its gets pre-empted by the Periodic task.

4 - The Periodic task is once again the highest priority task - it prints out its second message before entering the Blocked state again to wait for the next time period. This leaves just the Idle task able to run.

Figure 29 The sequence of execution when Example 12 is executed

3.3   COUNTING SEMAPHORES

Example 12 demonstrated a binary semaphore being used to synchronize a task with an interrupt. The execution sequence was as follows:

1. An interrupt occurred.

2. The interrupt service routine executed, 'giving' the semaphore to unblock the Handler task.

3. The Handler task executed as soon as the interrupt completed. The first thing the Handler task did was 'take' the semaphore.

4. The Handler task performed the event processing before attempting to 'take' the semaphore again – entering the Blocked state if the semaphore was not immediately available.

This sequence is perfectly adequate if interrupts can only occur at a relatively low frequency. If another interrupt occurred before the Handler task had completed its processing of the first then the binary semaphore would effectively latch the event, allowing the Handler task to process the new event immediately after it has completed processing the original event. The handler task would not enter the Blocked state between processing the two events as the latched semaphore would be available immediately when xSemaphoreTake() is called. This scenario is shown in Figure 30.

Figure 30 demonstrates that a binary semaphore can latch at most one interrupt event. Any subsequent events occurring before the latched event has been processed would be lost. This scenario can be avoided by using a counting semaphore in place of the binary semaphore.

Just as binary semaphores can conceptually be thought of as queues that have a length of one, counting semaphores can be thought of as queues that have a length of more than one. Tasks are not interested in the data that is stored in the queue – just whether the queue is empty or not.

Each time a counting semaphore is 'given' another space in its queue is used. The number of items in the queue is the semaphores 'count' value.

The semaphore is not available...

Task
xSemaphoreTake()

...so the task is blocked waiting for the semaphore

Interrupt!
xSemaphoreGiveFromISR()

An interrupt occurs...that 'gives' the semaphore....

Task
xSemaphoreTake()

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.

Interrupt!
xSemaphoreGiveFromISR()

Task

Another interrupt occurs while the task is still processing the first event. The ISR 'gives' the semaphore again, effectively latching the event so the event is not lost.

The task is still processing the first event.

Task
xSemaphoreTake()

When processing of the original event completes the task calls xSemaphoreTake() again. Because another interrupt has already occurred the semaphore is already available so the task takes the semaphore without ever entering the Blocked state.

Figure 30 A binary semaphore can latch at most one event
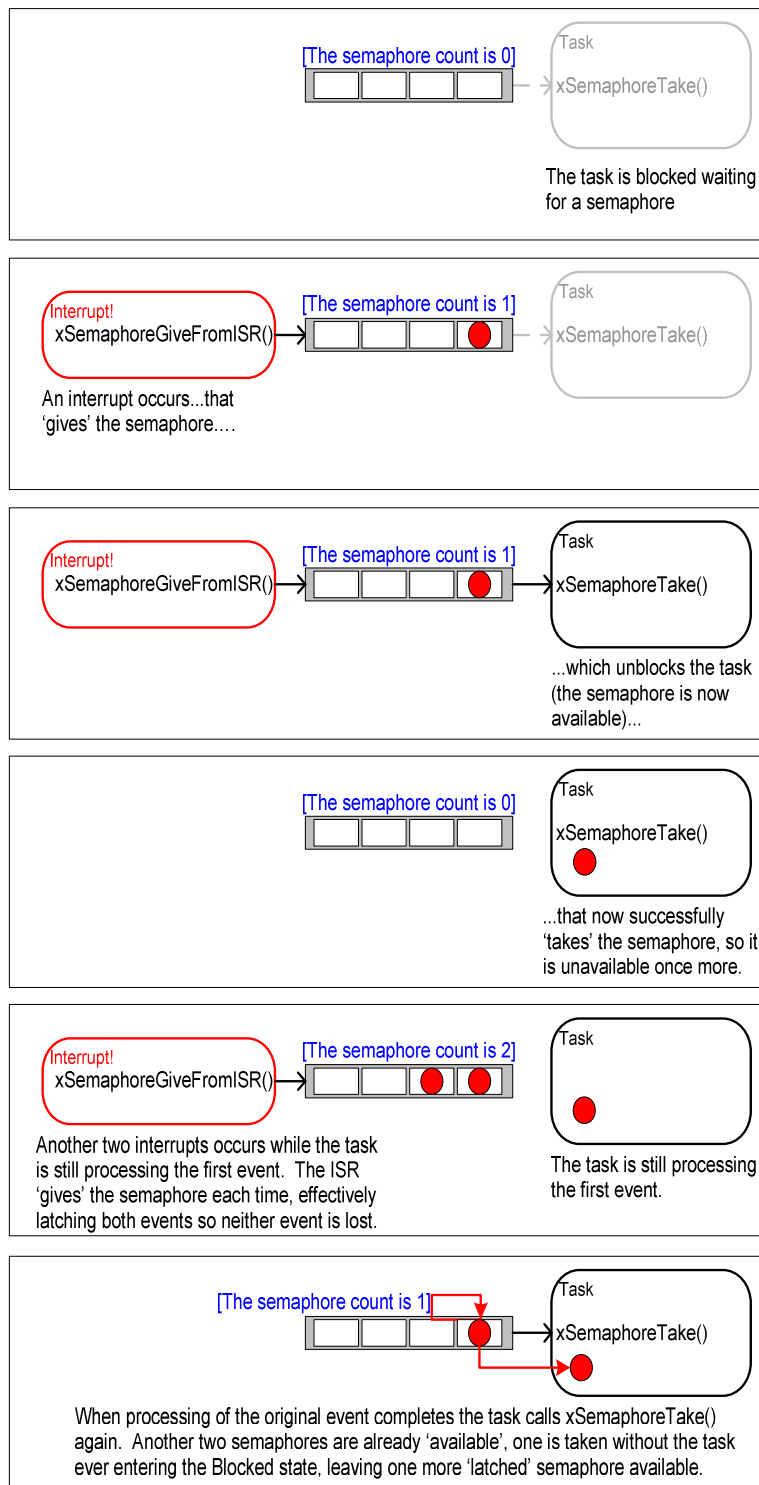
Figure 31 Using a counting semaphore to 'count' events

Counting semaphores are typically used for two things:

1. Counting events.

In this usage scenario an event handler will 'give' a semaphore each time an event occurs – causing the semaphores count value to be increment on each give. A handler task will 'take' a semaphore each time it processes an event – causing the semaphores count value to be decremented on each take. The count value is the difference between the number of events that have occurred and the number that have been processed. This mechanism is shown in Figure 31.

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

2. 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 semaphores 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 semaphores count value.

Counting semaphores that are used to manage resources are created so their initial count value equals the number of resources that are available. CHAPTER 4 covers using semaphores to manage resources.

xSemaphoreCreateCounting() API Function

Handles to all the various types of FreeRTOS semaphore are stored in a variable of type xSemaphoreHandle.

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

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

Listing 48 The xSemaphoreCreateCounting() API function prototype

Table 15 xSemaphoreCreateCounting() parameters and return value

Parameter Name / Returned Value	Description
uxMaxCount	The maximum value the semaphore will count to. To continue the queue analogy, the uxMaxCount value is effectively the length of the queue. When the semaphore is going to be used to count or latch events uxMaxCount is the maximum number of events that can be latched. When the semaphore is going to be used to manage access to a collection of resources uxMaxCount should be set to the total number of resources that are available.
uxInitialCount	The initial count value of the semaphore after it has been created. When the semaphore is going to be used to count or latch events uxInitialCount should be set to 0 – as presumably when the semaphore is created no events have yet occurred. When the semaphore is going to be used to manage access to a collection of resources uxInitialCount should be set to equal uxMaxCount – as presumably when the semaphore is created all the resources are available.
Returned value	If NULL is returned then the semaphore could not be created because there was insufficient heap memory available for FreeRTOS to allocate the semaphore data structures. CHAPTER 5 provides more information on memory management. A non-NULL value being returned indicates that the semaphore was created successfully. The returned value should be stored as the handle to the created semaphore.

Example 13. Using a Counting Semaphore to Synchronize a Task with an Interrupt

Example 13 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 vSemaphoreCreateBinary(). The new API call is shown in Listing 49.

```
/* 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 49 Using xSemaphoreCreateCounting() to create a counting semaphore

To simulate multiple events occurring at high frequency the interrupt service routine is changed to 'give' the semaphore more than once per interrupt. Each event is latched in the semaphores count value. The modified interrupt service routine is shown in Listing 50.

```
static void __interrupt __far vExampleInterruptHandler( void )
{
static portBASE_TYPE xHigherPriorityTaskWoken;

    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 );

    if( xHigherPriorityTaskWoken == pdTRUE )
    {
        /* Giving the semaphore unblocked a task, and the priority of the
        unblocked task is higher than the currently running task - force
        a context switch to ensure that the interrupt returns directly to
        the unblocked (higher priority) task.

        NOTE: The actual macro to use to force a context switch from an
        ISR is dependent on the port.  This is the correct macro for the
        Open Watcom DOS port.  Other ports may require different syntax.
        Refer to the examples provided for the port being used to determine
        the syntax required. */
        portSWITCH_CONTEXT();
    }
}
```

Listing 50 The implementation of the interrupt service routine used by Example 13

All the other functions remain unmodified from those used in Example 12.

The output produced when Example 13 is executed is shown in Figure 32. As can be seen, the Handler task processes all three [simulated] events each time an interrupt is generated. The events were latched into the count value of the semaphore, allowing the Handler task to process them in turn.

Figure 32 The output produced when Example 13 is executed

3.4   USING QUEUES WITHIN AN INTERRUPT SERVICE ROUTINE

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

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

xQueueSendToFrontFromISR() and xQueueSendToBackFromISR() API Functions

xQueueSendFromISR() is equivalent to and exactly the same as xQueueSendToBackFromISR().


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

Listing 51 The xQueueSendToFrontFromISR() API function prototype


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

Listing 52 The xQueueSendToBackFromISR() API function prototype

Table 16 xQueueSendToFrontFromISR() and xQueueSendToBackFromISR() parameters and return values

Parameter Name / Returned Value	Description
xQueue	The handle of the queue to which the data is being sent (written). The queue handle will have been returned from the call to xQueueCreate() used to create the queue.
pvItemToQueue	A pointer to the data that will be copied into the queue. The size of each item that the queue can hold is set when the queue is created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
pxHigherPriorityTaskWoken	It is possible that a single queue will have one or more tasks blocked on it waiting for data to become available. Calling xQueueSendToFrontFromISR() or xQueueSendToBackFromISR() can make data available, and so cause such a task to leave the Blocked state. If calling the API function causes a task to leave the Blocked state, and the unblocked task has a priority higher than the currently executing task (the task that was interrupted), then the API function will internally set *pxHigherPriorityTaskWoken to pdTRUE. If xQueueSendToFrontFromISR() or xQueueSendToBackFromISR() sets this value to pdTRUE then a context switch should be performed before the interrupt is exited. This ensure the interrupt returns directly to the highest priority Ready state task.
Returned value	There are two possible return values: 1. pdPASS pdPASS will only be returned if data was successfully sent to the queue. 2. errQUEUE_FULL errQUEUE_FULL will be returned if data could not be sent to the queue because the queue is already full.

Efficient Queue Usage

Most of the demo applications that are included in the FreeRTOS download include a simple UART driver that uses queues to pass characters into the transmit interrupt handler and to pass characters out of the receive interrupt handler. Every character that is transmitted or received gets individually

passed 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) so not recommended for production code. 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 after a complete message had been received, or a break in transmission had been detected.

- Interpret the received characters directly within the interrupt service routine, then use a queue to send the interpreted and decoded commands to a task for processing (in a similar manner to that shown by Figure 23). This technique is only suitable if interpreting the data stream is quick enough to be performed entirely from within an interrupt.

Example 14. Sending and Receiving on a Queue from Within an Interrupt

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

A periodic task is created that sends five numbers to a queue every 200 milliseconds. It generates a software interrupt only after all five values have been sent. The task implementation is shown in Listing 53.

```
static void vIntegerGenerator( void *pvParameters )
{
portTickType xLastExecutionTime;
unsigned portLONG ulValueToSend = 0;
int i;

    /* Initialize the variable used by the call to vTaskDelayUntil(). */
    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 );

        /* Send an incrementing number to the queue five times.  The values will
        be read from the queue by the interrupt service routine.  The interrupt
        service routine always empties the queue so this task is guaranteed to be
        able to write all five values, so a block time is not required. */
        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.\r\n" );
        __asm{ int 0x82 } /* This line generates the interrupt. */
        vPrintString( "Generator task - Interrupt generated.\r\n\r\n\r\n" );
    }
}
```

Listing 53 The implementation of the task that writes to the queue in Example 14

The interrupt service routine repeatedly calls xQueueReceiveFromISR() until all the values written to the queue by the periodic task have been removed and the queue is left empty. The last two bits of each received value are used as an index into an array of strings, with a pointer to the string at the corresponding index position being sent to a different queue using a call to xQueueSendFromISR(). The implementation of the interrupt service routine is shown in Listing 54.

```
static void __interrupt __far vExampleInterruptHandler( void )
{
static portBASE_TYPE xHigherPriorityTaskWoken;
static unsigned long ulReceivedNumber;

/* The strings are declared static const to ensure they are not allocated
on the stack of the ISR, and exist even when the ISR is not executing. */
static const char *pcStrings[] =
{
    "String 0\r\n",
    "String 1\r\n",
    "String 2\r\n",
    "String 3\r\n"
};

    xHigherPriorityTaskWoken = pdFALSE;

    /* Loop until the queue is empty. */
    while( xQueueReceiveFromISR( xIntegerQueue,
                                 &ulReceivedNumber,
                                 &xHigherPriorityTaskWoken ) != errQUEUE_EMPTY )
    {
        /* Truncate the received value to the last two bits (values 0 to 3 inc.), then
        send a pointer to the string that corresponds to the truncated value to a
        different queue. */
        ulReceivedNumber &= 0x03;
        xQueueSendToBackFromISR( xStringQueue,
                                 &pcStrings[ ulReceivedNumber ],
                                 &xHigherPriorityTaskWoken );
    }

    /* Did receiving on a queue or sending on a queue unblock a task that has a
    priority higher than the currently executing task?  If so, force a context
    switch here. */
    if( xHigherPriorityTaskWoken == pdTRUE )
    {
        /* NOTE: The actual macro to use to force a context switch from an ISR is
        dependent on the port.  This is the correct macro for the Open Watcom DOS
        port.  Other ports may require different syntax.  Refer to the examples
        provided for the port being used to determine the syntax required. */
        portSWITCH_CONTEXT();
    }
}
```

Listing 54 The implementation of the interrupt service routine used by Example 14

The task that receives the character pointers from the interrupt service routine simply blocks on the queue until a message arrives, printing out each string as it is received. Its implementation is shown in Listing 55.

```
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 );
    }
}
```

Listing 55 The task that prints out the strings received from the interrupt service routine in Example 14

As normal, main() creates the required queues and tasks before starting the scheduler. Its implementation is shown in Listing 56.

```
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 * ) );

    /* Install the interrupt handler. */
    _dos_setvect( 0x82, vExampleInterruptHandler );

    /* 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", 1000, 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", 1000, 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.
    CHAPTER 5 provides more information on memory management. */
    for( ;; );
}
```

Listing 56 The main() function for Example 14

The output produced when Example 14 is executed it shown in Figure 33. As can be seen, the interrupt is receiving all five integers and producing five strings in response. More explanation is given in Figure 34.

Figure 33 The output produced when Example 14 is executed
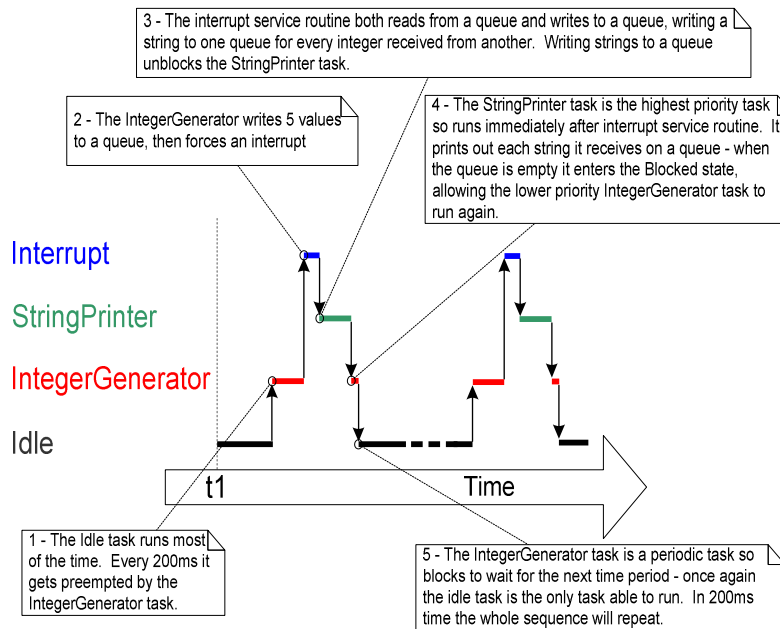
Figure 34 The sequence of execution produced by Example 14

3.5   INTERRUPT NESTING

The most recent FreeRTOS ports allow interrupts to nest. These ports require one or both of the constants detailed in Table 17 to be defined within FreeRTOSConfig.h.

Table 17 Constants that control interrupt nesting

Constant	Description
configKERNEL_INTERRUPT_PRIORITY	Sets the interrupt priority used by the tick interrupt. If the port does not use the configMAX_SYSCALL_INTERRUPT_PRIORITY constant then any interrupt that uses the interrupt safe FreeRTOS API functions must also execute at this priority.
configMAX_SYSCALL_INTERRUPT_PRIORITY	Sets the highest interrupt priority from which interrupt safe FreeRTOS API functions can be called.

A full interrupt nesting model is created by setting configMAX_SYSCALL_INTERRUPT_PRIORITY to a higher priority than configKERNEL_INTERRUPT_PRIORITY. This is demonstrated in Figure 35 which shows a *hypothetical* scenario where configMAX_SYSCALL_INTERRUPT_PRIORITY has been set to three and configKERNEL_INTERRUPT_PRIORITY has been set to one. An equally hypothetical microcontroller that has seven different interrupt priority levels is shown. The value seven is just an arbitrary number for this hypothetical example and is not intended to be representative of any particular microcontroller architecture.

It is common for confusion to arise between task priorities and interrupt priorities.   Figure 35 shows interrupt priorities, as defined by the microcontroller architecture. These are the hardware controlled priorities that interrupt service routines execute at relative to each other. Tasks do not run in interrupt service routines so the software priority assigned to a task is in no way related to the hardware priority assigned to an interrupt source.
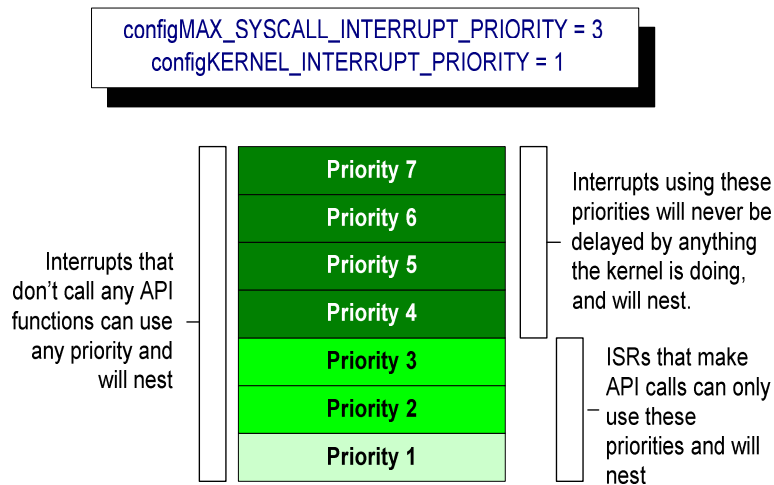
Figure 35 Constants affecting interrupt nesting behavior

Referring to Figure 35:

- Interrupts that use priorities 1 to 3 inclusive will be prevented from executing while the kernel or the application is within a critical section, but they can use the interrupt safe FreeRTOS API functions.

- Interrupts that use priority 4 or above are not affected by critical sections, so nothing the kernel does will prevent these interrupts from executing immediately – within the limitations of the microcontroller itself. 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 interrupts response time.

- Interrupts that do not make any FreeRTOS API calls are free to use any priority.

A Note to ARM Cortex M3 Users

The Cortex M3 use numerically low priority numbers to represent logically *high* priority interrupts. This can seem counter-intuitive and is easy to forget! If you wish to assign an interrupt a low priority then it must be assigned a high numeric value. Do not assign it a priority of 0 (or other low numeric value) as this can result in the interrupt actually having the highest priority in the system - and therefore potentially make your system crash if the priority is above configMAX_SYSCALL_INTERRUPT_PRIORITY.

The lowest priority on a Cortex M3 core is 255 although different Cortex M3 vendors implement a different number of priority bits and supply library functions that expect priorities to be specified in different ways. For example, on the STM32 the lowest priority that can be specified in an ST driver library call is 15 and the highest priority that can be specified is 0.

CHAPTER 4

RESOURCE MANAGEMENT

4.1   CHAPTER INTRODUCTION AND SCOPE

There is the potential for a conflict to arise in a multitasking system 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 left 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.

Following are some examples:

1. Accessing Peripherals

    Consider the following scenario where two tasks attempt to write to an LCD:

    - Task A executes and starts to write the string "Hello world" to the LCD.

    - Task A is pre-empted by Task B after outputting just the beginning of the string – "Hello w".

    - Task B writes "Abort, Retry, Fail?" to the LCD before entering the Blocked state.

    - Task A continues from the point at which it was pre-empted and completes outputting the remaining characters – "orld".

    The LCD will now be displaying the corrupted string "Hello wAbort, Retry, Fail?orld".

2. Read, Modify, Write Operations

    Listing 57 shows a line of C code and its resultant [ARM7] assembly output. It can be seen that the value of PORTA is first read from memory into a register, modified within the register, and then written back to memory. This is called a read, modify, write operation.

```
/* The C code being compiled. */
155:            PORTA |= 0x01;

/* The assembly code produced. */
0x00000264 481C LDR  R0,[PC,#0x0070]   ; Obtain the address of PORTA
0x00000266 6801 LDR  R1,[R0,#0x00]     ; Read the value of PORTA into R1
0x00000268 2201 MOV  R2,#0x01          ; Move the absolute constant 1 into R2
0x0000026A 4311 ORR  R1,R2             ; OR R1 (PORTA) with R2 (constant 1)
0x0000026C 6001 STR  R1,[R0,#0x00]     ; Store the new value back to PORTA
```

Listing 57 An example read, modify, write sequence

This is a 'non-atomic' operation because it takes more than one instruction to complete and can be interrupted. Consider the following scenario where two tasks attempt to update a memory mapped register called PORTA:

- Task A loads the value of PORTA into a register – the read portion of the operation.

- Task A is pre-empted by Task B before it completes the modify and write portions of the same operation.

- Task B updates the value of PORTA, then enters the Blocked state.

- Task A continues from the point at which it was pre-empted. It modifies the *copy* of the PORTA value that it already holds in a register before writing the updated value back to PORTA.

Task A updated and wrote back an out of date value for PORTA. Task B had modified PORTA between Task A taking a copy of the PORTA value and Task A writing its modified value back to the PORTA register. When Task A writes to PORTA it overwrites the modification that has already been performed by Task B, effectively corrupting the PORTA register value.

This example uses a peripheral register, but the same principal applies when performing read, modify, write operations on global variables.

3. Non-atomic Access to Variables

Updating multiple members of a structure or updating a variable that is larger than the natural word size of the architecture (for example, updating a 32 bit variable on a 16 bit machine) are both examples of non-atomic operations. If they are interrupted they can result in data loss or corruption.

4. 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.

Each task maintains its own stack and its own set of core register values. If a function does not access any data other than data that is allocated to the stack or held in a register then the function is reentrant. Listing 58 is an example of a reentrant function. Listing 59 is an example of a function that is not 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 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;
}
```

Listing 58 An example of a reentrant function

```
/* 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;
    }
}
```

Listing 59 An example of a function that is not reentrant

Mutual Exclusion

Access to a resource that is shared either between tasks or between tasks and interrupts needs to be managed using a 'mutual exclusion' technique to ensure 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 only accessed from a single task.

Scope

This chapter aims to give readers a good understanding of:

- When and why resource management and control is necessary.

- What a critical section is.

- What mutual exclusion means.

- What it means to suspend the scheduler.

- How to use a mutex.

- How to create and use a gatekeeper task.

- What priority inversion is and how priority inheritance can lessen (but not remove) its impact.

4.2 CRITICAL SECTIONS AND SUSPENDING THE SCHEDULER

Basic Critical Sections

Basic critical sections are regions of code that are surrounded by calls to the macros taskENTER_CRITICAL() and taskEXIT_CRITICAL() respectively, as demonstrated in Listing 60. Critical sections are also known as critical regions.

```
/* Ensure access to the PORTA register 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 on FreeRTOS ports that allow interrupt nesting, 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. */
PORTA |= 0x01;

/* We have finished accessing PORTA so can safely leave the critical
section. */
taskEXIT_CRITICAL();
```

Listing 60 Using a critical section to guard access to a register

The example projects that accompany this book use a function called vPrintString() to write strings to standard out – which is the terminal window for Open Watcom DOS executables. vPrintString() is called from many different tasks so in theory its implementation could protect access to standard out using a critical section as shown in Listing 61.

```
void vPrintString( const portCHAR *pcString )
{
    /* Write the string to stdout, using a critical section as a crude method
    of mutual exclusion. */
    taskENTER_CRITICAL();
    {
        printf( "%s", pcString );
        fflush( stdout );
    }
    taskEXIT_CRITICAL();

    /* Allow any key to stop the application running.  A real application that
    actually used the key value should protect access to the keyboard input too. */
    if( kbhit() )
    {
        vTaskEndScheduler();
    }
}
```

Listing 61 A possible implementation of vPrintString()

Critical sections implemented in this way are a very crude method of providing mutual exclusion. They work simply by disabling interrupts either completely, or up to the interrupt priority set by configMAX_SYSCALL_INTERRUPT_PRIORITY – depending on the FreeRTOS port being used. Preemptive context switches can only occur 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.

Critical sections must be kept very short otherwise they will adversely affect interrupt response times. Every call to taskENTER_CRITICAL() must be closely paired with a call to taskEXIT_CRITICAL(). For this reason standard out should not be protected using a critical section (as shown in Listing 61) because writing to the terminal can be a relatively long operation. Also the way the DOS emulator and Open Watcom handle terminal output is not compatible with this form of mutual exclusion as the library calls leave interrupts enabled. The examples in this chapter explore alternative solutions.

It is safe for critical sections to become nested because the kernel keeps a count of the nesting depth. The critical section will only be exited 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 (or Locking) the Scheduler

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

Basic critical sections protect a region of code from access by other tasks and by interrupts. A critical section implemented by suspending the scheduler only protects a region of code from access by other tasks because interrupts remain enabled.

A critical section that is too long to be implemented by simply disabling interrupts can instead be implemented by suspending the scheduler, but resuming (or 'un-suspending') the scheduler has the

potential to be a relatively lengthy operation so consideration as to which is the best method to use needs to be made in each case.

vTaskSuspendAll() API Function

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

The scheduler is suspended by calling vTaskSuspendAll(). Suspending the scheduler prevents a context switch from occurring but leaves interrupts enabled. If an interrupt requests a context switch while the scheduler is suspended then the request is held pending and is only performed when the scheduler is resumed (un-suspended).

FreeRTOS API functions should not be called while the scheduler is suspended.

xTaskResumeAll() API Function

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

Listing 63 The xTaskResumeAll() API function prototype

The scheduler is resumed (un-suspended) by calling xTaskResumeAll().

Table 18 xTaskResumeAll() return value

Returned Value	Description
Returned value	Context switches that were requested while the scheduler was suspended are held pending and only performed as the scheduler is being resumed. A previously pending context switch being performed before xTaskResumeAll() returns will result in the function returning pdTRUE. In all other cases xTaskResumeAll() will return pdFALSE.

It is safe for calls to vTaskSuspendAll() and xTaskResumeAll() to become nested because the kernel keeps a count of the nesting depth. The scheduler will only be resumed when the nesting depth returns to zero – which is when one call to xTaskResumeAll() has been executed for every preceding call to vTaskSuspendAll().

Listing 64 shows the actual implementation of vPrintString() which suspends the scheduler to protect access to the terminal output.

```
void vPrintString( const portCHAR *pcString )
{
    /* Write the string to stdout, suspending the scheduler as a method
    of mutual exclusion. */
    vTaskSuspendScheduler();
    {
        printf( "%s", pcString );
        fflush( stdout );
    }
    xTaskResumeScheduler();

    /* Allow any key to stop the application running.  A real application that
    actually used the key value should protect access to the keyboard input too. */
    if( kbhit() )
    {
        vTaskEndScheduler();
    }
}
```

Listing 64 The implementation of vPrintString()

4.3 MUTEXES (AND BINARY SEMAPHORES)

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 "<u>MUT</u>ual <u>EX</u>clusion".

When used in a mutual exclusion scenario the mutex can be conceptually thought of as a token that is associated with the resource being shared. For a task to legitimately access the resource 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. A task is not permitted to access the shared resource unless it holds the token. This mechanism is shown in Figure 36.

Even though mutexes and binary semaphores share many characteristics the scenario shown in Figure 36 (where a mutex is used for mutual exclusion) is completely different to that shown in Figure 30 (where a binary semaphore is used for synchronization). 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.

- A semaphore that is used for synchronization is normally discarded and not returned.
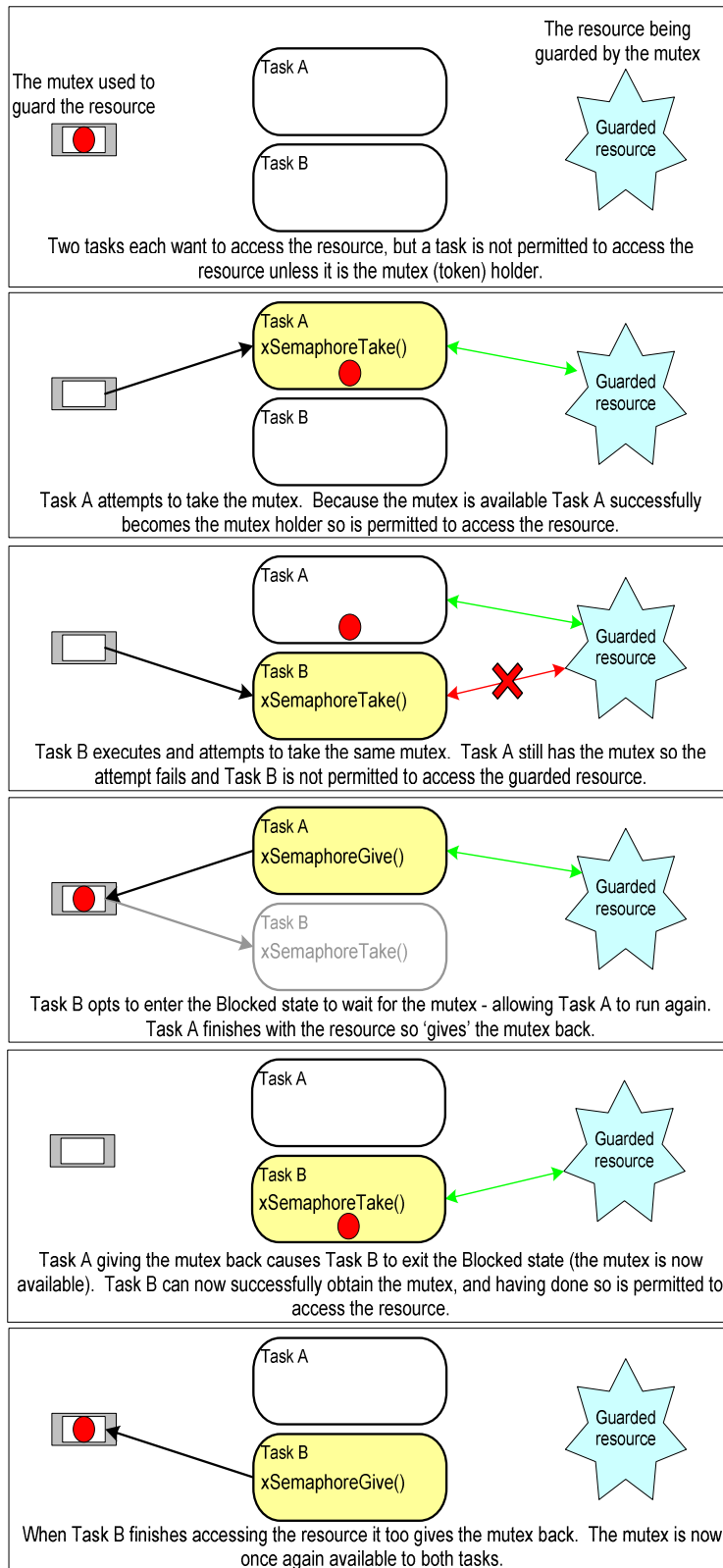
Figure 36 Mutual exclusion implemented using a mutex

The mechanism works purely through the discipline of the application writer. There is no reason why a task cannot access the resource at any time, but each task "agrees" not to unless they are first able to become the mutex holder.

xSemaphoreCreateMutex() API Function

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

Before a mutex can actually be used it must first be created. To create a mutex type semaphore use the xSemaphoreCreateMutex() API function.

```
xSemaphoreHandle xSemaphoreCreateMutex( void );
```

Listing 65 The xSemaphoreCreateMutex() API function prototype

Table 19 xSemaphoreCreateMutex() return value

Parameter Name / Returned Value	Description
Returned value	If NULL is returned then the mutex could not be created because there was insufficient heap memory available for FreeRTOS to allocate the mutex data structures. CHAPTER 5 provides more information on memory management.
	A non-NULL return value indicates that the mutex was created successfully. The returned value should be stored as the handle to the created mutex.

Example 15. Rewriting vPrintString() to Use a Semaphore

This example creates a new version of vPrintString() called prvNewPrintString(), then calls the new function from multiple tasks. prvNewPrintString() has identical functionality to vPrintString() but uses a mutex to control access to standard out in place of the basic critical section. The implementation of prvNewPrintString() is shown in Listing 66.

```
static void prvNewPrintString( const portCHAR *pcString )
{
    /* 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. */
        printf( "%s", pcString );
        fflush( stdout );

        /* The mutex MUST be given back! */
    }
    xSemaphoreGive( xMutex );

    /* Allow any key to stop the application running.  A real application that
    actually used the key value should protect access to the keyboard too.  A
    real application is very unlikely to have more than one task processing
    key presses though! */
    if( kbhit() )
    {
        vTaskEndScheduler();
    }
}
```

Listing 66 The implementation of prvNewPrintString()

prvNewPrintString() is repeatedly called by two instances of a task called prvPrintTask(). A random delay time is used between each call. The task parameter is used to pass a unique string into each instance of the task. The implementation of prvPrintTask() is shown in Listing 67.

```
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 ) );
    }
}
```

Listing 67 The implementation of prvPrintTask() for Example 15

As normal, main() simply creates the mutex, creates the tasks, then starts the scheduler. The implementation is shown in Listing 68.

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. As a mutex is used to ensure each task gets mutually exclusive access to the terminal even when one task is preemption by the other the strings that are displayed will be correct and in no way corrupted. The frequency of pre-emption can be increased by reducing the maximum time the tasks spend in the Blocked state, which is defaulted to 0x1ff ticks.

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

    /* Check the semaphore was created successfully before creating the tasks. */
    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", 1000,
                    "Task 1 ****************************************\r\n", 1, NULL );

        xTaskCreate( prvPrintTask, "Print2", 1000,
                    "Task 2 ----------------------------------------\r\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.
    CHAPTER 5 provides more information on memory management. */
    for( ;; );
}
```

Listing 68 The implementation of main() for Example 15

The output produced when Example 15 is executed is shown in Figure 37. A possible execution sequence is described in Figure 38.
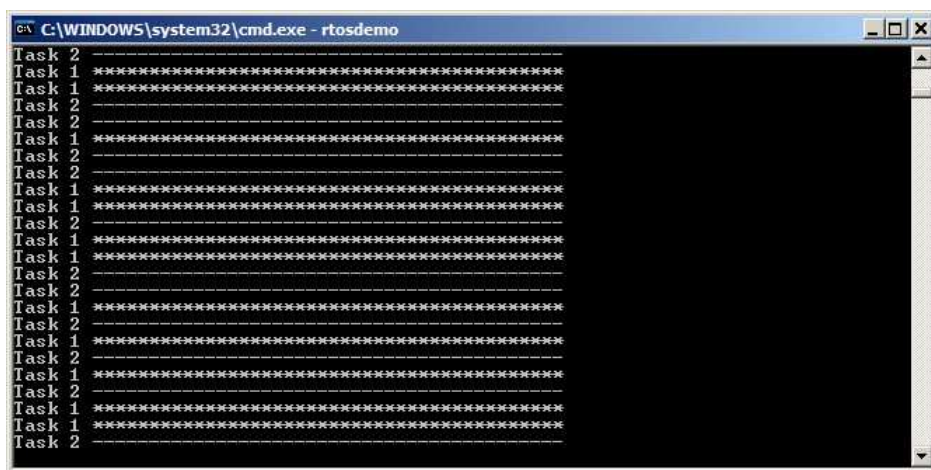

Figure 37 The output produced when Example 15 is executed

Figure 37 shows that, as expected, there is no corruption in the strings that are displayed on the terminal. The random ordering is a result of the random delay periods used by the tasks.
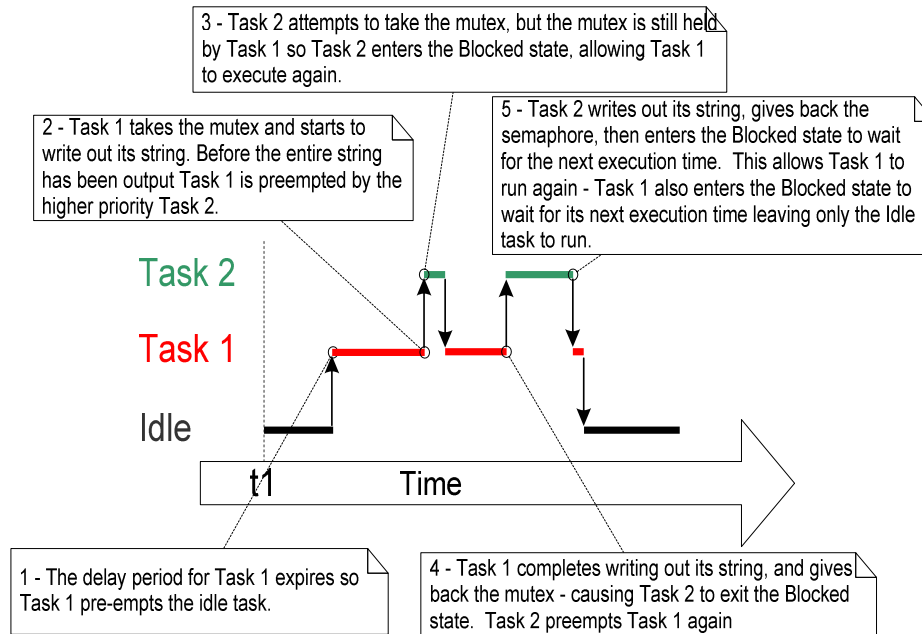
Figure 38 A possible sequence of execution for Example 15

Priority Inversion

Figure 38 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'. This undesirable behavior would be exaggerated further if a medium priority task started to execute while the high priority task was waiting for the semaphore – the result would be a high priority task waiting for a low priority task without the low priority task even being able to execute! This worst case scenario is shown in Figure 39.
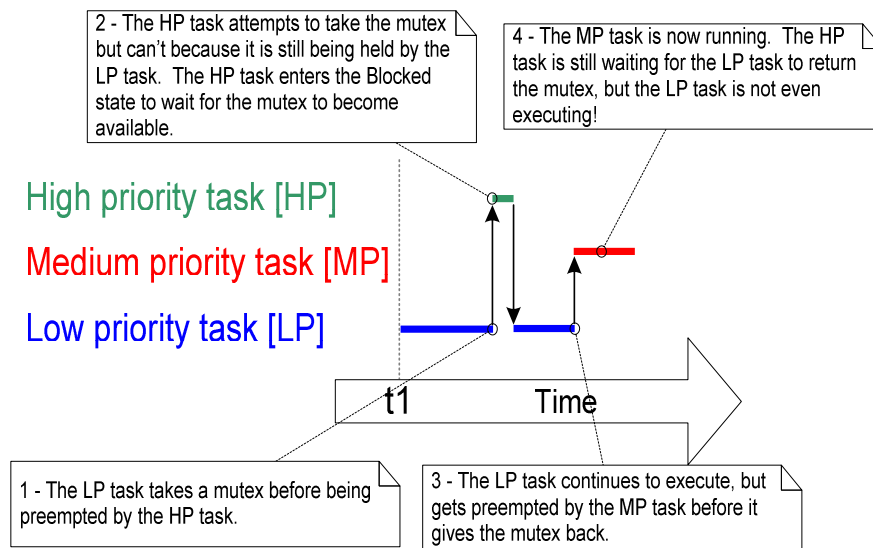
Figure 39 A worst case priority inversion scenario

Priority inversion can be a significant problem but in a small embedded system it can often be avoided by considering how resources are accessed at system design time.

Priority Inheritance

FreeRTOS mutexes and binary semaphores are very similar – the only difference being mutexes automatically provide a basic 'priority inheritance' mechanism. Priority inheritance is a scheme that minimizes the negative effects of priority inversion – it does not 'fix' priority inversion, it merely lessens its impact. Priority inheritance makes mathematical analysis of system behavior a more complex exercise so reliance on priority inheritance is not recommended if it can be avoided.

Priority inheritance works by temporarily raising the priority of the mutex holder to that of the highest priority task that is attempting to obtain the same mutex. The low priority task that holds the mutex 'inherits' the priority of the task waiting for the mutex. This is demonstrated by Figure 40. The priority of the mutex holder is automatically reset back to its original value when it gives the mutex back.

2 - The HP task attempts to take the mutex but can't because it is still being held by the LP task. The HP task enters the Blocked state to wait for the mutex to become available.

4 - The LP task returning the mutex causes the HP task to exit the Blocked state as the mutex holder. When the HP task has finished with the mutex it gives it back. The MP task only executes when the HP task returns to the Blocked state so the MP task never holds up the HP task.

High priority task [HP]

Medium priority task [MP]

Low priority task [LP]

t1   Time

1 - The LP task takes a mutex before being preempted by the HP task.

3 - The LP task is preventing the HP task from executing so inherits the priority of the HP task. The LP task cannot now be preempted by the MP task, so the amount of time that priority inversion exists is minimized. When the LP task gives the mutex back it returns to its original priority.

Figure 40 Priority inheritance minimizing the effect of priority inversion

Because the preference is to avoid priority inversion in the first place, and because FreeRTOS is targeted at memory constrained microcontrollers, the priority inheritance mechanism implemented by mutexes is only a basic form that assumes a task will only hold a single mutex at any one time.

Deadlock (or Deadly Embrace)

'Deadlock' is another potential pitfall of using mutexes for mutual exclusion. Deadlock is sometimes also known by the more dramatic name of 'deadly embrace'.

Deadlock occurs when two tasks cannot proceed because they are both waiting for a resource that is held by the other. Consider the following scenario where Task A and Task B both need to acquire mutex X *and* mutex Y in order to perform an action:

1. Task A executes and successfully takes mutex X.

2. Task A is pre-empted by Task B.

3. 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.

4. 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.

At the end of this scenario Task A is waiting for a mutex held by Task B, and Task B is waiting for a mutex held by Task A. Deadlock has occurred because neither task can proceed any further.

As with priority inversion, the best method of avoiding deadlock is to consider its potential at design time, and design the system so that it simply cannot occur. In practice deadlock is not a big problem for small embedded systems because the system designers can have a good understanding of the entire application and so identify and remove the areas where it could occur.

4.4   GATEKEEPER TASKS

Gatekeeper tasks provide a clean method of implementing mutual exclusion without the worry 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 needing to access the resource can only do so indirectly by using the services of the gatekeeper.

Example 16. Re-writing vPrintString() to Use a Gatekeeper Task

Example 16 also 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 need 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. When a message arrives the gatekeeper simply writes the message to standard out before returning to the Blocked sate to wait for the next message. The implementation of the gatekeeper task is shown by Listing 70.

Interrupts can send to queues so interrupt service routines can also safely use the services of the gatekeeper to write messages to the terminal. In this example a tick hook function is used to write out a message every 200 ticks.

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

- Set configUSE_TICK_HOOK to 1 in FreeRTOSConfig.h.

- Provide the implementation of the hook function, using the exact function name and prototype shown in Listing 69.


```
void vApplicationTickHook( void );
```

Listing 69 The name and prototype for a tick hook function

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

```
static void prvStdioGatekeeperTask( void *pvParameters )
{
char *pcMessageToPrint;

    /* 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. */
        printf( "%s", pcMessageToPrint );
        fflush( stdout );

        /* Now simply go back to wait for the next message. */
    }
}
```

Listing 70 The gatekeeper task

The task that prints out the message is similar to that used in Example 15, except this time the string is sent on the queue to the gatekeeper task rather than written out directly. The implementation is shown in Listing 71. As before, two separate instances of the task are created, each of which prints out a unique string passed to it via the task parameter.

```
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 ) );
    }
}
```

Listing 71 The print task implementation for Example 16

The tick hook function simply counts the number of times it is called, sending its message to the gatekeeper task each time the count reaches 200. Just for demonstration purposes the tick hook writes to the front of the queue and the print tasks write to the back of the queue. The tick hook implementation is shown in Listing 72.

```
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;
    }
}
```

Listing 72 The tick hook implementation

As normal, main() creates the queues and tasks necessary to run the example, then starts the scheduler. The implementation of main() is shown in Listing 73.

```
/* Define the strings that the tasks and interrupt will print out via the
gatekeeper. */
static char *pcStringsToPrint[] =
{
    "Task 1 ************************************************\r\n",
    "Task 2 -------------------------------------------------\r\n",
    "Message printed from the tick hook interrupt ############\r\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;

/*-----------------------------------------------------------*/

int main( void )
{
    /* Before a queue is used it must be explicitly created.  The queue is 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", 1000, ( void * ) 0, 1, NULL );
        xTaskCreate( prvPrintTask, "Print2", 1000, ( void * ) 1, 2, NULL );

        /* Create the gatekeeper task.  This is the only task that is permitted
        to directly access standard out. */
        xTaskCreate( prvStdioGatekeeperTask, "Gatekeeper", 1000, 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( ;; );
}
```

Listing 73 The implementation of main() for Example 16

The output produced when Example 16 is executed is shown in Figure 41. As can be seen, the strings originating from the tasks and the strings originating from the interrupt all print out correctly with no corruption.
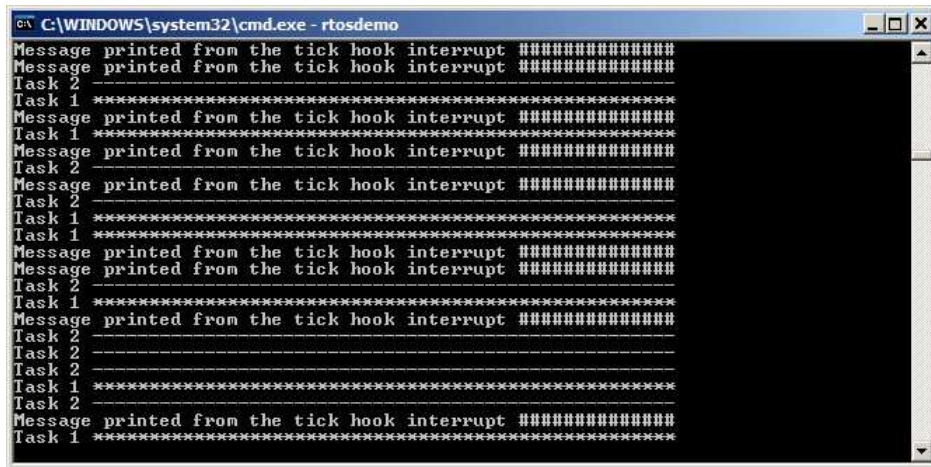

Figure 41 The output produced when Example 16 is executed

The gatekeeper task was assigned a lower priority than the print tasks – so messages sent to the gatekeeper remained in the queue until both print tasks were in the Blocked state. In some situations it would be appropriate to assign the gatekeeper a higher priority so 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.

CHAPTER 5

MEMORY MANAGEMENT

Designed For Microcontrollers;

5.1 CHAPTER INTRODUCTION AND SCOPE

The kernel has to dynamically allocate RAM each time a task, queue or semaphore is created. The standard malloc() and free() library functions can be used but can also suffer from one or more of the following problems:

1. They are not always available on small embedded systems.

2. Their implementation can be relatively large so take up valuable code space.

3. They are rarely thread safe.

4. They are not deterministic. The amount of time taken to execute the functions will differ from call to call.

5. They can suffer from memory fragmentation.

6. They can complicate the linker configuration.

Different embedded systems have varying RAM allocation and timing requirements so a single RAM allocation algorithm will only ever be appropriate for a subset of applications. FreeRTOS therefore treats memory allocation as part of the portable layer (as opposed to part of the core code base). This enables individual applications to provide their own specific implementation when appropriate.

When the kernel requires RAM, instead of calling malloc() directly it instead calls pvPortMalloc(). When RAM is being freed, instead of calling free() directly the kernel instead calls vPortFree(). pvPortMalloc() has the same prototype as malloc(), and vPortFree() has the same prototype as free().

FreeRTOS comes with three example implementations of both pvPortMalloc() and vPortFree(), all of which are documented in this chapter. Users of FreeRTOS can use one of the example implementations, or provide their own.

The three examples are defined in the files heap_1.c, heap_2.c and heap_3.c respectively – all of which are located in the FreeRTOS\Source\Portable\MemMang directory. The original memory pool and block allocation scheme used by very early versions of FreeRTOS has been removed because of the effort and understanding required to dimension the blocks and pools.

It is common for small embedded systems to only create tasks, queues and semaphores before the scheduler is started. When this is the case memory is only every dynamically allocated before the application starts to perform any true real time functionality, and once allocated memory is never again freed. This means the chosen allocation scheme does not need to consider any of the more complex issues such as determinism and fragmentation, and can instead only consider attributes such as code size and simplicity.

Scope

This chapter aims to give readers a good understanding of:

- When FreeRTOS allocates RAM.

- The three example memory allocation schemes supplied with FreeRTOS.

5.2   EXAMPLE MEMORY ALLOCATION SCHEMES

Heap_1.c

Heap_1.c implements a very basic version of pvPortMalloc() and does not implement vPortFree(). Any application that never deletes a task, queue or semaphore has the potential to use heap_1. Heap_1 is always deterministic.

The allocation scheme simply subdivides a simple array into smaller blocks as calls to pvPortMalloc() are made. The array is the FreeRTOS heap.

The total size (in bytes) of the array is set by the definition configTOTAL_HEAP_SIZE within FreeRTOSConfig.h.   Defining a large array in this manner can make the application appear to consume a lot of RAM – even before any of the array has actually been assigned.

Each created task requires a task control block (TCB) and a stack to be allocated from the heap. Figure 42 demonstrates how heap_1 subdivides the simple array as tasks are created. Referring to Figure 42:

- A shows the array before any tasks have been created, the entire array is free.

- B shows the array after one task has been created.

- C shows the array after three tasks have been created.
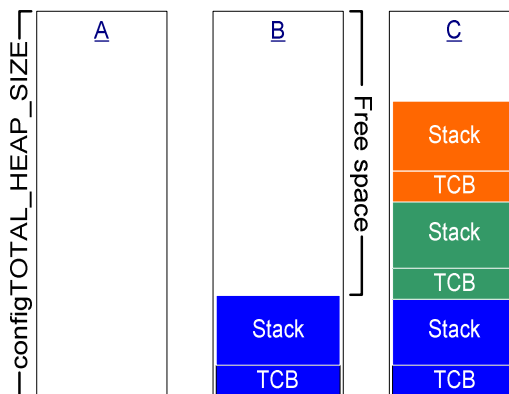
Figure 42 RAM being allocated within the array each time a task is created

Heap_2.c

Heap_2.c also uses a simple array dimensioned by configTOTAL_HEAP_SIZE. It uses a best fit algorithm to allocate memory and unlike heap_1 it does allow memory to be freed. Again the array is statically declared so will make the application appear to consume a lot of RAM even before any of the array has actually been assigned.

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

1. The heap contains three blocks of free memory that are 5 bytes, 25 bytes, and 100 bytes big respectively.

2. pvPortMalloc() is called to request 20 bytes of RAM.

The smallest free block of RAM into which the requested number of bytes will fit is the 25 byte block – so pvPortMalloc() splits the 25 byte block into one block of 20 bytes and one block of 5 bytes[3] before returning a pointer to the 20 byte block. The new 5 bytes block remains available to future calls to pvPortMalloc().

Heap_2.c does not combine adjacent free blocks into a single larger block so can suffer from fragmentation – however fragmentation will not be an issue if the blocks being allocated and subsequently freed are always the same size. Heap_2.c is suitable for an application that repeatedly creates and deletes tasks provided the size of the stack allocated to the created tasks does not change.
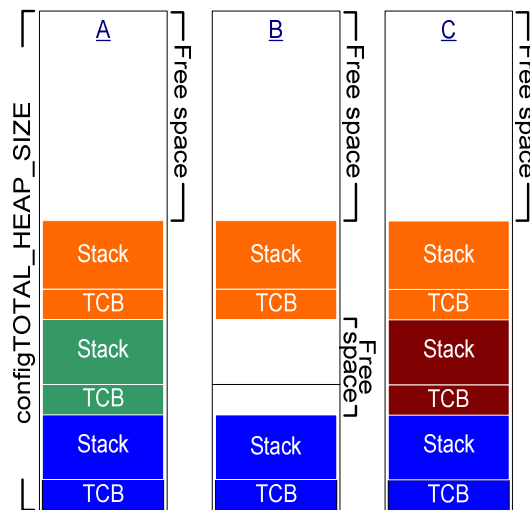
Figure 43 RAM being allocated from the array as tasks are created and deleted

Figure 43 demonstrates how the best fit algorithm works when a task is created, deleted, and then created again. Referring to Figure 43:

- A shows the array after three tasks have been created. A large free block remains at the top of the array.

---

[3] This is an over simplification because heap_2 stores information on the block sizes within the heap area, so the sum of the two split blocks will actually be less than 25.

- <u>B</u> shows the array after one of the tasks has been deleted. The large free block at the top of the array remains. There are now also two smaller free blocks that were previously allocated to the TCB and stack of the deleted task.

- <u>C</u> shows the situation after another task has been created. Creating the task resulted in two calls to pvPortMalloc(), one to allocate a new TCB and one to allocate the task stack (the calls to pvPortMalloc() occur internally within the xTaskCreate() API function).

  Every TCB is exactly the same size, so the best fit algorithm ensured that the block of RAM that had previously been allocated to the TCB of the deleted task is re-used to allocate the TCB of the new task.

  The size of the stack allocated to the newly created task is identical to that allocated to the previously deleted task, so the best fit algorithm ensured that the block of RAM that had previously been allocated to the stack of the deleted task is re-used to allocate the stack of the new task.

  The larger unallocated block at the top of the array remains untouched.

Heap_2.c is not deterministic but is more efficient than most standard library implementations of malloc() and free().

Heap_3.c

Heap_3.c simply uses the standard library malloc() and free() function but makes the calls thread safe by temporarily suspending the scheduler. The implementation is shown in Listing 74.

The size of the heap is not affected by configTOTAL_HEAP_SIZE and is instead defined by the linker configuration.

```
void *pvPortMalloc( size_t xWantedSize )
{
void *pvReturn;

    vTaskSuspendAll();
    {
        pvReturn = malloc( xWantedSize );
    }
    xTaskResumeAll();

    return pvReturn;
}

void vPortFree( void *pv )
{
    if( pv != NULL )
    {
        vTaskSuspendAll();
        {
            free( pv );
        }
        xTaskResumeAll();
    }
}
```

Listing 74 The heap_3.c implementation

CHAPTER 6

TROUBLE SHOOTING

6.1   CHAPTER INTRODUCTION AND SCOPE

This chapter aims to highlight the most common issues encountered by users who are new to FreeRTOS. It focuses mainly on stack overflow and stack overflow detection because stack issues have proven to be the most frequent source of support requests over the years. It then briefly and in an FAQ style touches on other common errors, their possible cause, and their solutions.

printf-stdarg.c

Stack usage can get particularly high when standard C library functions are used, especially IO and string handling functions such as sprintf(). The FreeRTOS download includes a file called printf-stdarg.c that contains a minimal and stack efficient version of sprintf() that can be used in place of the standard library version. In most cases this will permit a much smaller stack to be allocated to each task that calls sprintf() and related functions.

Printf-stdarg.c is open source but owned by a third party so it is licensed separately from FreeRTOS. The license terms are contained at the top of the source file.

6.2   STACK OVERFLOW

FreeRTOS provides several features to assist trapping and debugging stack related issues[4].

uxTaskGetStackHighWaterMark() API Function

Each task maintains its own stack, the total size of which is specified when the task is created. uxTaskGetStackHighWaterMark() is used to query how near a task has come to overflowing the stack space allocated to it. This value is called the stack 'high water mark'.

```
unsigned portBASE_TYPE uxTaskGetStackHighWaterMark( xTaskHandle xTask );
```

Listing 75 The uxTaskGetStackHighWaterMark() API function prototype

Table 20 uxTaskGetStackHighWaterMark() parameters and return value

Parameter Name/Returned Value	Description
xTask	The handle of the task whose stack high water mark is being queried (the subject task) – see the pxCreatedTask parameter of the xTaskCreate() API function for information on obtaining handles to tasks. A task can query its own stack high water mark by passing NULL in place of a valid task handle.
Returned value	The amount of stack the task is actually using will grow and shrink as the task executes and interrupts are processed. uxTaskGetStackHighWaterMark() returns the minimum amount of remaining stack space that was available since the task started executing. This is the amount of stack that remained unused when the stack usage was at its greatest (deepest) value. The closer the high water mark is to 0 the closer the task has come to overflowing its stack.

---

[4] Unfortunately these features cannot be used in a simulated DOS environment because DOS uses segmented memory. It is therefore not possible to include an Open Watcom example to demonstrate their use.

Run Time Stack Checking - Overview

FreeRTOS includes two optional run time stack checking mechanisms. These are controlled by the configCHECK_FOR_STACK_OVERFLOW compile time configuration constant within FreeRTOSConfig.h. Both methods will increase the time it takes to perform a context switch.

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

- Set configCHECK_FOR_STACK_OVERFLOW to either 1 or 2 in FreeRTOSConfig.h.

- Provide the implementation of the hook function, using the exact function name and prototype shown in Listing 76.

```
void vApplicationStackOverflowHook( xTaskHandle *pxTask, signed portCHAR *pcTaskName );
```

Listing 76 The stack overflow hook function prototype

The stack overflow hook is provided to make trapping and debugging stack errors easier but there is no real way of recovering from a stack overflow once it has occurred. The parameters pass the handle and name of the task that has overflowed its stack into the hook function, although it is possible the overflow will have corrupted the task name.

The stack overflow hook can get called from the context of an interrupt.

Some microcontrollers will generate a fault exception when they detect an incorrect memory access and it is possible that a fault will be triggered before the kernel has a chance to call the overflow hook function.

Run Time Stack Checking - Method 1

Method 1 is selected when configCHECK_FOR_STACK_OVERFLOW is set to 1.

A tasks entire execution context is saved onto its stack each time it gets swapped out. It is likely that this will be the time the stack usage reaches its peak. When configCHECK_FOR_STACK_OVERFLOW is set to 1 the kernel will check 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 of its valid range.

Method 1 is quick to execute but can miss stack overflows that occur between context saves.

Run Time Stack Checking - Method 2

Method 2 performs additional checks to those already described for method 1. It is selected when configCHECK_FOR_STACK_OVERFLOW is set to 2.

When a task is created its stack is filled with a known pattern. Method 2 walks the last valid 20 bytes of the task stack space to check 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 value.

Method 2 is not as quick to execute as method 1 but is still relatively fast as only 20 bytes are tested. It is very likely to catch all stack overflows, although it is conceivable (but highly improbably) that some could still be missed.

6.3 OTHER COMMON SOURCES OF ERROR

Symptom: 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 or semaphores to be added.

The idle task is automatically created when vTaskStartScheduler() is called. vTaskStartScheduler() will only return if there is not enough heap memory remaining for the idle task to be created. Including a null loop [for(;;);] after the call to vTaskStartScheduler() can make this error easier to debug.

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

Symptom: 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()".

Symptom: Sometimes the Application Crashes within an Interrupt Service Routine

The first thing to check is that the interrupt is not causing a stack overflow.

The way interrupts are defined and used differs between ports and between compilers – so 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 for the demo, and exactly as demonstrated by other interrupt service routines in the demo.

If the application is running on a Cortex M3 then ensure the priority assigned to each interrupt takes into account that numerically low priority numbers are used to represent logically *high* priority interrupts. This can seem counter-intuitive. It is a common mistake to accidentally assign an interrupt that uses a FreeRTOS API function a priority that is above that defined by configMAX_SYSCALL_INTERRUPT_PRIORITY.

Symptom: The Scheduler Crashes When Attempting to Start the First Task

If an ARM7 microcontroller is being used then ensure the processor is in Supervisor mode before vTaskStartScheduler() is called. The easiest way to achieve this is to place the processor into Supervisor mode within the C startup code before main() is called. This is how the ARM7 demo applications are configured.

The scheduler will not start unless the processor is in Supervisor mode.

Symptom: Critical Sections Do Not Nest Correctly

Do not alter the microcontroller interrupt enable bits or priority flags using any method other than calls to taskENTER_CRITICAL() and taskEXIT_CRITICAL(). These macros keep a count of the call nesting depth to ensure interrupts only become enabled again when the call nesting has unwound completely to zero.

Symptom: 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 is true of any interrupt service routine that attempts to send to or receive from a queue or semaphore. A context switch cannot occur until after the scheduler has started.

Many API functions cannot be called prior to the scheduler being started. It is best to restrict API usage to the creation of tasks, queues and semaphores until after vTaskStartScheduler() has been called.

Symptom: Calling API Functions While the Scheduler is Suspended Causes the Application to Crash

The scheduler is suspended by calling vTaskSuspendAll() and resumed (unsuspended) by calling xTaskResumeAll().

Do not call API functions while the scheduler is suspended.

Symptom: The Prototype For pxPortInitialiseStack() Causes Compilation to Fail

Every port requires a macro to be defined that ensures the correct kernel header files are included in the build. An error when compiling the pxPortInitialiseStack() prototype is almost definitely a symptom of this macro being set incorrectly for the port being used. See APPENDIX 4: for more information.

Base new applications on the provided demo project associated with the port being used. This way the correct files will be included and the correct compiler options will be set.

APPENDIX 1:   BUILDING THE EXAMPLES

This book presents numerous examples – the source code for which is provided in an accompanying .zip file along with project files that can be opened and built from within the free Open Watcom IDE. The resultant executables can then be executed either within a Windows command terminal or alternatively under the free DOSBox DOS emulator.   See http://www.openwatcom.org and http://www.dosbox.com for tool downloads.

Ensure to include the 16bit DOS target options when installing the Open Watcom compiler!

The Open Watcom project files are all called RTOSDemo.wpj and can be located in the Examples\Example0nn directories, where 'nn' is the example number.

DOS is far from an ideal target for FreeRTOS and the example applications will not run with true real time characteristics. DOS is used simply because it allows users to experiment with the examples without first having to invest in specialist hardware or tools.

Please note the Open Watcom debugger will allow interrupts to execute between step operations – even when stepping through code that is within a critical section. This unfortunately makes it impossible to step through the context switch process.

It is best to run the generated executables from a command prompt rather than from within the Open Watcom IDE.

APPENDIX 2:   THE DEMO APPLICATIONS

Each official FreeRTOS port comes with a demo application that should build without any errors or warnings being generated[5]. The demo application has several purposes:

1. To provide an example of a working and pre-configured project with the correct files included and the correct compiler options set.

2. To allow 'out of the box' experimentation with minimal setup or prior knowledge.

3. To demonstrate the FreeRTOS API.

4. As a base from which real applications can be created.

Each demo project is located in a unique directory under the Demo directory (see APPENDIX 3:). The directory name will indicate the port that the demo project relates to.

Every demo application also comes with a documentation page that is hosted on the FreeRTOS.org WEB site. The documentation page includes information on locating individual demo applications in FreeRTOS directory structure.

All the demo projects create tasks that are defined in source files that are located in the Demo\Common directory tree. Most use files from the Demo\Common\Minimal directory.

A file called main.c is included in each project. This contains the main() function, from where all the demo application tasks are created. See the comments within the individual main.c files for more information on what a specific demo application does.

---

[5] This is the ideal scenario, and is normally the case, but is dependent on the version of the compiler being used to build the demo. Upgraded compilers can sometimes generate warnings where their predecessors didn't.
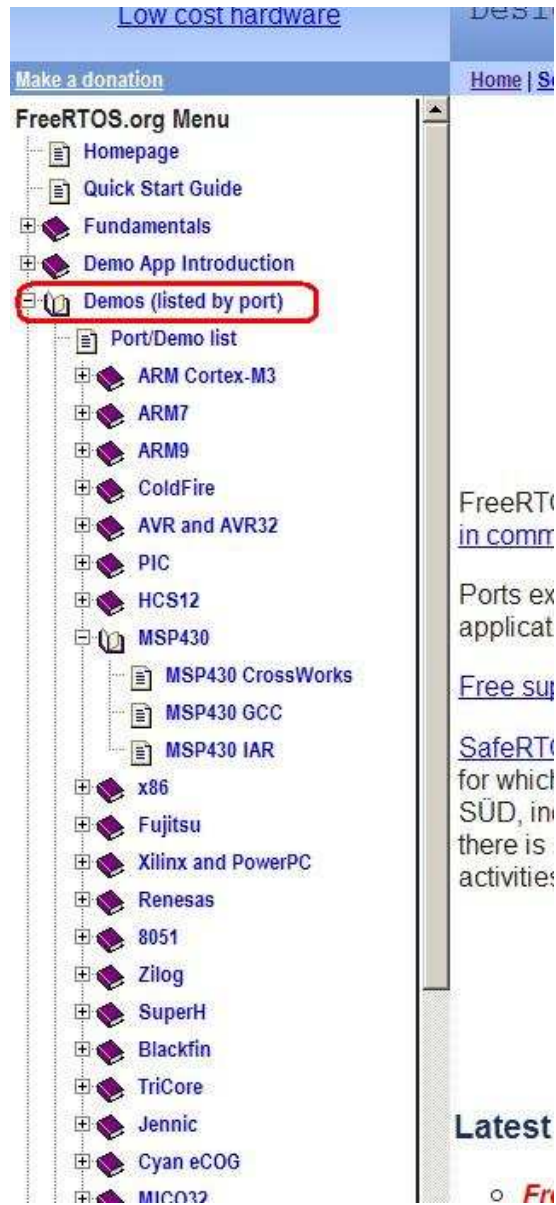
Figure 44 Locating the demo application documentation in the menu frame of the FreeRTOS.org WEB site

APPENDIX 3:   FREERTOS FILES AND DIRECTORIES

The directory structure described in this appendix relates only to the .zip file that can be downloaded from the FreeRTOS.org WEB site. The examples that come with this book use a slightly different organization.

FreeRTOS is downloaded as a single .zip file that contains:

- The core FreeRTOS source code. This is the code that is common to all ports.

- A port layer for each microcontroller and compiler combination that is supported.

- A project file or makefile to build a demo application for each microcontroller and compiler combination that is supported.

- A set of demo tasks that are common to each demo application. These demo tasks are referenced from the port specific demo projects.

The .zip file has two top level directories, one called Source and the other called Demo. The Source directory tree contains the entire FreeRTOS kernel implementation – both the common components and the port specific components. The Demo directory tree contains just the demo application project files and the source files that define the demo tasks.

```
FreeRTOS
    ¦
    +-Demo        Contains the demo application source and projects.
    ¦
    +-Source      Contains the implementation of the real time kernel.
```

Figure 45 The top level directories – Source and Demo

The core FreeRTOS source code is contained in just three C files that are common to all the microcontroller ports. These are called queue.c, tasks.c and list.c, and can be located directly under the Source directory. The port specific files are located within the Portable directory tree, which is also located directly within the Source directory.

A fourth optional source file called croutine.c implements the FreeRTOS co-routine functionality. It only needs to be included in the build if co-routines are actually going to be used.

```
FreeRTOS
    ¦
    +-Demo          Contains the demo application source and projects.
    ¦
    +-Source        Contains the implementation of the real time kernel.
        |
        +-tasks.c  One of the three core kernel files.
        +-queue.c  One of the three core kernel files.
        +-list.c   One of the three core kernel files.
        +-portable The sub-directory that contains all the port specific files.
```

Figure 46 The three core files that implement the FreeRTOS kernel

Removing Unused Files

The main FreeRTOS .zip file includes the files for all the ports and all the demo applications so contains many more files than are required to use any one port. The demo application project or makefile that accompanies the port being used can be used as a reference to which files are required and which can be deleted.

The 'portable layer' is the code that tailors the FreeRTOS kernel to a particular compiler and architecture. The portable layer source files are located within the FreeRTOS\Source\Portable\[compiler]\[architecture] directory, where [compiler] is the tool chain being used and [architecture] is the microcontroller variant being used.

- All the sub-directories under FreeRTOS\Source\Portable that do not relate to the tool chain being used can be deleted **except** the directory FreeRTOS\Source\Portable\MemMang.

- All the sub-directories under FreeRTOS\Source\Portable\[compiler] that do not relate to the microcontroller variant being used can be deleted.

- All the sub-directories under FreeRTOS\Demo that do not relate to the demo application being used can be deleted **except** the FreeRTOS\Demo\Common directory, which contains files that are referenced from all the demo applications.

  FreeRTOS\Demo\Common contains many more files than are referenced from any one demo application so this directory can also be thinned out if desired.

APPENDIX 4: CREATING A FREERTOS PROJECT

Adapting One of the Supplied Demo Projects

Every official FreeRTOS port comes with a pre-configured demo application that should build without any errors or warnings (see APPENDIX 2:). It is recommended that new projects are created by adapting one of these existing projects. This way the project will include the correct files and have the correct compiler options set.

To start a new application from an existing demo project:

1. Open up the supplied demo project and ensure it builds and executes as expected.

2. Strip out the source files that define the demo tasks. Any file that is located within the Demo\Common directory tree can be removed from the project file or makefile.

3. Delete all the functions within main.c other than prvSetupHardware().

4. Ensure configUSE_IDLE_HOOK, configUSE_TICK_HOOK and configCHECK_FOR_STACK_OVERFLOW are all set to 0 within FreeRTOSConfig.h. This will prevent the linker looking for any hook functions. Hook functions can be added later if required.

5. Create a new main() function from the template shown in Listing 77.

6. Check that the project still builds.

Following these steps will provide a project that includes the FreeRTOS source files but does not define any functionality.

```
int main( void )
{
    /* Perform any hardware setup necessary. */
    prvSetupHardware();

    /* --- APPLICATION TASKS CAN BE CREATED HERE --- */

    /* Start the created tasks running. */
    vTaskStartScheduler();

    /* Execution will only reach here if there was insufficient heap to
    start the scheduler. */
    for( ;; );
    return 0;
}
```

Listing 77 The template for a new main() function

Creating a New Project from Scratch

As just mentioned, it is recommended that new projects are created from the existing demo projects. If for some reason this is not desirable then a new project can be created by using the following steps:

1. Create a new empty project file or makefile using your chosen tool chain.

2. Add the files detailed within Table 21 to the newly created project or makefile.

3. Copy an existing FreeRTOSConfig.h file into the project directory.

4. Add both the project directory and FreeRTOS\Source\include to the path the project will search to locate header files.

5. Copy the compiler settings from the relevant demo project or makefile. In particular every port requires a macro to be set that ensures the correct kernel header files are included in the build. For example, builds targeted at the MegaAVR using the IAR compiler require IAR_MEGA_AVR to be defined, and builds targeted at the ARM Cortex M3 using the GCC compiler require GCC_ARMCM3 to be defined. The definitions get used by FreeRTOS\Source\include\portable.h – which can be inspected if it is not obvious which definition is required.

Table 21 FreeRTOS source files to include in the project

File	Location
tasks.c	FreeRTOS\Source
queue.c	FreeRTOS\Source
list.c	FreeRTOS\Source
port.c	FreeRTOS\Source\portable\[compiler]\[architecture] where [compiler] is the compiler being used and [architecture] is the microcontroller variant being used.
port.x	Some ports also require the project to include an assembly file. The file will be located in the same directory as port.c. The file name extension will depend on the tool chain being used – x should be replaced with the real file name extension.

Header Files

A source file that uses the FreeRTOS API must include "FreeRTOS.h", then the header file that contains the prototype for the API function being used – either "task.h", "queue.h" or "semphr.h".

APPENDIX 5: DATA TYPES AND CODING STYLE GUIDE

Data Types

Each port of FreeRTOS has a unique portable.h header file in which is defined a set of macros that detail the data types that are used. All the FreeRTOS source code and demo applications use these macro definitions rather than directly using base C data types – but there is absolutely no reason why applications that use FreeRTOS need to do the same. Application writers can substitute the real data types for each of the macros defined within Table 22.

Table 22 Data types used by FreeRTOS

Macro or typedef used	Actual type
portCHAR	char
portSHORT	short
portLONG	long
portTickType	This is used to store the tick count and specify block times.
	portTickType can be either an unsigned 16bit type or an unsigned 32bit type depending on the setting of configUSE_16_BIT_TICKS within FreeRTOSConfig.h.
	Using a 16bit type can greatly improve efficiency on 8 and 16bit architectures but severely limits the range of block times that can specified. It would not make sense to use a 16bit type on a 32bit architecture.
portBASE_TYPE	This will be defined to be the most efficient type for the architecture. Typically this would be a 32bit type on a 32bit architecture, a 16bit type on a 16bit architecture, and an 8 bit type on an 8bit architectures.
	portBASE_TYPE is generally used for return types that can only take a very limited range of values and for Booleans.

Some compilers make all unqualified char variables unsigned, while others make them signed. For this reason the FreeRTOS source code explicitly qualifies every use of portCHAR with either signed or unsigned.

int types are never used – only long and short.

Variable Names

Variables are pre-fixed with their type. 'c' for char, 's' for short, 'l' for long and 'x' for portBASE_TYPE and any other type (structures, task handles, queue handles, etc.).

If a variable is unsigned it is also prefixed with a 'u'. If a variable is a pointer it is also prefixed with a 'p'. Therefore a variable of type unsigned char will be prefixed with 'uc', and a variable of type pointer to char will be prefixed 'pc'.

Function Names

Functions are prefixed with both the type they return and the file they are defined within. For example:

- vTaskPrioritySet() returns a void and is defined within task.c.

- xQueueReceive() returns a variable of type portBASE_TYPE and is defined within queue.c.

- vSemaphoreCreateBinary() returns a void and is defined within semphr.h.

File scope (private) functions are prefixed with 'prv'.

Formatting

1 tab is always set to equal 4 spaces.

Macro Names

Most macros are written in capitals and prefixed with lower case letters that indicate where the macro is defined. Table 23 provides a list of prefixes.

Table 23 Macro prefixes

Prefix	Location of macro definition
port (for example, portMAX_DELAY)	portable.h
task (for example, taskENTER_CRITICAL())	task.h
pd (for example, pdTRUE)	projdefs.h
config (for example, configUSE_PREEMPTION)	FreeRTOSConfig.h
err (for example, errQUEUE_FULL)	projdefs.h

Note that the semaphore API is written almost entirely as a set of macros but follows the function naming convention rather than the macro naming convention.

The macros defined in Table 24 are used throughout the FreeRTOS source.

Table 24 Common macro definitions

Macro	Value
pdTRUE	1
pdFALSE	0
pdPASS	1
pdFAIL	0

Rationale for Excessive Type Casting

The FreeRTOS source code can be compiled with a lot of different compilers, all of which have different quirks as to how and when they generate warnings. In particular different compilers want casting to be used in different ways. As a result the FreeRTOS source code contains more type casting than would normally be warranted.

APPENDIX 6: LICENSING INFORMATION

FreeRTOS is licensed under a *modified* version of the GNU General Public License (GPL) and *can* be used in commercial applications under that license. An alternative and optional commercial license is also available if:

- You cannot fulfill the requirements stated in the "Open Source Modified GPL license" column of Table 25.

- You wish to receive direct technical support.

- You wish to have assistance with your development.

- You require guarantees and indemnification.

Table 25 Open Source Vs Commercial License Comparison

	Open source modified GPL license	Commercial license
Is it free?	Yes	No
Can I use it in a commercial application?	Yes	Yes
Is it royalty free?	Yes	Yes
Do I have to open source my application code?	No	No
Do I have to open source my changes to the FreeRTOS kernel?	Yes	No
Do I have to document that my product uses FreeRTOS.	Yes	No
Do I have to offer to provide the FreeRTOS source code to users of my application?	Yes (a WEB link to the FreeRTOS.org site is normally sufficient)	No
Can I receive support on a commercial basis?	No	Yes
Are any legal guarantees provided?	No	Yes

Open Source License Details

The FreeRTOS source code is licensed under version 2 of the GNU General Public License (GPL) *with an exception*. The full text of the GPL is available at http://www.freertos.org/license.txt. The text of the exception is provided below.

The exception permits the source code of applications that use FreeRTOS solely through the API published on the FreeRTOS.org WEB site to remain closed source, thus permitting the use of FreeRTOS in commercial applications without necessitating that the entire application be open sourced. The exception can only be used if you wish to combine FreeRTOS with a proprietary product and you comply with the terms stated in the exception itself.

GPL Exception Text

Note the exception text is subject to change. Consult the FreeRTOS.org WEB site for the most up to date version.

<u>Clause 1</u>

Linking FreeRTOS statically or dynamically with other modules is making a combined work based on FreeRTOS. Thus, the terms and conditions of the GNU General Public License cover the whole combination.

As a special exception, the copyright holder of FreeRTOS gives you permission to link FreeRTOS with independent modules that communicate with FreeRTOS solely through the FreeRTOS API interface, regardless of the license terms of these independent modules, and to copy and distribute the resulting combined work under terms of your choice, provided that:

1. *Every copy of the combined work is accompanied by a written statement that details to the recipient the version of FreeRTOS used and an offer by yourself to provide the FreeRTOS source code should the recipient request it.*

2. *The combined work is not itself an RTOS, scheduler, kernel or related product.*

3. *The combined work is not itself a library intended for linking into other software applications.*

Any FreeRTOS source code, whether modified or in it's original release form, or whether in whole or in part, can only be distributed by you under the terms of the GNU General Public License plus this exception. An independent module is a module which is not derived from or based on FreeRTOS.

<u>Clause 2</u>

FreeRTOS.org may not be used for any competitive or comparative purpose, including the publication of any form of run time or compile time metric, without the express permission of Richard Barry (this is the norm within the industry and is intended to ensure information accuracy).

INDEX