Using the FreeRTOS™ Real Time Kernel

ARM Cortex-M3 Edition

Richard Barry

Version 1.3.2.

http://www.freertos.org

Contents

List of Figures

List of Code Listings

List of Tables

List of Notation

API	Application Programming Interface
CMSIS	Cortex Microcontroller Software Interface Standard
FAQ	Frequently Asked Question
FIFO	First In First Out
HMI	Human Machine Interface
IDE	Integrated Development Environment
IRQ	Interrupt Request
ISR	Interrupt Service Routine
LCD	Liquid Crystal Display
MCU	Microcontroller
MPU	Memory Protection Unit
RMS	Rate Monotonic Scheduling
RTOS	Real-time Operating System
SIL	Safety Integrity Level
TCB	Task Control Block
UART	Universal Asynchronous Receiver/Transmitter

Preface

FreeRTOS and the Cortex-M3

Multitasking on a Cortex-M3 Microcontroller

An Introduction to Multitasking in Small Embedded Systems

Microcontrollers (MCUs) that contain an ARM Cortex-M3 core are available from many manufacturers and are ideally suited to deeply embedded real-time applications. Typically, applications of this type include a mix of both hard and soft real-time requirements.

Soft real-time requirements are those that state a time deadline—but breaching the deadline would not render the system useless. For example, responding to keystrokes too slowly may make a system seem annoyingly unresponsive without actually making it unusable.

Hard real-time requirements are those that state a time deadline—and breaching the deadline would result in absolute failure of the system. For example, a driver's airbag would be useless if it responded to crash sensor inputs too slowly.

FreeRTOS is a real-time kernel (or real-time scheduler) on top of which Cortex-M3 microcontroller applications can be built to meet their hard real-time requirements. It allows Cortex-M3 microcontroller applications to be organized as a collection of independent threads of execution. As most Cortex-M3 microcontroller have only one core, in reality only a single thread can be executing at any one time. The kernel decides which thread should be executing by examining the priority assigned to each thread by the application designer. In the simplest case, the application designer could assign higher priorities to threads that implement hard real-time requirements, and lower priorities to threads that implement soft real-time requirements. This would ensure that hard real-time threads are always executed ahead of soft real-time threads, but priority assignment decisions are not always that simplistic.

Do not be concerned if you do not fully understand the concepts in the previous paragraph yet. The following chapters provide a detailed explanation, with many examples, to help you understand how to use a real-time kernel, and how to use FreeRTOS in particular.

A Note About Terminology

In FreeRTOS, each thread of execution is called a 'task'. There is no consensus on terminology within the embedded community, but I prefer 'task' to 'thread' as 'thread' can have a more specific meaning in some fields of application.

Why Use a Real-time Kernel?

There are many well established techniques for writing good embedded software without the use of a kernel, and, if the system being developed is simple, then these techniques might provide the most appropriate solution. In more complex cases, it is likely that using a kernel would be preferable, but where the crossover point occurs will always be subjective.

As already described, task prioritization can help ensure an application meets its processing deadlines, but a kernel can bring other less obvious benefits, too. Some of these are listed very briefly below:

- Abstracting away timing information

  The kernel is responsible for execution timing and provides a time-related API to the application. This allows the structure of the application code to be simpler and the overall code size to be smaller.

- Maintainability/Extensibility

  Abstracting away timing details results in fewer interdependencies between modules and allows the software to evolve in a controlled and predictable way. Also, the kernel is responsible for timing, so application performance is less susceptible to changes in the underlying hardware.

- Modularity

  Tasks are independent modules, each of which should have a well-defined purpose.

- Team development

  Tasks should also have well-defined interfaces, allowing easier development by teams.

- Easier testing

  If tasks are well-defined independent modules with clean interfaces, they can be tested in isolation.

- Code reuse

  Greater modularity and fewer interdependencies can result in code that can be re-used with less effort.

- Improved efficiency

Using a kernel allows software to be completely event-driven, so no processing time is wasted by polling for events that have not occurred. Code executes only when there is something that must be done.

Counter to the efficiency saving is the need to process the RTOS tick interrupt and to switch execution from one task to another.

- Idle time

The Idle task is created automatically when the kernel is started. It executes whenever there are no application tasks wishing to execute. The idle task can be used to measure spare processing capacity, to perform background checks, or simply to place the processor into a low-power mode.

- Flexible interrupt handling

Interrupt handlers can be kept very short by deferring most of the required processing to handler tasks. Section 3.2 demonstrates this technique.

- Mixed processing requirements

Simple design patterns can achieve a mix of periodic, continuous, and event-driven processing within an application. In addition, hard and soft real-time requirements can be met by selecting appropriate task and interrupt priorities.

- Easier control over peripherals

Gatekeeper tasks can be used to serialize access to peripherals.

The Cortex-M3 Port of FreeRTOS

The Cortex-M3 port includes all the standard FreeRTOS features:

- Pre-emptive or co-operative operation
- Very flexible task priority assignment
- Queues
- Binary semaphores
- Counting semaphores

- Recursive semaphores
- Mutexes
- Tick hook functions
- Idle hook functions
- Stack overflow checking
- Trace hook macros
- Optional commercial licensing and support

FreeRTOS also manages interrupt nesting, and allows interrupts above a user-definable priority level to remain unaffected by the activity of the kernel. Using FreeRTOS will not introduce any additional timing jitter or latency for these interrupts.

There are two separate FreeRTOS ports for the Cortex-M3:

1. FreeRTOS-MPU

   FreeRTOS-MPU includes full Memory Protection Unit (MPU) support. In this version, tasks can execute in either User mode or Privileged mode. Also, access to Flash, RAM, and peripheral memory regions can be tightly controlled, on a task-by-task basis.

   Not all Cortex-M3 microcontrollers include MPU hardware.

2. FreeRTOS (the original port)

   This does not include any MPU support. All tasks execute in the Privileged mode and can access the entire memory map.

The examples that accompany this text use the original FreeRTOS version without MPU support, but a chapter describing FreeRTOS-MPU is included for completeness (see Chapter 7).

Resources Used By FreeRTOS

FreeRTOS makes use of the Cortex-M3 SysTick, PendSV, and SVC interrupts. These interrupts are not available for use by the application.

FreeRTOS has a very small footprint. A typical kernel build will consume approximately 6K bytes of Flash space and a few hundred bytes of RAM. Each task also requires RAM to be allocated for use as the task stack.

The FreeRTOS, OpenRTOS, and SafeRTOS Family

FreeRTOS uses a modified GPL license. The modification is included to ensure:

1. FreeRTOS can be used in commercial applications.

2. FreeRTOS itself remains open source.

3. FreeRTOS users retain ownership of their intellectual property.

When you link FreeRTOS into an application, you are obliged to open source only the kernel, including any additions or modifications you may have made. Components that merely use FreeRTOS through its published API can remain closed source and proprietary. Appendix 1: contains the modification text.

OpenRTOS shares the same code base as FreeRTOS, but is provided under standard commercial license terms. The commercial license removes the requirement to open source any code at all and provides IP infringement protection.

OpenRTOS can be purchased with a professional support contract and a selection of other useful components such as TCP/IP stacks and drivers, USB stacks and drivers, and various different file systems. Evaluation versions can be downloaded from http://www.OpenRTOS.com.

Table 1 provides an overview of the differences between the FreeRTOS and OpenRTOS license models.

SafeRTOS has been developed in accordance with the practices, procedures, and processes necessary to claim compliance with various internationally recognized safety related standards.

IEC 61508 is an international standard covering the development and use of electrical, electronic, and programmable electronic safety-related systems. The standard defines the analysis, design, implementation, production, and test requirements for safety-related systems, in accordance with the Safety Integrity Level (SIL) assigned to the system. The SIL is assigned according to the risks associated with the use of the system under development, with a maximum SIL of 4 being assigned to systems with the highest perceived risk. The SafeRTOS development process has been independently certified by TÜV SÜD as being in compliance with that required by IEC 61508 for SIL 3 applications. SafeRTOS is supplied with

complete lifecycle compliance evidence and has itself been certified for use in IEC 61508, IEC 62304 and FDA 510(K) applications.

SafeRTOS was originally derived from FreeRTOS and retains a similar usage model. Visit http://www.SafeRTOS.com for additional information.

Table 1. Comparing the FreeRTOS license with the OpenRTOS license

	FreeRTOS License	OpenRTOS 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 that makes use of FreeRTOS services?	No, as long as the code provides functionality that is distinct from that provided by FreeRTOS	No
Do I have to open source my changes to the kernel?	Yes	No
Do I have to document that my product uses FreeRTOS?	Yes	No
Do I have to offer to provide the FreeRTOS code to users of my application?	Yes	No
Can I buy an annual support contract?	No	Yes
Is a warranty provided?	No	Yes
Is legal protection provided?	No	Yes, IP infringement protection is provided

Using the Examples that Accompany this Book

Required Tools and Hardware

The examples described in this book are included in an accompanying .zip file. You can download the .zip file from http://www.FreeRTOS.org/Documentation/code if you did not receive a copy with the book.

.zip files are provided for Cortex-M3 microcontrollers from several different manufacturers and using several different compilers. Each .zip file also contains the appropriate build instructions.

Chapter 1

Task Management

1.1 Chapter Introduction and Scope

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.

Readers should also 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. The only thing special about them is their prototype, which must return void and take a void pointer parameter. 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 an infinite loop. */
    for( ;; )
    {
        /* The code to implement the task functionality will go here. */
    }

    /* Should the task implementation ever break out of the above loop
    then the task must be deleted before reaching the end of 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 contains a single core, then only one task can 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 that the Not Running state actually contains a number of sub-states.

When a task is in the Running state, the processor is 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 the instruction it was about to execute before it last left the Running state.

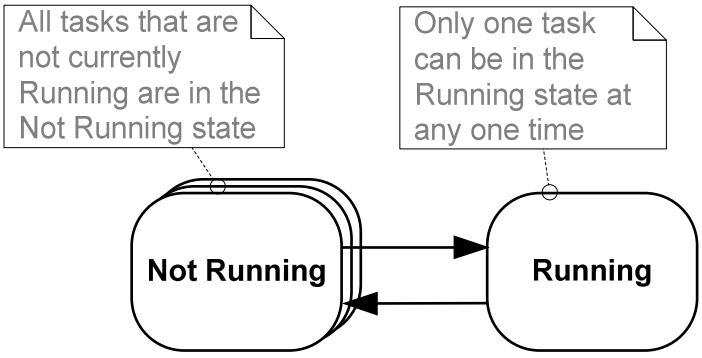


Figure 1. Top level task states and transitions

A task transitioned from the Not Running state 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

The 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.

Section 8.5 describes the data types and naming conventions used.

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

Listing 3. The xTaskCreate() API function prototype

Table 2. xTaskCreate() parameters and return value

Parameter Name/ Returned Value	Description
pvTaskCode	Tasks are simply 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 identify it by 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 result in the string being silently truncated.

Table 2. xTaskCreate() parameters and return value

Parameter Name/ Returned Value	Description
usStackDepth	Each task has its own unique stack that is allocated by the kernel to the task when the task is created. The usStackDepth value tells the kernel how large to make the stack. The value specifies the number of words the stack can hold, not the number of bytes. For example, the Cortex-M3 stack is 32 bits wide so, if usStackDepth is passed in as 100, then 400 bytes of stack space will be allocated (100 * 4 bytes). 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 standard FreeRTOS Cortex-M3 demo applications is the minimum recommended for any task. If your task uses a lot of stack space, then you must assign a larger value. There is no easy way to determine 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 that the space allocated is indeed adequate, and that RAM is not being wasted unnecessarily. 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 in this document demonstrate how the parameter can be used.

Table 2. xTaskCreate() parameters and return value

Parameter Name/ Returned Value	Description
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 to 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 required, to avoid wasting RAM. Passing a uxPriority value above (configMAX_PRIORITIES – 1) will result in the priority assigned to the task being capped silently to the maximum legitimate value.
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 in 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 has been created successfully. 2. errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY This indicates that the task has not been created because there is 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

This example demonstrates the steps needed to create two simple tasks then start the tasks executing. The tasks simply print out a string periodically, using a crude null loop to create the period delay. Both tasks are created at the same priority and are identical except for 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\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\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 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. */
                    240,     /* Stack depth in words. */
                    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", 240, 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

The output generated by vPrintString() is displayed in the chosen IDE. Executing this example produces the output shown in Figure 2, which is a screen shot from the Red Suite IDE.

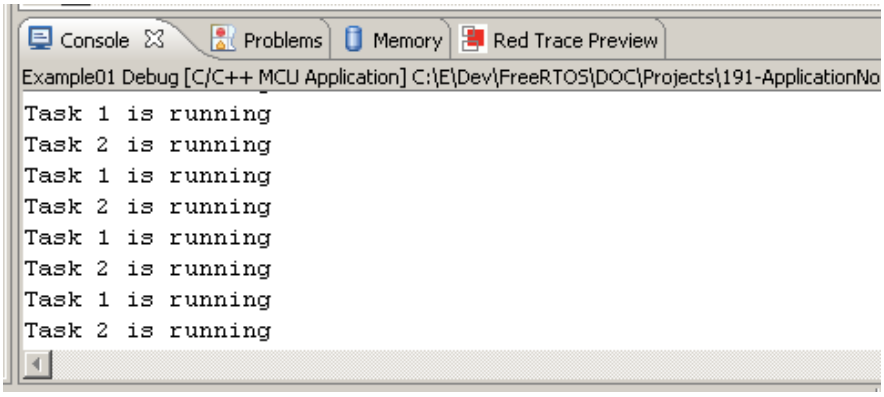


Figure 2. The output produced when Example 1 is executed

Figure 2 shows the two tasks appearing to execute simultaneously; however, as both tasks are executing on the same processor, this cannot be the case. In reality, both tasks are rapidly entering and exiting the Running state. Both tasks are running at the same priority, and 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, Task 1 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).

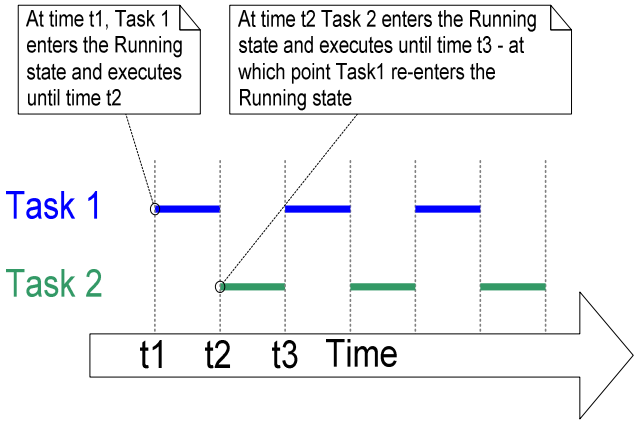


Figure 3. The execution pattern of the two Example 1 tasks

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 Task 1 from main(), and then created Task 2 from within Task 1. Were we to do this, our Task 1 function would change as shown by Listing 7. Task 2 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\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", 240, 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 are almost identical, the only difference between them being the text string they print 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 it 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.

```
/* 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\n";
static const char *pcTextForTask2 = "Task 2 is running\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. */
                    240,                    /* Stack depth in words */
                    (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", 240, (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 imposes no 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 desired (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. Therefore, the range of available priorities is 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 are created at the same priority and both are 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 must 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 10 milliseconds. 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 ‘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 have to consider overflows when specifying delay periods, as time consistency is managed internally by the kernel.

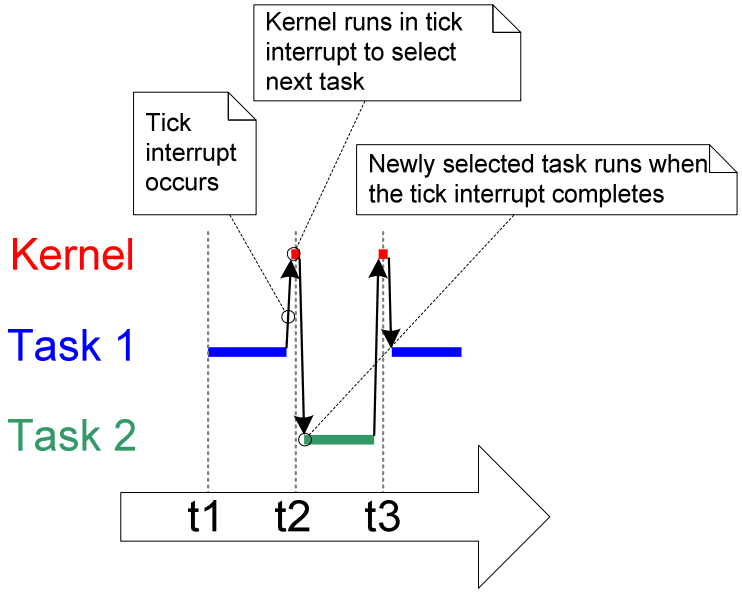


Figure 4. The execution sequence expanded to show the tick interrupt executing

In Figure 4, the short top lines show when the kernel itself is running. The 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 simply prints out a string periodically, 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\n";
static const char *pcTextForTask2 = "Task 2 is running\n";

int main( void )
{
    /* Create the first task at priority 1.  The priority is the second to last
    parameter. */
    xTaskCreate( vTaskFunction, "Task 1", 240, (void*)pcTextForTask1, 1, NULL );

    /* Create the second task at priority 2. */
    xTaskCreate( vTaskFunction, "Task 2", 240, (void*)pcTextForTask2, 2, NULL );

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

    /* If all is well we will never reach here as the scheduler will now be
    running.  If we do reach here then it is likely that there was insufficient
    heap available for the idle task to be created. */
    for( ;; );
}
```

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.

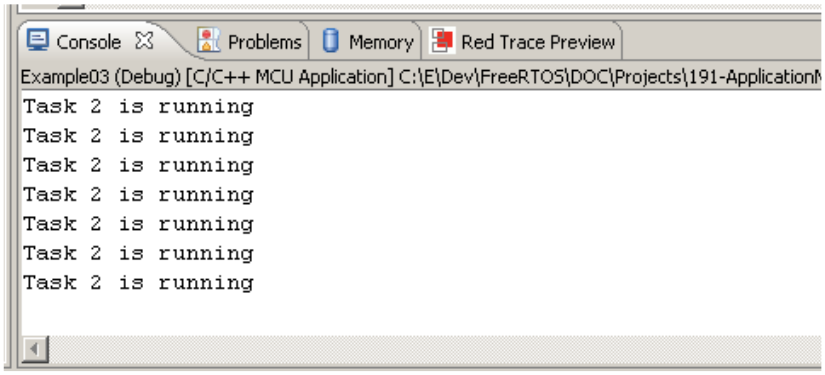


Figure 5. Running both test tasks at different priorities

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

Figure 6 shows the execution sequence for Example 3.

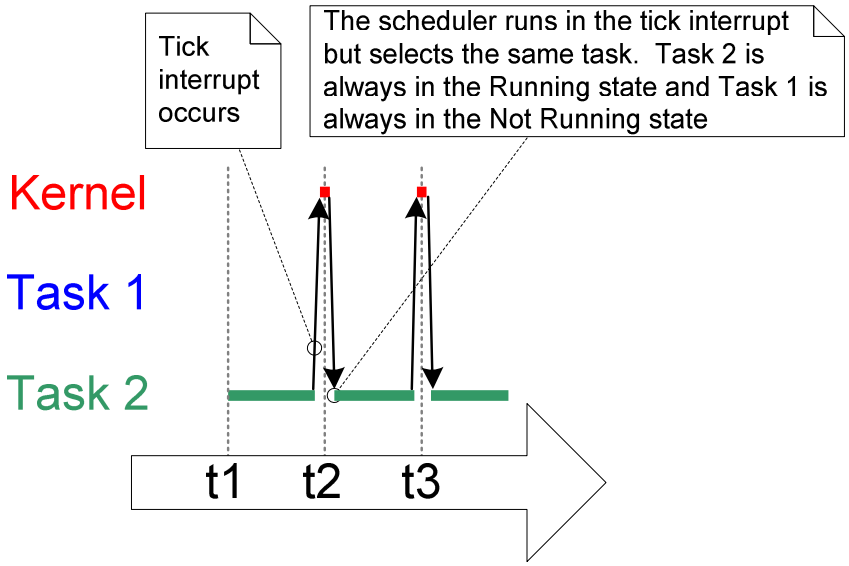


Figure 6. The execution pattern when one task has a higher priority than the other

1.6 Expanding the 'Not Running' State

So far, the created tasks have always had processing to perform and have never had 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 useful, we need a way to allow them to be event-driven. An event-driven task has work (processing) to perform only after the occurrence of the event that triggers it, and is not *able* to enter the Running state before that 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 that the scheduler cannot select them and must, instead, select a lower priority task that is able to run. Therefore, using event-driven tasks means that tasks can be created at 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, the only way out being through a call to the vTaskResume() or xTaskResumeFromISR() API functions. Most applications do not use the Suspended state.

The Ready State

Tasks that are in the Not Running state 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 are 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 the Blocked or Suspended states; they have only transitioned between the Ready state and the Running state—highlighted by the bold lines in Figure 7.

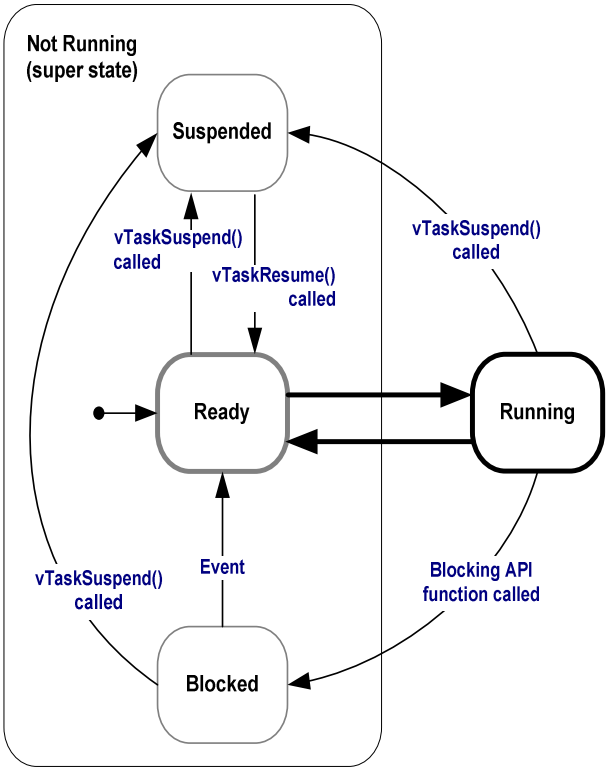


Figure 7. Full task state machine

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 and 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. During 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. Note that the vTaskDelay() API function is available only when INCLUDE_vTaskDelay is set to 1 in FreeRTOSConfig.h.

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

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

Listing 11. The vTaskDelay() API function prototype

Table 3. 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.

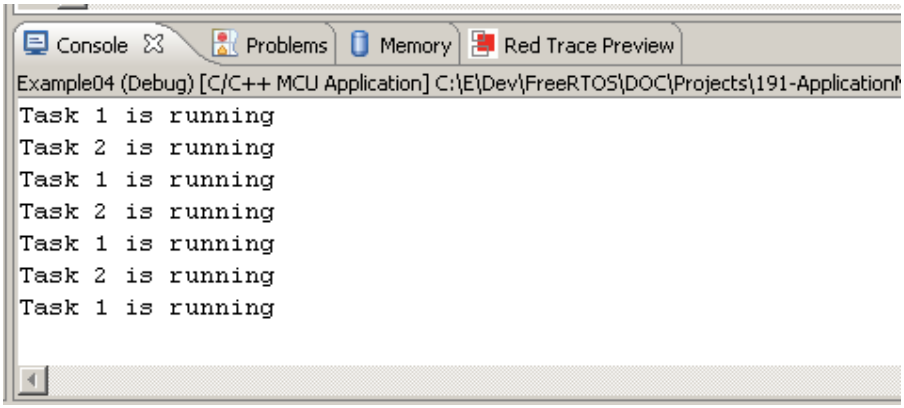


Figure 8. The output produced when Example 4 is executed

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.

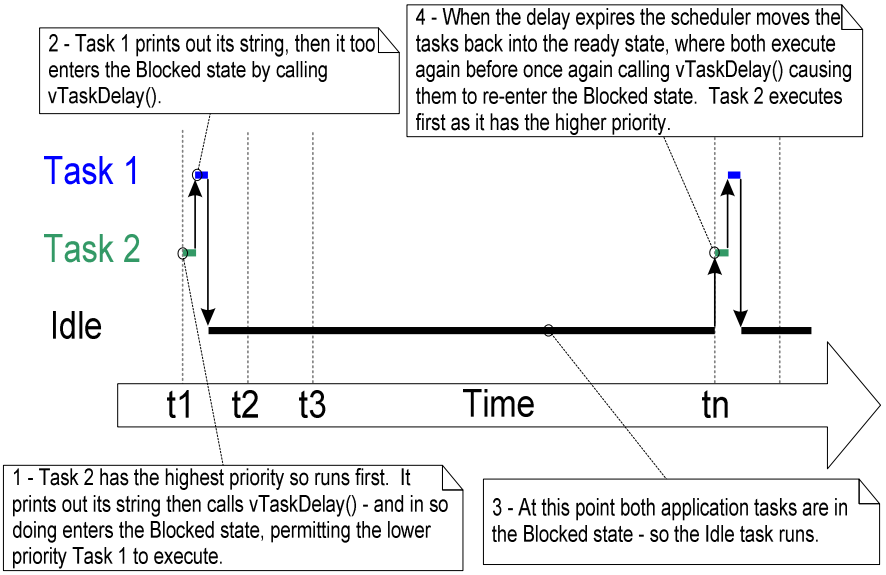


Figure 9. The execution sequence when the tasks use vTaskDelay() in place of the NULL loop

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

Figure 4 shows the execution pattern when the tasks use a null loop to create a delay—so are always able to run and use 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 use processor time only when they actually have work that needs to be performed (in this case simply a message to be printed out).

In the Figure 9 scenario, each time the tasks leave the Blocked state they 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 (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.

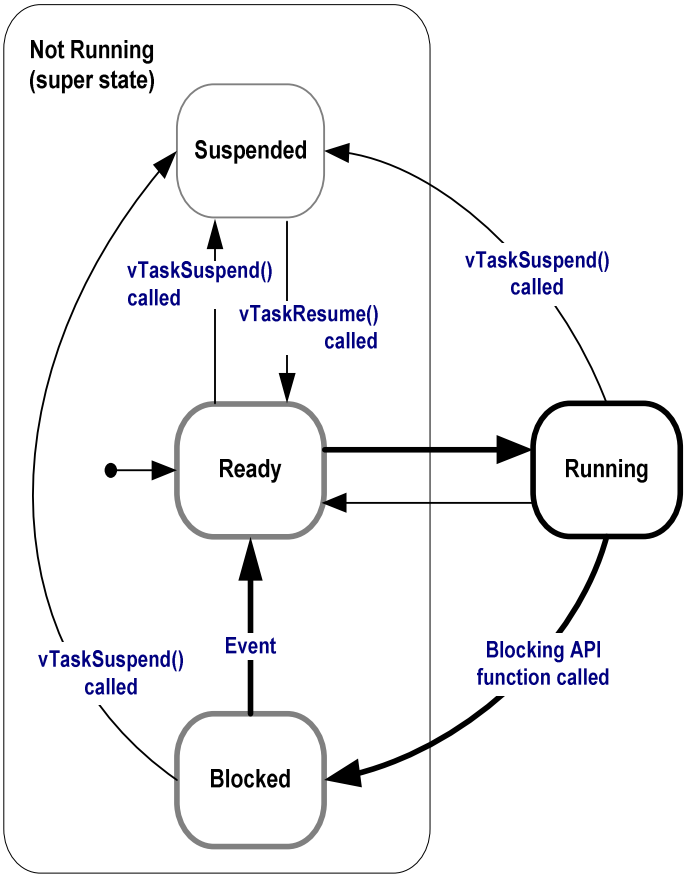


Figure 10. Bold lines indicate the state transitions performed by the tasks in Example 4

The 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 length 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() specify, instead, 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()).

Note that the vTaskDelayUntil() API function is available only when INCLUDE_vTaskDelayUntil is set to 1 in FreeRTOSConfig.h.

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

Listing 13. vTaskDelayUntil() API function prototype

Table 4. 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; it 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.

Table 4. vTaskDelayUntil() parameters

Parameter Name	Description
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 to ticks.

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

The two tasks created in Example 4 are periodic tasks, but using vTaskDelay() does not guarantee that the frequency at which they run is 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() instead of vTaskDelay() solves 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 is 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,

1. 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.

2. A third task is then created at priority 2; that is, 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 function 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…….\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.

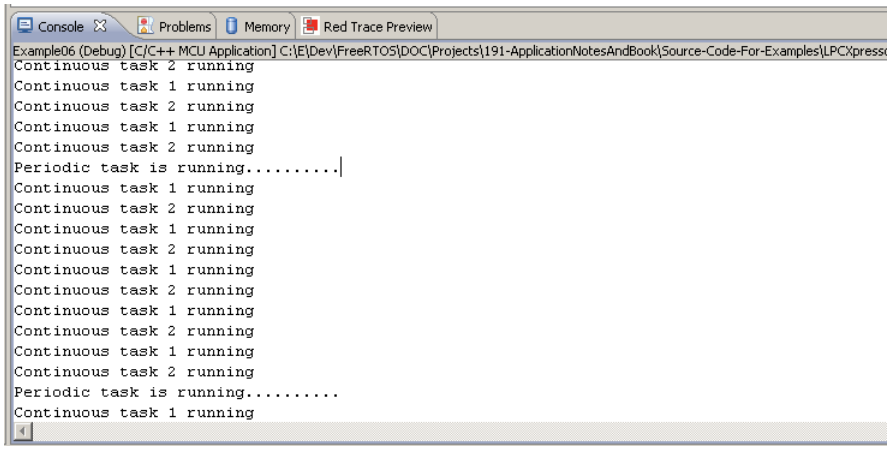


Figure 11. The output produced when Example 6 is executed

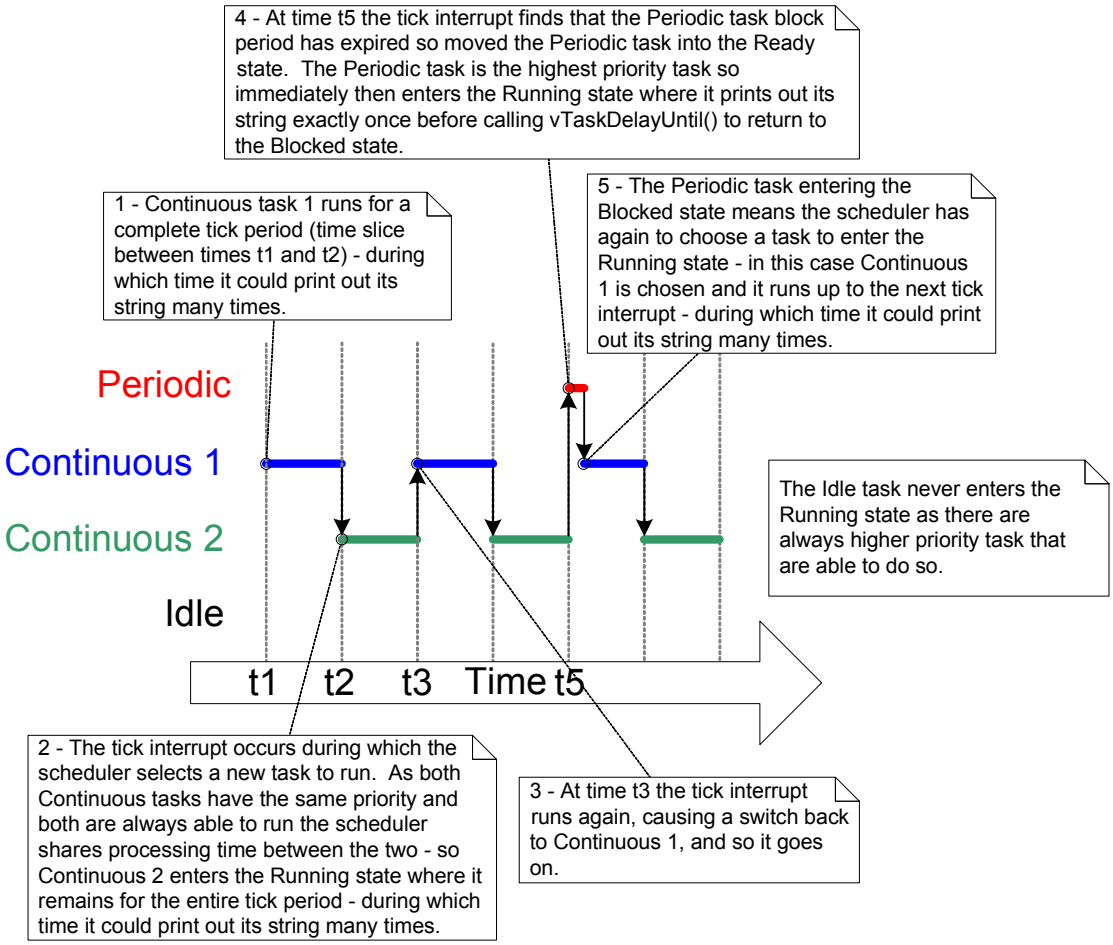


Figure 12. The execution pattern of Example 6

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 zero), to ensure it never prevents a higher priority application task from entering the Running state—although there is nothing to prevent application designers creating tasks at, and therefore sharing, the idle task priority, if desired.

Running at the lowest priority ensures that the Idle task is transitioned immediately 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 idle callback) function—a function that is called automatically 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 run only when all 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. An idle task hook function must never attempt to block or suspend. The Idle task will execute only when no other tasks are able to do so (unless application tasks are sharing the idle priority).

   Note: Blocking the idle task in any way could 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 in 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 creates a lot of idle time—time when the Idle task is executing because both application tasks are in the Blocked state. Example 7 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 of 250 milliseconds. */
        vTaskDelay( 250 / portTICK_RATE_MS );
    }
}
```

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

The output produced by Example 7 is shown in Figure 13 and shows that the idle task hook function is called approximately 830000 times between each iteration of the application tasks.

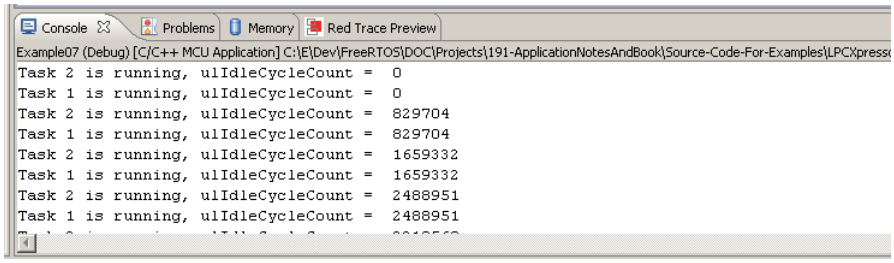


Figure 13. The output produced when Example 7 is executed

1.8 Changing the Priority of a Task

The vTaskPrioritySet() API Function

The vTaskPrioritySet() API function can be used to change the priority of any task after the scheduler has been started. Note that the vTaskPrioritySet() API function is available only when INCLUDE_vTaskPrioritySet is set to 1 in FreeRTOSConfig.h.

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

Listing 20. The vTaskPrioritySet() API function prototype

Table 5. 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 is capped automatically to the maximum available priority of (configMAX_PRIORITIES – 1), where configMAX_PRIORITIES is a compile time option set in the FreeRTOSConfig.h header file.

The uxTaskPriorityGet() API Function

The uxTaskPriorityGet() API function can be used to query the priority of a task. Note that the vTaskPriorityGet() API function is available only when INCLUDE_vTaskPriorityGet is set to 1 in FreeRTOSConfig.h.

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

Listing 21. The uxTaskPriorityGet() API function prototype

Table 6. 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 makes any API function calls that could cause it to enter the Blocked state, so both are always in either the Ready state 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:

1. Task 1 (Listing 22) is created with the highest priority, so is guaranteed to run first. Task 1 prints out a couple of strings before raising the priority of Task 2 (Listing 23) to above its own priority.

2. Task 2 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 Task 2 is in the Running state, Task 1 is in the Ready state.

3. Task 2 prints out a message before setting its own priority back to below that of Task 1.

4. Task 2 setting its priority back down means Task 1 is once again the highest priority task, so Task 1 re-enters the Running state, forcing Task 2 back into the Ready state.

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

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

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

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

        /* Setting the Task 2 priority above the Task 1 priority will cause
        Task 2 to immediately start running (as then Task 2 will have the higher
        priority of the two created tasks).  Note the use of the handle to task
        2 (xTask2Handle) in the call to vTaskPrioritySet().  Listing 24 shows how
        the handle was obtained. */
        vPrintString( "About to raise the Task 2 priority\n" );
        vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );

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

Listing 22. The implementation of Task 1 in Example 8

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

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

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

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

        Print out the name of this task. */
        vPrintString( "Task2 is running\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 Task 1 will cause Task 1 to immediately start
        running again – pre-empting this task. */
        vPrintString( "About to lower the Task 2 priority\n" );
        vTaskPrioritySet( NULL, ( uxPriority - 2 ) );
    }
}
```

Listing 23. The implementation of Task 2 in Example 8

Each task can both query and set its own priority, without the use of a valid task handle, by simply using NULL, instead. A task handle is required only when a task wishes to reference a task other than itself, such as when Task 1 changes the priority of Task 2. To allow Task 1 to do this, the Task 2 handle is obtained and saved when Task 2 is created, as highlighted in the comments in Listing 24.

```
/* Declare a variable that is used to hold the handle of Task 2. */
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", 240, NULL, 2, NULL );
    /* The task is created at priority 2 _____^. */

    /* Create the second task at priority 1 - which is lower than the priority
    given to Task 1.  Again the task parameter is not used so is set to NULL -
    BUT this time the task handle is required so the address of xTask2Handle
    is passed in the last parameter. */
    xTaskCreate( vTask2, "Task 2", 240, 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.

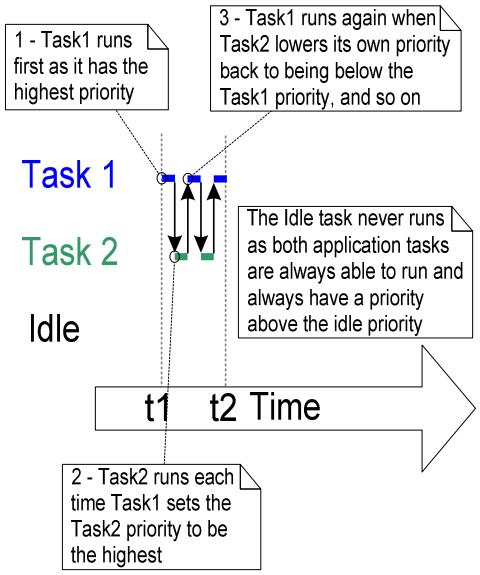


Figure 14. The sequence of task execution when running Example 8

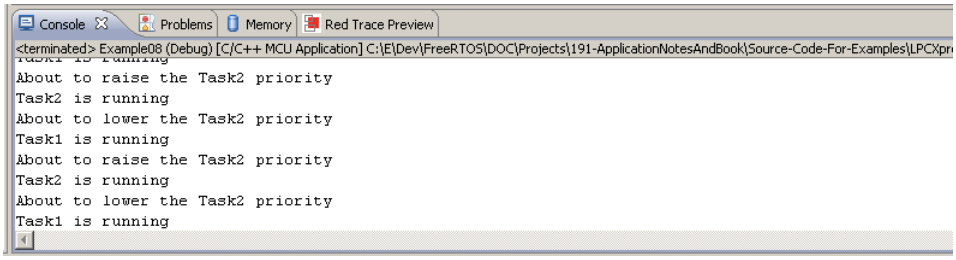


Figure 15. The output produced when Example 8 is executed

1.9 Deleting a Task

The vTaskDelete() API Function

A task can use the vTaskDelete() API function to delete itself or any other task. Note that the vTaskDelete() API function is available only when INCLUDE_vTaskDelete is set to 1 in FreeRTOSConfig.h.

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

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

Note that only memory allocated to a task by the kernel itself will be freed automatically 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 7. 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.

1. 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 it starts to execute immediately. The source for main() is shown in Listing 26, and for Task 1 in Listing 27.

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

3. When Task 2 has been deleted, Task 1 is again the highest priority task, so continues executing—at which point it calls vTaskDelay() to block for a short period.

4. The Idle task executes while Task 1 is in the blocked state and frees the memory that was allocated to the now deleted Task 2.

5. When Task 1 leaves the blocked state it again becomes the highest priority Ready state task and so pre-empts the Idle task. When it enters the Running state it creates Task 2 again, and so it goes on.

```
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", 240, NULL, 1, NULL );
    /* The task is created at priority 1 _____^. */

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

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

Listing 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( "Task 1 is running\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", 240, NULL, 2, &xTask2Handle );
        /* The task handle is the last parameter ____^^^^^^^^^^^^^ */

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

Listing 27. The implementation of Task 1 for Example 9

```
void vTask2( void *pvParameters )
{
    /* Task 2 does nothing but delete itself.  To do this it could call vTaskDelete()
    using NULL as the parameter, but instead and purely for demonstration purposes it
    instead calls vTaskDelete() passing its own task handle. */
    vPrintString( "Task2 is running and about to delete itself\n" );
    vTaskDelete( xTask2Handle );
}
```

Listing 28. The implementation of Task 2 for Example 9

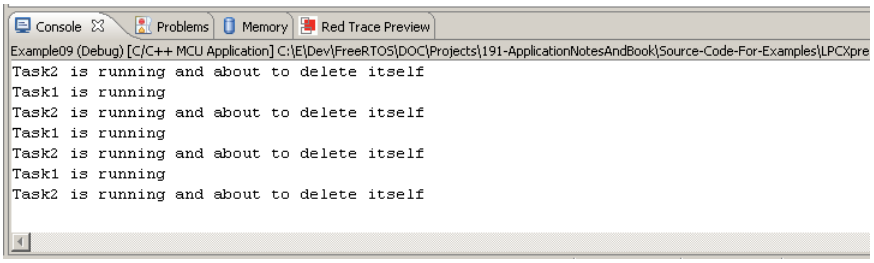


Figure 16. The output produced when Example 9 is executed

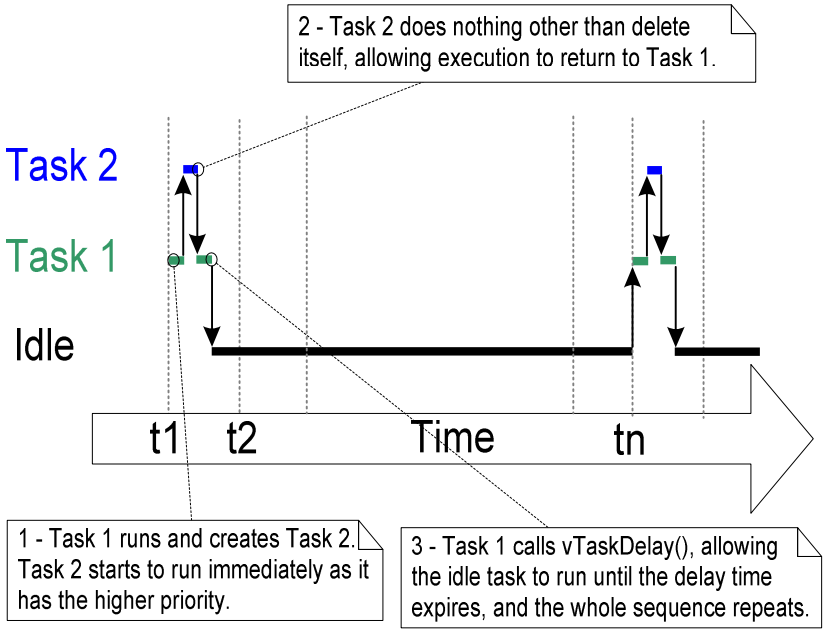


Figure 17. The execution sequence for Example 9

1.10 The Scheduling Algorithm—A Summary

Prioritized Pre-emptive Scheduling

The examples in this chapter illustrate 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 always selects the highest priority Ready state task to enter the Running state.

This type of scheme is called 'Fixed Priority Pre-emptive Scheduling'—'Fixed Priority' because each task is assigned a priority that is not altered by the kernel itself (only tasks can change priorities); 'Pre-emptive' 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 are 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 a queue or to 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.

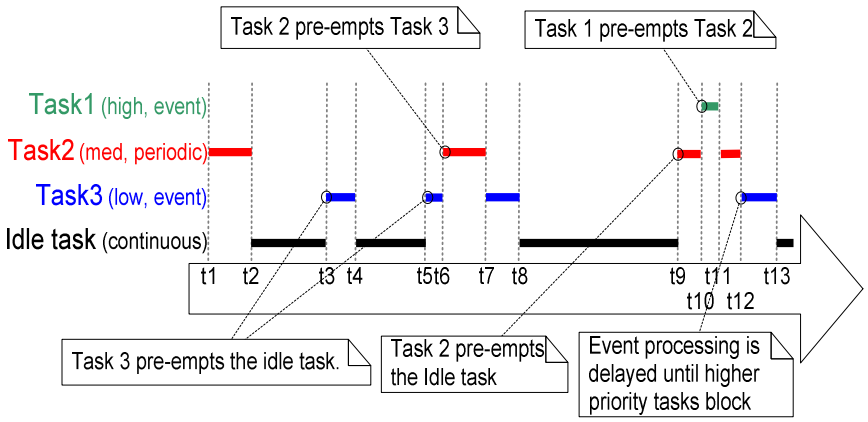


Figure 18. Execution pattern with pre-emption points highlighted

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 and 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 complete its processing only after Task 1 has re-entered the Blocked at time t11.

Selecting Task Priorities

Figure 18 shows the fundamental importance of priority assignment 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 which dictates that 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, make absolute calculations a complex process.

Co-operative Scheduling

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

When a pure co-operative scheduler is used, a context switch will occur only 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, where interrupt service routines are used to explicitly cause a context switch, is also possible. This allows synchronization events to cause pre-emption, but not temporal events. The result is a pre-emptive system without time slicing. 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 these autonomous tasks will have to communicate with each other so that, collectively, they can provide useful system functionality. The 'queue' is 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 of task priorities 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 of each operation on the data stored in the queue.

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, whereas 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 moved automatically from the Blocked state to the Ready state if the specified block time expires before data becomes available.

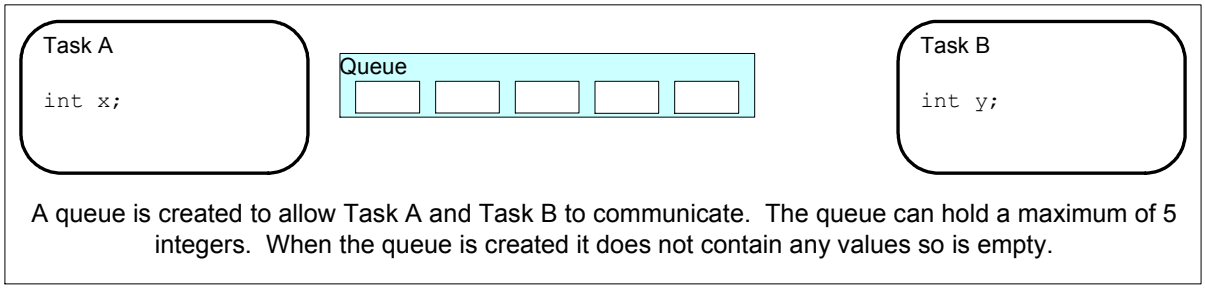
Queues can have multiple readers so it is possible for a single queue to have more than one task blocked on it waiting for data. When this is the case, only one task will be unblocked when data becomes available. The task that is unblocked will always be the highest priority

task that is waiting for data. If the blocked tasks have equal priority, then the task that has been waiting for data the longest will be unblocked.

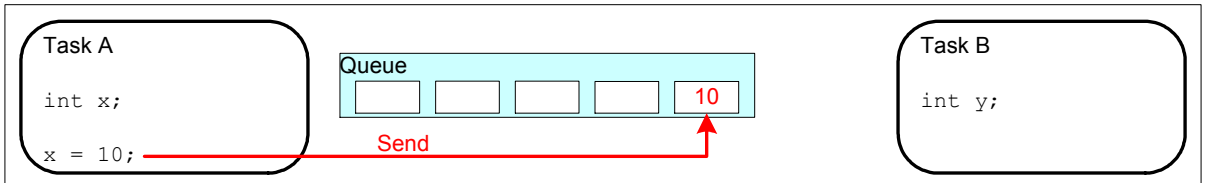
Blocking on Queue Writes

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.

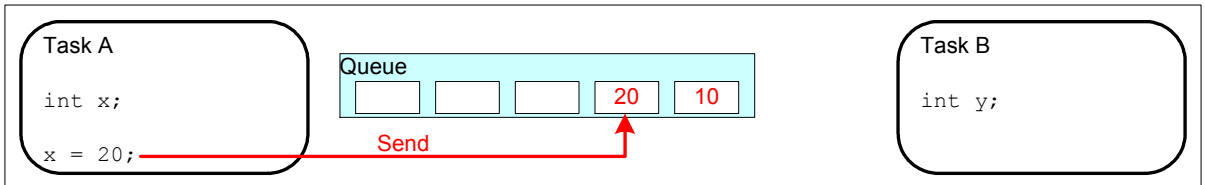
Queues can have multiple writers, so it is possible for a full queue to have more than one task blocked on it waiting to complete a send operation. When this is the case, only one task will be unblocked when space on the queue becomes available. The task that is unblocked will always be the highest priority task that is waiting for space. If the blocked tasks have equal priority, then the task that has been waiting for space the longest will be unblocked.



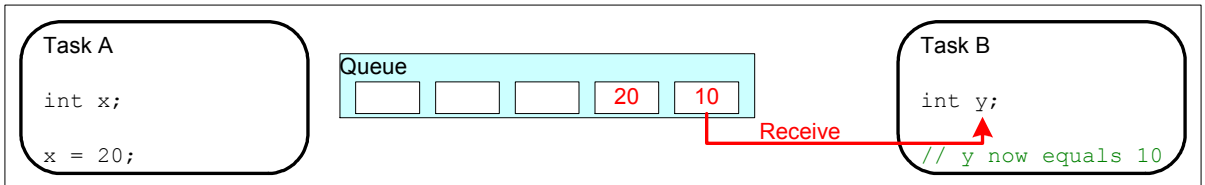
A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.



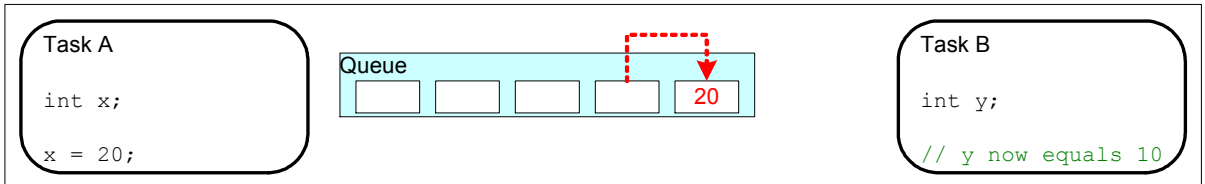
Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.



Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. The queue has three empty spaces remaining.



Task B reads (receives) from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (10 in this illustration).



Task B has removed one item, leaving only the second value written by Task A remaining in the queue. This is the value Task B would receive next if it read from the queue again. The queue now has four empty spaces remaining.

Figure 19. An example sequence of writes and reads to and from a queue

2.3 Using a Queue

The 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 8, 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 cannot be created because there is 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 has been created successfully. The returned value should be stored as the handle to the created queue.

The 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().

Note: 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 9. 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 to 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 9. 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 zero and the queue is already full. 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.

Table 9. xQueueSendToFront() and xQueueSendToBack() function parameters and return value

Parameter Name/ Returned Value	Description
Returned value	There are two possible return values: 1. pdPASS pdPASS will be returned only 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.

The 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() receives 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.

Note: 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
                            );
```

Listing 32. The xQueueReceive() API function prototype

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

Listing 33. The xQueuePeek() API function prototype

Table 10. 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 10. 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.

Table 10. xQueueReceive() and xQueuePeek() function parameters and return values

Parameter Name/ Returned value	Description
Returned value	There are two possible return values: 1. pdPASS pdPASS will be returned only if data was successfully read from 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 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 will be returned if data cannot be read from the queue because the queue is 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.

The uxQueueMessagesWaiting() API Function

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

Note: 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 34. The uxQueueMessagesWaiting() API function prototype

Table 11. 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 zero 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, whereas 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 that 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 35 shows the implementation of the task that writes to the queue. Two instances of this task are created, one that writes continuously the value 100 to the queue, and another that writes continuously 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.\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 35. Implementation of the sending task used in Example 10

Listing 36 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 writing continuously 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!\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.\n" );
        }
    }
}
```

Listing 36. Implementation of the receiver task for Example 10

Listing 37 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 five long values, even though the priorities of the tasks are set such that the queue will never contain more than one item at a time.

```
/* Declare a variable of type xQueueHandle.  This is used to store the handle
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", 240, ( void * ) 100, 1, NULL );
        xTaskCreate( vSenderTask, "Sender2", 240, ( 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", 240, 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 37. The implementation of main() for 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 20.

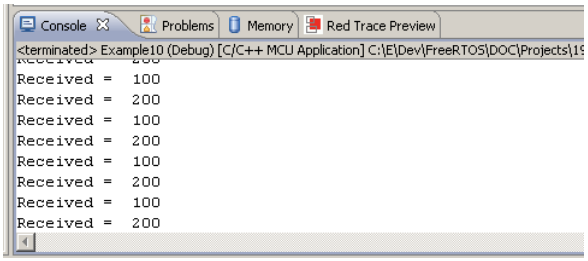


Figure 20. The output produced when Example 10 is executed

Figure 21 demonstrates the sequence of execution.

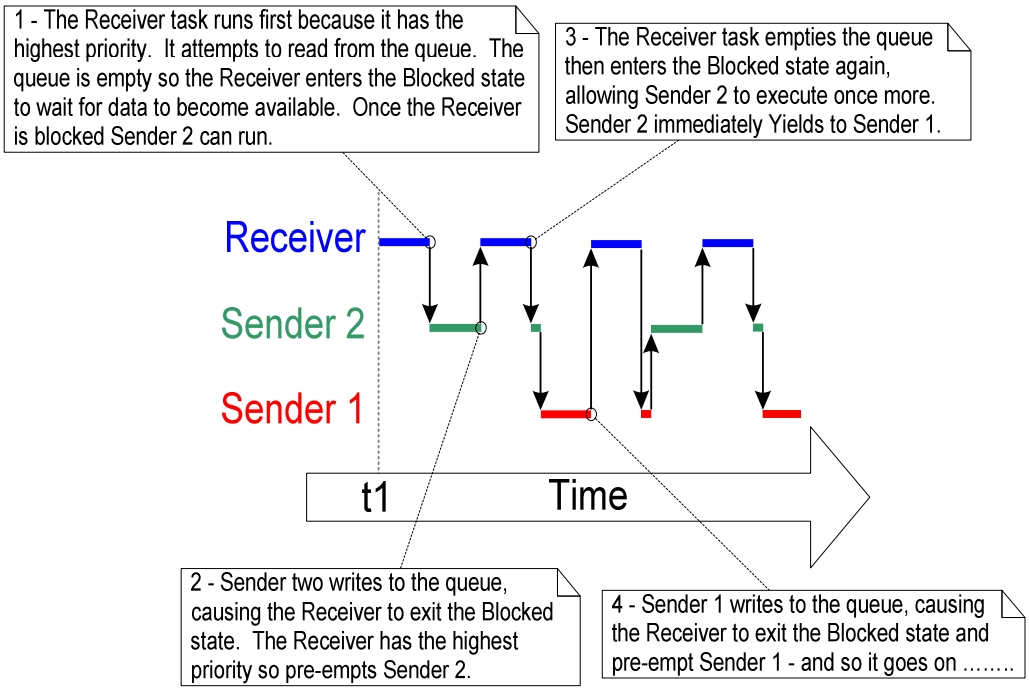


Figure 21. The sequence of execution produced by Example 10

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, to allow it to determine how the data should be processed. A simple way to achieve 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 22.

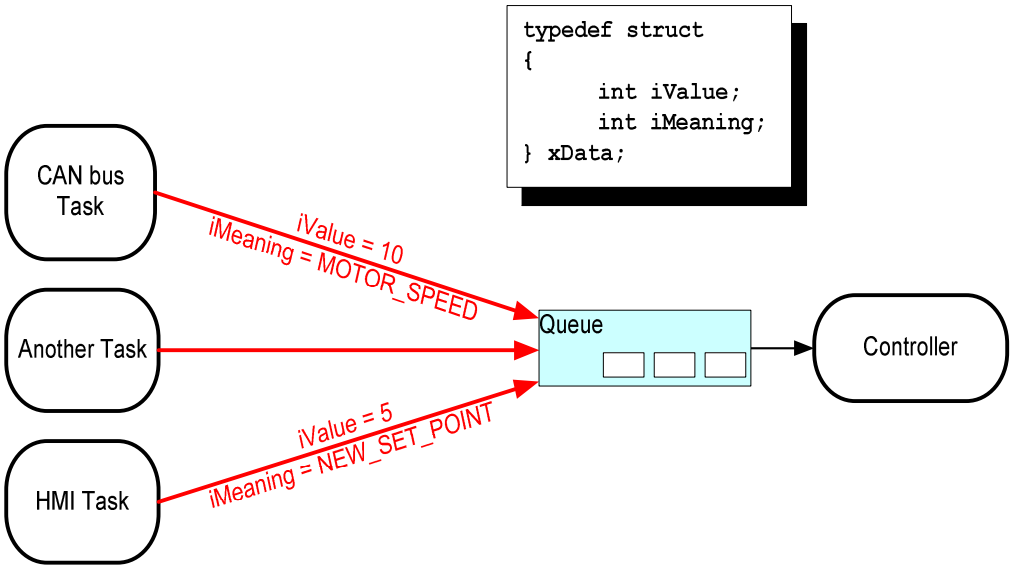


Figure 22. An example scenario where structures are sent on a queue

Referring to Figure 22:

- 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 Controller task know what the data is—in the depicted case it is a motor speed value. The iValue member of the transferred structure is used to let the Controller 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 query 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 Controller 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 Controller task know the actual set point value.

Example 11. Blocking when sending to a queue or 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, rather than simple long integers, between the tasks.

Listing 38 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 38. 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 has the highest priority, so the queue never contains more than one item. This is caused by the receiving task pre-empting the sending tasks as soon as data is placed into the queue. In Example 11, the sending tasks have the higher priority, so the queue will normally be full. This occurs because, as soon as the receiving task removes an item from the queue, it is pre-empted by one of the sending tasks which then immediately re-fills the queue. The sending task then re-enters the Blocked state to wait for space to become available on the queue again.

Listing 39 shows the implementation of the sending task. The sending task specifies a block time of 100 milliseconds, so it enters the Blocked state to wait for space to become available each time the queue becomes full. It leaves 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 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.\n" );
        }

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

Listing 39. The implementation of the sending task for Example 11.

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

The implementation of the receiving task is shown in Listing 40.

```
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!\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.\n" );
        }
    }
}
```

Listing 40. 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 41.

```
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", 240, &( xStructsToSend[ 0 ] ), 2, NULL );
        xTaskCreate( vSenderTask, "Sender2", 240, &( 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", 240, 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 41. 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 23.

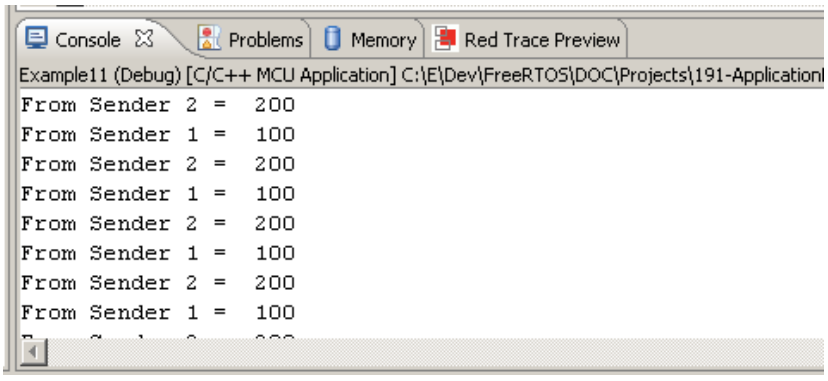


Figure 23. The output produced by Example 11

Figure 24 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 24 is provided in Table 13.

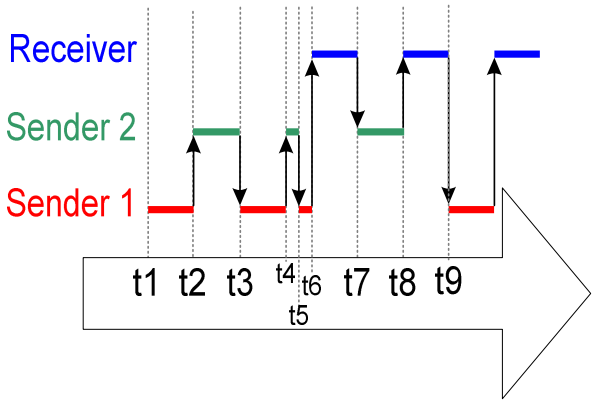


Figure 24. The sequence of execution produced by Example 11

Table 12. Key to Figure 24

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.

Table 12. Key to Figure 24

Time	Description
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 task 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 allow the chosen strategy to be implemented in a simple and maintainable way.

Note that only API functions and macros ending in ‘FromISR’ or ‘FROM_ISR’ should 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 of the Cortex-M3 FreeRTOS port.

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 that the handler task always pre-empts the other tasks in the system. The ISR can then be implemented to include a context switch to ensure that the ISR returns directly to the handler task when the ISR itself has completed executing. This has the effect of ensuring that the entire event processing executes contiguously in time, just as if it had all been implemented within the ISR itself. This scheme is demonstrated in Figure 25.

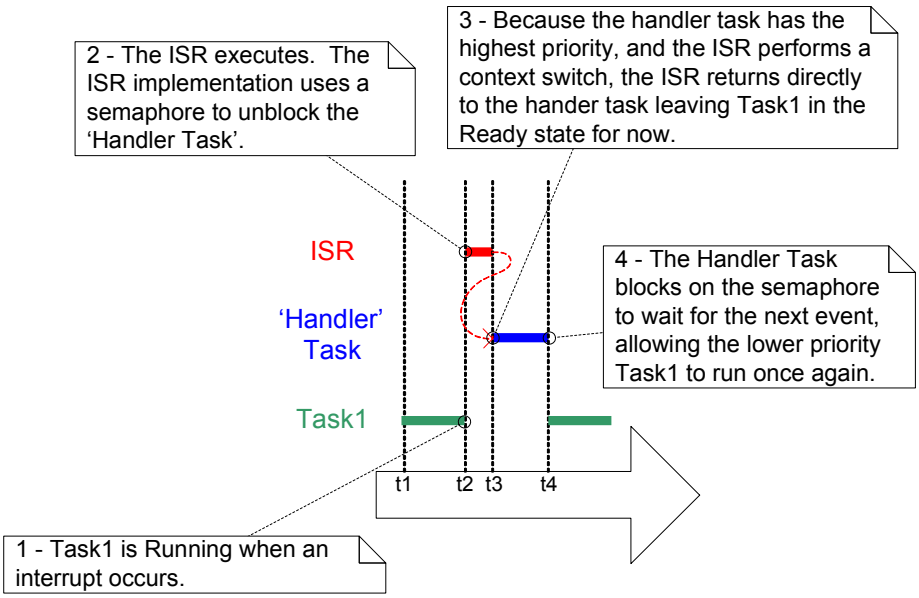


Figure 25. The interrupt interrupts one task but returns to another

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 that the required event processing can proceed.

'Taking a semaphore' and 'giving a semaphore' are concepts that have 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 binary semaphore can be considered conceptually as a queue with a length of one. The queue can contain a maximum of one item at any time, so is always either empty or full (hence, binary). By calling xSemaphoreTake(), the 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 uses the xSemaphoreGiveFromISR() function to place a token (the semaphore) into the queue, making the queue full. This causes the handler task to exit the Blocked state and remove the token, leaving the queue empty once more. When the handler task has completed its processing, it once more attempts to read from the queue and, finding the queue empty, re-enters the Blocked state to wait for the next event. This sequence is demonstrated in Figure 26.

Figure 26 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 being conceptually 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.

Writing FreeRTOS Interrupt Handlers

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

The vSemaphoreCreateBinary() API Function

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

Before a semaphore can be used, it must be created. To create a binary semaphore, use the vSemaphoreCreateBinary() API function[1].

[1] The Semaphore API is actually implemented as a set of macros, not functions. For simplicity, they are referred to as functions throughout this book.

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

Listing 42. The vSemaphoreCreateBinary() API function prototype

Table 13. 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.

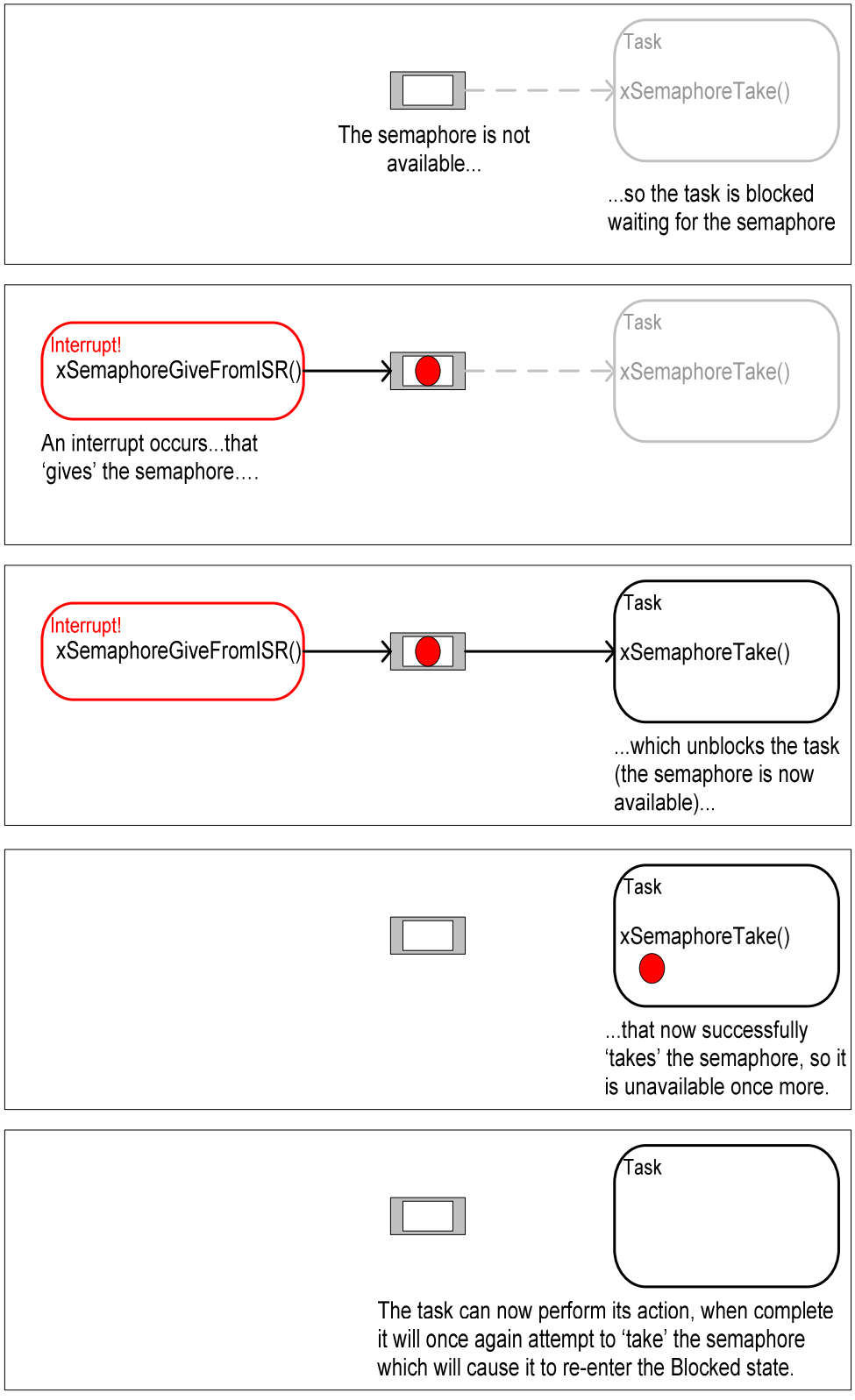


Figure 26. Using a binary semaphore to synchronize a task with an interrupt

The xSemaphoreTake() API Function

'Taking' a semaphore means to 'obtain' or 'receive' the semaphore. The semaphore can be taken only 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 43. The xSemaphoreTake() API function prototype

Table 14. 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. It must be explicitly created before it can be used.
xTicksToWait	The maximum amount of time the task should remain in the Blocked state to wait for the semaphore, if it is not already available. If xTicksToWait is zero, then xSemaphoreTake() will return immediately, if the semaphore is not available. The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The 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.

Table 14. xSemaphoreTake() parameters and return value

Parameter Name/ Returned Value	Description
Returned value	There are two possible return values: 1. pdPASS pdPASS is returned only if the call to xSemaphoreTake() was successful in obtaining the semaphore. If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed 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 is not available. If a block time was specified (xTicksToWait was not zero), then the calling task will have been placed into the Blocked state to wait for the semaphore to become available, but the block time expired before this happened.

The 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 44. The xSemaphoreGiveFromISR() API function prototype

Table 15. 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 or equal to the currently executing task (the task that was interrupted), then, internally, xSemaphoreGiveFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE. If xSemaphoreGiveFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt is exited. This will ensure that the interrupt returns directly to the highest priority Ready state task.
Returned value	There are two possible return values: 1. pdPASS pdPASS will be returned only if the call to xSemaphoreGiveFromISR() is successful. 2. 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 an interrupt every 500 milliseconds. In this case, a software generated interrupt is used because it allows the time at which the interrupt occurs to be controlled, which in turn allows the sequence of execution to be observed more easily. Listing 45 shows the implementation of the periodic task. mainTRIGGER_INTERRUPT() simply sets a bit in the interrupt controller's Set Pending register.

```
static void vPeriodicTask( void *pvParameters )
{
    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* This task is just used to 'simulate' an interrupt.  This is done by
        periodically generating a software interrupt. */
        vTaskDelay( 500 / portTICK_RATE_MS );

        /* Generate the interrupt, printing a message both before hand and
        afterwards so the sequence of execution is evident from the output. */
        vPrintString( "Periodic task - About to generate an interrupt.\n" );
        mainTRIGGER_INTERRUPT();
        vPrintString( "Periodic task - Interrupt generated.\n\n" );
    }
}
```

Listing 45. Implementation of the task that periodically generates a software interrupt in Example 12

Listing 46 shows the implementation of the handler task—the task that is synchronized with the software interrupt through the use of a binary semaphore. 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.

```
static void vHandlerTask( void *pvParameters )
{
    /* As per most tasks, this task is implemented within an infinite loop.

    Take the semaphore once to start with so the semaphore is empty before the
    infinite loop is entered.  The semaphore was created before the scheduler
    was started so before this task ran for the first time.*/
    xSemaphoreTake( xBinarySemaphore, 0 );

    for( ;; )
    {
        /* Use the semaphore to wait for the event.  The task blocks
        indefinitely meaning this function call will only return once the
        semaphore has been successfully obtained - so there is no need to check
        the returned value. */
        xSemaphoreTake( xBinarySemaphore, portMAX_DELAY );

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

Listing 46. The implementation of the handler task (the task that synchronizes with the interrupt) in Example 12

Listing 47 shows the interrupt service routine, which is simply a standard C function. It does very little other than clear the interrupt and ‘give’ the semaphore to unblock the handler task.

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

Note how xHigherPriorityTaskWoken is used. It is initialized to pdFALSE before being passed by reference into xSemaphoreGiveFromISR(), where it will get set to pdTRUE only if xSemaphoreGiveFromISR() causes a task of equal or higher priority than the currently executing task to leave the blocked state. portEND_SWITCHING_ISR() then performs a context switch only if xHigherPriorityTaskWoken equals pdTRUE. In all other cases, a context switch is not necessary, because the task that was executing before the interrupt occurs will still be the highest priority task that is able to run.

```
void vSoftwareInterruptHandler( void )
{
portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

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

    /* Clear the software interrupt bit using the interrupt controllers
    Clear Pending register. */
    mainCLEAR_INTERRUPT();

    /* Giving the semaphore may have unblocked a task - if it did and the
    unblocked task has a priority equal to or above the currently executing
    task then xHigherPriorityTaskWoken will have been set to pdTRUE and
    portEND_SWITCHING_ISR() will force a context switch to the newly unblocked
    higher priority task.

    NOTE: The syntax for forcing a context switch within an ISR varies between
    FreeRTOS ports.  The portEND_SWITCHING_ISR() macro is provided as part of
    the Corte M3 port layer for this purpose.  taskYIELD() must never be called
    from an ISR! */
    portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
}
```

Listing 47. The software interrupt handler used in Example 12

The main() function creates the binary semaphore and the tasks, configures the software interrupt, and starts the scheduler. The implementation is shown in Listing 48.

```
int main( void )
{
    /* Configure both the hardware and the debug interface. */
    vSetupEnvironment();

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

    /* Check the semaphore was created successfully. */
    if( xBinarySemaphore != NULL )
    {
        /* Enable the software interrupt and set its priority. */
        prvSetupSoftwareInterrupt();

        /* 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", 240, NULL, 3, NULL );

        /* Create the task that will periodically generate a software interrupt.
        This is created with a priority below the handler task to ensure it will
        get preempted each time the handler task exits the Blocked state. */
        xTaskCreate( vPeriodicTask, "Periodic", 240, NULL, 1, NULL );

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

    /* If all is well we will never reach here as the scheduler will now be
    running the tasks.  If we do reach here then it is likely that there was
    insufficient heap memory available for a resource to be created. */
    for( ;; );
}
```

Listing 48. The implementation of main() for Example 12

Example 12 produces the output shown in Figure 27. As expected, the handler task executes as soon as 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 28.

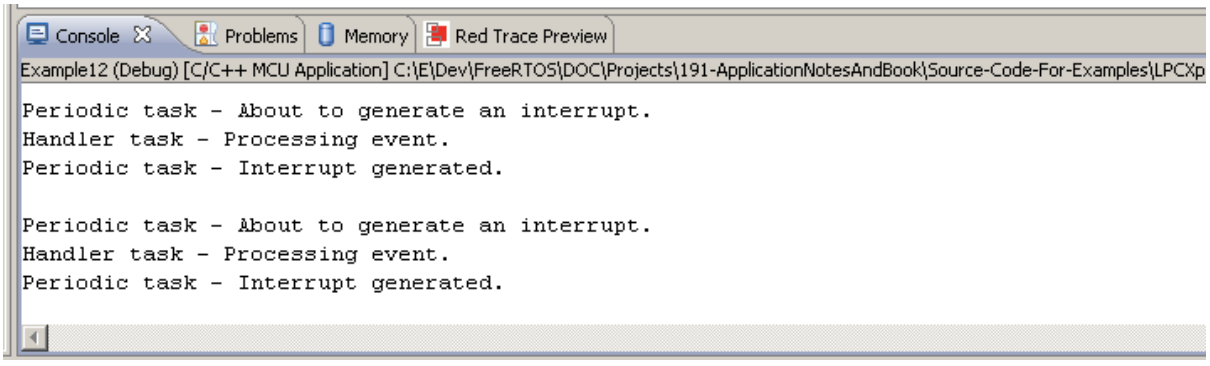


Figure 27. The output produced when Example 12 is executed

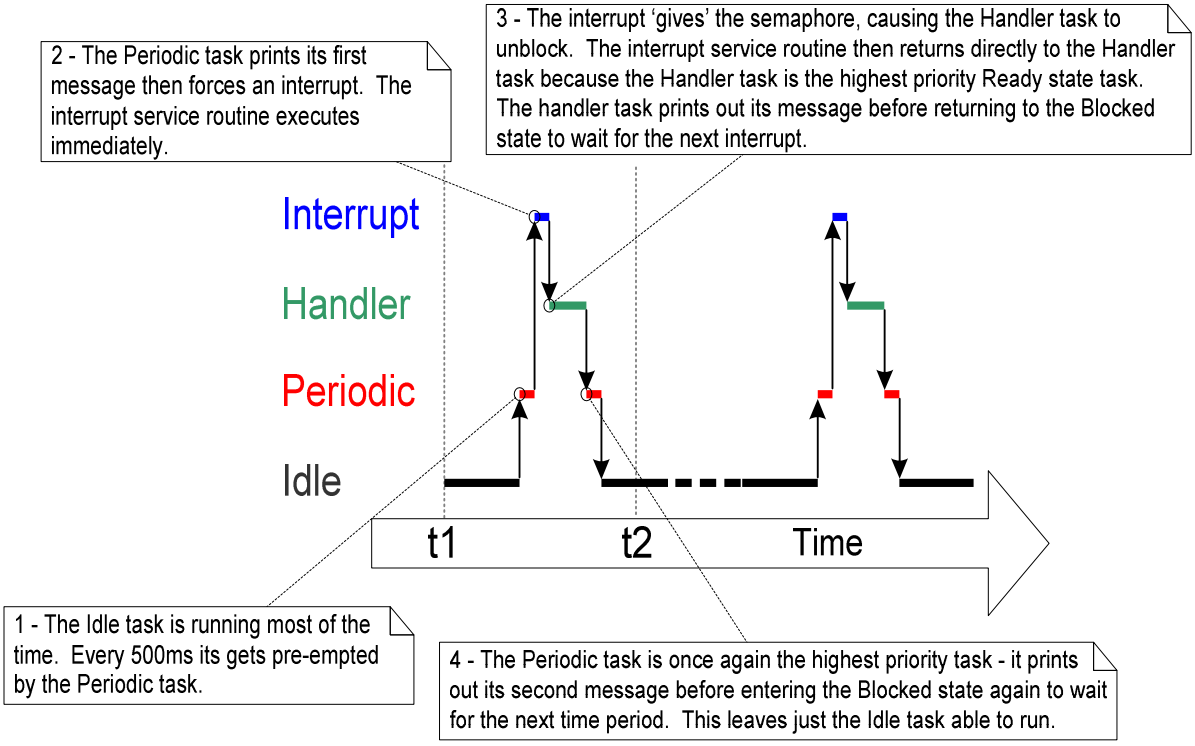


Figure 28. The sequence of execution when Example 12 is executed

3.3 Counting Semaphores

Example 12 demonstrates a binary semaphore being used to synchronize a task with an interrupt. The execution sequence is as follows.

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

This sequence is perfectly adequate if interrupts can occur only at a relatively low frequency. If another interrupt occurs before the Handler task has completed its processing of the first interrupt, then the binary semaphore will effectively latch the event, allowing the Handler task to process the new event immediately after it has completed processing the original event. The handler task will 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 29.

Figure 29 demonstrates that a binary semaphore can latch, at most, one interrupt event. Any subsequent events, occurring before the latched event has been processed, will be lost. This scenario can be avoided by using a counting semaphore in place of the binary semaphore.

Just as binary semaphores can be thought of as queues having a length of one, so counting semaphores can be thought of as queues having 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 semaphore's 'count' value.

configUSE_COUNTING_SEMAPHORES must be set to 1 in FreeRTOSConfig.h for counting semaphores to be available.

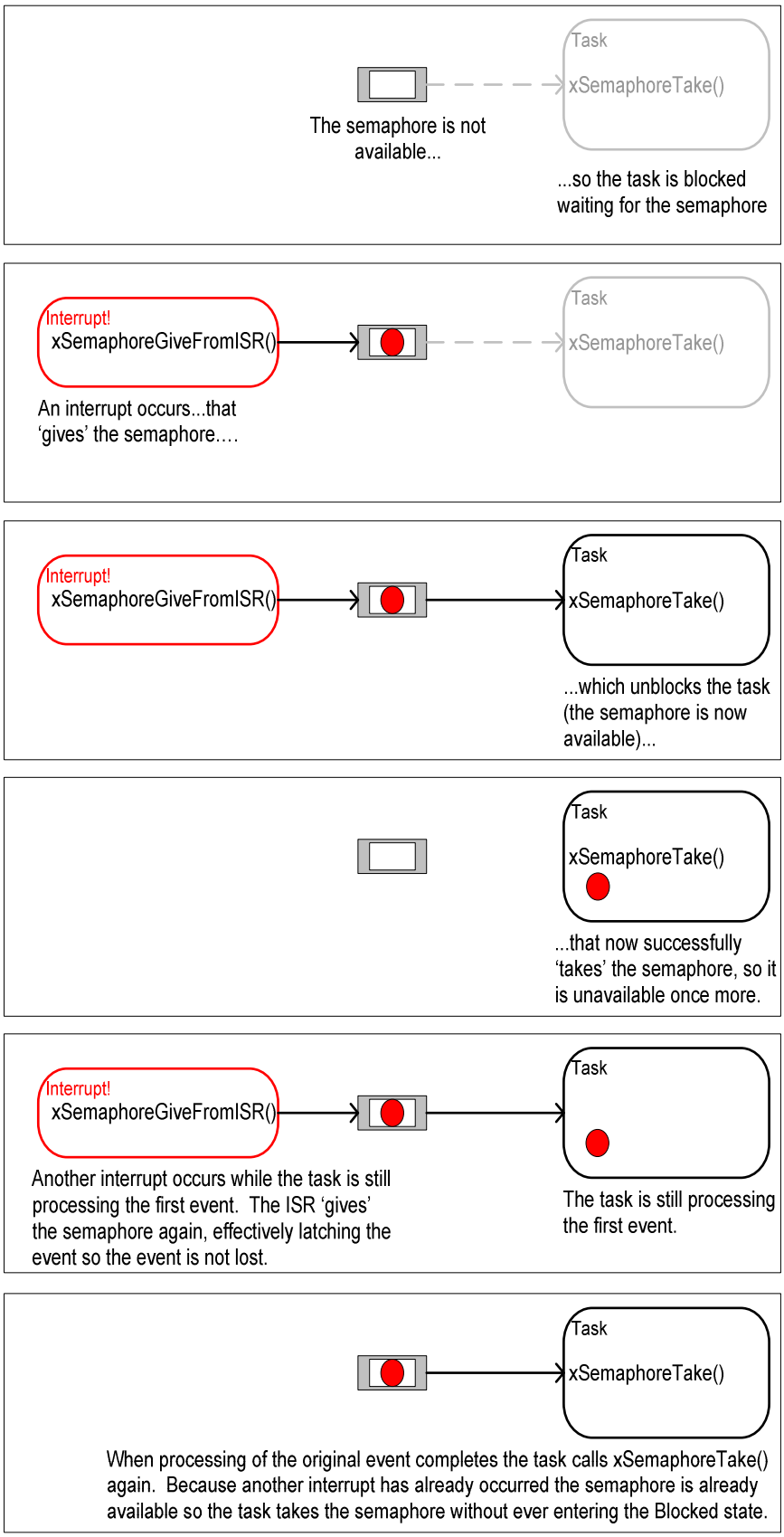


Figure 29. A binary semaphore can latch at most one event

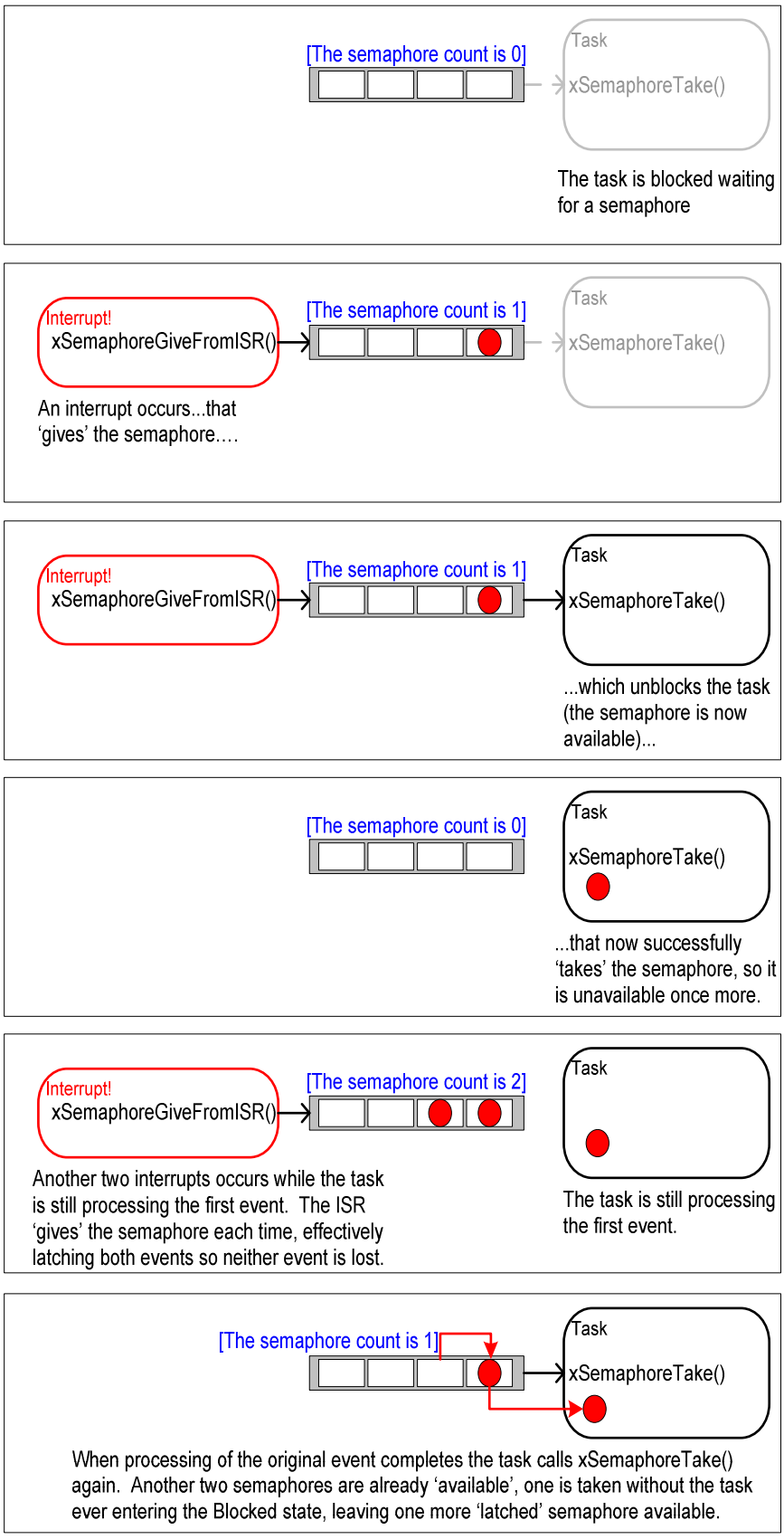


Figure 30. Using a counting semaphore to 'count' events

Counting semaphores are typically used for two things:

1. Counting events.

In this scenario, an event handler will 'give' a semaphore each time an event occurs—causing the semaphore's count value to be incremented on each 'give'. A handler task will 'take' a semaphore each time it processes an event—causing the semaphore's count value to be decremented on each take. The count value is the difference between the number of events that have occurred and the number that have been processed. This mechanism is shown in Figure 30.

Counting semaphores that are used 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 semaphore's count value. When the count value reaches zero, there are no free resources. When a task finishes with the resource, it 'gives' the semaphore back—incrementing the semaphore's count value.

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

The xSemaphoreCreateCounting() API Function

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

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

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

Listing 49. The xSemaphoreCreateCounting() API function prototype

Table 16. 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 to be used to count or latch events, uxMaxCount is the maximum number of events that can be latched. When the semaphore is to be used to manage access to a collection of resources, uxMaxCount should be set to the total number of resources that are available.
uxInitialCount	The initial count value of the semaphore after it has been created. When the semaphore is to be used to count or latch events, uxInitialCount should be set to zero—as, presumably, when the semaphore is created, no events have yet occurred. When the semaphore is to be used to manage access to a collection of resources, uxInitialCount should be set to equal uxMaxCount—as, presumably, when the semaphore is created, all the resources are available.
Returned value	If NULL is returned, the semaphore cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the semaphore data structures. Chapter 5 provides more information on memory management. A non-NULL value being returned indicates that the semaphore has been created successfully. The returned value should be stored as the handle to the created semaphore.

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 50.

```
    /* 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 50. 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 semaphore's count value. The interrupt service routine is shown in Listing 51.

```
void vSoftwareInterruptHandler( void )
{
portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

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

    /* Clear the software interrupt bit using the interrupt controllers Clear
    Pending register. */
    mainCLEAR_INTERRUPT();

    /* Giving the semaphore may have unblocked a task - if it did and the
    unblocked task has a priority equal to or above the currently executing
    task then xHigherPriorityTaskWoken will have been set to pdTRUE and
    portEND_SWITCHING_ISR() will force a context switch to the newly unblocked
    higher priority task.

    NOTE: The syntax for forcing a context switch within an ISR varies between
    FreeRTOS ports.  The portEND_SWITCHING_ISR() macro is provided as part of
    the Cortex-M3 port layer for this purpose.  taskYIELD() must never be called
    from an ISR! */
    portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
}
```

Listing 51. 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 31. As can be seen, the Handler task processes all three [simulated] events each time an interrupt is generated. The events are latched into the count value of the semaphore, allowing the Handler task to process them in turn.

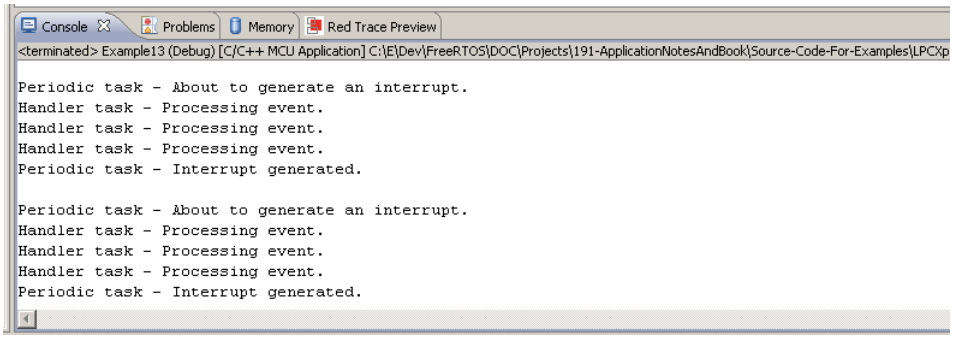


Figure 31. The output produced when Example 13 is executed

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 communicate events and to transfer data.

The 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 52. The xQueueSendToFrontFromISR() API function prototype

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

Listing 53. The xQueueSendToBackFromISR() API function prototype

Table 17. 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.

Table 17. xQueueSendToFrontFromISR() and xQueueSendToBackFromISR() parameters and return values

Parameter Name/ Returned Value	Description
pvItemToQueue	A pointer to the data to be copied into the queue. The size of each item that the queue can hold is set when the queue is created, so this number of 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 equal to or higher than the currently executing task (the task that was interrupted), then, internally, the API function will set *pxHigherPriorityTaskWoken to pdTRUE. If xQueueSendToFrontFromISR() or xQueueSendToBackFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt is exited. This will ensure that the interrupt returns directly to the highest priority Ready state task.
Returned value	There are two possible return values: 1. pdPASS pdPASS is returned only if data has been sent successfully to the queue. 2. errQUEUE_FULL errQUEUE_FULL is returned if data cannot be sent to the queue because the queue is already full.

Efficient Queue Usage

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

- Interpreting the received characters directly within the interrupt service routine, then using a queue to send the interpreted and decoded commands to a task for processing (in a similar manner to that shown in Figure 22). This technique is suitable only 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 54.

```
static void vIntegerGenerator( void *pvParameters )
{
portTickType xLastExecutionTime;
unsigned long 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.\n" );
        mainTRIGGER_INTERRUPT();
        vPrintString( "Generator task - Interrupt generated.\n\n" );
    }
}
```

Listing 54. The implementation of the task that writes to the queue in Example 14

The interrupt service routine calls xQueueReceiveFromISR() repeatedly, 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 55.

```
void vSoftwareInterruptHandler( void )
{
portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
static unsigned long ulReceivedNumber;

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

    /* 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 the string that corresponds to the truncated value to the other
        queue. */
        ulReceivedNumber &= 0x03;
        xQueueSendToBackFromISR( xStringQueue,
                                 &pcStrings[ ulReceivedNumber ],
                                 &xHigherPriorityTaskWoken );
    }

    /* Clear the software interrupt bit using the interrupt controllers Clear
    Pending register. */
    mainCLEAR_INTERRUPT();

    /* xHigherPriorityTaskWoken was initialised to pdFALSE.  It will have then
    been set to pdTRUE only if reading from or writing to a queue caused a task
    of equal or greater priority than the currently executing task to leave the
    Blocked state.  When this is the case a context switch should be performed.
    In all other cases a context switch is not necessary.

    NOTE: The syntax for forcing a context switch within an ISR varies between
    FreeRTOS ports.  The portEND_SWITCHING_ISR() macro is provided as part of
    the Cortex-M3 port layer for this purpose.  taskYIELD() must never be called
    from an ISR! */
    portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
}
```

Listing 55. The implementation of the interrupt service routine used by Example 14

The task that receives the character pointers from the interrupt service routine blocks on the queue until a message arrives, printing out each string as it is received. Its implementation is shown in Listing 56.

```
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 56. 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 57.

```
int main( void )
{
    /* Before a queue can be used it must first be created.  Create both queues
    used by this example.  One queue can hold variables of type unsigned long,
    the other queue can hold variables of type char*.  Both queues can hold a
    maximum of 10 items.  A real application should check the return values to
    ensure the queues have been successfully created. */
    xIntegerQueue = xQueueCreate( 10, sizeof( unsigned long ) );
    xStringQueue = xQueueCreate( 10, sizeof( char * ) );

    /* Enable the software interrupt and set its priority. */
    prvSetupSoftwareInterrupt();

    /* Create the task that uses a queue to pass integers to the interrupt service
    routine.  The task is created at priority 1. */
    xTaskCreate( vIntegerGenerator, "IntGen", 240, NULL, 1, NULL );

    /* Create the task that prints out the strings sent to it from the interrupt
    service routine.  This task is created at the higher priority of 2. */
    xTaskCreate( vStringPrinter, "String", 240, NULL, 2, NULL );

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

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks.  If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    Chapter 5 provides more information on memory management. */
    for( ;; );
}
```

Listing 57. The main() function for Example 14

The output produced when Example 14 is executed is shown in Figure 32. As can be seen, the interrupt receives all five integers and produces five strings in response. More explanation is given in Figure 33.

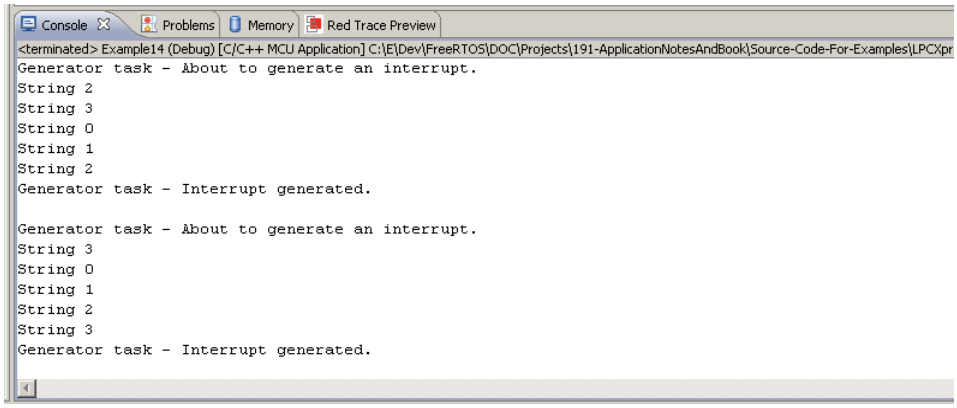


Figure 32. The output produced when Example 14 is executed

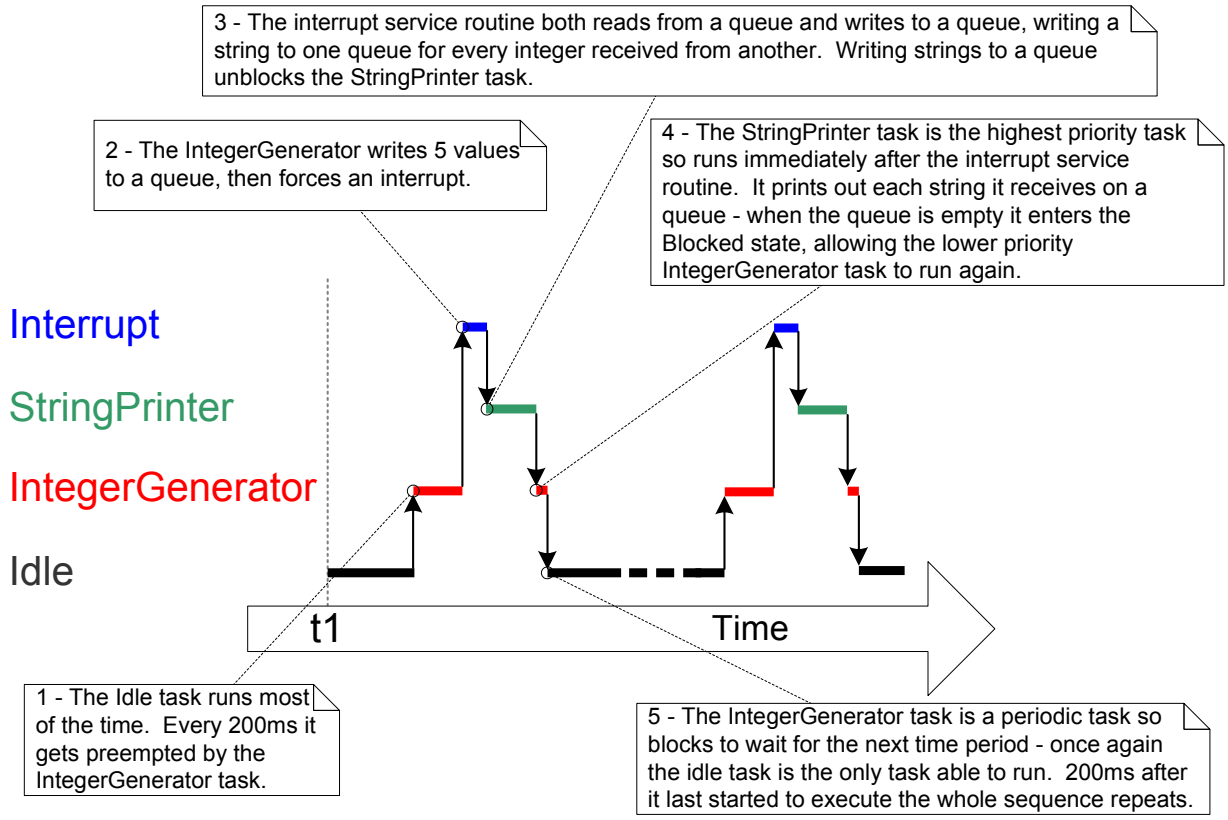


Figure 33. The sequence of execution produced by Example 14

3.5 Interrupt Nesting

Interrupt nesting behavior is defined by the configuration constants documented in Table 18. Both constants are defined in FreeRTOSConfig.h.

The Cortex-M3 core can use up to eight bits to specify an interrupt priority—allowing a maximum of 256 different priorities. It is essential to note that high numeric values denote low interrupt priorities. Therefore, if all eight bits are used, the lowest possible interrupt priority is 255 and the highest interrupt priority is zero. However, most Cortex-M3 implementations do not use all eight bits. To confuse matters further, for reasons of forward compatibility it is the most significant bits that are used. Therefore, if five bits are used then the lowest possible interrupt priority is 248, or 11111000 in binary, and the highest is zero. If three bits are used then the lowest possible interrupt priority is 224, or 11100000 in binary, and the highest is zero.

There are two ways to simplify the specification of interrupt priority values:

1. Think of the priorities as if the implemented bits are the least significant bits instead of the most significant bits. Then shift the priority left by the number of unimplemented priority bits before writing to the priority registers in the interrupt controller.

   For example, if five priority bits are used the lowest priority is 31, or 00011111 in binary. The highest priority is zero. If you assign an interrupt a priority of 29, the value written to the interrupt controller is (29 << 3). The priority value is shifted by three because there are five implemented priority bits so three unimplemented priority bits.

   As another example, if three priority bits are used the lowest priority is 7, or 00000111 in binary. The highest priority is zero. If you assign an interrupt a priority of 4, the value written to the interrupt controller is (4 << 5). The priority value is shifted by five because there are three implemented priority bits so five unimplemented priority bits.

2. Use the functions provided by the Cortex Microcontroller Software Interface Standard (CMSIS) to access the interrupt controller. CMSIS functions allow priorities to be specified as if they are implemented using the least significant bits but without having to perform the shift by three. For example, the watchdog interrupt priority can be set to 5 using the CMSIS call shown in Listing 58.

```
/* Using the CMSIS NVIC_SetPriority() function to set the watchdog timer
interrupt priority.  WDT_IRQn is defined in the CMSIS header file. */
NVIC_SetPriority( WDT_IRQn, 5 );
```

Listing 58. Using a CMSIS function to set an interrupt priority

Cortex-M3 microcontroller vendors will often provide vendor specific functions that are equivalent to the CMSIS NVIC_SetPriority() function. Extra care must be taken when using a vendor specific function as some expect their input parameters to use the most significant bit positions, while others expect their input parameters to use the least significant bit positions.

Table 18. Constants that affect interrupt nesting

Constant	Description
configKERNEL_INTERRUPT_PRIORITY	Sets the priority of interrupts used by the kernel itself—namely the timer interrupt used to generate the tick and the PendSV (Pend Service Call) interrupt used within the API. configKERNEL_INTERRUPT_PRIORITY will almost always be set to the lowest possible interrupt priority.
configMAX_SYSCALL_INTERRUPT_PRIORITY	Defines the highest interrupt priority from which FreeRTOS API functions can be called. Only API functions that end in 'FromISR' can be called from within an interrupt.

Full interrupt nesting functionality is achieved by setting configMAX_SYSCALL_INTERRUPT_PRIORITY to a higher interrupt priority (meaning a lower numeric priority value) than configKERNEL_INTERRUPT_PRIORITY. This is demonstrated in Figure 34, which shows a scenario where configKERNEL_INTERRUPT_PRIORITY has been set to 31 and configMAX_SYSCALL_INTERRUPT_PRIORITY has been set to 29[2]. For simplicity, these priority values have been written in the range expected by the CMSIS functions, so 31 is the lowest possible interrupt priority and 29 is two priorities above the

[2] The number 31 and 29 assume the microcontroller being used implements at least five interrupt priority bits. These numbers would be invalid on a microcontroller that implements three priority bits.

lowest. The definitions of configKERNEL_INTERRUPT_PRIORITY and configMAX_SYSCALL_INTERRUPT_PRIORITY within FreeRTOSConfig.h actually require the real values, so (31 << 3) and (29 << 3) in this example.

The default priority for all interrupts is zero—the highest possible priority value. If an interrupt uses a FreeRTOS API function, then its priority must never be left uninitialized, unless configMAX_SYSCALL_INTERRUPT_PRIORITY is also set to zero.

It is common for confusion to arise between task priorities and interrupt priorities. Figure 34 shows interrupt priorities, as defined by the microcontroller architecture. These are the hardware controlled priorities at which interrupt service routines execute 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.

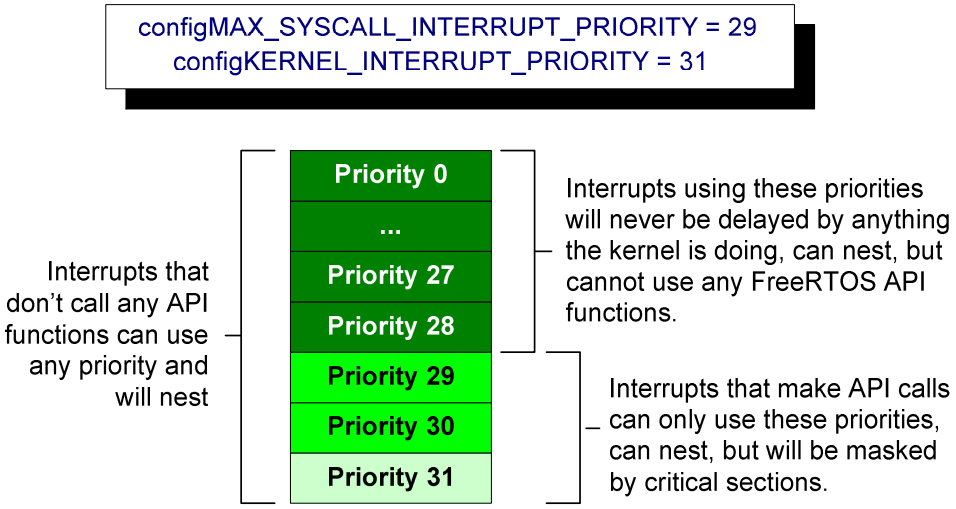


Figure 34. Constants affecting interrupt nesting behavior – this illustration assumes the microcontroller being used implements at least five interrupt priority bits

Referring to Figure 34:

- Interrupts that use priorities 31 to 29, inclusive, are prevented from executing while the kernel or the application is inside a critical section. They can, however, make use of any API function ending in 'FromISR'.

- Interrupts that use priorities 28 to 0 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. Functionality that requires very strict timing accuracy (motor control, for example) would use a priority above configMAX_SYSCALL_INTERRUPT_PRIORITY to ensure that the scheduler does not

introduce jitter into the interrupt response times. Interrupts at these priority levels cannot use any FreeRTOS API functions.

Chapter 4

Resource Management

4.1 Chapter Introduction and Scope

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

Following are some examples:

1. Accessing Peripherals

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

   1. Task A executes and starts to write the string “Hello world” to the LCD.
   2. Task A is pre-empted by Task B after outputting just the beginning of the string—“Hello w”.
   3. Task B writes “Abort, Retry, Fail?” to the LCD before entering the Blocked state.
   4. Task A continues from the point at which it was pre-empted and completes outputting the remaining characters—“orld”.

   The LCD now displays the corrupted string “Hello wAbort, Retry, Fail?orld”.

2. Read, Modify, Write Operations

   Listing 59 shows a line of C code and its resultant assembly output. It can be seen that the value of GlobalVar 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. */
GlobalVar |= 0x01;

/* The assembly code produced. */
LDR      r4,[pc,#284]
LDR      r0,[r4,#0x08] /* Load the value of GlobalVar into r0. */
ORR      r0,r0,#0x01   /* Set bit 0 of r0. */
STR      r0,[r4,#0x08] /* Write the new r0 value back to GlobalVar. */
```

Listing 59. 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 variable called GlobalVar:

1. Task A loads the value of GlobalVar into a register—the read portion of the operation.

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

3. Task B updates the value of GlobalVar, then enters the Blocked state.

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

In this scenario, Task A updates and writes back an out-of-date value for GlobalVar. Task B modifies GlobalVar after Task A takes a copy of the GlobalVar value and before Task A writes its modified value back to the GlobalVar variable. When Task A writes to GlobalVar, it overwrites the modification that has already been performed by Task B, effectively corrupting the GlobalVar variable value.

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 64-bit variable on a 32-bit machine), are 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 stored on the stack or held in a register, then the function is reentrant. Listing 60 is an example of a reentrant function. Listing 61 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 the compiler and optimization level.  Each
task or interrupt that calls this function will have its own copy
of lVar2. */
long lVar2;

    lVar2 = lVar1 + 100;

    /* Most likely the return value will be placed in a CPU register,
    although it too could be placed on the stack. */
    return lVar2;
}
```

Listing 60. 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 61. An example of a function that is not reentrant

Mutual Exclusion

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

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

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 reduce (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 62. Critical sections are also known as critical regions.

```
    /* Ensure access to the GlobalVar variable cannot be interrupted by
    placing it within a critical section.  Enter the critical section. */
    taskENTER_CRITICAL();

    /* A switch to another task cannot occur between the call to
    taskENTER_CRITICAL() and the call to taskEXIT_CRITICAL().  Interrupts
    may still execute, but only interrupts whose priority is above the
    value assigned to the configMAX_SYSCALL_INTERRUPT_PRIORITY constant
    – and those interrupts are not permitted to call FreeRTOS API
    functions. */
    GlobalVar |= 0x01;

    /* Access to GlobalVar is complete so the critical section can be exited. */
    taskEXIT_CRITICAL();
```

Listing 62. Using a critical section to guard access to a variable

The example projects that accompany this book use a function called vPrintString() to write strings to standard out. 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 63.

```
void vPrintString( const char *pcString )
{
static char cBuffer[ ioMAX_MSG_LEN ];

    /* Write the string to stdout, using a critical section as a crude method
    of mutual exclusion. */
    taskENTER_CRITICAL();
    {
        sprintf( cBuffer, "%s", pcString );
        consoleprint( cBuffer );
    }
    taskEXIT_CRITICAL();
}
```

Listing 63. A possible implementation of vPrintString()

Critical sections implemented in this way are a very crude method of providing mutual exclusion. They work by disabling interrupts up to the interrupt priority set by

configMAX_SYSCALL_INTERRUPT_PRIORITY. Pre-emptive context switches can occur only from within an interrupt, so, as long as interrupts remain disabled, the task that called taskENTER_CRITICAL() is guaranteed to remain in the Running state until the critical section is exited.

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 (stdout, or the stream where a computer writes its output data) should not be protected using a critical section (as shown in Listing 63), because writing to the terminal can be a relatively long operation. 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 be exited only when the nesting depth returns to zero—which is when one call to taskEXIT_CRITICAL() has been executed for every preceding call to taskENTER_CRITICAL().

Suspending (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 protects a region of code only 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, However, resuming (or ‘un-suspending’) the scheduler can be a relatively long operation, so consideration must be given to which is the best method to use in each case.

The vTaskSuspendAll() API Function

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

Listing 64. The vTaskSuspendAll() API function prototype

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 performed only when the scheduler is resumed (un-suspended).

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

The xTaskResumeAll() API Function

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

Listing 65. The xTaskResumeAll() API function prototype

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

Table 19. xTaskResumeAll() return value

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

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

Listing 66 shows the actual implementation of vPrintString(), which suspends the scheduler to protect access to the terminal output.

```
void vPrintString( const char *pcString )
{
static char cBuffer[ ioMAX_MSG_LEN ];

    /* Write the string to stdout, suspending the scheduler as a method
    of mutual exclusion. */
    vTaskSuspendScheduler();
    {
        sprintf( cBuffer, "%s", pcString );
        consoleprint( cBuffer );
    }
    xTaskResumeScheduler();
}
```

Listing 66. 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 'MUTual EXclusion'.

When used in a mutual exclusion scenario, the mutex can be thought of as a token that is associated with the resource being shared. For a task to access the resource legitimately, it must first successfully 'take' the token (be the token holder). When the token holder has finished with the resource, it must 'give' the token back. Only when the token has been returned can another task successfully take the token and then safely access the same shared resource. A task is not permitted to access the shared resource unless it holds the token. This mechanism is shown in Figure 35.

Even though mutexes and binary semaphores share many characteristics, the scenario shown in Figure 35 (where a mutex is used for mutual exclusion) is completely different to that shown in Figure 29 (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.

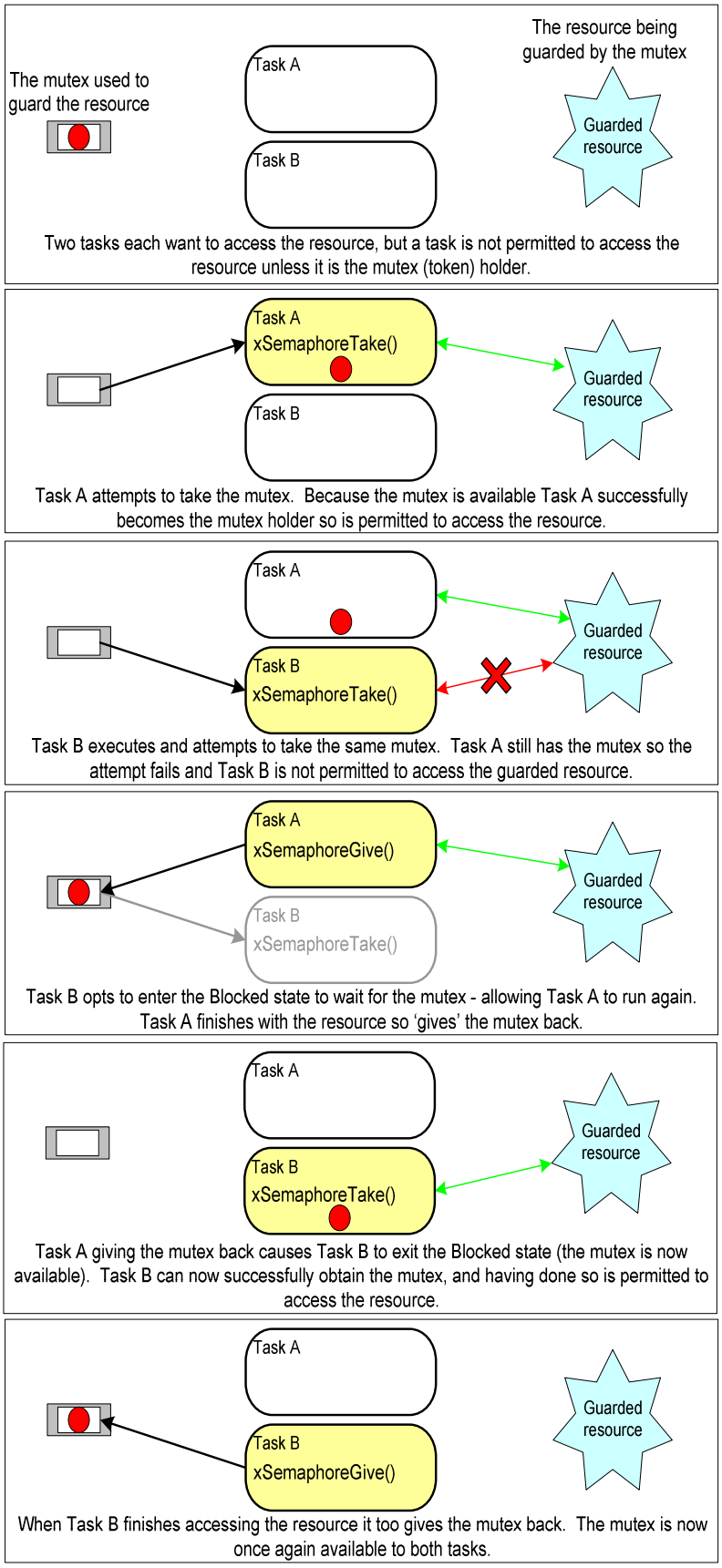


Figure 35. Mutual exclusion implemented using a mutex

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 do so, unless it is able to become the mutex holder.

The 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 be used, it must be created. To create a mutex type semaphore, use the xSemaphoreCreateMutex() API function.

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

Listing 67. The xSemaphoreCreateMutex() API function prototype

Table 20. xSemaphoreCreateMutex() return value

Parameter Name/ Returned Value	Description
Returned value	If NULL is returned, then the mutex could not be created because there is insufficient heap memory available for FreeRTOS to allocate the mutex data structures. Chapter 5 provides more information on memory management. A non-NULL return value indicates that the mutex has been 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() is functionally identical 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 68.

```
static void prvNewPrintString( const char *pcString )
{
static char cBuffer[ mainMAX_MSG_LEN ];

    /* The mutex is created before the scheduler is started so already
    exists by the time this task first executes.

    Attempt to take the mutex, blocking indefinitely to wait for the mutex if
    it is not available straight away.  The call to xSemaphoreTake() will only
    return when the mutex has been successfully obtained so there is no need to
    check the function return value.  If any other delay period was used then
    the code must check that xSemaphoreTake() returns pdTRUE before accessing
    the shared resource (which in this case is standard out). */
    xSemaphoreTake( xMutex, portMAX_DELAY );
    {
        /* The following line will only execute once the mutex has been
        successfully obtained.  Standard out can be accessed freely now as
        only one task can have the mutex at any one time. */
        sprintf( cBuffer, "%s", pcString );
        consoleprint( cBuffer );

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

Listing 68. The implementation of prvNewPrintString()

prvNewPrintString() is called repeatedly by two instances of a task implemented by 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 69.

```
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 69. The implementation of prvPrintTask() for Example 15

As normal, main() creates the mutex, creates the tasks, then starts the scheduler. The implementation is shown in Listing 70.

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 pre-emption occurs, 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 );

    /* Only create the tasks if the semaphore was created successfully. */
    if( xMutex != NULL )
    {
        /* Create two instances of the tasks that write to stdout.  The string
        they write is passed in as the task parameter.  The tasks are created
        at different priorities so some pre-emption will occur. */
        xTaskCreate( prvPrintTask, "Print1", 240,
                    "Task 1 ****************************************\n", 1, NULL );

        xTaskCreate( prvPrintTask, "Print2", 240,
                    "Task 2 ----------------------------------------\n", 2, NULL );

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

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks.  If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    Chapter 5 provides more information on memory management. */
    for( ;; );
}
```

Listing 70. The implementation of main() for Example 15

The output produced when Example 15 is executed is shown in Figure 36. A possible execution sequence is described in Figure 37.

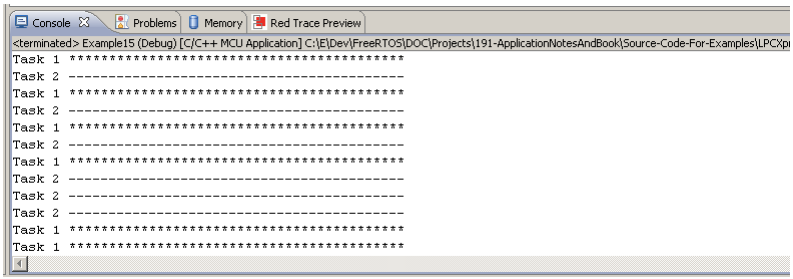


Figure 36. The output produced when Example 15 is executed

Figure 36 shows that, as expected, there is no corruption in the strings that are displayed in the terminal. The random ordering is a result of the random delay periods used by the tasks.

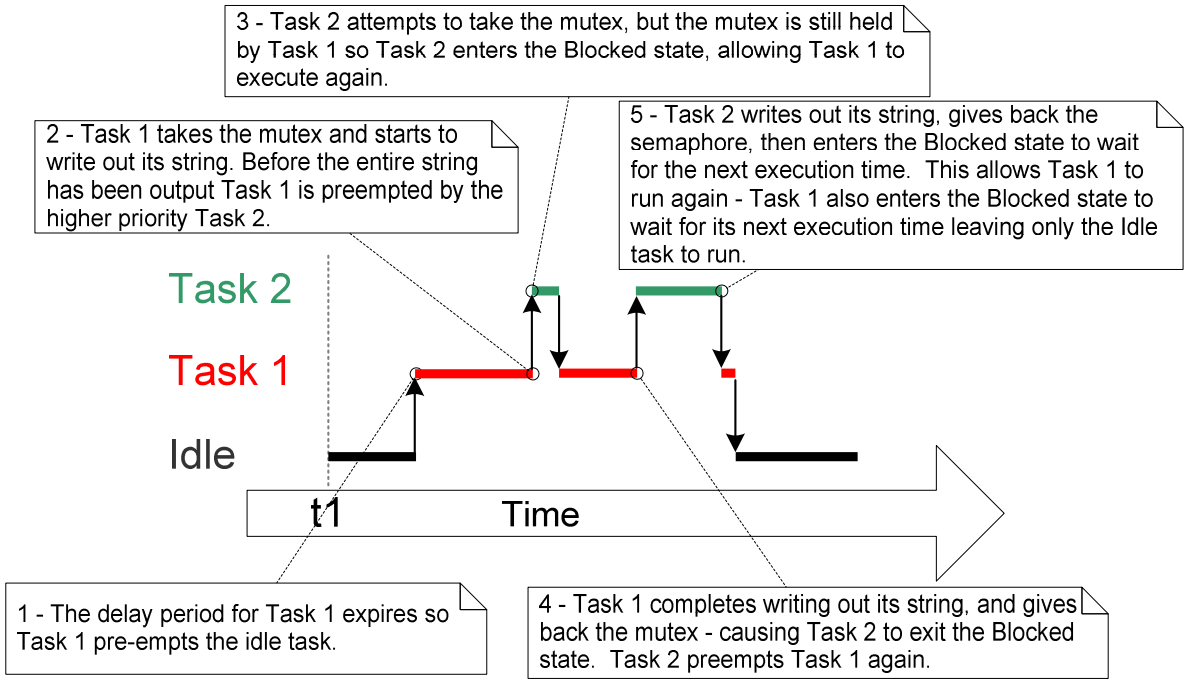


Figure 37. A possible sequence of execution for Example 15

Priority Inversion

Figure 37 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 38.

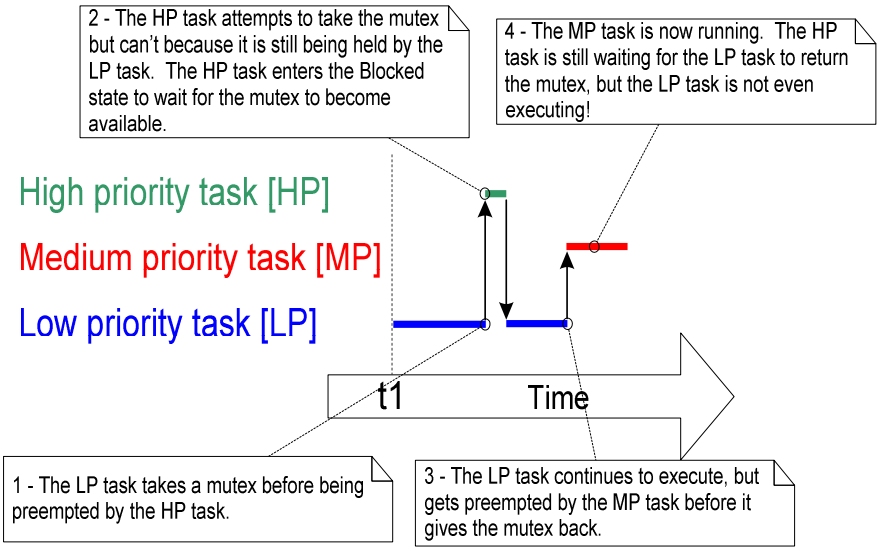


Figure 38. A worst case priority inversion scenario

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

Priority Inheritance

FreeRTOS mutexes and binary semaphores are very similar—the difference being that mutexes include a basic 'priority inheritance' mechanism, whereas binary semaphores do not. Priority inheritance is a scheme that minimizes the negative effects of priority inversion. It does not 'fix' priority inversion; it merely lessens its impact by ensuring that the inversion is always time bounded. However, priority inheritance complicates system timing analysis; it is not good practice to rely on it for correct system operation.

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 39. The priority of the mutex holder is reset automatically to its original value when it gives the mutex back.

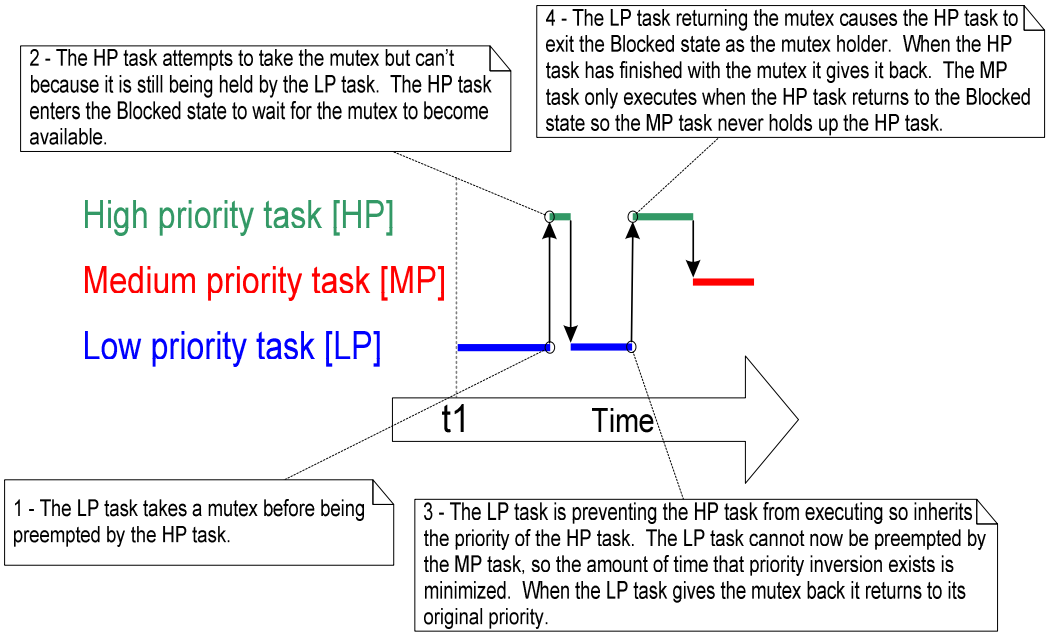


Figure 39. Priority inheritance minimizing the effect of priority inversion

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 hold only a single mutex at any one time.

Deadlock (or Deadly Embrace)

'Deadlock' is another potential pitfall that can occur when using mutexes for mutual exclusion. Deadlock is sometimes also known by the more dramatic name '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.

As with priority inversion, the best method of avoiding deadlock is to consider its potential at design time, and design the system to ensure that deadlock cannot occur. In practice, deadlock is not a big problem in small embedded systems, because the system designer can have a good understanding of the entire application, and so can 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 risk of priority inversion or deadlock.

A gatekeeper task is a task that has sole ownership of a resource. Only the gatekeeper task is allowed to access the resource directly—any other task requiring access to the resource can do so only indirectly by using the services of the gatekeeper.

Example 16. Re-writing vPrintString() to use a gatekeeper task

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

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

The gatekeeper task spends most of its time in the Blocked state, waiting for messages to arrive on the queue. When a message arrives, the gatekeeper writes the message to standard out, before returning to the Blocked state to wait for the next message. The implementation of the gatekeeper task is shown by Listing 72.

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

1. Set configUSE_TICK_HOOK to 1 in FreeRTOSConfig.h.

2. Provide the implementation of the hook function, using the exact function name and prototype shown in Listing 71.

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

Listing 71. The name and prototype for a tick hook function

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

```
static void prvStdioGatekeeperTask( void *pvParameters )
{
char *pcMessageToPrint;
static char cBuffer[ mainMAX_MSG_LEN ];

    /* This is the only task that is allowed to write to the terminal output.
    Any other task wanting to write a string to the output does not access the
    terminal directly, but instead sends the string to this task.  As only this
    task accesses standard out there are no mutual exclusion or serialization
    issues to consider within the implementation of the task itself. */
    for( ;; )
    {
        /* Wait for a message to arrive.  An indefinite block time is specified
        so there is no need to check the return value – the function will only
        return when a message has been successfully received. */
        xQueueReceive( xPrintQueue, &pcMessageToPrint, portMAX_DELAY );

        /* Output the received string. */
        sprintf( cBuffer, "%s", pcMessageToPrint );
        consoleprint( cBuffer );

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

Listing 72. The gatekeeper task

The task that prints out the message is similar to that used in Example 15, except that, here, the string is sent on the queue to the gatekeeper task, rather than written out directly. The implementation is shown in Listing 73. 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 73. The print task implementation for Example 16

The tick hook function counts the number of times it is called, sending its message to the gatekeeper task each time the count reaches 200. For demonstration purposes only, 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 74.

```
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 74. 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 75.

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

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

/* Declare a variable of type xQueueHandle.  This is used to send messages from
the print tasks and the tick interrupt to the gatekeeper task. */
xQueueHandle xPrintQueue;

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

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", 240, ( void * ) 0, 1, NULL );
        xTaskCreate( prvPrintTask, "Print2", 240, ( void * ) 1, 2, NULL );

        /* Create the gatekeeper task.  This is the only task that is permitted
        to directly access standard out. */
        xTaskCreate( prvStdioGatekeeperTask, "Gatekeeper", 240, NULL, 0, NULL );

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

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks.  If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    Chapter 5 provides more information on memory management. */
    for( ;; );
}
```

Listing 75. The implementation of main() for Example 16

The output produced when Example 16 is executed is shown in Figure 40. As can be seen, the strings originating from the tasks and the strings originating from the interrupt all print out correctly with no corruption.

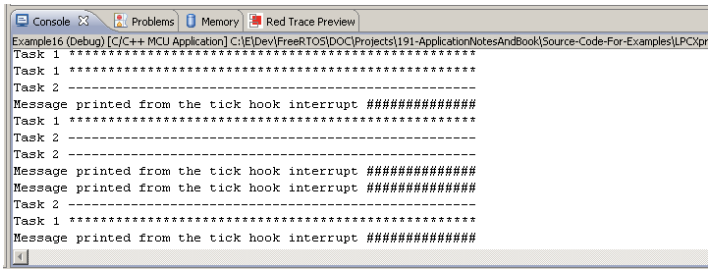


Figure 40. The output produced when Example 16 is executed

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

Chapter 5

Memory Management

5.1 Chapter Introduction and Scope

The kernel has to allocate RAM dynamically each time a task, queue, or semaphore is created. The standard malloc() and free() library functions can be used, but they may not be suitable or appropriate for one or more of the following reasons:

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

When the kernel requires RAM, instead of calling malloc() directly it calls pvPortMalloc(). When RAM is being freed, instead of calling free() directly, the kernel 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(); these examples are all 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—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 have been removed because of the effort and understanding required to dimension the blocks and pools.

It is common for small embedded systems only to create tasks, queues, and semaphores before the scheduler has been started. When this is the case, memory only gets dynamically allocated by the kernel before the application starts to perform any real-time functionality, and the memory remains allocated for the lifetime of the application. This means that the chosen allocation scheme does not have to consider any of the more complex issues such as determinism and fragmentation, and can instead consider only 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(). Applications that never delete a task, queue, or semaphore have the potential to use heap_1. Heap_1 is always deterministic.

The allocation scheme 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 been assigned.

Each created task requires a task control block (TCB) and a stack to be allocated from the heap. Figure 41 demonstrates how heap_1 subdivides the simple array as tasks are created.

Referring to Figure 41:

- 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.

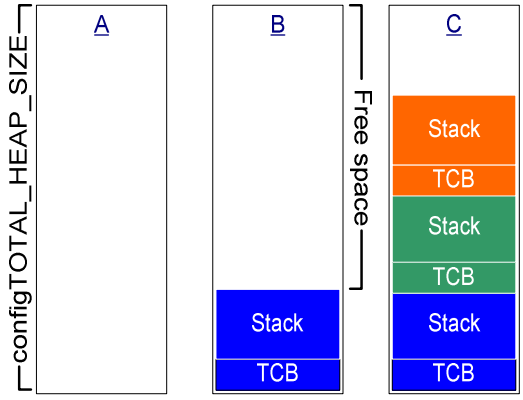


Figure 41. RAM being allocated within the array each time a task is created

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 declared statically, so will make the application appear to consume a lot of RAM, even before any of the array has been assigned.

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

- The heap contains three blocks of free memory that are 5 bytes, 25 bytes, and 100 bytes, respectively.
- pvPortMalloc() 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-byte block remains available to future calls to pvPortMalloc().

Heap_2.c does not combine adjacent free blocks into a single larger block, so it can suffer from fragmentation. However, fragmentation is not an issue if the blocks being allocated and subsequently freed are always the same size. Heap_2.c is suitable for an application that creates and deletes tasks repeatedly, provided the size of the stack allocated to the created tasks does not change.

[3] This is an oversimplification, 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.

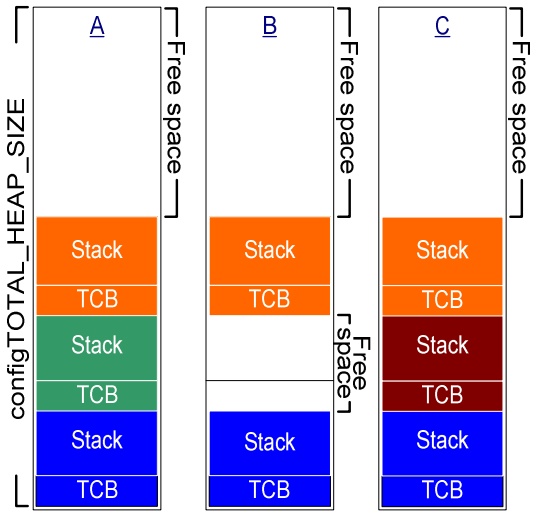


Figure 42. RAM being allocated from the array as tasks are created and deleted

Figure 42 demonstrates how the best fit algorithm works when a task is created, deleted, and then created again. Referring to Figure 42:

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

2. B 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.

3. C shows the situation after another task has been created. Creating the task has 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 ensures that the block of RAM previously allocated to the TCB of the deleted task is reused 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 ensures that the block of RAM previously allocated to the stack of the deleted task is reused 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 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 76.

The size of the heap is not affected by configTOTAL_HEAP_SIZE; instead, it is 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 76. The heap_3.c implementation

The xPortGetFreeHeapSize() API Function

xPortGetFreeHeapSize() is available only when heap_1.c or heap_2.c is being used. It provides a simple method of optimizing the heap size by returning the current number of unallocated bytes. For example, if xPortGetFreeHeapSize() returns 2000 after all the required tasks, queues, and semaphores have been created, then configTOTAL_HEAP_SIZE can be reduced by 2000.

```
size_t xPortGetFreeHeapSize( void );
```

Listing 77. The xPortGetFreeHeapSize() API function prototype

Table 21. xPortGetFreeHeapSize() return value

Parameter Name/ Returned Value	Description
Returned value	The number of bytes that remain unallocated in the heap.

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(), which 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 is owned by a third party. Therefore, 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.

The 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 close 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 78. The uxTaskGetStackHighWaterMark() API function prototype

Table 22. 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 used by the task grows and shrinks as the task executes and interrupts are processed. uxTaskGetStackHighWaterMark() returns the minimum amount of remaining stack space that has been available since the task started executing. This is the amount of stack that remains unused when stack usage is at its greatest (or deepest) value. The closer the high water mark is to zero, the closer the task has come to overflowing its stack.

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 increase the time it takes to perform a context switch.

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

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

2. Provide the implementation of the hook function, using the exact function name and prototype shown in Listing 79.

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

Listing 79. 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 to recover from a stack overflow when it occurs. The parameters pass the handle and name of the task (that has overflowed its stack) into the hook function; however, it is possible that the overflow has corrupted the task name.

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

Run Time Stack Checking—Method 1

Method 1 is selected when configCHECK_FOR_STACK_OVERFLOW is set to 1.

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

Method 1 is quick to execute but can miss stack overflows that occur between context 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 tests the last valid 20 bytes of the task stack space to verify that this pattern has not been overwritten. The stack overflow hook function is called if any of the 20 bytes have changed from their expected values.

Method 2 is not as quick to execute as method 1 but is still relatively fast, as only 20 bytes are tested. Most likely, it will catch all stack overflows; however, it is possible (but highly improbable) that some overflows will be missed.

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 created automatically when vTaskStartScheduler() is called. vTaskStartScheduler() will return only 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()'. In particular, do not attempt to create a critical section within an interrupt.

Do not use any API functions from an interrupt that has been assigned an interrupt priority above configMAX_SYSCALL_INTERRUPT_PRIORITY. Remember that interrupt priorities above configMAX_SYSCALL_INTERRUPT_PRIORITY are those that have a numeric value lower than configMAX_SYSCALL_INTERRUPT_PRIORITY. This can seem counter-intuitive and is a very common source of errors.

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. Therefore, the second thing to check is that the syntax, macros, and calling conventions used in the interrupt service routine are exactly as described on the documentation page for the demo, and exactly as demonstrated by other interrupt service routines in the demo.

Do not use any API functions from an interrupt that has been assigned an interrupt priority above configMAX_SYSCALL_INTERRUPT_PRIORITY. Remember that interrupt priorities above configMAX_SYSCALL_INTERRUPT_PRIORITY are those that have a numeric value lower than configMAX_SYSCALL_INTERRUPT_PRIORITY. This can seem counter-intuitive and is a very common source of errors.

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 become enabled again only 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 applies to 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

Check the project options to ensure that either the pre-processor macro required to include the correct portmacro.h file within portable.h is defined, or the include search path includes the path to the correct portmacro.h file.

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

Chapter 7

FreeRTOS-MPU

7.1 Chapter Introduction and Scope

Most Cortex-M3 microcontrollers include a Memory Protection Unit (MPU). This allows the entire memory map (including Flash, RAM, and peripherals) to be sub-divided into a number of regions, and access permissions to be assigned to each region, individually. A region is an address range consisting of a start address and a size.

FreeRTOS-MPU is a FreeRTOS Cortex-M3 port that includes integrated MPU support. It permits additional functionality and includes a slightly extended API, but is otherwise backward compatible with the standard Cortex-M3 port.

Using FreeRTOS-MPU will always:

- Protect the kernel from invalid execution by tasks.
- Protect the data used by the kernel from invalid access by tasks.
- Protect the configuration of Cortex-M3 core resources, such as the SysTick timer.
- Guarantee that all task stack overflows are detected as soon as they occur.

Also, at the application level, it is possible to ensure that tasks are isolated in their own memory space and that peripherals are protected from unintended modification.

FreeRTOS-MPU provides a simple interface to the MPU by hiding the register level MPU configuration from the user. However, writing an application for an environment that does not permit free access to all data can be challenging.

Scope

This chapter aims to give readers a good understanding of:

- The constraints the MPU hardware places on how memory regions can be defined.
- The access permissions that can be assigned to each memory region.
- The difference between User Mode tasks and Privileged Mode tasks.
- The FreeRTOS-MPU specific API.

7.2 Access Permissions

User Mode and Privileged Mode

The Cortex-M3 can execute code in either Privileged mode or User (unprivileged) mode. The standard FreeRTOS Cortex-M3 port executes all tasks in Privileged mode. FreeRTOS-MPU can execute tasks in either Privileged mode or User mode. The processor switches automatically to Privileged mode before executing an interrupt service routine. The kernel always switches to Privileged mode whenever a FreeRTOS-MPU API function is called, returning to its previous mode when the API function completes.

Tasks that execute in Privileged mode are not prevented from accessing any part of the Cortex-M3 core or from executing any of the Cortex-M3 instructions. MPU region access permissions can be used to prevent a Privileged mode task from making certain memory accesses—for example, writes to a region that is configured as read-only.

Tasks that execute in User mode are prevented from accessing certain Cortex-M3 resources and from executing certain Cortex-M3 instructions. For example, a User mode task cannot access the interrupt controller or execute CPS (Change Processor State) instructions[4]. MPU regions can be configured to prevent User mode access, while still permitting Privileged mode access.

Access Permission Attributes

Table 23 lists the access permission related definitions available in FreeRTOS-MPU. Examples of their use are provided later in this chapter.

[4] For complete details on User mode restrictions, refer to the 'ARM V7-M Architecture Application Level Reference Manual', and the 'Cortex-M3 Technical Reference Manual', both of which are available directly from ARM.

Table 23. MPU region access permissions

FreeRTOS-MPU definition	Access for Privileged mode tasks	Access for User mode tasks
portMPU_REGION_READ_WRITE	Full Access	Full Access
portMPU_REGION_PRIVILEGED_READ_ONLY	Read Only	No Access
portMPU_REGION_READ_ONLY	Read Only	Read Only
portMPU_REGION_PRIVILEGED_READ_WRITE	Full Access	No Access
portMPU_REGION_EXECUTE_NEVER	Region cannot contain executable code.	

7.3 Defining an MPU Region

Overlapping Regions

A region is an address range to which access permissions can be applied. A maximum of eight regions can be defined at any one time. Regions are numbered from zero to seven.

If multiple regions define overlapping memory ranges, then the access permissions of the highest of the overlapping region numbers will be applied.[5] For example, if region two configures an address range for both read-and-write access at the same time as region three configures the same address range for read-only access, then the memory region will be configured for read-only access.

Predefined Regions and Task Definable Regions

Regions zero to four are used by the kernel to pre-configure a usable run time environment where:

- The Running state task has access to its own stack, but all other RAM is accessible only when the Cortex-M3 microcontroller is running in Privileged mode.
- The area of Flash memory in which the kernel is located and the system peripherals are accessible only when the Cortex-M3 microcontroller is running in Privileged mode.
- The Flash memory, other than that in which the kernel is located, and all non system peripherals (for example, UARTS and analog inputs) can be accessed by both Privileged and User mode tasks.

The kernel reconfigures the MPU during each context switch, so the remaining three regions can be defined differently by each task. The task-defined regions use the highest region numbers, so can be used to override the kernel-defined regions, although there are few circumstances in which that would be desirable.

[5] This applies to any range of memory that appears within more than one region definition—whether the two regions are completely coincident, or only partially overlapping.

Region Start Address and Size Constraints

The MPU hardware imposes two rules that region start address and size definitions must comply with:

1. The region size must be a binary power of two between 32 bytes and 64 gigabytes, inclusive. For example, 32 bytes, 64 bytes, 128 bytes, 256 bytes, and so on are all valid region sizes.

2. The start address must be a multiple of the region size. For example, a region that is configured to be 65536 bytes long must start on an address that is exactly divisible by 65536.

Most cross compilers include language extensions that can be used to force a variable to be placed on a specified address alignment. Listing 80 shows the syntax used for this purpose by the GCC, IAR, and Keil compilers.

```
/* Define and align an array using GCC syntax. */
char cAnArray[ 1024 ] __attribute__((aligned(1024)));

/* Define and align an array using IAR syntax. */
#pragma data_alignment=1024
char cAnArray[ 1024 ];

/* Define and align an array using Keil syntax.  Note this will only work for global
variables.  Keil also has a GCC compatibility mode where __attribute__ can be used.
*/
__align( 1024 ) char cAnArray[ 1024 ];
```

Listing 80. Syntax required by GCC, IAR, and Keil compilers to force a variable onto a particular byte alignment (1024-byte alignment in this example)

```
/* Define two arrays, access to each of which will be controlled by separate MPU
Regions (GCC syntax is shown). */
char cFirstArray[ 1024 ] __attribute__((aligned(1024)));
char cSecondArray[ 256 ] __attribute__((aligned(256)))
```

Listing 81. Defining two arrays that may be placed in adjacent memory

It is necessary to consider also how variables are placed in relation to each other. For example, consider the case shown in Listing 81. cFirstArray starts and ends on a 1024-byte boundary. cSecondArray starts and ends on a 256-byte boundary. As 1024 is divisible by 256, it is likely that the linker will place cSecondArray directly after and adjacent to cFirstArray. If a task has configured one MPU region to provide write access to cFirstArray, and another MPU region to provide write access to cSecond array, then the MPU will not prevent a write off

the end of cFirstArray, as might be the intent. A write outside the boundary of the first MPU region would not result in a memory protection fault but would result, instead, in a valid write into the second MPU. This situation can be avoided by making the size of cFirstArray 1025 bytes and the size of cSecondArray 257 bytes. The alignment requirements then prevent the linker from placing the arrays directly adjacent to each other. The actual alignment of the arrays, and the size of the MPU regions that control access to the arrays, do not change.

7.4 The FreeRTOS-MPU API

All the API functions available in the standard FreeRTOS Cortex-M3 port are also available in FreeRTOS-MPU. This section highlights some minor differences in the way xTaskCreate() is used, and introduces the API extensions that are specific to the MPU enabled kernel.

The xTaskCreateRestricted() API Function

xTaskCreateRestricted() is an extended version of xTaskCreate() that is used to create tasks with restricted execution privileges and restricted memory access rights.

xTaskCreateRestricted() requires all the parameters used by xTaskCreate(), plus four additional parameters that define the three task-specific MPU regions and a stack buffer. Attempting to use this number of parameters in a normal function parameter list would be cumbersome and could, potentially, make heavy use of stack space. Instead, FreeRTOS-MPU defines a structure called xTaskParameters that contains a member for each required parameter. xTaskParameters structures can be declared const and therefore remain in Flash. xTaskCreateRestricted() takes a pointer to an xTaskParameters structure as one of its two parameters. The second parameter is used to pass out a handle to the task being created—exactly as with the xTaskCreate() parameter of the same name. pxCreatedTask can be set to NULL if a handle to the task is not required.

```
portBASE_TYPE xTaskCreateRestricted( xTaskParameters *pxTaskDefinition,
                                     xTaskHandle *pxCreatedTask );
```

Listing 82. The xTaskCreateRestricted() API function prototype

Listing 83 contains the xTaskParameters structure definition, and the definition of the xMemoryRegion structure that xTaskParameters contains. The structure members are described in Table 24 and Table 25. Listing 82 shows how the structures are used.

```
/*
 * Defines a single MPU region.
 */
typedef struct xMEMORY_REGION
{
    void *pvBaseAddress;
    unsigned long ulLengthInBytes;
    unsigned long ulParameters;
} xMemoryRegion;

/*
 * Contains a member for each parameter required to create a restricted task.
 */
typedef struct xTASK_PARAMTERS
{
    pdTASK_CODE pvTaskCode;
    const signed char * const pcName;
    unsigned short usStackDepth;
    void *pvParameters;
    unsigned portBASE_TYPE uxPriority;
    portSTACK_TYPE *puxStackBuffer;
    xMemoryRegion xRegions[ portNUM_CONFIGURABLE_REGIONS ];
} xTaskParameters;
```

Listing 83. Definition of the structures required by the xTaskCreateRestricted() API function

Table 24. xMemoryRegion structure members

Structure Member	Description
pvBaseAddress	The region start address. This must be a multiple of the region size as defined by the ulLengthInBytes value.
ulLengthInBytes	The region size in bytes. This must be a binary power of two having a value between 32 bytes and 4 gigabytes, inclusive.
ulParameters	The access permissions for the region, defined as the bitwise OR of the definitions contained in Table 23.

Table 25. xTaskParameters structure members

Structure Member	Description
pvTaskCode, pcName, usStackDepth, pvParameters	These parameters are the same as their xTaskCreate() equivalents. See Table 2.
uxPriority	In xTaskCreate(), uxPriority is used just to set the priority at which the task is initially created. In xTaskCreateRestricted(), it is also used to set the task to either Privileged mode or User mode. To create a User mode task, set uxPriority to the desired task priority. To create a Privileged mode task, bitwise OR the required task priority with portPRIVILEGE_BIT. For example, to create a User mode task at priority three, set uxPriority to 3. To create a Privileged mode task at priority three, set uxPriority to (3 | portPRIVILEGE_BIT). Source code examples are provided later in this chapter.

Table 25. xTaskParameters structure members

Structure Member	Description
puxStackBuffer	FreeRTOS-MPU uses an MPU region to ensure that the currently executing task can access its own stack, and that writes outside the valid stack space result in a memory protection fault. This means that the task stack start address and size must comply with the MPU region constraints already discussed—the size must be a binary power of two between 32 and 4 gigabytes, and the start address must be a multiple of the size. There are two ways to ensure compliance with the byte alignment requirements: 1. Provide an implementation of pvPortMallocAligned() that will allocate RAM from the heap with the specified byte alignment. The implementation is likely to be complex and potentially wasteful, so nothing further is mentioned in this book about this option. By default, pvPortMallocAligned() is not defined, and the standard pvPortMalloc() is used in its place. If pvPortMallocAligned() is implemented, then puxStackBuffer can be set to NULL. 2. Statically allocate a buffer (array) for use as a stack by the task being created, and use the compiler extensions to ensure that the buffer is correctly aligned. puxStackBuffer should then point to the start of the buffer. This is the method demonstrated later in this chapter.

Table 25. xTaskParameters structure members

Structure Member	Description
xRegions	An array of xMemoryRegion structures that define up to a maximum of three MPU regions (portNUM_CONFIGURABLE_REGIONS equals three). The kernel will automatically configure the MPU to use these regions each time the task being created enters the Running state. The regions can later be redefined using the vTaskAllocateMPURegions() API function. All three region definitions must be present in the xRegions array, even if only one or two are going to be used. To prevent a region definition being used, set all the members of its defining xMemoryRegion structure to zero.

Listing 84 shows an example of an xTaskParameters structure configured to define a User mode task. Changing the uxPriority value from 1 to (1 | portPRIVILEGE_BIT) would cause the structure to define a Privileged mode task, instead.

```
/* A User task is to be created that requires read only access to an array.  First
define the array to comply with the size and alignment rules.  This example uses GCC
syntax. */
char cArray[ 128 ] __attribute__((aligned(128)));

/* Next define the xTaskParameters structure that includes an MPU definition giving
the task the required array access.  Only one of the possible three MPU regions are
being used, but all three have to be defined. */
static const xTaskParameters xCheckTaskParameters =
{
    vDemoTask,  /* pvTaskCode - the function that implements the task. */
    "Demo",     /* pcName             */
    400,        /* usStackDepth - defined in words, not bytes. */
    NULL,       /* pvParameters - not being used in this case. */
    1,          /* uxPriority - User mode priority 1. */
    cTaskStack, /* puxStackBuffer - the array to use as the task stack. */

    /* xRegions - In this case the xRegions array is used to create a single MPU
    region to provide read only access to just one array.  The parameters for
    the two unused regions are just set to 0 to prevent them having any effect. */
    {
        /* Base address     Length      Parameters */
        { cArray,           128,        portMPU_REGION_READ_ONLY },
        { 0,                0,          0 },
        { 0,                0,          0 }
    }
};
```

Listing 84. Using the xTaskParameters structure

Listing 84 shows the simple case where the MPU is being used to control access to a single variable (in this case an array), but the same technique can be used to control access to a set of variables by grouping the variables into a single structure. If this is not practical, then compiler extensions can be used to place the variables manually into a correctly sized and aligned memory area or section defined within the linker script.

Using xTaskCreate() with FreeRTOS-MPU

xTaskCreate() can be used to create both User mode and Privileged mode tasks, but cannot be used to allocate MPU regions to the tasks at the point of their creation. Instead, Privileged mode tasks will have access to the entire memory map, whereas User mode tasks will have access to any Flash and RAM memory that is not configured for Privileged-only access.

As with xTaskCreateRestricted(), set uxPriority to the desired task priority to create a User mode task, or bitwise OR the required task priority with portPRIVILEGE_BIT to create a Privileged mode task. This is demonstrated by Listing 85.

```
int main( void )
{
    /* Create a User mode task using xTaskCreate(). */
    xTaskCreate
       (
            vOldStyleUserModeTask,       /* The function that implements the task. */
            "Task1",                     /* Text name for the task. */
            100,                         /* Stack depth in words. */
            NULL,                        /* Task parameters. */
            3,                           /* Priority and mode (User in this case). */
            NULL                         /* Handle. */
       );

    /* Create a Privileged mode task using xTaskCreate().  Note the use of
    portPRIVILEGE_BIT where the task priority is specified. */
    xTaskCreate
       (
            vOldStylePrivilegedModeTask, /* The function that implements the task. */
            ( signed char * ) "Task2",   /* Text name for the task. */
            100,                         /* Stack depth in words. */
            NULL,                        /* Task parameters. */
            ( 3 | portPRIVILEGE_BIT ),   /* Priority and mode (Privileged in this
                                         case). */
            NULL                         /* Handle. */
       );

    /* Start the scheduler. */
    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 85. Using xTaskCreate() to create both User mode and Privileged mode task with FreeRTOS-MPU

The vTaskAllocateMPURegions() API Function

Up to three MPU region definitions can be assigned to a task as the task is created. The regions can then be redefined using the vTaskAllocateMPURegions() API function.

```
void vTaskAllocateMPURegions( xTaskHandle xTask, const xMemoryRegion * const pxRegions );
```

Listing 86. The vTaskAllocateMPURegions() API function prototype

Table 26. vTaskAllocateMPURegions() parameters

Parameter Name/ Returned Value	Description
xTask	The handle of the task whose MPU region definitions are being modified (the subject task)—see the pxCreatedTask parameter of the xTaskCreate()/xTaskCreateRestricted() API function for information on obtaining handles to tasks. A task can modify the MPU regions assigned to it by passing NULL in place of a valid task handle.
pxRegions	An array of exactly three xMemoryRegion structures. To prevent a region definition from being used, set all members of its defining xMemoryRegion structure to zero. The kernel will automatically configure the MPU to use these definitions each time the task being modified enters the Running state.

```
void vAFunction( xTaskHandle xTask )
{
/* Define an xMemoryRegion array that defines an 8K block from address 0 to
be read only, and a 2K block from address 0x10004000 to be accessible only from
privileged mode. The array defines only two of the possible three MPU regions,
but must contain all three entries.  The members of the unused entry are just set
to zero so it has no effect. */
static const xMemoryRegion xRegions[ 3 ] =
{
    /* Base address Length  Access parameters */
    { 0x00,         8096,   portMPU_REGION_READ_ONLY },
    { 0x10004000,   2048,   portMPU_REGION_PRIVILEGED_READ_WRITE },
    { 0,            0,      0 } /* The third entry is not used so is just set to
                               zero. */
}

    /* Change the MPU regions of the task referenced by xTask to those defined by
    xRegions. */
    vTaskAllocateMPURegions( xTask, xRegions );

    /* Also change the MPU regions used by this task to those defined by xRegions. */
    vTaskAllocateMPURegions( NULL, xRegions );
}
```

Listing 87. Using vTaskAllocateMPURegions() to redefine the MPU regions associated with a task

The portSWITCH_TO_USER_MODE() API Macro

A Privileged mode task can call portSWITCH_TO_USER_MODE() to lower its own privilege to User mode. There is no way for a User mode task to raise its privilege to Privileged mode.

portSWITCH_TO_USER_MODE() does not take any parameters.

7.5 Linker Configuration

FreeRTOS-MPU requires the linker script to define two named sections as described by Table 27, and eight linker variables as described by Table 28.

The syntax used to define the required sections and variable depends on the tool chain being used. Listing 88 and Listing 89 provide an example that uses GNU LD syntax. LD is the linker that is distributed with GCC. The easiest way to generate a suitable linker script is to start with a pre-configured example from a FreeRTOS-MPU demo application.

Table 27. Named linker sections required by FreeRTOS-MPU

Section name	Description
privileged_functions	The section into which the kernel executable image is to be placed. privileged_functions should incorporate the vector table, starting at address zero, with the kernel image starting immediately after the vector table. An MPU region is used to protect access to the privileged_functions section, so its size must be a binary power of two to comply with the MPU region definition rules.
privileged_data	The section into which the kernel data is to be placed. As the section is protected by an MPU region, its start address and size must comply with the MPU region definition rules.

Table 28. Linker variables required by FreeRTOS-MPU

Variable name	Variable value
__FLASH_segment_start__	The start address of the microcontroller Flash memory.
__FLASH_segment_end__	The end address of the microcontroller Flash memory.
__privileged_functions_end__	The end address of the privileged_functions named section.
__SRAM_segment_start__	The start address of the microcontroller SRAM memory.
__SRAM_segment_end__	The end address of the microcontroller SRAM memory.

Table 28. Linker variables required by FreeRTOS-MPU

Variable name	Variable value
__privileged_data_start__	The start address of the privileged_data named section.
__privileged_data_end__	The end address of the privileged_data named section.

```
/* Given the memory map…. */
MEMORY
{
    FLASH (rx) : ORIGIN = 0x0,        LENGTH = 0x80000
    SRAM (rwx) : ORIGIN = 0x10000000, LENGTH = 0x8000
    AHBRAM0    : ORIGIN = 0x2007c000, LENGTH = 0x4000
    AHBRAM1    : ORIGIN = 0x20080000, LENGTH = 0x4000
}

/* ….define the variables required by FreeRTOS-MPU.  First ensure the section sizes
are a binary power of two to comply with the MPU region size rules. */
_Privileged_Functions_Region_Size = 16K;
_Privileged_Data_Region_Size = 256;

/* Then define the variables themselves. */
__FLASH_segment_start__        = ORIGIN( FLASH );
__FLASH_segment_end__          = __FLASH_segment_start__ + LENGTH( FLASH );
__privileged_functions_start__ = ORIGIN( FLASH );
__privileged_functions_end__   = __privileged_functions_start__ +
                                 _Privileged_Functions_Region_Size;
__SRAM_segment_start__         = ORIGIN( SRAM );
__SRAM_segment_end__           = __SRAM_segment_start__ + LENGTH( SRAM );
__privileged_data_start__      = ORIGIN( SRAM );
__privileged_data_end__        = ORIGIN( SRAM ) + _Privileged_Data_Region_Size;
```

Listing 88. Defining the memory map and linker variables using GNU LD syntax

```
/* Defining privileged_functions at the start of the Flash memory, but after the
vector table. */
SECTIONS
{
    /* Privileged section at the start of the flash - vectors must be first
    whatever. */
    privileged_functions :
    {
        KEEP(*(.isr_vector))
        *(privileged_functions)
    } > FLASH

    .text :
    {
        /* Non privileged code kept out of the first 16K of flash. */
         = __privileged_functions_start__ + _Privileged_Functions_Region_Size;

        *(.text*)
        *(.rodata*)

    } > FLASH

       /* Rest of section definitions go here – including the privileged_data
       definition. */
}
```

Listing 89. Defining the privileged_functions named section using GNU LD syntax

7.6 Practical Usage Tips

Accessing Data from a User Mode Task

A User mode task cannot access RAM that is outside its own stack space, unless the address falls within the range of one of the task's MPU region definitions. If, for example, a User mode task needs the value of a globally declared queue handle, then, to be accessible, the value must first be copied into a variable that is on the task stack. There are several ways to achieve this, including:

- Initially, create the task in Privileged mode, and then copy the global variable value into a stack variable, before switching the task into the required User mode. This method is demonstrated in Listing 90.

- Pass the value of the global variable into the task using the task parameter. This method is demonstrated in Listing 91.

```
/* The handle to a queue is stored in a global (or file scope) variable. */
xQueueHandle xGlobalQueue;

void vATask( void *pvParameters )
{
xQueueHandle xStackQueue;

    /* This task was created in Privileged mode so can access the global variable.
    Copy the value of the global variable into a stack variable while the task is
    still in Privileged mode. */
    xStackQueue = xGlobalQueue;

    /* Now set the task into User mode.  From this point on the task can no longer
    access the value of the global variable, but can access its local stack copy. */
    portSWITCH_TO_USER_MODE();

    for( ;; )
    {
        /* The main task functionality is performed in User mode. Data can be sent
        to or from the queue using xStackQueue as the handle. */
    }
}
```

Listing 90. Copying data into a stack variable before setting the task into User mode

```
/* The handle to a queue is stored in a global (or file scope) variable. */
xQueueHandle xGlobalQueue;

void vATask( void *pvParameters )
{
xQueueHandle xStackQueue;

    /* This task was created in User mode so cannot access the global variable.  It
    can access variables stored on its own stack and the task parameter.  The value
    of xGlobalQueue is passed into this task using the task parameter and then copied
    into the local stack variable, casting to the appropriate type. */
    xStackQueue = ( xQueueHandle ) pvParameters;

    for( ;; )
    {
        /* The main task functionality is done here.  Data can be sent to or from the
        queue using xStackQueue as the handle. */
    }
}
```

Listing 91. Copying the value of a global variable into a stack variable using the task parameter

Intertask Communication from User Mode

Code executing in User mode cannot access RAM outside its own stack and the MPU regions that are configured for it. This does not prevent User mode tasks from using queues or semaphores to communicate with other tasks or interrupts.

The RAM used by queues and semaphores is owned and controlled by the kernel and can be accessed only when the processor is executing in Privileged mode. Calling an API function such as xQueueSend() causes the processor to switch temporarily into Privileged mode, from where the data being queued can be copied from the User mode task into the kernel controlled queue storage area. Similarly, calling an API function such as xQueueReceive() causes the processor to switch temporarily into Privileged mode, from where the data being received can be copied from the kernel controlled queue storage area into the User mode task.

FreeRTOS-MPU Demo Projects

FreeRTOS-MPU is included in the main FreeRTOS download. Some heavily commented FreeRTOS-MPU demo applications are located in sub-directories with names that start ‘Cortex-MPU’ within the FreeRTOS\Demo directory.

Chapter 8

The FreeRTOS Download

8.1 Chapter Introduction and Scope

FreeRTOS is distributed as a single .zip file archive containing all the official FreeRTOS ports and a large number of pre-configured demo applications. The large number of files can seem overwhelming, but only a subset will actually be required.

Scope

This chapter aims to help users orientate themselves with the FreeRTOS files and directories by:

- Providing a top level view of the FreeRTOS directory structure.
- Describing which files are actually required by Cortex-M3 microcontroller projects.
- Introducing the demo applications.
- Providing information on how a new project can be created.

The description here relates only to the main FreeRTOS .zip file distribution. The examples that come with this book use a slightly different organization.

8.2 Files and Directories

The official FreeRTOS distribution contains:

- The core FreeRTOS source code. This is the code that is common to all ports.
- A port layer for each supported microcontroller and compiler combination.
- A project file or makefile to build a demo application for each supported microcontroller and compiler combination.
- A set of common demo tasks. These are simple tasks that are used by most of the demo applications.

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 only 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 43. 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 are 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. This arrangement is shown in Figure 44.

An optional fourth source file called croutine.c implements the FreeRTOS co-routine functionality. It need only be included in the build if co-routines are actually going to be used. Co-routines are intended for use on very small microcontrollers, so it is unlikely that they will be used in a Cortex-M3 microcontroller project.

```
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 44. The three core files that implement the FreeRTOS kernel

Removing Unused Source Files

The 'portable layer' is the code that tailors the FreeRTOS kernel to a particular compiler and microcontroller combination. The portable layer source files for the Cortex-M3 are located in the FreeRTOS\Source\portable\[compiler]\ARM_CM3 directories, where [compiler] must be substituted with RVDS, IAR, or GCC to locate the port files for the RVDS/Keil, IAR, and GCC compilers, respectively.

When using the Cortex-M3 port:

- All the sub-directories under FreeRTOS\Source\portable can be deleted, except FreeRTOS\Source\portable\[compiler] and FreeRTOS\Source\portable\MemMang.
- All the sub-directories under FreeRTOS\Source\portable\[compiler] can be deleted, except FreeRTOS\Source\portable\[compiler]\ARM_CM3.

The FreeRTOS\Source directories that must remain are shown in Figure 45.

```
FreeRTOS
    |
    +-Source                Contains the implementation of the real time kernel.
        |
        +-portable          Contains all the port specific files.
            |
            +-MemMang       Contains the example memory management implementations.
            +-[compiler]    [Compiler] can be RVDS, IAR or GCC.
                |
                +-ARM_CM3   Contains the Cortex-M3 port files specific to
                            [compiler].
```

Figure 45. The source directories required to build a Cortex-M3 microcontroller demo application

8.3 Demo Applications

Each official FreeRTOS port comes with a demo application that should build with no errors or warnings being generated[6]. The demo application has several purposes:

- To provide an example of a working and pre-configured project with the correct files included and the correct compiler options set.
- To allow 'out of the box' experimentation with minimal setup or prior knowledge.
- As a demonstration of how the FreeRTOS API can be used.
- As a base from which real applications can be created.

Each demo project is located in a unique sub-directory under the Demo directory. The sub-directory name indicates the port to which the demo project relates. Several demo applications are provided for various Cortex-M3 based microcontroller families—each contained in a unique sub-directory that starts 'CORTEX_….'.

Every demo application also has its own documentation page on the FreeRTOS.org website. The documentation page includes information on:

- How to locate the project file or makefile for the demo within the FreeRTOS directory structure.
- Which hardware the project is configured to use.
- How to set up the hardware for running the demo.
- How to build the demo.
- How the demo is expected to behave.

All the demo projects create a subset of the common demo tasks, the implementations of which are contained in the FreeRTOS\Demo\Common\Minimal directory. The common demo tasks exist purely to demonstrate how the FreeRTOS API can be used—they do not implement any particular useful functionality.

[6] This is the ideal scenario, and is normally the case, but is dependent on the version of the compiler used to build the demo. Upgraded compilers can sometimes generate warnings where their predecessors did not.

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.

Removing Unused Demo Files

When using a provided demo application:

- All the sub-directories under FreeRTOS\Demo can be deleted, except the directory containing the demo being used and FreeRTOS\Demo\Common.
- FreeRTOS\Demo\Common contains many more files than are referenced from any one demo application, so this directory can be trimmed down, if desired. Inspect the demo application makefile or project file to identify files that can be deleted. In general, FreeRTOS\Demo\Common\Minimal should not be deleted.

The FreeRTOS\Demo directories that must remain are shown in Figure 46.

```
FreeRTOS
    |
    +-Demo            Contains all the demo application source and projects.
       |
       +-[Demo]       Contains the makefile or project for the demo being built.
       |
       +-Common       Contains files common to all demo applications.
```

Figure 46. The demo directories required to build a demo application

8.4 Creating a FreeRTOS Project

Adapting One of the Supplied Demo Projects

Every official FreeRTOS port comes with at least one pre-configured demo application that should build with no errors or warnings. It is recommended that new projects are created by adapting one of these existing projects; this will allow the project to have the correct files included and the correct compiler options set.

To start a new application from an existing demo project:

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

2. Remove the source files that define the demo tasks. Any file that is located within the Demo\Common directory tree can be removed.

3. Delete all the functions within main.c, except prvSetupHardware().

4. Ensure that the following constants are all set to 0 within FreeRTOSConfig.h. This will prevent the linker from looking for any hook functions. Hook functions can be added later, if required.

    - configUSE_IDLE_HOOK
    - configUSE_TICK_HOOK
    - configUSE_MALLOC_FAILED_HOOK
    - configCHECK_FOR_STACK_OVERFLOW

5. Create a new main() function from the template shown in Listing 92.

6. Check that the project still builds.

Following these steps will create a project that includes the correct 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 92. The template for a new main() function

Creating a New Project from Scratch

As already mentioned, it is recommended that new projects are created from an existing demo project. If this is not desirable, then a new project can be created using the following procedure:

1. Create a new empty project file or makefile using your chosen tool chain.
2. Add the files detailed in Table 29 to the newly created project or makefile.
3. Copy an existing FreeRTOSConfig.h file into the project directory.
4. Add the following directories to the path the project will search to locate header files:
    - FreeRTOS\Source\include
    - FreeRTOS\Source\portable\[compiler]\ARM_CM3 (where [compiler] is RVDS, IAR, or GCC)
5. Copy the compiler settings from the relevant demo project or makefile.

Table 29. 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]\ARM_CM3
port_asm.s	FreeRTOS\Source\portable\[compiler]\ARM_CM3. An assembly file is not required when using GCC.
heap_n.c	FreeRTOS\Source\portable\MemMang, where n is either 1, 2 or 3

Header Files

A source file that uses the FreeRTOS API must include ‘FreeRTOS.h’, followed by the header file that contains the prototype for the API function being used—either ‘task.h’, ‘queue.h’, or ‘semphr.h’.

8.5 Data Types and Coding Style Guide

Data Types

Each port of FreeRTOS has a unique portmacro.h header file that contains (amongst other things) definitions for two special data types, portTickType and portBASE_TYPE. These data types are described in Table 30.

Table 30. Special data types used by FreeRTOS

Macro or typedef used	Actual type
portTickType	This is used to store the tick count value and to specify block times. portTickType can be either an unsigned 16-bit type or an unsigned 32-bit type, depending on the setting of configUSE_16_BIT_TICKS within FreeRTOSConfig.h. Using a 16-bit type can greatly improve efficiency on 8-bit and 16-bit architectures, but severely limits the maximum block period that can be specified. There is no reason to use a 16-bit type on a 32-bit architecture, so configUSE_16_BIT_TICKS should be set to 0.
portBASE_TYPE	This is always defined as the most efficient data type for the architecture. Typically, this is a 32-bit type on a 32-bit architecture, a 16-bit type on a 16-bit architecture, and an 8-bit type on an 8-bit architecture. portBASE_TYPE is generally used for return types that can take only a very limited range of values, and for Booleans. Cortex-M3 ports define portBASE_TYPE as type ‘long’.

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 char with either ‘signed’ or ‘unsigned’.

Plain int types are never used—only long and short.

Variable Names

Variables are prefixed 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 with ‘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

One tab is always set to equal four spaces.

Macro Names

Most macros are written in upper case and prefixed with lower case letters that indicate where the macro is defined. Table 31 provides a list of prefixes.

Table 31. 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 32 are used throughout the FreeRTOS source code.

Table 32. 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 many different compilers, all of which differ in 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 1: 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 33.
- You wish to receive direct technical support.
- You wish to have assistance with your development.
- You require guarantees and indemnification.

Table 33. Comparing the open source license with the commercial license

	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 website 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 be used only 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 that the exception text is subject to change. Consult the FreeRTOS.org website for the most recent version.

Clause 1

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 (including any modifications you may have made) should the recipient request it.*

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

3. *The independent modules add significant and primary functionality to FreeRTOS and do not merely extend the existing functionality already present in FreeRTOS.*

An independent module is a module which is not derived from or based on FreeRTOS.

Clause 2

FreeRTOS 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 Real Time Engineers ltd. (this is the norm within the industry and is intended to ensure information accuracy).

INDEX

P

Q

R

S

T

U

V

X

Z