FreeRTOS Kernel

Developer Guide

Table of Contents

About the FreeRTOS Kernel

The FreeRTOS kernel is an open source software maintained by Amazon.

The FreeRTOS kernel is ideally suited to deeply embedded real-time applications that use microcontrollers or small microprocessors. This type of application normally includes 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 does not render the system useless. For example, responding to keystrokes too slowly might 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 does result in absolute failure of the system. For example, a driver's airbag has the potential to do more harm than good if it responded to crash sensor inputs too slowly.

The FreeRTOS kernel is a real-time kernel (or real-time scheduler) on top of which embedded applications can be built to meet hard real-time requirements. It allows applications to be organized as a collection of independent threads of execution. On a processor that has only one core, 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.

Don't be concerned if you don't fully understand the concepts in the previous paragraph yet. The guide covers them in detail and provides many examples to help you understand how to use a real-time kernel and the FreeRTOS kernel in particular.

Value Proposition

The unprecedented global success of the FreeRTOS kernel comes from its compelling value proposition. The FreeRTOS kernel is professionally developed, strictly quality-controlled, robust, supported, does not contain any intellectual property ownership ambiguity, and is truly free to use in commercial applications without any requirement to expose your proprietary source code. You can take a product to market using the FreeRTOS kernel without paying any fees, and thousands of people do just that. If, at any time, you would like to receive additional backup, or if your legal team require additional written guarantees or indemnification, then there is a simple low cost commercial upgrade path. Peace of mind comes with the knowledge that you can opt to take the commercial route at any time you choose.

A Note About Terminology

In the FreeRTOS kernel, each thread of execution is called a *task*. Although there is no consensus on terminology in the embedded community, *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, using a kernel is preferable, but where the crossover point occurs is always subjective.

Task prioritization can help ensure an application meets its processing deadlines, but a kernel can bring other less obvious benefits, too:

- 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 result in code that can be reused 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. However, applications that don't make use of an RTOS normally include some form of tick interrupt anyway.

- Idle time

The Idle task is created automatically when the scheduler 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.

- Power management

The efficiency gains by using an RTOS allow the processor to spend more time in a low power mode.

Power consumption can be decreased significantly by placing the processor into a low power state each time the Idle task runs. The FreeRTOS kernel also has a tickless mode that allows the processor to enter and remain in the low power mode for longer.

- Flexible interrupt handling

Interrupt handlers can be kept very short by deferring processing to either a task created by the application writer or the FreeRTOS daemon task.

- 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.

FreeRTOS Kernel Features

The FreeRTOS kernel has the following standard features:

- Preemptive or cooperative operation
- Very flexible task priority assignment
- Flexible, fast, and lightweight task notification mechanism
- Queues
- Binary semaphores
- Counting semaphores
- Mutexes
- Recursive mutexes
- Software timers
- Event groups
- Tick hook functions
- Idle hook functions
- Stack overflow checking
- Trace recording
- Task runtime statistics gathering
- Optional commercial licensing and support
- Full interrupt nesting model (for some architectures)
- A tickless capability for extreme low power applications
- Software-managed interrupt stack when appropriate (this can help save RAM)

Licensing

The FreeRTOS kernel is available to users under the terms of the MIT license.

Included Source Files and Projects

Source code, preconfigured project files, and full build instructions for all the examples presented are provided 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. The zip file may not include the latest version of the FreeRTOS kernel.

The screen shots included in this book were taken while the examples were executing in a Microsoft Windows environment, using the FreeRTOS Windows port. The project that uses the FreeRTOS Windows port is preconfigured to build using the free Express edition of Visual Studio, which can be downloaded from https://www.visualstudio.com/vs/community/. Although the FreeRTOS Windows port provides a convenient evaluation, test and development platform, it does not provide true real-time behavior.

FreeRTOS Kernel Distribution

The FreeRTOS kernel is distributed as a single zip file archive that contains all the official FreeRTOS kernel ports and a large number of preconfigured demo applications.

Understanding the FreeRTOS Kernel Distribution

The FreeRTOS kernel can be built with approximately 20 different compilers, and can run on more than 30 different processor architectures. Each supported combination of compiler and processor is considered to be a separate FreeRTOS port.

Building the FreeRTOS Kernel

You can think of FreeRTOS as a library that provides multitasking capabilities to what would otherwise be a bare metal application.

FreeRTOS is supplied as a set of C source files. Some of the source files are common to all ports, while others are specific to a port. Build the source files as part of your project to make the FreeRTOS API available to your application. To make this easy for you, each official FreeRTOS port is provided with a demo application. The demo application is preconfigured to build the correct source files and include the correct header files.

Demo applications should build out of the box. However, a change made in the build tools since the demo was released can cause issues. For more information, see Demo Applications later in this topic.

FreeRTOSConfig.h

FreeRTOS is configured by a header file called FreeRTOSConfig.h.

FreeRTOSConfig.h is used to tailor FreeRTOS for use in a specific application. For example, FreeRTOSConfig.h contains constants such as configUSEPREEMPTION, the setting of which defines whether the cooperative or preemptive scheduling algorithm will be used. FreeRTOSConfig.h contains application-specific definitions, so it should be located in a directory that is part of the application being built, not in a directory that contains the FreeRTOS source code.

A demo application is provided for every FreeRTOS port, and every demo application contains a FreeRTOSConfig.h file. Therefore, you never need to create a FreeRTOSConfig.h file from scratch. Instead, we recommend that you start with and then adapt the FreeRTOSConfig.h used by the demo application provided for the FreeRTOS port in use.

The Official FreeRTOS Kernel Distribution

FreeRTOS is distributed in a single zip file. The zip file contains source code for all the FreeRTOS ports and project files for all the FreeRTOS demo applications. It also contains a selection of FreeRTOS+ ecosystem components and a selection of FreeRTOS+ ecosystem demo applications.

Don't worry about the number of files in the FreeRTOS distribution. Only a small number of files are required in any one application.

The Top Directories in the FreeRTOS Distribution

The first- and second-level directories of the FreeRTOS distribution are shown and described here.

FreeRTOS

| |

| ├─**Source** directory containing the FreeRTOS source files

| |

| └─**Demo** directory containing preconfigured and port-specific FreeRTOS demo projects

|

FreeRTOS-Plus

|

├─**Source** directory containing source code for some FreeRTOS + ecosystem components

|

└─**Demo** directory containing demo projects for FreeRTOS+ ecosystem components

The zip file contains only one copy of the FreeRTOS source files, all the FreeRTOS demo projects, and all the FreeRTOS+ demo projects. You should find the FreeRTOS source files in the FreeRTOS/Source directory. The files might not build if the directory structure is changed.

FreeRTOS Source Files Common to All Ports

The core FreeRTOS source code is contained in just two C files that are common to all the FreeRTOS ports. These are called tasks.c, and list.c. They are located directly in the FreeRTOS/Source directory. The following source files are located in the same directory:

- queue.c

  queue.c provides both queue and semaphore services. queue.c is nearly always required.
- timers.c

  timers.c provides software timer functionality. You need to include it in the build only if software timers are going to be used.
- eventgroups.c

  eventgroups.c provides event group functionality. You need to include it in the build only if event groups are going to be used.
- croutine.c

  croutine.c implements the FreeRTOS co-routine functionality. You need to include it in the build only if co-routines are going to be used. Co-routines were intended for use on very small microcontrollers.

They are rarely used now and are therefore not maintained to the same level as other FreeRTOS features. Co-routines are not covered in this guide.

FreeRTOS

|

└Source

|

├tasks.c FreeRTOS source file - always required

├list.c FreeRTOS source file - always required

├queue.c FreeRTOS source file - nearly always required

├timers.c FreeRTOS source file - optional

├eventgroups.c FreeRTOS source file - optional

└croutine.c FreeRTOS source file - optional

It is recognized that the file names might result in name space clashes because many projects will already include files that have the same names. However, changing the names of the files now would be problematic because doing so would break compatibility with the many thousands of projects that use FreeRTOS, as well as automation tools and IDE plug-ins.

FreeRTOS Source Files Specific to a Port

Source files specific to a FreeRTOS port are contained within the FreeRTOS/Source/portable directory. The portable directory is arranged as a hierarchy, first by compiler, then by processor architecture.

If you are running FreeRTOS on a processor with architecture '*architecture*' using compiler '*compiler*' then, in addition to the core FreeRTOS source files, you must also build the files located in FreeRTOS/Source/portable/[*compiler*]/[*architecture*] directory.

As will be described in Chapter 2, Heap Memory Management, FreeRTOS also considers heap memory allocation to be part of the portable layer. Projects that use a FreeRTOS version older than V9.0.0 must include a heap memory manager. From FreeRTOS V9.0.0 a heap memory manager is only required if configSUPPORTDYNAMICALLOCATION is set to 1 in FreeRTOSConfig.h, or if configSUPPORTDYNAMICALLOCATION is left undefined.

FreeRTOS provides five example heap allocation schemes. The five schemes are named heap1 to heap5, and are implemented by the source files heap1.c to heap5.c respectively. The example heap allocation schemes are contained in the FreeRTOS/Source/portable/MemMang directory. If you have configured FreeRTOS to use dynamic memory allocation then it is necessary to build *one* of these five source files in your project, unless your application provides an alternative implementation.

The following figure shows the port-specific source files within the FreeRTOS directory tree.

FreeRTOS

|

└Source

│

└portable Directory containing all port-specific source files

│

├MemMang Directory containing the 5 alternative heap allocation source files

│

├[compiler 1] Directory containing port files specific to compiler 1

│ │

│ ├[architecture 1] Contains files for the compiler 1 architecture 1 port

│ ├[architecture 2] Contains files for the compiler 1 architecture 2 port

│ └[architecture 3] Contains files for the compiler 1 architecture 3 port

│

└[compiler 2] Directory containing port files specific to compiler 2

│

├[architecture 1] Contains files for the compiler 2 architecture 1 port

├[architecture 2] Contains files for the compiler 2 architecture 2 port

└[etc.]

Include Paths

FreeRTOS requires three directories to be included in the compiler's include path:

1. The path to the core FreeRTOS header files, which is always FreeRTOS/Source/include.
2. The path to the source files that are specific to the FreeRTOS port in use. As described above, this is FreeRTOS/Source/portable/[*compiler*]/[*architecture*].
3. A path to the FreeRTOSConfig.h header file.

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', 'semphr.h', 'timers.h', or 'eventgroups.h'.

Demo Applications

Each FreeRTOS port comes with at least one demo application that should build with no errors or warnings being generated, although some demos are older than others, and sometimes a change in the build tools made since the demo was released can cause an issue.

Note for Linux users: FreeRTOS is developed and tested on a Windows host. Occasionally this results in build errors when demo projects are built on a Linux host. Build errors are almost always related to the

case of letters used when referencing file names, or the direction of slash characters used in file paths. Please use the FreeRTOS contact form (http://www.FreeRTOS.org/contact) to alert us to any such errors.

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 FreeRTOS/Demo directory. The name of the sub-directory indicates the port to which the demo project relates.

Every demo application is also described by a web page on the FreeRTOS.org web site. The web page includes information on:

- How to locate the project file 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.

More recent demo projects can also build a beginners 'blinky' project. Blinky projects are very basic. Typically they will create just two tasks and one queue.

Every demo project includes a file called main.c. This contains the main() function, from where all the demo application tasks are created. See the comments within the individual main.c files for information specific to that demo.

The following fire shows the FreeRTOS/Demo directory hierarchy.

FreeRTOS

|

└─Demo Directory containing all the demo projects

|

├─[Demo x] Contains the project file that builds demo 'x'

|

├─[Demo y] Contains the project file that builds demo 'y'

|

├─[Demo z] Contains the project file that builds demo 'z'

|

└**Common Contains files that are built by all the demo applications**

Creating a FreeRTOS Project

Every FreeRTOS port comes with at least one preconfigured 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, the correct interrupt handlers installed, 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 can be removed from the project.
3. Delete all the function calls within main(), except prvSetupHardware() and vTaskStartScheduler(), as shown in Listing 1.
4. Check 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;

}
```

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. Using your chosen tool chain, create a new project that does not yet include any FreeRTOS source files.
2. Ensure the new project can be built, downloaded to your target hardware, and executed.
3. Only when you are sure you already have a working project, add the FreeRTOS source files detailed in Table 1 to the project.

4. Copy the FreeRTOSConfig.h header file used by the demo project provided for the port in use into the project directory.
5. Add the following directories to the path the project will search to locate header files:

- FreeRTOS/Source/include
- **FreeRTOS/Source/portable/[*compiler*]/[*architecture*] (where**

  [*compiler*] and [*architecture*] are correct for your chosen port)
- The directory containing the FreeRTOSConfig.h header file

1. Copy the compiler settings from the relevant demo project.
2. Install any FreeRTOS interrupt handlers that might be necessary. Use the web page that describes the port in use, and the demo project provided for the port in use, as a reference.

The following table lists the FreeRTOS source files to include in the project.

File	Location
tasks.c	FreeRTOS/Source
queue.c	FreeRTOS/Source
list.c	FreeRTOS/Source
timers.c	FreeRTOS/Source
eventgroups.c	FreeRTOS/Source
All C and assembler files	FreeRTOS/Source/portable/[compiler]/[architecture]
heapn.c	FreeRTOS/Source/portable/MemMang, where n is either 1, 2, 3, 4 or 5. This file became optional from FreeRTOS V9.0.0.

Projects that use a FreeRTOS version older than V9.0.0 must build one of the heapn.c files. From FreeRTOS V9.0.0 a heapn.c file is only required if configSUPPORTDYNAMICALLOCATION is set to 1 in FreeRTOSConfig.h or if configSUPPORTDYNAMICALLOCATION is left undefined. Refer to Chapter 2, Heap Memory Management, for more information.

Data Types and Coding Style Guide

Each port of FreeRTOS has a unique portmacro.h header file that contains (amongst other things) definitions for two port-specific data types: TickTypet and BaseTypet.

The following table lists data types used by FreeRTOS.

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', unless the char is used to hold an ASCII character, or a pointer to char is used to point to a string.

Plain int types are never used.

Variable Names

Variables are prefixed with their type: 'c' for char, 's' for int16t (short), 'l' int32t (long), and 'x' for BaseTypet and any other non-standard types (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'. For example, a variable of type uint8t 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 BaseType_t and is defined within queue.c.
- pvTimerGetTimerID() returns a pointer to void and is defined within timers.c.

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 uppercase and prefixed with lowercase letters that indicate where the macro is defined.

The following table lists of the macro prefixes.

Prefix	**Location of macro definition**
port (for example, portMAXDELAY)	portable.h or portmacro.h
task (for example, taskENTERCRITICAL())	task.h
pd (for example, pdTRUE)	projdefs.h
config (for example, configUSEPREEMPTION)	FreeRTOSConfig.h
err (for example, errQUEUEFULL)	projdefs.h

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 following table lists the macros used in the FreeRTOS source code.

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.

Heap Memory Management

Starting with V9.0.0, FreeRTOS applications can be completely statically allocated, which means there is no need to include a heap memory manager.

This section covers:

- When FreeRTOS allocates RAM.
- The five example memory allocation schemes included with FreeRTOS.
- Use cases for each memory allocation scheme.

Prerequisites

You need a good understanding of C programming to use FreeRTOS. Specifically, you should be familiar with:

- How a C project is built, including the compiling and linking phases.
- The concepts of a stack and heap.
- The standard C library malloc() and free() functions.

Dynamic Memory Allocation and Its Relevance to FreeRTOS

This guide introduces kernel objects such as tasks, queues, semaphores, and event groups. To make FreeRTOS as easy to use as possible, these kernel objects are not statically allocated at compile time, but dynamically allocated at runtime. FreeRTOS allocates RAM each time a kernel object is created, and frees RAM each time a kernel object is deleted. This policy reduces design and planning effort, simplifies the API, and minimizes the RAM footprint.

Dynamic memory allocation is a C programming concept. It is not a concept that is specific to either FreeRTOS or multitasking. It is relevant to FreeRTOS because kernel objects are allocated dynamically, and the dynamic memory allocation schemes provided by general purpose compilers are not always suitable for real-time applications.

Memory can be allocated using the standard C library malloc() and free() functions, but they might 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 fragmentation. The heap is considered to be fragmented if the free RAM in the heap is broken into small blocks that are separated from each other. If the heap is fragmented, then an attempt to allocate a block will fail if no single free block in the heap is large enough to contain the block, even if the total size of all the separate free blocks in the heap is many times greater than the size of the block that cannot be allocated.
- They can complicate the linker configuration.
- They can be the source of errors that are difficult to debug if the heap space is allowed to grow into memory used by other variables.

Options for Dynamic Memory Allocation

Earlier versions of FreeRTOS used a memory pools allocation scheme, whereby pools of different size memory blocks were pre-allocated at compile time, and then returned by the memory allocation functions. Although this is a scheme commonly used in real-time systems, it generated a lot of support requests. The scheme was dropped because it could not use RAM efficiently enough to make it viable for really small embedded systems.

FreeRTOS now treats memory allocation as part of the portable layer (as opposed to part of the core code base). This is in recognition of the varying dynamic memory allocation and timing requirements of embedded systems. A single dynamic-memory allocation algorithm is appropriate only for a subset of applications. Also, removing dynamic memory allocation from the core code base enables application writers to provide their own specific implementations, when appropriate.

When FreeRTOS requires RAM, it calls pvPortMalloc() instead of malloc(). When RAM is being freed, the kernel calls vPortFree() instead of free(). pvPortMalloc() has the same prototype as the standard C library malloc() function. vPortFree() has the same prototype as the standard C library free() function.

pvPortMalloc() and vPortFree() are public functions, so they can also be called from application code.

FreeRTOS comes with five example implementations of both pvPortMalloc() and vPortFree(), all of which are documented here. FreeRTOS applications can use one of these example implementations or provide their own.

The five examples are defined in the heap_1.c, heap_2.c, heap_3.c, heap_4.c and heap_5.c source files located in the FreeRTOS/Source/portable/MemMang directory.

Example Memory Allocation Schemes

Because FreeRTOS applications can be completely statically allocated, you do not need to include a heap memory manager.

Heap_1

It is common for small dedicated embedded systems to create tasks and other kernel objects only before the scheduler has been started. Memory is dynamically allocated by the kernel before the application starts to perform any real-time functionality, and memory remains allocated for the lifetime of the application. This means the chosen allocation scheme does not have to consider any of the more complex memory allocation issues, such as determinism and fragmentation. It can consider instead attributes such as code size and simplicity.

Heap_1.c implements a very basic version of pvPortMalloc(). It does not implement vPortFree(). Applications that never delete a task or other kernel object can use heap_1.

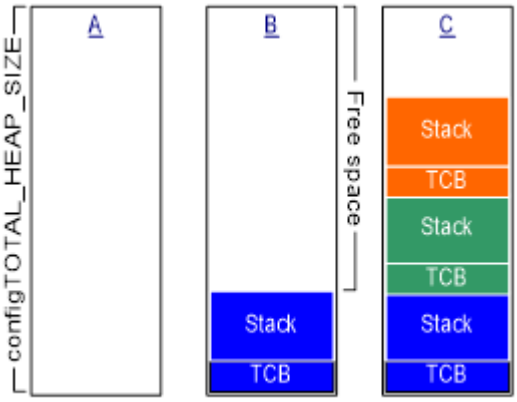
Some commercially critical and safety-critical systems that would otherwise prohibit the use of dynamic memory allocation might also be able to use heap_1. Critical systems often prohibit dynamic memory allocation because of the uncertainties associated with non-determinism, memory fragmentation, and failed allocations, but heap_1 is always deterministic and cannot fragment memory.

As calls to pvPortMalloc() are made, the heap_1 allocation scheme subdivides a simple array into smaller blocks. The array is called the FreeRTOS heap.

The total size of the array (in bytes) is set by the definition configTOTAL_HEAP_SIZE in FreeRTOSConfig.h. Defining a large array in this way can make the application appear to consume a lot of RAM even before any memory has been allocated from the array.

Each created task requires a task control block (TCB) and a stack to be allocated from the heap.

The following figure shows how heap_1 subdivides the simple array as tasks are created. RAM is allocated from the heap_1 array each time a task is created.



- **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.

Heap_2

Heap_2 is included in the FreeRTOS distribution for backward compatibility. It is not recommended for new designs. Consider using heap_4 instead because it provides more functionality.

Heap_2.c also works by subdividing an array that is dimensioned by configTOTAL_HEAP_SIZE. It uses a best fit algorithm to allocate memory. Unlike heap_1, it does allow memory to be freed. Again, the array is statically declared, so the application appears to consume a lot of RAM, even before any memory from the array has been assigned.

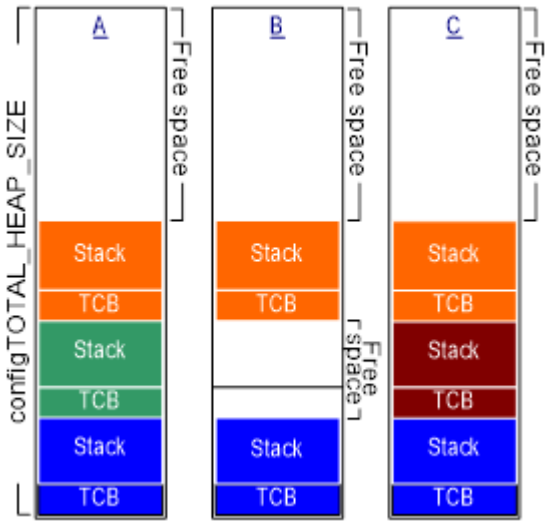
The best fit algorithm ensures that pvPortMalloc() uses the free block of memory that is closest in size to the number of bytes requested. For example, consider the scenario where:

- 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, before returning a pointer to the 20-byte block. (This is an oversimplification, because heap_2 stores information on the block sizes in the heap area, so the sum of the two split blocks will actually be less than 25.) The new 5-byte block remains available to future calls to pvPortMalloc().

Unlike heap_4, heap_2 does not combine adjacent free blocks into a single larger block. For this reason, it is more susceptible to fragmentation. However, fragmentation is not an issue if the blocks being allocated and subsequently freed are always the same size. Heap_2 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.

The following figure shows RAM being allocated and freed from the heap_2 array as tasks are created and deleted.



The figure shows how the best fit algorithm works when a task is created, deleted, and then created again.

- **A shows the array after three tasks have been created. A large free**

  block remains at the top of the array.
- **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.
- **C shows the array after another task has been created. Creating**

  the task resulted in two calls to pvPortMalloc(): one to allocate a new TCB and one to allocate the task stack. Tasks are created using the xTaskCreate() API function, which is described in Creating Tasks (p. 27). The calls to pvPortMalloc() occur internally within xTaskCreate().

  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 is not deterministic, but it is faster than most standard library implementations of malloc() and free().

Heap_3

Heap_3.c uses the standard library malloc() and free() functions, so the size of the heap is defined by the linker configuration. The configTOTAL_HEAP_SIZE setting has no effect.

Heap_3 makes malloc() and free() thread-safe by temporarily suspending the FreeRTOS scheduler. For information about thread safety and scheduler suspension, see the Resource Management (p. 155) section.

Heap_4

Like heap_1 and heap_2, heap_4 subdivides an array into smaller blocks. The array is statically declared and dimensioned by configTOTAL_HEAP_SIZE, so the application appears to consume a lot of RAM even before any memory has been allocated from the array.

Heap_4 uses a first fit algorithm to allocate memory. Unlike heap_2, it combines (coalescences) adjacent free blocks of memory into a single larger block. This minimizes the risk of memory fragmentation.

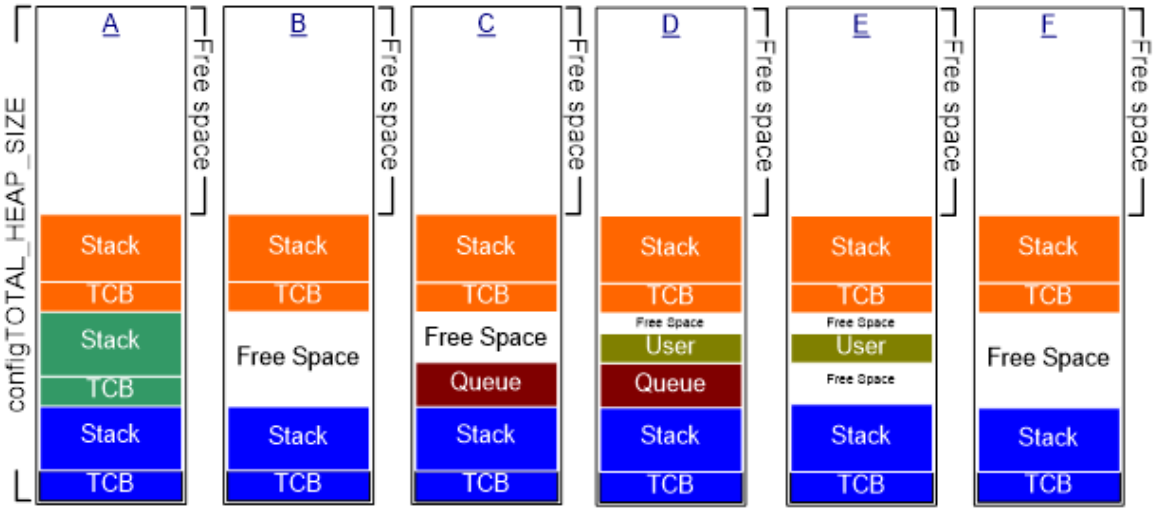
The first fit algorithm ensures pvPortMalloc() uses the first free block of memory that is large enough to hold the number of bytes requested. For example, consider the scenario where:

- The heap contains three blocks of free memory. They appear in this order in the array 5 bytes, 200 bytes, and 100 bytes.
- pvPortMalloc() is called to request 20 bytes of RAM.

The first free block of RAM into which the requested number of bytes will fit is the 200-byte block, so pvPortMalloc() splits the 200-byte block into one block of 20 bytes and one block of 180 bytes, before returning a pointer to the 20-byte block. (This is an oversimplification because heap_4 stores information on the block sizes within the heap area, so the sum of the two split blocks will be less than 200 bytes.) The new 180-byte block remains available to future calls to pvPortMalloc().

Heap_4 combines (coalescences) adjacent free blocks into a single larger block, minimizing the risk of fragmentation. Heap_4 is suitable for applications that repeatedly allocate and free different-sized blocks of RAM.

The following figure shows RAM being allocated and freed from the heap_4 array. It demonstrates how the heap_4 first fit algorithm with memory coalescence works, as memory is allocated and freed.



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

B shows the array after one of the tasks has been deleted. The large free block at the top of the array remains. There is also a free block where the TCB and stack of the task that has been deleted were previously allocated. Memory freed when the TCB was deleted and memory freed when the stack was deleted does not remain as two separate free blocks. Instead, they are combined to create a larger, single free block.

C shows the array after a FreeRTOS queue has been created. Queues are created using the xQueueCreate() API function, which is described in Using a Queue (p. 67). xQueueCreate() calls pvPortMalloc() to allocate the RAM used by the queue. Because heap_4 uses a first fit algorithm,

pvPortMalloc() allocates RAM from the first free RAM block that is large enough to hold the queue. In the figure, this was the RAM freed when the task was deleted. The queue does not consume all the RAM in the free block, so the block is split in two. The unused portion remains available to future calls to pvPortMalloc().

D shows the array after pvPortMalloc() has been called directly from application code rather than indirectly by calling a FreeRTOS API function. The user-allocated block was small enough to fit in the first free block, which is the block between the memory allocated to the queue and the memory allocated to the following TCB. The memory freed when the task was deleted has now been split into three separate blocks. The first block holds the queue. The second block holds the user-allocated memory. The third remains free.

E shows the array after the queue has been deleted, which automatically frees the memory that was allocated to the deleted queue. There is now free memory on either side of the user-allocated block.

F shows the array after the user-allocated memory has also been freed. The memory that had been used by the user-allocated block has been combined with the free memory on either side to create a larger, free block.

Heap_4 is not deterministic, but is faster than most standard library implementations of malloc() and free().

Setting a Start Address for the Array Used by Heap_4

Note: This section contains advanced information. You do not need to read this section in order to use heap_4.

Sometimes an application writer must place the array used by heap_4 at a specific memory address. For example, the stack used by a FreeRTOS task is allocated from the heap, so it might be necessary to ensure the heap is located in fast internal memory rather than slow external memory.

By default, the array used by heap_4 is declared inside the heap_4.c source file. Its start address is set automatically by the linker. However, if the configAPPLICATION_ALLOCATED_HEAP compile time configuration constant is set to 1 in FreeRTOSConfig.h, then the array must be declared by the application using FreeRTOS. If the array is declared as part of the application, then the application's writer can set its start address.

If configAPPLICATION_ALLOCATED_HEAP is set to 1 in FreeRTOSConfig.h, then a uint8_t array called ucHeap and dimensioned by the configTOTAL_HEAP_SIZE setting must be declared in one of the application's source files.

The syntax required to place a variable at a specific memory address is depends on the compiler. For information, see the documentation for your compiler.

Examples for two compilers follow.

Here is the syntax required by the GCC compiler to declare the array and place it in a memory allocation called .my_heap:

```
uint8_t ucHeap[ configTOTAL_HEAP_SIZE ] __attribute__ ( (section( ".my_heap" ) ) );
```

Here is the syntax required by the IAR compiler to declare the array and place it at the absolute memory address 0x20000000:

```
uint8_t ucHeap[ configTOTAL_HEAP_SIZE ] @ 0x20000000;
```

Heap_5

The algorithm used by heap_5 to allocate and free memory is identical to the one used by heap_4. Unlike heap_4, heap_5 is not limited to allocating memory from a single statically declared array. Heap_5 can allocate memory from multiple and separated memory spaces. Heap_5 is useful when the RAM provided by the system on which FreeRTOS is running does not appear as a single contiguous (without space) block in the system's memory map.

Heap_5 is the only provided memory allocation scheme that must be explicitly initialized before pvPortMalloc() can be called. It is initialized using the vPortDefineHeapRegions() API function. When heap_5 is used, vPortDefineHeapRegions() must be called before any kernel objects (tasks, queues, semaphores, and so on) can be created.

vPortDefineHeapRegions() API Function

The vPortDefineHeapRegions() is used to specify the start address and size of each separate memory area. Together they make up the total memory used by heap_5.

void vPortDefineHeapRegions(const HeapRegion_t * const pxHeapRegions);

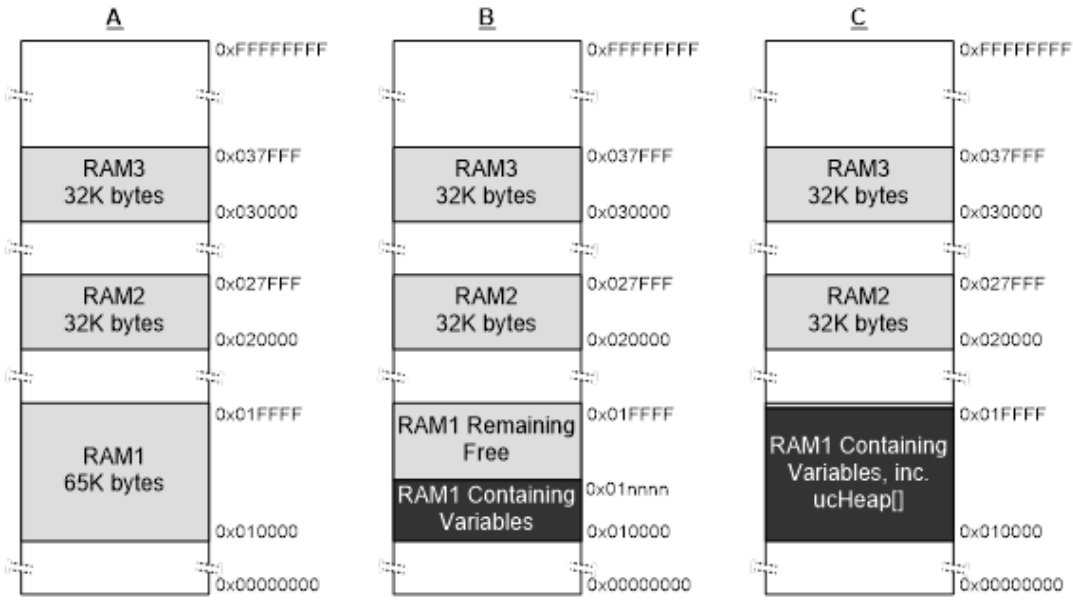
Each separate memory area is described by a structure of type HeapRegion_t. A description of all the available memory areas is passed into vPortDefineHeapRegions() as an array of HeapRegion_t structures.

```
typedef struct HeapRegion

{

/* The start address of a block of memory that will be part of the heap.*/

uint8_t *pucStartAddress;

/* The size of the block of memory in bytes. */

size_t xSizeInBytes;

} HeapRegion_t;
```

The following table lists the vPortDefineHeapRegions() parameters.

Parameter Name/ Returned Value	Description
pxHeapRegions	A pointer to the start of an array of HeapRegion_t structures. Each structure in the array describes the start address and length of a memory area that will be part of the heap when heap_5 is used. The HeapRegion_t structures in the array must be ordered by start address. The HeapRegion_t structure that describes the memory area with the lowest start address must be the first structure in the array, and the HeapRegion_t structure that describes the memory area with the highest start address must be the last structure in the array. The end of the array is marked by a HeapRegion_t structure that has its pucStartAddress member set to NULL.

By way of example, consider the hypothetical memory map shown in the following figure, which contains three separate blocks of RAM: RAM1, RAM2, and RAM3. Executable code is placed in read-only memory, but not shown.



The following code shows an array of HeapRegion_t structures. Together they describe the three blocks of RAM.

```
/* Define the start address and size of the three RAM regions. */

#define RAM1_START_ADDRESS ( ( uint8_t * ) 0x00010000 )

#define RAM1_SIZE ( 65 * 1024 )

#define RAM2_START_ADDRESS ( ( uint8_t * ) 0x00020000 )

#define RAM2_SIZE ( 32 * 1024 )

#define RAM3_START_ADDRESS ( ( uint8_t * ) 0x00030000 )

#define RAM3_SIZE ( 32 * 1024 )

/* Create an array of HeapRegion_t definitions, with an index for each of the three RAM
 regions, and terminating the array with a NULL address. The HeapRegion_t structures must
 appear in start address order, with the structure that contains the lowest start address
 appearing first. */

const HeapRegion_t xHeapRegions[] =

{

    { RAM1_START_ADDRESS, RAM1_SIZE },

    { RAM2_START_ADDRESS, RAM2_SIZE },

    { RAM3_START_ADDRESS, RAM3_SIZE },

    { NULL, 0 } /* Marks the end of the array. */

};

int main( void )
```

```
{

/* Initialize heap_5. */

vPortDefineHeapRegions( xHeapRegions );

/* Add application code here. */

}
```

Although the code correctly describes the RAM, this is not a usable example because it allocates all the RAM to the heap, leaving no RAM free for use by other variables.

When a project is built, the linking phase of the build process allocates a RAM address to each variable. The RAM available for use by the linker is normally described by a linker configuration file, such as a linker script. In **B** of the preceding figure, it is assumed the linker script included information on RAM1, but not on RAM2 or RAM3. The linker has therefore placed variables in RAM1, leaving only the portion of RAM1 above address 0x0001nnnn available for use by heap_5. The actual value of 0x0001nnnn will depend on the combined size of all the variables included in the application being linked. The linker has left all of RAM2 and RAM3 unused, so they are available for use by heap_5.

If the preceding code were used, the RAM allocated to heap_5 below address 0x0001nnnn would overlap the RAM used to hold variables. To avoid that, the first HeapRegion_t structure in the xHeapRegions[] array could use a start address of 0x0001nnnn rather than 0x00010000.

This is not a recommended solution because:

- The start address might not be easy to determine.
- The amount of RAM used by the linker might change in future builds, necessitating an update to the start address used in the HeapRegion_t structure.
- The build tools will not know and therefore cannot warn the application writer if the RAM used by the heap_5 overlap.

The following code demonstrates a more convenient and maintainable example. It declares an array called ucHeap. ucHeap is a normal variable, so it becomes part of the data allocated to RAM1 by the linker. The first HeapRegion_t structure in the xHeapRegions array describes the start address and size of ucHeap, so ucHeap becomes part of the memory managed by heap_5. The size of ucHeap can be increased until the RAM used by the linker consumes all of RAM1, as shown in **C** in the preceding figure.

```
/* Define the start address and size of the two RAM regions not used by the linker. */

#define RAM2_START_ADDRESS ( ( uint8_t * ) 0x00020000 )

#define RAM2_SIZE ( 32 * 1024 )

#define RAM3_START_ADDRESS ( ( uint8_t * ) 0x00030000 )

#define RAM3_SIZE ( 32 * 1024 )

/* Declare an array that will be part of the heap used by heap_5. The array will be placed
 in RAM1 by the linker. */

#define RAM1_HEAP_SIZE ( 30 * 1024 )

static uint8_t ucHeap[ RAM1_HEAP_SIZE ];

/* Create an array of HeapRegion_t definitions. Whereas in previous code listing, the first
 entry described all of RAM1, so heap_5 will have used all of RAM1, this time the first
```

```
 entry only describes the ucHeap array, so heap_5 will only use the part of RAM1 that
 contains the ucHeap array. The HeapRegion_t structures must still appear in start address
 order, with the structure that contains the lowest start address appearing first. */

const HeapRegion_t xHeapRegions[] =

{

    { ucHeap, RAM1_HEAP_SIZE },

    { RAM2_START_ADDRESS, RAM2_SIZE },

    { RAM3_START_ADDRESS, RAM3_SIZE },

    { NULL, 0 } /* Marks the end of the array. */

};
```

In the preceding code, an array of HeapRegion_t structures describes all of RAM2, all of RAM3, but only part of RAM1.

The advantages of the technique demonstrated here include:

- You do not need to use a hard-coded start address.
- The address used in the HeapRegion_t structure will be set by the linker automatically, so it will always be correct, even if the amount of RAM used by the linker changes in future builds.
- It is not possible for RAM allocated to heap_5 to overlap data placed into RAM1 by the linker.
- The application will not link if ucHeap is too big.

Heap-Related Utility Functions

xPortGetFreeHeapSize() API Function

The xPortGetFreeHeapSize() API function returns the number of free bytes in the heap at the time the function is called. It can be used to optimize the heap size. For example, if xPortGetFreeHeapSize() returns 2000 after all the kernel objects have been created, then the value of configTOTAL_HEAP_SIZE can be reduced by 2000.

xPortGetFreeHeapSize() is not available when heap_3 is used.

The xPortGetFreeHeapSize() API function prototype

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

The following table lists the xPortGetFreeHeapSize() return value.

Parameter Name/ Returned Value	Description
Returned value	The number of bytes that remain unallocated in the heap at the time xPortGetFreeHeapSize() is called.

xPortGetMinimumEverFreeHeapSize() API Function

The xPortGetMinimumEverFreeHeapSize() API function returns the minimum number of unallocated bytes that have existed in the heap since the FreeRTOS application started executing.

The value returned by xPortGetMinimumEverFreeHeapSize() is an indication of how close the application has come to running out of heap space. For example, if xPortGetMinimumEverFreeHeapSize() returns 200, then at some time since the application started executing, it came within 200 bytes of running out of heap space.

xPortGetMinimumEverFreeHeapSize() is available only when heap_4 or heap_5 is used.

The following table lists the xPortGetMinimumEverFreeHeapSize() return value.

Parameter Name/ Returned Value	**Description**
Returned value	The minimum number of unallocated bytes that have existed in the heap since the FreeRTOS application started executing.

Malloc Failed Hook Functions

pvPortMalloc() can be called directly from application code. It is also called within FreeRTOS source files each time a kernel object is created. Kernel objects include tasks, queues, semaphores, and event groups, all of which are described later.

Just like the standard library malloc() function, if pvPortMalloc() cannot return a block of RAM because a block of the requested size does not exist, then it will return NULL. If pvPortMalloc() is executed because the application writer is creating a kernel object, and the call to pvPortMalloc() returns NULL, then the kernel object will not be created.

All the example heap allocation schemes can be configured to call a hook (or callback) function if a call to pvPortMalloc() returns NULL.

If configUSE_MALLOC_FAILED_HOOK is set to 1 in FreeRTOSConfig.h, then the application must provide a malloc failed hook function that has the name and prototype shown here. The function can be implemented in any way that is appropriate for the application.

```
void vApplicationMallocFailedHook( void );
```

Task Management

The concepts presented in this section are fundamental to understanding how to use FreeRTOS and how FreeRTOS applications behave. This section covers:

- 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 in which a task can exist.

This section will also show you:

- 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 using a task. For more information, see software_tier_management.
- When the idle task will execute and how it can be used.

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, as shown here.

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

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.

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 be explicitly deleted.

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

The structure of a typical task is shown here.

```
void ATaskFunction( void *pvParameters )

{
```

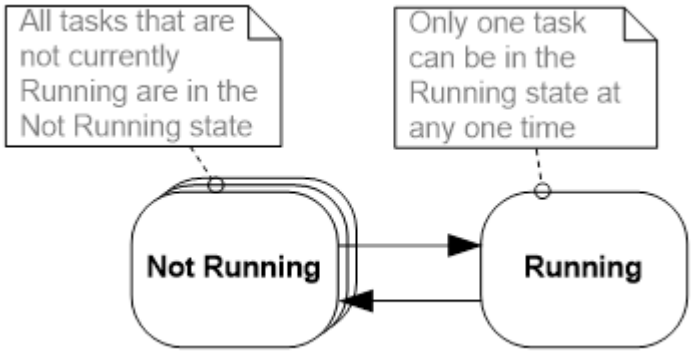
```
/* Variables can be declared just as per a normal function. Each instance of a task created
 using this example function will have its own copy of the lVariableExample variable. This
 would not be true if the variable was declared static, in which case only one copy of the
 variable would exist,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

int32_t lVariableExample = 0;

/* A task will normally be implemented as an infinite loop. */

for( ;; )

{

/* The code to implement the task functionality will go here. */

}

/* Should the task implementation ever break out of the above loop, then the task  must be
 deleted before reaching the end of its implementing function. The NULL parameter passed to
 the vTaskDelete() API function indicates that the task to be deleted is the calling (this)
 task. The convention used to name API functions is described in Data Types and Coding
 Style Guide. */

vTaskDelete( NULL );

}
```

Top-Level Task States

An application can consist of many tasks. If the processor 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. Keep in mind that it is an oversimplification. Later in this section you'll 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 the task's code. When a task is in the Not Running state, the task is dormant, its status having been saved so it is ready 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.

The following figure shows the 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.

Creating Tasks

xTaskCreate() API Function

FreeRTOS V9.0.0 also includes the xTaskCreateStatic() function, which allocates the memory required to create a task statically at compile time. Tasks are created using the FreeRTOS xTaskCreate() API function. This is probably the most complex of all the API functions. Because tasks are the most fundamental component of a multitasking system, they must be mastered first. There are many examples of the xTaskCreate() function in this guide.

For information about the data types and naming conventions used, see Data Types and Coding Style Guide in FreeRTOS Kernel Distribution (p. 5).

The following shows the xTaskCreate() API function prototype.

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

Here is a list of xTaskCreate() parameters and return values.

- 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 that implements the task (in effect, just the name of the function).
- pcName: A descriptive name for the task. This is not used by FreeRTOS in any way. It is included only 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.
- 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, if the stack is 32 bits wide and 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 uint16_t. The size of the stack used by the Idle task is defined by the application-defined constant configMINIMAL_STACK_SIZE. This is the only way the FreeRTOS source code uses the configMINIMAL_STACK_SIZE setting. The constant is also used inside demo applications to help make the demos portable across multiple processor architectures. The value assigned to this constant in the FreeRTOS demo application for the processor architecture being used 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, and then use FreeRTOS features to ensure that the space allocated is indeed adequate, and that RAM is not being wasted. For information about how to query the maximum stack space that has been used by a task, see Stack Overflow (p. 239) in the Troubleshooting section.
- pvParameters: Task functions accept a parameter of type pointer to void (void*). The value assigned to pvParameters is the value passed into the task. There are examples in this guide that demonstrate how the parameter can be used.
- uxPriority: Defines the priority at which the task will execute. Priorities can be assigned from 0, which is the lowest priority, to (configMAX_PRIORITIES - 1), which is the highest priority.

configMAX_PRIORITIES is a user-defined constant that is described in Task Priorities (p. 33). 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: This parameter 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.

There are two possible return values: pdPASS, which indicates that the task has been created successfully and pdFAIL, which 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. For more information, see Heap Memory Management (p. 14).

Creating Tasks (Example 1)

This example demonstrates the steps required to create two simple tasks and then start executing them. 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. The following code shows the implementation of the first task used in Example 1.

```
 void vTask1( void *pvParameters )

{

 const char *pcTaskName = "Task 1 is running\r\n";

 volatile uint32_t ul; /* volatile to ensure ul is not optimized away.*/

 /* 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. */

         }

     }

 }
```

The following code shows the implementation of the second task used in Example 1.

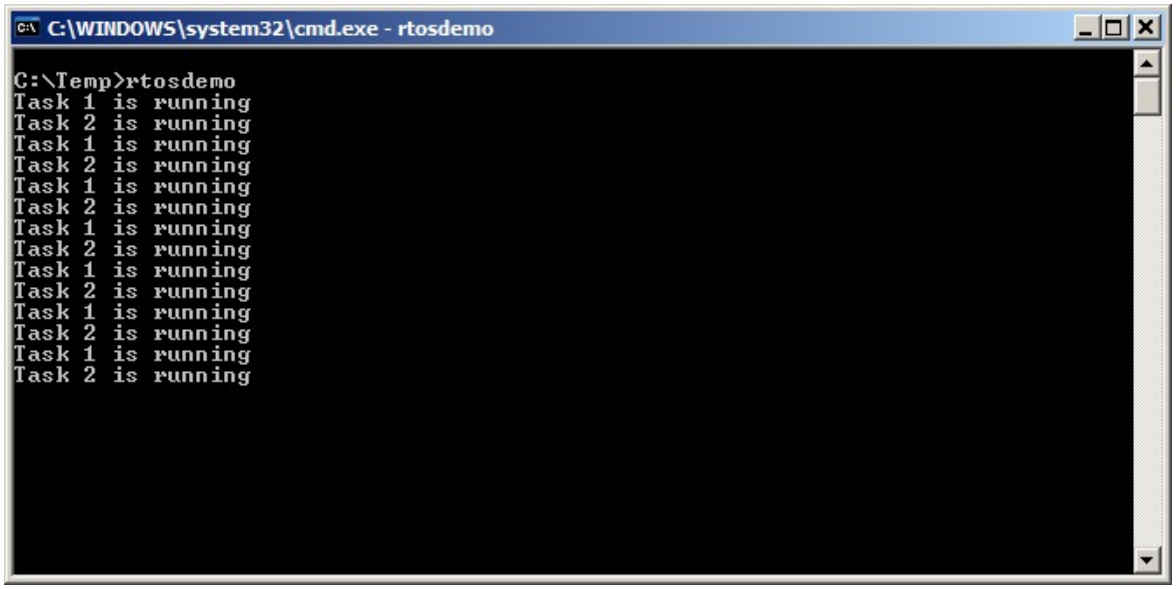
```
void vTask2( void *pvParameters )

{

    const char *pcTaskName = "Task 2 is running\r\n";
```

```
    volatile uint32_t ul; /* volatile to ensure ul is not optimized away.*/

    /* As per most tasks, this task is implemented in an infinite loop. */

    for( ;; )

    {

        /* Print out the name of this task. */

        vPrintString( pcTaskName );

        /* Delay for a period. */

        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )

        {

            /* This loop is just a very crude delay implementation. There is nothing to do
 in here. Later examples will replace this crude loop with a proper delay/sleep function.
 */

        }

    }

}
```

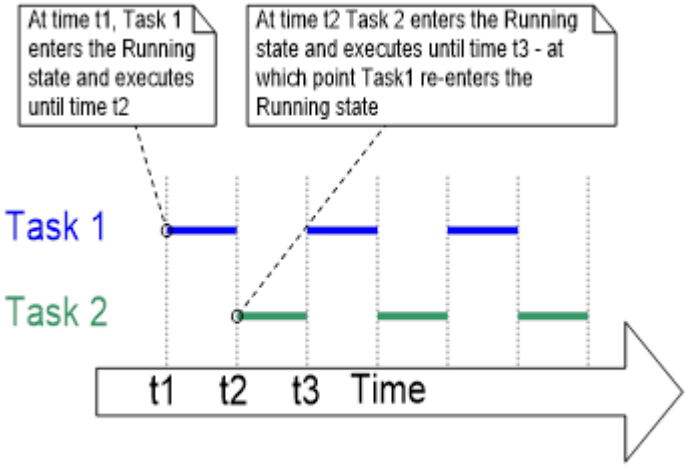
The main() function creates the tasks before starting the scheduler.

```
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 thetask. */ "Task 1",/
* Text name for the task. This is to facilitate debugging only. */ 1000, /* Stack depth -
 small microcontrollers will use much less stack than this. */ NULL, /* This example does
 not use the task parameter. */ 1, /* This task will run at priority 1. */ NULL ); /* This
 example does not use the task handle. */

            /* Create the other task in exactly the same way and at the same priority. */

    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    /* Start the scheduler so the tasks start executing. */

    vTaskStartScheduler();

    /* If all is well then main() will never reach here as the scheduler will now be
 running the tasks. If main() does reach here then it is likely that there was insufficient
 heap memory available for the idle task to be created. For more information, see Heap
 Memory Management. */

    for( ;; );

}
```

The output is shown here.



The two tasks appear to execute simultaneously. However, because both tasks are executing on the same processor core, 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 same processor core. Their execution pattern is shown in the following figure.



The arrow along the bottom of this figure shows the passing of time from time t1 onward. 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).

Both tasks are created from within main(), prior to starting the scheduler. It is also possible to create a task from within another task. For example, Task 2 could have been created from within Task 1.

The following code shows a task created from within another task after the scheduler has started.

```
void vTask1( void *pvParameters )
```

```
{

    const char *pcTaskName = "Task 1 is running\r\n";

    volatile uint32_t ul; /* volatile to ensure ul is not optimized away.*/

    /* If this task code is executing then the scheduler must already have been started.
 Create the other task before entering the infinite loop. */

    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    for( ;; )

    {

        /* Print out the name of this task. */

        vPrintString( pcTaskName );

        /* Delay for a period. */

        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )

        {

            /* This loop is just a very crude delay implementation. There is nothing to do
 in here. Later examples will replace this crude loop with a proper delay/sleep function.
 */

        }

    }

}
```

Using the Task Parameter (Example 2)

The two tasks created in Example 1 are almost identical. The only difference is the text string they print out. This duplication can be removed by 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.

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

```
void vTaskFunction( void *pvParameters )

{

    char *pcTaskName;

    volatile uint32_t ul; /* volatile to ensure ul is not optimized away.*/

    /* The string to print out is passed in via the parameter. Cast this to a character
 pointer. */

    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */

    for( ;; )

    {
```

```
        /* Print out the name of this task. */

        vPrintString( pcTaskName );

        /* Delay for a period. */

        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )

        {

            /* This loop is just a very crude delay implementation. There is nothing to do
 in here. Later exercises will replace this crude loop with a delay/sleep function. */

        }

    }

}
```

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 following code shows how the pvParameters parameter to the xTaskCreate() function is used to pass the text string into the task.

```
/* Define the strings that will be passed in as the task parameters. These are defined
 const and not on the stack to ensure they remain valid when the tasks are executing. */

static const char *pcTextForTask1 = "Task 1 is running\r\n";

static const char *pcTextForTask2 = "Task 2 is running\r\n";

int main( void )

{

    /* Create one of the two tasks. */

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

    /* Create the other task in exactly the same way. Note this time that multiple tasks
 are being created from the SAME task implementation (vTaskFunction). Only the value passed
 in the parameter is different. Two instances of the same task are being created. */

    xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 1, NULL );

    /* Start the scheduler so the tasks start executing. */

    vTaskStartScheduler();

    /* If all is well then main() will never reach here as the scheduler will now be
 running the tasks. If main() does reach here then it is likely that there was insufficient
 heap memory available for the idle task to be created. For more information, see Heap
 Memory Management. */

    for( ;; );
```

```
}
```

Task Priorities

The uxPriority parameter of the xTaskCreate() API function assigns an initial priority to the task being created. You can change the priority 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. 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). Any number of tasks can share the same priority, ensuring maximum design flexibility.

The FreeRTOS scheduler can use one of two methods to decide which task will be in the Running state. The maximum value to which configMAX_PRIORITIES can be set depends on the method used:

1. Generic method

   This method is implemented in C and can be used with all the FreeRTOS architecture ports.

   When the generic method is used, FreeRTOS does not limit the maximum value to which configMAX_PRIORITIES can be set. However, we recommend you keep the configMAX_PRIORITIES value at the minimum required because the higher its value, the more RAM will be consumed, and the longer the worst-case execution time will be.

   This method will be used if configUSE_PORT_OPTIMISED_TASK_SELECTION is set to 0 in FreeRTOSConfig.h, or if configUSE_PORT_OPTIMISED_TASK_SELECTION is left undefined, or if the generic method is the only method provided for the FreeRTOS port in use.
2. Architecture-optimized method

   This method uses a small amount of assembler code, and is faster than the generic method. The configMAX_PRIORITIES setting does not affect the worst case execution time.

   If you use this method, then configMAX_PRIORITIES cannot be greater than 32. As with the generic method, you should keep configMAX_PRIORITIES at the minimum required because the higher its value, the more RAM will be consumed.

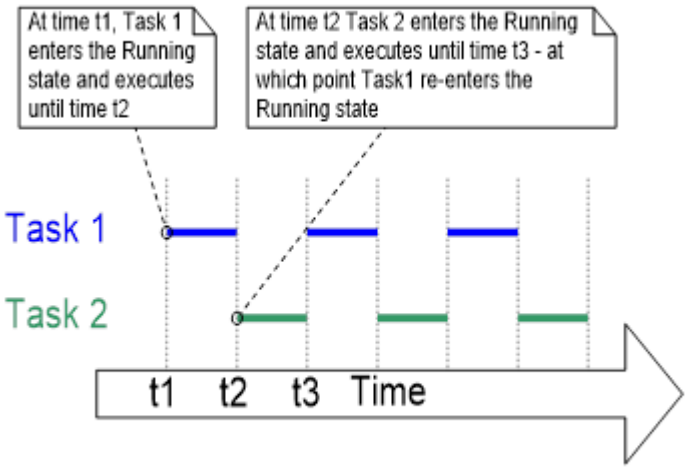
   This method will be used if configUSE_PORT_OPTIMISED_TASK_SELECTION is set to 1 in FreeRTOSConfig.h.

   Not all FreeRTOS ports provide an architecture-optimized method.

The FreeRTOS 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.

Time Measurement and the Tick Interrupt

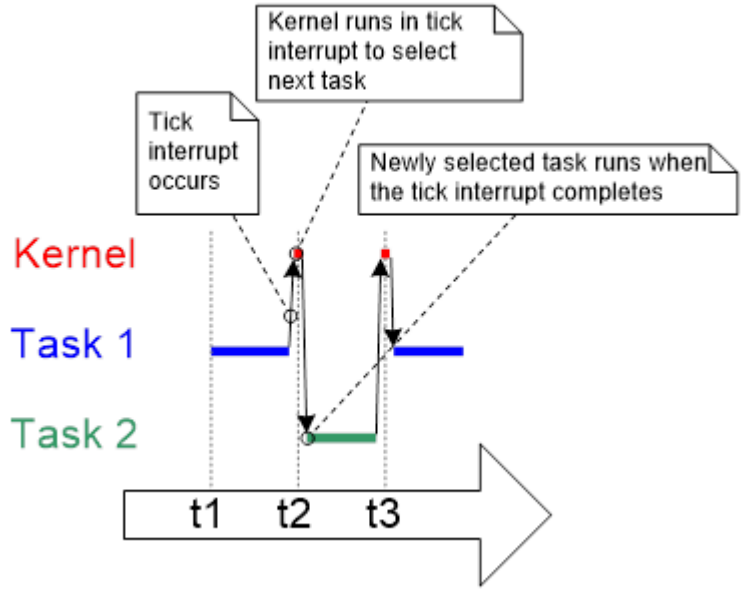
The Scheduling Algorithms (p. 56) section describes an optional feature called *time slicing*. Time slicing has been used in the examples presented so far. It is the behavior observed in the output they produced. In the examples, both tasks were created at the same priority, and both tasks were always able to run. Therefore, each task executed for a time slice, entering the Running state at the start of a time slice and exiting the Running state at the end of a time slice. In this figure, 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. The end of a time slice is not the only place that the scheduler can select a new task to run. The scheduler will also select a new task to run immediately after the currently executing task enters the Blocked state, or when an interrupt moves a higher priority task into the Ready state. 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 application-defined configTICK_RATE_HZ compile-time configuration constant within FreeRTOSConfig.h. For example, if configTICK_RATE_HZ is set to 100 (Hz), then the time slice will be 10 milliseconds. The time between two tick interrupts is called the *tick period*. One time slice equals one tick period.

The following figure shows the execution sequence expanded to show the tick interrupt executing. The top line shows when the scheduler is executing. The thin arrows show the sequence of execution from a task to the tick interrupt, and then from the tick interrupt back to a different task.

The optimal value for configTICK_RATE_HZ depends on the application being developed, but a value of 100 is typical.



FreeRTOS API calls always specify time in multiples of tick periods, which are often referred to simply as *ticks*. The pdMS_TO_TICKS() macro converts a time specified in milliseconds into a time specified in ticks. The resolution available depends on the defined tick frequency. pdMS_TO_TICKS() cannot be used if the tick frequency is above 1 KHz (if configTICK_RATE_HZ is greater than 1000). The following code shows how to use pdMS_TO_TICKS() to convert a time specified as 200 milliseconds into an equivalent time specified in ticks.

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

TickType_t xTimeInTicks = pdMS_TO_TICKS( 200 );
```

Note: We do not recommend that you specify times in ticks directly in the application. Instead, use the pdMS_TO_TICKS() macro to specify times in milliseconds. This ensures times specified in the application do not change if the tick frequency is changed.

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 because time consistency is managed internally by FreeRTOS.

For information about configuration constants that affect when the scheduler will select a new task to run and when a tick interrupt will execute, see Scheduling Algorithms (p. 56).

Experimenting with Priorities (Example 3)

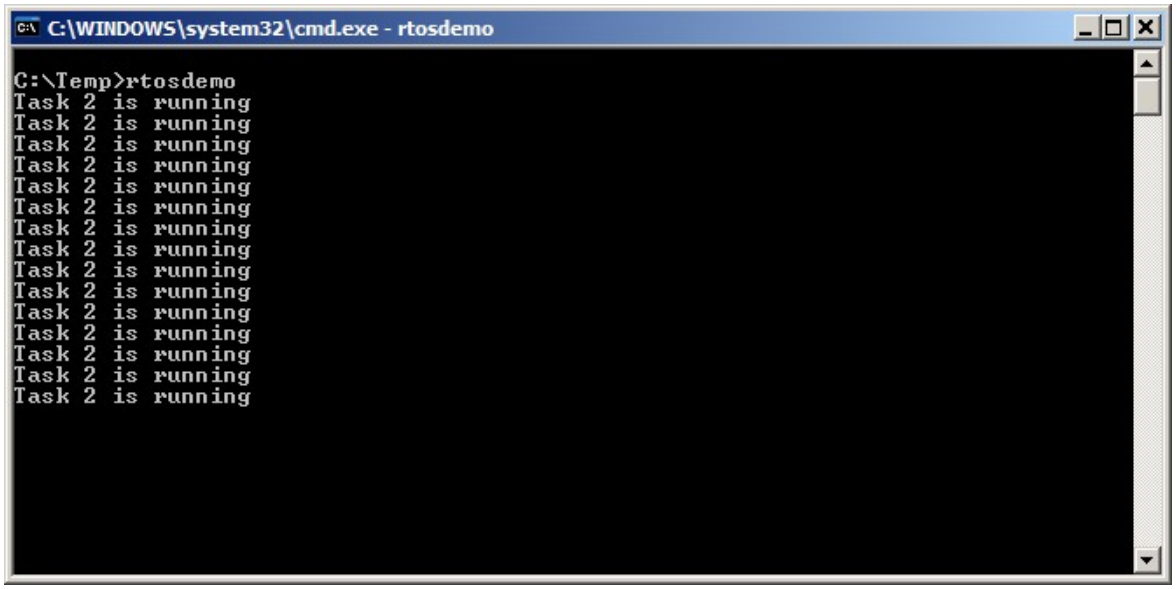
The scheduler will always ensure that the highest priority task that is able to run is the task selected to enter the Running state. In the examples used so far, two tasks have been created at the same priority, so both entered and exited the Running state in turn. This example shows what happens when the priority of one of the two tasks created in Example 2 is changed. This time, the first task is created at priority 1, and the second at priority 2.

The sample code to create the tasks at different priorities is shown here. 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\r\n";

static const char *pcTextForTask2 = "Task 2 is running\r\n";

int main( void )

{

    /* Create the first task at priority 1. The priority is the second to last parameter.
 */

    xTaskCreate( vTaskFunction, "Task 1", 1000, (void*)pcTextForTask1, 1, NULL );

    /* Create the second task at priority 2, which is higher than a priority of 1. The
 priority is the second to last parameter. */

    xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 2, NULL );

    /* Start the scheduler so the tasks start executing. */
```

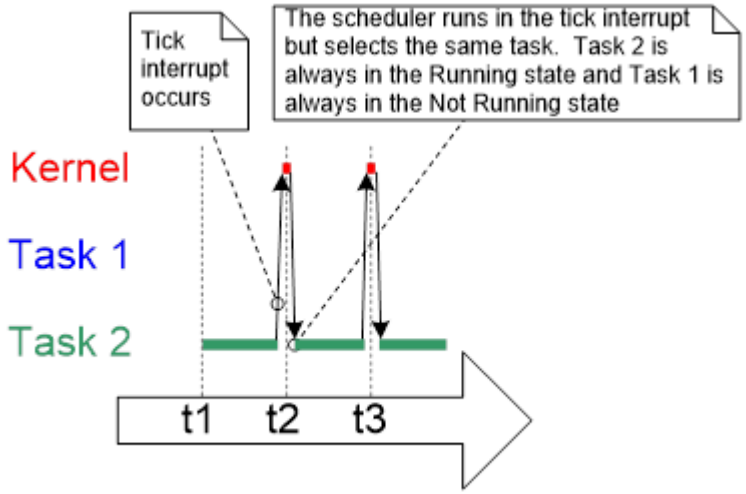
```
    vTaskStartScheduler();

    /* Will not reach here. */

    return 0;

}
```

The output produced by this example is shown here. 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. Because 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.



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.

The following figure shows the execution sequence for the preceding sample code.



Expanding the Not Running State

So far, the created tasks have always had processing to perform and have never had to wait for anything. Because 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 the tasks can only be created at the very lowest priority. If they run at any other priority, they will prevent tasks of lower priority from ever running at all.

To make the tasks useful, they must be rewritten 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 types of events:

1. Temporal (time-related) events, where the events are either a delay period expiring or an absolute time being reached. For example, a task can 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 can 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, mutexes, recursive mutexes, event groups, and direct-to-task notifications can all be used to create synchronization events.

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 can 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 of the Suspended state is through a call to the vTaskResume() or xTaskResumeFromISR() API functions. Most applications do not use the Suspended state.

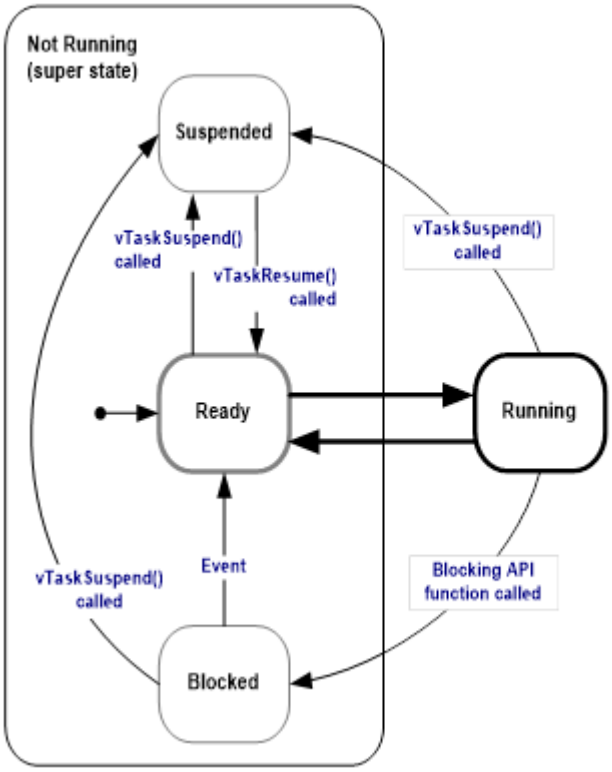
The Ready State

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

The following figure expands on the oversimplified 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, as shown by the bold lines here.



Using the Blocked State to Create a Delay (Example 4)

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. The higher priority task remained in the Running state while it executed the null loop, starving the lower priority 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 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. The task does not use any processing time while it is in the Blocked state, so the task only uses processing time when there is actually work to be done.

The vTaskDelay() API function prototype is shown here.

```
void vTaskDelay( TickType_t xTickToDelay );
```

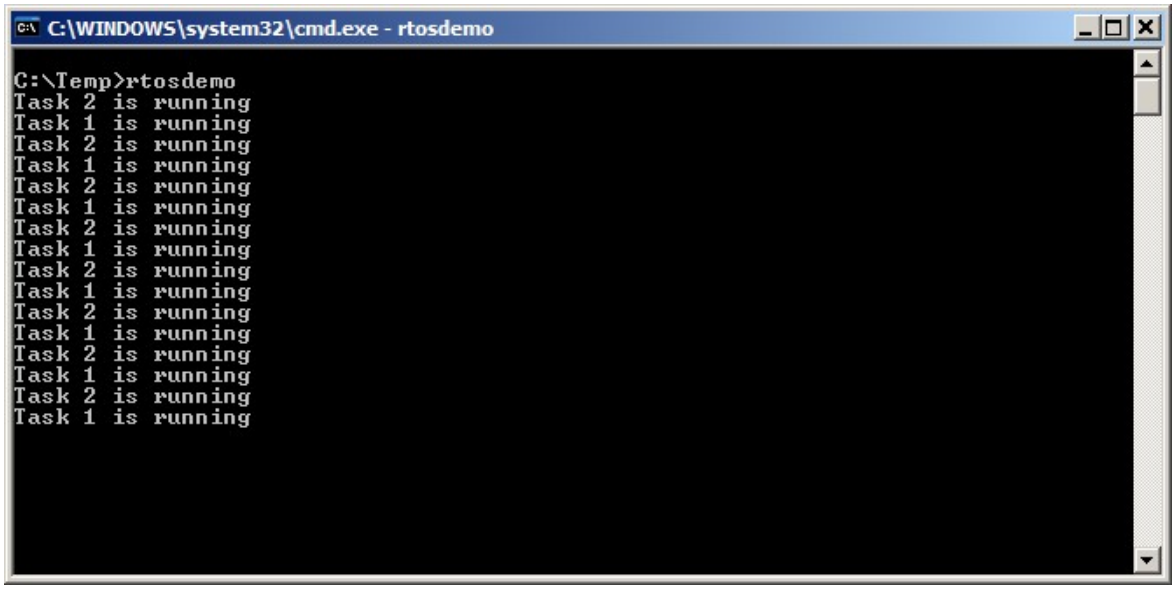
The following table lists the vTaskDelay() parameter.

Parameter Name	Description
xTicksToDelay	The number of tick interrupts for which the calling task will remain in the Blocked state before being transitioned back into the Ready state. For example, if a task called vTaskDelay(100) when the tick count was 10,000, then it would immediately enter the Blocked state, and remain in the Blocked state until the tick count reached 10,100. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into ticks. For example, calling vTaskDelay(pdMS_TO_TICKS(100)) will result in the calling task remaining in the Blocked state for 100 milliseconds.

In the following code, the example task after the null loop delay has been replaced by a call to vTaskDelay(). This code shows the new task definition.

```
void vTaskFunction( void *pvParameters )

{

    char *pcTaskName;

    const TickType_t xDelay250ms = pdMS_TO_TICKS( 250 );

    /* The string to print out is passed in via the parameter. Cast this to a character
 pointer. */

    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */

    for( ;; )

    {

        /* Print out the name of this task. */

        vPrintString( pcTaskName );

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

        vTaskDelay( xDelay250ms );

    }

}
```

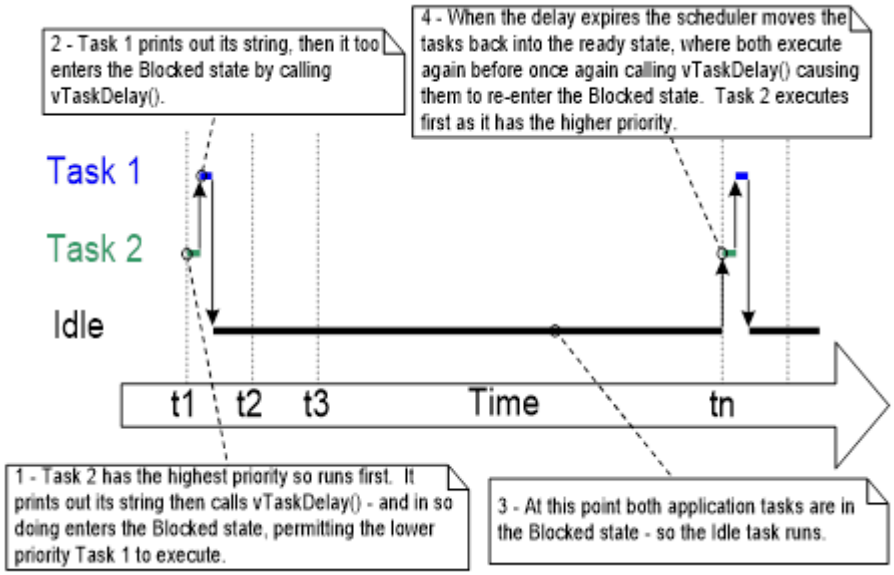
Even though the two tasks are still being created at different priorities, both will now run. The output confirms the expected behavior.



The execution sequence shown in the following figure explains why both tasks run, even though they are created at different priorities. The execution of the scheduler itself is omitted for simplicity.

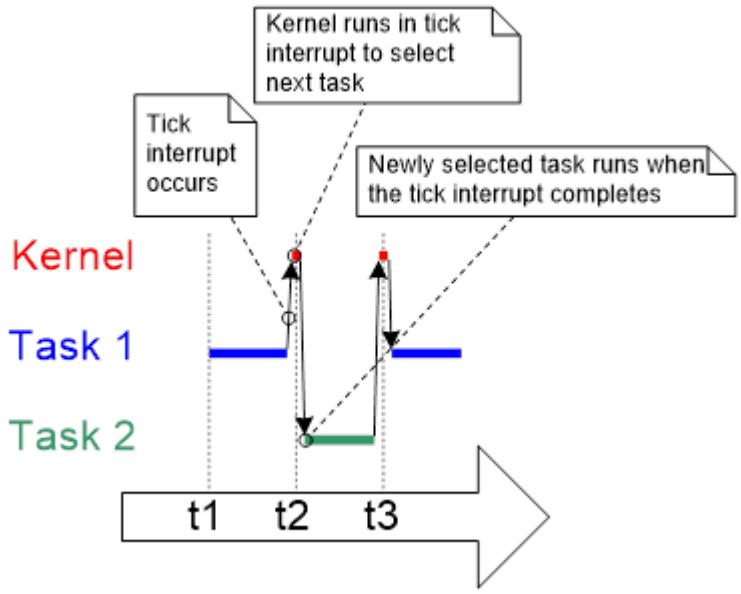
The idle task is created automatically when the scheduler is started to ensure there is always at least one task that is able to run (at least one task in the Ready state). For more information about this task, see The Idle Task and the Idle Task Hook (p. 46).

This figure shows the execution sequence when the tasks use vTaskDelay() in place of the NULL loop.



Only the implementation of the two tasks has changed, not their functionality. If you compare this figure with the figure in Time Measurement and the Tick Interrupt (p. 33), you'll see that this functionality is being achieved in a much more efficient manner.

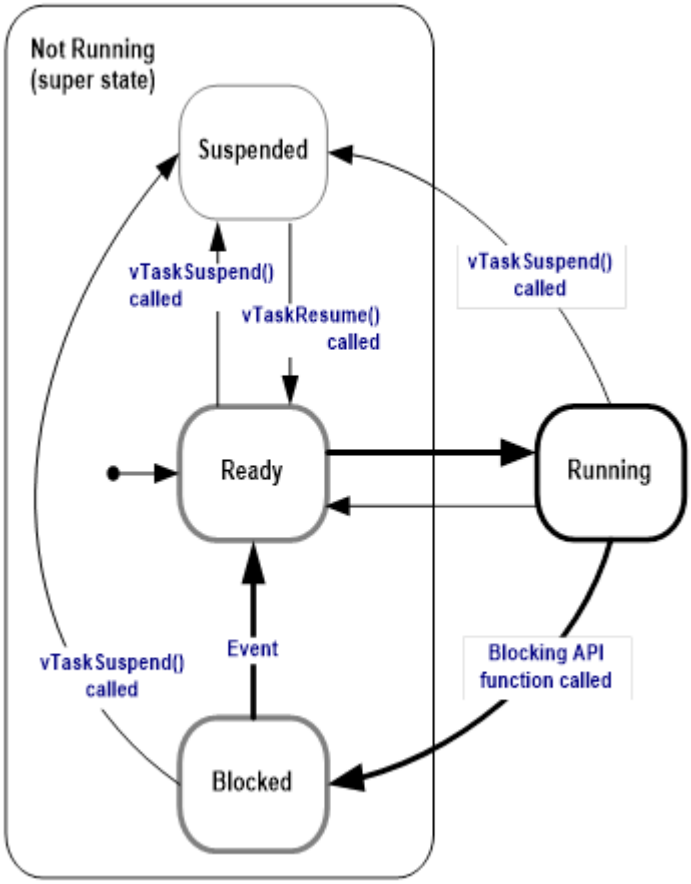
This figure shows the execution pattern when the tasks enter the Blocked state for the entirety of their delay period, so they use processor time only when they actually have work that must be performed (in this case, simply printing out a message).



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 allocated to the idle task is a measure of the spare processing capacity in the system. Using an RTOS can significantly increase the spare processing capacity simply by allowing an application to be completely event-driven.

The bold lines in the following figure 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.

Bold lines indicate the state transitions performed by the tasks.



vTaskDelayUntil() API Function

vTaskDelayUntil() is similar to vTaskDelay(). 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 time at which the task leaves the Blocked state is relative to the time at which vTaskDelay() was called.

The parameters to vTaskDelayUntil() specify the exact tick count value at which the calling task should be moved from the Blocked state into the Ready state. vTaskDelayUntil() is the API function that should be used when a fixed execution period is required (where you want your task to execute periodically with a fixed frequency) because 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()).

The vTaskDelayUntil() API function prototype is shown here.

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

The following table lists the 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, but must be initialized to the current tick count before it is used for the first time.
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 is set by the xTimeIncrement value. xTimeIncrement is specified in ticks. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into ticks.

Converting the Example Tasks to Use vTaskDelayUntil() (Example 5)

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 because 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.

The following code shows the implementation of the example task using vTaskDelayUntil().

```
void vTaskFunction( void *pvParameters )

    {

        char *pcTaskName;

        TickType_t xLastWakeTime;

        /* The string to print out is passed in by the parameter. Cast this to a character
 pointer. */

        pcTaskName = ( char * ) pvParameters;

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

        xLastWakeTime = xTaskGetTickCount();

        /* As per most tasks, this task is implemented in an infinite loop. */

        for( ;; )

        {
```

```
            /* Print out the name of this task. */

            vPrintString( pcTaskName );

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

            vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 250 ) );

        }

}
```

Combining Blocking and Non-Blocking Tasks (Example 6)

Earlier examples have examined the behavior of both polling and blocking tasks in isolation. This example reinforces the stated expected system behavior by demonstrating an execution sequence when the two schemes are combined.

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

   These tasks never make any API function calls that could cause them to enter the Blocked state, so they are always in either the Ready or the Running state. Tasks like this are called *continuous processing tasks* because they always have work to do (even if in this case it's trivial work).
2. A third task is then created at priority 2, so above the priority of the other two tasks. The third task also just prints out a string, but this time periodically, so it uses the vTaskDelayUntil() API function to place itself into the Blocked state between each print iteration.

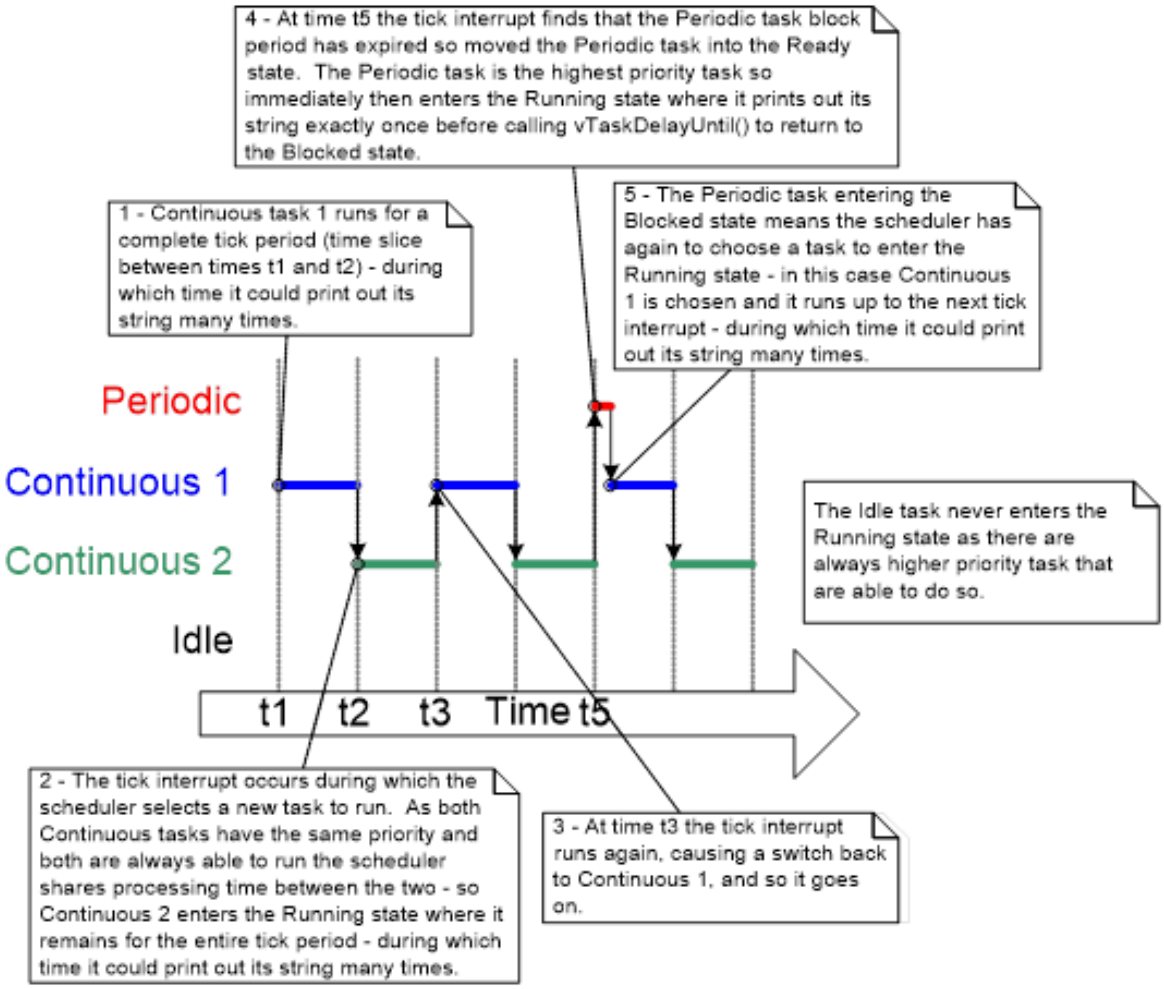
The following code shows the continuous processing task.

```
void vContinuousProcessingTask( void *pvParameters )

{

    char *pcTaskName;

    /* The string to print out is passed in by 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 );

    }

}
```

The following code shows the periodic task.

```
void vPeriodicTask( void *pvParameters )

{

    TickType_t xLastWakeTime;

    const TickType_t xDelay3ms = pdMS_TO_TICKS( 3 );

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

    xLastWakeTime = xTaskGetTickCount();

    /* As per most tasks, this task is implemented in an infinite loop. */

    for( ;; )

    {

        /* Print out the name of this task. */

        vPrintString( "Periodic task is running\r\n" );

        /* The task should execute every 3 milliseconds exactly. See the declaration of
 xDelay3ms in this function. */

        vTaskDelayUntil( &xLastWakeTime, xDelay3ms );

    }

}
```

The following figure shows the execution sequence.



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, so they cannot be selected by the scheduler.

There must always be at least one task that can enter the Running state. This is the case even when the special low-power features of FreeRTOS are being used. The microcontroller on which FreeRTOS is executing will be placed into a low power mode if none of the tasks created by the application are able to execute.

To ensure that at least one task can enter the Running state, the scheduler creates an idle task when vTaskStartScheduler() is called. The idle task does little more than sit in a loop so, like the tasks in the first 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. However, there is nothing to prevent you from creating tasks at, and therefore sharing, the idle task priority. The configIDLE_SHOULD_YIELD compile-time configuration constant in FreeRTOSConfig.h can be used to prevent the idle task from consuming processing time that would be more productively allocated to applications tasks. For more information about configIDLE_SHOULD_YIELD, see Scheduling Algorithms (p. 56).

Running at the lowest priority ensures the idle task is transitioned out of the Running state as soon as a higher priority task enters the Ready state. This can be seen at time tn in the execution sequence figure for Example 4, where the idle task is immediately swapped out to allow Task 2 to execute as soon as it leaves the Blocked state. Task 2 is said to have preempted the idle task. Preemption occurs automatically and without the knowledge of the task being preempted.

Note: If an application uses the vTaskDelete() API function, then it is essential that the idle task is not starved of processing time. This is because the idle task is responsible for cleaning up kernel resources after a task has been deleted.

Idle Task Hook Functions

You can add application-specific functionality directly into the idle task through the use of an idle hook (or idle callback) function. This is 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 functionality.
- Measuring the amount of spare processing capacity. (The idle task will run only when all higher priority application 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 easy and automatic method for saving power whenever there is no application processing to be performed. (You can save more power by using the tickless idle mode.)

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.

   Note: Blocking the idle task in any way can cause a scenario in which 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 cleanup cannot occur.

Idle task hook functions must have the name and prototype shown here.

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

Defining an Idle Task Hook Function (Example 7)

The use of blocking vTaskDelay() API calls in Example 4 created 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 here.

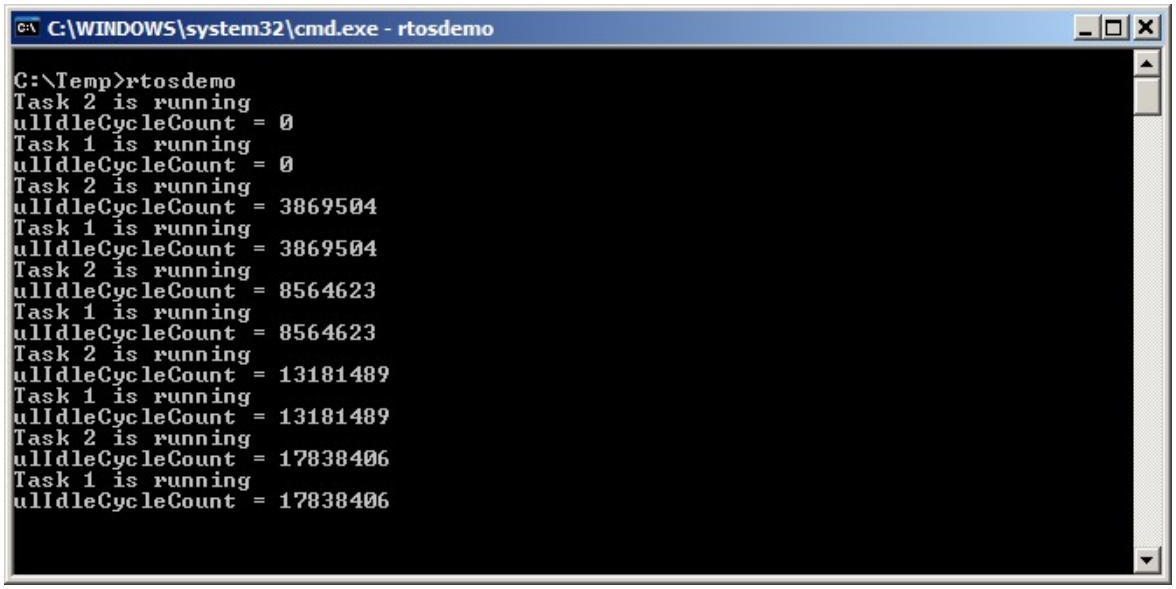
This following code describes a very simple idle hook function.

```
/* Declare a variable that will be incremented by the hook function.*/

volatile uint32_t 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++;

}



To call the idle
hook function, calledconfigUSE_IDLE_HOOK must be set to 1 in FreeRTOSConfig.h.
```

The function that implements the created tasks is modified slightly to print out the ulIdleCycleCount value, as shown here. The following code shows how to print out the ulIdleCycleCount value.

```
void vTaskFunction( void *pvParameters )

{

    char *pcTaskName;

    const TickType_t xDelay250ms = pdMS_TO_TICKS( 250 );

    /* The string to print out is passed in by 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( xDelay250ms );

    }

}
```

The output produced by Example 7 is shown here. You'll see the idle task hook function is called approximately 4 million times between each iteration of the application tasks. (The number of iterations depends on the speed of the hardware on which the demo is executed.)



Changing the Priority of a Task

vTaskPrioritySet() API Function

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

The following shows the vTaskPrioritySet() API function prototype.

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

The following table lists the vTaskPrioritySet() parameters.

Parameter Name	Description
pxTask	The handle of the task whose priority is being modified (the subject task). For information about obtaining handles to tasks, see the pxCreatedTask parameter of the xTaskCreate() API function. 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 constant set in the FreeRTOSConfig.h header file.

uxTaskPriorityGet() API Function

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

The following shows the uxTaskPriorityGet() API function prototype.

```
UBaseType_t uxTaskPriorityGet( TaskHandle_t pxTask );
```

The following table lists the uxTaskPriorityGet() parameter and return value.

Parameter Name/ Return Value	Description
pxTask	The handle of the task whose priority is being queried (the subject task). For information about obtaining handles to tasks, see the pxCreatedTask parameter of the xTaskCreate() API function. 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.

Changing Task Priorities (Example 8)

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.

The sample code used here creates two tasks 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. Therefore, the task with the highest relative priority will always be the task selected by the scheduler to be in the Running state.

1. Task 1 (shown in the code immediately following) 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 (shown second) 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 down to below that of Task 1.
4. When Task 2 sets its priority back down, Task 1 is once again the highest priority task. Task 1 re-enters the Running state, forcing Task 2 back into the Ready state.

```
/*The implementation of Task 1*/
void vTask1( void *pvParameters )

{

    UBaseType_t uxPriority;

    /* This task will always run before Task 2 because it is created with the higher
 priority. Task 1 and Task 2 never block, so both will always be in either the Running or
```

```
 the Ready state. Query the priority at which this task is running. Passing in NULL means
 "return the calling task's priority". */

    uxPriority = uxTaskPriorityGet( NULL );

    for( ;; )

    {

        /* Print out the name of this task. */

        vPrintString( "Task 1 is running\r\n" );

        /* Setting the Task 2 priority above the Task 1 priority will cause Task 2 to
 immediately start running (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(). The
 code that follows shows how the handle was obtained. */

        vPrintString( "About to raise the Task 2 priority\r\n" );

        vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );

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

    }

}
```

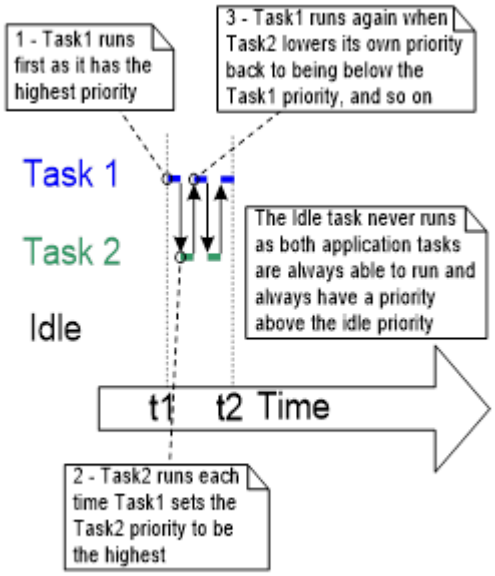
```
/*The implementation of Task 2 */

void vTask2( void *pvParameters )

{

    UBaseType_t uxPriority;

    /* Task 1 will always run before this task because Task 1 is created with the higher
 priority. Task 1 and Task 2 never block so they will always be in either the Running or
 the Ready state. Query the priority at which this task is running. Passing in NULL means
 "return the calling task's priority". */

    uxPriority = uxTaskPriorityGet( NULL );

    for( ;; )

    {

        /* For this task to reach this point Task 1 must have already run and set the
 priority of this task higher than its own. Print out the name of this task. */

        vPrintString( "Task 2 is running\r\n" );

        /* Set the priority of this task back down to its original value. Passing in NULL
 as the task handle means "change the priority of the calling task". Setting the priority
 below that of Task 1 will cause Task 1 to immediately start running again, preempting this
 task. */

        vPrintString( "About to lower the Task 2 priority\r\n" );

        vTaskPrioritySet( NULL, ( uxPriority - 2 ) );

    }
```

```
}
```

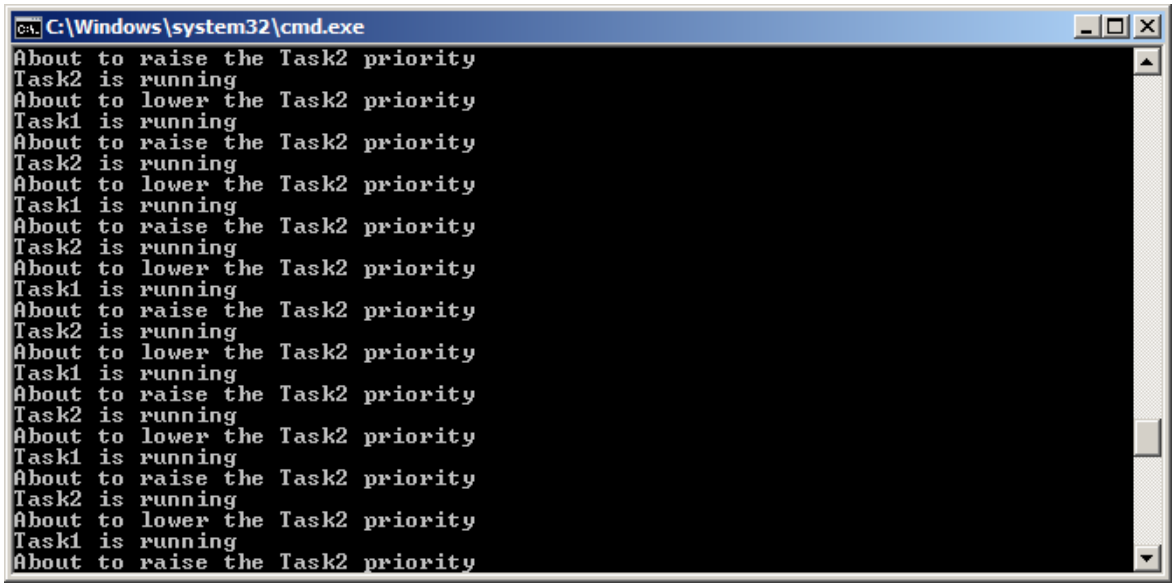
Each task can both query and set its own priority without the use of a valid task handle by simply using NULL. A task handle is required only when a task references a task other than itself, such as when Task 1 changes the priority of Task 2. To allow Task 1 to do this, the Task 2 handle is obtained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

```
/* Declare a variable that is used to hold the handle of Task 2. */

TaskHandle_t xTask2Handle = NULL;

int main( void )

{

    /* Create the first task at priority 2. The task parameter is not used and set to NULL.
 The task handle is also not used so is also set to NULL. */

    xTaskCreate( vTask1, "Task 1", 1000, NULL, 2, NULL );

    /* The task is created at priority 2 ______^. */

    /* Create the second task at priority 1, which is lower than the priority given to
 Task 1. Again, the task parameter is not used so it is set to NULL, but this time the task
 handle is required so the address of xTask2Handle is passed in the last parameter. */

    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, &xTask2Handle );

    /* The task handle is the last parameter _____^^^^^^^^^^^^ */

    /* Start the scheduler so the tasks start executing. */

    vTaskStartScheduler();

    /* If all is well, then main() will never reach here because the scheduler will now be
 running the tasks. If main() does reach here, then it is likely there was insufficient
 heap memory available for the idle task to be created. For information, see Heap Memory
 Management. */

    for( ;; );

}
```

The following figure shows the sequence in which the tasks execute.



The output is shown here.



Deleting a Task

vTaskDelete() API Function

A task can use the vTaskDelete() API function to delete itself or any other task. 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: Only memory allocated to a task by the kernel will be freed automatically when the task is deleted. Any memory or other resource allocated by the implementation of the task must be freed explicitly.

The vTaskDelete() API function prototype is shown here.

```
void vTaskDelete( TaskHandle_t pxTaskToDelete );
```

The following table lists the vTaskDelete() parameter.

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

Deleting Tasks (Example 9)

Here is a very simple example.

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 the first code listing. The source for Task 1 is shown in the second code listing.
2. Task 2 does nothing other than 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 the third code listing.
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 preempts the Idle task. When it enters the Running state, it creates Task 2 again, and so it goes on.

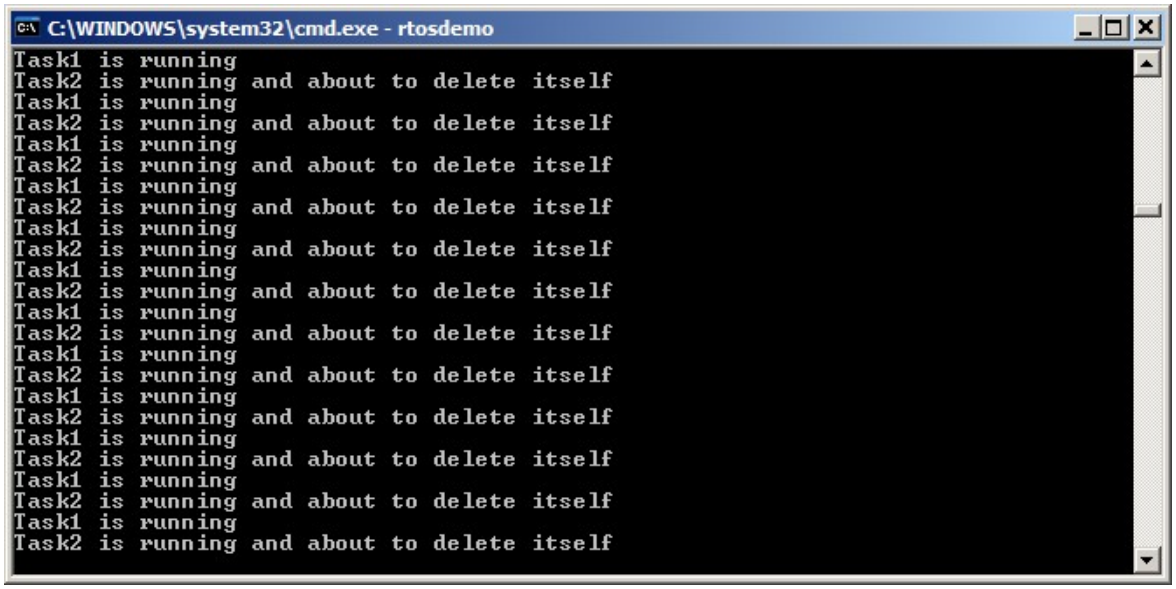
```
int main( void )

{

    /* 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", 1000, NULL, 1, NULL );

    /* The task is created at priority 1 ______^. */

    /* Start the scheduler so the task starts executing. */
```

```
    vTaskStartScheduler();

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

    for( ;; );

}
```

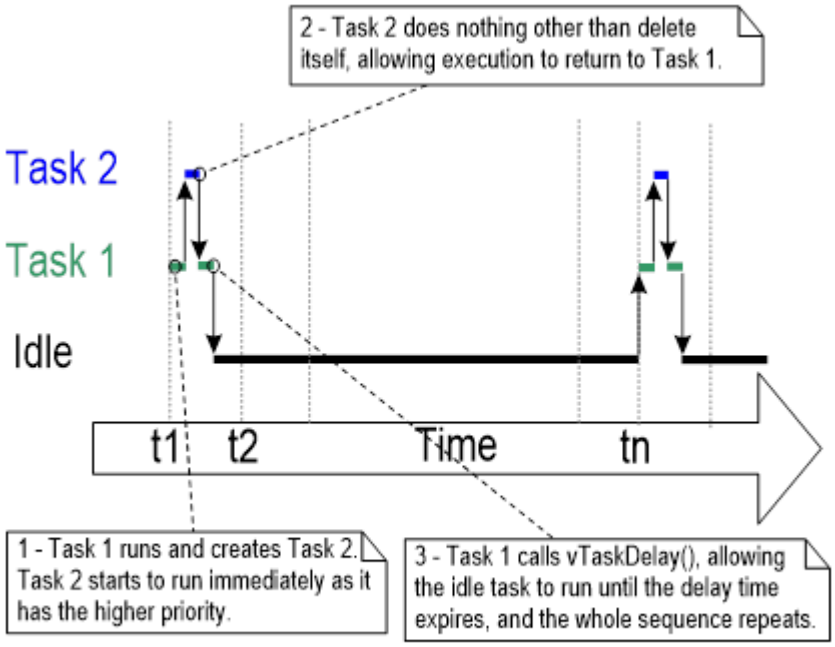
```
TaskHandle_t xTask2Handle = NULL;

void vTask1( void *pvParameters )

{

    const TickType_t xDelay100ms = pdMS_TO_TICKS( 100UL );

    for( ;; )

    {

        /* Print out the name of this task. */

        vPrintString( "Task 1 is running\r\n" );

        /* Create task 2 at a higher priority. Again, the task parameter is not used so it
 is set to NULL, but this time the task handle is required so the address of xTask2Handle
 is passed as the last parameter. */

        xTaskCreate( vTask2, "Task 2", 1000, NULL, 2, &xTask2Handle );

        /* The task handle is the last parameter _____^^^^^^^^^^^^^ */

        /* Task 2 has/had the higher priority, so for Task 1 to reach here, Task 2 must
 have already executed and deleted itself. Delay for 100 milliseconds. */

        vTaskDelay( xDelay100ms );

    }

}
```

```
void vTask2( void *pvParameters )

{

    /* Task 2 does nothing but delete itself. To do this, it could call vTaskDelete() using
 NULL as the parameter, but for demonstration purposes, it calls vTaskDelete(), passing its
 own task handle. */

    vPrintString( "Task 2 is running and about to delete itself\r\n" );

    vTaskDelete( xTask2Handle );

}
```

The output is shown here.



The following figure shows the execution sequence.



Scheduling Algorithms

A Recap of Task States and Events

The task that is actually running (using processing time) is in the Running state. On a single core processor there can only be one task in the Running state at any given time.

Tasks that are not running, but are not in the Blocked state or the Suspended state are in the Ready state. Tasks that are in the Ready state are available to be selected by the scheduler as the task to enter the Running state. The scheduler will always choose the highest priority Ready state task to enter the Running state.

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) and are normally used to implement periodic or timeout behavior. Synchronization events occur when a task or interrupt service routine sends information using a task notification, queue, event group, or one of the many types of semaphore. They are generally used to signal asynchronous activity, such as data arriving at a peripheral.

Configuring the Scheduling Algorithm

The scheduling algorithm is the software routine that decides which Ready state task to transition into the Running state.

All the examples shown so far use the same scheduling algorithm, but you can change it by using the configUSE_PREEMPTION and configUSE_TIME_SLICING configuration constants. Both constants are defined in FreeRTOSConfig.h.

A third configuration constant, configUSE_TICKLESS_IDLE, also affects the scheduling algorithm. Its use can result in the tick interrupt being turned off completely for extended periods. configUSE_TICKLESS_IDLE is an advanced option provided specifically for use in applications that must minimize their power consumption. The descriptions provided in this section assume configUSE_TICKLESS_IDLE is set to 0, which is the default setting if the constant is left undefined.

In all possible configurations, the FreeRTOS scheduler will ensure tasks that share a priority are selected to enter the Running state in turn. This take-it-in-turn policy is often referred to as *round robin scheduling*. A round robin scheduling algorithm does not guarantee time is shared equally between tasks of equal priority, only that Ready state tasks of equal priority will enter the Running state in turn.

Prioritized Preemptive Scheduling with Time Slicing

The configuration shown in the following table sets the FreeRTOS scheduler to use a scheduling algorithm called Fixed Priority Preemptive Scheduling with Time Slicing, which is the scheduling algorithm used by most small RTOS applications and the algorithm used by all the examples presented so far.

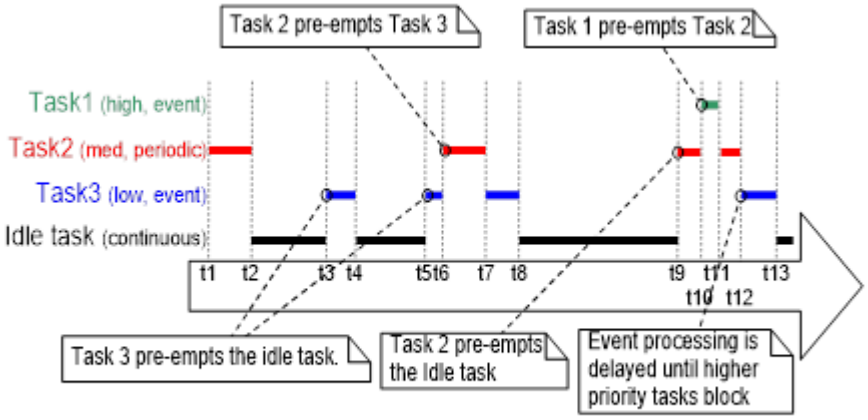
Constant	**Value**
configUSE_PREEMPTION	1
configUSE_TIME_SLICING	1

Here is terminology used in the algorithm:

- **fixed priority tasks**: Scheduling algorithms described as fixed priority do not change the priority assigned to the tasks being scheduled, but also do not prevent the tasks themselves from changing their priority or that of other tasks.
- **preemptive state**: Preemptive scheduling algorithms will immediately preempt the Running state task if a task that has a higher priority enters the Ready state. Being preempted means being involuntarily moved (without explicitly yielding or blocking) out of the Running state and into the Ready state to allow a different task to enter the Running state.
- **time slicing**: Time slicing is used to share processing time between tasks of equal priority, even when the tasks do not explicitly yield or enter the Blocked state. Scheduling algorithms described as using

time slicing will select a new task to enter the Running state at the end of each time slice if there are other Ready state tasks that have the same priority as the Running task. A time slice is equal to the time between two RTOS tick interrupts.

The following figure shows the sequence in which tasks are selected to enter the Running state when all the tasks in an application have a unique priority.



This figure shows the execution pattern, highlighting task prioritization and preemption in a hypothetical application in which each task has been assigned a unique priority.

1. Idle Task

   The idle task is running at the lowest priority, so gets preempted 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 priority. It spends most of its time in the Blocked state waiting for its event of interest, transitioning from the Blocked state to the Ready state each time the event occurs. All FreeRTOS inter-task communication mechanisms (task notifications, queues, semaphores, event groups, and so on) 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 because 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

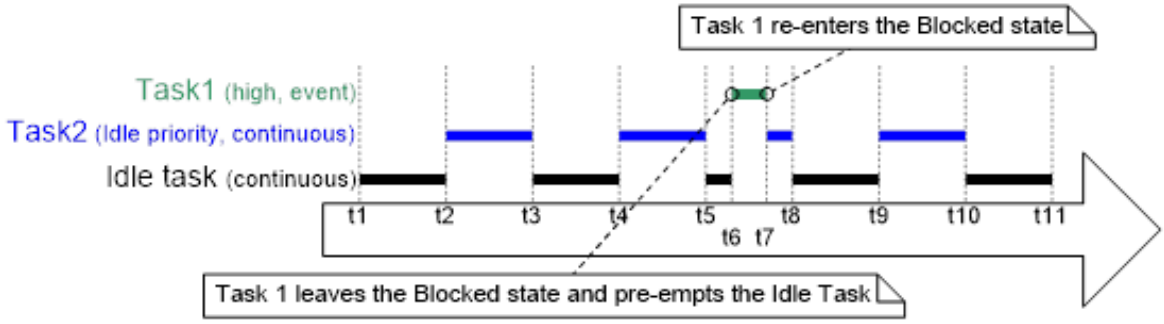
   Task 2 is a periodic task that executes at a priority above the priority of Task 3, but below the priority of Task 1. The task's period interval means Task 2 wants to execute at times t1, t6, and t9. At time t6, Task 3 is in the Running state, but Task 2 has the higher relative priority so it preempts Task 3 and starts executing immediately. Task 2 completes its processing and reenters the Blocked state at time t7, at which point Task 3 can reenter the Running state to complete its processing. Task 3 blocks at time t8.

4. Task 1

   Task 1 is also an event-driven task. It executes with the highest priority of all, so it can preempt any other task in the system. The only Task 1 event shown occurs at time t10, at which time Task 1

preempts Task 2. Task 2 can complete its processing only after Task 1 has reentered the Blocked state at time t11.

The following figure shows the execution pattern, highlighting task prioritization and time slicing in a hypothetical application in which two tasks run at the same priority.



1. Idle Task and Task 2

   The Idle task and Task 2 are both continuous processing tasks, and both have a priority of 0 (the lowest possible priority). The scheduler only allocates processing time to the priority 0 tasks when there are no higher priority tasks that are able to run, and shares the time that is allocated to the priority 0 tasks by time slicing. A new time slice starts on each tick interrupt, which is at times t1, t2, t3, t4, t5, t8, t9, t10, and t11.

   The Idle task and Task 2 enter the Running state in turn, which can result in both tasks being in the Running state for part of the same time slice, as happens between time t5 and time t8.

2. Task 1

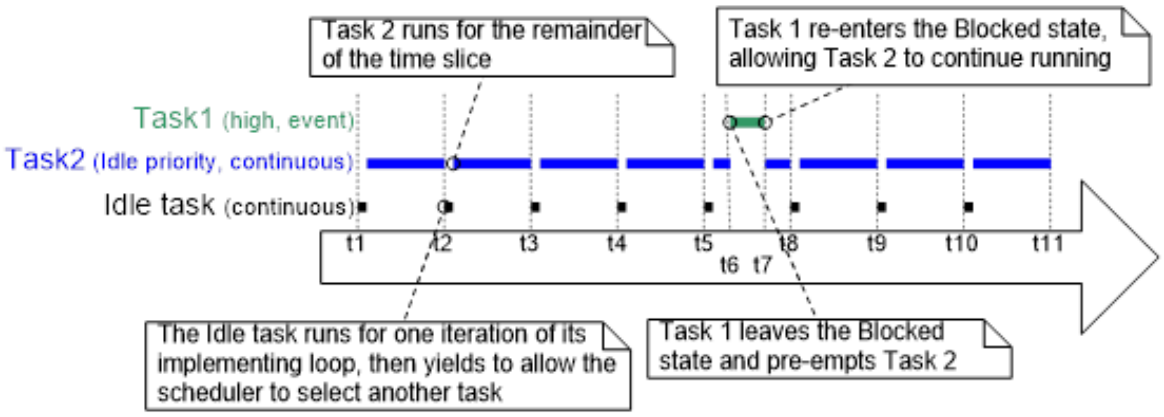
   The priority of Task 1 is higher than the Idle priority. Task 1 is an event-driven task that spends most of its time in the Blocked state waiting for its event of interest, transitioning from the Blocked state to the Ready state each time the event occurs. The event of interest occurs at time t6 when Task 1 becomes the highest priority task that is able to run, and therefore Task 1 preempts the Idle task part way through a time slice. Processing of the event completes at time t7, at which point Task 1 reenters the Blocked state.

The preceding figure shows the Idle task sharing processing time with a task created by the application writer. You might not want to allocate that much processing time to the Idle task if the Idle priority tasks created by the application writer have work to do, but the Idle task does not. You can use the configIDLE_SHOULD_YIELD compile-time configuration constant to change how the Idle task is scheduled:

- If configIDLE_SHOULD_YIELD is set to 0, then the Idle task will remain in the Running state for the entirety of its time slice, unless it is preempted by a higher priority task.
- If configIDLE_SHOULD_YIELD is set to 1, then the Idle task will yield (voluntarily give up whatever remains of its allocated time slice) on each iteration of its loop if there are other Idle priority tasks in the Ready state.

The execution pattern shown in the preceding figure is what would be observed when configIDLE_SHOULD_YIELD is set to 0.

The execution pattern shown in in the following figure is what would be observed in the same scenario when configIDLE_SHOULD_YIELD is set to 1. It shows the sequence in which tasks are selected to enter the Running state when two tasks in an application share a priority.



This figure also shows that when configIDLE_SHOULD_YIELD is set to 0, the task selected to enter the Running state after the Idle task does not execute for an entire time slice, but instead executes for whatever remains of the time slice during which the Idle task yielded.

Prioritized Preemptive Scheduling (Without Time Slicing)

Prioritized preemptive scheduling without time slicing maintains the same task selection and preemption algorithms described in the previous section, but it does not use time slicing to share processing time between tasks of equal priority.

The following table lists FreeRTOSConfig.h settings that configure the FreeRTOS scheduler to use prioritized preemptive scheduling without time slicing.

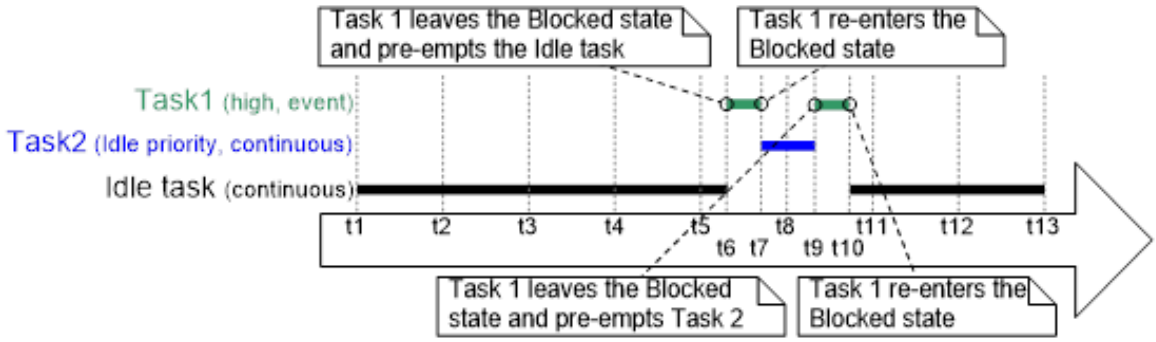
Constant	Value
configUSE_PREEMPTION	1
configUSE_TIME_SLICING	0

If time slicing is used and there is more than one Ready state task at the highest priority that is able to run, then the scheduler will select a new task to enter the Running state during each RTOS tick interrupt (a tick interrupt marking the end of a time slice). If time slicing is not used, then the scheduler will select a new task to enter the Running state only if:

- A higher priority task enters the Ready state.
- The task in the Running state enters the Blocked or Suspended state.

When time slicing is not used, there are fewer task context switches. Therefore, turning off time slicing results in a reduction in the scheduler's processing overhead. However, turning time slicing off can also result in tasks of equal priority receiving very different amounts of processing time. Running the scheduler without time slicing is considered an advanced technique. It should be used by experienced users only.

This figure shows how tasks of equal priority can receive very different amounts of processing time when time slicing is not used.



In the preceding figure, configIDLE_SHOULD_YIELD is set to 0.

1. Tick interrupts

   Tick interrupts occur at times t1, t2, t3, t4, t5, t8, t11, t12, and t13.

2. Task 1

   Task 1 is a high priority, event-driven task that spends most of its time in the Blocked state waiting for its event of interest. Task 1 transitions from the Blocked state to the Ready state (and subsequently, because it is the highest priority Ready state task, into the Running state) each time the event occurs. The preceding figure shows Task 1 processing an event between times t6 and t7, and then again between times t9 and t10.

3. Idle Task and Task 2

   The Idle task and Task 2 are both continuous processing tasks, and both have a priority of 0 (the idle priority). Continuous processing tasks do not enter the Blocked state.

   Time slicing is not being used, so an idle priority task that is in the Running state will remain in the Running state until it is preempted by the higher priority Task 1.

   In the preceding figure, the Idle task starts running at time t1 and remains in the Running state until it is preempted by Task 1 at time t6, which is more than four complete tick periods after it entered the Running state.

   Task 2 starts running at time t7, which is when Task 1 reenters the Blocked state to wait for another event. Task 2 remains in the Running state until it too is preempted by Task 1 at time t9, which is less than one tick period after it entered the Running state.

   At time t10, the Idle task re-enters the Running state, despite having already received more than four times more processing time than Task 2.

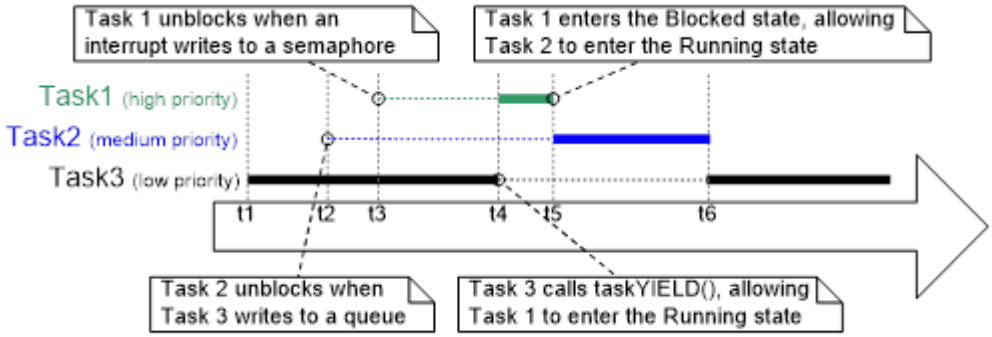
Cooperative Scheduling

FreeRTOS can also use cooperative scheduling. The following table lists FreeRTOSConfig.h settings that configure the FreeRTOS scheduler to use cooperative scheduling.

Constant	**Value**
configUSE_PREEMPTION	0
configUSE_TIME_SLICING	Any value

When the cooperative scheduler is used, a context switch will occur only when the Running state task enters the Blocked state, or the Running state task explicitly yields (manually requests a reschedule) by calling taskYIELD(). Tasks are never preempted, so time slicing cannot be used.

The following figure shows the behavior of the cooperative scheduler. The horizontal dashed lines show when a task is in the Ready state.



1. Task 1

   Task 1 has the highest priority. It starts in the Blocked state, waiting for a semaphore.

   At time t3, an interrupt gives the semaphore, causing Task 1 to leave the Blocked state and enter the Ready state. For information about giving semaphores from interrupts, see Interrupt Management (p. 120).

   At time t3, Task 1 is the highest priority Ready state task, and if the preemptive scheduler had been used, Task 1 would become the Running state task. However, because the cooperative scheduler is being used, Task 1 remains in the Ready state until time t4, which is when the Running state task calls taskYIELD().
2. Task 2

   The priority of Task 2 is between that of Task 1 and Task 3. It starts in the Blocked state, waiting for a message that is sent to it by Task 3 at time t2.

   At time t2, Task 2 is the highest priority Ready state task, and if the preemptive scheduler had been used, Task 2 would become the Running state task. However, because the cooperative scheduler is being used, Task 2 remains in the Ready state until the Running state task either enters the Blocked state or calls taskYIELD().

   The Running state task calls taskYIELD() at time t4, but by then Task 1 is the highest priority Ready state task, so Task 2 does not actually become the Running state task until Task 1 re-enters the Blocked state at time t5.

   At time t6, Task 2 re-enters the Blocked state to wait for the next message, at which point Task 3 is once again the highest priority Ready state task.

In a multitasking application, the application writer must take care that a resource is not accessed by more than one task simultaneously because simultaneous access can corrupt the resource. Consider the following scenario in which the resource being accessed is a UART (serial port). Two tasks are writing strings to the UART. Task 1 is writing "abcdefghijklmnop" and Task 2 is writing "123456789":

1. Task 1 is in the Running state and starts to write its string. It writes "abcdefg" to the UART, but leaves the Running state before writing any further characters.
2. Task 2 enters the Running state and writes "123456789" to the UART, before leaving the Running state.

3. Task 1 re-enters the Running state and writes the remaining characters of its string to the UART.

In that scenario, what is actually written to the UART is "abcdefg123456789hijklmnop." The string written by Task 1 has not been written to the UART in an unbroken sequence as intended. Instead, it has been corrupted because the string written to the UART by Task 2 appears within it.

You can avoid problems caused by simultaneous access by using the cooperative scheduler. Methods for safely sharing resources between tasks are covered later in this guide. Resources provided by FreeRTOS, such as queues and semaphores, are always safe to share between tasks.

- When the preemptive scheduler is used, the Running state task can be preempted at any time, including when a resource it is sharing with another task is in an inconsistent state. As the UART example shows, leaving a resource in an inconsistent state can result in data corruption.
- When the cooperative scheduler is used, the application writer controls when a switch to another task can occur. The application writer can therefore ensure a switch to another task does not occur while a resource is in an inconsistent state.
- In the UART example, the application writer can ensure Task 1 does not leave the Running state until its entire string has been written to the UART, and in doing so, removes the possibility the string is corrupted by the activation of another task.

Systems will be less responsive when the cooperative scheduler is used.

- When the preemptive scheduler is used, the scheduler will start running a task immediately after the task becomes the highest priority Ready state task. This is often essential in real-time systems that must respond to high priority events within a defined time period.
- When the cooperative scheduler is used, a switch to a task that has become the highest priority Ready state task is not performed until the Running state task enters the Blocked state or calls taskYIELD().

Queue Management

Queues provide a task-to-task, task-to-interrupt, and interrupt-to-task communication mechanism. This section covers task-to-task communication. For information about task-to-interrupt and interrupt-to-task communication, see Interrupt Management (p. 120).

This section covers:

- 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.
- How to block on multiple queues.
- How to overwrite data in a queue.
- How to clear a queue.
- The effect of task priorities when writing to and reading from a queue.

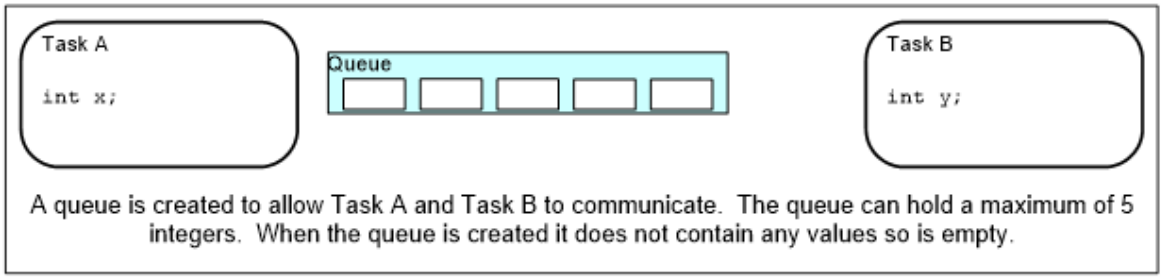
Characteristics of a Queue

Data Storage

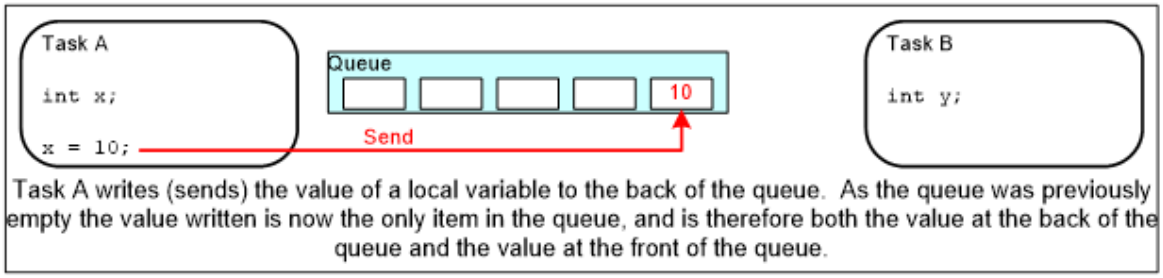
A queue can hold a finite number of fixed-size data items. The maximum number of items a queue can hold is called its *length*. Both the length and the size of each data item are set when the queue is created.

Queues are normally 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.

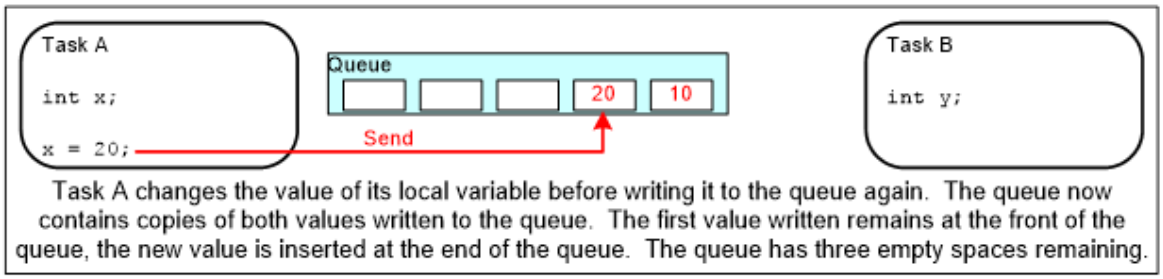
The following figure shows data being written to and read from a queue that is being used as a FIFO. It is also possible to write to the front of a queue and to overwrite data that is already at the front of a queue.



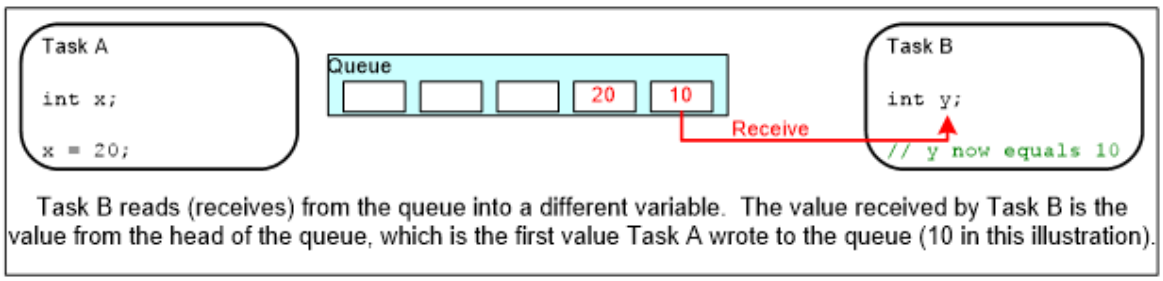
A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.



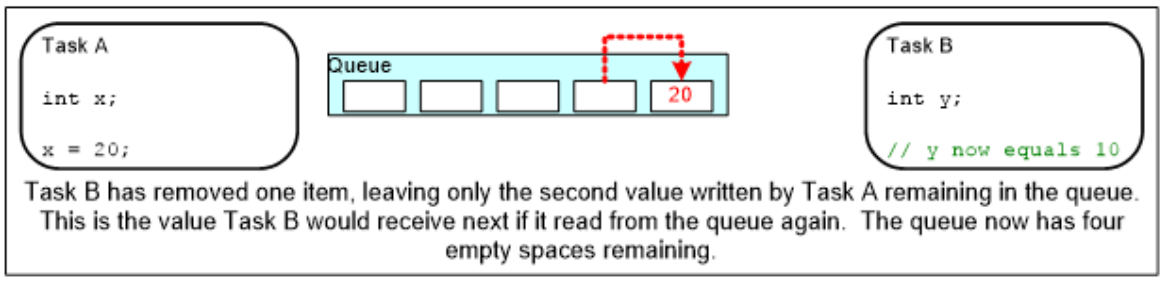
Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.



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.

There are two ways to implement queue behavior:

1. Queue by copy

   Data sent to the queue is copied byte for byte into the queue.

2. Queue by reference

   The queue holds pointers to the data sent to the queue, not the data itself.

FreeRTOS uses queue by copy. This method is considered more powerful and simpler to use than queueing by reference because:

- A stack variable can be sent directly to a queue, even though the variable will not exist after the function in which it is declared has exited.
- Data can be sent to a queue without first allocating a buffer to hold the data, and then copying the data into the allocated buffer.
- The sending task can immediately reuse the variable or buffer that was sent to the queue.
- The sending task and the receiving task are completely decoupled. Application designers do not need to concern themselves with which task owns the data or which task is responsible for releasing the data.
- Queuing by copy does not prevent the queue from also being used to queue by reference. For example, when the size of the data being queued makes it impractical to copy the data into the queue, then a pointer to the data can be copied into the queue instead.
- RTOS takes complete responsibility for allocating the memory used to store data.
- In a memory-protected system, the RAM that a task can access is restricted. In that case, queueing by reference can be used only if the sending and receiving tasks can access the RAM in which the data is stored. Queuing by copy does not impose that restriction. The kernel always runs with full privileges, allowing a queue to be used to pass data across memory protection boundaries.

Access by Multiple Tasks

Queues are objects that can be accessed by any task or Interrupt Service Register (ISR) that knows of their existence. Any number of tasks can write to the same queue, and any number of tasks can read from the same queue. It is very common for a queue to have multiple writers, but much less common for a queue to have multiple readers.

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 will be kept in the Blocked state to wait for data to be available from the queue, should the queue already be empty. A task 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.

Blocking on Multiple Queues

Queues can be grouped into sets, allowing a task to enter the Blocked state to wait for data to become available on any of the queues in the set. For more information about queue sets, see Receiving from Multiple Queues (p. 86).

Using a Queue

xQueueCreate() API Function

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

Queues are referenced by handles, which are variables of type QueueHandle_t. The xQueueCreate() API function creates a queue and returns a QueueHandle_t that references the queue it created.

FreeRTOS V9.0.0 also includes the xQueueCreateStatic() function, which allocates the memory required to create a queue statically at compile time. 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.

The xQueueCreate() API function prototype is shown here.

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength, UBaseType_t uxItemSize );
```

The following table lists the 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. If a non-NULL value is returned, the queue has been created successfully. The returned value should be stored as the handle to the created queue.

After a queue has been created, the xQueueReset() API function can be used to return the queue to its original empty state.

xQueueSendToBack() and xQueueSendToFront() API Functions

xQueueSendToBack() is used to send data to the back (tail) of a queue. xQueueSendToFront() is used to send data to the front (head) of a queue.

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

Note: Do not call xQueueSendToFront() or xQueueSendToBack() from an interrupt service routine. Use the interrupt-safe versions, xQueueSendToFrontFromISR() and xQueueSendToBackFromISR(), instead.

The xQueueSendToFront() API function prototype is shown here.

```
BaseType_t xQueueSendToFront( QueueHandle_t xQueue,

const void * pvItemToQueue,

TickType_t xTicksToWait );
```

The xQueueSendToBack() API function prototype is shown here.

```
BaseType_t xQueueSendToBack( QueueHandle_t xQueue,

const void * pvItemToQueue,

TickType_t xTicksToWait );
```

The following table lists the 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.
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 macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.
Returned value	There are two possible return values: 1. pdPASS Returned only if data was successfully sent to the queue. If a block time was specified (xTicksToWait was not zero), then it is possible the calling task was placed into 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 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 space in the queue, but the specified block time expired before that happened.

xQueueReceive() API Function

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

Note: Do not call xQueueReceive() from an interrupt service routine. Use the interrupt-safe xQueueReceiveFromISR() API function instead.

The xQueueReceive() API function prototype is shown here.

```
BaseType_t xQueueReceive( QueueHandle_t xQueue,

void * const pvBuffer,

TickType_t xTicksToWait );
```

The following table lists the xQueueReceive() 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.
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 xQueueReceive() will return immediately if the queue is already empty. The block time is specified in tick periods, so the absolute time it represents depends on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.
Returned value	There are two possible return values: 1. pdPASS Returned only if data was successfully read from the queue. If a block time was specified (xTicksToWait was not zero), then it is possible the calling task was placed into 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 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 that happened.

uxQueueMessagesWaiting() API Function

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

Note: Do not call uxQueueMessagesWaiting() from an interrupt service routine. Use the interrupt-safe uxQueueMessagesWaitingFromISR() instead.

The uxQueueMessagesWaiting() API function prototype is shown here.

```
UBaseType_t uxQueueMessagesWaiting( QueueHandle_t xQueue );
```

The following table lists the uxQueueMessagesWaiting() function parameter 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.

Blocking When Receiving from a Queue (Example 10)

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 int32_t. The tasks that send to the queue do not specify a block time, but the task that receives from the queue does.

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

The following code 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 )

{

    int32_t lValueToSend;

    BaseType_t xStatus;

    /* Two instances of this task are created so the value that is sent to the queue
 is passed in through the task parameter. This way, each instance can use a different
 value. The queue was created to hold values of type int32_t, so cast the parameter to the
 required type. */
```

```
    lValueToSend = ( int32_t ) 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 because the queue should never contain more than one item! */

            vPrintString( "Could not send to the queue.\r\n" );

        }

    }

}
```

The following code shows the implementation of the task that receives data from the queue. The receiving task specifies a block time of 100 milliseconds, so it 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 because there are two tasks continuously writing to the queue.

```
static void vReceiverTask( void *pvParameters )

{

    /* Declare the variable that will hold the values received from the queue. */

    int32_t lReceivedValue;

    BaseType_t xStatus;

    const TickType_t xTicksToWait = pdMS_TO_TICKS( 100 );

    /* This task is also defined within an infinite loop. */

    for( ;; )

    {

        /* This call should always find the queue empty because this task will immediately
 remove any data that is written to the queue. */

        if( uxQueueMessagesWaiting( xQueue ) != 0 )

        {
```

```
            vPrintString( "Queue should have been empty!\r\n" );

        }

        /* Receive data from the queue. The first parameter is the queue from which data is
 to be received. The queue is created before the scheduler is started, and therefore before
 this task runs for the first time.  The second parameter is the buffer into which the
 received data will be placed. In this case the buffer is simply the address of a variable
 that has the required size to hold the received data. The last parameter is the block
 time, the maximum amount of time that the task will remain in the Blocked state to wait
 for data to be available should the queue already be empty. */

        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 because the sending tasks are free running and will be continuously writing to
 the queue. */

            vPrintString( "Could not receive from the queue.\r\n" );

        }

    }

}
```

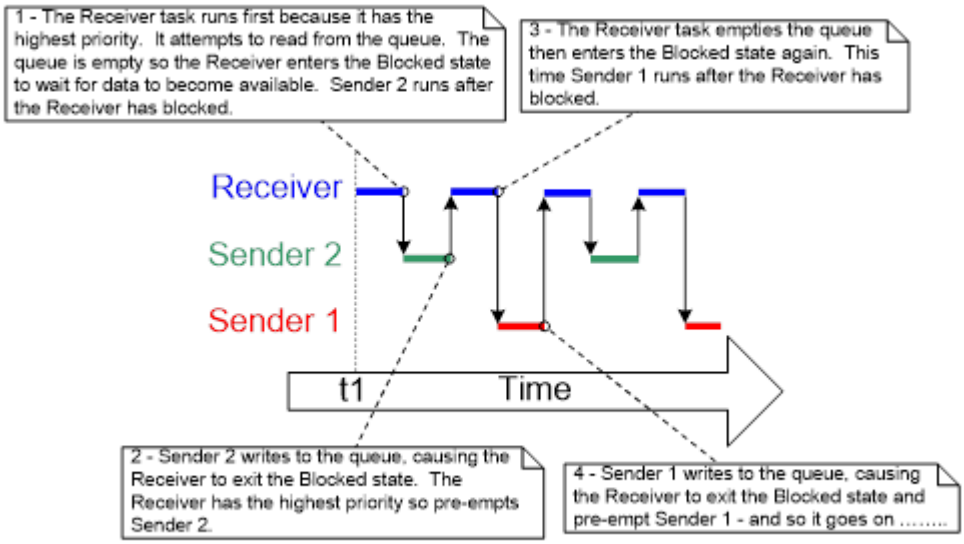
The following code contains the definition of the main() function. This simply creates the queue and the three tasks before starting the scheduler. The queue is created to hold a maximum of five int32_t values, even though the priorities of the tasks are set in such a way that the queue will never contain more than one item at a time.

```
/* Declare a variable of type QueueHandle_t. This is used to store the handle to the queue
 that is accessed by all three tasks. */

QueueHandle_t 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 int32_t. */

    xQueue = xQueueCreate( 5, sizeof( int32_t ) );

    if( xQueue != NULL )

    {
```

```
        /* Create two instances of the task that will send to the queue. The task parameter
    is used to pass the value that the task will write to the queue, so one task will
    continuously write 100 to the queue while the other task will continuously write 200 to
    the queue. Both tasks are created at priority 1. */

        xTaskCreate( vSenderTask, "Sender1", 1000, ( void * ) 100, 1, NULL );

        xTaskCreate( vSenderTask, "Sender2", 1000, ( void * ) 200, 1, NULL );

        /* Create the task that will read from the queue. The task is created with priority
    2, so above the priority of the sender tasks. */

        xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 2, NULL );

        /* Start the scheduler so the created tasks start executing. */

        vTaskStartScheduler();

    }

    else

    {

        /* The queue could not be created. */

    }

    /* If all is well then main() will never reach here as the scheduler will now be
    running the tasks. If main() does reach here then it is likely that there was insufficient
    FreeRTOS heap memory available for the idle task to be created. For more information, see
    Heap Memory Management. */

    for( ;; );

}
```

Both tasks that send to the queue have an identical priority. This causes the two sending tasks to send data to the queue in turn.

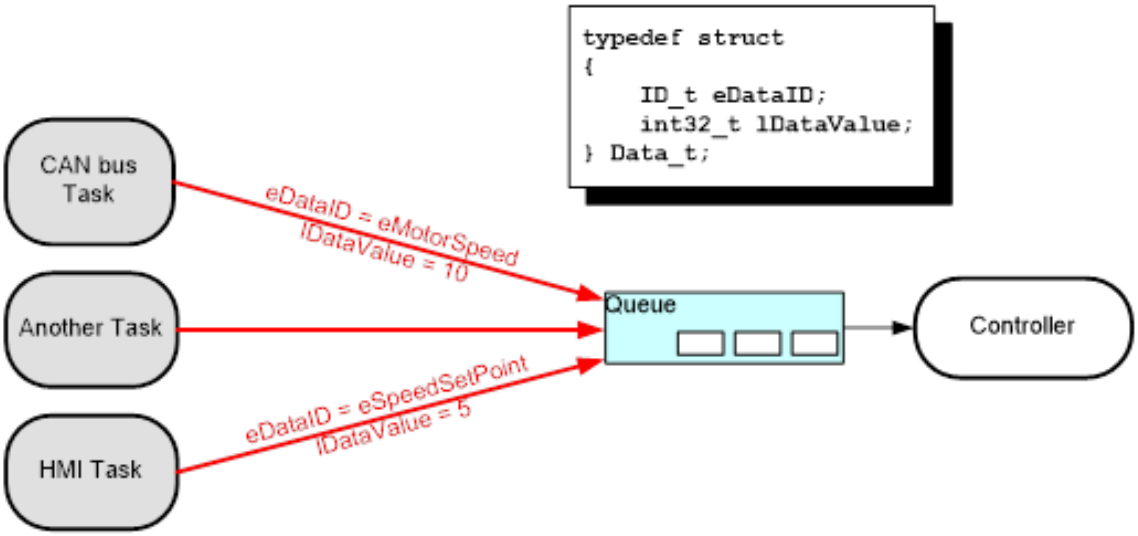
The following figure shows the sequence of execution.



Receiving Data from Multiple Sources

It is common in FreeRTOS designs for a task to receive data from more than one source. The receiving task must know where the data came from to determine how the data should be processed. An easy design solution is to use a single queue to transfer structures with both the value of the data and the source of the data contained in the structure's fields.

The following figure shows an example scenario where structures are sent on a queue.



- A queue is created that holds structures of type Data_t. The structure members allow a data value and an enumerated type to be sent to the queue in one message. The enumerated type is used to indicate what the data means.
- 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 a Data_t structure. The eDataID member of the transferred structure is used to let the Controller task know what the data is (in this case, a motor speed value). The lDataValue member of the transferred structure is used to let the Controller task know the 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 a Data_t structure. The eDataID member of the transferred structure is used to let the Controller task know what the data is (in this case, a new set point value). The lDataValue member of the transferred structure is used to let the Controller task know the set point value.

Blocking When Sending to a Queue and Sending Structures on a Queue (Example 11)

This example is similar to the previous example, 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 integers.

The following code shows the definition of the structure that is to be passed on a queue and the declaration of two variables.

```
/* Define an enumerated type used to identify the source of the data.*/

typedef enum

{

    eSender1,

    eSender2

} DataSource_t;

/* Define the structure type that will be passed on the queue. */

typedef struct

{

    uint8_t ucValue;

    DataSource_t eDataSource;

} Data_t;

/* Declare two variables of type Data_t that will be passed on the queue. */

static const Data_t xStructsToSend[ 2 ] =

{

    { 100, eSender1 }, /* Used by Sender1. */

    { 200, eSender2 } /* Used by Sender2. */

};
```

In the previous example, the receiving task has the highest priority, so the queue never contains more than one item. This results from the receiving task preempting the sending tasks as soon as data is placed into the queue. In the following example, the sending tasks have the higher priority, so the queue will normally be full. This is because as soon as the receiving task removes an item from the queue, it is preempted by one of the sending tasks, which then immediately refills the queue. The sending task then re-enters the Blocked state to wait for space to become available on the queue again.

The following code 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 because the receiving task is continuously making space by removing items from the queue.

```
static void vSenderTask( void *pvParameters )

{

    BaseType_t xStatus;

    const TickType_t xTicksToWait = pdMS_TO_TICKS( 100 );

    /* 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 100 ms. This
 must be an error because the receiving task should make space in the queue as soon as both
 sending tasks are in the Blocked state. */

            vPrintString( "Could not send to the queue.\r\n" );

        }

    }

}
```

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 when it does not specify a block time.

The following code shows the implementation of the receiving task.

```
static void vReceiverTask( void *pvParameters )

{

    /* Declare the structure that will hold the values received from the queue. */

    Data_t xReceivedStructure;

    BaseType_t 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, which is 3 in this case. */

        if( uxQueueMessagesWaiting( xQueue ) != 3 )

        {

            vPrintString( "Queue should have been full!\r\n" );
```

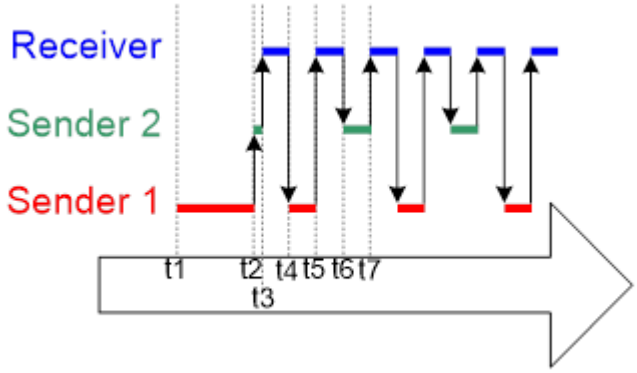
```
        }

        /* Receive from the queue.  The second parameter is the buffer into which the
 received data will be placed. In this case, the buffer is simply the address of a variable
 that has the required size to hold the received structure. The last parameter is the block
 time, the maximum amount of time that the task will remain in the Blocked state to wait
 for data to be available if the queue is already empty. In this case, a block time is not
 required 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.eDataSource == eSender1 )

            {

                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 because this task
 should only run when the queue is full. */

            vPrintString( "Could not receive from the queue.\r\n" );

        }

    }

}
```

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

```
int main( void )

{

    /* The queue is created to hold a maximum of 3 structures of type Data_t. */

    xQueue = xQueueCreate( 3, sizeof( Data_t ) );

    if( xQueue != NULL )
```

```
    {

        /* Create two instances of the task that will write to the queue. The parameter
    is used to pass the structure that the task will write to the queue, so one task will
    continuously send xStructsToSend[ 0 ] to the queue while the other task will continuously
    send xStructsToSend[ 1 ]. Both tasks are created at priority 2, which is above the
    priority of the receiver. */

        xTaskCreate( vSenderTask, "Sender1", 1000, &( xStructsToSend[ 0 ] ), 2, NULL );

        xTaskCreate( vSenderTask, "Sender2", 1000, &( xStructsToSend[ 1 ] ), 2, NULL );

        /* Create the task that will read from the queue. The task is created with priority
    1, so below the priority of the sender tasks. */

        xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 1, NULL );

        /* Start the scheduler so the created tasks start executing. */

        vTaskStartScheduler();

    }

    else

    {

        /* The queue could not be created. */

    }

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

    for( ;; );

}
```

The following figure shows the sequence of execution that results from having the priority of the sending tasks above the priority of the receiving task.



This table describes why the first four message originate from the same task.

Time	Description

t1	Task Sender 1 executes and sends three data items to the queue.
t2	The queue is full so Sender 1 enters the Blocked state to wait for its next send to complete. Task Sender 2 is now the highest priority task that is able to run, so enters the Running state.
t3	Task Sender 2 finds the queue is already full, so enters the Blocked state to wait for its first send to complete. Task Receiver is now the highest priority task that is able to run, so enters the Running state.
t4	Two tasks that have a priority higher than the receiving task's priority are waiting for space to become available on the queue, resulting in task Receiver being preempted as soon as it has removed one item from the queue. Tasks Sender 1 and Sender 2 have the same priority, so the scheduler selects the task that has been waiting the longest as the task that will enter the Running state (in this case, that is task Sender 1).
t5	Task Sender 1 sends another data item to the queue. There was only one space in the queue, so task Sender 1 enters the Blocked state to wait for its next send to complete. Task Receiver is again the highest priority task that is able to run so enters the Running state. Task Sender 1 has now sent four items to the queue, and task Sender 2 is still waiting to send its first item to the queue.
t6	Two tasks that have a priority higher than the receiving task's priority are waiting for space to become available on the queue, so task Receiver is preempted as soon as it has removed one item from the queue. This time Sender 2 has been waiting longer than Sender 1, so Sender 2 enters the Running state.
t7	Task Sender 2 sends a data item to the queue. There was only one space in the queue, so Sender 2 enters the Blocked state to wait for its next send to complete. Both tasks Sender 1 and Sender 2 are waiting for space to become available on the queue, so task Receiver is the only task that can enter the Running state.

Working with Large or Variable-Sized Data

Queuing Pointers

If the size of the data being stored in the queue is large, then it is better to use the queue to transfer pointers to the data rather than copy the data 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 you are queuing pointers, make sure that:

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

When using a pointer to share memory between tasks, you must make sure 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.

- The RAM being pointed to remains valid.

If the memory being pointed to was allocated dynamically or obtained from a pool of preallocated buffers, then exactly one task should be responsible for freeing the memory. No tasks should attempt to access the memory after it has been freed.

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

The following code examples demonstrate how to use a queue to send a pointer to a buffer from one task to another.

The following code creates a queue that can hold up to five pointers.

```
/* Declare a variable of type QueueHandle_t to hold the handle of the queue being created.
 */

QueueHandle_t xPointerQueue;

/* Create a queue that can hold a maximum of 5 pointers (in this case, character pointers).
 */

xPointerQueue = xQueueCreate( 5, sizeof( char * ) );
```

The following code allocates a buffer, writes a string to the buffer, and then sends a pointer to the buffer to the queue.

```
/* A task that obtains a buffer, writes a string to the buffer, and then sends the address
 of the buffer to the queue created in the previous listing. */

void vStringSendingTask( void *pvParameters )

{

    char *pcStringToSend;

    const size_t xMaxStringLength = 50;
```

```
    BaseType_t xStringNumber = 0;

    for( ;; )

    {

        /* Obtain a buffer that is at least xMaxStringLength characters big. The
 implementation of prvGetBuffer() is not shown. It might obtain the buffer from a pool of
 preallocated buffers or just allocate the buffer dynamically. */

        pcStringToSend = ( char * ) prvGetBuffer( xMaxStringLength );

        /* Write a string into the buffer. */

        snprintf( pcStringToSend, xMaxStringLength, "String number %d\r\n",
 xStringNumber );
        /* Increment the counter so the string is different on each iteration of this task.
 */

        xStringNumber++;

        /* Send the address of the buffer to the queue that was created in the previous
 listing. The address of the buffer is stored in the pcStringToSend variable.*/

        xQueueSend( xPointerQueue, /* The handle of the queue. */ &pcStringToSend, /* The
 address of the pointer that points to the buffer. */ portMAX_DELAY );

    }

}
```

The following code receives a pointer to a buffer from the queue, and then prints the string contained in the buffer.

```
/* A task that receives the address of a buffer from the queue created in the first listing
 and written to in the second listing. The buffer contains a string, which is printed out.
 */

void vStringReceivingTask( void *pvParameters )

{

    char *pcReceivedString;

    for( ;; )

    {

        /* Receive the address of a buffer. */

        xQueueReceive( xPointerQueue, /* The handle of the queue. */ &pcReceivedString, /*
 Store the buffer's address in pcReceivedString. */ portMAX_DELAY );

        /* The buffer holds a string. Print it out. */

        vPrintString( pcReceivedString );

        /* The buffer is not required anymore. Release it so it can be freed or reused. */

        prvReleaseBuffer( pcReceivedString );

    }

}
```

Using a Queue to Send Different Types and Lengths of Data

Sending structures to a queue and sending pointers to a queue are two powerful design patterns. When you combine these techniques, a task can use a single queue to receive any data type from any data source. The implementation of the FreeRTOS+TCP TCP/IP stack provides a practical example for doing this.

The TCP/IP stack, which runs in its own task, must process events from many different sources. Different event types are associated with different types and lengths of data. All events that occur outside of the TCP/IP task are described by a structure of type IPStackEvent_t, and sent to the TCP/IP task on a queue. The pvData member of the IPStackEvent_t structure is a pointer that can be used to hold a value directly or point to a buffer.

The IPStackEvent_t structure used to send events to the TCP/IP stack task in FreeRTOS+TCP is shown here.

```
/* A subset of the enumerated types used in the TCP/IP stack to identify events. */

typedef enum

{

    eNetworkDownEvent = 0, /* The network interface has been lost or needs (re)connecting.
 */

    eNetworkRxEvent, /* A packet has been received from the network. */

    eTCPAcceptEvent, /* FreeRTOS_accept() called to accept or wait for a new client. */

    /* Other event types appear here but are not shown in this listing. */

} eIPEvent_t;

/* The structure that describes events and is sent on a queue to the TCP/IP task. */

typedef struct IP_TASK_COMMANDS

{

    /* An enumerated type that identifies the event. See the eIPEvent_t definition. */

    eIPEvent_t eEventType;

    /* A generic pointer that can hold a value or point to a buffer. */

    void *pvData;

} IPStackEvent_t;
```

Example TCP/IP events and their associated data include:

- eNetworkRxEvent: A packet of data has been received from the network.

  Data received from the network is sent to the TCP/IP task using a structure of type IPStackEvent_t. The structure's eEventType member is set to eNetworkRxEvent. The structure's pvData member is used to point to the buffer that contains the received data.

  This pseudo code shows how an IPStackEvent_t structure is used to send data received from the network to the TCP/IP task.

```
    void vSendRxDataToTheTCPTask( NetworkBufferDescriptor_t    *pxRxedData )

{

    IPStackEvent_t xEventStruct;

    /* Complete the IPStackEvent_t structure. The received data is stored in pxRxedData.
 */

    xEventStruct.eEventType = eNetworkRxEvent;

    xEventStruct.pvData = ( void * ) pxRxedData;

    /* Send the IPStackEvent_t structure to the TCP/IP task. */

    xSendEventStructToIPTask( &xEventStruct );

}
```

- eTCPAcceptEvent: A socket is to accept or wait for a connection from a client.

  Accept events are sent from the task that called FreeRTOS_accept() to the TCP/IP task using a structure of type IPStackEvent_t. The structure's eEventType member is set to eTCPAcceptEvent. The structure's pvData member is set to the handle of the socket that is accepting a connection.

  This pseudo code shows how an IPStackEvent_t structure is used to send the handle of a socket that is accepting a connection to the TCP/IP task.

```
    void vSendAcceptRequestToTheTCPTask( Socket_t xSocket )

{

    IPStackEvent_t xEventStruct;

    /* Complete the IPStackEvent_t structure. */

    xEventStruct.eEventType = eTCPAcceptEvent;

    xEventStruct.pvData = ( void * ) xSocket;

    /* Send the IPStackEvent_t structure to the TCP/IP task. */

    xSendEventStructToIPTask( &xEventStruct );

}
```

- eNetworkDownEvent: The network needs connecting or reconnecting.

  Network down events are sent from the network interface to the TCP/IP task using a structure of type IPStackEvent_t. The structure's eEventType member is set to eNetworkDownEvent. Network down events are not associated with any data, so the structure's pvData member is not used.

  This pseudo code shows how an IPStackEvent_t structure is used to send a network down event to the TCP/IP task.

```
void vSendNetworkDownEventToTheTCPTask( Socket_t xSocket )

{

    IPStackEvent_t xEventStruct;
```

```
    /* Complete the IPStackEvent_t structure. */

    xEventStruct.eEventType = eNetworkDownEvent;

    xEventStruct.pvData = NULL; /* Not used, but set to NULL for completeness. */

    /* Send the IPStackEvent_t structure to the TCP/IP task. */

    xSendEventStructToIPTask( &xEventStruct );

}
```

The code that receives and processes these events within the TCP/IP task is shown here. The eEventType member of the IPStackEvent_t structures received from the queue is used to determine how the pvData member is to be interpreted. This pseudo code shows how an IPStackEvent_t structure is used to send a network down to the TCP/IP task.

```
IPStackEvent_t xReceivedEvent;

/* Block on the network event queue until either an event is received, or xNextIPSleep
 ticks pass without an event being received. eEventType is set to eNoEvent in case the
 call to xQueueReceive() returns because it timed out, rather than because an event was
 received. */

xReceivedEvent.eEventType = eNoEvent;

xQueueReceive( xNetworkEventQueue, &xReceivedEvent, xNextIPSleep );

/* Which event was received, if any? */

switch( xReceivedEvent.eEventType )

{

    case eNetworkDownEvent :

        /* Attempt to (re)establish a connection. This event is not associated with any
 data. */

        prvProcessNetworkDownEvent();

        break;

    case eNetworkRxEvent:

        /* The network interface has received a new packet. A pointer to the received data
 is stored in the pvData member of the received IPStackEvent_t structure. Process the
 received data. */

        prvHandleEthernetPacket( ( NetworkBufferDescriptor_t * )
( xReceivedEvent.pvData ) );

        break;

    case eTCPAcceptEvent:

        /* The FreeRTOS_accept() API function was called. The handle of the socket that
 is accepting a connection is stored in the pvData member of the received IPStackEvent_t
 structure. */

        pxSocket = ( FreeRTOS_Socket_t * ) ( xReceivedEvent.pvData );

        xTCPCheckNewClient( pxSocket );
```

```
        break;

        /* Other event types are processed in the same way, but are not shown here. */
}
```

Receiving from Multiple Queues

Queue Sets

Application designs often require a single task to receive data of different sizes, data of different meaning, and data from different sources. The previous section described how you can do this in a neat and efficient way by using a single queue that receives structures. However, sometimes you might be working with constraints that limit your design choices, requiring the use of a separate queue for some data sources. For example, third-party code that is being integrated into a design might assume the presence of a dedicated queue. In such cases, you can use a *queue set*.

Queue sets allow a task to receive data from more than one queue without the task polling each queue in turn to determine which, if any, contains data.

A design that uses a queue set to receive data from multiple sources is less neat and efficient than a design that achieves the same functionality using a single queue that receives structures. For that reason, we recommend that you use queue sets only used if your design constraints make their use absolutely necessary.

The following sections describe how to:

1. Create a queue set.
2. Add queues to the set.

   You can also add semaphores to a queue set. Semaphores are described in Interrupt Management (p. 120).
3. Read from the queue set to determine which queues within the set contain data.

   When a queue that is a member of a set receives data, the handle of the receiving queue is sent to the queue set, and returned when a task calls a function that reads from the queue set. Therefore, if a queue handle is returned from a queue set, then the queue referenced by the handle is known to contain data, and the task can then read from the queue directly.

   Note: If a queue is a member of a queue set, then do not read data from the queue unless the queue's handle has first been read from the queue set.

To enable queue set functionality, in FreeRTOSConfig.h, set the configUSE_QUEUE_SETS compile-time configuration constant to 1.

xQueueCreateSet() API Function

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

Queues sets are referenced by handles, which are variables of type QueueSetHandle_t. The xQueueCreateSet() API function creates a queue set and returns a QueueSetHandle_t that references the queue set it created.

The xQueueCreateSet() API function prototype is shown here.

```
QueueSetHandle_t xQueueCreateSet( const UBaseType_t uxEventQueueLength);
```

The following table lists the xQueueCreateSet() parameters and return value.

Parameter Name	Description
uxEventQueueLength	When a queue that is a member of a queue set receives data, the handle of the receiving queue is sent to the queue set. uxEventQueueLength defines the maximum number of queue handles the queue set being created can hold at any one time. Queue handles are only sent to a queue set when a queue within the set receives data. A queue cannot receive data if it is full, so no queue handles can be sent to the queue set if all the queues in the set are full. Therefore, the maximum number of items the queue set will ever have to hold at one time is the sum of the lengths of every queue in the set. As an example, if there are three empty queues in the set, and each queue has a length of five, then in total the queues in the set can receive fifteen items (three queues multiplied by five items each) before all the queues in the set are full. In that example, uxEventQueueLength must be set to fifteen to guarantee the queue set can receive every item sent to it. Semaphores can also be added to a queue set. Binary and counting semaphores are covered later in this guide. For the purposes of calculating the uxEventQueueLength, the length of a binary semaphore is one, and the length of a counting semaphore is given by the semaphore's maximum count value. As another example, if a queue set will contain a queue that has a length of three, and a binary semaphore (which has a length of one), uxEventQueueLength must be set to four (three plus one).
Return Value	If NULL is returned, then the queue set cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the queue set data structures and storage area. If a non-NULL value is returned, the queue set has been created successfully. The returned value should be stored as the handle to the created queue set.

xQueueAddToSet() API Function

xQueueAddToSet() adds a queue or semaphore to a queue set. For information about semaphores, see Interrupt Management (p. 120).

The xQueueAddToSet() API function prototype is shown here.

```
BaseType_t xQueueAddToSet( QueueSetMemberHandle_t xQueueOrSemaphore, QueueSetHandle_t
 xQueueSet );
```

The following table lists the xQueueAddToSet() parameters and return value.

xQueueSelectFromSet() API Function

xQueueSelectFromSet() reads a queue handle from the queue set.

When a queue or semaphore that is a member of a set receives data, the handle of the receiving queue or semaphore is sent to the queue set, and returned when a task calls xQueueSelectFromSet(). If a handle is returned from a call to xQueueSelectFromSet(), then the queue or semaphore referenced by the handle is known to contain data and the calling task must then read from the queue or semaphore directly.

Note: Do not read data from a queue or semaphore that is a member of a set unless the handle of the queue or semaphore has first been returned from a call to xQueueSelectFromSet(). Only read one item from a queue or semaphore each time the queue handle or semaphore handle is returned from a call to xQueueSelectFromSet().

The xQueueSelectFromSet() API function prototype is shown here.

```
QueueSetMemberHandle_t xQueueSelectFromSet( QueueSetHandle_t xQueueSet, const TickType_t
 xTicksToWait );
```

The following lists the xQueueSelectFromSet() parameters and return value.

xQueueSet

The handle of the queue set from which a queue handle or semaphore handle is being received (read). The queue set handle will have been returned from the call to xQueueCreateSet() used to create the queue set.

xTicksToWait

The maximum amount of time the calling task should remain in the Blocked state to wait to receive a queue or semaphore handle from the queue set, if all the queues and semaphore in the set are empty. If xTicksToWait is zero, then xQueueSelectFromSet() will return immediately if all the queues and semaphores in the set are empty. The block time is specified in tick periods, so the absolute time it represents depends on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds to 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.

Return Value

A return value that is not NULL will be the handle of a queue or semaphore that is known to contain data. If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state to wait for data to become available from a queue or semaphore in the set, but a handle was successfully read from the queue set before the block time expired. Handles

are returned as a QueueSetMemberHandle_t type, which can be cast to either a QueueHandle_t type or SemaphoreHandle_t type.

If the return value is NULL, then a handle could not be read from the queue set. 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 a queue or semaphore in the set, but the block time expired before that happened.

Using a Queue Set (Example 12)

This example creates two sending tasks and one receiving task. The sending tasks send data to the receiving task on two separate queues, one queue for each task. The two queues are added to a queue set, and the receiving task reads from the queue set to determine which of the two queues contain data.

The tasks, queues, and the queue set are all created in main().

```
/* Declare two variables of type QueueHandle_t. Both queues are added to the same queue
 set. */

static QueueHandle_t xQueue1 = NULL, xQueue2 = NULL;

/* Declare a variable of type QueueSetHandle_t. This is the queue set to which the two
 queues are added. */

static QueueSetHandle_t xQueueSet = NULL;

int main( void )

{

    /* Create the two queues, both of which send character pointers. The priority of the
 receiving task is above the priority of the sending tasks, so the queues will never have
 more than one item in them at any one time*/

    xQueue1 = xQueueCreate( 1, sizeof( char * ) );

    xQueue2 = xQueueCreate( 1, sizeof( char * ) );

    /* Create the queue set. Two queues will be added to the set, each of which can contain
 1 item, so the maximum number of queue handles the queue set will ever have to hold at one
 time is 2 (2 queues multiplied by 1 item per queue). */

    xQueueSet = xQueueCreateSet( 1 * 2 );

    /* Add the two queues to the set. */

    xQueueAddToSet( xQueue1, xQueueSet );

    xQueueAddToSet( xQueue2, xQueueSet );

    /* Create the tasks that send to the queues. */

    xTaskCreate( vSenderTask1, "Sender1", 1000, NULL, 1, NULL );

    xTaskCreate( vSenderTask2, "Sender2", 1000, NULL, 1, NULL );

    /* Create the task that reads from the queue set to determine which of the two queues
 contain data. */

    xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 2, NULL );
```

```
    /* Start the scheduler so the created tasks start executing. */

    vTaskStartScheduler();

    /* As normal, vTaskStartScheduler() should not return, so the following lines  will
 never execute. */

    for( ;; );

    return 0;

}
```

The first sending task uses xQueue1 to send a character pointer to the receiving task every 100 milliseconds. The second sending task uses xQueue2 to send a character pointer to the receiving task every 200 milliseconds. The character pointers are set to point to a string that identifies the sending task. The implementation of both sending tasks is shown here.

```
void vSenderTask1( void *pvParameters )

{

    const TickType_t xBlockTime = pdMS_TO_TICKS( 100 );

    const char * const pcMessage = "Message from vSenderTask1\r\n";

    /* As per most tasks, this task is implemented within an infinite loop. */

    for( ;; )

    {

        /* Block for 100ms. */

        vTaskDelay( xBlockTime );

        /* Send this task's string to xQueue1. It is not necessary to use a block time,
 even though the queue can only hold one item. This is because the priority of the task
 that reads from the queue is higher than the priority of this task. As soon as this task
 writes to the queue, it will be preempted by the task that reads from the queue, so the
 queue will already be empty  again by the time the call to xQueueSend() returns. The block
 time is set to 0. */

        xQueueSend( xQueue1, &pcMessage, 0 );

    }

}

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

void vSenderTask2( void *pvParameters )

{

    const TickType_t xBlockTime = pdMS_TO_TICKS( 200 );

    const char * const pcMessage = "Message from vSenderTask2\r\n";

    /* As per most tasks, this task is implemented within an infinite loop. */

    for( ;; )

    {
```

```
        /* Block for 200ms. */

        vTaskDelay( xBlockTime );

        /* Send this task's string to xQueue2. It is not necessary to use a block time,
 even though the queue can only hold one item. This is because the priority of the task
 that reads from the queue is higher than the priority of this task. As soon as this task
 writes to the queue, it will be preempted by the task that reads from the queue, so the
 queue will already be empty again by the time the call to xQueueSend() returns. The block
 time is set to 0. */

        xQueueSend( xQueue2, &pcMessage, 0 );

    }

}
```

The queues that are written to by the sending tasks are members of the same queue set. Each time a task sends to one of the queues, the handle of the queue is sent to the queue set. The receiving task calls xQueueSelectFromSet() to read the queue handles from the queue set. After the receiving task has received a queue handle from the set, it knows the queue referenced by the received handle contains data, so it reads the data from the queue directly. The data it reads from the queue is a pointer to a string, which the receiving task prints out.

If a call to xQueueSelectFromSet() times out, then it will return NULL. In the preceding code, xQueueSelectFromSet() is called with an indefinite block time, so will never time out and can only return a valid queue handle. Therefore, the receiving task does not need to check to see if xQueueSelectFromSet() returned NULL before the return value is used.

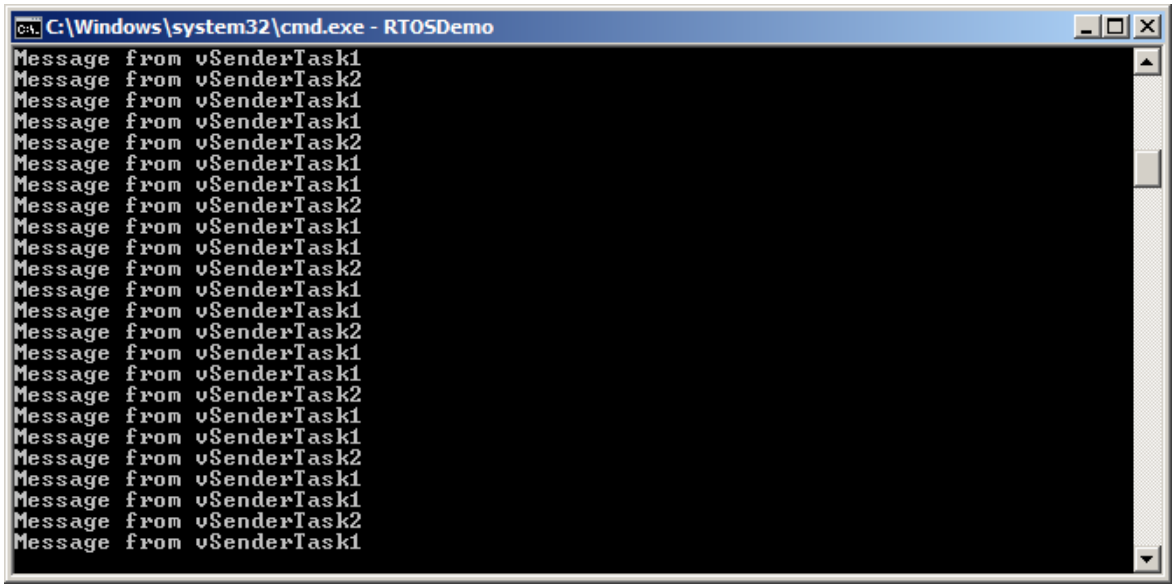
xQueueSelectFromSet() will only return a queue handle if the queue referenced by the handle contains data, so it is not necessary to use a block time when reading from the queue.

The implementation of the receive task is shown here.

```
void vReceiverTask( void *pvParameters )

{

    QueueHandle_t xQueueThatContainsData;

    char *pcReceivedString;

    /* As per most tasks, this task is implemented within an infinite loop.*/

    for( ;; )

    {

        /* Block on the queue set to wait for one of the queues in the set to contain
 data.  Cast the QueueSetMemberHandle_t value returned from xQueueSelectFromSet() to a
 QueueHandle_t because it is known all the members of the set are queues (the queue set
 does not contain any semaphores). */

        xQueueThatContainsData = ( QueueHandle_t ) xQueueSelectFromSet(xQueueSet,
 portMAX_DELAY );

        /* An indefinite block time was used when reading from the queue set, so
 xQueueSelectFromSet() will not have returned unless one of the queues in the set contained
 data, and xQueueThatContainsData cannot be NULL. Read from the queue. It is not necessary
 to specify a block time because it is known the queue contains data. The block time is set
 to 0. */
```

```
        xQueueReceive( xQueueThatContainsData, &pcReceivedString, 0 );

        /* Print the string received from the queue. */

        vPrintString( pcReceivedString );

    }

}
```

The output is shown here. The receiving task receives strings from both sending tasks. The block time used by vSenderTask1() is half of the block time used by vSenderTask2(), causing the strings sent by vSenderTask1() to be printed out twice as often as those sent by vSenderTask2().



More Realistic Queue Set Use Cases

In the previous example, the queue set contained two queues only. The queues were used to send a character pointer. In a real application, a queue set might contain both queues and semaphores, and the queues might not all hold the same data type. When this is the case, you need to test the value returned by xQueueSelectFromSet() before the returned value is used.

The following code shows how to use the value returned from xQueueSelectFromSet() when the set has the following members:

1. A binary semaphore.
2. A queue from which character pointers are read.
3. A queue from which uint32_t values are read.

This code assumes the queues and semaphore have already been created and added to the queue set.

```
/* The handle of the queue from which character pointers are received. */

QueueHandle_t xCharPointerQueue;

/* The handle of the queue from which uint32_t values are received.*/
```

```
QueueHandle_t xUint32tQueue;

/* The handle of the binary semaphore. */

SemaphoreHandle_t xBinarySemaphore;

/* The queue set to which the two queues and the binary semaphore belong. */

QueueSetHandle_t xQueueSet;

void vAMoreRealisticReceiverTask( void *pvParameters )

{

    QueueSetMemberHandle_t xHandle;

            char *pcReceivedString;

    uint32_t ulRecievedValue;

    const TickType_t xDelay100ms = pdMS_TO_TICKS( 100 );

    for( ;; )

    {

        /* Block on the queue set for a maximum of 100ms to wait for one of the members of
 the set to contain data. */

        xHandle = xQueueSelectFromSet( xQueueSet, xDelay100ms );

        /* Test the value returned from xQueueSelectFromSet(). If the returned value
 is NULL, then the call to xQueueSelectFromSet() timed out. If the returned value is
 not NULL, then the returned value will be the handle of one of the set's members. The
 QueueSetMemberHandle_t value can be cast to either a QueueHandle_t or a SemaphoreHandle_t.
 Whether an explicit cast is required depends on the compiler. */

        if( xHandle == NULL )

        {

            /* The call to xQueueSelectFromSet() timed out. */

        }

        else if( xHandle == ( QueueSetMemberHandle_t ) xCharPointerQueue )

        {

            /* The call to xQueueSelectFromSet() returned the handle of the queue that
 receives character pointers. Read from the queue. The queue is known to contain data, so a
 block time of 0 is used. */

            xQueueReceive( xCharPointerQueue, &pcReceivedString, 0 );

            /* The received character pointer can be processed here... */

        }

        else if( xHandle == ( QueueSetMemberHandle_t ) xUint32tQueue )

        {

            /* The call to xQueueSelectFromSet() returned the handle of the queue that
 receives uint32_t types. Read from the queue. The queue is known to contain data, so a
 block time of 0 is used. */
```

```
            xQueueReceive(xUint32tQueue, &ulRecievedValue, 0 );

            /* The received value can be processed here... */

         }

        else if( xHandle == ( QueueSetMemberHandle_t ) xBinarySemaphore )

        {

            /* The call to xQueueSelectFromSet() returned the handle of the binary
 semaphore. Take the semaphore now. The semaphore is known to be available, so a block time
 of 0 is used. */

            xSemaphoreTake( xBinarySemaphore, 0 );

            /* Whatever processing is necessary when the semaphore is taken can be
 performed here... */

        }

    }

}
```

Using a Queue to Create a Mailbox

There is no consensus in the embedded community on the meaning of the term *mailbox*. In this guide, we use the term to refer to a queue that has a length of one. A queue might get described as a mailbox because of the way it is used in the application, rather than because it has a functional difference to a queue.

- A queue is used to send data from one task to another task, or from an interrupt service routine to a task. The sender places an item in the queue, and the receiver removes the item from the queue. The data passes through the queue from the sender to the receiver.
- A mailbox is used to hold data that can be read by any task or interrupt service routine. The data does not pass through the mailbox. Instead, it remains in the mailbox until it is overwritten. The sender overwrites the value in the mailbox. The receiver reads the value from the mailbox, but does not remove the value from the mailbox.

The xQueueOverwrite() and xQueuePeek() API functions allow a queue to be used as a mailbox.

The following code shows a queue being created for use as a mailbox.

```
   /* A mailbox can hold a fixed-size data item. The size of the data item is set when the
 mailbox (queue) is created. In this example, the mailbox is created to hold an Example_t
 structure. Example_t includes a timestamp to allow the data held in the mailbox to note
 the time at which the mailbox was last updated. The timestamp used in this example is for
 demonstration purposes only. A mailbox can hold any data the application writer wants, and
 the data does not need to include a timestamp. */

typedef struct xExampleStructure

{

    TickType_t xTimeStamp;

    uint32_t ulValue;
```

```
} Example_t;

/* A mailbox is a queue, so its handle is stored in a variable of type

QueueHandle_t. */

QueueHandle_t xMailbox;

void vAFunction( void )

{

    /* Create the queue that is going to be used as a mailbox. The queue has a length of 1
 to allow it to be used with the xQueueOverwrite() API function, which is described below.
 */

    xMailbox = xQueueCreate( 1, sizeof( Example_t ) );

}
```

xQueueOverwrite() API Function

Like the xQueueSendToBack() API function, the xQueueOverwrite() API function sends data to a queue. Unlike xQueueSendToBack(), if the queue is already full, then xQueueOverwrite() will overwrite data that is already in the queue.

xQueueOverwrite() should only be used with queues that have a length of one. That restriction avoids the need for the function's implementation to make an arbitrary decision as to which item in the queue to overwrite if the queue is full.

Note: Do not call xQueueOverwrite() from an interrupt service routine. Use the interrupt-safe version, xQueueOverwriteFromISR(), instead.

The xQueueOverwrite() API function prototype is shown here.

```
BaseType_t xQueueOverwrite( QueueHandle_t xQueue, const void * pvItemToQueue );
```

The following table lists the xQueueOverwrite() 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.
Returned value	xQueueOverwrite() will write to the queue even when the queue is full, so pdPASS is the only possible return value.

The following code shows xQueueOverwrite() being used to write to the mailbox (queue) that was created earlier.

```
    void vUpdateMailbox( uint32_t ulNewValue )

{

    /* Example_t was defined in the earlier code example. */

    Example_t xData;

    /* Write the new data into the Example_t structure.*/

    xData.ulValue = ulNewValue;

    /* Use the RTOS tick count as the timestamp stored in the Example_t structure. */

    xData.xTimeStamp = xTaskGetTickCount();

    /* Send the structure to the mailbox, overwriting any data that is already in the
 mailbox. */

    xQueueOverwrite( xMailbox, &xData );

}
```

xQueuePeek() API Function

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

Note: Do not call xQueuePeek() from an interrupt service routine. Use the interrupt-safe version, xQueuePeekFromISR(), instead.

xQueuePeek() has the same function parameters and return value as xQueueReceive().

The following code shows xQueuePeek() being used to receive the item posted to the mailbox (queue) created in a previous sample.

```
BaseType_t vReadMailbox( Example_t *pxData )

{

    TickType_t xPreviousTimeStamp;

    BaseType_t xDataUpdated;

    /* This function updates an Example_t structure with the latest value received from the
 mailbox. Record the timestamp already contained in *pxData before it gets overwritten by
 the new data. */

    xPreviousTimeStamp = pxData->xTimeStamp;

    /* Update the Example_t structure pointed to by pxData with the data contained in
 the mailbox. If xQueueReceive() was used here, then the mailbox would be left empty
 and the data could not then be read by any other tasks. Using xQueuePeek() instead of
 xQueueReceive() ensures the data remains in the mailbox. A block time is specified, so
 the calling task will be placed in the Blocked state to wait for the mailbox to contain
 data should the mailbox be empty. An infinite block time is used, so it is not necessary
```

```
 to check the value returned from xQueuePeek(). xQueuePeek() will only return when data is
 available. */

    xQueuePeek( xMailbox, pxData, portMAX_DELAY );

    /* Return pdTRUE if the value read from the mailbox has been updated since this
 function was last called. Otherwise, return pdFALSE. */

    if( pxData->xTimeStamp > xPreviousTimeStamp )

    {

        xDataUpdated = pdTRUE;

    }

    else

    {

        xDataUpdated = pdFALSE;

    }

    return xDataUpdated;

}
```

Software Timer Management

This section covers:

- The characteristics of a software timer compared to the characteristics of a task.
- The RTOS daemon task.
- The timer command queue.
- The difference between a one-shot software timer and a periodic software timer.
- How to create, start, reset, and change the period of a software timer.

Software timers are used to schedule the execution of a function at a set time in the future or periodically with a fixed frequency. The function executed by the software timer is called the software timer's *callback function*.

Software timers are implemented by and under the control of the FreeRTOS kernel. They do not require hardware support. They are not related to hardware timers or hardware counters.

In keeping with the FreeRTOS philosophy of using innovative design to ensure maximum efficiency, software timers do not use any processing time unless a software timer callback function is executing.

Software timer functionality is optional. To include software timer functionality:

1. Build the FreeRTOS source file, FreeRTOS/Source/timers.c, as part of your project.
2. In FreeRTOSConfig.h, set configUSE_TIMERS to 1.

Software Timer Callback Functions

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

The callback function prototype is shown here.

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

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

Note: Software timer callback functions execute in the context of a task that is created automatically when the FreeRTOS scheduler is started. Therefore, they must never call FreeRTOS API functions that will result in the calling task entering the Blocked state. You can call functions like xQueueReceive(), but only if the function's xTicksToWait parameter (which specifies the function's block time) is set to 0. You cannot call functions like vTaskDelay() because that will always place the calling task into the Blocked state.

Attributes and States of a Software Timer

Period of a Software Timer

A software timer's *period* is the time between the start of the software timer and the execution of its callback function.

One-Shot and Auto-Reload Timers

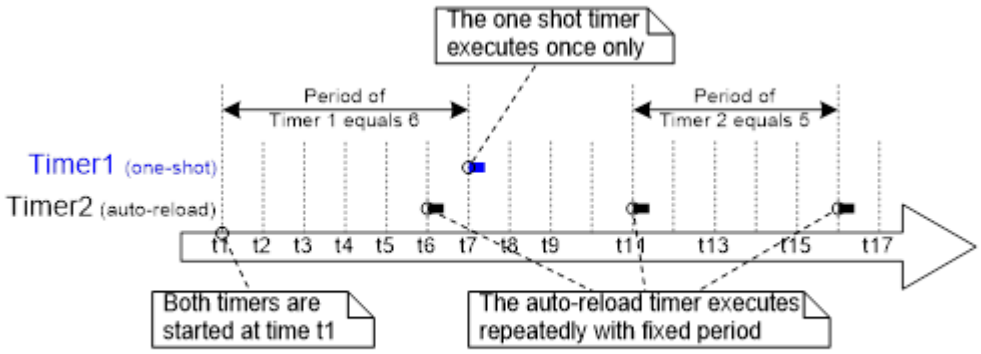
There are two types of software timer:

- One-shot timers

  After it's started, a one-shot timer will execute its callback function once only. A one-shot timer can be restarted manually, but it will not restart itself.
- Auto-reload timers

  After it's started, an auto-reload timer will restart itself each time it expires, resulting in periodic execution of its callback function.

The following figure shows the difference in behavior between a one-shot timer and an auto-reload timer. The dashed vertical lines mark the times at which a tick interrupt occurs.



- Timer 1 is a one-shot timer that has a period of 6 ticks. It is started at time t1, so its callback function executes 6 ticks later, at time t7. Because timer 1 is a one-shot timer, its callback function does not execute again.
- Timer 2 is an auto-reload timer that has a period of 5 ticks. It is started at time t1, so its callback function executes every 5 ticks after time t1. In the figure, this is at times t6, t11, and t16.

Software Timer States

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

- Dormant

  A dormant software timer exists and can be referenced by its handle, but is not running, so its callback functions will not execute.
- Running

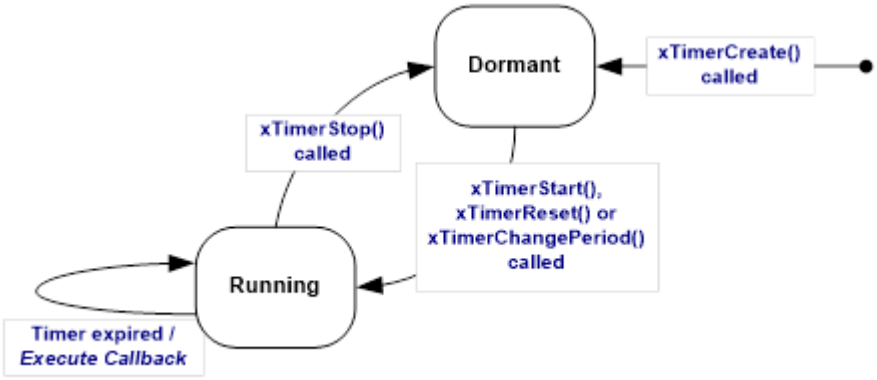
  A running software timer will execute its callback function after a time equal to its period has elapsed since the software timer entered the Running state or since the software timer was last reset.

The following two figures show the possible transitions between the dormant and running states for an auto-reload timer and a one-shot timer, respectively. The key difference between the two figures is the state entered after the timer has expired. The auto-reload timer executes its callback function, and

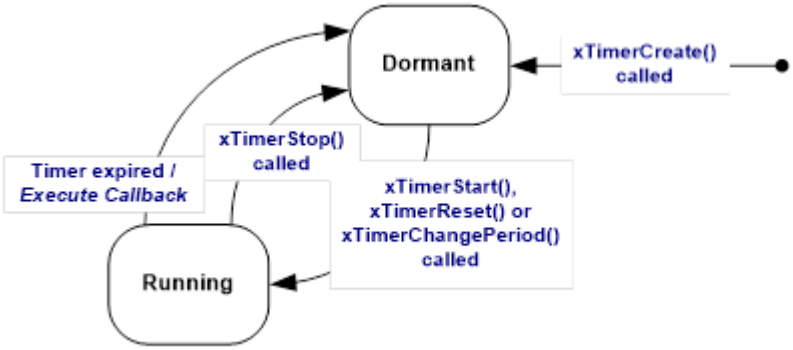
then re-enters the Running state. The one-shot timer executes its callback function, and then enters the dormant state.

The xTimerDelete() API function deletes a timer. A timer can be deleted at any time.

The following figure shows auto-reload software timer states and transitions.



The following figure shows one-shot software timer states and transitions.



The Context of a Software Timer

RTOS Daemon (Timer Service) Task

All software timer callback functions execute in the context of the same RTOS daemon (or timer service) task. (The task used to be called the timer service task because originally it was only used to execute software timer callback functions. Now the same task is used for other purposes too, so it is known by the more generic name of the RTOS daemon task.)

The daemon task is a standard FreeRTOS task that is created automatically when the scheduler is started. Its priority and stack size are set by the configTIMER_TASK_PRIORITY and configTIMER_TASK_STACK_DEPTH compile-time configuration constants, respectively. Both constants are defined in FreeRTOSConfig.h.

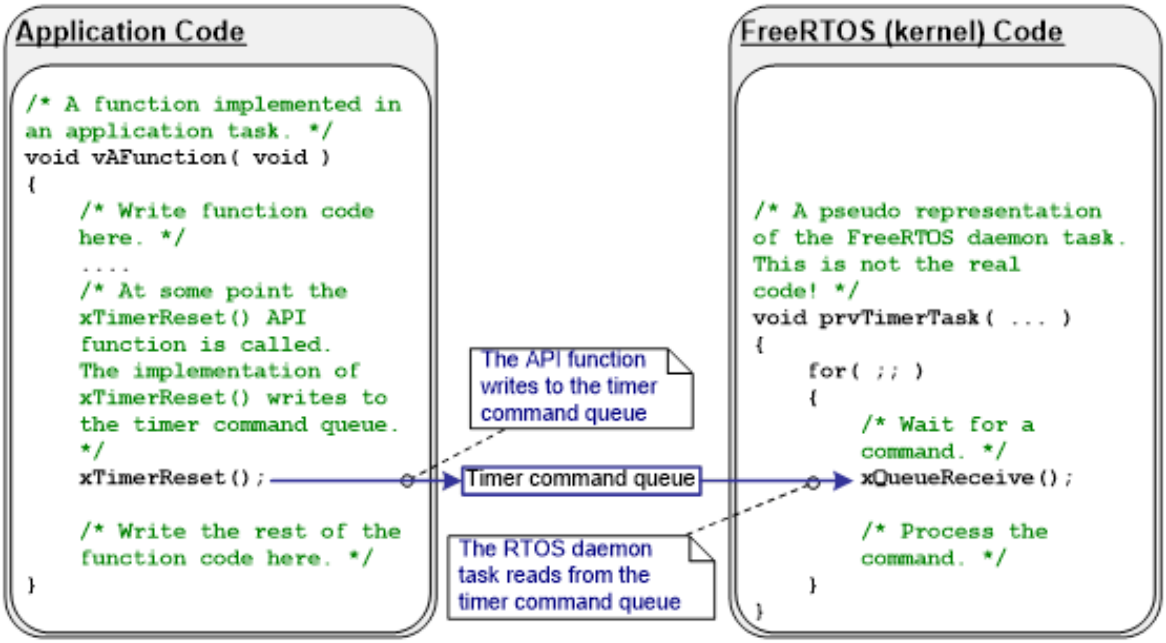
Software timer callback functions must not call FreeRTOS API functions that will result in the calling task entering the Blocked state because to do so will result in the daemon task entering the Blocked state.

Timer Command Queue

Software timer API functions send commands from the calling task to the daemon task on a queue called the *timer command queue*. Examples of commands include 'start a timer', 'stop a timer,' and 'reset a timer.'

The timer command queue is a standard FreeRTOS queue that is created automatically when the scheduler is started. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH compile-time configuration constant in FreeRTOSConfig.h.

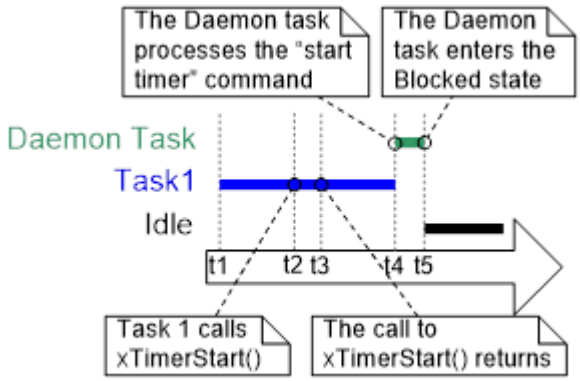
The following figure shows the command queue being used by a software timer API function to communicate with the RTOS daemon task.



Daemon Task Scheduling

The daemon task is scheduled like any other FreeRTOS task. It will process commands or execute timer callback functions only when it is the highest priority task that is able to run. The following figures demonstrate how the configTIMER_TASK_PRIORITY setting affects the execution pattern.

This figure shows the execution pattern when the priority of the daemon task is below the priority of a task that calls the xTimerStart() API function.



1. At time t1

   Task 1 is in the Running state, and the daemon task is in the Blocked state.

   The daemon task will leave the Blocked state if a command is sent to the timer command queue, in which case it will process the command. If a software timer expires, it will execute the software timer's callback function.

2. At time t2

   Task 1 calls xTimerStart().

   xTimerStart() sends a command to the timer command queue, causing the daemon task to leave the Blocked state. The priority of Task 1 is higher than the priority of the daemon task, so the daemon task does not preempt Task 1.

   Task 1 is still in the Running state. The daemon task has left the Blocked state and entered the Ready state.

3. At time t3

   Task 1 completes executing the xTimerStart() API function. Task 1 executed xTimerStart() from the start of the function to the end of the function without leaving the Running state.

4. At time t4

   Task 1 calls an API function that results in it entering the Blocked state. The daemon task is now the highest priority task in the Ready state, so the scheduler selects the daemon task as the task to enter the Running state. The daemon task then starts to process the command sent to the timer command queue by Task 1.

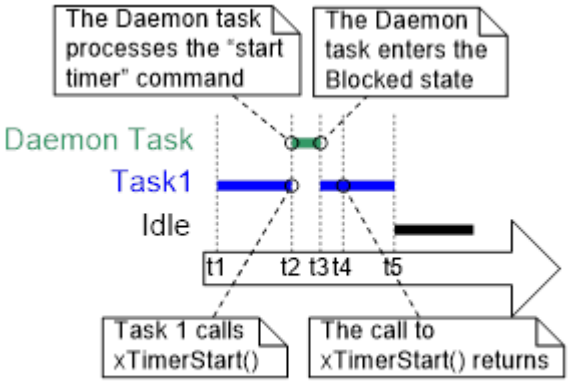
   Note: The time at which the software timer that is being started will expire is calculated from the time the 'start a timer' command was sent to the timer command queue. It is not calculated from the time the daemon task received the 'start a timer' command from the timer command queue.

5. At time t5

   The daemon task has completed processing the command sent to it by Task 1 and attempts to receive more data from the timer command queue. The timer command queue is empty, so the daemon task re-enters the Blocked state. The daemon task will leave the Blocked state again if a command is sent to the timer command queue or a software timer expires.

   The Idle task is now the highest priority task in the Ready state, so the scheduler selects the Idle task as the task to enter the Running state.

The following figure shows a scenario similar to the previous figure. This time the priority of the daemon task is above the priority of the task that calls xTimerStart().



1. At time t1

   As before, Task 1 is in the Running state, and the daemon task is in the Blocked state.

2. At time t2

   Task 1 calls xTimerStart().

   xTimerStart() sends a command to the timer command queue, causing the daemon task to leave the Blocked state. The priority of the daemon task is higher than the priority of Task 1, so the scheduler selects the daemon task as the task to enter the Running state.

   Task 1 was preempted by the daemon task before it had completed executing the xTimerStart() function, and is now in the Ready state.

   The daemon task starts to process the command sent to the timer command queue by Task 1.

3. At time t3

   The daemon task has completed processing the command sent to it by Task 1 and attempts to receive more data from the timer command queue. The timer command queue is empty, so the daemon task re-enters the Blocked state.

   Task 1 is now the highest priority task in the Ready state, so the scheduler selects Task 1 as the task to enter the Running state.

4. At time t4

   Task 1 was preempted by the daemon task before it had completed executing the xTimerStart() function and exits (returns from) xTimerStart() only after it has re-entered the Running state.

5. At time t5

   Task 1 calls an API function that results in it entering the Blocked state. The Idle task is now the highest priority task in the Ready state, so the scheduler selects the Idle task as the task to enter the Running state.

In the figure for the first scenario, time passed between Task 1 sending a command to the timer command queue and the daemon task receiving and processing the command. The daemon task had received and processed the command sent to it by Task 1 before Task 1 returned from the function that sent the command.

Commands sent to the timer command queue contain a timestamp. The timestamp is used to account for any time that passes between a command being sent by an application task and being processed by the daemon task. For example, if a 'start a timer' command is sent to start a timer that has a period of 10 ticks, the timestamp is used to ensure the timer being started expires 10 ticks after the command was sent, not 10 ticks after the command was processed by the daemon task.

Creating and Starting a Software Timer

xTimerCreate() API Function

FreeRTOS V9.0.0 also includes the xTimerCreateStatic() function, which allocates the memory required to create a timer statically at compile time. A software timer must be explicitly created before it can be used.

Software timers are referenced by variables of type TimerHandle_t. xTimerCreate() is used to create a software timer and returns a TimerHandle_t to reference the software timer it creates. Software timers are created in the Dormant state.

Software timers can be created before the scheduler is running or from a task after the scheduler has been started.

The xTimerCreate() API function prototype is shown here.

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

The following table lists the xTimerCreate() parameters and return value.

Parameter Name/ Returned Value	Description
pcTimerName	A descriptive name for the timer. This is not used by FreeRTOS. It is included only as a debugging aid. Identifying a timer by a human-readable name is much simpler than attempting to identify it by its handle.
xTimerPeriodInTicks	The timer's period specified in ticks. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds into ticks.
uxAutoReload	Set uxAutoReload to pdTRUE to create an auto-reload timer. Set uxAutoReload to pdFALSE to create a one-shot timer.
pvTimerID	Each software timer has an ID value. The ID is a void pointer and can be used by the application writer for any purpose. The ID is particularly useful when the same callback function is used by more than one software timer because it can be used to provide timer-specific storage. pvTimerID sets an initial value for the ID of the task being created.
pxCallbackFunction	Software timer callback functions are simply C functions that conform to the prototype shown

	in Software Timer Callback Functions (p. 98). The pxCallbackFunction parameter is a pointer to the function (in effect, just the function name) to use as the callback function for the software timer being created.
Returned value	If NULL is returned, then the software timer cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the data structure. If a non-NULL value is returned, the software timer has been created successfully. The returned value is the handle of the created timer.

xTimerStart() API Function

xTimerStart() is used to start a software timer that is in the Dormant state or reset (restart) one that is in the Running state. xTimerStop() is used to stop a software timer that is in the Running state. Stopping a software timer is the same as transitioning the timer into the Dormant state.

You can call xTimerStart() before the scheduler is started, but the software timer will not start until the time at which the scheduler starts.

Note: Do not call xTimerStart() from an interrupt service routine. Use the interrupt-safe version, xTimerStartFromISR(), instead.

The xTimerStart() API function prototype is shown here.

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

The following table lists the xTimerStart() parameters and return value.

Parameter Name/ Returned Value	Description
xTimer	The handle of the software timer being started or reset. The handle will have been returned from the call to xTimerCreate() used to create the software timer.
xTicksToWait	xTimerStart() uses the timer command queue to send the 'start a timer' command to the daemon task. xTicksToWait specifies the maximum amount of time the calling task should remain in the Blocked state to wait for space to become available on the timer command queue, should the queue already be full. xTimerStart() will return immediately if xTicksToWait is zero and the timer command queue is already full. The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into ticks.

	If INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h, then setting xTicksToWait to portMAX_DELAY will result in the calling task remaining in the Blocked state indefinitely (without a timeout) to wait for space to become available in the timer command queue. If xTimerStart() is called before the scheduler has been started, then the value of xTicksToWait is ignored, and xTimerStart() behaves as if xTicksToWait had been set to zero.
Returned value	There are two possible return values: 1. pdPASS pdPASS will be returned only if the 'start a timer' command was successfully sent to the timer command queue. If the priority of the daemon task is above the priority of the task that called xTimerStart(), then the scheduler will ensure the start command is processed before xTimerStart() returns. This is because the daemon task will preempt the task that called xTimerStart() as soon as there is data in the timer command queue. If a block time was specified (xTicksToWait was not zero), then it is possible the calling task was placed into the Blocked state to wait for space to become available in the timer command queue before the function returned, but data was successfully written to the timer command queue before the block time expired. 2. pdFALSE pdFALSE will be returned if the 'start a timer' command could not be written to the timer command 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 the daemon task to make room in the timer command queue, but the specified block time expired before that happened.

Creating One-Shot and Auto-Reload Timers (Example 13)

This example creates and starts a one-shot timer and an auto-reload timer.

```
/* The periods assigned to the one-shot and auto-reload timers are 3.333 second and half a
 second, respectively. */

#define mainONE_SHOT_TIMER_PERIOD pdMS_TO_TICKS( 3333 )

#define mainAUTO_RELOAD_TIMER_PERIOD pdMS_TO_TICKS( 500 )

int main( void )

{

    TimerHandle_t xAutoReloadTimer, xOneShotTimer;

    BaseType_t xTimer1Started, xTimer2Started;

    /* Create the one-shot timer, storing the handle to the created timer in xOneShotTimer.
 */

    xOneShotTimer = xTimerCreate( /* Text name for the software timer - not
 used by FreeRTOS. */ "OneShot", /* The software timer's period, in ticks. */
 mainONE_SHOT_TIMER_PERIOD, /* Setting uxAutoRealod to pdFALSE creates a one-shot software
 timer. */ pdFALSE, /* This example does not use the timer ID. */ 0, /* The callback
 function to be used by the software timer being created. */ prvOneShotTimerCallback );

    /* Create the auto-reload timer, storing the handle to the created timer in
 xAutoReloadTimer. */

    xAutoReloadTimer = xTimerCreate( /* Text name for the software timer - not
 used by FreeRTOS. */ "AutoReload", /* The software timer's period, in ticks. */
 mainAUTO_RELOAD_TIMER_PERIOD, /* Setting uxAutoRealod to pdTRUE creates an auto-reload
 timer. */ pdTRUE, /* This example does not use the timer ID. */ 0, /* The callback
 function to be used by the software timer being created. */ prvAutoReloadTimerCallback );

    /* Check the software timers were created. */

    if( ( xOneShotTimer != NULL ) && ( xAutoReloadTimer != NULL ) )

    {

        /* Start the software timers, using a block time of 0 (no block time). The
 scheduler has not been started yet so any block time specified here would be ignored
 anyway. */

        xTimer1Started = xTimerStart( xOneShotTimer, 0 );

        xTimer2Started = xTimerStart( xAutoReloadTimer, 0 );

        /* The implementation of xTimerStart() uses the timer command queue, and
 xTimerStart()  will fail if the timer command queue gets full. The timer service task does
 not get created until the scheduler is started, so all commands sent to the command queue
 will stay in the queue until after the scheduler has been started. Check both calls to
 xTimerStart() passed. */

        if( ( xTimer1Started == pdPASS ) && ( xTimer2Started == pdPASS ) )

        {

            /* Start the scheduler. */

            vTaskStartScheduler();

        }

    }
```

```
    /* As always, this line should not be reached. */

    for( ;; );

}
```

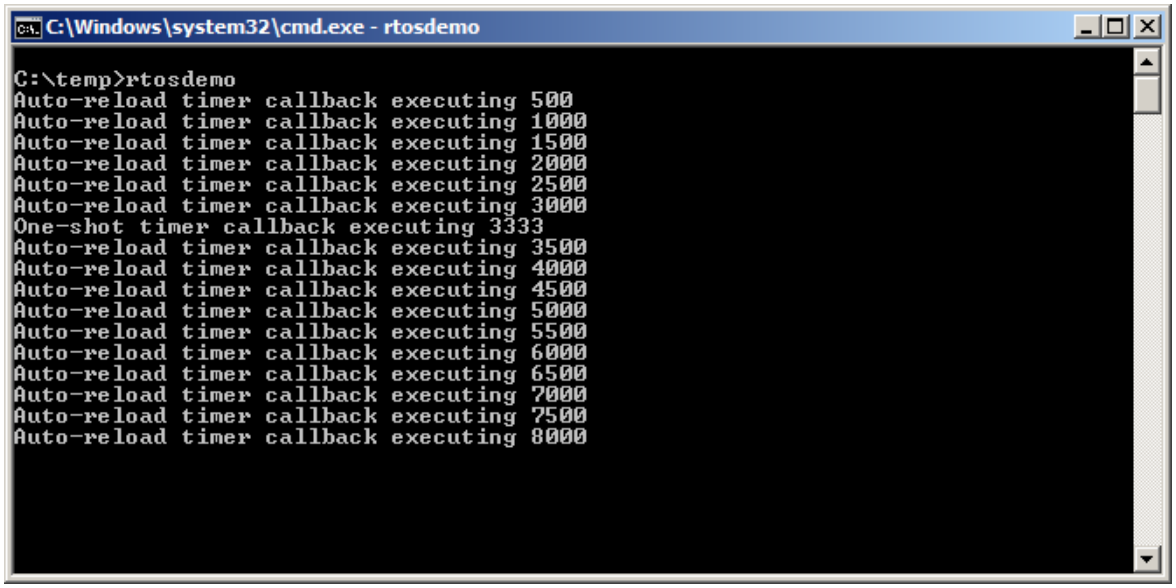
The timers' callback functions just print a message each time they are called. The implementation of the one-shot timer callback function is shown here.

```
static void prvOneShotTimerCallback( TimerHandle_t xTimer )

{

    TickType_t xTimeNow;

    /* Obtain the current tick count. */

    xTimeNow = xTaskGetTickCount();

    /* Output a string to show the time at which the callback was executed. */

    vPrintStringAndNumber( "One-shot timer callback executing", xTimeNow );

    /* File scope variable. */

    ulCallCount++;

}
```

The implementation of the auto-reload timer callback function is shown here.

```
static void prvAutoReloadTimerCallback( TimerHandle_t xTimer )

{

    TickType_t xTimeNow;

    /* Obtain the current tick count. */

    xTimeNow = xTaskGetTickCount();

    /* Output a string to show the time at which the callback was executed. */

    vPrintStringAndNumber( "Auto-reload timer callback executing", xTimeNow);

    ulCallCount++;

}
```

Executing this example produces the following output. It shows the auto-reload timer's callback function executing with a fixed period of 500 ticks (mainAUTO_RELOAD_TIMER_PERIOD is set to 500) and the one-shot timer's callback function executing only once, when the tick count is 3333 (mainONE_SHOT_TIMER_PERIOD is set to 3333).



Timer ID

Each software timer has an ID, which is a tag value that can be used by the application writer for any purpose. The ID is stored in a void pointer (`void *`), so it can store an integer value directly, point to any other object, or be used as a function pointer.

An initial value is assigned to the ID when the software timer is created. The ID can be updated using the vTimerSetTimerID() API function and queried using the pvTimerGetTimerID() API function.

Unlike other software timer API functions, vTimerSetTimerID() and pvTimerGetTimerID() access the software timer directly. They do not send a command to the timer command queue.

vTimerSetTimerID() API Function

The vTimerSetTimerID() function prototype is shown here.

```
void vTimerSetTimerID( const TimerHandle_t xTimer, void *pvNewID );
```

The following table lists the vTimerSetTimerID() parameters.

Parameter Name/ Returned Value	Description
xTimer	The handle of the software timer being updated with a new ID value. The handle will have been returned from the call to xTimerCreate() used to create the software timer.
pvNewID	The value to which the software timer's ID will be set.

pvTimerGetTimerID() API Function

The pvTimerGetTimerID() function prototype is shown here.

```
void *pvTimerGetTimerID( TimerHandle_t xTimer );
```

The following table lists the pvTimerGetTimerID() parameter and return value.

Parameter Name/ Returned Value	Description
xTimer	The handle of the software timer being queried. The handle will have been returned from the call to xTimerCreate() used to create the software timer.
Returned value	The ID of the software timer being queried.

Using the Callback Function Parameter and the Software Timer ID (Example 14)

The same callback function can be assigned to more than one software timer. When that is done, the callback function parameter is used to determine which software timer expired.

Example 13 used two separate callback functions: one for the one-shot timer and the other for the auto-reload timer. This example creates similar functionality, but assigns a single callback function to both software timers.

The following code shows how to use the same callback function prvTimerCallback() for both timers.

```
/* Create the one-shot software timer, storing the handle in xOneShotTimer. */

xOneShotTimer = xTimerCreate( "OneShot", mainONE_SHOT_TIMER_PERIOD, pdFALSE, /* The
 timer's ID is initialized to 0. */ 0, /* prvTimerCallback() is used by both timers. */
 prvTimerCallback );

/* Create the auto-reload software timer, storing the handle in xAutoReloadTimer */

xAutoReloadTimer = xTimerCreate( "AutoReload", mainAUTO_RELOAD_TIMER_PERIOD, pdTRUE, /* The
 timer's ID is initialized to 0. */ 0, /* prvTimerCallback() is used by both timers. */
 prvTimerCallback );
```

prvTimerCallback() will execute when either timer expires. The implementation of prvTimerCallback() uses the function's parameter to determine if it was called because the one-shot timer or the auto-reload timer expired.

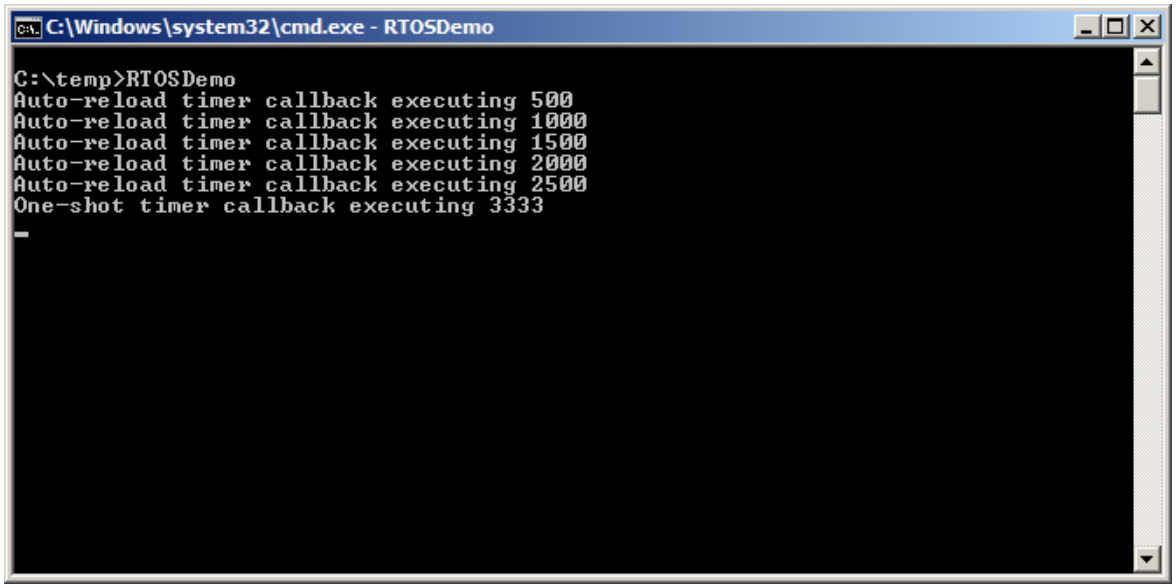
prvTimerCallback() also demonstrates how to use the software timer ID as timer-specific storage. Each software timer keeps a count of the number of times it has expired in its own ID. The auto-reload timer uses the count to stop itself the fifth time it executes.

The implementation of prvTimerCallback() is shown here.

```
static void prvTimerCallback( TimerHandle_t xTimer )

{

    TickType_t xTimeNow;

    uint32_t ulExecutionCount;
```

```
   /* A count of the number of times this software timer has expired is stored in the
timer's ID. Obtain the ID, increment it, then save it as the new ID value. The ID is a
void pointer, so is cast to a uint32_t. */

   ulExecutionCount = ( uint32_t ) pvTimerGetTimerID( xTimer );

   ulExecutionCount++;

   vTimerSetTimerID( xTimer, ( void * ) ulExecutionCount );

   /* Obtain the current tick count. */

   xTimeNow = xTaskGetTickCount();

   /* The handle of the one-shot timer was stored in xOneShotTimer when the timer was
created. Compare the handle passed into this function with xOneShotTimer to determine if
it was the one-shot or auto-reload timer that expired, then output a string to show the
time at which the callback was executed. */

   if( xTimer == xOneShotTimer )

   {

       vPrintStringAndNumber( "One-shot timer callback executing", xTimeNow );

   }

   else

   {

       /* xTimer did not equal xOneShotTimer, so it must have been the auto-reload timer
that expired. */

       vPrintStringAndNumber( "Auto-reload timer callback executing", xTimeNow );

       if( ulExecutionCount == 5 )

       {

           /* Stop the auto-reload timer after it has executed 5 times. This callback
function executes in the context of the RTOS daemon task, so must not call any functions
that might place the daemon task into the Blocked state. Therefore, a block time of 0 is
used. */

           xTimerStop( xTimer, 0 );

       }

   }

}
```

The output is shown here. The auto-reload timer executes five times only.



Changing the Period of a Timer

Every official FreeRTOS port is provided with one or more example projects. Most example projects are self-checking. An LED is used to give visual feedback of the project's status. If the self checks have always passed, then the LED is toggled slowly. If a self check has ever failed, then the LED is toggled quickly.

Some example projects perform the self checks in a task and use the vTaskDelay() function to control the rate at which the LED toggles. Other example projects perform the self checks in a software timer callback function and use the timer's period to control the rate at which the LED toggles.

xTimerChangePeriod() API Function

You use the xTimerChangePeriod() function to change the period of a software timer.

If xTimerChangePeriod() is used to change the period of a timer that is already running, then the timer will use the new period value to recalculate its expiry time. The recalculated expiry time is relative to when xTimerChangePeriod() was called, not when the timer was originally started.

If xTimerChangePeriod() is used to change the period of a timer that is in the Dormant state (a timer that is not running), then the timer will calculate an expiry time and transition to the Running state (the timer will start running).

Note: Do not call xTimerChangePeriod() from an interrupt service routine. Use the interrupt-safe version, xTimerChangePeriodFromISR(), instead.

The xTimerChangePeriod() API function prototype is shown here.

```
BaseType_t xTimerChangePeriod( TimerHandle_t xTimer, TickType_t xNewTimerPeriodInTicks,
 TickType_t xTicksToWait );
```

The following table lists the xTimerChangePeriod() parameters and return value.

Parameter Name/ Returned Value	Description

xTimer	The handle of the software timer being updated with a new period value. The handle will have been returned from the call to xTimerCreate() used to create the software timer.
xTimerPeriodInTicks	The new period for the software timer, specified in ticks. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds into ticks.
xTicksToWait	xTimerChangePeriod() uses the timer command queue to send the 'change period' command to the daemon task. xTicksToWait specifies the maximum amount of time the calling task should remain in the Blocked state to wait for space to become available on the timer command queue, should the queue already be full. xTimerChangePeriod() will return immediately if xTicksToWait is zero and the timer command queue is already full. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into ticks. If INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h, then setting xTicksToWait to portMAX_DELAY will result in the calling task remaining in the Blocked state indefinitely (without a timeout) to wait for space to become available in the timer command queue. If xTimerChangePeriod() is called before the scheduler has been started, then the value of xTicksToWait is ignored, and xTimerChangePeriod() behaves as if xTicksToWait had been set to zero.

Returned value	There are two possible return values: 1. pdPASS pdPASS will be returned only if data was successfully sent to the timer command queue. If a block time was specified (xTicksToWait was not zero), then it is possible the calling task was placed into the Blocked state to wait for space to become available in the timer command queue before the function returned, but data was successfully written to the timer command queue before the block time expired. 2. pdFALSE pdFALSE will be returned if the 'change period' command could not be written to the timer command 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 the daemon task to make room in the queue, but the specified block time expired before that happened.

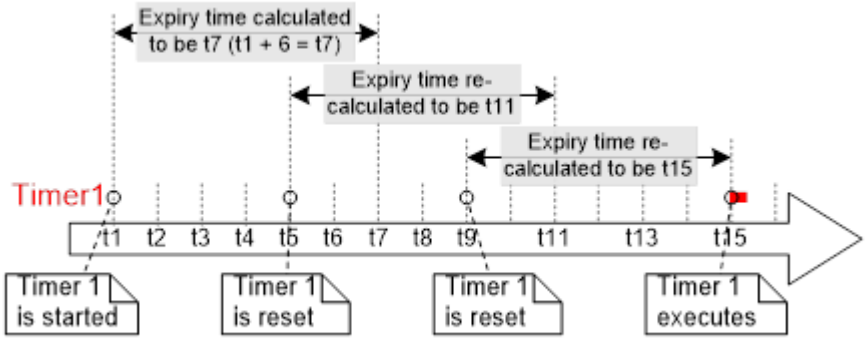
The following code shows how self-checking functionality in a software timer callback function can use xTimerChangePeriod() to increase the rate at which an LED toggles if a self check fails. The software timer that performs the self checks is referred to as the *check timer*.

```
/* The check timer is created with a period of 3000 milliseconds, resulting in the LED
 toggling every 3 seconds. If the self-checking functionality detects an unexpected state,
 then the check timer's period is changed to just 200 milliseconds, resulting in a much
 faster toggle rate. */

const TickType_t xHealthyTimerPeriod = pdMS_TO_TICKS( 3000 );

const TickType_t xErrorTimerPeriod = pdMS_TO_TICKS( 200 );

/* The callback function used by the check timer. */

static void prvCheckTimerCallbackFunction( TimerHandle_t xTimer )

{

    static BaseType_t xErrorDetected = pdFALSE;

    if( xErrorDetected == pdFALSE )

    {

        /* No errors have yet been detected. Run the self-checking function again.
 The function asks each task created by the example to report its own status, and also
 checks that all the tasks are actually still running (and so able to report their status
 correctly). */

        if( CheckTasksAreRunningWithoutError() == pdFAIL )
```

```
        {

            /* One or more tasks reported an unexpected status. An error might have
 occurred.  Reduce the check timer's period to increase the rate at which this callback
 function  executes, and in so doing also increase the rate at which the LED is toggled.
 This callback function is executing in the context of the RTOS daemon task, so a block
 time of 0 is used to ensure the Daemon task never enters the Blocked state. */

            xTimerChangePeriod( xTimer, /* The timer being updated. */ xErrorTimerPeriod, /
* The new period for the timer. */ 0 ); /* Do not block when sending this command. */

        }

        /* Latch that an error has already been detected. */

        xErrorDetected = pdTRUE;

    }

    /* Toggle the LED. The rate at which the LED toggles will depend on how
 often this function is called, which is determined by the period of the check
 timer. The timer's period will have been reduced from 3000ms to just 200ms if
 CheckTasksAreRunningWithoutError() has ever returned pdFAIL. */

    ToggleLED();

}
```

Resetting a Software Timer

Resetting a software timer means to restart the timer. The timer's expiry time is recalculated to be relative to when the timer was reset rather than when the timer was originally started. The following figure shows a timer that has a period of 6 being started and then reset twice before eventually expiring and executing its callback function.



- Timer 1 is started at time t1. It has a period of 6, so the time at which it will execute its callback function is originally calculated to be t7, which is 6 ticks after it was started.
- Timer 1 is reset before time t7 is reached, so before it had expired and executed its callback function. Timer 1 is reset at time t5, so the time at which it will execute its callback function is recalculated to be t11, which is 6 ticks after it was reset.
- Timer 1 is reset again before time t11, so before it had expired and executed its callback function. Timer 1 is reset at time t9, so the time at which it will execute its callback function is recalculated to be t15, which is 6 ticks after it was last reset.
- Timer 1 is not reset again, so it expires at time t15, and its callback function is executed accordingly.

xTimerReset() API Function

You use the xTimerReset() API function to reset a timer.

You can also use xTimerReset() to start a timer that is in the Dormant state.

Note: Do not call xTimerReset() from an interrupt service routine. Use the interrupt-safe version, xTimerResetFromISR(), instead.

The xTimerReset() API function prototype is shown here.

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

The following table lists the xTimerReset() parameters and return value.

Parameter Name/ Returned Value	Description
xTimer	The handle of the software timer being reset or started. The handle will have been returned from the call to xTimerCreate() used to create the software timer.
xTicksToWait	xTimerChangePeriod() uses the timer command queue to send the 'reset' command to the daemon task. xTicksToWait specifies the maximum amount of time the calling task should remain in the Blocked state to wait for space to become available on the timer command queue, should the queue already be full. xTimerReset() will return immediately if xTicksToWait is zero and the timer command queue is already full. If INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h, then setting xTicksToWait to portMAX_DELAY will result in the calling task remaining in the Blocked state indefinitely (without a timeout) to wait for space to become available in the timer command queue.
Returned value	There are two possible return values: 1. pdPASS pdPASS will be returned only if data was successfully sent to the timer command queue. If a block time was specified (xTicksToWait was not zero), then it is possible the calling task was placed into the Blocked state to wait for space to become available in the timer command queue before the function returned, but data was successfully written to the timer command queue before the block time expired. 2. pdFALSE

	pdFALSE will be returned if the 'reset' command could not be written to the timer command 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 the daemon task to make room in the queue, but the specified block time expired before that happened.

Resetting a Software Timer (Example 15)

This example simulates the behavior of the backlight on a cell phone. The backlight:

- Turns on when a key is pressed.
- Remains on provided other keys are pressed within a certain time period.
- Turns off automatically if no other keys are pressed within a certain time period.

A one-shot software timer is used to implement this behavior.

- The (simulated) backlight is turned on when a key is pressed and turned off in the software timer's callback function.
- The software timer is reset each time a key is pressed.
- The time period during which a key must be pressed to prevent the backlight from being turned off is therefore equal to the period of the software timer. If the software timer is not reset by a key press before the timer expires, then the timer's callback function executes and the backlight is turned off.

The xSimulatedBacklightOn variable holds the backlight state. If xSimulatedBacklightOn is set to pdTRUE, the backlight is on. If xSimulatedBacklightOn is set to pdFALSE, the backlight is off.

The software timer callback function is shown here.

```
static void prvBacklightTimerCallback( TimerHandle_t xTimer )

{

    TickType_t xTimeNow = xTaskGetTickCount();

    /* The backlight timer expired. Turn the backlight off. */

    xSimulatedBacklightOn = pdFALSE;

    /* Print the time at which the backlight was turned off. */

    vPrintStringAndNumber("Timer expired, turning backlight OFF at time\t\t", xTimeNow );

}
```

Using FreeRTOS allows your application to be event-driven. Event-driven designs use processing time efficiently. The time is used only if an event has occurred. Processing time is not wasted polling for events that have not occurred. The example could not be made event-driven because it is not practical

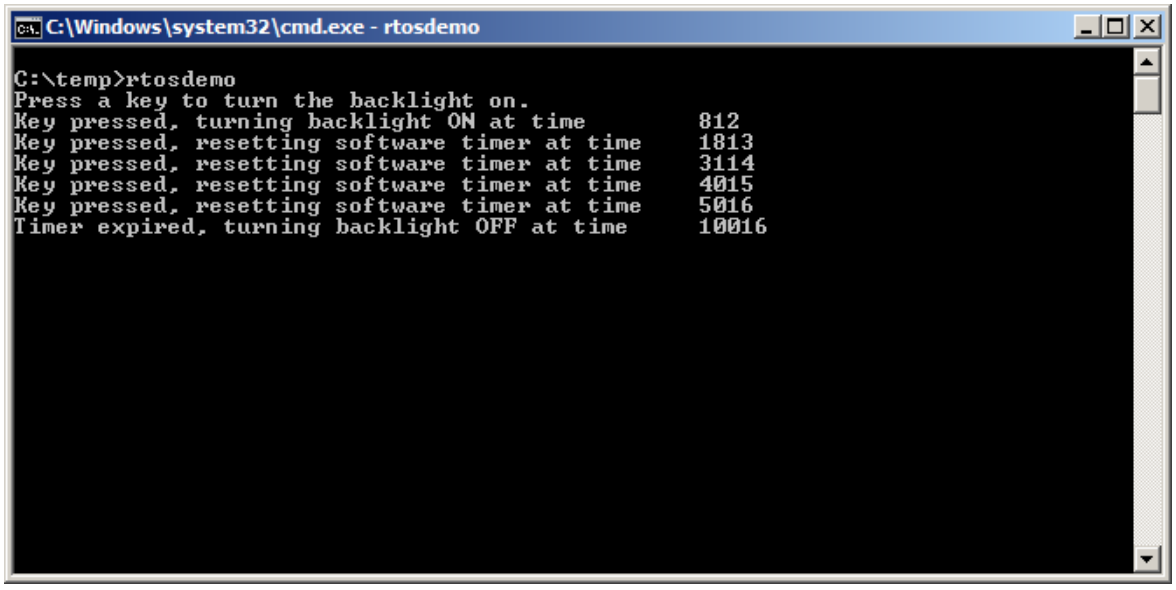
to process keyboard interrupts when using the FreeRTOS Windows port. The much less efficient polling technique had to be used instead.

The following code shows the task to poll the keyboard. If it were an interrupt service routine, then xTimerResetFromISR() would be used in place of xTimerReset().

```
static void vKeyHitTask( void *pvParameters )

{

    const TickType_t xShortDelay = pdMS_TO_TICKS( 50 );

    TickType_t xTimeNow;

    vPrintString( "Press a key to turn the backlight on.\r\n" );

    /* Ideally an application would be event-driven, and use an interrupt to process key
 presses. It is not practical to use keyboard interrupts when using the FreeRTOS Windows
 port, so this task is used to poll for a key press. */

    for( ;; )

    {

        /* Has a key been pressed? */

        if( _kbhit() != 0 )

        {

            /* A key has been pressed. Record the time. */

            xTimeNow = xTaskGetTickCount();

            if( xSimulatedBacklightOn == pdFALSE )

            {

                /* The backlight was off, so turn it on and print the time at which it was
 turned on. */

                xSimulatedBacklightOn = pdTRUE;

                vPrintStringAndNumber( "Key pressed, turning backlight ON at time\t\t",
 xTimeNow );

            }

            else

            {

                /* The backlight was already on, so print a message to say the timer is
 about to be reset and the time at which it was reset. */

                vPrintStringAndNumber("Key pressed, resetting software timer at time\t\t",
 xTimeNow );


            }

            /* Reset the software timer. If the backlight was previously off, then this
 call will start the timer. If the backlight was previously on, then this call will restart
 the timer. A real application might read key presses in an interrupt. If this function
```

```
 was an interrupt service routine, then xTimerResetFromISR() must be used instead of
 xTimerReset(). */

            xTimerReset( xBacklightTimer, xShortDelay );

            /* Read and discard the key that was pressed. It is not required by this simple
 example. */

            ( void ) _getch();

        }

    }

}
```

The output is shown here.



- The first key press occurred when the tick count was 812. At that time, the backlight was turned on, and the one-shot timer was started.
- Key presses occurred when the tick count was 1813, 3114, 4015, and 5016. All of these key presses resulted in the timer being reset before it had expired.
- The timer expired when the tick count was 10016. At that time, the backlight was turned off.

The timer had a period of 5000 ticks. The backlight was turned off exactly 5000 ticks after a key was last pressed, so 5000 ticks after the timer was last reset.

Interrupt Management

This section covers:

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

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. Consider these questions when choosing your 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 and how much outside? It is recommended to keep each interrupt service routine as short as possible.
3. How are events 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 event-processing strategy on you as the application designer, but it does provide features that allow your selected strategy to be implemented in a simple and maintainable way.

It is important to draw a distinction between the priority of a task and the priority of an interrupt:

- A task is a software feature that is unrelated to the hardware on which FreeRTOS is running. The priority of a task is assigned in software by the application writer. A software algorithm (the scheduler) decides which task will be in the Running state.
- Although written in software, an interrupt service routine is a hardware feature because the hardware controls which interrupt service routine will run and when. Tasks will run only when there are no interrupt service routines running, so the lowest priority interrupt will interrupt the highest priority task. There is no way for a task to preempt an interrupt service routine.

All architectures on which FreeRTOS will run are capable of processing interrupts, but details about interrupt entry and interrupt priority assignment vary between architectures.

The Interrupt-Safe API

Often it is necessary to use the functionality provided by a FreeRTOS API function from an interrupt service routine, but many FreeRTOS API functions perform actions that are not valid inside an interrupt service routine. The most notable of these is placing the task that called the API function into the Blocked state. If an API function is called from an interrupt service routine, then it is not being called from a task, so there is no calling task that can be placed into the Blocked state. FreeRTOS solves this problem by providing two versions of some API functions: one version for use from tasks and one version

for use from interrupt service routines. Functions intended for use from ISRs have "FromISR" appended to their name.

Note: Within an ISR, do not call a FreeRTOS API function that does not have "FromISR" in its name.

Advantages of Using a Separate Interrupt-Safe API

Having a separate API for use in interrupts allows task and code to be more efficient and interrupt entry to be simpler. Consider the alternative solution, which is to provide a single version of each API function that could be called from both a task and an ISR. If the same version of an API function could be called from both a task and an interrupt service routine, then:

- The API functions would need additional logic to determine if they had been called from a task or an ISR. The additional logic would introduce new paths through the function, making the functions longer, more complex, and harder to test.
- Some API function parameters would be obsolete when the function was called from a task, while others would be obsolete when the function was called from an ISR.
- Each FreeRTOS port would need to provide a mechanism for determining the execution context (task or ISR).
- Architectures on which it is not easy to determine the execution context (task or ISR) would require additional, wasteful, more complex, and nonstandard interrupt entry code that allowed the execution context to be provided by software.

Disadvantages of Using a Separate Interrupt-Safe API

Having two versions of some API functions allows both tasks and ISRs to be more efficient, but it introduces a new problem. Sometimes it is necessary to call a function that is not part of the FreeRTOS API, but makes use of the FreeRTOS API, from both a task and an ISR.

This is usually only a problem when integrating third-party code because that is the only time the software's design is out of the control of the application writer. You can use one of the following techniques:

1. Defer interrupt processing to a task, so the API function is only called from the context of a task.
2. If you are using a FreeRTOS port that supports interrupt nesting, then use the version of the API function that ends in "FromISR" because that version can be called from tasks and ISRs. (The reverse is not true. API functions that do not end in "FromISR" must not be called from an ISR).
3. Third-party code normally includes an RTOS abstraction layer that can be implemented to test the context from which the function is being called (task or interrupt), and then call the API function that is appropriate for the context.

xHigherPriorityTaskWoken Parameter

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

Some FreeRTOS API functions can move a task from the Blocked state to the Ready state. This has already been seen with functions such as xQueueSendToBack(), which will unblock a task if there was a task waiting in the Blocked state for data to become available on the subject queue.

If the priority of a task that is unblocked by a FreeRTOS API function is higher than the priority of the task in the Running state then, in accordance with the FreeRTOS scheduling policy, a switch to the higher priority task should occur. When the switch to the higher priority task actually occurs depends on the context from which the API function is called.

- If the API function was called from a task:

  If configUSE_PREEMPTION is set to 1 in FreeRTOSConfig.h, then the switch to the higher priority task occurs automatically within the API function, so before the API function has exited. This was shown in software_tier_management, where writing to the timer command queue resulted in a switch to the RTOS daemon task before the function that wrote to the command queue had exited.
- If the API function was called from an interrupt:

  A switch to a higher priority task will not occur automatically inside an interrupt. Instead, a variable is set to inform the application writer that a context switch should be performed. Interrupt-safe API functions (those that end in "FromISR") have a pointer parameter called pxHigherPriorityTaskWoken that is used for this purpose.

  If a context switch should be performed, then the interrupt-safe API function will set `*pxHigherPriorityTaskWoken` to pdTRUE. To be able to detect this has happened, the variable pointed to by pxHigherPriorityTaskWoken must be initialized to pdFALSE before it is used for the first time.

  If the application writer opts not to request a context switch from the ISR, then the higher priority task will remain in the Ready state until the next time the scheduler runs, which, in the worst case, will be during the next tick interrupt.

  FreeRTOS API functions can only set `*pxHighPriorityTaskWoken` to pdTRUE. If an ISR calls more than one FreeRTOS API function, then the same variable can be passed as the pxHigherPriorityTaskWoken parameter in each API function call. The variable must be initialized to pdFALSE only before it is used for the first time.

There are several reasons why context switches do not occur automatically inside the interrupt-safe version of an API function:

1. Avoiding unnecessary context switches

   An interrupt might execute more than once before it is necessary for a task to perform any processing. For example, consider a scenario where a task processes a string that was received by an interrupt-driven UART. It would be wasteful for the UART ISR to switch to the task each time a character was received because the task would only have processing to perform after the complete string had been received.
2. Control over the execution sequence

   Interrupts can occur sporadically and at unpredictable times. Expert FreeRTOS users might want to temporarily avoid an unpredictable switch to a different task at specific points in their application. You can do this by using the FreeRTOS scheduler locking mechanism.
3. Portability

   It is the simplest mechanism that can be used across all FreeRTOS ports.
4. Efficiency

   Ports that target smaller processor architectures only allow a context switch to be requested at the very end of an ISR. Removing that restriction would require additional and more complex code. It also allows more than one call to a FreeRTOS API function within the same ISR without generating more than one request for a context switch within the same ISR.
5. Execution in the RTOS tick interrupt

You can add application code into the RTOS tick interrupt. The result of attempting a context switch inside the tick interrupt depends on the FreeRTOS port in use. At best, it will result in an unnecessary call to the scheduler.

Use of the pxHigherPriorityTaskWoken parameter is optional. If it is not required, then set pxHigherPriorityTaskWoken to NULL.

portYIELD_FROM_ISR() and portEND_SWITCHING_ISR() Macros

taskYIELD() is a macro that can be called in a task to request a context switch. portYIELD_FROM_ISR() and portEND_SWITCHING_ISR() are both interrupt-safe versions of taskYIELD(). portYIELD_FROM_ISR() and portEND_SWITCHING_ISR() are used in the same way and do the same thing. Historically, portEND_SWITCHING_ISR() was the name used in FreeRTOS ports that required interrupt handlers to use an assembly code wrapper. portYIELD_FROM_ISR() was the name used in FreeRTOS ports that allowed the entire interrupt handler to be written in C. Some FreeRTOS ports only provide one of the two macros. Newer FreeRTOS ports provide both macros.

```
portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
```

```
portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
```

The xHigherPriorityTaskWoken parameter passed out of an interrupt-safe API function can be used directly as the parameter in a call to portYIELD_FROM_ISR().

If the portYIELD_FROM_ISR() xHigherPriorityTaskWoken parameter is pdFALSE (zero), then a context switch is not requested, and the macro has no effect. If the portYIELD_FROM_ISR() xHigherPriorityTaskWoken parameter is not pdFALSE, then a context switch is requested, and the task in the Running state might change. The interrupt will always return to the task in the Running state, even if the task in the Running state changed while the interrupt was executing.

Most FreeRTOS ports allow portYIELD_FROM_ISR() to be called anywhere within an ISR. A few FreeRTOS ports (mostly those for smaller architectures) allow portYIELD_FROM_ISR() to be called only at the very end of an ISR.

Deferred Interrupt Processing

It is a best practice to keep ISRs as short as possible because:

- Even if tasks have been assigned a very high priority, they will only run if no interrupts are being serviced by the hardware.
- ISRs can disrupt (add jitter to) the start and execution time of a task.
- Depending on the architecture on which FreeRTOS is running, it might not be possible to accept any new interrupts, or at least a subset of new interrupts, while an ISR is executing.
- The application writer must consider the consequences of and guard against resources such as variables, peripherals, and memory buffers being accessed by a task and an ISR at the same time.
- Some FreeRTOS ports allow interrupts to nest, but interrupt nesting can increase complexity and reduce predictability. The shorter an interrupt is, the less likely it is to nest.

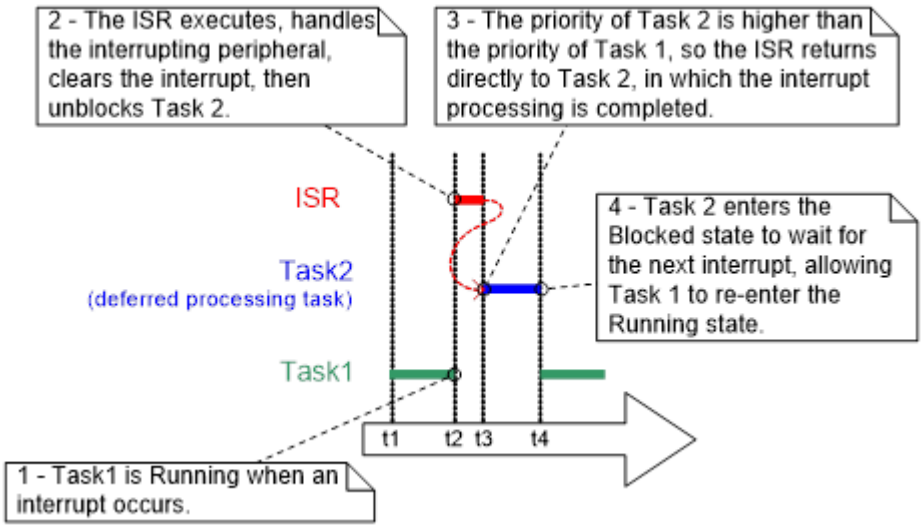
An interrupt service routine must record the cause of the interrupt and clear the interrupt. Any other processing required by the interrupt can often be performed in a task, allowing the interrupt service

routine to exit as quickly as possible. This is called *deferred interrupt processing* because the processing required by the interrupt is deferred from the ISR to a task.

Deferring interrupt processing to a task also allows the application writer to prioritize the processing relative to other tasks in the application and use all the FreeRTOS API functions.

If the priority of the task to which interrupt processing is deferred is above the priority of any other task, then the processing will be performed immediately, just as if the processing had been performed in the ISR itself.

The following figure shows a scenario in which Task 1 is a normal application task and Task 2 is the task to which interrupt processing is deferred.



In this figure, interrupt processing starts at time t2 and effectively ends at time t4, but only the period between times t2 and t3 is spent in the ISR. If deferred interrupt processing had not been used, then the entire period between times t2 and t4 would have been spent in the ISR.

There is no absolute rule as to when it is best to perform all processing required by an interrupt in the ISR and when it is best to defer part of the processing to a task. Deferring processing to a task is most useful when:

- The processing required by the interrupt is not trivial. For example, if the interrupt is just storing the result of an analog to digital conversion, then this is best performed inside the ISR, but if the result of the conversion must also be passed through a software filter, then it might be better to execute the filter in a task.
- It is convenient for the interrupt processing to perform an action that cannot be performed inside an ISR, such as write to a console or allocate memory.
- The interrupt processing is not deterministic (that is, it is not known in advance how long the processing will take).

The following sections describe and demonstrate FreeRTOS features that can be used to implement deferred interrupt processing.

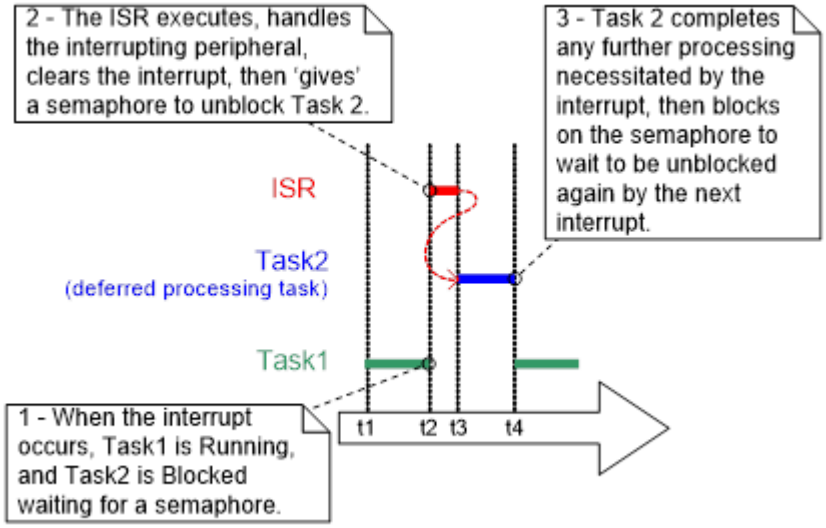
Binary Semaphores Used for Synchronization

You can use the interrupt-safe version of the Binary Semaphore API 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 binary semaphore is used to defer interrupt processing to a task. It is more efficient to use a direct-to-task notification to unblock a task from an interrupt than it is to use a binary semaphore. For more information about direct-to-task notifications, see Task Notifications (p. 195).

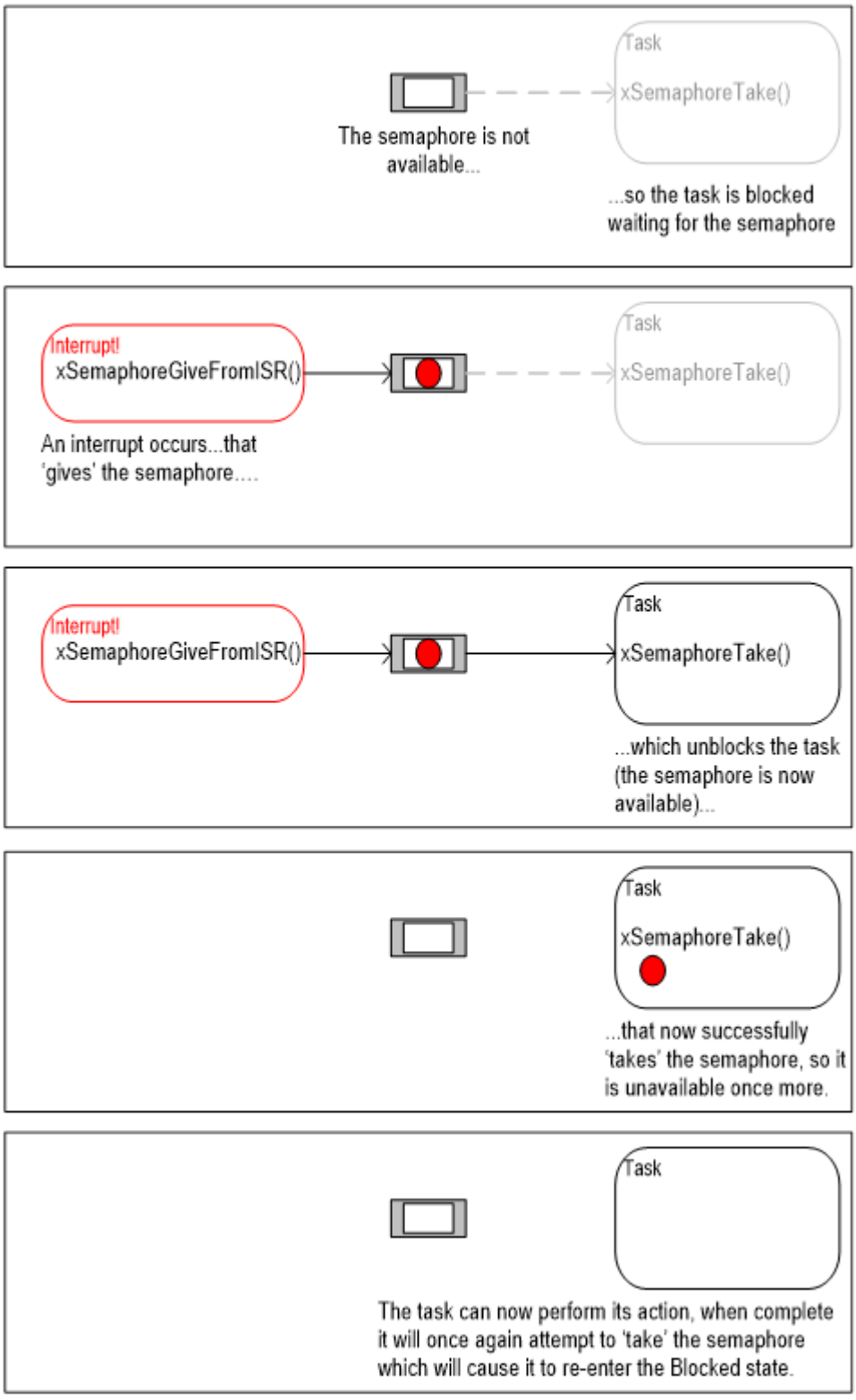
If the interrupt processing is particularly time-critical, then the priority of the deferred processing task can be set to ensure the task always preempts the other tasks in the system. The ISR can then be implemented to include a call to portYIELD_FROM_ISR(), ensuring the ISR returns directly to the task to which interrupt processing is being deferred. This has the effect of ensuring the entire event processing executes contiguously (without a break) in time, just as if it had all been implemented within the ISR itself.

The following figure uses the scenario shown in previous figure, but with the text updated to describe how the execution of the deferred processing task can be controlled by using a semaphore.



The deferred processing 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 the scenario. 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 (binary). By calling xSemaphoreTake(), the task to which interrupt processing is deferred 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 task to exit the Blocked state and remove the token, leaving the queue empty once more. When the task has completed its processing, it once more 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 shown here.



This figure 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 because 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 scenarios described in Resource Management (p. 155).

xSemaphoreCreateBinary() API Function

FreeRTOS V9.0.0 also includes the xSemaphoreCreateBinaryStatic() function, which allocates the memory required to create a binary semaphore statically at compile time. Handles to the various types of FreeRTOS semaphore are stored in a variable of type SemaphoreHandle_t.

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

```
SemaphoreHandle_t xSemaphoreCreateBinary( void );
```

The following table lists the xSemaphoreCreateBinary() return value.

Parameter Name	Description
Returned value	If NULL is returned, then the semaphore cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the semaphore data structures. If a non-NULL value is returned, the semaphore has been created successfully. The returned value should be stored as the handle to the created semaphore.

xSemaphoreTake() API Function

Taking a semaphore means to obtain or receive the semaphore. The semaphore can be taken only if it is available.

All types of FreeRTOS semaphore, except recursive mutexes, can be taken using the xSemaphoreTake() function. xSemaphoreTake() must not be used from an interrupt service routine.

```
BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait );
```

The following table lists the xSemaphoreTake() parameters and return value.

Parameter Name/ Returned Value	Description
xSemaphore	The semaphore being taken. A semaphore is referenced by a variable of type SemaphoreHandle_t. 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 depends on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds to ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without a timeout) if INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.
Returned value	There are two possible return values: 1. pdPASS pdPASS 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 into 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.

xSemaphoreGiveFromISR() API Function

Binary and counting semaphores can be given using the xSemaphoreGiveFromISR() function.

xSemaphoreGiveFromISR() is the interrupt-safe version of xSemaphoreGive(), so it has the pxHigherPriorityTaskWoken parameter.

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

The following lists the xSemaphoreGiveFromISR() parameters and return value.

xSemaphore

The semaphore being given. A semaphore is referenced by a variable of type SemaphoreHandle_t 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 a task that was waiting for the semaphore to leave the Blocked state. If calling xSemaphoreGiveFromISR() causes a task to leave the Blocked state, and the unblocked task has a priority higher than the currently executing task (the task that was interrupted), then, internally, xSemaphoreGiveFromISR() will set `*pxHigherPriorityTaskWoken` to pdTRUE. If xSemaphoreGiveFromISR() sets this value to pdTRUE, then normally 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.

There are two possible return values:

pdPASS

Returned only if the call to xSemaphoreGiveFromISR() is successful.

pdFAIL

If a semaphore is already available, it cannot be given, and xSemaphoreGiveFromISR() will return pdFAIL,

Using a Binary Semaphore to Synchronize a Task with an Interrupt (Example 16)

This example uses a binary semaphore to unblock a task from an interrupt service routine, effectively synchronizing the task with the interrupt.

A simple periodic task is used to generate a software interrupt every 500 milliseconds. A software interrupt is used for convenience because of the complexity of hooking into a real interrupt in some target environments.

The following code shows the implementation of the periodic task. The task prints out a string before and after the interrupt is generated. You can see the sequence of the execution in the output.

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

#define mainINTERRUPT_NUMBER 3

static void vPeriodicTask( void *pvParameters )

{

    const TickType_t xDelay500ms = pdMS_TO_TICKS( 500UL );

    /* As per most tasks, this task is implemented within an infinite loop.*/

    for( ;; )

    {

        /* Block until it is time to generate the software interrupt again. */

        vTaskDelay( xDelay500ms );

        /* Generate the interrupt, printing a message both before and after the interrupt
 has been generated, so the sequence of execution is evident from the output. The syntax
 used to generate a software interrupt is dependent on the FreeRTOS port being used.
```

```
 The syntax used below can only be used with the FreeRTOS Windows port, in which such
 interrupts are only simulated.*/

        vPrintString( "Periodic task - About to generate an interrupt.\r\n" );

        vPortGenerateSimulatedInterrupt( mainINTERRUPT_NUMBER );

        vPrintString( "Periodic task - Interrupt generated.\r\n\r\n\r\n"
    );

}

}
```

The following code shows the implementation of the task to which the interrupt processing is deferred, the task that is synchronized with the software interrupt through the use of a binary semaphore. Again, a string is printed out on each iteration of the task. You'll see the sequence in which the task and the interrupt execute in the output.

Although this code is adequate as a sample where interrupts are generated by software, it is not adequate for scenarios where interrupts are generated by hardware peripherals. The structure of the code must be changed to make it suitable for use with hardware-generated interrupts.

```
static void vHandlerTask( void *pvParameters )

{

    /* As per most tasks, this task is implemented within an infinite loop.*/

    for( ;; )

    {

        /* Use the semaphore to wait for the event. The semaphore was created before
 the scheduler was started, so before this task ran for the first time. The task blocks
 indefinitely, meaning this function call will only return once the semaphore has been
 successfully obtained so there is no need to check the value returned by xSemaphoreTake().
 */

        xSemaphoreTake( xBinarySemaphore, portMAX_DELAY );

        /* To get here the event must have occurred. Process the event (in this case, just
 print out a message). */

        vPrintString( "Handler task - Processing event.\r\n" );

    }

}
```

The following code shows the ISR. This does very little other than give the semaphore to unblock the task to which interrupt processing is deferred.

The xHigherPriorityTaskWoken variable is used. It is set to pdFALSE before calling xSemaphoreGiveFromISR(), and then used as the parameter when portYIELD_FROM_ISR() is called. A context switch will be requested inside the portYIELD_FROM_ISR() macro if xHigherPriorityTaskWoken equals pdTRUE.

The prototype of the ISR and the macro called to force a context switch are correct for the FreeRTOS Windows port, but might be different for other FreeRTOS ports. To find the syntax required for the port you are using, see the port-specific documentation pages on the FreeRTOS.org website and the examples provided in the FreeRTOS download.

Unlike most architectures on which FreeRTOS runs, the FreeRTOS Windows port requires an ISR to return a value. The implementation of the portYIELD_FROM_ISR() macro provided with the Windows port includes the return statement, so this code does not show a value being returned explicitly.

```
static uint32_t ulExampleInterruptHandler( void )

{

    BaseType_t xHigherPriorityTaskWoken;

    /* The xHigherPriorityTaskWoken parameter must be initialized to pdFALSE because it
 will get set to pdTRUE inside the interrupt-safe API function if a context switch is
 required. */

    xHigherPriorityTaskWoken = pdFALSE;

    /* Give the semaphore to unblock the task, passing in the address of
 xHigherPriorityTaskWoken as the interrupt-safe API function's pxHigherPriorityTaskWoken
 parameter. */

    xSemaphoreGiveFromISR( xBinarySemaphore, &xHigherPriorityTaskWoken );

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

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

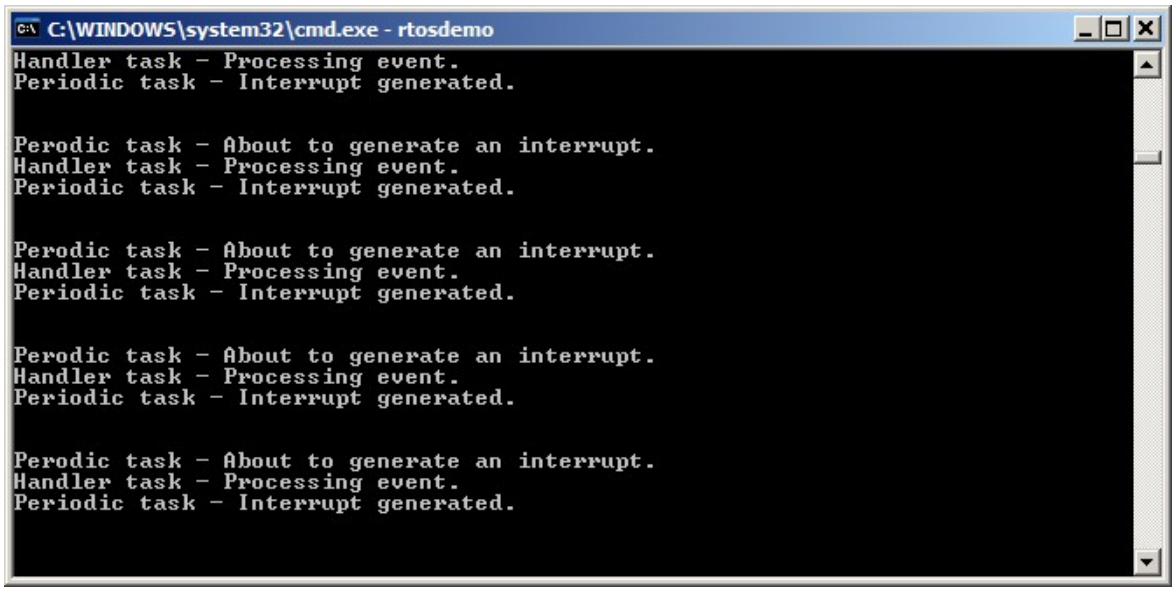
The main() function creates the binary semaphore and tasks, installs the interrupt handler, and starts the scheduler. The following codes shows the implementation.

The syntax of the function called to install an interrupt handler is specific to the FreeRTOS Windows port. It might be different for other FreeRTOS ports. To find the syntax required for the port you are using, see the port-specific documentation on the FreeRTOS.org website and the examples provided in the FreeRTOS download.

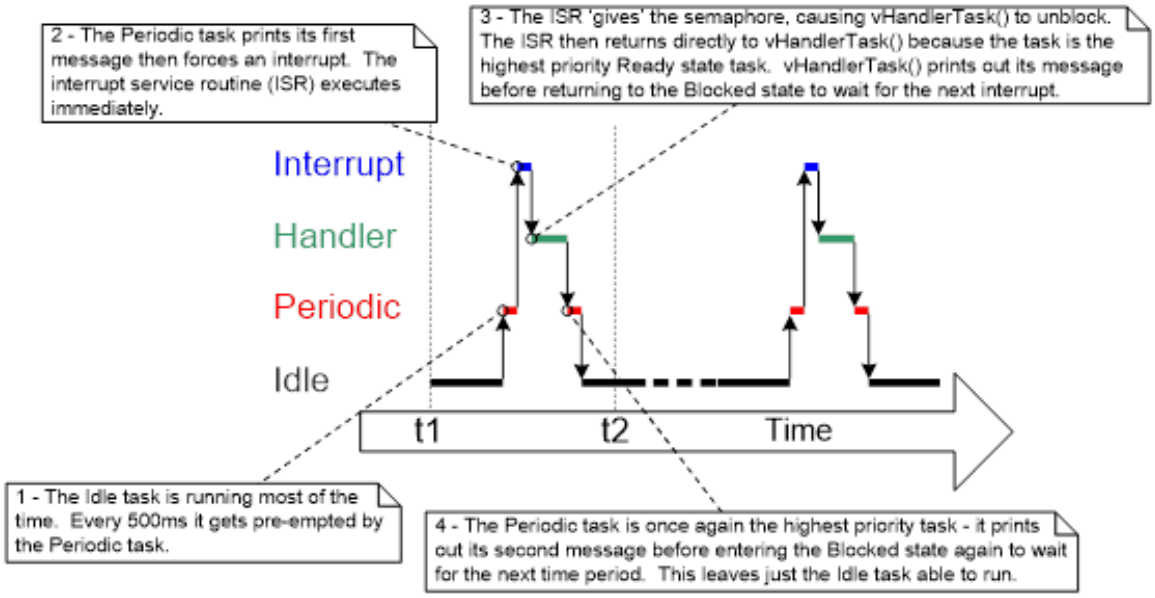
```
int main( void )

{

    /* Before a semaphore is used, it must be explicitly created. In this example, a binary
 semaphore is created. */

    xBinarySemaphore = xSemaphoreCreateBinary();

    /* Check that the semaphore was created successfully. */

    if( xBinarySemaphore != NULL )

    {

        /* Create the 'handler' task, which is the task to which interrupt processing is
 deferred. This is the task that will be synchronized with the interrupt. The handler task
 is created with a high priority to ensure it runs immediately after the interrupt exits.
 In this case, a priority of 3 is chosen. */

        xTaskCreate( vHandlerTask, "Handler", 1000, NULL, 3, NULL );
```

```
        /* Create the task that will periodically generate a software interrupt. This is
created with a priority below the handler task to ensure it will get preempted each time
the handler task exits the Blocked state. */

        xTaskCreate( vPeriodicTask, "Periodic", 1000, NULL, 1, NULL );

        /* Install the handler for the software interrupt. The syntax required to do this
is depends on the FreeRTOS port being used. The syntax shown here can only be used with
the FreeRTOS Windows port, where such interrupts are only simulated. */

        vPortSetInterruptHandler( mainINTERRUPT_NUMBER, ulExampleInterruptHandler );

        /* Start the scheduler so the created tasks start executing. */

        vTaskStartScheduler();

        }

        /* As normal, the following line should never be reached. */

        for( ;; );

    }
```

The code produces the following output. As expected, vHandlerTask() enters the Running state as soon as the interrupt is generated, so the output from the task splits the output produced by the periodic task.



Here is the sequence of execution.



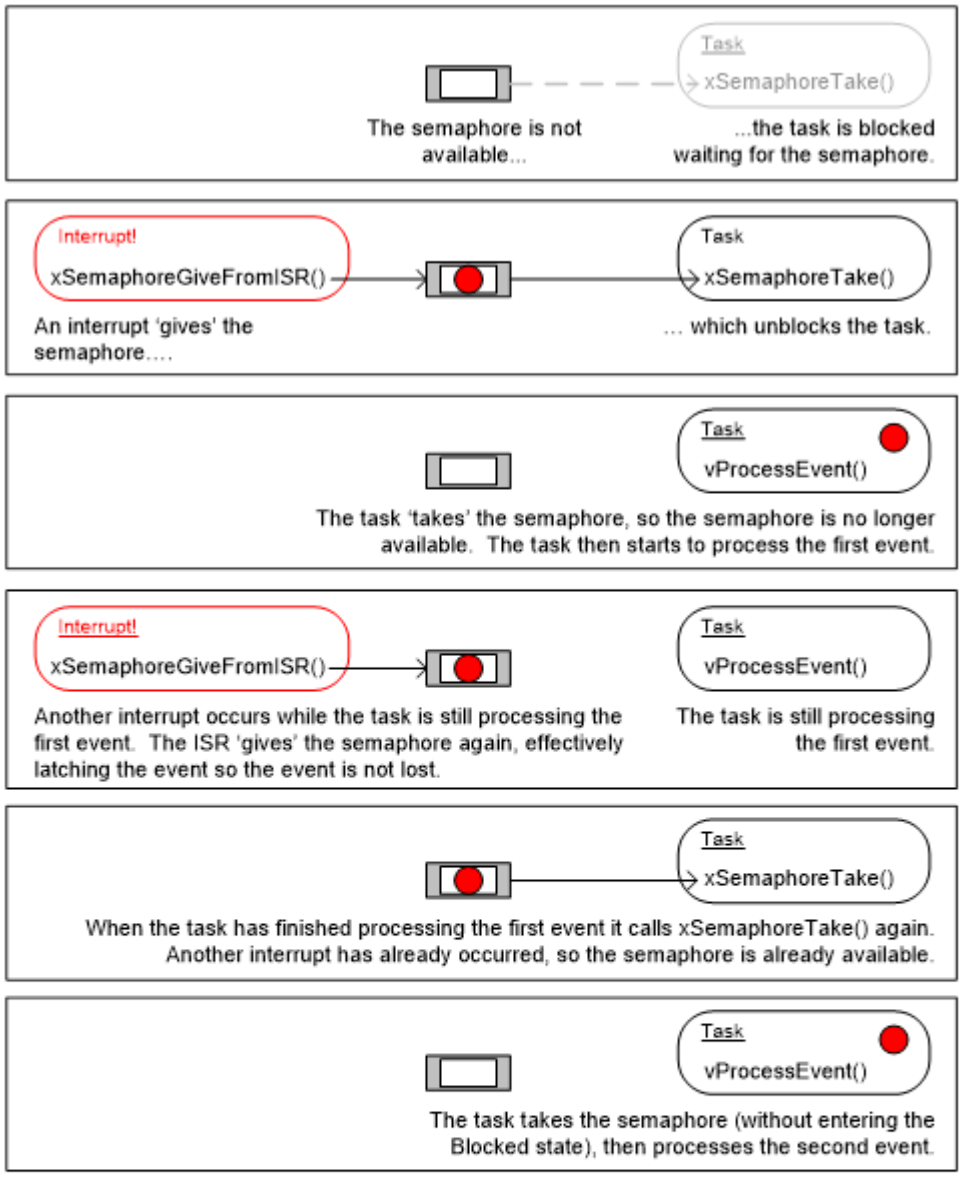
Improving the Implementation of the Task Used in Example 16

Example 16 uses a binary semaphore to synchronize a task with an interrupt. Here is the execution sequence:

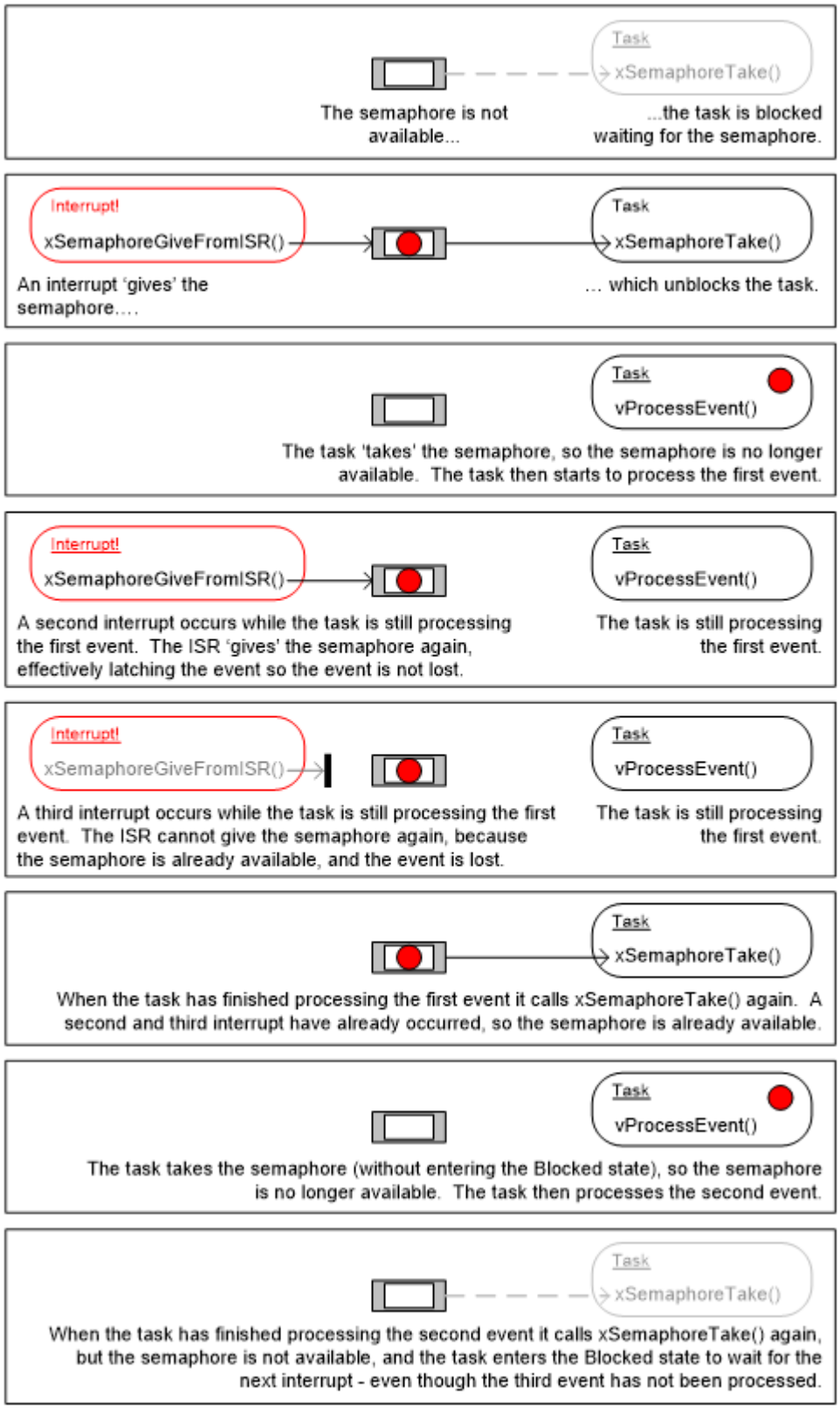
1. The interrupt occurred.
2. The ISR executed and gave the semaphore to unblock the task.
3. The task executed immediately after the ISR and took the semaphore.
4. The task processed the event, and then attempted to take the semaphore again, entering the Blocked state because the semaphore was not yet available (another interrupt had not yet occurred).

The structure of the task used in Example 16 is adequate only if interrupts occur at a relatively low frequency. Consider what would happen if a second and then a third interrupt had occurred before the task had completed its processing of the first interrupt.

When the second ISR executed, the semaphore would be empty, so the ISR would give the semaphore, and the task would process the second event immediately after it had completed processing the first event. That scenario is shown here.



When the third ISR executed, the semaphore would already be available, preventing the ISR from giving the semaphore again, so the task would not know the third event had occurred. That scenario is shown here.


Task
xSemaphoreTake()
The semaphore is not available...
...the task is blocked waiting for the semaphore.
Interrupt!
xSemaphoreGiveFromISR()
Task
xSemaphoreTake()
An interrupt 'gives' the semaphore....
... which unblocks the task.
Task
vProcessEvent()
The task 'takes' the semaphore, so the semaphore is no longer available. The task then starts to process the first event.
Interrupt!
xSemaphoreGiveFromISR()
Task
vProcessEvent()
A second interrupt occurs while the task is still processing the first event. The ISR 'gives' the semaphore again, effectively latching the event so the event is not lost.
The task is still processing the first event.
Interrupt!
xSemaphoreGiveFromISR()
Task
vProcessEvent()
A third interrupt occurs while the task is still processing the first event. The ISR cannot give the semaphore again, because the semaphore is already available, and the event is lost.
The task is still processing the first event.
Task
xSemaphoreTake()
When the task has finished processing the first event it calls xSemaphoreTake() again. A second and third interrupt have already occurred, so the semaphore is already available.
Task
vProcessEvent()
The task takes the semaphore (without entering the Blocked state), so the semaphore is no longer available. The task then processes the second event.
Task
xSemaphoreTake()
When the task has finished processing the second event it calls xSemaphoreTake() again, but the semaphore is not available, and the task enters the Blocked state to wait for the next interrupt - even though the third event has not been processed.

The deferred interrupt handling task, `static void vHandlerTask( void *pvParameters )`, is structured so that it only processes one event between each call to xSemaphoreTake(). That was adequate for Example 16 because the interrupts that generated the events were triggered by software and occurred at a predictable time. In real applications, interrupts are generated by hardware and occur at unpredictable times. Therefore, to minimize the chance of an interrupt being missed, the deferred interrupt handling task must be structured so that it processes all the events that are already available between each call to xSemaphoreTake(). The following code shows how a deferred interrupt handler for a UART could be structured. It assumes the UART generates a receive interrupt each time a character is received, and that the UART places received characters into a hardware FIFO (a hardware buffer).

The deferred interrupt handling task used in Example 16 had one other weakness. It did not use a timeout when it called xSemaphoreTake(). Instead, the task passed portMAX_DELAY as the xSemaphoreTake() xTicksToWait parameter, which results in the task waiting indefinitely (without a timeout) for the semaphore to be available. Indefinite timeouts are often used in example code because their use simplifies the structure of the example and makes it easier to understand. In real applications, however, indefinite timeouts are usually a bad practice because they make it difficult to recover from an error. Consider the scenario where a task is waiting for an interrupt to give a semaphore, but an error state in the hardware is preventing the interrupt from being generated.

- If the task is waiting without a timeout, it will not know about the error state and will wait forever.
- If the task is waiting with a timeout, then xSemaphoreTake() will return pdFAIL when the timeout expires and the task can then detect and clear the error the next time it executes. This scenario is also demonstrated here.

```
static void vUARTReceiveHandlerTask( void *pvParameters )

{

    /* xMaxExpectedBlockTime holds the maximum time expected between two interrupts. */

    const TickType_t xMaxExpectedBlockTime = pdMS_TO_TICKS( 500 );

    /* As per most tasks, this task is implemented within an infinite loop. */

    for( ;; )

    {

        /* The semaphore is given by the UART's receive (Rx) interrupt. Wait a maximum of
 xMaxExpectedBlockTime ticks for the next interrupt. */

        if( xSemaphoreTake( xBinarySemaphore, xMaxExpectedBlockTime ) == pdPASS)

        {

            /* The semaphore was obtained. Process ALL pending Rx events before calling
 xSemaphoreTake() again. Each Rx event will have placed a character in the UART's receive
 FIFO, and UART_RxCount() is assumed to return the number of characters in the FIFO. */

            while( UART_RxCount() > 0 )

            {

                /* UART_ProcessNextRxEvent() is assumed to process one Rx character,
 reducing the number of characters in the FIFO by 1. */

                UART_ProcessNextRxEvent();

            }
```

```
            /* No more Rx events are pending (there are no more characters in the FIFO),
 so loop back and call xSemaphoreTake() to wait for the next interrupt. Any interrupts
 occurring between this point in the code and the call to xSemaphoreTake() will be latched
 in the semaphore, so will not be lost. */

        }

        else

        {

            /* An event was not received within the expected time. Check for and, if
 necessary, clear any error conditions in the UART that might be preventing the UART from
 generating any more interrupts. */

            UART_ClearErrors();

        }

    }

}
```

Counting Semaphores

Just as binary semaphores can be thought of as queues that have a length of one, counting semaphores can be thought of as queues that have a length of more than one. Tasks are not interested in the data that is stored in the queue, just the number of items in the queue. To make counting semaphores available, in FreeRTOSConfig.h, set configUSE_COUNTING_SEMAPHORES to 1.

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.

Counting semaphores are typically used for:

1. Counting events

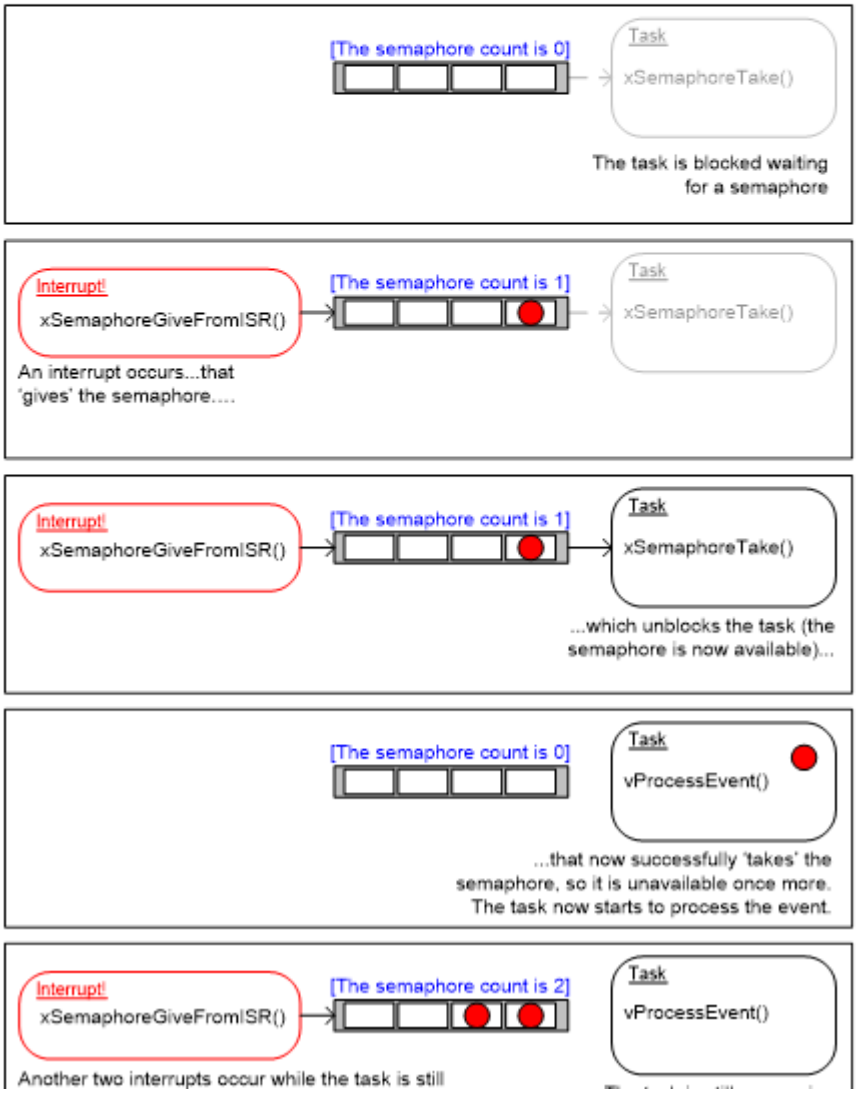
   In this scenario, an event handler will give a semaphore each time an event occurs, causing the semaphore's count value to be incremented on each give. A task will take a semaphore each time it processes an event, causing the semaphore's count value to be decremented on each take. The count value is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

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

2. Resource management

   In this scenario, the count value indicates the number of resources available. To obtain control of a resource, a task must first obtain a semaphore, decrementing the semaphore's count value. When the count value reaches zero, there are no free resources. When a task finishes with the resource, it gives the semaphore back, incrementing the semaphore's count value.

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


[The semaphore count is 0]
Task
xSemaphoreTake()
The task is blocked waiting for a semaphore
Interrupt!
xSemaphoreGiveFromISR()
[The semaphore count is 1]
Task
xSemaphoreTake()
An interrupt occurs...that 'gives' the semaphore....
Interrupt!
xSemaphoreGiveFromISR()
[The semaphore count is 1]
Task
xSemaphoreTake()
...which unblocks the task (the semaphore is now available)...
[The semaphore count is 0]
Task
vProcessEvent()
...that now successfully 'takes' the semaphore, so it is unavailable once more. The task now starts to process the event.
Interrupt!
xSemaphoreGiveFromISR()
[The semaphore count is 2]
Task
vProcessEvent()
Another two interrupts occur while the task is still

xSemaphoreCreateCounting() API Function

FreeRTOS V9.0.0 also includes the xSemaphoreCreateCountingStatic() function, which allocates the memory required to create a counting semaphore statically at compile time. Handles to all the various types of FreeRTOS semaphore are stored in a variable of type SemaphoreHandle_t.

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

The following table lists the xSemaphoreCreateCounting() parameters and return value.

Parameter Name/ Returned Value	Description
uxMaxCount	The maximum value to which the semaphore will count. To continue the queue analogy, the uxMaxCount value is effectively the length of the queue. When the semaphore is to be used to count or latch events, uxMaxCount is the maximum number of events that can be latched. When the semaphore is to be used to manage access to a collection of resources, uxMaxCount should be set to the total number of resources that are available.
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 because 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 because 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. For more information, see Heap Memory Management (p. 14). If a non-NULL value is returned, the semaphore has been created successfully. The returned value should be stored as the handle to the created semaphore.

Using a Counting Semaphore to Synchronize a Task with an Interrupt (Example 17)

Example 17 improves on the Example 16 implementation by using a counting semaphore in place of the binary semaphore. main() is changed to include a call to xSemaphoreCreateCounting() in place of the call to xSemaphoreCreateBinary().

The following codes shows the new API call.

```
/* 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 );
```

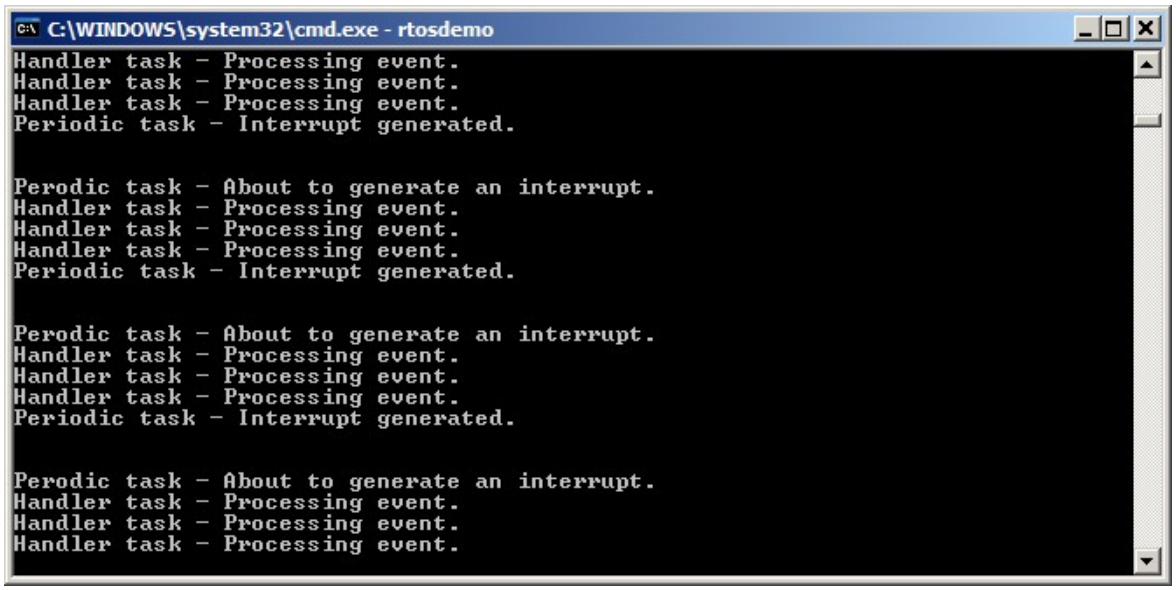
To simulate multiple events occurring at high frequency, the interrupt service routine is changed to give the semaphore more than once per interrupt. Each event is latched in the semaphore's count value.

The following code shows the modified interrupt service routine.

```
static uint32_t ulExampleInterruptHandler( void )

{

    BaseType_t xHigherPriorityTaskWoken;

    /* The xHigherPriorityTaskWoken parameter must be initialized to pdFALSE because it
 will get set to pdTRUE inside the interrupt-safe API function if a context switch is
 required. */

    xHigherPriorityTaskWoken = pdFALSE;

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

    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );

    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );

    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );

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

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

All the other functions are the same as those used in Example 16.

Here is the output when the code is executed. You'll see the task to which interrupt handling is deferred processes all three (simulated) events each time an interrupt is generated. The events are latched into the count value of the semaphore, allowing the task to process them in turn.



Deferring Work to the RTOS Daemon Task

The deferred interrupt handling examples presented so far have required the application writer to create a task for each interrupt that uses the deferred processing technique. It is also possible to use the xTimerPendFunctionCallFromISR() API function to defer interrupt processing to the RTOS daemon task, removing the need to create a separate task for each interrupt. Deferring interrupt processing to the daemon task is called *centralized deferred interrupt processing*.

The daemon task was originally called the timer service task because it was only used to execute software timer callback functions. For this reason, xTimerPendFunctionCall() is implemented in timers.c, and, in accordance with the convention of prefixing a function's name with the name of the file in which the function is implemented, the function's name is prefixed with 'Timer'.

The software_tier_management section described how FreeRTOS API functions related to software timers send commands to the daemon task on the timer command queue. The xTimerPendFunctionCall() and xTimerPendFunctionCallFromISR() API functions use the same timer command queue to send an 'execute function' command to the daemon task. The function sent to the daemon task is then executed in the context of the daemon task.

Advantages of centralized deferred interrupt processing:

- Lower resource usage

  There is no need to create a separate task for each deferred interrupt.
- Simplified user model

  The deferred interrupt handling function is a standard C function.

Disadvantages of centralized deferred interrupt processing:

- Less flexibility

  You cannot set the priority of each deferred interrupt handling task separately. Each deferred interrupt handling function executes at the priority of the daemon task. The priority of the daemon task is set by the configTIMER_TASK_PRIORITY compile time configuration constant in FreeRTOSConfig.h.
- Less determinism

xTimerPendFunctionCallFromISR() sends a command to the back of the timer command queue. Commands that were already in the timer command queue will be processed by the daemon task before the 'execute function' command sent to the queue by xTimerPendFunctionCallFromISR().

Different interrupts have different timing constraints, so it is common to use both methods in the same application.

xTimerPendFunctionCallFromISR() API Function

xTimerPendFunctionCallFromISR() is the interrupt-safe version of xTimerPendFunctionCall(). Both API functions allow a function provided by the application writer to be executed by, and therefore in the context of, the RTOS daemon task. The function to be executed and the value of its input parameters are sent to the daemon task on the timer command queue. When the function executes depends on the priority of the daemon task relative to other tasks in the application.

The xTimerPendFunctionCallFromISR() API function prototype is shown here.

```
BaseType_t xTimerPendFunctionCallFromISR( PendedFunction_t xFunctionToPend, void
 *pvParameter1, uint32_t ulParameter2, BaseType_t *pxHigherPriorityTaskWoken );
```

The prototype to which a function passed in the xFunctionToPend parameter of xTimerPendFunctionCallFromISR() must conform is shown here.

```
void vPendableFunction( void *pvParameter1, uint32_t ulParameter2 );
```

The following lists the xTimerPendFunctionCallFromISR() parameters and return value.

xFunctionToPend

A pointer to the function that will be executed in the daemon task (in effect, just the function name). The prototype of the function must be the same as that shown in the code.

pvParameter1

The value that will be passed into the function that is executed by the daemon task as the function's pvParameter1 parameter. The parameter has a void * type to allow it to be used to pass any data type. For example, integer types can be directly cast to a void *. Alternatively, the void * can be used to point to a structure.

ulParameter2

The value that will be passed into the function that is executed by the daemon task as the function's ulParameter2 parameter.

pxHigherPriorityTaskWoken

xTimerPendFunctionCallFromISR() writes to the timer command queue. If the RTOS daemon task was in the Blocked state to wait for data to become available on the timer command queue, then writing to the timer command queue will cause the daemon task to leave the Blocked state. If the priority of the daemon task is higher than the priority of the currently executing task (the task that was interrupted), then, internally, xTimerPendFunctionCallFromISR() will set `*pxHigherPriorityTaskWoken` to pdTRUE. If xTimerPendFunctionCallFromISR() sets this value to pdTRUE, then a context switch must be performed

before the interrupt is exited. This will ensure that the interrupt returns directly to the daemon task because the daemon task will be the highest priority Ready state task.

There are two possible return values:

- pdPASS

Returned if the 'execute function' command was written to the timer command.

- pdFAIL

Returned if the 'execute function' command could not be written to the timer command queue because the timer command queue was already full. For information about how to set the length of the timer command, see software_tier_management.

Centralized Deferred Interrupt Processing (Example 18)

Example 18 provides functionality similar to Example 16, but without using a semaphore and creating a task to perform the processing required by the interrupt. Instead, the processing is performed by the RTOS daemon task.

The interrupt service routine used by Example 18 is shown in the following code. It calls xTimerPendFunctionCallFromISR() to pass a pointer to a function called vDeferredHandlingFunction() to the daemon task. The deferred interrupt processing is performed by the vDeferredHandlingFunction() function.

The interrupt service routine increments a variable called ulParameterValue each time it executes. ulParameterValue is used as the value of ulParameter2 in the call to xTimerPendFunctionCallFromISR(), so will also be used as the value of ulParameter2 in the call to vDeferredHandlingFunction() when vDeferredHandlingFunction() is executed by the daemon task. The function's other parameter, pvParameter1, is not used in this example.

```
    static uint32_t ulExampleInterruptHandler( void )

{

    static uint32_t ulParameterValue = 0;

    BaseType_t xHigherPriorityTaskWoken;

    /* The xHigherPriorityTaskWoken parameter must be initialized to pdFALSE because it
 will get set to pdTRUE inside the interrupt-safe API function if a context switch is
 required. */

    xHigherPriorityTaskWoken = pdFALSE;

    /* Send a pointer to the interrupt's deferred handling function to the daemon task.
 The deferred handling function's pvParameter1 parameter is not used, so just set to NULL.
 The deferred handling function's ulParameter2 parameter is used to pass a number that is
 incremented by one each time this interrupt handler executes. */

    xTimerPendFunctionCallFromISR( vDeferredHandlingFunction, /* Function to
 execute. */ NULL, /* Not used. */ ulParameterValue, /* Incrementing value. */
 &xHigherPriorityTaskWoken );

    ulParameterValue++;
```

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

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

The implementation of vDeferredHandlingFunction() is shown here. It prints out a fixed string and the value of its ulParameter2 parameter.

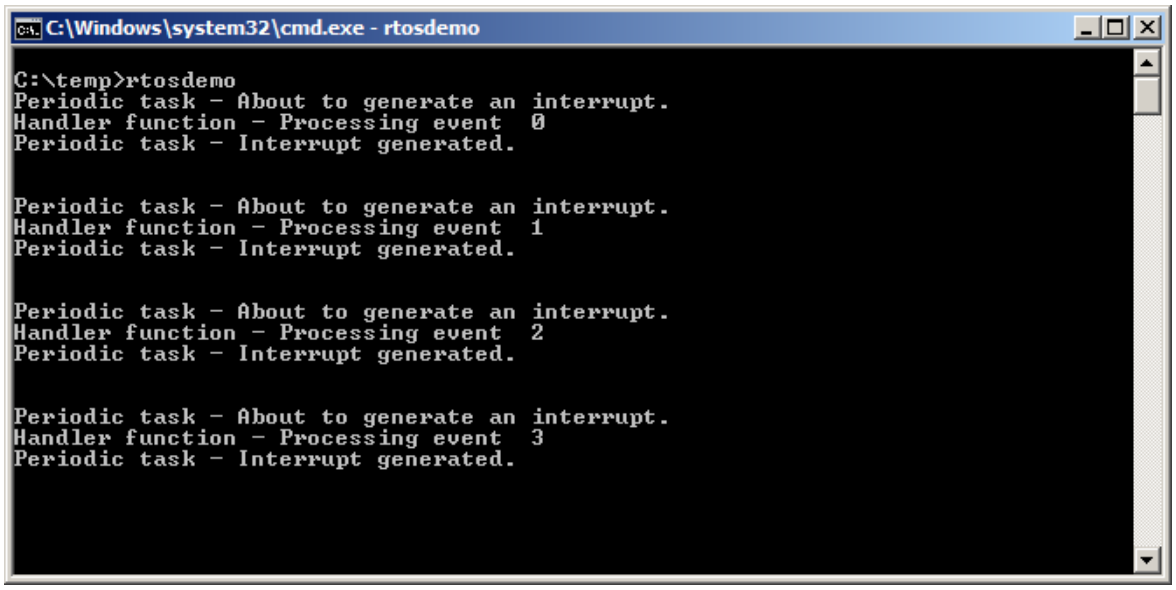
```
static void vDeferredHandlingFunction( void *pvParameter1, uint32_t ulParameter2 )

{

    /* Process the event - in this case, just print out a message and the value of
 ulParameter2. pvParameter1 is not used in this example. */

    vPrintStringAndNumber( "Handler function - Processing event ", ulParameter2 );

}
```

The main() function used is shown here. vPeriodicTask() is the task that periodically generates software interrupts. It is created with a priority below the priority of the daemon task to ensure it is preempted by the daemon task as soon as the daemon task leaves the Blocked state.

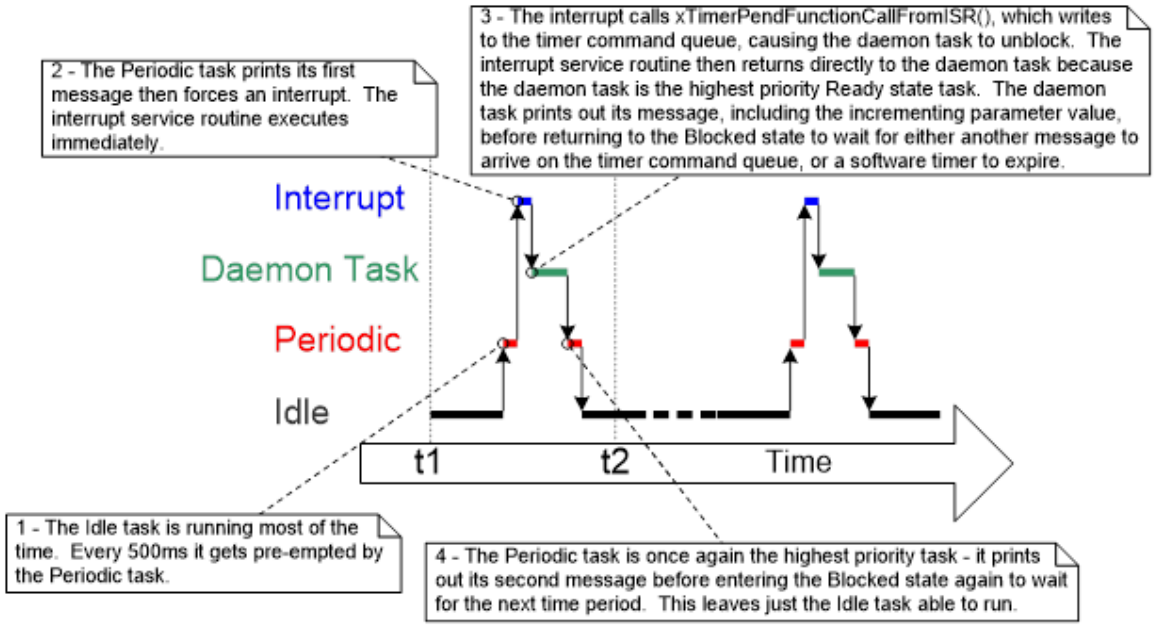
```
int main( void )

{

    /* The task that generates the software interrupt is created at a priority below
 the priority of the daemon task. The priority of the daemon task is set by the
 configTIMER_TASK_PRIORITY compile time configuration constant in FreeRTOSConfig.h. */

    const UBaseType_t ulPeriodicTaskPriority = configTIMER_TASK_PRIORITY - 1;

    /* Create the task that will periodically generate a software interrupt. */

    xTaskCreate( vPeriodicTask, "Periodic", 1000, NULL, ulPeriodicTaskPriority, NULL );

    /* Install the handler for the software interrupt. The syntax necessary to do this is
 dependent on the FreeRTOS port being used. The syntax shown here can only be used with the
 FreeRTOS windows port, where such interrupts are only simulated. */

    vPortSetInterruptHandler( mainINTERRUPT_NUMBER, ulExampleInterruptHandler );

    /* Start the scheduler so the created task starts executing. */

    vTaskStartScheduler();

    /* As normal, the following line should never be reached. */

    for( ;; );

}
```

The code produces the output shown here. The priority of the daemon task is higher than the priority of the task that generates the software interrupt, so vDeferredHandlingFunction() is executed

by the daemon task as soon as the interrupt is generated. As a result, the message output by vDeferredHandlingFunction() appears between the two messages output by the periodic task, just as it did when a semaphore was used to unblock a dedicated deferred interrupt processing task.



Here is the sequence of execution.



Using Queues Within an Interrupt Service Routine

Binary and counting semaphores are used to communicate events. Queues are used to communicate events and transfer data.

xQueueSendToFrontFromISR() is the version of xQueueSendToFront() that is safe to use in an interrupt service routine. xQueueSendToBackFromISR() is the version of xQueueSendToBack() that is safe to use in

an interrupt service routine. xQueueReceiveFromISR() is the version of xQueueReceive() that is safe to use in an interrupt service routine.

xQueueSendToFrontFromISR() and xQueueSendToBackFromISR() API Functions

The xQueueSendToFrontFromISR() API function prototype is shown here.

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

The xQueueSendToBackFromISR() API function prototype is shown here.

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

xQueueSendFromISR() and xQueueSendToBackFromISR() are functionally equivalent.

The following lists the xQueueSendToFrontFromISR() and xQueueSendToBackFromISR() parameters and return values.

xQueue

The handle of the queue to which the data is being sent (written). The queue handle will have been returned from the call to xQueueCreate() used to create the queue.

pvItemToQueue

A pointer to the data that will be copied into the queue. The size of each item the queue can hold is set when the queue is created, so this many bytes will be copied from pvItemToQueue into the queue storage area.

pxHigherPriorityTaskWoken

It is possible that a single queue will have one or more tasks blocked on it, waiting for data to become available. Calling xQueueSendToFrontFromISR() or xQueueSendToBackFromISR() can make data available, and so cause such a task to leave the Blocked state. If calling the API function causes a task to leave the Blocked state, and the unblocked task has a priority higher than the currently executing task (the task that was interrupted), then, 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.

There are two possible return values:

- pdPASS

Returned only if data has been sent successfully to the queue.

- errQUEUE_FULL

Returned if data cannot be sent to the queue because the queue is already full.

Considerations When Using a Queue from an ISR

Queues provide an easy and convenient way to pass data from an interrupt to a task. It is not efficient to use a queue if data is arriving at a high frequency.

Many of the demo applications in the FreeRTOS download include a simple UART driver that uses a queue to pass characters out of the UART's receive ISR. In those demos, a queue is to demonstrate the use of queues from an ISR and to deliberately load the system in order to test the FreeRTOS port. The ISRs that use a queue in this manner do not represent an efficient design, and unless the data is arriving slowly, we do not recommend you use this technique in production code. More efficient techniques include:

- Using Direct Memory Access (DMA) hardware to receive and buffer characters. This method has practically no software overhead. A direct-to-task notification can then be used to unblock the task that will process the buffer only after a break in transmission has been detected. Direct-to-task notifications provide the most efficient method for unblocking a task from an ISR. For more information, see Task Notifications (p. 195).
- Copying each received character into a thread-safe RAM buffer. The 'Stream Buffer' that is provided as part of FreeRTOS+TCP can be used for this purpose. Again, a direct-to-task notification can be used to unblock the task that will process the buffer after a complete message has been received or after a break in transmission has been detected.
- Processing the received characters directly within the ISR, and then using a queue to send just the result of processing the data (rather than the raw data) to a task.

Sending and Receiving on a Queue from Within an Interrupt (Example 19)

This example demonstrates the use of xQueueSendToBackFromISR() and xQueueReceiveFromISR() within the same interrupt. For convenience, the interrupt is generated by software.

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

```
    static void vIntegerGenerator( void *pvParameters )

{

    TickType_t xLastExecutionTime;

    uint32_t 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 200 ms. */

        vTaskDelayUntil( &xLastExecutionTime, pdMS_TO_TICKS( 200 ) );
```

```
        /* Send five numbers to the queue, each value one higher than the previous value.
 The numbers are 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 without needing to specify a block time. */

        for( i = 0; i < 5; i++ )

        {

            xQueueSendToBack( xIntegerQueue, &ulValueToSend, 0 );

            ulValueToSend++;

        }

        /* Generate the interrupt so the interrupt service routine can read the values from
 the queue. The syntax used to generate a software interrupt depends on the FreeRTOS port
 being used. The syntax used below can only be used with the FreeRTOS Windows port, in
 which such interrupts are only simulated.*/

        vPrintString( "Generator task - About to generate an interrupt.\r\n" );

        vPortGenerateSimulatedInterrupt( mainINTERRUPT_NUMBER );

        vPrintString( "Generator task - Interrupt generated.\r\n\r\n\r\n");

    }

}
```

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

```
static uint32_t ulExampleInterruptHandler( void )

{

    BaseType_t xHigherPriorityTaskWoken;

    uint32_t ulReceivedNumber;

    /* The strings are declared static const to ensure they are not allocated on the
 interrupt service routine's stack, and so exist even when the interrupt service routine is
 not executing. */

    static const char *pcStrings[] =

    {

        "String 0\r\n",

        "String 1\r\n",

        "String 2\r\n",

        "String 3\r\n"

     };

    /* As always, xHigherPriorityTaskWoken is initialized to pdFALSE to be able to detect
 it getting set to pdTRUE inside an interrupt-safe API function. Because an interrupt-safe
```

```
API function can only set xHigherPriorityTaskWoken to pdTRUE, it is safe to use the same
xHigherPriorityTaskWoken variable in both the call to xQueueReceiveFromISR() and the call
to xQueueSendToBackFromISR(). */

   xHigherPriorityTaskWoken = pdFALSE;

   /* Read from the queue 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 inclusive), and
then use the truncated value as an index into the pcStrings[] array to select a string
(char *) to send on the other queue. */

       ulReceivedNumber &= 0x03;

       xQueueSendToBackFromISR( xStringQueue,  &pcStrings[ ulReceivedNumber ],
&xHigherPriorityTaskWoken );

   }

   /* If receiving from xIntegerQueue caused a task to leave the Blocked state, and if the
priority of the task that left the Blocked state is higher than the priority of the task
in the Running state, then xHigherPriorityTaskWoken will have been set to pdTRUE inside
xQueueReceiveFromISR(). If sending to xStringQueue caused a task to leave the Blocked
state, and if the priority of the task that left the Blocked state is higher than the
priority of the task in the Running state, then xHigherPriorityTaskWoken will have been
set to pdTRUE inside xQueueSendToBackFromISR(). xHigherPriorityTaskWoken is used as the
parameter to portYIELD_FROM_ISR(). If xHigherPriorityTaskWoken equals pdTRUE, then calling
portYIELD_FROM_ISR() will request a context switch. If xHigherPriorityTaskWoken is still
pdFALSE, then calling portYIELD_FROM_ISR() will have no effect. The implementation of
portYIELD_FROM_ISR() used by the Windows port includes a return statement, which is why
this function does not explicitly return a value. */

   portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

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 here.

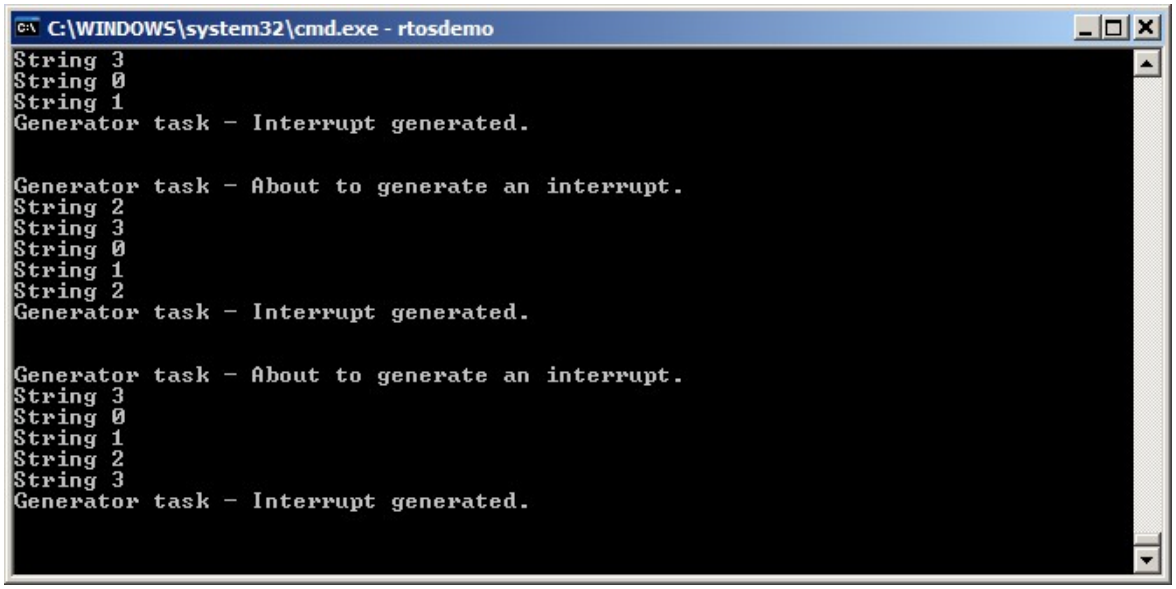
```
static void vStringPrinter( void *pvParameters )

{

    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 );

    }
```

```
}
```

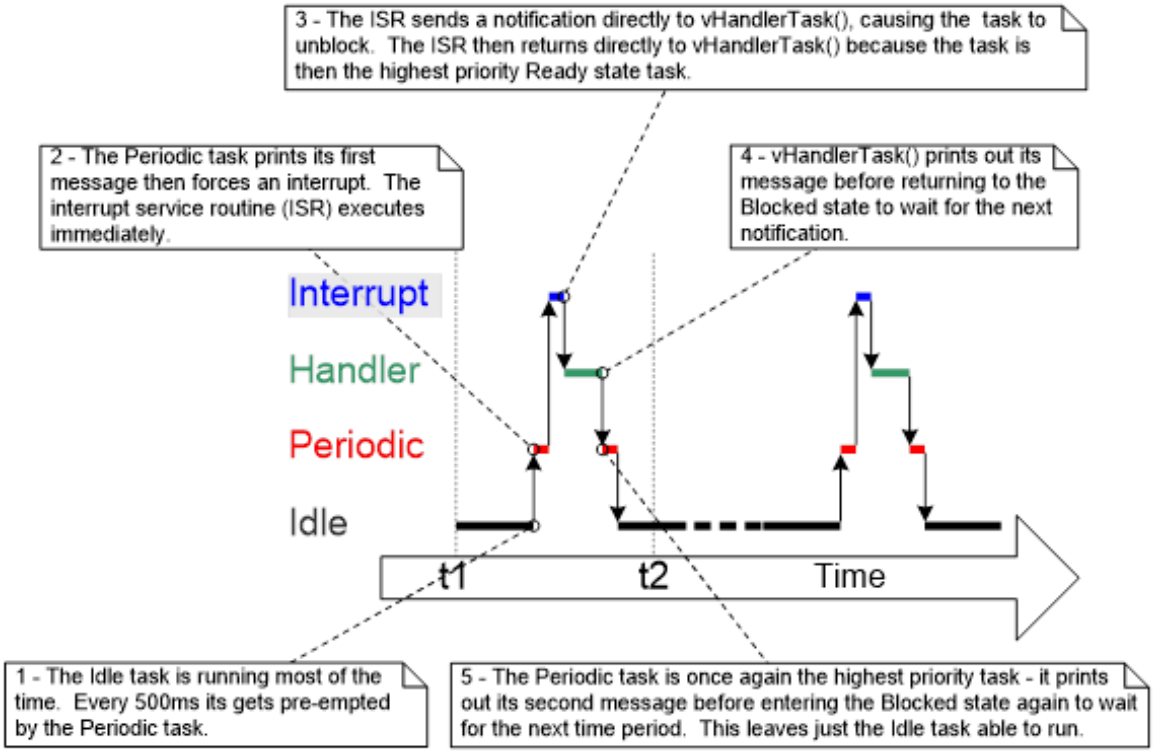
As normal, main() creates the required queues and tasks before starting the scheduler. Its implementation is shown here.

```
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 uint32_t. 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( uint32_t ) );

    xStringQueue = xQueueCreate( 10, sizeof( char * ) );

    /* Create the task that uses a queue to pass integers to the interrupt service routine.
 The task is created at priority 1. */

    xTaskCreate( vIntegerGenerator, "IntGen", 1000, NULL, 1, NULL );

    /* Create the task that prints out the strings sent to it from the interrupt service
 routine. This task is created at the higher priority of 2. */

    xTaskCreate( vStringPrinter, "String", 1000, NULL, 2, NULL );

    /* Install the handler for the software interrupt. The syntax required to do this is
 depends on the FreeRTOS port being used. The syntax shown here can only be used with the
 FreeRTOS Windows port, where such interrupts are only simulated. */

    vPortSetInterruptHandler( mainINTERRUPT_NUMBER, ulExampleInterruptHandler );

    /* Start the scheduler so the created tasks start executing. */

    vTaskStartScheduler();

    /* If all is well, then main() will never reach here because the scheduler will
 now be running the tasks. If main() does reach here, then it is likely that there was
 insufficient heap memory available for the idle task to be created. */

    for( ;; );

}
```

The output is shown here. You'll see the interrupt receives all five integers and produces five strings in response.



Here is the sequence of execution.



Interrupt Nesting

It is common to confuse task priorities and interrupt priorities. This section discusses interrupt priorities, which are the priorities at which ISRs execute relative to each other. The priority assigned to a task is in no way related to the priority assigned to an interrupt. Hardware decides when an ISR will execute. Software decides when a task will execute. An ISR executed in response to a hardware interrupt will interrupt a task, but a task cannot preempt an ISR.

Ports that support interrupt nesting require one or both of the constants listed in the following table to be defined in FreeRTOSConfig.h. configMAX_SYSCALL_INTERRUPT_PRIORITY and configMAX_API_CALL_INTERRUPT_PRIORITY both define the same property. Older FreeRTOS ports use configMAX_SYSCALL_INTERRUPT_PRIORITY. Newer FreeRTOS ports use configMAX_API_CALL_INTERRUPT_PRIORITY.

The following table lists constants that control interrupt nesting.

Constant	Description
configMAX_SYSCALL_INTERRUPT_PRIORITY or configMAX_API_CALL_INTERRUPT_PRIORITY	Sets the highest interrupt priority from which interrupt-safe FreeRTOS API functions can be called.
configKERNEL_INTERRUPT_PRIORITY	Sets the interrupt priority used by the tick interrupt and must always be set to the lowest possible interrupt priority. If the FreeRTOS port in use does not also use the configMAX_SYSCALL_INTERRUPT_PRIORITY constant, then any interrupt that uses interrupt-safe FreeRTOS API functions must also execute at the priority defined by configKERNEL_INTERRUPT_PRIORITY.

Each interrupt source has a numeric and logical priority.

- Numeric

  The number assigned to the interrupt priority. For example, if an interrupt is assigned a priority of 7, then its numeric priority is 7. Likewise, if an interrupt is assigned a priority of 200, then its numeric priority is 200.
- Logical

  Describes the interrupt's precedence over other interrupts. If two interrupts of differing priority occur at the same time, then the processor will execute the ISR for the interrupt that has the higher logical priority.

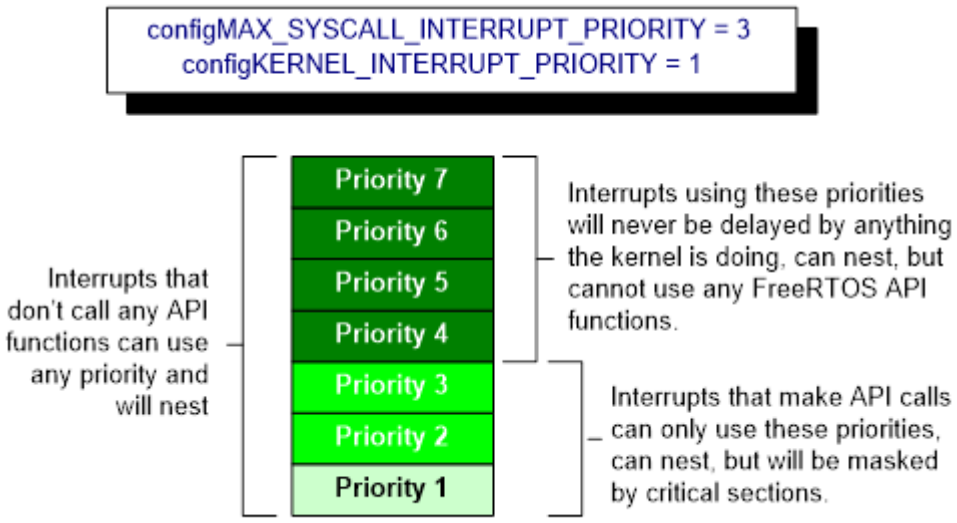
  An interrupt can interrupt (nest with) any interrupt that has a lower logical priority, but it cannot interrupt one that has an equal or higher logical priority.

The relationship between an interrupt's numeric priority and logical priority depends on the processor architecture. On some processors, the higher the numeric priority assigned to an interrupt, the higher that interrupt's logical priority will be. On other processor architectures, the higher the numeric priority assigned to an interrupt, the lower that interrupt's logical priority will be.

You can create a full interrupt nesting model by setting configMAX_SYSCALL_INTERRUPT_PRIORITY to a higher logical interrupt priority than configKERNEL_INTERRUPT_PRIORITY. The following figure shows a scenario in which:

- The processor has 7 unique interrupt priorities.
- Interrupts assigned a numeric priority of 7 have a higher logical priority than interrupts assigned a numeric priority of 1.
- configKERNEL_INTERRUPT_PRIORITY is set to 1.
- configMAX_SYSCALL_INTERRUPT_PRIORITY is set to 3.

This figure shows constants that affect interrupt nesting behavior.



- Interrupts that use priorities 1 to 3, inclusive, are prevented from executing while the kernel or the application is inside a critical section. ISRs running at these priorities can use interrupt-safe FreeRTOS API functions. For information about critical sections, see Resource Management (p. 155).
- Interrupts that use priority 4 or above are not affected by critical sections, so nothing the scheduler does will prevent these interrupts from executing immediately within the limitations of the hardware itself. ISRs executing at these priorities cannot use any FreeRTOS API functions.
- Typically, functionality that requires very strict timing accuracy (motor control, for example) would use a priority above configMAX_SYSCALL_INTERRUPT_PRIORITY to ensure the scheduler does not introduce jitter into the interrupt response time.

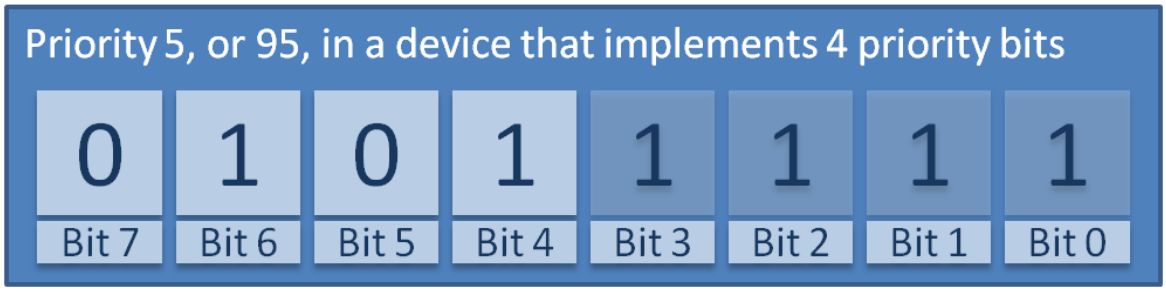
ARM Cortex-M and ARM GIC Users

This section only partially applies to Cortex-M0 and Cortex-M0+ cores. Interrupt configuration on Cortex-M processors is confusing and prone to error. To assist your development, the FreeRTOS Cortex-M ports automatically check the interrupt configuration, but only if configASSERT() is defined. For information about configASSERT(), see Developer Support (p. 221).

The ARM Cortex cores and ARM Generic Interrupt Controllers (GICs) use numerically low priority numbers to represent logically high priority interrupts. This seems counterintuitive. If you want to assign an interrupt a logically low priority, then it must be assigned a numerically high value. If you want to assign an interrupt a logically high priority, then it must be assigned a numerically low value.

The Cortex-M interrupt controller allows a maximum of eight bits to be used to specify each interrupt priority, making 255 the lowest possible priority. Zero is the highest priority. However, Cortex-M microcontrollers normally only implement a subset of the eight possible bits. The number of bits implemented depends on the microcontroller family.

When only a subset has been implemented, only the most significant bits of the byte can be used. The least significant bits are left unimplemented. Unimplemented bits can take any value, but it is normal to set them to 1. This figure shows how a priority of binary 101 is stored in a Cortex-M microcontroller that implements four priority bits.



You'll see the binary value 101 has been shifted into the most significant four bits because the least significant four bits are not implemented. The unimplemented bits have been set to 1.

Some library functions expect priority values to be specified after they have been shifted into the implemented (most significant) bits. When using such a function, the priority shown in this figure can be specified as decimal 95. Decimal 95 is binary 101 shifted up by four to make binary 101nnnn (where *n* is an unimplemented bit). The unimplemented bits are set to 1 to make binary 1011111.

Some library functions expect priority values to be specified before they have been shifted up into the implemented (most significant) bits. When using such a function, the priority shown in the figure must be specified as decimal 5. Decimal 5 is binary 101 without any shift.

configMAX_SYSCALL_INTERRUPT_PRIORITY and configKERNEL_INTERRUPT_PRIORITY must be specified in a way that allows them to be written directly to the Cortex-M registers (that is, after the priority values have been shifted up into the implemented bits).

configKERNEL_INTERRUPT_PRIORITY must always be set to the lowest possible interrupt priority. Unimplemented priority bits can be set to 1, so the constant can always be set to 255, no matter how many priority bits are implemented.

Cortex-M interrupts will default to a priority of zero, the highest possible priority. The implementation of the Cortex-M hardware does not permit configMAX_SYSCALL_INTERRUPT_PRIORITY to be set to 0, so the priority of an interrupt that uses the FreeRTOS API must never be left at its default value.

Resource Management

This section covers:

- 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.

In a multitasking system, there is potential for error 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 can result in data corruption or other similar issue.

The following are some examples:

1. Accessing peripherals

   Consider the following scenario where two tasks attempt to write to an Liquid Crystal Display (LCD).

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

   b. Task A is preempted by Task B after outputting just the beginning of the string "Hello w".

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

   d. Task A continues from the point at which it was preempted and outputs the remaining characters of its string ("orld").

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

2. Read, modify, write operations

The following shows a line of C code and an example of how it would typically be translated into assembly code. You'll see that the value of PORTA is first read from memory into a register, modified in the register, and then written back to memory. This is called a read, modify, write operation.

```
/* The C code being compiled. */

PORTA |= 0x01;

/* The assembly code produced when the C code is compiled. */

LOAD R1,[#PORTA] ; Read a value from PORTA into R1

MOVE R2,#0x01 ; Move the absolute constant 1 into R2

OR R1,R2 ; Bitwise OR R1 (PORTA) with R2 (constant 1)

STORE R1,[#PORTA] ; Store the new value back to PORTA
```

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

1. Task A loads the value of PORTA into a register, the read portion of the operation.
2. Task A is preempted by Task B before it completes the modify and write portions of the same operation.
3. Task B updates the value of PORTA, and then enters the Blocked state.
4. Task A continues from the point at which it was preempted. It modifies the copy of the PORTA value that it already holds in a register before writing the updated value back to PORTA.

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

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

1. Non-atomic access to variables

   Updating multiple members of a structure or updating a variable that is larger than the natural word size of the architecture (for example, updating a 32-bit variable on a 16-bit machine) are examples of non-atomic operations. If they are interrupted, they can result in data loss or corruption.
2. 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. Reentrant functions are said to be *thread-safe* because they can be accessed from more than one thread of execution without the risk of data or logical operations becoming corrupted. Each task maintains its own stack and its own set of processor (hardware) register values. If a function does not access data other than data stored on the stack or held in a register, then the function is reentrant and thread-safe. Here is an example of a reentrant function.

```
/* A parameter is passed into the function. This will either be passed on the stack, or
 in a processor register. Either way is safe because each task or interrupt that calls
 the function maintains its own stack and its own set of register values, so each task or
 interrupt that calls the function will have its own copy of lVar1. */
long lAddOneHundred( 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;

    return lVar2;

}
```

Here is an example of a function that is not reentrant.

```
/* In this case lVar1 is a global variable, so every task that calls lNonsenseFunction will
 access the same single copy of the variable. */
```

```
long lVar1;

long lNonsenseFunction( void )

{

    /* lState is static, so is not allocated on the stack. Each task that calls this
 function will access the same single copy of the variable. */

    static long lState = 0;

    long lReturn;

    switch( lState )

    {

        case 0 : lReturn = lVar1 + 10;

            lState = 1;

            break;

        case 1 : lReturn = lVar1 + 20;

            lState = 0;

            break;

    }

}
```

Mutual Exclusion

To ensure data consistency is maintained at all times, use a *mutual exclusion* technique to manage access to a resource that is shared between tasks or between tasks and interrupts. The goal is to ensure that, once a task starts to access a shared resource that is not reentrant and thread-safe, the same task has exclusive access to the resource 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 practical, design the application in such a way that resources are not shared, and each resource is accessed only from a single task.

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. Critical sections are also known as critical regions.

taskENTER_CRITICAL() and taskEXIT_CRITICAL() do not take any parameters or return a value. (A function-like macro does not really return a value in the same way a real function does, but it is simplest to think of the macro as if it were a function.)

Their use is shown here, where a critical section is used to guard access to a register.

```
/* Ensure access to the PORTA register cannot be interrupted by placing it within a
 critical section. Enter the critical section. */

taskENTER_CRITICAL();

/* A switch to another task cannot occur between the call to taskENTER_CRITICAL() and the
 call to taskEXIT_CRITICAL(). Interrupts might still execute on FreeRTOS ports that allow
 interrupt nesting, but only interrupts whose logical priority is above the value assigned
 to the configMAX_SYSCALL_INTERRUPT_PRIORITY constant. Those interrupts are not permitted
 to call FreeRTOS API functions. */

PORTA |= 0x01;

/* Access to PORTA has finished, so it is safe to exit the critical section. */

taskEXIT_CRITICAL();
```

Several of the examples use a function called vPrintString() to write strings to standard out, which is the terminal window when the FreeRTOS Windows port is used. vPrintString() is called from many different tasks, so in theory its implementation could protect access to standard out using a critical section, as shown here.

```
void vPrintString( const char *pcString )

{

        /* Write the string to stdout, using a critical section as a crude method of mutual
 exclusion. */

        taskENTER_CRITICAL();

        {

            printf( "%s", pcString );

            fflush( stdout );

        }

        taskEXIT_CRITICAL();

}
```

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

Basic 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 the preceding code) because writing to the terminal can be a relatively long operation. The examples in this section 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().

Calling taskENTER_CRITICAL() and taskEXIT_CRITICAL() is the only legitimate way for a task to alter the interrupt enable state of the processor on which FreeRTOS is running. Altering the interrupt enable state by any other means will invalidate the macro's nesting count.

taskENTER_CRITICAL() and taskEXIT_CRITICAL() do not end in 'FromISR', so they must not be called from an interrupt service routine. taskENTER_CRITICAL_FROM_ISR() is an interrupt-safe version of taskENTER_CRITICAL(). taskEXIT_CRITICAL_FROM_ISR() is an interrupt-safe version of taskEXIT_CRITICAL(). The interrupt-safe versions are provided only for FreeRTOS ports that allow interrupts to nest. They would be obsolete in ports that do not allow interrupts to nest.

taskENTER_CRITICAL_FROM_ISR() returns a value that must be passed into the matching call to taskEXIT_CRITICAL_FROM_ISR(), as shown here.

```
void vAnInterruptServiceRoutine( void )

{

    /* Declare a variable in which the return value from taskENTER_CRITICAL_FROM_ISR() will
 be saved. */

    UBaseType_t uxSavedInterruptStatus;

    /* This part of the ISR can be interrupted by any higher priority interrupt. */

    /* Use taskENTER_CRITICAL_FROM_ISR() to protect a region of this ISR. Save the value
 returned from taskENTER_CRITICAL_FROM_ISR() so it can be passed into the matching call to
 taskEXIT_CRITICAL_FROM_ISR(). */

    uxSavedInterruptStatus = taskENTER_CRITICAL_FROM_ISR();

    /* This part of the ISR is between the call to taskENTER_CRITICAL_FROM_ISR() and
 taskEXIT_CRITICAL_FROM_ISR(), so can only be interrupted by interrupts that have a
 priority above that set by the configMAX_SYSCALL_INTERRUPT_PRIORITY constant. */

    /* Exit the critical section again by calling taskEXIT_CRITICAL_FROM_ISR(), passing in
 the value returned by the matching call to taskENTER_CRITICAL_FROM_ISR(). */

    taskEXIT_CRITICAL_FROM_ISR( uxSavedInterruptStatus );

    /* This part of the ISR can be interrupted by any higher priority interrupt. */

}
```

It is wasteful to use more processing time executing the code that enters and then subsequently exits a critical section than the code being protected by the critical section. Basic critical sections are very fast to enter, very fast to exit, and always deterministic, making their use ideal when the region of code being protected is very short.

Suspending (or Locking) the Scheduler

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

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

A critical section that is too long to be implemented by simply disabling interrupts can instead be implemented by suspending the scheduler. However, interrupt activity while the scheduler is suspended

can make resuming (or *unsuspending*) the scheduler a relatively long operation. Consider the best method to use in each case.

vTaskSuspendAll() API Function

The vTaskSuspendAll() API function prototype is shown here.

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

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

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

xTaskResumeAll() API Function

The xTaskResumeAll() API function prototype is shown here.

```
BaseType_t xTaskResumeAll( void );
```

The scheduler is resumed (unsuspended) by calling xTaskResumeAll().

The following table lists 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. If a pending context switch is performed before xTaskResumeAll() returns, then pdTRUE is returned. Otherwise, pdFALSE is returned.

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().

The following code shows the implementation of vPrintString(), which suspends the scheduler to protect access to the terminal output.

```
void vPrintString( const char *pcString )

{

    /* Write the string to stdout, suspending the scheduler as a method of mutual
 exclusion. */
```

```
    vTaskSuspendScheduler();

    {

        printf( "%s", pcString );

        fflush( stdout );

    }

    xTaskResumeScheduler();

}
```

Mutexes (and Binary Semaphores)

A mutex (MUTual EXclusion) is a special type of binary semaphore that is used to control access to a resource that is shared between two or more tasks. To make mutexes available, in FreeRTOSConfig.h, configUSE_MUTEXES must be set to 1.

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.

Even though mutexes and binary semaphores share many characteristics, the scenario in which a mutex is used for mutual exclusion is completely different from one in which 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.

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.

xSemaphoreCreateMutex() API Function

FreeRTOS V9.0.0 also includes the xSemaphoreCreateMutexStatic() function, which allocates the memory required to create a mutex statically at compile time. A mutex is a type of semaphore. Handles to all the various types of FreeRTOS semaphore are stored in a variable of type SemaphoreHandle_t.

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

The xSemaphoreCreateMutex() API function prototype is shown here.

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

The following table lists the xSemaphoreCreateMutex() return value.

Rewriting vPrintString() to Use a Semaphore (Example 20)

This example creates a new version of vPrintString() called prvNewPrintString(), and then calls the new function from multiple tasks. prvNewPrintString() is functionally identical to vPrintString(), but controls access to standard out using a mutex, rather than by locking the scheduler.

The implementation of prvNewPrintString() is shown here.

```
static void prvNewPrintString( const char *pcString )

{

    /* The mutex is created before the scheduler is started, so already exists by the time
 this task executes. Attempt to take the mutex, blocking indefinitely to wait for the mutex
 if it is not available right 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).
 Indefinite timeouts are not recommended for production code. */

    xSemaphoreTake( xMutex, portMAX_DELAY );

    {

        /* The following line will only execute after the mutex has been successfully
 obtained. Standard out can be accessed freely now because only one task can have the mutex
 at any one time. */

        printf( "%s", pcString );

        fflush( stdout );

        /* The mutex MUST be given back! */

    }

    xSemaphoreGive( xMutex );

}
```

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 here.

```
static void prvPrintTask( void *pvParameters )

{

    char *pcStringToPrint;

    const TickType_t xMaxBlockTimeTicks = 0x20;

    /* Two instances of this task are created. The string printed by the task is passed
 into the task using the task's parameter. The parameter is cast 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 because 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() % xMaxBlockTimeTicks ) );

    }

}
```

As normal, main() simply creates the mutex and tasks, and then starts the scheduler.

The two instances of prvPrintTask() are created at different priorities, so the lower priority task will sometimes be preempted by the higher priority task. Because a mutex is used to ensure each task gets mutually exclusive access to the terminal, even when preemption occurs, the strings that are displayed will be correct and in no way corrupted. You can increase the frequency of preemption by reducing the maximum time the tasks spend in the Blocked state, which is set by the xMaxBlockTimeTicks constant.

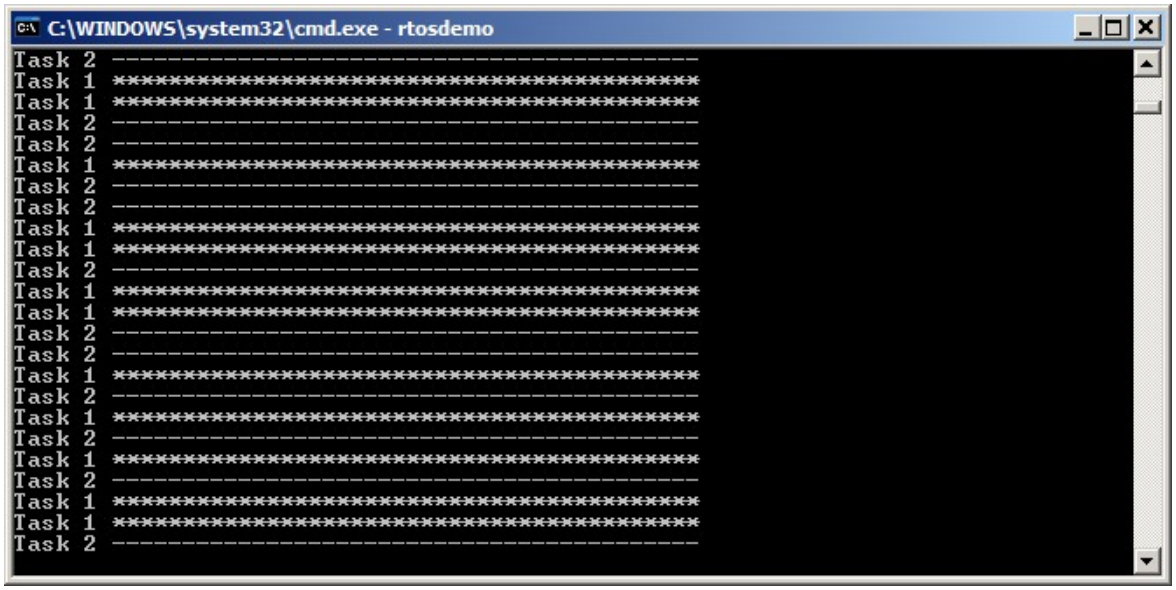
Notes for using Example 20 with the FreeRTOS Windows port:

- Calling printf() generates a Windows system call. Windows system calls are outside the control of FreeRTOS and can introduce instability.
- The way in which Windows system calls execute means it is rare to see a corrupted string, even when the mutex is not used.

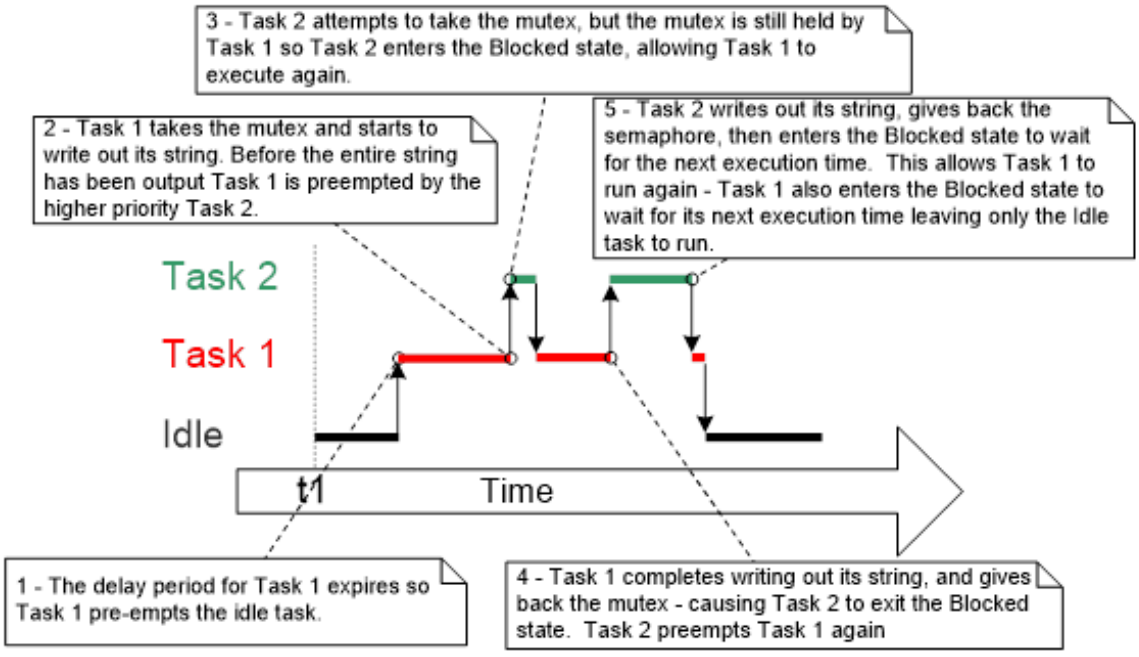
```
int main( void )

{

    /* Before a semaphore is used it must be explicitly created. In this example, a mutex
 type semaphore is created. */

    xMutex = xSemaphoreCreateMutex();

    /* Check that the semaphore was created successfully before creating the tasks. */

    if( xMutex != NULL )

    {

        /* Create two instances of the tasks that write to stdout. The string they write
 is passed in to the task as the task's parameter. The tasks are created at different
 priorities so some preemption will occur. */

        xTaskCreate( prvPrintTask, "Print1", 1000, "Task 1
 ***************************************\r\n", 1, NULL );

        xTaskCreate( prvPrintTask, "Print2", 1000, "Task 2
 ---------------------------------------\r\n", 2, NULL );

        /* Start the scheduler so the created tasks start executing. */

        vTaskStartScheduler();

    }
```

```
    /* If all is well, then main() will never reach here because the scheduler will
 now be running the tasks. If main() does reach here, then it is likely that there was
 insufficient heap memory available for the idle task to be created.  */

    for( ;; );

}
```

Here is the output.



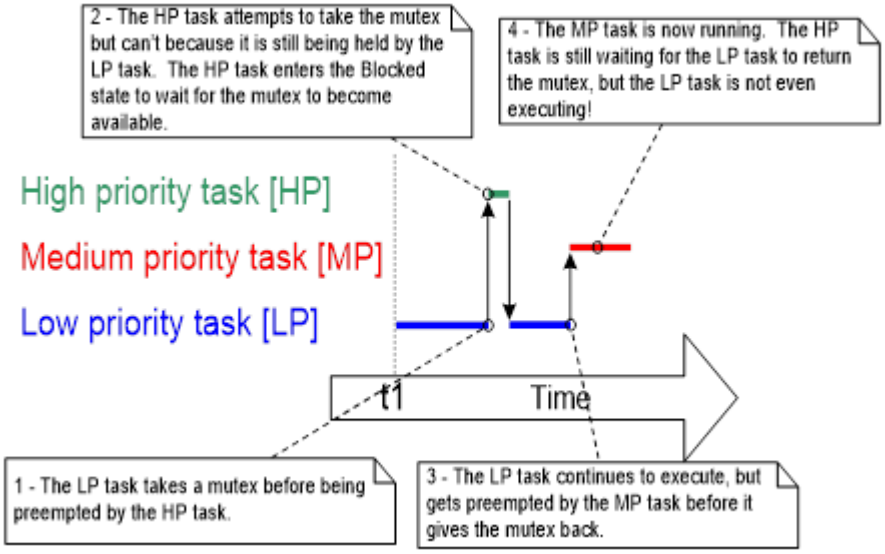
This figure shows a possible execution sequence.



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

Priority Inversion

The preceding figure illustrates one of the potential pitfalls of using a mutex to provide mutual exclusion. The 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 exacerbated 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 figure shows the worst case scenario.

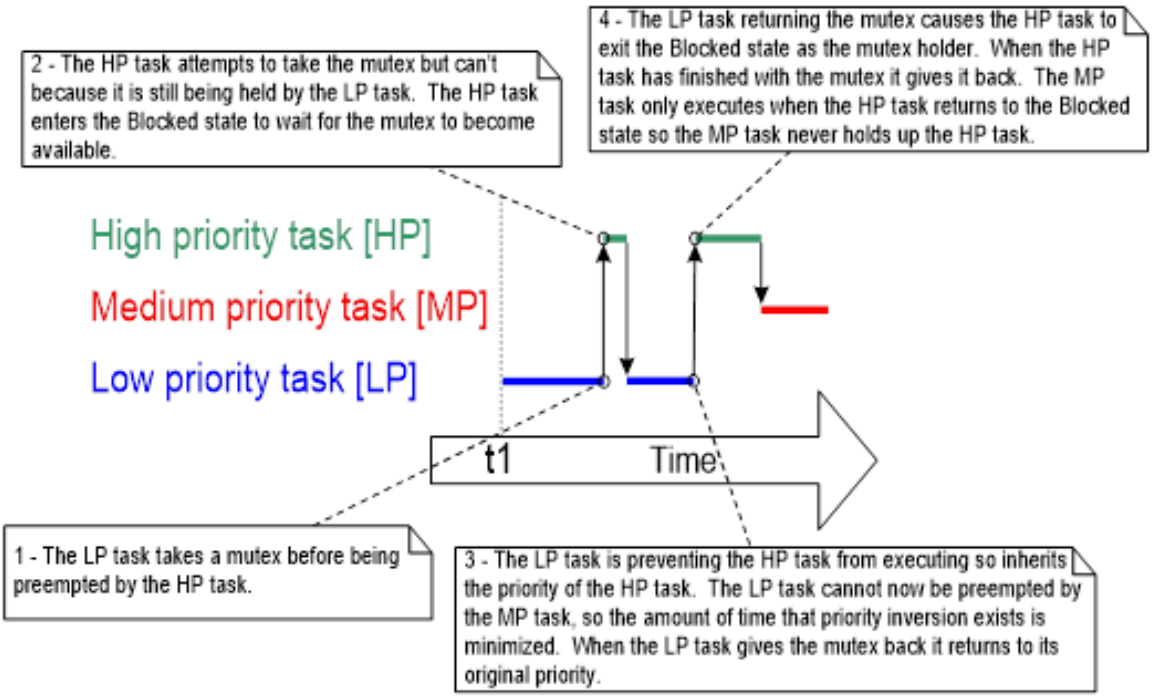


Priority inversion can be a significant problem, but in small embedded systems you can avoid it by considering how resources are accessed when you design your system.

Priority Inheritance

FreeRTOS mutexes and binary semaphores are very similar. The difference is that mutexes include a basic *priority inheritance* mechanism and binary semaphores do not. Priority inheritance is a scheme that minimizes the negative effects of priority inversion. It does not fix priority inversion, but reduces 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 the priority 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. The following figure shows the priority of the mutex holder is reset automatically to its original value when it gives the mutex back.



Priority inheritance functionality affects the priority of tasks that are using the mutex. For this reason, mutexes must not be used from an interrupt service routines.

Deadlock (or Deadly Embrace)

A *deadlock* or *deadly embrace* is another potential pitfall of using mutexes for mutual exclusion.

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 preempted 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.

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 for avoiding deadlock is to design your system to ensure that deadlock cannot occur. It is normally bad practice for a task to wait indefinitely (without a timeout) to obtain a mutex. Instead, use a timeout that is a little longer than the maximum time it is expected to have to wait for the mutex. Failure to obtain the mutex within that time will be a symptom of a design error, which might be a deadlock.

Deadlock is not a big problem in small embedded systems because if you as a system designer have a good understanding of the entire application, you can identify and remove the areas where it could occur.

Recursive Mutexes

It is also possible for a task to deadlock with itself. This will happen if a task attempts to take the same mutex more than once, without first returning the mutex. Consider the following scenario:

1. A task successfully obtains a mutex.
2. While holding the mutex, the task calls a library function.
3. The implementation of the library function attempts to take the same mutex and enters the Blocked state to wait for the mutex to become available.

The task is in the Blocked state to wait for the mutex to be returned, but the task is already the mutex holder. A deadlock has occurred because the task is in the Blocked state to wait for itself.

You can avoid this type of deadlock by using a recursive mutex in place of a standard mutex. A recursive mutex can be taken more than once by the same task. It will be returned only after one call to give the recursive mutex has been executed for every preceding call to take the recursive mutex.

Standard mutexes and recursive mutexes are created and used in a similar way:

- Standard mutexes are created using xSemaphoreCreateMutex(). Recursive mutexes are created using xSemaphoreCreateRecursiveMutex(). The two API functions have the same prototype.
- Standard mutexes are taken using xSemaphoreTake(). Recursive mutexes are taken using xSemaphoreTakeRecursive(). The two API functions have the same prototype.
- Standard mutexes are given using xSemaphoreGive(). Recursive mutexes are given using xSemaphoreGiveRecursive(). The two API functions have the same prototype.

The following code demonstrates how to create and use a recursive mutex.

```
/* Recursive mutexes are variables of type SemaphoreHandle_t. */

SemaphoreHandle_t xRecursiveMutex;

/* The implementation of a task that creates and uses a recursive mutex. */

void vTaskFunction( void *pvParameters )

{

    const TickType_t xMaxBlock20ms = pdMS_TO_TICKS( 20 );

    /* Before a recursive mutex is used it must be explicitly created. */

    xRecursiveMutex = xSemaphoreCreateRecursiveMutex();

    /* Check the semaphore was created successfully. configASSERT() is described in section
 11.2. */

    configASSERT( xRecursiveMutex );

    /* As per most tasks, this task is implemented as an infinite loop. */

    for( ;; )
```

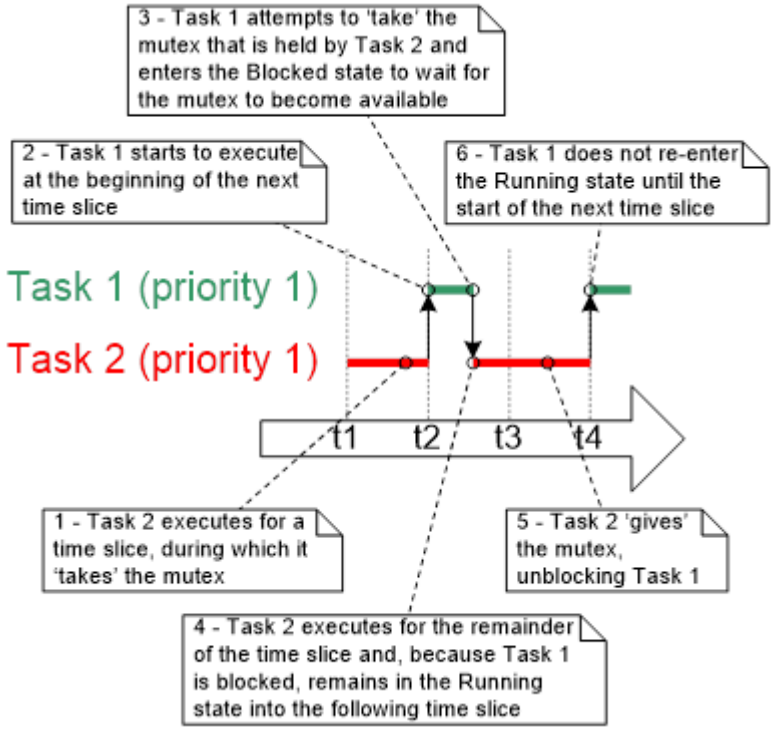
```
    {

        /* ... */

        /* Take the recursive mutex. */

        if( xSemaphoreTakeRecursive( xRecursiveMutex, xMaxBlock20ms ) == pdPASS)

        {

             /* The recursive mutex was successfully obtained. The task can now access the
resource the mutex is protecting. At this point the recursive call count (which is the
number of nested calls to xSemaphoreTakeRecursive()) is 1 because the recursive mutex has
only been taken once. */

            /* While it already holds the recursive mutex, the task takes the mutex
again. In a real application, this is only likely to occur inside a subfunction called
by this task because there is no practical reason to knowingly take the same mutex
more than once. The calling task is already the mutex holder, so the second call to
xSemaphoreTakeRecursive() does nothing more than increment the recursive call count to 2.
*/

            xSemaphoreTakeRecursive( xRecursiveMutex, xMaxBlock20ms );

            /* ... */

            /* The task returns the mutex after it has finished accessing the resource the
mutex is protecting. At this point the recursive call count is 2, so the first call to
xSemaphoreGiveRecursive() does not return the mutex. Instead, it simply decrements the
recursive call count back to 1. */

            xSemaphoreGiveRecursive( xRecursiveMutex );

            /* The next call to xSemaphoreGiveRecursive() decrements the recursive call
count to 0, so this time the recursive mutex is returned.*/

            xSemaphoreGiveRecursive( xRecursiveMutex );

            /* Now one call to xSemaphoreGiveRecursive() has been executed for every
proceeding call to xSemaphoreTakeRecursive(), so the task is no longer the mutex holder.
*/
               }

    }

}
```

Mutexes and Task Scheduling

If two tasks of different priority use the same mutex, then the FreeRTOS scheduling policy makes the order in which the tasks will execute clear. The highest priority task that is able to run will be selected as the task that enters the Running state. For example, if a high priority task is in the Blocked state to wait for a mutex that is held by a low priority task, then the high priority task will preempt the low priority task as soon as the low priority task returns the mutex. The high priority task will then become the mutex holder. This scenario has already been covered in the Priority Inheritance (p. 165) section.

It's common to incorrectly assume the order in which the tasks will execute when the tasks have the same priority. If Task 1 and Task 2 have the same priority, and Task 1 is in the Blocked state to wait for a mutex that is held by Task 2, then Task 1 will not preempt Task 2 when Task 2 gives the mutex. Instead, Task 2 will remain in the Running state. Task 1 will simply move from the Blocked state to the Ready state.

In this figure, the vertical lines mark the times at which a tick interrupt occurs.



You'll see the FreeRTOS scheduler does not make Task 1 the Running state task as soon as the mutex is available because:

1. Task 1 and Task 2 have the same priority, so unless Task 2 enters the Blocked state, a switch to Task 1 should not occur until the next tick interrupt (assuming configUSE_TIME_SLICING is set to 1 in FreeRTOSConfig.h).
2. If a task uses a mutex in a tight loop, and a context switch occurred each time the task gave the mutex, then the task would remain in the Running state for a short time. If two or more tasks used the same mutex in a tight loop, then processing time would be wasted by rapidly switching between the tasks.

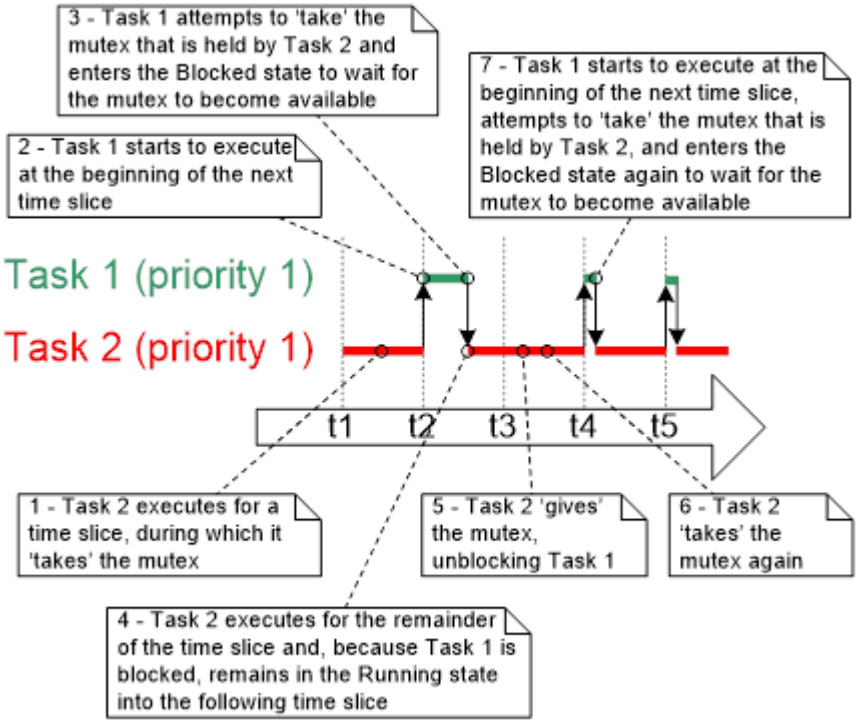
If a mutex is used in a tight loop by more than one task, and the tasks that use the mutex have the same priority, then make sure the tasks receive an approximately equal amount of processing time. The preceding figure shows a sequence of execution that could occur if two instances of the task shown in the following code are created at the same priority.

```
/* The implementation of a task that uses a mutex in a tight loop. The task creates a text
 string in a local buffer, and then writes the string to a display. Access to the display
 is protected by a mutex. */


void vATask( void *pvParameter )

{

    extern SemaphoreHandle_t xMutex;

    char cTextBuffer[ 128 ];

    for( ;; )
```

```
    {
        /* Generate the text string. This is a fast operation. */
        vGenerateTextInALocalBuffer( cTextBuffer );
        /* Obtain the mutex that is protecting access to the display. */
        xSemaphoreTake( xMutex, portMAX_DELAY );
        /* Write the generated text to the display. This is a slow operation. */
        vCopyTextToFrameBuffer( cTextBuffer );
        /* The text has been written to the display, so return the mutex. */
        xSemaphoreGive( xMutex );
    }
}
```

The comments in the code note that creating the string is a fast operation and updating the display is a slow operation. Therefore, as the mutex is held while the display is being updated, the task will hold the mutex for the majority of its runtime.

In the following figure, the vertical lines mark the times at which a tick interrupt occurs.



Step 7 in this figure shows Task 1 re-entering the Blocked state. That happens inside the xSemaphoreTake() API function.

Task 1 will be prevented from obtaining the mutex until the start of a time slice coincides with one of the short periods during which Task 2 is not the mutex holder.

You can avoid this scenario by adding a call to taskYIELD() after the call to xSemaphoreGive(). This is demonstrated in the code that follows, where taskYIELD() is called if the tick count changed while the task held the mutex. The code ensures that tasks that use a mutex in a loop receive a more equal amount of processing time, while also ensuring processing time is not wasted by switching between tasks too rapidly.

```
void vFunction( void *pvParameter )

{

    extern SemaphoreHandle_t xMutex;

    char cTextBuffer[ 128 ];

    TickType_t xTimeAtWhichMutexWasTaken;

    for( ;; )

    {

            /* Generate the text string. This is a fast operation. */

            vGenerateTextInALocalBuffer( cTextBuffer );

            /* Obtain the mutex that is protecting access to the display. */

            xSemaphoreTake( xMutex, portMAX_DELAY );

            /* Record the time at which the mutex was taken. */

            xTimeAtWhichMutexWasTaken = xTaskGetTickCount();

            /* Write the generated text to the display. This is a slow operation. */

            vCopyTextToFrameBuffer( cTextBuffer );

            /* The text has been written to the display, so return the mutex. */

            xSemaphoreGive( xMutex );

            /* If taskYIELD() was called on each iteration, then this task would only ever
 remain in the Running state for a short period of time, and processing time would be
 wasted by rapidly switching between tasks. Therefore, only call taskYIELD() if the tick
 count changed while the mutex was held. */

            if( xTaskGetTickCount() != xTimeAtWhichMutexWasTaken )

            {

                taskYIELD();

            }

    }

}
```

Gatekeeper Tasks

Gatekeeper tasks provide a clean method for 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 that needs to access the resource can do so only indirectly by using the services of the gatekeeper.

Rewriting vPrintString() to Use a Gatekeeper Task (Example 21)

This example provides another 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 standard out, it does not call a print function directly. Instead, it sends the message to the gatekeeper.

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

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

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. In FreeRTOSConfig.h, set configUSE_TICK_HOOK to 1.
2. Provide the implementation of the hook function, using the exact function name and prototype shown here.

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

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

The implementation of the gatekeeper task is shown here. The scheduler will always execute immediately after the tick hook function, so interrupt-safe FreeRTOS API functions called from the tick hook do not need to use their pxHigherPriorityTaskWoken parameter, and the parameter can be set to NULL.

```
static void prvStdioGatekeeperTask( void *pvParameters )

{

    char *pcMessageToPrint;

    /* This is the only task that is allowed to write to standard out. Any other task
 wanting to write a string to the output does not access standard out directly, but
 instead sends the string to this task. Because this is the only task that 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 return only when a message has been
 successfully received. */

        xQueueReceive( xPrintQueue, &pcMessageToPrint, portMAX_DELAY );

        /* Output the received string. */

        printf( "%s", pcMessageToPrint );

        fflush( stdout );

        /* Loop back to wait for the next message. */

    }

}
```

The task that writes to the queue is shown here. As before, two separate instances of the task are created, and the string the task writes to the queue is passed into the task using the task parameter.

```
static void prvPrintTask( void *pvParameters )

{

    int iIndexToString;

    const TickType_t xMaxBlockTimeTicks = 0x20;

    /* 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 by passing a pointer to the string to
 the gatekeeper task through 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 because 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() % xMaxBlockTimeTicks ) );

    }

}
```

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 tasks write to the back of the queue. The tick hook implementation is shown here.

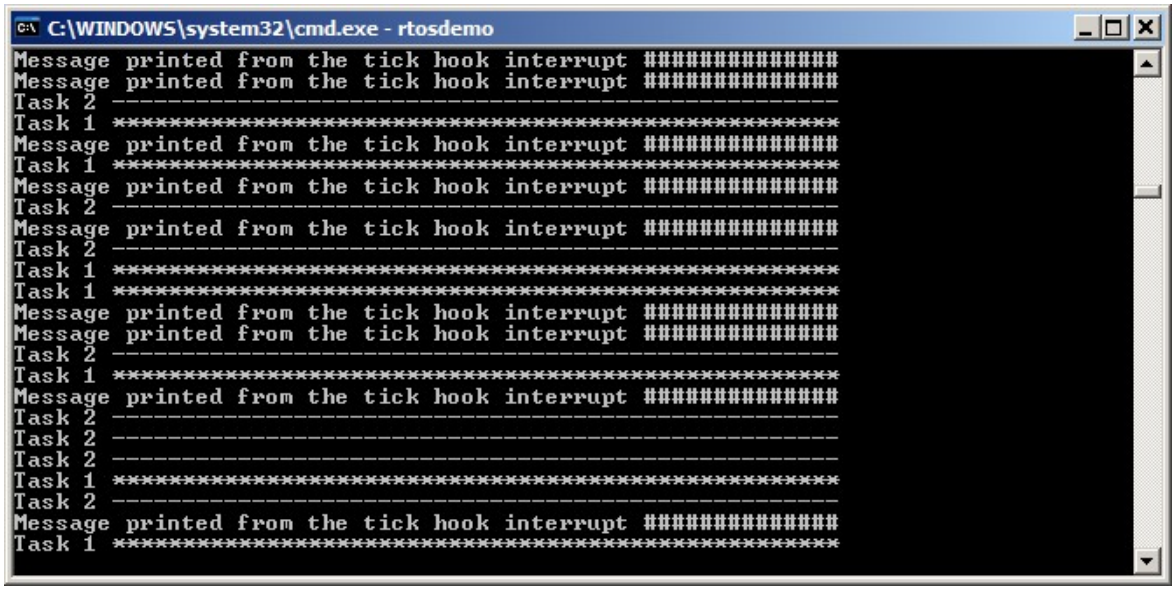
```
void vApplicationTickHook( void )

{

    static int iCount = 0;

    /* Print out a message every 200 ticks. The message is not written out directly, but
 sent to the gatekeeper task. */

    iCount++;

    if( iCount >= 200 )

    {

        /* Because xQueueSendToFrontFromISR() is being called from the tick hook, it is not
 necessary to use the xHigherPriorityTaskWoken parameter (the third parameter), and the
 parameter is set to NULL. */

        xQueueSendToFrontFromISR( xPrintQueue, &( pcStringsToPrint[ 2 ] ), NULL );

        /* Reset the count ready to print out the string again in 200 ticks time. */

        iCount = 0;

    }

}
```

As normal, main() creates the queues and tasks required to run the example, and then starts the scheduler. The implementation of main() is shown here.

```
/* Define the strings that the tasks and interrupt will print out via the gatekeeper. */

static char *pcStringsToPrint[] =

{

    "Task 1 ****************************************************\r\n", "Task 2
 ---------------------------------------------------\r\n", "Message printed from the tick
 hook interrupt ##############\r\n"

};

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

/* Declare a variable of type QueueHandle_t. The queue is used to send messages from the
 print tasks and the tick interrupt to the gatekeeper task. */

QueueHandle_t 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 * ) );

    /* 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 through the task parameter (the
 4th parameter to xTaskCreate()). The tasks are created at different priorities, so the
 higher priority task will occasionally preempt the lower priority task. */

        xTaskCreate( prvPrintTask, "Print1", 1000, ( void * ) 0, 1, NULL );

        xTaskCreate( prvPrintTask, "Print2", 1000, ( void * ) 1, 2, NULL );

        /* Create the gatekeeper task. This is the only task that is permitted to directly
 access standard out. */

        xTaskCreate( prvStdioGatekeeperTask, "Gatekeeper", 1000, NULL, 0, NULL
    );

    /* Start the scheduler so the created tasks start executing. */

    vTaskStartScheduler();

    }

    /* If all is well, then main() will never reach here because the scheduler will
 now be running the tasks. If main() does reach here, then it is likely that there was
 insufficient heap memory available for the idle task to be created.*/
     for( ;; );

}
```

The output is shown here. You'll see the strings originating from the tasks and the strings originating from the interrupt all print out correctly with no corruption.



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's appropriate to assign the gatekeeper a higher priority so messages get processed immediately. Doing so would be at the cost of the gatekeeper delaying lower priority tasks until it has completed accessing the protected resource.

Event Groups

This section covers:

- Practical uses for event groups.
- The advantages and disadvantages of event groups relative to other FreeRTOS features.
- How to set bits in an event group.
- How to wait in the Blocked state for bits to be set in an event group.
- How to use an event group to synchronize a set of tasks.

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

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

These unique properties of event groups make them useful for synchronizing multiple tasks, broadcasting events to more than one task, allowing a task to wait in the Blocked state for any one of a set of events to occur, and allowing a task to wait in the Blocked state for multiple actions to complete.

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

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

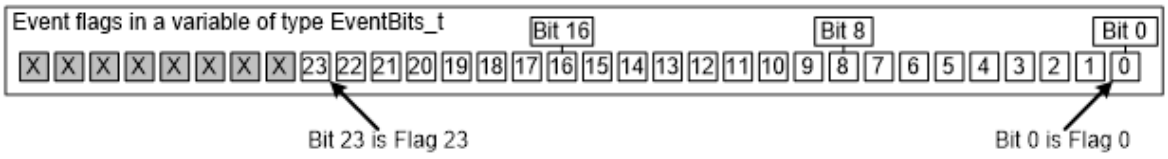
Characteristics of an Event Group

Event Groups, Event Flags, and Event Bits

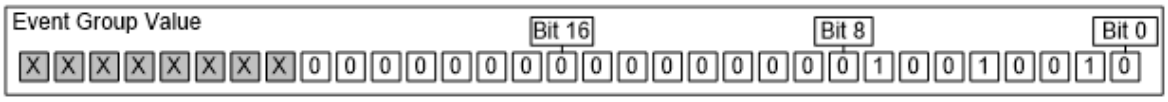
An event flag is a Boolean (1 or 0) value used to indicate if an event has occurred. An event group is a set of event flags.

An event flag can only be 1 or 0, which allows its state to be stored in a single bit and the state of all the event flags in an event group to be stored in a single variable. The state of each event flag in an event group is represented by a single bit in a variable of type EventBits_t. For that reason, event flags are also known as *event bits*. If a bit is set to 1 in the EventBits_t variable, then the event represented by that bit has occurred. If a bit is set to 0 in the EventBits_t variable, then the event represented by that bit has not occurred.

This figure shows how individual event flags are mapped to individual bits in a variable of type EventBits_t.



As an example, if the value of an event group is 0x92 (binary 1001 0010), then only event bits 1, 4, and 7 are set, so only the events represented by bits 1, 4, and 7 have occurred. This figure shows a variable of type EventBits_t that has event bits 1, 4, and 7 set and all the other event bits clear, giving the event group a value of 0x92.



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

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

More About the EventBits_t Data Type

The number of event bits in an event group depends on the configUSE_16_BIT_TICKS compile time configuration constant in FreeRTOSConfig.h. This constant configures the type used to hold the RTOS tick count, so it would seem unrelated to the event groups feature. Its effect on the EventBits_t type is a consequence of the FreeRTOS internal implementation and desirable because configUSE_16_BIT_TICKS should only be set to 1 when FreeRTOS is executing on an architecture that can handle 16-bit types more efficiently than 32-bit types.

- If configUSE_16_BIT_TICKS is 1, then each event group contains 8 usable event bits.
- If configUSE_16_BIT_TICKS is 0, then each event group contains 24 usable event bits.

Access by Multiple Tasks

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

A Practical Example of the Use of an Event Group

The implementation of the FreeRTOS+TCP TCP/IP stack provides a practical example of how an event group can be used to simultaneously simplify a design and minimize resource usage.

A TCP socket must respond to many different events, including accept events, bind events, read events, and close events. The events a socket can expect at any given time depends on the state of the socket. For example, if a socket has been created, but not yet bound to an address, then it can expect to receive a bind event, but not a read event. (It cannot read data if it does not have an address.)

The state of a FreeRTOS+TCP socket is held in a structure called FreeRTOS_Socket_t. The structure contains an event group that has an event bit defined for each event the socket must process. FreeRTOS +TCP API calls that block to wait for an event or group of events simply block on the event group.

The event group also contains an 'abort' bit, which allows a TCP connection to be aborted no matter which event the socket is waiting for at the time.

Event Management Using Event Groups

xEventGroupCreate() API Function

FreeRTOS V9.0.0 also includes the xEventGroupCreateStatic() function, which allocates the memory required to create an event group statically at compile time. An event group must be explicitly created before it can be used.

Event groups are referenced using variables of type EventGroupHandle_t. The xEventGroupCreate() API function is used to create an event group. It returns an EventGroupHandle_t to reference the event group it creates.

The xEventGroupCreate() API function prototype is shown here.

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

The following table lists the xEventGroupCreate() return value.

Parameter Name	Description
Return Value	If NULL is returned, then the event group cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the event group data structures. For more information, see Heap Memory Management (p. 14). If a non-NULL value is returned, the event group has been created successfully. The returned value should be stored as the handle to the created event group.

xEventGroupSetBits() API Function

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

Note: Do not call xEventGroupSetBits() from an interrupt service routine. Use the interrupt-safe version, xEventGroupSetBitsFromISR(), instead.

The xEventGroupSetBits() API function prototype is shown here.

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

The following table lists the xEventGroupSetBits() parameters and return value.

Parameter Name	Description
xEventGroup	The handle of the event group in which bits are being set. The event group handle will have been returned from the call to xEventGroupCreate() used to create the event group.

uxBitsToSet	A bit mask that specifies the event bit or bits to set to 1 in the event group. The value of the event group is updated by bitwise ORing the event group's existing value with the value passed in uxBitsToSet. As an example, setting uxBitsToSet to 0x04 (binary 0100) will result in event bit 3 in the event group becoming set (if it was not already set), while leaving all the other event bits in the event group unchanged.
Returned Value	The value of the event group at the time the call to xEventGroupSetBits() returned. The value returned will not necessarily have the bits specified by uxBitsToSet set because the bits might have been cleared again by a different task.

xEventGroupSetBitsFromISR() API Function

xEventGroupSetBitsFromISR() is the interrupt-safe version of xEventGroupSetBits().

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

The FreeRTOS design and implementation standard does not permit non-deterministic operations to be performed inside an interrupt service routine or when interrupts are disabled. For that reason, xEventGroupSetBitsFromISR() does not set event bits directly inside the interrupt service routine. Instead, it defers the action to the RTOS daemon task.

The xEventGroupSetBitsFromISR() API function prototype is shown here.

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

The following lists the xEventGroupSetBitsFromISR() parameters and return value.

xEventGroup

The handle of the event group in which bits are being set. The event group handle will have been returned from the call to xEventGroupCreate() used to create the event group.

uxBitsToSet

A bit mask that specifies the event bit, or event bits, to set to 1 in the event group. The value of the event group is updated by bitwise ORing the event group's existing value with the value passed to uxBitsToSet. As an example, setting uxBitsToSet to 0x05 (binary 0101) will result in event 3 and event bit 0 becoming set (if they were not already set), while leaving all the other event bits in the event group unchanged.

pxHigherPriorityTaskWoken

xEventGroupSetBitsFromISR() does not set the event bits directly inside the interrupt service routine. Instead, it defers the action to the RTOS daemon task by sending a command on the timer command queue. If the daemon task was in the Blocked state to wait for data to become available on the timer command queue, then writing to the timer command queue will cause the daemon task to

leave the Blocked state. If the priority of the daemon task is higher than the priority of the currently executing task (the task that was interrupted), then, internally, xEventGroupSetBitsFromISR() will set pxHigherPriorityTaskWoken to pdTRUE.

If xEventGroupSetBitsFromISR() 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 daemon task because the daemon task will be the highest priority Ready state task.

There are two possible return values:

pdPASS is returned only if data was successfully sent to the timer command queue.

pdFALSE is returned if the 'set bits' command could not be written to the timer command queue because the queue was already full.

xEventGroupWaitBits() API Function

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

The xEventGroupWaitBits() API function prototype is shown here.

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

The *unblock condition* is the condition used by the scheduler to determine if a task will enter the Blocked state and when a task will leave the Blocked state. The unblock condition is specified by a combination of the uxBitsToWaitFor and the xWaitForAllBits parameter values:

- uxBitsToWaitFor, which specifies the event bits in the event group to test.
- xWaitForAllBits, which specifies whether to use a bitwise OR test or a bitwise AND test.

A task will not enter the Blocked state if its unblock condition is met at the time xEventGroupWaitBits() is called.

The following table provides examples of conditions that will result in a task either entering or exiting the Blocked state. It only shows the least significant four binary bits of the event group and uxBitsToWaitFor values. The other bits of those two values are assumed to be zero.

Existing Event Group Value	uxBitsToWaitFor value	xWaitForAllBits value	Resultant Behavior
0000	0101	pdFALSE	The calling task will enter the Blocked state because neither of bit 0 or bit 2 are set in the event group, and will leave the Blocked state when either bit 0 OR bit 2 are set in the event group.
0100	0101	pdTRUE	The calling task will enter the Blocked state because bit 0 and bit 2 are not both set in

			the event group, and will leave the Blocked state when both bit 0 AND bit 2 are set in the event group.
0100	0110	pdFALSE	The calling task will not enter the Blocked state because xWaitForAllBits is pdFALSE, and one of the two bits specified by uxBitsToWaitFor is already set in the event group.
0100	0110	pdTRUE	The calling task will enter the Blocked state because xWaitForAllBits is pdTRUE, and only one of the two bits specified by uxBitsToWaitFor is already set in the event group. The task will leave the Blocked state when both bit 2 and bit 3 are set in the event group.

The calling task specifies bits to test using the uxBitsToWaitFor parameter. The calling task will most likely need to clear these bits back to zero after its unblock condition has been met. Event bits can be cleared using the xEventGroupClearBits() API function, but using that function to manually clear event bits will lead to race conditions in the application code if:

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

The xClearOnExit parameter is provided to avoid these potential race conditions. If xClearOnExit is set to pdTRUE, then the testing and clearing of event bits appears to the calling task to be an atomic operation (uninterruptable by other tasks or interrupts).

The following table lists the xEventGroupWaitBits() parameters and return value.

Parameter Name	**Description**
xEventGroup	The handle of the event group that contains the event bits being read. The event group handle will have been returned from the call to xEventGroupCreate() used to create the event group.
uxBitsToWaitFor	A bit mask that specifies the event bit, or event bits, to test in the event group. For example, if the calling task wants to wait for event bit 0 and/or event bit 2 to become set in

	the event group, then set uxBitsToWaitFor to 0x05 (binary 0101). Refer to Table 45 for further examples.
xClearOnExit	If the calling task's unblock condition has been met, and xClearOnExit is set to pdTRUE, then the event bits specified by uxBitsToWaitFor will be cleared back to 0 in the event group before the calling task exits the xEventGroupWaitBits() API function. If xClearOnExit is set to pdFALSE, then the state of the event bits in the event group are not modified by the xEventGroupWaitBits() API function.
xWaitForAllBits	The uxBitsToWaitFor parameter specifies the event bits to test in the event group. xWaitForAllBits specifies if the calling task should be removed from the Blocked state when one or more of the events bits specified by the uxBitsToWaitFor parameter are set, or only when all of the event bits specified by the uxBitsToWaitFor parameter are set. If xWaitForAllBits is set to pdFALSE, then a task that entered the Blocked state to wait for its unblock condition to be met will leave the Blocked state when any of the bits specified by uxBitsToWaitFor become set (or the timeout specified by the xTicksToWait parameter expires). If xWaitForAllBits is set to pdTRUE, then a task that entered the Blocked state to wait for its unblock condition to be met will only leave the Blocked state when all of the bits specified by uxBitsToWaitFor are set (or the timeout specified by the xTicksToWait parameter expires). See the previous table for examples.
xTicksToWait	The maximum amount of time the task should remain in the Blocked state to wait for its unblock condition to be met. xEventGroupWaitBits() will return immediately if xTicksToWait is zero or the unblock condition is met at the time xEventGroupWaitBits() is called. The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

Returned Value	If xEventGroupWaitBits() returned because the calling task's unblock condition was met, then the returned value is the value of the event group at the time the calling task's unblock condition was met (before any bits were automatically cleared if xClearOnExit was pdTRUE). In this case, the returned value will also meet the unblock condition. If xEventGroupWaitBits() returned because the block time specified by the xTicksToWait parameter expired, then the returned value is the value of the event group at the time the block time expired. In this case, the returned value will not meet the unblock condition.

Experimenting with Event Groups (Example 22)

This example shows how to:

- Create an event group.
- Set bits in an event group from an ISR.
- Set bits in an event group from a task.
- Block on an event group.

The effect of the xEventGroupWaitBits() xWaitForAllBits parameter is demonstrated by first executing the example with xWaitForAllBits set to pdFALSE, and then executing the example with xWaitForAllBits set to pdTRUE.

Event bit 0 and event bit 1 are set from a task. Event bit 2 is set from an ISR. These three bits are given descriptive names using the #define statements shown here.

```
/* Definitions for the event bits in the event group. */

#define mainFIRST_TASK_BIT ( 1UL << 0UL ) /* Event bit 0, which is set by a task. */

#define mainSECOND_TASK_BIT ( 1UL << 1UL ) /* Event bit 1, which is set by a task. */

#define mainISR_BIT ( 1UL << 2UL ) /* Event bit 2, which is set by an ISR. */
```

The following code shows the implementation of the task that sets event bit 0 and event bit 1. It sits in a loop, repeatedly setting one bit, and then the other, with a delay of 200 milliseconds between each call to xEventGroupSetBits(). A string is printed out before each bit is set to allow the sequence of execution to be seen in the console.

```
static void vEventBitSettingTask( void *pvParameters )

{

    const TickType_t xDelay200ms = pdMS_TO_TICKS( 200UL ), xDontBlock = 0;

    for( ;; )

    {

        /* Delay for a short while before starting the next loop. */
```

```
        vTaskDelay( xDelay200ms );

        /* Print out a message to say event bit 0 is about to be set by the task, and then
 set event bit 0. */

        vPrintString( "Bit setting task -\t about to set bit 0.\r\n" );

        xEventGroupSetBits( xEventGroup, mainFIRST_TASK_BIT );

        /* Delay for a short while before setting the other bit. */

        vTaskDelay( xDelay200ms );

        /* Print out a message to say event bit 1 is about to be set by the task, and then
 set event bit 1. */

        vPrintString( "Bit setting task -\t about to set bit 1.\r\n" );

        xEventGroupSetBits( xEventGroup, mainSECOND_TASK_BIT );

    }

}
```

The following code shows the implementation of the interrupt service routine that sets bit 2 in the event group. Again, a string is printed out before the bit is set to allow the sequence of execution to be seen in the console. In this case, because console output should not be performed directly in an interrupt service routine, xTimerPendFunctionCallFromISR() is used to perform the output in the context of the RTOS daemon task.

As in earlier examples, the interrupt service routine is triggered by a simple periodic task that forces a software interrupt. In this example, the interrupt is generated every 500 milliseconds.

```
static uint32_t ulEventBitSettingISR( void )

{

    /* The string is not printed within the interrupt service routine, but is instead
 sent to the RTOS daemon task for printing. It is therefore declared static to ensure the
 compiler does not allocate the string on the stack of the ISR because the ISR's stack
 frame will not exist when the string is printed from the daemon task. */

    static const char *pcString = "Bit setting ISR -\t about to set bit 2.\r\n";

    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Print out a message to say bit 2 is about to be set. Messages cannot be printed from
 an ISR, so defer the actual output to the RTOS daemon task by pending a function call to
 run in the context of the RTOS daemon task. */

    xTimerPendFunctionCallFromISR( vPrintStringFromDaemonTask, ( void * ) pcString,  0,
 &xHigherPriorityTaskWoken );

   /* Set bit 2 in the event group. */

    xEventGroupSetBitsFromISR( xEventGroup, mainISR_BIT, &xHigherPriorityTaskWoken );

    /* xTimerPendFunctionCallFromISR() and xEventGroupSetBitsFromISR() both write to the
 timer command queue, and both used the same xHigherPriorityTaskWoken variable. If writing
 to the timer command queue resulted in the RTOS daemon task leaving the Blocked state,
 and if the priority of the RTOS daemon task is higher than the priority of the currently
 executing task (the task this interrupt interrupted), then xHigherPriorityTaskWoken
 will have been set to pdTRUE. xHigherPriorityTaskWoken is used as the parameter
```

```
 to portYIELD_FROM_ISR(). If xHigherPriorityTaskWoken equals pdTRUE, then calling
 portYIELD_FROM_ISR() will request a context switch. If xHigherPriorityTaskWoken is still
 pdFALSE, then calling portYIELD_FROM_ISR() will have no effect. The implementation of
 portYIELD_FROM_ISR() used by the Windows port includes a return statement, which is why
 this function does not explicitly return a value. */

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

The following code shows the implementation of the task that calls xEventGroupWaitBits() to block on the event group. The task prints out a string for each bit that is set in the event group.

The xEventGroupWaitBits() xClearOnExit parameter is set to pdTRUE, so the event bit or bits that caused the call to xEventGroupWaitBits() to return will be cleared automatically before xEventGroupWaitBits() returns.

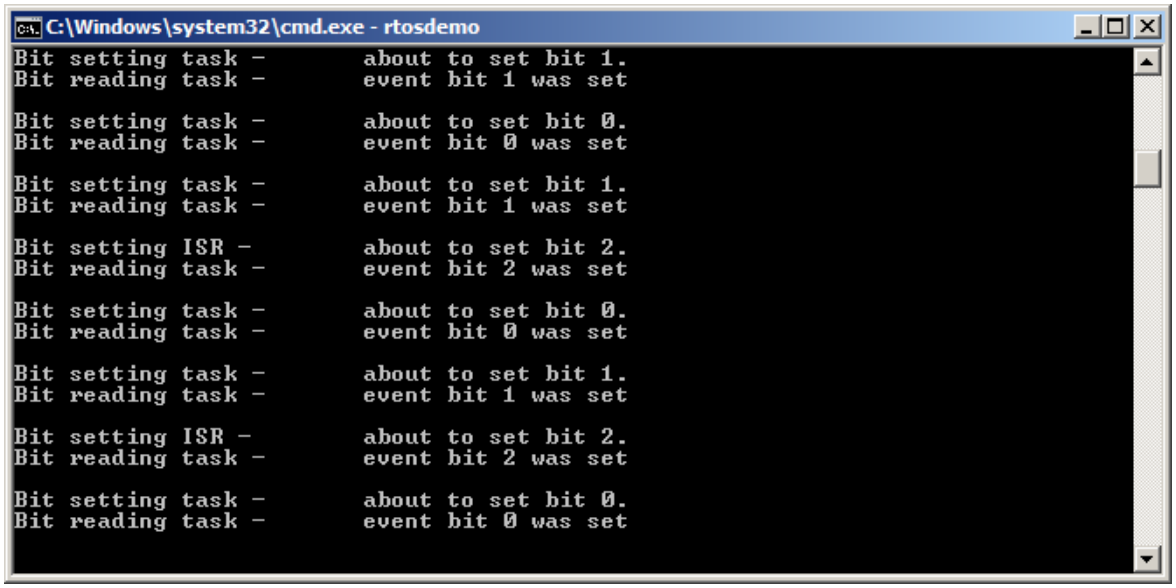
```
static void vEventBitReadingTask( void *pvParameters )

{

    EventBits_t xEventGroupValue;

    const EventBits_t xBitsToWaitFor = ( mainFIRST_TASK_BIT | mainSECOND_TASK_BIT |
 mainISR_BIT );

    for( ;; )

    {

        /* Block to wait for event bits to become set within the event group.*/

        xEventGroupValue = xEventGroupWaitBits( /* The event group to read. */
 xEventGroup, /* Bits to test. */  xBitsToWaitFor,  /* Clear bits on exit if the unblock
 condition is met. */ pdTRUE, /* Don't wait for all bits. This parameter is set to pdTRUE
 for the second execution. */ pdFALSE, /* Don't time out. */ portMAX_DELAY );

        /* Print a message for each bit that was set. */

        if( ( xEventGroupValue & mainFIRST_TASK_BIT ) != 0 )

        {

            vPrintString( "Bit reading task -\t Event bit 0 was set\r\n" );

        }

        if( ( xEventGroupValue & mainSECOND_TASK_BIT ) != 0 )

        {

            vPrintString( "Bit reading task -\t Event bit 1 was set\r\n" );

        }

        if( ( xEventGroupValue & mainISR_BIT ) != 0 )

        {

            vPrintString( "Bit reading task -\t Event bit 2 was set\r\n" );

        }

    }
```

```
}
```

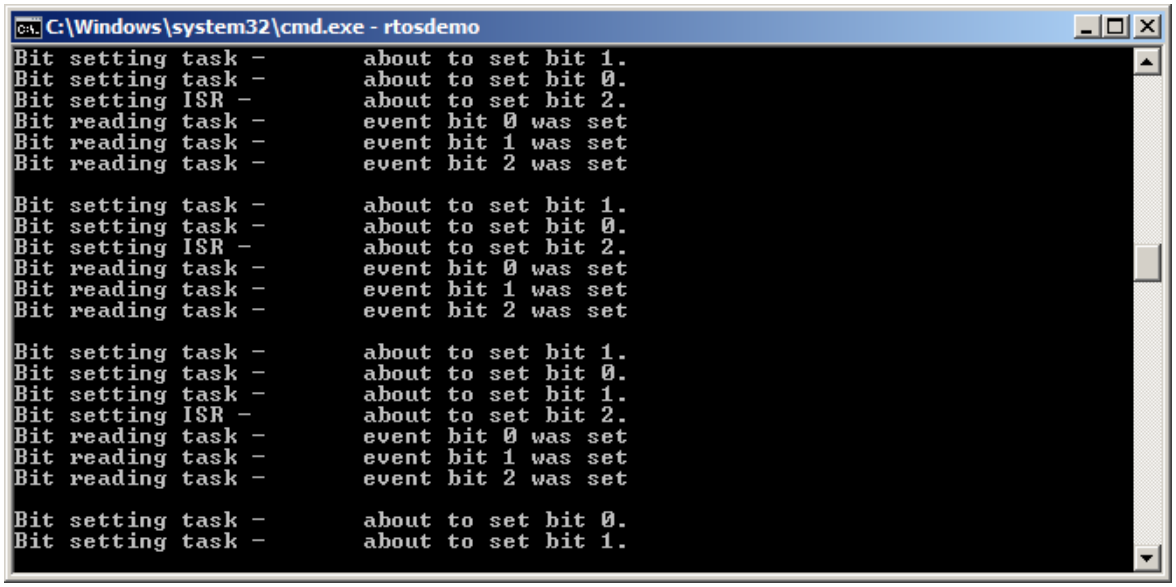
The main() function creates the event group and tasks before starting the scheduler. The following code shows its implementation. The priority of the task that reads from the event group is higher than the priority of the task that writes to the event group, ensuring the reading task will preempt the writing task each time the reading task's unblock condition is met.

```
int main( void )

{

    /* Before an event group can be used it must first be created. */

    xEventGroup = xEventGroupCreate();

    /* Create the task that sets event bits in the event group. */

    xTaskCreate( vEventBitSettingTask, "Bit Setter", 1000, NULL, 1, NULL );

    /* Create the task that waits for event bits to get set in the event group. */

    xTaskCreate( vEventBitReadingTask, "Bit Reader", 1000, NULL, 2, NULL );

    /* Create the task that is used to periodically generate a software interrupt. */

    xTaskCreate( vInterruptGenerator, "Int Gen", 1000, NULL, 3, NULL );

    /* Install the handler for the software interrupt. The syntax required to do this is
 depends on the FreeRTOS port being used. The syntax shown here can only be used with the
 FreeRTOS Windows port, where such interrupts are only simulated. */

    vPortSetInterruptHandler( mainINTERRUPT_NUMBER, ulEventBitSettingISR );

    /* Start the scheduler so the created tasks start executing. */

    vTaskStartScheduler();

    /* The following line should never be reached. */

    for( ;; );

    return 0;

}
```

The output produced when Example 22 is executed with the xEventGroupWaitBits() xWaitForAllBits parameter set to pdFALSE is shown here. You'll see that because the xWaitForAllBits parameter in the call to xEventGroupWaitBits() was set to pdFALSE, the task that reads from the event group leaves the Blocked state and executes immediately every time any of the event bits are set.



The output produced when the code is executed with the xEventGroupWaitBits() xWaitForAllBits parameter set to pdTRUE is shown here. You'll see that because the xWaitForAllBits parameter was set to pdTRUE, the task that reads from the event group leaves the Blocked state only after all three of the event bits are set.



Task Synchronization Using an Event Group

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

You'll find a less abstract example of the need for this type of task synchronization in one of the FreeRTOS+TCP demonstration projects. The demonstration shares a TCP socket between two tasks. One task sends data to the socket, and a different task receives data from the same socket. (This is currently the only way a single FreeRTOS+TCP socket can be shared between tasks.) It is not safe for either task to close the TCP socket until it is sure the other task will not attempt to access the socket again. If either of the two tasks wants to close the socket, then it must inform the other task of its intent, and then wait for the other task to stop using the socket before proceeding.

The scenario in which it is the task that sends data to the socket that wants to close the socket is trivial because there are only two tasks that need to synchronize with each other. It is easy to see how the scenario would become more complex and require more tasks to join the synchronization if other tasks were performing processing that was dependent on the socket being open.

```
void SocketTxTask( void *pvParameters )

{

        xSocket_t xSocket;

        uint32_t ulTxCount = 0UL;

        for( ;; )

        {

            /* Create a new socket. This task will send to this socket, and another task
 will receive from this socket. */

            xSocket = FreeRTOS_socket( ... );

            /* Connect the socket. */

            FreeRTOS_connect( xSocket, ... );

            /* Use a queue to send the socket to the task that receives data. */

            xQueueSend( xSocketPassingQueue, &xSocket, portMAX_DELAY );

            /* Send 1000 messages to the socket before closing the socket. */

            for( ulTxCount = 0; ulTxCount < 1000; ulTxCount++ )

            {

                if( FreeRTOS_send( xSocket, ... ) < 0 )

                    {

                            /* Unexpected error - exit the loop, after which the socket
 will be closed. */

                            break;

                    }

             }

            /* Let the Rx task know the Tx task wants to close the socket. */

            TxTaskWantsToCloseSocket();

            /* This is the Tx task's synchronization point. The Tx task waits here
 for the Rx task to reach its synchronization point. The Rx task will only reach its
```

```
synchronization point when it is no longer using the socket, and the socket can be closed
safely. */

           xEventGroupSync( ... );

           /* Neither task is using the socket. Shut down the connection, and then close
the socket. */

           FreeRTOS_shutdown( xSocket, ... );

           WaitForSocketToDisconnect();

           FreeRTOS_closesocket( xSocket );

       }

   }

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

   void SocketRxTask( void *pvParameters )

   {

       xSocket_t xSocket;

       for( ;; )

       {

           /* Wait to receive a socket that was created and connected by the Tx task. */

           xQueueReceive( xSocketPassingQueue, &xSocket, portMAX_DELAY );

           /* Keep receiving from the socket until the Tx task wants to close the socket.
*/

           while( TxTaskWantsToCloseSocket() == pdFALSE )

           {

               /* Receive then process data. */

               FreeRTOS_recv( xSocket, ... );

               ProcessReceivedData();

           }

           /* This is the Rx task's synchronization point. It reaches here only when it is
no longer using the socket, and it is therefore safe for the Tx task to close the socket.
*/

           xEventGroupSync( ... );

       }

   }
```

The preceding pseudo code shows two tasks that synchronize with each other to ensure a shared TCP socket is no longer in use by either task before the socket is closed.

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

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

The xEventGroupSetBits() and xEventGroupWaitBits() API functions cannot be used in this scenario. If they were used, then the setting of a bit (to indicate a task had reached its synchronization point) and the testing of bits (to determine if the other synchronizing tasks had reached their synchronization point) would be performed as two separate operations. To see why that would be a problem, consider a scenario where Task A, Task B, and Task C attempt to synchronize using an event group:

1. Task A and Task B have already reached the synchronization point, so their event bits are set in the event group. They are in the Blocked state to wait for task C's event bit to be set.
2. Task C reaches the synchronization point and uses xEventGroupSetBits() to set its bit in the event group. As soon as Task C's bit is set, Task A and Task B leave the Blocked state and clear all three event bits.
3. Task C then calls xEventGroupWaitBits() to wait for all three event bits to be set, but by that time, all three event bits have been cleared, Task A and Task B have left their respective synchronization points, and so the synchronization has failed.

To successfully use an event group to create a synchronization point, the setting of an event bit and the subsequent testing of event bits must be performed as a single uninterruptable operation. The xEventGroupSync() API function is provided for that purpose.

xEventGroupSync() API Function

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

The xEventGroupSync() uxBitsToWaitFor parameter specifies the calling task's unblock condition. The event bits specified by uxBitsToWaitFor will be cleared back to zero before xEventGroupSync() returns, if xEventGroupSync() returned because the unblock condition had been met.

The xEventGroupSync() API function prototype is shown here.

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

The following table lists the xEventGroupSync() parameters and return value.

Parameter Name	Description
xEventGroup	The handle of the event group in which event bits are to be set, and then tested. The event group handle will have been returned from the call to xEventGroupCreate() used to create the event group.
uxBitsToSet	A bit mask that specifies the event bit, or event bits, to set to 1 in the event group. The value of the event group is updated by bitwise ORing the

	event group's existing value with the value passed in uxBitsToSet. As an example, setting uxBitsToSet to 0x04 (binary 0100) will result in event bit 3 becoming set (if it was not already set), while leaving all the other event bits in the event group unchanged.
uxBitsToWaitFor	A bit mask that specifies the event bit, or event bits, to test in the event group. For example, if the calling task wants to wait for event bits 0, 1 and 2 to become set in the event group, then set uxBitsToWaitFor to 0x07 (binary 111).
xTicksToWait	The maximum amount of time the task should remain in the Blocked state to wait for its unblock condition to be met. xEventGroupSync() will return immediately if xTicksToWait is zero, or the unblock condition is met at the time xEventGroupSync() is called. The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.
Returned Value	If xEventGroupSync() returned because the calling task's unblock condition was met, then the returned value is the value of the event group at the time the calling task's unblock condition was met (before any bits were automatically cleared back to zero). In this case, the returned value will also meet the calling task's unblock condition. If xEventGroupSync() returned because the block time specified by the xTicksToWait parameter expired, then the returned value is the value of the event group at the time the block time expired. In this case, the returned value will not meet the calling task's unblock condition.

Synchronizing Tasks (Example 23)

The following code shows how to synchronize three instances of a single task implementation. The task parameter is used to pass into each instance the event bit the task will set when it calls xEventGroupSync().

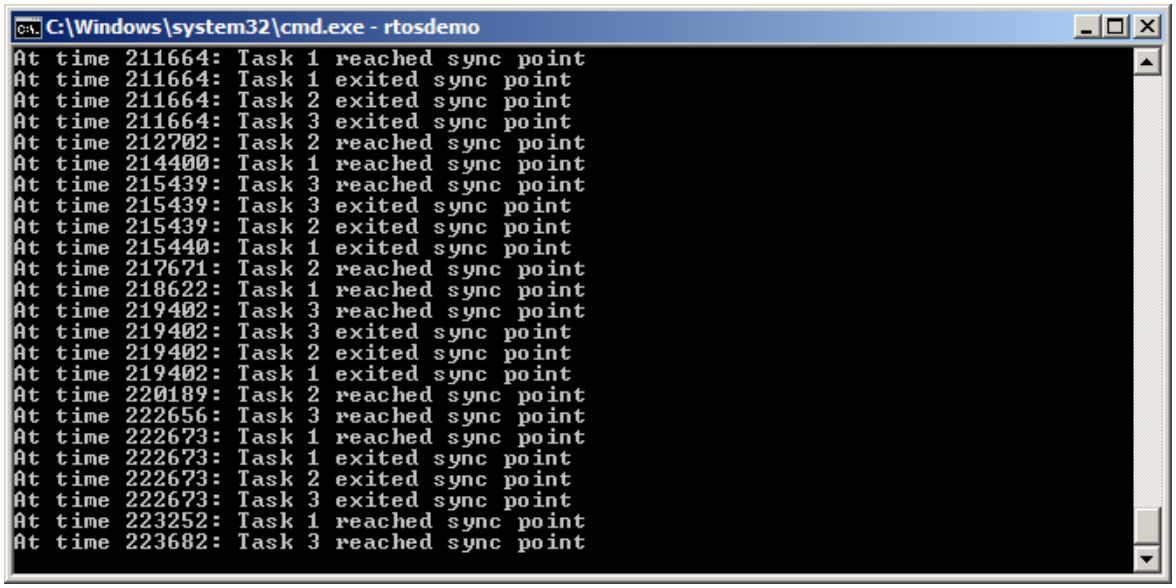
The task prints a message before calling xEventGroupSync() and again after the call to xEventGroupSync() has returned. Each message includes a timestamp. This allows the sequence of execution to be observed in the output produced. A pseudo-random delay is used to prevent all the tasks reaching the synchronization point at the same time.

```
static void vSyncingTask( void *pvParameters )

{

    const TickType_t xMaxDelay = pdMS_TO_TICKS( 4000UL );

    const TickType_t xMinDelay = pdMS_TO_TICKS( 200UL );

    TickType_t xDelayTime;

    EventBits_t uxThisTasksSyncBit;

    const EventBits_t uxAllSyncBits = ( mainFIRST_TASK_BIT | mainSECOND_TASK_BIT |
 mainTHIRD_TASK_BIT );

    /* Three instances of this task are created - each task uses a different event bit in
 the synchronization. The event bit to use is passed into each task instance using the task
 parameter. Store it in the uxThisTasksSyncBit variable. */

    uxThisTasksSyncBit = ( EventBits_t ) pvParameters;

    for( ;; )

    {

        /* Simulate this task taking some time to perform an action by delaying for a
 pseudo-random time. This prevents all three instances of this task from reaching  the
 synchronization point at the same time, and so allows the example's behavior to be
 observed more easily. */

        xDelayTime = ( rand() % xMaxDelay ) + xMinDelay;

        vTaskDelay( xDelayTime );

        /* Print out a message to show this task has reached its synchronization  point.
 pcTaskGetTaskName() is an API function that returns the name assigned to the task when the
 task was created. */

        vPrintTwoStrings( pcTaskGetTaskName( NULL ), "reached sync point" );

        /* Wait for all the tasks to have reached their respective synchronization points.
 */

        xEventGroupSync( /* The event group used to synchronize. */ xEventGroup, /*
 The bit set by this task to indicate it has reached the synchronization point. */
 uxThisTasksSyncBit, /* The bits to wait for, one bit for each task taking part in the
 synchronization. */ uxAllSyncBits, /* Wait indefinitely for all three tasks to reach the
 synchronization point. */ portMAX_DELAY );

        /* Print out a message to show this task has passed its synchronization point. As
 an indefinite delay was used the following line will only be executed after all the tasks
 reached their respective synchronization  points. */

        vPrintTwoStrings( pcTaskGetTaskName( NULL ), "exited sync point" );

    }

}
```

The main() function creates the event group, creates all three tasks, and then starts the scheduler. The following code shows its implementation.

```
/* Definitions for the event bits in the event group. */

#define mainFIRST_TASK_BIT ( 1UL << 0UL ) /* Event bit 0, set by the first task. */

#define mainSECOND_TASK_BIT( 1UL << 1UL ) /* Event bit 1, set by the second task. */

#define mainTHIRD_TASK_BIT ( 1UL << 2UL ) /* Event bit 2, set by the third task. */

/* Declare the event group used to synchronize the three tasks. */

EventGroupHandle_t xEventGroup;

int main( void )

{

        /* Before an event group can be used it must first be created. */
        xEventGroup = xEventGroupCreate();

        /* Create three instances of the task. Each task is given a different name, which
 is later printed out to give a visual indication of which task is executing. The event bit
 to use when the task reaches its synchronization point is passed into the task using the
 task parameter. */

        xTaskCreate( vSyncingTask, "Task 1", 1000, mainFIRST_TASK_BIT, 1, NULL);

        xTaskCreate( vSyncingTask, "Task 2", 1000, mainSECOND_TASK_BIT, 1, NULL );

        xTaskCreate( vSyncingTask, "Task 3", 1000, mainTHIRD_TASK_BIT, 1, NULL );

        /* Start the scheduler so the created tasks start executing. */

        vTaskStartScheduler();

        /* As always, the following line should never be reached. */

        for( ;; );

        return 0;

 }
```

The output produced when Example 23 is executed is shown here. You'll see that even though each task reaches the synchronization point at a different (pseudo-random) time, each task exits the synchronization point at the same time (which is the time at which the last task reached the synchronization point). The figure shows the example running in the FreeRTOS Windows port, which does not provide true real-time behavior (especially when using Windows system calls to print to the console). Therefore, there will be some timing variation.


C:\Windows\system32\cmd.exe - rtosdemo
At time 211664: Task 1 reached sync point
At time 211664: Task 1 exited sync point
At time 211664: Task 2 exited sync point
At time 211664: Task 3 exited sync point
At time 212702: Task 2 reached sync point
At time 214400: Task 1 reached sync point
At time 215439: Task 3 reached sync point
At time 215439: Task 3 exited sync point
At time 215439: Task 2 exited sync point
At time 215440: Task 1 exited sync point
At time 217671: Task 2 reached sync point
At time 218622: Task 1 reached sync point
At time 219402: Task 3 reached sync point
At time 219402: Task 3 exited sync point
At time 219402: Task 2 exited sync point
At time 219402: Task 1 exited sync point
At time 220189: Task 2 reached sync point
At time 222656: Task 3 reached sync point
At time 222673: Task 1 reached sync point
At time 222673: Task 1 exited sync point
At time 222673: Task 2 exited sync point
At time 222673: Task 3 exited sync point
At time 223252: Task 1 reached sync point
At time 223682: Task 3 reached sync point

Task Notifications

This section covers:

- A task's notification state and notification value.
- How and when a task notification can be used in place of a communication object, such as a semaphore.
- The advantages of using a task notification in place of a communication object.

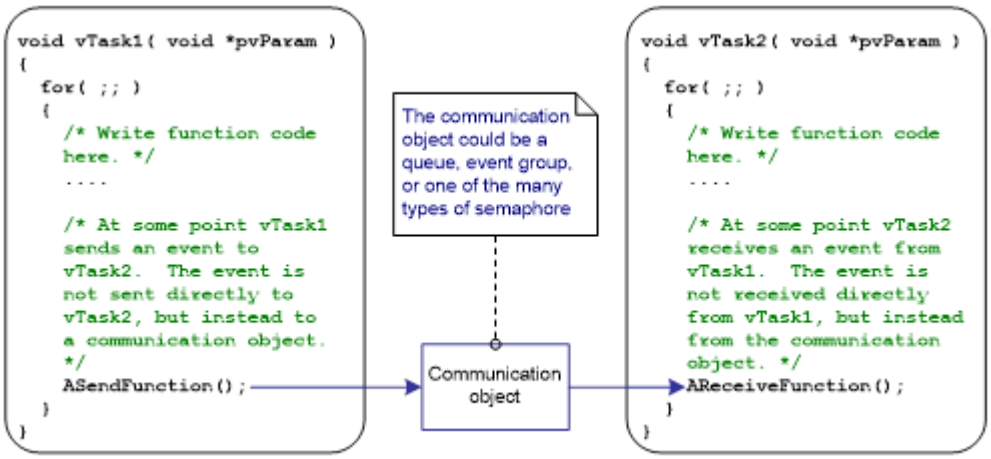
Applications that use FreeRTOS are structured as a set of independent tasks and these autonomous tasks have to communicate with each other so that, collectively, they can provide useful system functionality.

Communicating Through Intermediary Objects

The methods in which tasks can communicate with each other have so far required the creation of a communication object like a queues, event group, and different types of semaphore.

When you use a communication object, events and data are not sent directly to a receiving task or receiving ISR, but to the communication object. Likewise, tasks and ISRs receive events and data from the communication object rather than directly from the task or ISR that sent the event or data.

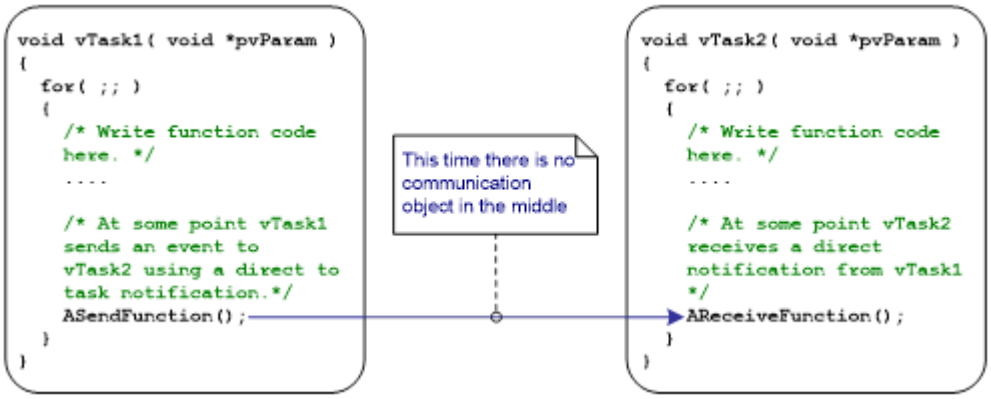
This figure shows a communication object being used to send an event from one task to another.



Task Notifications: Direct-to-Task Communication

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

This figure shows a task notification used to send an event directly from one task to another.



Task notification functionality is optional. To include task notification functionality, in FreeRTOSConfig.h, set configUSE_TASK_NOTIFICATIONS to 1.

When configUSE_TASK_NOTIFICATIONS is set to 1, each task has a notification state, which can be either pending or not-pending, and a notification value, which is a 32-bit unsigned integer. When a task receives a notification, its notification state is set to pending. When a task reads its notification value, its notification state is set to not-pending.

A task can wait in the Blocked state, with an optional timeout, for its notification state to become pending.

Benefits and Limitations of Task Notifications

Using a task notification to send an event or data to a task is significantly faster than using a queue, semaphore, or event group.

Likewise, using a task notification to send an event or data to a task requires significantly less RAM than using a queue, semaphore, or event group. This is because each communication object (queue, semaphore, or event group) must be created before it can be used, whereas enabling task notification functionality has a fixed overhead of just eight bytes of RAM per task.

Limitations of Task Notifications

Task notifications cannot be used in these scenarios:

- Sending an event or data to an ISR.

  Communication objects can be used to send events and data from an ISR to a task, and from a task to an ISR.

  Task notifications can be used to send events and data from an ISR to a task, but they cannot be used to send events or data from a task to an ISR.
- Enabling more than one receiving task.

  A communication object can be accessed by any task or ISR that knows its handle (which might be a queue handle, semaphore handle, or event group handle). Any number of tasks and ISRs can process events or data sent to any given communication object.

  Task notifications are sent directly to the receiving task, so can be processed only by the task to which the notification is sent. This is rarely a limitation in most cases because, while it is common to have

multiple tasks and ISRs sending to the same communication object, it is rare to have multiple tasks and ISRs receiving from the same communication object.

- Buffering multiple data items.

  A queue is a communication object that can hold more than one data item at a time. Data that has been sent to the queue, but not yet received from the queue, is buffered inside the queue object.

  Task notifications send data to a task by updating the receiving task's notification value. A task's notification value can hold only one value at a time.
- Broadcasting to more than one task.

  An event group is a communication object that can be used to send an event to more than one task at a time.

  Task notifications are sent directly to the receiving task, so can only be processed by the receiving task.
- Waiting in the blocked state for a send to complete.

  If a communication object is temporarily in a state that means no more data or events can be written to it (for example, when a queue is full no more data can be sent to the queue), then tasks attempting to write to the object can optionally enter the Blocked state to wait for their write operation to complete.

  If a task attempts to send a task notification to a task that already has a notification pending, then it is not possible for the sending task to wait in the Blocked state for the receiving task to reset its notification state. This is rarely a limitation in most cases in which a task notification is used.

Using Task Notifications

Task notifications are a very powerful feature that can often be used in place of a binary semaphore, a counting semaphore, an event group, and sometimes even a queue.

Task Notification API Options

You can use the xTaskNotify() API function to send a task notification and the xTaskNotifyWait() API function to receive a task notification.

However, in most cases, the flexibility provided by the xTaskNotify() and xTaskNotifyWait() API functions is not required. A simpler functions would suffice. The xTaskNotifyGive() API function is a simpler, but less flexible alternative to xTaskNotify(). The ulTaskNotifyTake() API function is a simpler, but less flexible alternative to xTaskNotifyWait().

xTaskNotifyGive() API Function

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

The xTaskNotifyGive() API function is provided to allow a task notification to be used as a lightweight and faster alternative to a binary or counting semaphore. The xTaskNotifyGive() API function prototype is shown here.

```
BaseType_t xTaskNotifyGive( TaskHandle_t xTaskToNotify );
```

The following table lists the xTaskNotifyGive() parameters and return value.

Parameter Name/ Returned Value	Description
xTaskToNotify	The handle of the task to which the notification is being sent. For information about obtaining handles to task, see the pxCreatedTask parameter of the xTaskCreate() API function.
Returned value	xTaskNotifyGive() is a macro that calls xTaskNotify(). The parameters passed into xTaskNotify() by the macro are set such that pdPASS is the only possible return value. xTaskNotify() is described later in this section.

vTaskNotifyGiveFromISR() API Function

vTaskNotifyGiveFromISR() is a version of xTaskNotifyGive() that can be used in an interrupt service routine. The vTaskNotifyGiveFromISR() API function prototype is shown here.

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

The following table lists the vTaskNotifyGiveFromISR() parameters and return value.

Parameter Name/ Returned Value	Description
xTaskToNotify	The handle of the task to which the notification is being sent. For information about obtaining handles to tasks, see the pxCreatedTask parameter of the xTaskCreate() API function.
pxHigherPriorityTaskWoken	If the task to which the notification is being sent is waiting in the Blocked state to receive a notification, then sending the notification will cause the task to leave the Blocked state. If calling vTaskNotifyGiveFromISR() causes a task to leave the Blocked state, and the unblocked task has a priority higher than the priority of the currently executing task (the task that was interrupted), then, internally, vTaskNotifyGiveFromISR() will set <problematic>*</problematic> pxHigherPriorityTaskWoken to pdTRUE. If vTaskNotifyGiveFromISR() 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. As with all interrupt-safe API functions, the pxHigherPriorityTaskWoken parameter must be set to pdFALSE before it is used.

ulTaskNotifyTake() API Function

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

The ulTaskNotifyTake() API function is provided to allow a task notification to be used as a lightweight and faster alternative to a binary or counting semaphore. The ulTaskNotifyTake() API function prototype is shown here.

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

The following table list the ulTaskNotifyTake() parameters and return value.

Parameter Name/ Returned Value	Description
xClearCountOnExit	If xClearCountOnExit is set to pdTRUE, then the calling task's notification value will be cleared to zero before the call to ulTaskNotifyTake() returns. If xClearCountOnExit is set to pdFALSE, and the calling task's notification value is greater than zero, then the calling task's notification value will be decremented before the call to ulTaskNotifyTake() returns.
xTicksToWait	The maximum amount of time the calling task should remain in the Blocked state to wait for its notification value to be greater than zero. The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds to ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.
Returned value	The returned value is the calling task's notification value before it was either cleared to zero or decremented, as specified by the value of the xClearCountOnExit parameter. If a block time was specified (xTicksToWait was not zero), and the return value is not zero, then it is possible the calling task was placed into the Blocked state to wait for its notification value to be greater than zero, and its notification value was updated before the block time expired. If a block time was specified (xTicksToWait was not zero), and the return value is zero, then the calling task was placed into the Blocked state, to wait for its notification value to be greater than

	zero, but the specified block time expired before that happened.

Method 1 for Using a Task Notification in Place of a Semaphore (Example 24)

Example 16 used a binary semaphore to unblock a task from within an interrupt service routine, effectively synchronizing the task with the interrupt. This example replicates the functionality of Example 16, but uses a direct-to-task notification in place of the binary semaphore.

The following code shows the implementation of the task that is synchronized with the interrupt. The call to xSemaphoreTake() that was used in Example 16 has been replaced by a call to ulTaskNotifyTake().

The ulTaskNotifyTake() xClearCountOnExit parameter is set to pdTRUE, which results in the receiving task's notification value being cleared to zero before ulTaskNotifyTake() returns. It is therefore necessary to process all the events that are already available between each call to ulTaskNotifyTake(). In Example 16, because a binary semaphore was used, the number of pending events had to be determined from the hardware, which is not always practical. In this example, the number of pending events is returned from ulTaskNotifyTake().

Interrupt events that occur between calls to ulTaskNotifyTake are latched in the task's notification value, and calls to ulTaskNotifyTake() will return immediately if the calling task already has notifications pending.

```
/* The rate at which the periodic task generates software interrupts.*/

const TickType_t xInterruptFrequency = pdMS_TO_TICKS( 500UL );

static void vHandlerTask( void *pvParameters )

{

    /* xMaxExpectedBlockTime is set to be a little longer than the maximum expected time
 between events. */

    const TickType_t xMaxExpectedBlockTime = xInterruptFrequency + pdMS_TO_TICKS( 10 );

    uint32_t ulEventsToProcess;

    /* As per most tasks, this task is implemented within an infinite loop. */

    for( ;; )

    {

        /* Wait to receive a notification sent directly to this task from the interrupt
 service routine. */

        ulEventsToProcess = ulTaskNotifyTake( pdTRUE, xMaxExpectedBlockTime );

        if( ulEventsToProcess != 0 )

        {

            /* To get here at least one event must have occurred. Loop here until all
 the pending events have been processed (in this case, just print out a message for each
 event). */
```

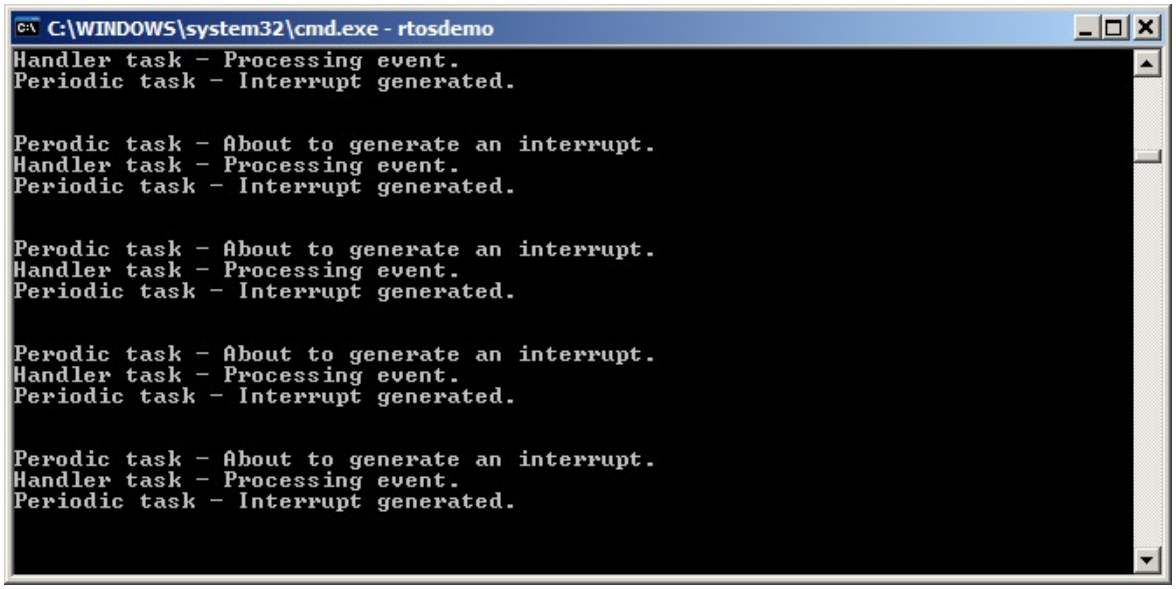
```
            while( ulEventsToProcess > 0 )

            {

                vPrintString( "Handler task - Processing event.\r\n" );

                ulEventsToProcess--;

            }

        }

        else

        {

            /* If this part of the function is reached, then an interrupt did not arrive
 within the expected time. In a real application, it might be necessary to perform some
 error recovery operations. */

        }

    }

}
```

The periodic task used to generate software interrupts prints a message before and after the interrupt is generated. This allows the sequence of execution to be observed in the output.

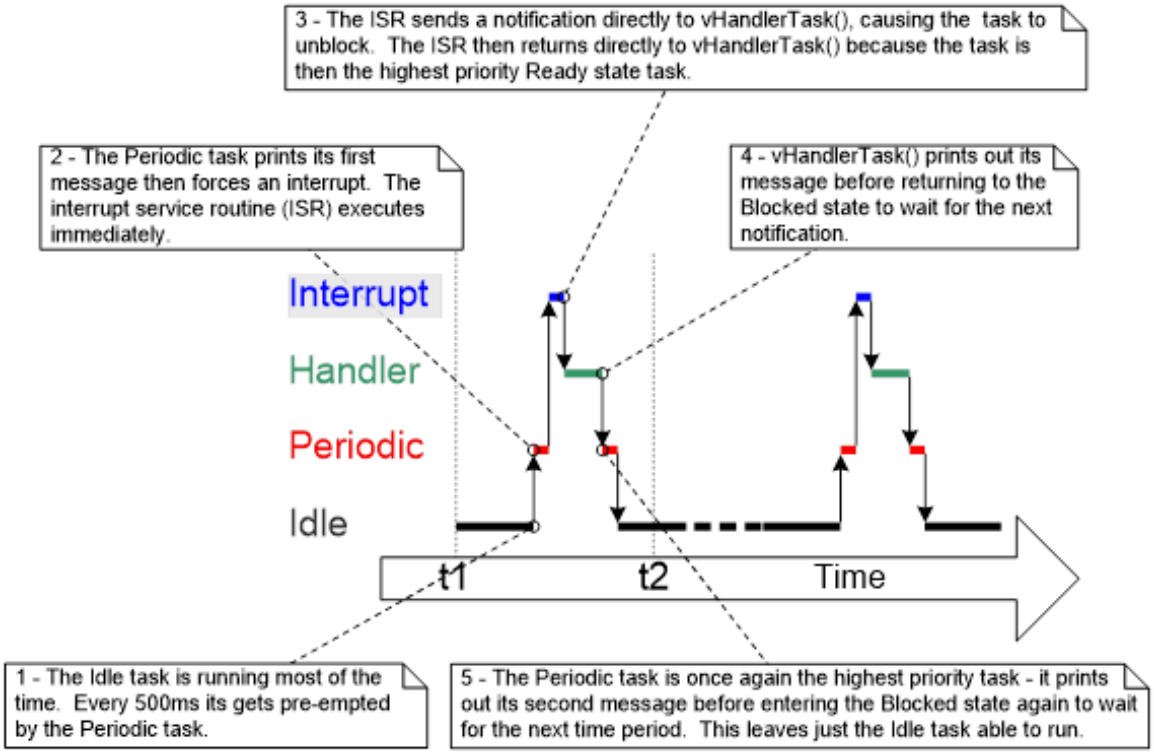
The following code shows the interrupt handler. This does little more than send a notification directly to the task to which interrupt handling is deferred.

```
static uint32_t ulExampleInterruptHandler( void )

{

    BaseType_t xHigherPriorityTaskWoken;

    /* The xHigherPriorityTaskWoken parameter must be initialized to pdFALSE because it
 will get set to pdTRUE inside the interrupt-safe API function if a context switch is
 required. */

    xHigherPriorityTaskWoken = pdFALSE;

    /* Send a notification directly to the task to which interrupt processing is being
 deferred. */

    vTaskNotifyGiveFromISR( /* The handle of the task to which the notification is
 being sent. The handle was saved when the task was created. */ xHandlerTask,  /*
 xHigherPriorityTaskWoken is used in the usual way. */ &xHigherPriorityTaskWoken );

    /* Pass the xHigherPriorityTaskWoken value into portYIELD_FROM_ISR(). If
 xHigherPriorityTaskWoken was set to pdTRUE inside vTaskNotifyGiveFromISR(),  then calling
 portYIELD_FROM_ISR() will request a context switch. If xHigherPriorityTaskWoken is still
 pdFALSE, then calling portYIELD_FROM_ISR() will have no effect. The implementation of
 portYIELD_FROM_ISR() used by the Windows port includes a return statement, which is why
 this function does not explicitly return a value. */

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

The output produced when the code is executed is shown here.



As expected, it is identical to the output produced when Example 16 is executed. vHandlerTask() enters the Running state as soon as the interrupt is generated, so the output from the task splits the output produced by the periodic task. The sequence of execution is shown here.



Method 2 for Using a Task Notification in Place of a Semaphore (Example 25)

In Example 24, the ulTaskNotifyTake() xClearOnExit parameter was set to pdTRUE. Example 25 modifies Example 24 slightly to demonstrate the behavior when the ulTaskNotifyTake() xClearOnExit parameter is set to pdFALSE.

When xClearOnExit is pdFALSE, calling ulTaskNotifyTake() will only decrement (reduce by one) the calling task's notification value, instead of clearing it to zero. The notification count is therefore the difference between the number of events that have occurred and the number of events that have been processed. That allows the structure of vHandlerTask() to be simplified in two ways:

1. The number of events waiting to be processed is held in the notification value, so it does not need to be held in a local variable.
2. It is only necessary to process one event between each call to ulTaskNotifyTake().

The implementation of vHandlerTask() is shown here.

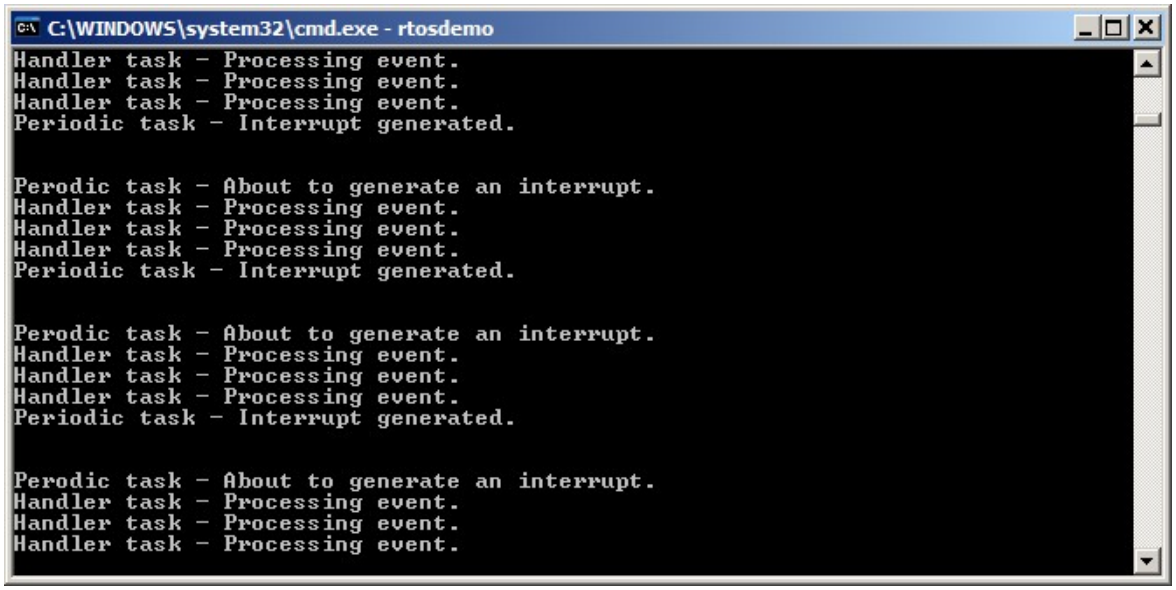
```
static void vHandlerTask( void *pvParameters )

{

    /* xMaxExpectedBlockTime is set to be a little longer than the maximum expected time
 between events. */

    const TickType_t xMaxExpectedBlockTime = xInterruptFrequency + pdMS_TO_TICKS( 10 );

    /* As per most tasks, this task is implemented within an infinite loop. */

    for( ;; )

    {

        /* Wait to receive a notification sent directly to this task from the interrupt
 service routine. The xClearCountOnExit parameter is now pdFALSE, so the task's
 notification value will be decremented by ulTaskNotifyTake() and not cleared to zero. */

        if( ulTaskNotifyTake( pdFALSE, xMaxExpectedBlockTime ) != 0 )

        {

            /* To get here, an event must have occurred. Process the event (in this case,
 just print out a message). */

            vPrintString( "Handler task - Processing event.\r\n" );

        }

        else

        {

             /* If this part of the function is reached, then an interrupt did not arrive
 within the expected time. In a real application, it might be necessary to perform some
 error recovery operations. */

        }

    }
```

```
}
```

For demonstration purposes, the interrupt service routine has also been modified to send more than one task notification per interrupt, and in so doing, simulate multiple interrupts occurring at high frequency. The implementation of the interrupt service routine is shown here.

```
static uint32_t ulExampleInterruptHandler(void)

{

    BaseType_t xHigherPriorityTaskWoken;

    xHigherPriorityTaskWoken = pdFALSE;

    /* Send a notification to the handler task multiple times. The first give will unblock
 the task. The following gives are to demonstrate that the receiving task's notification
 value is being used to count (latch) events, allowing the task to process each event in
 turn. */

    vTaskNotifyGiveFromISR(xHandlerTask, &xHigherPriorityTaskWoken);

    vTaskNotifyGiveFromISR(xHandlerTask, &xHigherPriorityTaskWoken);

        vTaskNotifyGiveFromISR(xHandlerTask, &xHigherPriorityTaskWoken);

    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);

}
```

The output is shown here. You'll see vHandlerTask() processes all three events each time an interrupt is generated.



xTaskNotify() and xTaskNotifyFromISR() API Functions

xTaskNotify() is a more capable version of xTaskNotifyGive() that can be used to update the receiving task's notification value in any of the following ways:

- Increment (add one to) the receiving task's notification value, in which case xTaskNotify() is equivalent to xTaskNotifyGive().
- Set one or more bits in the receiving task's notification value. This allows a task's notification value to be used as a lightweight and faster alternative to an event group.
- Write a completely new number into the receiving task's notification value, but only if the receiving task has read its notification value since it was last updated. This allows a task's notification value to provide similar functionality to that provided by a queue that has a length of one.
- Write a completely new number into the receiving task's notification value, even if the receiving task has not read its notification value since it was last updated. This allows a task's notification value to provide similar functionality to that provided by the xQueueOverwrite() API function. The resultant behavior is sometimes referred to as a *mailbox*.

xTaskNotify() is more flexible and powerful than xTaskNotifyGive(). It is also a little more complex to use.

xTaskNotifyFromISR() is a version of xTaskNotify() that can be used in an interrupt service routine. It therefore has an additional pxHigherPriorityTaskWoken parameter.

Calling xTaskNotify() will always set the receiving task's notification state to pending, if it was not already pending.

```
BaseType_t xTaskNotify( TaskHandle_t xTaskToNotify, uint32_t ulValue, eNotifyAction
 eAction );
```

```
BaseType_t xTaskNotifyFromISR( TaskHandle_t xTaskToNotify, uint32_t ulValue, eNotifyAction
 eAction, BaseType_t *pxHigherPriorityTaskWoken );
```

The following table lists the xTaskNotify() parameters and return value.

Parameter Name/ Returned Value	Description
xTaskToNotify	The handle of the task to which the notification is being sent. For information about obtaining handles to tasks, see the pxCreatedTask parameter of the xTaskCreate() API function.
ulValue	How ulValue is used is dependent on the eNotifyAction value. See Table 52.
eNotifyAction	An enumerated type that specifies how to update the receiving task's notification value. See the fol.
Returned value	xTaskNotify() will return pdPASS except in the one case noted in the following.

The following table lists valid xTaskNotify() eNotifyAction parameter values and their resultant effect on the receiving task's notification value.

eNotifyAction Value	Resultant Effect on Receiving Task
eNoAction	The receiving task's notification state is set to pending without its notification value being updated. The xTaskNotify() ulValue parameter is not used. The eNoAction action allows a task notification to be used as a faster and lightweight alternative to a binary semaphore.
eSetBits	The receiving task's notification value is bitwise OR'ed with the value passed in the xTaskNotify() ulValue parameter. For example, if ulValue is set to 0x01, then bit 0 will be set in the receiving task's notification value. As another example, if ulValue is 0x06 (binary 0110), then bit 1 and bit 2 will be set in the receiving task's notification value. The eSetBits action allows a task notification to be used as a faster and lightweight alternative to an event group.
eIncrement	The receiving task's notification value is incremented. The xTaskNotify() ulValue parameter is not used. The eIncrement action allows a task notification to be used as a faster and lightweight alternative to a binary or counting semaphore. It is equivalent to the simpler xTaskNotifyGive() API function.
eSetValueWithoutOverwrite	If the receiving task had a notification pending before xTaskNotify() was called, then no action is taken and xTaskNotify() will return pdFAIL. If the receiving task did not have a notification pending before xTaskNotify() was called, then the receiving task's notification value is set to the value passed in the xTaskNotify() ulValue parameter.
eSetValueWithOverwrite	The receiving task's notification value is set to the value passed in the xTaskNotify() ulValue parameter, regardless of whether the receiving task had a notification pending before xTaskNotify() was called.

xTaskNotifyWait() API Function

xTaskNotifyWait() is a more capable version of ulTaskNotifyTake(). It allows a task to wait, with an optional timeout, for the calling task's notification state to become pending, if it is not already be pending. xTaskNotifyWait() provides options for bits to be cleared in the calling task's notification value both on entry to the function, and on exit from the function.

```
BaseType_t xTaskNotifyWait( uint32_t ulBitsToClearOnEntry, uint32_t ulBitsToClearOnExit,
 uint32_t *pulNotificationValue, TickType_t xTicksToWait );
```

The following table lists the xTaskNotifyWait() parameters and return value.

Parameter Name/ Returned Value	Description
ulBitsToClearOnEntry	If the calling task did not have a notification pending before it called xTaskNotifyWait(), then any bits set in ulBitsToClearOnEntry will be cleared in the task's notification value on entry to the function. For example, if ulBitsToClearOnEntry is 0x01, then bit 0 of the task's notification value will be cleared. As another example, setting ulBitsToClearOnEntry to 0xffffffff (ULONG_MAX) will clear all the bits in the task's notification value, effectively clearing the value to 0.
ulBitsToClearOnExit	If the calling task exits xTaskNotifyWait() because it received a notification, or because it already had a notification pending when xTaskNotifyWait() was called, then any bits set in ulBitsToClearOnExit will be cleared in the task's notification value before the task exits the xTaskNotifyWait() function. The bits are cleared after the task's notification value has been saved in <problematic>*</problematic> pulNotificationValue (see the description of pulNotificationValue below). For example, if ulBitsToClearOnExit is 0x03, then bit 0 and bit 1 of the task's notification value will be cleared before the function exits. Setting ulBitsToClearOnExit to 0xffffffff (ULONG_MAX) will clear all the bits in the task's notification value, effectively clearing the value to 0.
pulNotificationValue	Used to pass out the task's notification value. The value copied to <problematic>*</problematic> pulNotificationValue is the task's notification value as it was before any bits were cleared due to the ulBitsToClearOnExit setting. pulNotificationValue is an optional parameter and can be set to NULL if it is not required.
xTicksToWait	The maximum amount of time the calling task should remain in the Blocked state to wait for its notification state to become pending.

	The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds to a time specified in ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.
Returned value	There are two possible return values: 1. pdTRUE This indicates xTaskNotifyWait() returned because a notification was received, or because the calling task already had a notification pending when xTaskNotifyWait() was called. If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state, to wait for its notification state to become pending, but its notification state was set to pending before the block time expired. 1. pdFALSE This indicates that xTaskNotifyWait() returned without the calling task receiving a task notification. If xTicksToWait was not zero, then the calling task will have been held in the Blocked state to wait for its notification state to become pending, but the specified block time expired before that happened.

Task Notifications Used in Peripheral Device Drivers: UART Example

Peripheral driver libraries provide functions that perform common operations on hardware interfaces. Peripherals for these libraries include Universal Asynchronous Receivers and Transmitters (UARTs), Serial Peripheral Interface (SPI) ports, analog-to-digital converters (ADCs), and Ethernet ports. Functions commonly provided by these libraries include functions to initialize a peripheral, send data to a peripheral, and receive data from a peripheral.

Some operations on peripherals take a relatively long time to complete. These include a high precision ADC conversion and the transmission of a large data packet on a UART. In these cases, the driver library function can be implemented to poll (repeatedly read) the peripheral's status registers to determine when the operation is complete. However, polling in this way is almost always wasteful because it uses 100% of the processor's time while no productive processing is being performed. The waste is

particularly expensive in a multitasking system, where a task that is polling a peripheral might be preventing the execution of a lower priority task that does have productive processing to perform.

To avoid the potential for wasted processing time, an efficient RTOS-aware device driver should be interrupt-driven and give a task that initiates a lengthy operation the option of waiting in the Blocked state for the operation to complete. That way, lower priority tasks can execute while the task performing the lengthy operation is in the Blocked state, and no tasks use processing time unless they can use it productively.

It is common practice for RTOS-aware driver libraries to use a binary semaphore to place tasks into the Blocked state. The technique is demonstrated by the following pseudo code, which provides the outline of an RTOS-aware library function that transmits data on a UART port. In the following code listing:

- xUART is a structure that describes the UART peripheral and holds state information. The xTxSemaphore member of the structure is a variable of type SemaphoreHandle_t. It is assumed the semaphore has already been created.
- The xUART_Send() function does not include any mutual exclusion logic. If more than one task is going to use the xUART_Send() function, then the application writer will have to manage mutual exclusion within the application itself. For example, a task might be required to obtain a mutex before calling xUART_Send().
- The xSemaphoreTake() API function is used to place the calling task into the Blocked state after the UART transmission has been initiated.
- The xSemaphoreGiveFromISR() API function is used to remove the task from the Blocked state after the transmission has completed, which is when the UART peripheral's transmit end interrupt service routine executes.

```
/* Driver library function to send data to a UART. */

BaseType_t xUART_Send( xUART *pxUARTInstance, uint8_t *pucDataSource, size_t uxLength )

{

    BaseType_t xReturn;

    /* Ensure the UART's transmit semaphore is not already available by attempting to take
 the semaphore without a timeout. */

    xSemaphoreTake( pxUARTInstance->xTxSemaphore, 0 );

    /* Start the transmission. */

    UART_low_level_send( pxUARTInstance, pucDataSource, uxLength );

    /* Block on the semaphore to wait for the transmission to complete. If the semaphore is
 obtained, then xReturn will get set to pdPASS. If the semaphore take operation times out,
 then xReturn will get set to pdFAIL. If the interrupt occurs between UART_low_level_send()
 being called and xSemaphoreTake() being called, then the event will be latched in the
 binary semaphore, and the call to xSemaphoreTake() will return immediately. */

    xReturn = xSemaphoreTake( pxUARTInstance->xTxSemaphore, pxUARTInstance->xTxTimeout );

    return xReturn;

}

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

/* The service routine for the UART's transmit end interrupt, which executes after the last
 byte has been sent to the UART. */
```

```
void xUART_TransmitEndISR( xUART *pxUARTInstance )

{

    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Clear the interrupt. */

    UART_low_level_interrupt_clear( pxUARTInstance );

    /* Give the Tx semaphore to signal the end of the transmission. If a task is Blocked
 waiting for the semaphore, then the task will be removed from the Blocked state. */

    xSemaphoreGiveFromISR( pxUARTInstance->xTxSemaphore, &xHigherPriorityTaskWoken );

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

The technique demonstrated in this code is workable and commonly used, but it has some drawbacks:

- The library uses multiple semaphores, which increases its RAM footprint.
- Semaphores cannot be used until they have been created, so a library that uses semaphores cannot be used until it has been explicitly initialized.
- Semaphores are generic objects that are applicable to a wide range of use cases. They include logic to allow any number of tasks to wait in the Blocked state for the semaphore to become available and to select (in a deterministic manner) which task to remove from the Blocked state when the semaphore becomes available. Executing that logic takes a finite time. That processing overhead is unnecessary in the scenario shown in this code, in which there cannot be more than one task waiting for the semaphore at any given time.

The following code demonstrates how to avoid these drawbacks by using a task notification in place of a binary semaphore.

Note: If a library uses task notifications, then the library's documentation must clearly state that calling a library function can change the calling task's notification state and value.

In the following code:

- The xTxSemaphore member of the xUART structure has been replaced by the xTaskToNotify member. xTaskToNotify is a variable of type TaskHandle_t, and is used to hold the handle of the task that is waiting for the UART operation to complete.
- The xTaskGetCurrentTaskHandle() FreeRTOS API function is used to obtain the handle of the task that is in the Running state.
- The library does not create any FreeRTOS objects, so does not incur a RAM overhead, and does not need to be explicitly initialized.
- The task notification is sent directly to the task that is waiting for the UART operation to complete, so no unnecessary logic is executed.

The xTaskToNotify member of the xUART structure is accessed from both a task and an interrupt service routine, which requires consideration of how the processor will update its value:

- If xTaskToNotify is updated by a single memory write operation, then it can be updated outside of a critical section, exactly as shown in the following code. This would be the case if xTaskToNotify is a 32-bit variable (TaskHandle_t was a 32-bit type), and the processor on which FreeRTOS is running is a 32-bit processor.

- If more than one memory write operation is required to update xTaskToNotify, then xTaskToNotify must only be updated from within a critical section. Otherwise, the interrupt service routine might access xTaskToNotify while it is in an inconsistent state. This would be the case if xTaskToNotify is a 32-bit variable, and the processor on which FreeRTOS is running is a 16-bit processor, because it would require two 16-bit memory write operations to update all 32-bits.

Internally, within the FreeRTOS implementation, TaskHandle_t is a pointer, so sizeof(TaskHandle_t) always equals sizeof(void *).

```
/* Driver library function to send data to a UART. */

BaseType_t xUART_Send( xUART *pxUARTInstance, uint8_t *pucDataSource, size_t uxLength )

{

    BaseType_t xReturn;

    /* Save the handle of the task that called this function. The book text contains notes
 as to whether the following line needs to be protected by a critical section or not. */

    pxUARTInstance->xTaskToNotify = xTaskGetCurrentTaskHandle();

    /* Ensure the calling task does not already have a notification pending by calling
 ulTaskNotifyTake() with the xClearCountOnExit parameter set to pdTRUE, and a block time of
 0  (don't block). */

    ulTaskNotifyTake( pdTRUE, 0 );

    /* Start the transmission. */

    UART_low_level_send( pxUARTInstance, pucDataSource, uxLength );

    /* Block until notified that the transmission is complete. If the notification is
 received, then xReturn will be set to 1 because the ISR will have incremented this task's
 notification value to 1 (pdTRUE). If the operation times out, then xReturn will be 0
 (pdFALSE) because this task's notification value will not have been changed since it was
 cleared to 0 above. If the ISR executes between the calls to UART_low_level_send() and
 the call to ulTaskNotifyTake(), then the event will be latched in the task's notification
 value, and the call to ulTaskNotifyTake() will return immediately.*/

    xReturn = ( BaseType_t ) ulTaskNotifyTake( pdTRUE, pxUARTInstance->xTxTimeout );

    return xReturn;

}

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

/* The ISR that executes after the last byte has been sent to the UART. */

void xUART_TransmitEndISR( xUART *pxUARTInstance )

{

    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* This function should not execute unless there is a task waiting to be notified.
 Test this condition with an assert. This step is not strictly necessary, but will aid
 debugging. configASSERT() is described in Developer Support.*/

    configASSERT( pxUARTInstance->xTaskToNotify != NULL );

    /* Clear the interrupt. */
```

```
    UART_low_level_interrupt_clear( pxUARTInstance );

    /* Send a notification directly to the task that called xUART_Send(). If the task
 is Blocked waiting for the notification, then the task will be removed from the Blocked
 state. */

    vTaskNotifyGiveFromISR( pxUARTInstance->xTaskToNotify, &xHigherPriorityTaskWoken );

    /* Now there are no tasks waiting to be notified. Set the xTaskToNotify member of the
 xUART structure back to NULL. This step is not strictly necessary, but will aid debugging.
 */

    pxUARTInstance->xTaskToNotify = NULL;

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

Task notifications can also replace semaphores in receive functions, as demonstrated in the following pseudo code, which provides the outline of an RTOS-aware library function that receives data on a UART port.

- The xUART_Receive() function does not include any mutual exclusion logic. If more than one task is going to use the xUART_Receive() function, then the application writer will have to manage mutual exclusion within the application itself. For example, a task might be required to obtain a mutex before calling xUART_Receive().
- The UART's receive interrupt service routine places the characters that are received by the UART into a RAM buffer. The xUART_Receive() function returns characters from the RAM buffer.
- The xUART_Receive() uxWantedBytes parameter is used to specify the number of characters to receive. If the RAM buffer does not already contain the requested number characters, then the calling task is placed into the Blocked state to wait to be notified that the number of characters in the buffer has increased. The while() loop is used to repeat this sequence until either the receive buffer contains the requested number of characters or a timeout occurs.
- The calling task can enter the Blocked state more than once. The block time is therefore adjusted to take into account the amount of time that has already passed since xUART_Receive() was called. The adjustments ensure the total time spent inside xUART_Receive() does not exceed the block time specified by the xRxTimeout member of the xUART structure. The block time is adjusted using the FreeRTOS vTaskSetTimeOutState() and xTaskCheckForTimeOut() helper functions.

```
/* Driver library function to receive data from a UART. */

size_t xUART_Receive( xUART *pxUARTInstance, uint8_t *pucBuffer, size_t uxWantedBytes )

{

    size_t uxReceived = 0;

    TickType_t xTicksToWait;

    TimeOut_t xTimeOut;

    /* Record the time at which this function was entered. */

    vTaskSetTimeOutState( &xTimeOut );

    /* xTicksToWait is the timeout value. It is initially set to the maximum receive
 timeout for this UART instance. */

    xTicksToWait = pxUARTInstance->xRxTimeout;
```

```
    /* Save the handle of the task that called this function. */

    pxUARTInstance->xTaskToNotify = xTaskGetCurrentTaskHandle();

    /* Loop until the buffer contains the wanted number of bytes or a timeout occurs. */

    while( UART_bytes_in_rx_buffer( pxUARTInstance ) < uxWantedBytes )

    {

        /* Look for a timeout, adjusting xTicksToWait to account for the time spent in this
 function so far. */

        if( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) != pdFALSE )

        {

            /* Timed out before the wanted number of bytes were available, exit the loop.
 */

            break;

        }

        /* The receive buffer does not yet contain the required amount of bytes. Wait for
 a maximum of xTicksToWait ticks to be notified that the receive interrupt service routine
 has placed more data into the buffer. It does not matter if the calling task already had a
 notification pending when it called this function. If it did, it would just iterate around
 this while loop one extra time. */

        ulTaskNotifyTake( pdTRUE, xTicksToWait );

    }

    /* No tasks are waiting for receive notifications, so set xTaskToNotify back to NULL.
 */

    pxUARTInstance->xTaskToNotify = NULL;

    /* Attempt to read uxWantedBytes from the receive buffer into pucBuffer. The actual
 number of bytes read (which might be less than uxWantedBytes) is returned. */

    uxReceived = UART_read_from_receive_buffer( pxUARTInstance, pucBuffer, uxWantedBytes );

    return uxReceived;

}

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

/* The interrupt service routine for the UART's receive interrupt */

void xUART_ReceiveISR( xUART *pxUARTInstance )

{

    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Copy received data into this UART's receive buffer and clear the interrupt. */

    UART_low_level_receive( pxUARTInstance );

    /* If a task is waiting to be notified of the new data, then notify it now. */

    if( pxUARTInstance->xTaskToNotify != NULL )
```

```
    {

            vTaskNotifyGiveFromISR( pxUARTInstance->xTaskToNotify,
 &xHigherPriorityTaskWoken );

            portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

    }
}
```

Task Notifications Used in Peripheral Device Drivers: ADC Example

The previous section demonstrated how to use vTaskNotifyGiveFromISR() to send a task notification from an interrupt to a task. vTaskNotifyGiveFromISR() is a simple function to use, but its capabilities are limited. It can only send a task notification as a valueless event. It cannot send data. This section demonstrates how to use xTaskNotifyFromISR() to send data with a task notification event. The technique is demonstrated by the following pseudo code, which provides the outline of an RTOS-aware interrupt service routine for an analog-to-digital converter (ADC).

- It is assumed an ADC conversion is started at least every 50 milliseconds.
- ADC_ConversionEndISR() is the interrupt service routine for the ADC's conversion end interrupt, which is the interrupt that executes each time a new ADC value is available.
- The task implemented by vADCTask() processes each value generated by the ADC. It is assumed the task's handle was stored in xADCTaskToNotify when the task was created.
- ADC_ConversionEndISR() uses xTaskNotifyFromISR() with the eAction parameter set to eSetValueWithoutOverwrite to send a task notification to the vADCTask() task and write the result of the ADC conversion into the task's notification value.
- The vADCTask() task uses xTaskNotifyWait() to wait to be notified that a new ADC value is available and to retrieve the result of the ADC conversion from its notification value.

```
/* A task that uses an ADC. */

void vADCTask( void *pvParameters )

{

    uint32_t ulADCValue;

    BaseType_t xResult;

    /* The rate at which ADC conversions are triggered. */

    const TickType_t xADCConversionFrequency = pdMS_TO_TICKS( 50 );

    for( ;; )

    {

        /* Wait for the next ADC conversion result. */

        xResult = xTaskNotifyWait( /* The new ADC value will overwrite the old value, so
 there is no need to clear any bits before waiting for the new notification value. */ 0, /*
 Future ADC values will overwrite the existing value, so there is no need to clear any bits
 before exiting xTaskNotifyWait(). */  0, /* The address of the variable into which the
```

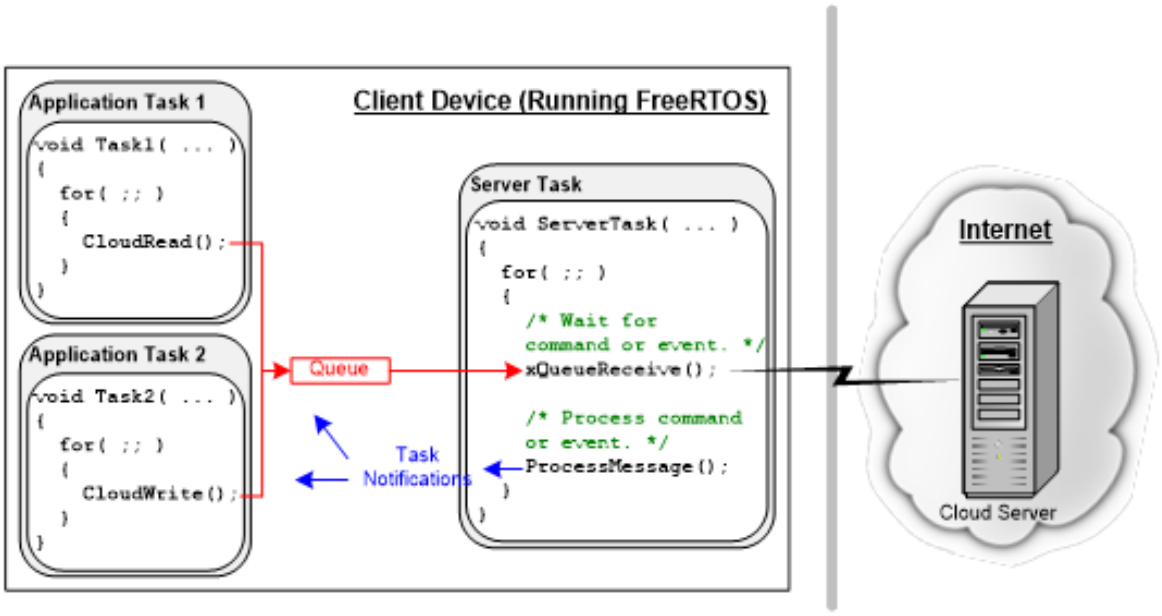
```
 task's notification value (which holds the latest ADC conversion result) will be copied.
 */ &ulADCValue, /* A new ADC value should be received every xADCConversionFrequency ticks.
 */ xADCConversionFrequency * 2 );

        if( xResult == pdPASS )

        {

            /* A new ADC value was received. Process it now. */

            ProcessADCResult( ulADCValue );

        }

        else

        {

            /* The call to xTaskNotifyWait() did not return within the expected time.
 Something must be wrong with the input that triggers the ADC conversion or with the ADC
 itself. Handle the error here. */

        }

    }

}

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

/* The interrupt service routine that executes each time an ADC conversion completes. */

void ADC_ConversionEndISR( xADC *pxADCInstance )

{

    uint32_t ulConversionResult;

    BaseType_t xHigherPriorityTaskWoken = pdFALSE, xResult;

    /* Read the new ADC value and clear the interrupt. */

    ulConversionResult = ADC_low_level_read( pxADCInstance );

    /* Send a notification, and the ADC conversion result, directly to vADCTask(). */

    xResult = xTaskNotifyFromISR( xADCTaskToNotify, /* xTaskToNotify parameter. */
 ulConversionResult, /* ulValue parameter. */ eSetValueWithoutOverwrite, /* eAction
 parameter. */ &xHigherPriorityTaskWoken );

    /* If the call to xTaskNotifyFromISR() returns pdFAIL then the task is not keeping
 up with the rate at which ADC values are being generated. configASSERT() is described in
 section 11.2.*/

    configASSERT( xResult == pdPASS );

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

Task Notifications Used Directly Within an Application

This section demonstrates the use of task notifications in a hypothetical application that includes the following functionality:

1. The application communicates across a slow internet connection to send data to, and request data from, a remote data server (the *cloud server*).
2. After requesting data from the cloud server, the requesting task must wait in the Blocked state for the requested data to be received.
3. After sending data to the cloud server, the sending task must wait in the Blocked state for an acknowledgement that the cloud server received the data correctly.

The software design is shown here.



- The complexity of handling multiple internet connections to the cloud server is encapsulated in a single FreeRTOS task. The task acts as a proxy server within the FreeRTOS application and is referred to as the *server task*.
- Application tasks read data from the cloud server by calling CloudRead(). CloudRead() does not communicate with the cloud server directly. Instead, it sends the read request to the server task on a queue and receives the requested data from the server task as a task notification.
- Application tasks write date to the cloud server by calling CloudWrite(). CloudWrite() does not communicate with the cloud server directly. Instead, it sends the write request to the server task on a queue and receives the result of the write operation from the server task as a task notification.

The following code shows the structure sent to the server task by the CloudRead() and CloudWrite() functions.

```
typedef enum CloudOperations

{
```

```
    eRead, /* Send data to the cloud server. */

    eWrite /* Receive data from the cloud server. */

} Operation_t;

typedef struct CloudCommand

{

    Operation_t eOperation; /* The operation to perform (read or write). */

    uint32_t ulDataID; /* Identifies the data being read or written. */

    uint32_t ulDataValue; /* Only used when writing data to the cloud server. */

    TaskHandle_t xTaskToNotify;/* The handle of the task performing the operation. */

} CloudCommand_t;
```

The pseudo code for CloudRead() is shown here. The function sends its request to the server task, and then calls xTaskNotifyWait() to wait in the Blocked state until it is notified that the requested data is available.

```
/* ulDataID identifies the data to read. pulValue holds the address of the variable into
 which the data received from the cloud server is to be written. */

BaseType_t CloudRead( uint32_t ulDataID, uint32_t *pulValue )

{

    CloudCommand_t xRequest;

    BaseType_t xReturn;

    /* Set the CloudCommand_t structure members to be correct for this read request. */

    xRequest.eOperation = eRead; /* This is a request to read data. */

    xRequest.ulDataID = ulDataID; /* A code that identifies the data to read. */

    xRequest.xTaskToNotify = xTaskGetCurrentTaskHandle(); /* Handle of the calling task. */

    /* Ensure there are no notifications already pending by reading the notification value
 with a block time of 0, and then send the structure to the server task. */

    xTaskNotifyWait( 0, 0, NULL, 0 );

    xQueueSend( xServerTaskQueue, &xRequest, portMAX_DELAY );

    /* Wait for a notification from the server task. The server task writes the value
 received from the cloud server directly into this task's notification value, so there
 is no need to clear any bits in the notification value on entry to or exit from the
 xTaskNotifyWait() function. The received value is written to *pulValue, so pulValue is
 passed as the address to which the notification value is written. */

    xReturn = xTaskNotifyWait( 0, /* No bits cleared on entry. */  0, /* No bits to clear
 on exit. */ pulValue, /* Notification value into *pulValue. */  pdMS_TO_TICKS( 250 ) );
            /* Wait a maximum of 250ms. */
            /* If xReturn is pdPASS, then the value was obtained. If xReturn is pdFAIL,
 then the request timed out. */

    return xReturn;
```

```
}
```

The pseudo code that shows how the server task manages a read request is shown here. When the data has been received from the cloud server, the server task unblocks the application task and sends the received data to the application task by calling xTaskNotify() with the eAction parameter set to eSetValueWithOverwrite.

This is a simplified scenario because it assumes GetCloudData() does not have to wait to obtain a value from the cloud server.

```
void ServerTask( void *pvParameters )

{

    CloudCommand_t xCommand;

    uint32_t ulReceivedValue;

    for( ;; )

    {

        /* Wait for the next CloudCommand_t structure to be received from a task. */

        xQueueReceive( xServerTaskQueue, &xCommand, portMAX_DELAY );

        switch( xCommand.eOperation ) /* Was it a read or write request? */

        {

            case eRead:

            /* Obtain the requested data item from the remote cloud server. */

            ulReceivedValue = GetCloudData( xCommand.ulDataID );

            /* Call xTaskNotify() to send both a notification and the value received from
 the cloud server to the task that made the request. The handle of the task is obtained
 from the CloudCommand_t structure. */

            xTaskNotify( xCommand.xTaskToNotify, /* The task's handle is in the structure.
 */ ulReceivedValue, /* Cloud data sent as notification value. */ eSetValueWithOverwrite );

            break;

            /* Other switch cases go here. */

        }

    }

}
```

The pseudo code for CloudWrite() is shown here. For the purpose of demonstration, CloudWrite() returns a bitwise status code, where each bit in the status code is assigned a unique meaning. Four example status bits are shown by the #define statements at the top.

The task clears the four status bits, sends its request to the server task, and then calls xTaskNotifyWait() to wait in the Blocked state for the status notification.

```
/* Status bits used by the cloud write operation. */
```

```
#define SEND_SUCCESSFUL_BIT ( 0x01 << 0 )

#define OPERATION_TIMED_OUT_BIT ( 0x01 << 1

#define NO_INTERNET_CONNECTION_BIT ( 0x01 << 2 )

#define CANNOT_LOCATE_CLOUD_SERVER_BIT ( 0x01 << 3 )

/* A mask that has the four status bits set. */

#define CLOUD_WRITE_STATUS_BIT_MASK ( SEND_SUCCESSFUL_BIT \|

OPERATION_TIMED_OUT_BIT \|

NO_INTERNET_CONNECTION_BIT \|

CANNOT_LOCATE_CLOUD_SERVER_BIT )

uint32_t CloudWrite( uint32_t ulDataID, uint32_t ulDataValue )

{

    CloudCommand_t xRequest;

    uint32_t ulNotificationValue;

    /* Set the CloudCommand_t structure members to be correct for this write request. */

    xRequest.eOperation = eWrite; /* This is a request to write data. */

    xRequest.ulDataID = ulDataID; /* A code that identifies the data being written. */

    xRequest.ulDataValue = ulDataValue; /* Value of the data written to the cloud server.
 */

    xRequest.xTaskToNotify = xTaskGetCurrentTaskHandle(); /* Handle of the calling task. */

    /* Clear the three status bits relevant to the write operation by
 calling xTaskNotifyWait() with the ulBitsToClearOnExit parameter set to
 CLOUD_WRITE_STATUS_BIT_MASK, and a block time of 0. The current notification value is not
 required, so the pulNotificationValue parameter is set to NULL. */

    xTaskNotifyWait( 0, CLOUD_WRITE_STATUS_BIT_MASK, NULL, 0 );

    /* Send the request to the server task. */

    xQueueSend( xServerTaskQueue, &xRequest, portMAX_DELAY );

    /* Wait for a notification from the server task. The server task writes a bitwise
 status code into this task's notification value, which is written to ulNotificationValue.
 */

    xTaskNotifyWait( 0, /* No bits cleared on entry. */ CLOUD_WRITE_STATUS_BIT_MASK, /
* Clear relevant bits to 0 on exit. */ &ulNotificationValue, /* Notified value. */
 pdMS_TO_TICKS( 250 ) ); /* Wait a maximum of 250ms. */ /* Return the status code to the
 calling task. */ return ( ulNotificationValue & CLOUD_WRITE_STATUS_BIT_MASK );
```

The pseudo code that demonstrates how the server task manages a write request is shown here. When the data has been sent to the cloud server, the server task unblocks the application task and sends the bitwise status code to the application task, by calling xTaskNotify() with the eAction parameter set to eSetBits. Only the bits defined by the CLOUD_WRITE_STATUS_BIT_MASK constant can get altered in the receiving task's notification value, so the receiving task can use other bits in its notification value for other purposes.

This is a simplified scenario because it assumes SetCloudData() does not have to wait to obtain an acknowledgement from the remote cloud server.

```
void ServerTask( void *pvParameters )

{

    CloudCommand_t xCommand;

    uint32_t ulBitwiseStatusCode;

    for( ;; )

    {

        /* Wait for the next message. */

        xQueueReceive( xServerTaskQueue, &xCommand, portMAX_DELAY );

        /* Was it a read or write request? */

        switch( xCommand.eOperation )

        {

            case eWrite:

            /* Send the data to the remote cloud server. SetCloudData() returns a bitwise
 status code that only uses the bits defined by the CLOUD_WRITE_STATUS_BIT_MASK definition
 (shown in the preceding code). */

            ulBitwiseStatusCode = SetCloudData( xCommand.ulDataID, xCommand.ulDataValue );

            /* Send a notification to the task that made the write request. The eSetBits
 action is used so any status bits set in ulBitwiseStatusCode will be set in the
 notification value of the task being notified. All the other bits remain unchanged. The
 handle of the task is obtained from the CloudCommand_t structure. */

            xTaskNotify( xCommand.xTaskToNotify, /* The task's handle is in the structure.
 */ ulBitwiseStatusCode, /* Cloud data sent as notification value. */ eSetBits );

            break;

            /* Other switch cases go here. */

        }

    }

}
```

Developer Support

This section covers features that maximize productivity by:

- Providing insight into how an application is behaving.
- Highlighting opportunities for optimization.
- Trapping errors at the point at which they occur.

configASSERT()

In C, the macro assert() is used to verify an *assertion* (assumption) made by the program. The assertion is written as a C expression. If the expression evaluates to false (0), then the assertion is deemed to have failed. For example, this code tests the assertion that the pointer pxMyPointer is not NULL.

```
/* Test the assertion that pxMyPointer is not NULL */

assert( pxMyPointer != NULL );
```

The application writer specifies the action to take if an assertion fails by providing an implementation of the assert() macro.

The FreeRTOS source code does not call assert(), because assert() is not available with all the compilers with which FreeRTOS is compiled. Instead, the FreeRTOS source code contains lots of calls to a macro called configASSERT(), which can be defined by the application writer in FreeRTOSConfig.h, and behaves exactly like the standard C assert().

A failed assertion must be treated as a fatal error. Do not attempt to execute past a line that has failed an assertion.

Using configASSERT() improves productivity by immediately trapping and identifying many of the most common sources of error. We strongly recommend that you define configASSERT() while you are developing or debugging a FreeRTOS application.

Defining configASSERT() will help you with runtime debugging, but it will also increase the application code size and therefore slow down its execution. If a definition of configASSERT() is not provided, then the default empty definition will be used, and all the calls to configASSERT() will be completely removed by the C preprocessor.

Example configASSERT() Definitions

The definition of configASSERT() shown in the following code is useful when an application is being executed under the control of a debugger. It will halt execution on any line that fails an assertion, so the line that failed the assertion will be displayed by the debugger when the debug session is paused.

```
/* Disable interrupts so the tick interrupt stops executing, and then sit in a loop so
 execution does not move past the line that failed the assertion. If the hardware supports
```

```
 a debug break instruction, then the debug break instruction can be used in place of the
 for() loop. */

#define configASSERT( x ) if( ( x ) == 0 ) { taskDISABLE_INTERRUPTS();

for(;;); }
```

The definition of configASSERT() is useful when an application is not being executed under the control of a debugger. It prints out or otherwise records the source code line that failed an assertion. You can identify the line that failed the assertion by using the standard C __FILE__ macro to obtain the name of the source file and the standard C __LINE__ macro to obtain the line number in the source file.

This code shows a configASSERT() definition that records the source code line that failed an assertion.

Tracealyzer

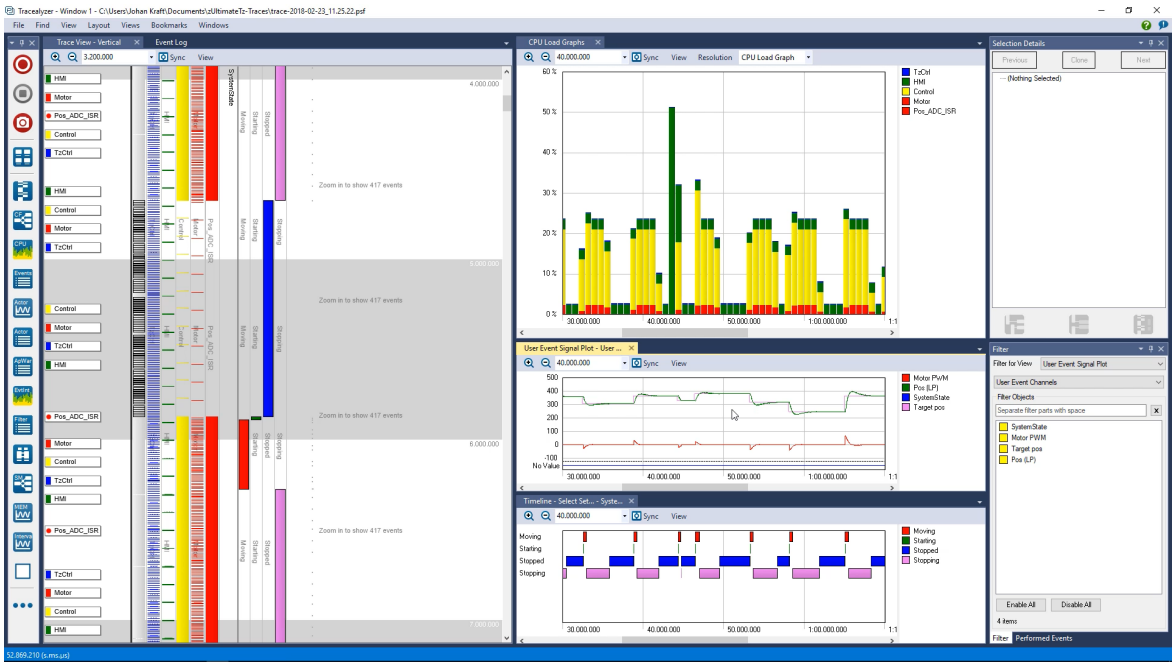
Tracealyzer is a runtime diagnostic and optimization tool provided by our partner company, Percepio.

Tracealyzer captures valuable dynamic behavior information and presents it in interconnected graphical views. The tool can also display multiple synchronized views.

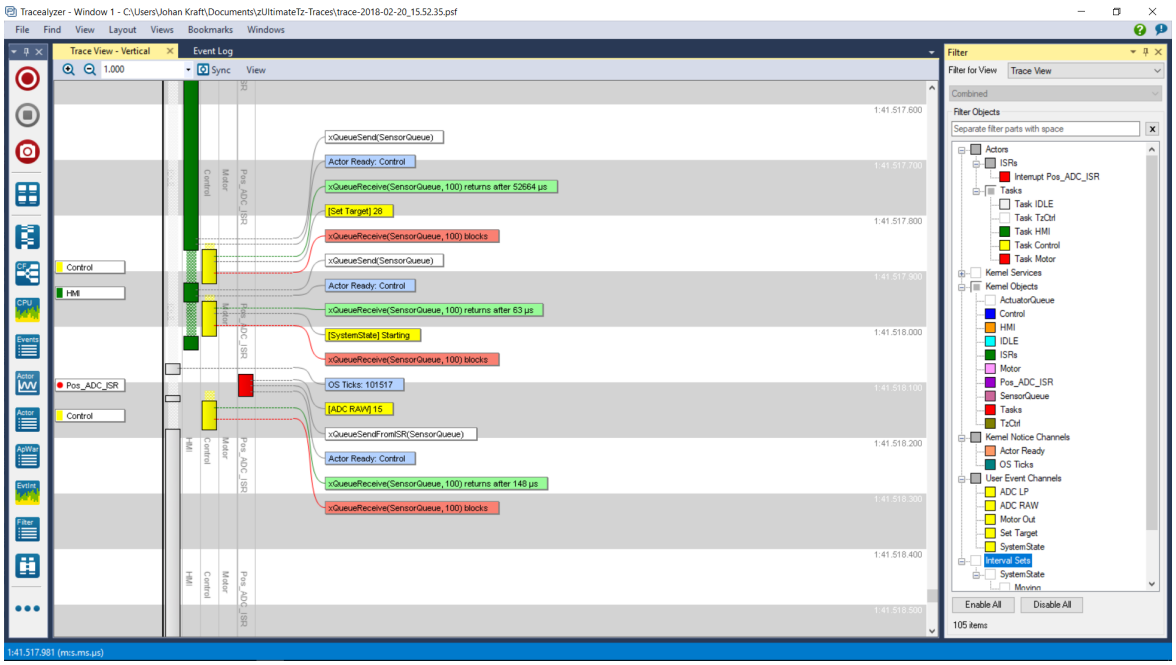
The captured information is valuable when analyzing, troubleshooting, or simply optimizing a FreeRTOS application.

Tracealyzer can be used side by side with a traditional debugger. It complements the debugger's view with a higher level, time-based perspective.

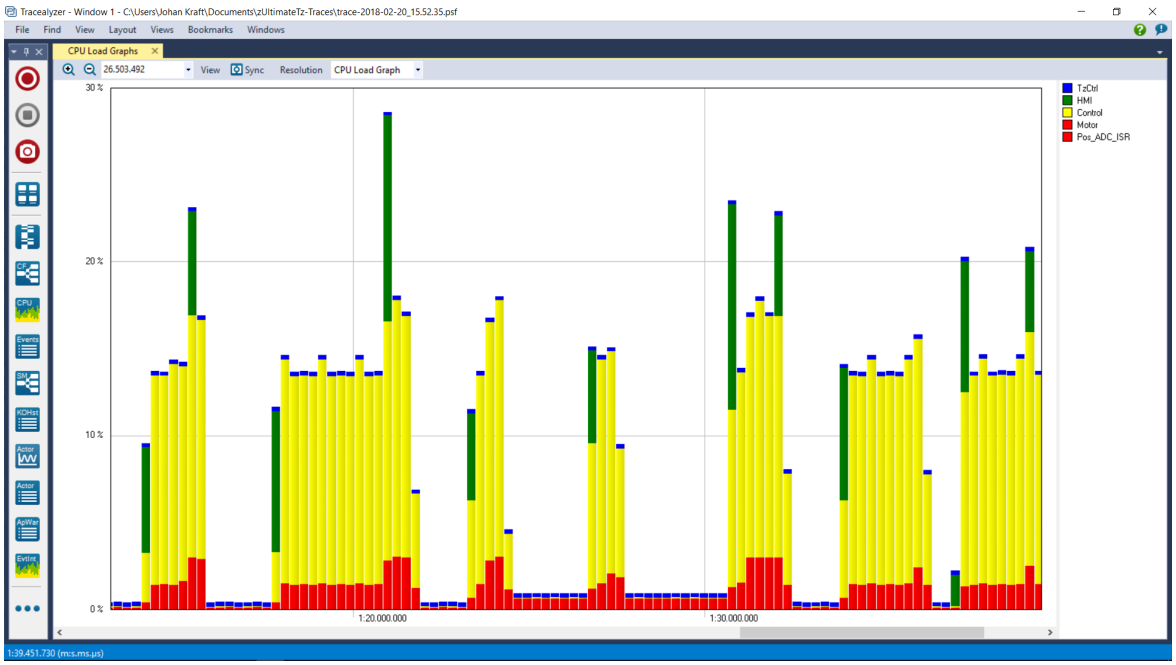
Tracealyzer includes more than 20 interconnected views:



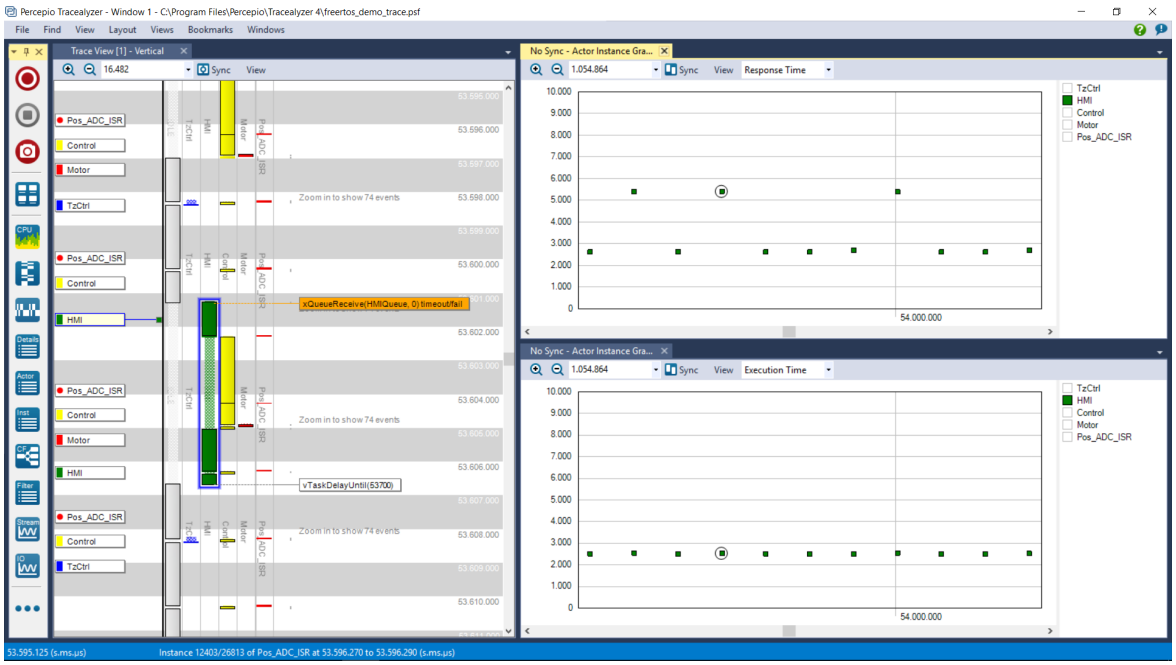
Vertical trace view:



CPU load view:

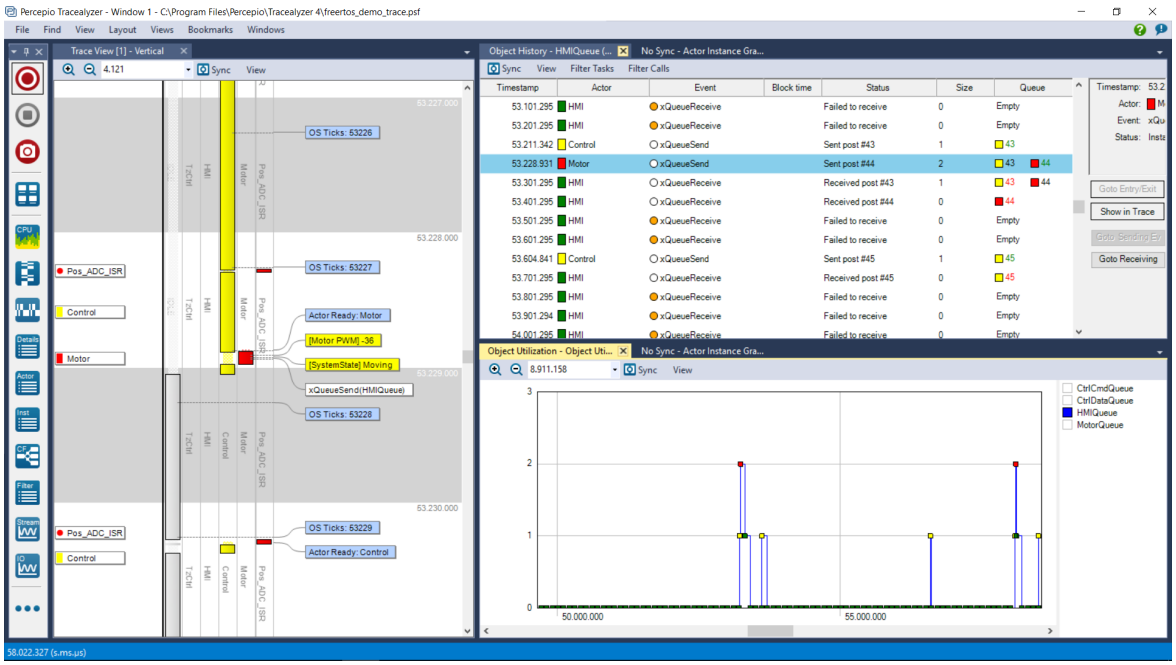


Response time view:



User event plot view:

Object history view:



Debug-Related Hook (Callback) Functions

Defining a malloc failed hook ensures the application developer is notified immediately if an attempt to create a task, queue, semaphore, or event group fails. For more information about the malloc failed hook (or callback), see Heap Memory Management (p. 14).

Defining a stack overflow hook ensures the application developer is notified if the amount of stack used by a task exceeds the stack space allocated to the task. For more information about the stack overflow hook, see the Stack Overflow (p. 239) section of Troubleshooting.

Viewing Runtime and Task State Information

Task Runtime Statistics

Task runtime statistics provide information about the amount of processing time each task has received. A task's *runtime* is the total time the task has been in the Running state since the application booted.

Runtime statistics are intended to be used as a profiling and debugging aid during the development phase of a project. The information they provide is only valid until the counter used as the runtime statistics clock overflows. Collecting runtime statistics will increase the task context switch time.

To obtain binary runtime statistics information, call the uxTaskGetSystemState() API function. To obtain runtime statistics information as a human-readable ASCII table, call the vTaskGetRunTimeStats() helper function.

The Runtime Statistics Clock

Runtime statistics need to measure fractions of a tick period. For this reason, the RTOS tick count is not used as the runtime statistics clock. The clock is provided by the application code instead. We

recommend you make the frequency of the runtime statistics clock between 10 and 100 times faster than the frequency of the tick interrupt. The faster the runtime statistics clock, the more accurate the statistics will be, but also the sooner the time value will overflow.

Ideally, the time value will be generated by a free-running 32-bit peripheral timer/counter, the value of which can be read with no other processing overhead. If the available peripherals and clock speeds do not make that technique possible, then alternative but less efficient techniques include:

1. Configuring a peripheral to generate a periodic interrupt at the desired runtime statistics clock frequency, and then using a count of the number of interrupts generated as the runtime statistics clock.

   This method is very inefficient if the periodic interrupt is only used for the purpose of providing a runtime statistics clock. However, if the application already uses a periodic interrupt with a suitable frequency, then it is simple and efficient to add a count of the number of interrupts generated into the existing interrupt service routine.
2. Generate a 32-bit value by using the current value of a free running 16-bit peripheral timer as the 32-bit value's least significant 16-bits and the number of times the timer has overflowed as the 32-bit value's most significant 16-bits.

You can with somewhat complex manipulation generate a runtime statistics clock by combining the RTOS tick count with the current value of an ARM Cortex-M SysTick timer. Some of the demo projects in the FreeRTOS download show you how to do this.

Configuring an Application to Collect Runtime Statistics

The following table lists the macros required to collect task runtime statistics. The macros are prefixed with 'port' because they were originally intended to be included in the RTOS port layer. It is more practical to define them in FreeRTOSConfig.h.

Macro	Description
configGENERATE_RUN_TIME_STATS	This macro must be set to 1 in FreeRTOSConfig.h. When this macro is set to 1, the scheduler will call the other macros detailed in this table at the appropriate times.
portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()	This macro must be provided to initialize the peripheral used to provide the runtime statistics clock.
portGET_RUN_TIME_COUNTER_VALUE(), or	One of these two macros must be provided to return the current runtime statistics clock value. This is the total time the application has been running, in runtime statistics clock units, since the application first booted.
portALT_GET_RUN_TIME_COUNTER_VALUE(Time)	If the first macro is used, it must be defined to evaluate to the current clock value. If the second macro is used, it must be defined to set its 'Time' parameter to the current clock value.

uxTaskGetSystemState() API Function

uxTaskGetSystemState() provides a snapshot of status information for each task under the control of the FreeRTOS scheduler. The information is provided as an array of TaskStatus_t structures, with one index in the array for each task. The uxTaskGetSystemState() API function prototype is shown here.

The following table lists the uxTaskGetSystemState() parameters and return value.

Parameter Name	Description
pxTaskStatusArray	A pointer to an array of TaskStatus_t structures. The array must contain at least one TaskStatus_t structure for each task. The number of tasks can be determined using the uxTaskGetNumberOfTasks() API function. The TaskStatus_t structure is shown in the following code. The TaskStatus_t structure members are described in the following table.
uxArraySize	The size of the array pointed to by the pxTaskStatusArray parameter. The size is specified as the number of indexes in the array (the number of TaskStatus_t structures contained in the array), not by the number of bytes in the array.
pulTotalRunTime	If configGENERATE_RUN_TIME_STATS is set to 1 in FreeRTOSConfig.h, then <problematic>*</problematic> pulTotalRunTime is set by uxTaskGetSystemState() to the total runtime (as defined by the runtime statistics clock provided by the application) since the target booted. pulTotalRunTime is optional, and can be set to NULL if the total runtime is not required.
Returned value	The number of TaskStatus_t structures that were populated by uxTaskGetSystemState() is returned. The returned value should equal the number returned by the uxTaskGetNumberOfTasks() API function, but will be zero if the value passed in the uxArraySize parameter was too small.

Here is the TaskStatus_t structure.

```
    typedef struct xTASK_STATUS

{

    TaskHandle_t xHandle;

    const char *pcTaskName;

    UBaseType_t xTaskNumber;
```

```
    eTaskState eCurrentState;

    UBaseType_t uxCurrentPriority;

    UBaseType_t uxBasePriority;

    uint32_t ulRunTimeCounter;

    uint16_t usStackHighWaterMark;

} TaskStatus_t;
```

The following table lists the TaskStatus_t structure members.

Parameter Name/ Returned Value	Description
xHandle	The handle of the task to which the information in the structure relates.
pcTaskName	The human-readable text name of the task.
xTaskNumber	Each task has a unique xTaskNumber value. If an application creates and deletes tasks at runtime, then it is possible that a task will have the same handle as a task that was previously deleted. xTaskNumber is provided to allow application code and kernel-aware debuggers to distinguish between a task that is still valid and a deleted task that had the same handle as the valid task.
eCurrentState	An enumerated type that holds the state of the task. eCurrentState can be one of the following values: eRunning, eReady, eBlocked, eSuspended, eDeleted. A task will only be reported as being in the eDeleted state for the short period between the time the task was deleted by a call to vTaskDelete() and the time the Idle task frees the memory that was allocated to the deleted task's internal data structures and stack. After that time, the task will no longer exist in any way, and it is invalid to attempt to use its handle.
uxCurrentPriority	The priority at which the task was running at the time uxTaskGetSystemState() was called. uxCurrentPriority will only be higher than the priority assigned to the task by the application writer if the task has temporarily been assigned a higher priority in accordance with the priority inheritance mechanism described in Resource Management (p. 155).
uxBasePriority	The priority assigned to the task by the application writer. uxBasePriority is only valid if configUSE_MUTEXES is set to 1 in FreeRTOSConfig.h.

ulRunTimeCounter	The total runtime used by the task since the task was created. The total runtime is provided as an absolute time that uses the clock provided by the application writer for the collection of runtime statistics. ulRunTimeCounter is only valid if configGENERATE_RUN_TIME_STATS is set to 1 in FreeRTOSConfig.h.
usStackHighWaterMark	The task's stack high water mark. This is the minimum amount of stack space that has remained for the task since the task was created. It is an indication of how close the task has come to overflowing its stack. The closer this value is to zero, the closer the task has come to overflowing its stack. usStackHighWaterMark is specified in bytes.

vTaskList() Helper Function

vTaskList() provides task status information similar to that provided by uxTaskGetSystemState(), but it presents the information as a human-readable ASCII table rather than an array of binary values.

vTaskList() is a very processor-intensive function. It leaves the scheduler suspended for an extended period. Therefore, we recommend that you use the function for debugging purposes only, and not in a production real-time system.

vTaskList() is available if configUSE_TRACE_FACILITY and configUSE_STATS_FORMATTING_FUNCTIONS are both set to 1 in FreeRTOSConfig.h.

```
void vTaskList( signed char *pcWriteBuffer );
```

The following table lists the vTaskList() parameter.

Parameter Name	**Description**
pcWriteBuffer	A pointer to a character buffer into which the formatted and human-readable table is written. The buffer must be large enough to hold the entire table. No boundary checking is performed.

The output generated by vTaskList() is shown here.

```
tcpip           R       3       393     0
Tmr Svc         R       3       111     48
QConsB1         R       1       143     3
QProdB5         R       0       144     7
QConsB6         R       0       143     8
PolSEM1         R       0       145     11
PolSEM2         R       0       145     12
GenQ            R       0       155     17
MuLow           R       0       147     18
Rec3            R       0       141     30
SUSP_RX         R       0       148     36
Math1           R       0       167     38
Math2           R       0       167     39
```

In the output:

- Each row provides information on a single task.
- The first column is the task's name.
- The second column is the task's state, where 'R' means Ready, 'B' means Blocked, 'S' means Suspended, and 'D' means the task has been deleted. A task will only be reported as being in the deleted state for the short period between the time the task was deleted by a call to vTaskDelete() and the time the Idle task frees the memory that was allocated to the deleted task's internal data structures and stack. After that time, the task will no longer exist in any way, and it is invalid to attempt to use its handle.
- The third column is the task's priority.
- The fourth column is the task's stack high water mark. See the description of usStackHighWaterMark in the table for TaskStatus_t structure members.
- The fifth column is the unique number allocated to the task. See the description of xTaskNumber in the table for TaskStatus_t structure members.

vTaskGetRunTimeStats() Helper Function

vTaskGetRunTimeStats() formats collected runtime statistics into a human-readable ASCII table.

vTaskGetRunTimeStats() is a very processor-intensive function. It leaves the scheduler suspended for an extended period. For this reason, we recommend that you use the function for debug purposes only, and not in a production real-time system.

vTaskGetRunTimeStats() is available when configGENERATE_RUN_TIME_STATS and configUSE_STATS_FORMATTING_FUNCTIONS are both set to 1 in FreeRTOSConfig.h.

The vTaskGetRunTimeStats() API function prototype is shown here.

```
void vTaskGetRunTimeStats( signed char *pcWriteBuffer );
```

The following table lists the vTaskGetRunTimeStats() parameter.

Parameter Name	Description

pcWriteBuffer	A pointer to a character buffer into which the formatted and human-readable table is written. The buffer must be large enough to hold the entire table. No boundary checking is performed.

The output generated by vTaskGetRunTimeStats() is shown here.

```
PolSEM1         994             <1%
PolSEM2         23248           1%
GenQ            194479          16%
MuLow           3690            <1%
Rec3            229450          18%
CNT1            242720          19%
PeekL           94              <1%
CNT_INC         165             <1%
CNT2            243166          20%
SUSP_RX         243192          20%
IDLE            55              <1%
```

In the output:

- Each row provides information on a single task.
- The first column is the task name.
- The second column is the amount of time the task has spent in the Running state as an absolute value. See the description of ulRunTimeCounter in the table for TaskStatus_t structure members.
- The third column is the amount of time the task has spent in the Running state as a percentage of the total time since the target was booted. The total of the displayed percentage times will normally be less than the expected 100% because statistics are collected and calculated using integer calculations that round down to the nearest integer value.

Generating and Displaying Runtime Statistics, a Worked Example

This example uses a hypothetical 16-bit timer to generate a 32-bit runtime statistics clock. The counter is configured to generate an interrupt each time the 16-bit value reaches its maximum value, effectively creating an overflow interrupt. The interrupt service routine counts the number of overflow occurrences.

The 32-bit value is created by using the count of overflow occurrences as the two most significant bytes of the 32-bit value, and the current 16-bit counter value as the least significant two bytes of the 32-bit value. Here is the pseudo code for the interrupt service routine.

```
void TimerOverflowInterruptHandler( void )

{

    /* Just count the number of interrupts. */

    ulOverflowCount++;

    /* Clear the interrupt. */
```

```
    ClearTimerInterrupt();

}
```

Here is the code that enables the collection of runtime statistics.

```
/* Set configGENERATE_RUN_TIME_STATS to 1 to enable collection of runtime
 statistics. When this is done, both portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() and
 portGET_RUN_TIME_COUNTER_VALUE() or portALT_GET_RUN_TIME_COUNTER_VALUE(x) must also be
 defined. */

#define configGENERATE_RUN_TIME_STATS 1

/* portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() is defined to call the function that sets up
 the hypothetical 16-bit timer. (The function's implementation is not shown.) */

void vSetupTimerForRunTimeStats( void );

#define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()

vSetupTimerForRunTimeStats()

/* portALT_GET_RUN_TIME_COUNTER_VALUE() is defined to set its parameter to the current
 runtime counter/time value. The returned time value is 32-bits long, and is formed by
 shifting the count of 16-bit timer overflows into the top two bytes of a 32-bit number,
 and then bitwise ORing the result with the current 16-bit counter value. */

#define portALT_GET_RUN_TIME_COUNTER_VALUE( ulCountValue ) \\

{

    extern volatile unsigned long ulOverflowCount;



    /* Disconnect the clock from the counter so it does not change while its value is being
 used. */

    PauseTimer();

    /* The number of overflows is shifted into the most significant two bytes of the
 returned 32-bit value. */

    ulCountValue = ( ulOverflowCount << 16UL );



    /* The current counter value is used as the least significant two bytes of the returned
 32-bit value. */

    ulCountValue |= ( unsigned long ) ReadTimerCount();

    /* Reconnect the clock to the counter. */

    ResumeTimer(); \\

}
```

The task shown here prints out the collected runtime statistics every 5 seconds.

```
/* For clarity, calls to fflush() have been omitted from this codelisting. */

static void prvStatsTask( void *pvParameters )
```

```
{

    TickType_t xLastExecutionTime;

    /* The buffer used to hold the formatted runtime statistics text needs to be quite
 large. It is therefore declared static to ensure it is not allocated on the task stack.
 This makes this function non-reentrant. */

    static signed char cStringBuffer[ 512 ];

    /* The task will run every 5 seconds. */

    const TickType_t xBlockPeriod = pdMS_TO_TICKS( 5000 );

    /* Initialize xLastExecutionTime to the current time. This is the only time this
 variable needs to be written to explicitly. Afterward, it is updated internally within the
 vTaskDelayUntil() API function. */

    xLastExecutionTime = xTaskGetTickCount();

    /* As per most tasks, this task is implemented in an infinite loop. */

    for( ;; )

    {

        /* Wait until it is time to run this task again. */

        vTaskDelayUntil( &xLastExecutionTime, xBlockPeriod );

        /* Generate a text table from the runtime stats. This must fit into the
 cStringBuffer array. */

        vTaskGetRunTimeStats( cStringBuffer );

        /* Print out column headings for the runtime stats table. */

        printf( "\nTask\t\tAbs\t\t\t%%\n" );

        printf("-------------------------------------------------------------\n" );


        /* Print out the runtime stats themselves. The table of data contains multiple
 lines, so the vPrintMultipleLines() function is called instead of calling printf()
 directly. vPrintMultipleLines() simply calls printf() on each line individually, to ensure
 the line buffering works as expected. */

        vPrintMultipleLines( cStringBuffer );

    }

}
```

Trace Hook Macros

Trace macros have been placed at key points in the FreeRTOS source code. By default, the macros are empty, so they do not generate any code and have no runtime overhead. By overriding the default empty implementations, an application writer can:

- Insert code into FreeRTOS without modifying the FreeRTOS source files.

- Output detailed execution sequencing information by any means available on the target hardware. Trace macros appear in enough places in the FreeRTOS source code to allow them to be used to create a full and detailed scheduler activity trace and profiling log.

Available Trace Hook Macros

The following table lists the macros that are the most useful to an application writer.

Many of the descriptions in this table refer to a variable called pxCurrentTCB. pxCurrentTCB is a FreeRTOS private variable that holds the handle of the task in the Running state. It is available to any macro that is called from the FreeRTOS/Source/tasks.c source file.

Macro	Description
traceTASK_INCREMENT_TICK(xTickCount)	Called during the tick interrupt, after the tick count is incremented. The xTickCount parameter passes the new tick count value into the macro.
traceTASK_SWITCHED_OUT()	Called before a new task is selected to run. At this point, pxCurrentTCB contains the handle of the task about to leave the Running state.
traceTASK_SWITCHED_IN()	Called after a task is selected to run. At this point, pxCurrentTCB contains the handle of the task about to enter the Running state.
traceBLOCKING_ON_QUEUE_RECEIVE(pxQueue)	Called immediately before the currently executing task enters the Blocked state following an attempt to read from an empty queue, or an attempt to 'take' an empty semaphore or mutex. The pxQueue parameter passes the handle of the target queue or semaphore into the macro.
traceBLOCKING_ON_QUEUE_SEND(pxQueue)	Called immediately before the currently executing task enters the Blocked state following an attempt to write to a queue that is full. The pxQueue parameter passes the handle of the target queue into the macro.
traceQUEUE_SEND(pxQueue)	Called from within xQueueSend(), xQueueSendToFront(), xQueueSendToBack(), or any of the semaphore 'give' functions, when the queue send or semaphore 'give' is successful. The pxQueue parameter passes the handle of the target queue or semaphore into the macro.
traceQUEUE_SEND_FAILED(pxQueue)	Called from within xQueueSend(), xQueueSendToFront(), xQueueSendToBack(), or any of the semaphore 'give' functions, when the queue send or semaphore 'give' operation fails. A queue send or semaphore 'give' will fail if the queue is full and remains full for the duration of any block time specified. The pxQueue parameter passes the handle of the target queue or semaphore into the macro.

traceQUEUE_RECEIVE(pxQueue)	Called from within xQueueReceive() or any of the semaphore 'take' functions, when the queue receive or semaphore 'take' is successful. The pxQueue parameter passes the handle of the target queue or semaphore into the macro.
traceQUEUE_RECEIVE_FAILED(pxQueue)	Called from within xQueueReceive() or any of the semaphore 'take' functions, when the queue or semaphore receive operation fails. A queue receive or semaphore 'take' operation will fail if the queue or semaphore is empty and remains empty for the duration of any block time specified. The pxQueue parameter passes the handle of the target queue or semaphore into the macro.
traceQUEUE_SEND_FROM_ISR(pxQueue)	Called from within xQueueSendFromISR() when the send operation is successful. The pxQueue parameter passes the handle of the target queue into the macro.
traceQUEUE_SEND_FROM_ISR_FAILED(pxQueue)	Called from within xQueueSendFromISR() when the send operation fails. A send operation will fail if the queue is already full. The pxQueue parameter passes the handle of the target queue into the macro.
traceQUEUE_RECEIVE_FROM_ISR(pxQueue)	Called from within xQueueReceiveFromISR() when the receive operation is successful. The pxQueue parameter passes the handle of the target queue into the macro.
traceQUEUE_RECEIVE_FROM_ISR_FAILED(pxQueue)	Called from within xQueueReceiveFromISR() when the receive operation fails due to the queue already being empty. The pxQueue parameter passes the handle of the target queue into the macro.
traceTASK_DELAY_UNTIL()	Called from within vTaskDelayUntil() immediately before the calling task enters the Blocked state.
traceTASK_DELAY()	Called from within vTaskDelay() immediately before the calling task enters the Blocked state.

Defining Trace Hook Macros

Each trace macro has a default empty definition. You can override the default definition by providing a new macro definition in FreeRTOSConfig.h. If trace macro definitions become long or complex, then they can be implemented in a new header file that is then itself included from FreeRTOSConfig.h.

In accordance with software engineering best practice, FreeRTOS maintains a strict data hiding policy. Trace macros allow user code to be added to the FreeRTOS source files, so the data types visible to the trace macros will be different from those visible to application code:

- Inside the FreeRTOS/Source/tasks.c source file, a task handle is a pointer to the data structure that describes a task. This is the task's Task Control Block (TCB). Outside of the FreeRTOS/Source/tasks.c source file, a task handle is a pointer to void.

- Inside the FreeRTOS/Source/queue.c source file, a queue handle is a pointer to the data structure that describes a queue. Outside of the FreeRTOS/Source/queue.c source file, a queue handle is a pointer to void.

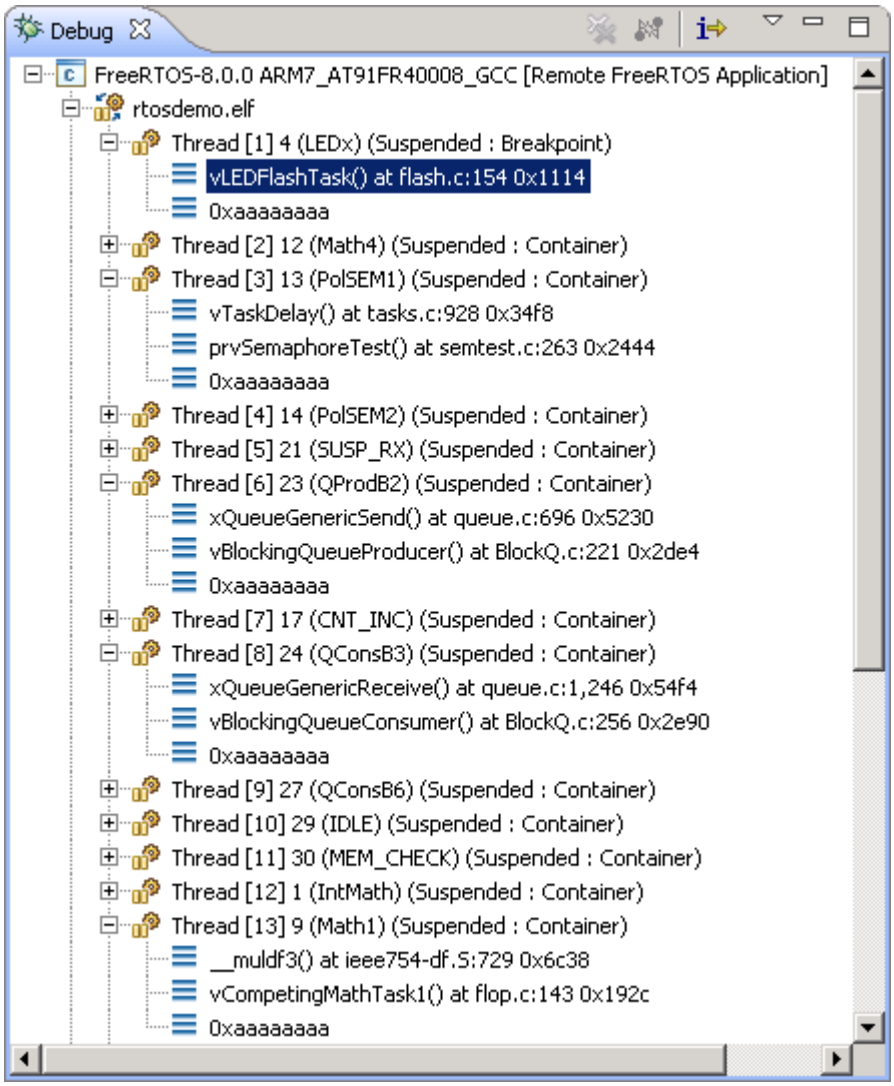
Use extreme caution if a normally private FreeRTOS data structure is accessed directly by a trace macro. Private data structures might change between FreeRTOS versions.

FreeRTOS-Aware Debugger Plugins

Plugins that provide some FreeRTOS awareness are available for the following IDEs. This list is not exhaustive.

- Eclipse (StateViewer)
- Eclipse (ThreadSpy)
- IAR
- ARM DS-5
- Atollic TrueStudio
- Microchip MPLAB
- iSYSTEM WinIDEA

The following figure shows the FreeRTOS ThreadSpy Eclipse plugin from Code Confidence Ltd.


Debug
FreeRTOS-8.0.0 ARM7_AT91FR40008_GCC [Remote FreeRTOS Application]
rtosdemo.elf
Thread [1] 4 (LEDx) (Suspended : Breakpoint)
vLEDFlashTask() at flash.c:154 0x1114
0xaaaaaaaa
Thread [2] 12 (Math4) (Suspended : Container)
Thread [3] 13 (PolSEM1) (Suspended : Container)
vTaskDelay() at tasks.c:928 0x34f8
prvSemaphoreTest() at semtest.c:263 0x2444
0xaaaaaaaa
Thread [4] 14 (PolSEM2) (Suspended : Container)
Thread [5] 21 (SUSP_RX) (Suspended : Container)
Thread [6] 23 (QProdB2) (Suspended : Container)
xQueueGenericSend() at queue.c:696 0x5230
vBlockingQueueProducer() at BlockQ.c:221 0x2de4
0xaaaaaaaa
Thread [7] 17 (CNT_INC) (Suspended : Container)
Thread [8] 24 (QConsB3) (Suspended : Container)
xQueueGenericReceive() at queue.c:1,246 0x54f4
vBlockingQueueConsumer() at BlockQ.c:256 0x2e90
0xaaaaaaaa
Thread [9] 27 (QConsB6) (Suspended : Container)
Thread [10] 29 (IDLE) (Suspended : Container)
Thread [11] 30 (MEM_CHECK) (Suspended : Container)
Thread [12] 1 (IntMath) (Suspended : Container)
Thread [13] 9 (Math1) (Suspended : Container)
__muldf3() at ieee754-df.S:729 0x6c38
vCompetingMathTask1() at flop.c:143 0x192c
0xaaaaaaaa

Troubleshooting

Chapter Introduction and Scope

This section covers the most common issues encountered by new FreeRTOS users, specifically:

- Incorrect interrupt priority assignment
- Stack overflow
- Inappropriate use of printf().

Using configASSERT() improves productivity by immediately trapping and identifying many of the most common sources of error. We strongly recommend that you define configASSERT() while you are developing or debugging a FreeRTOS application. For more information about configASSERT(), see Developer Support (p. 221).

Interrupt Priorities

Note: This is the number one cause of support requests. In most ports, defining configASSERT() will trap the error immediately.

If the FreeRTOS port in use supports interrupt nesting, and the service routine for an interrupt makes use of the FreeRTOS API, then you must set the interrupt's priority at or below configMAX_SYSCALL_INTERRUPT_PRIORITY, as described in Interrupt Management (p. 120). If you don't set the priority correctly, your critical sections will be ineffective, which, in turn, will result in intermittent failures.

Use caution if you are running FreeRTOS on a processor where:

- Interrupt priorities default to having the highest possible priority, which is the case on some ARM Cortex processors. On these processors, the priority of an interrupt that uses the FreeRTOS API cannot be left uninitialized.
- Numerically high priority numbers represent logically low interrupt priorities, which might seem counterintuitive. Again, this is the case on ARM Cortex processors and possibly others.
- For example, on this kind of processor an interrupt that is executing at priority 5 can itself be interrupted by an interrupt that has a priority of 4. Therefore, if configMAX_SYSCALL_INTERRUPT_PRIORITY is set to 5, any interrupt that uses the FreeRTOS API can only be assigned a priority numerically higher than or equal to 5. In that case, interrupt priorities of 5 or 6 are valid, but an interrupt priority of 3 is definitely invalid.
- Different library implementations expect the priority of an interrupt to be specified in a different way. Again, this is particularly relevant to libraries that target ARM Cortex processors, where interrupt priorities are bit-shifted before being written to the hardware registers. Some libraries will perform the bit shift themselves, whereas others expect the bit shift to be performed before the priority is passed into the library function.
- Different implementations of the same architecture implement a different number of interrupt priority bits. For example, a Cortex-M processor from one manufacturer might implement 3 priority bits, while a Cortex-M processor from another manufacturers might implement 4 priority bits.

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

In some FreeRTOS ports, configMAX_SYSCALL_INTERRUPT_PRIORITY has the alternative name configMAX_API_CALL_INTERRUPT_PRIORITY.

Stack Overflow

Stack overflow is the second most common source of support requests. FreeRTOS provides several features to assist trapping and debugging stack-related issues. (These features are not available in the FreeRTOS Windows port.)

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 *high water mark* of the stack.

The uxTaskGetStackHighWaterMark() API function prototype is shown here.

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

The following table lists the 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). For information about obtaining handles to tasks, see the pxCreatedTask parameter of the xTaskCreate() API function. 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.

Overview of Runtime Stack Checking

FreeRTOS includes two optional runtime stack checking mechanisms. These are controlled by the configCHECK_FOR_STACK_OVERFLOW compile time configuration constant in 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 function name and prototype shown here.

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

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

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

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

Method 1 for Runtime Stack Checking

Method 1 is quick to execute, but can miss stack overflows that occur between context switches. This method is used 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 2 for Runtime Stack Checking

Method 2 performs additional checks. This method is used 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 because only 20 bytes are tested. It will most likely catch all stack overflows.

Inappropriate Use of printf() and sprintf()

The inappropriate use of printf() is a common source of error. Application developers who are unaware of this sometimes add more calls to printf() to aid debugging, and in the process, exacerbate the problem.

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

If a printf() implementation that is specifically designed for small embedded systems is not available, be careful using a generic printf() implementation because:

- Just including a call to printf() or sprintf() can massively increase the size of the application's executable.

- printf() and sprintf() might call malloc(), which might be invalid if a memory allocation scheme other than heap_3 is in use. For more information, see Example Memory Allocation Schemes (p. 15).
- printf() and sprintf() might require a stack that is many times bigger than would otherwise be required.

Printf-stdarg.c

Many of the FreeRTOS demonstration projects use a file called printf-stdarg.c, which provides a minimal and stack-efficient implementation of sprintf() that can be used in place of the standard library version. In most cases, this will permit a much smaller stack to be allocated to each task that calls sprintf() and related functions.

printf-stdarg.c also provides a mechanism for directing the printf() output to a port character by character. While slow, this allows stack usage to be decreased even further.

Not all copies of printf-stdarg.c implement snprintf(). Copies that do not implement snprintf() simply ignore the buffer size parameter because they map directly to sprintf().

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

Other Common Errors

This section includes other common errors, their possible causes, and their solutions.

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 more tasks, queues, event groups, or semaphores to be added.

The idle task and possibly the RTOS daemon task are created automatically when vTaskStartScheduler() is called. vTaskStartScheduler() will return only if there is not enough heap memory remaining for these tasks 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. For more information, see Example Memory Allocation Schemes (p. 15).

Symptom: Using an API function in an interrupt causes the application to crash

Do not use API functions in interrupt service routines unless the name of the API function ends with '...FromISR()'. In particular, do not create a critical section in an interrupt unless you are using the interrupt-safe macros. For more information, see Interrupt Management (p. 120).

In FreeRTOS ports that support interrupt nesting, do not use API functions in an interrupt that have been assigned an interrupt priority above configMAX_SYSCALL_INTERRUPT_PRIORITY. For more information, see Interrupt Nesting (p. 151).

Symptom: Sometimes the application crashes in an interrupt service routine

The first thing to check is that the interrupt is not causing a stack overflow. Some ports check for stack overflow only in tasks, not in interrupts.

The way interrupts are defined and used differs between ports and compilers. Therefore, the second thing to check is that the syntax, macros, and calling conventions used in the interrupt service routine are exactly as described on the documentation page provided for the port being used and exactly as demonstrated in the demo application provided with the port.

If the application is running on a processor that uses numerically low priority numbers to represent logically high priorities, then ensure the priority assigned to each interrupt takes that into account because it can seem counterintuitive. If the application is running on a processor that defaults the priority of each interrupt to the maximum possible priority, then ensure the priority of each interrupt is not left at its default value. For more information, see Interrupt Nesting (p. 151).

Symptom: The scheduler crashes when attempting to start the first task

Mke sure the FreeRTOS interrupt handlers have been installed. For an example, see the demo application provided for the port.

Some processors must be in a privileged mode before the scheduler can be started. The easiest way to do this is to place the processor into a privileged mode in the C startup code, before main() is called.

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

If a FreeRTOS API function is called before the scheduler has been started, then interrupts will deliberately be left disabled. They will not be reenabled until the first task starts to execute. This protects the system from crashes caused by interrupts attempting to use FreeRTOS API functions during system initialization, before the scheduler has been started, and while the scheduler might be in an inconsistent state.

Use calls to taskENTER_CRITICAL() and taskEXIT_CRITICAL() to alter the microcontroller interrupt enable bits or priority flags. Do not use any other method. These macros keep a count of their call nesting depth to ensure interrupts become enabled again only when the call nesting has unwound completely to zero. Be aware that some library functions might enable and disable interrupts.

Symptom: The application crashes even before the scheduler is started

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

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

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

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

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