FreeRTOS is customised using a configuration file called FreeRTOSConfig.h. Every FreeRTOS application must have a FreeRTOSConfig.h header file in its pre-processor include path. FreeRTOSConfig.h tailors the RTOS kernel to the application being built. It is therefore specific to the application, not the RTOS, and should be located in an application directory, not in one of the RTOS kernel source code directories.

Each demo application included in the RTOS source code download has its own FreeRTOSConfig.h file. Some of the demos are quite old and do not contain all the available configuration options. Configuration options that are omitted are set to a default value within an RTOS source file.

Here is a typical FreeRTOSConfig.h definition, followed by an explanation of each parameter:

#ifndef FREERTOS\_CONFIG\_H

#define FREERTOS\_CONFIG\_H

/\* Here is a good place to include header files that are required across

your application. \*/

#include "something.h"

#define [configUSE\_PREEMPTION](http://www.freertos.org/a00110.html#configUSE_PREEMPTION) 1

#define [configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION](http://www.freertos.org/a00110.html#configUSE_PORT_OPTIMISED_TASK_SELECTION) 0

#define [configUSE\_TICKLESS\_IDLE](http://www.freertos.org/a00110.html#configUSE_TICKLESS_IDLE) 0

#define [configCPU\_CLOCK\_HZ](http://www.freertos.org/a00110.html#configCPU_CLOCK_HZ) 60000000

#define [configTICK\_RATE\_HZ](http://www.freertos.org/a00110.html#configTICK_RATE_HZ) 250

#define [configMAX\_PRIORITIES](http://www.freertos.org/a00110.html#configMAX_PRIORITIES) 5

#define [configMINIMAL\_STACK\_SIZE](http://www.freertos.org/a00110.html#configMINIMAL_STACK_SIZE) 128

#define [configTOTAL\_HEAP\_SIZE](http://www.freertos.org/a00110.html#configTOTAL_HEAP_SIZE) 10240

#define [configMAX\_TASK\_NAME\_LEN](http://www.freertos.org/a00110.html#configMAX_TASK_NAME_LEN) 16

#define [configUSE\_16\_BIT\_TICKS](http://www.freertos.org/a00110.html#configUSE_16_BIT_TICKS) 0

#define [configIDLE\_SHOULD\_YIELD](http://www.freertos.org/a00110.html#configIDLE_SHOULD_YIELD) 1

#define [configUSE\_MUTEXES](http://www.freertos.org/a00110.html#configUSE_MUTEXES) 0

#define [configUSE\_RECURSIVE\_MUTEXES](http://www.freertos.org/a00110.html#configUSE_RECURSIVE_MUTEXES) 0

#define [configUSE\_COUNTING\_SEMAPHORES](http://www.freertos.org/a00110.html#configUSE_COUNTING_SEMAPHORES) 0

#define [configUSE\_ALTERNATIVE\_API](http://www.freertos.org/a00110.html#configUSE_ALTERNATIVE_API) 0 /\* Deprecated! \*/

#define [configQUEUE\_REGISTRY\_SIZE](http://www.freertos.org/a00110.html#configQUEUE_REGISTRY_SIZE) 10

#define [configUSE\_QUEUE\_SETS](http://www.freertos.org/a00110.html#configUSE_QUEUE_SETS) 0

#define [configUSE\_TIME\_SLICING](http://www.freertos.org/a00110.html#configUSE_TIME_SLICING) 0

#define [configUSE\_NEWLIB\_REENTRANT](http://www.freertos.org/a00110.html#configUSE_NEWLIB_REENTRANT) 0

#define [configENABLE\_BACKWARD\_COMPATIBILITY](http://www.freertos.org/a00110.html#configENABLE_BACKWARD_COMPATIBILITY) 0

/\* Hook function related definitions. \*/

#define [configUSE\_IDLE\_HOOK](http://www.freertos.org/a00110.html#configUSE_IDLE_HOOK) 0

#define [configUSE\_TICK\_HOOK](http://www.freertos.org/a00110.html#configUSE_TICK_HOOK) 0

#define [configCHECK\_FOR\_STACK\_OVERFLOW](http://www.freertos.org/a00110.html#configCHECK_FOR_STACK_OVERFLOW) 0

#define [configUSE\_MALLOC\_FAILED\_HOOK](http://www.freertos.org/a00110.html#configUSE_MALLOC_FAILED_HOOK) 0

/\* Run time and task stats gathering related definitions. \*/

#define [configGENERATE\_RUN\_TIME\_STATS](http://www.freertos.org/a00110.html#configGENERATE_RUN_TIME_STATS) 0

#define [configUSE\_TRACE\_FACILITY](http://www.freertos.org/a00110.html#configUSE_TRACE_FACILITY) 0

#define [configUSE\_STATS\_FORMATTING\_FUNCTIONS](http://www.freertos.org/a00110.html#configUSE_STATS_FORMATTING_FUNCTIONS) 0

/\* Co-routine related definitions. \*/

#define [configUSE\_CO\_ROUTINES](http://www.freertos.org/a00110.html#configUSE_CO_ROUTINES) 0

#define [configMAX\_CO\_ROUTINE\_PRIORITIES](http://www.freertos.org/a00110.html#configMAX_CO_ROUTINE_PRIORITIES) 1

/\* Software timer related definitions. \*/

#define [configUSE\_TIMERS](http://www.freertos.org/a00110.html#configUSE_TIMERS) 1

#define [configTIMER\_TASK\_PRIORITY](http://www.freertos.org/a00110.html#configTIMER_TASK_PRIORITY) 3

#define [configTIMER\_QUEUE\_LENGTH](http://www.freertos.org/a00110.html#configTIMER_QUEUE_LENGTH) 10

#define [configTIMER\_TASK\_STACK\_DEPTH](http://www.freertos.org/a00110.html#configTIMER_TASK_STACK_DEPTH) configMINIMAL\_STACK\_SIZE

/\* Interrupt nesting behaviour configuration. \*/

#define [configKERNEL\_INTERRUPT\_PRIORITY](http://www.freertos.org/a00110.html#kernel_priority) [dependent of processor]

#define [configMAX\_SYSCALL\_INTERRUPT\_PRIORITY](http://www.freertos.org/a00110.html#kernel_priority) [dependent on processor and application]

#define [configMAX\_API\_CALL\_INTERRUPT\_PRIORITY](http://www.freertos.org/a00110.html#kernel_priority) [dependent on processor and application]

/\* Define to trap errors during development. \*/

#define [configASSERT](http://www.freertos.org/a00110.html#configASSERT)( ( x ) ) if( ( x ) == 0 ) vAssertCalled( \_\_FILE\_\_, \_\_LINE\_\_ )

/\* FreeRTOS MPU specific definitions. \*/

#define [configINCLUDE\_APPLICATION\_DEFINED\_PRIVILEGED\_FUNCTIONS](http://www.freertos.org/a00110.html#configINCLUDE_APPLICATION_DEFINED_PRIVILEGED_FUNCTIONS) 0

/\* Optional functions - most linkers will remove unused functions anyway. \*/

#define [INCLUDE\_vTaskPrioritySet](http://www.freertos.org/a00110.html#include_parameters) 1

#define [INCLUDE\_uxTaskPriorityGet](http://www.freertos.org/a00110.html#include_parameters) 1

#define [INCLUDE\_vTaskDelete](http://www.freertos.org/a00110.html#include_parameters) 1

#define [INCLUDE\_vTaskSuspend](http://www.freertos.org/a00110.html#include_parameters) 1

#define [INCLUDE\_xResumeFromISR](http://www.freertos.org/a00110.html#include_parameters) 1

#define [INCLUDE\_vTaskDelayUntil](http://www.freertos.org/a00110.html#include_parameters) 1

#define [INCLUDE\_vTaskDelay](http://www.freertos.org/a00110.html#include_parameters) 1

#define [INCLUDE\_xTaskGetSchedulerState](http://www.freertos.org/a00110.html#include_parameters) 1

#define [INCLUDE\_xTaskGetCurrentTaskHandle](http://www.freertos.org/a00110.html#include_parameters) 1

#define [INCLUDE\_uxTaskGetStackHighWaterMark](http://www.freertos.org/a00110.html#include_parameters) 0

#define [INCLUDE\_xTaskGetIdleTaskHandle](http://www.freertos.org/a00110.html#include_parameters) 0

#define [INCLUDE\_xTimerGetTimerDaemonTaskHandle](http://www.freertos.org/a00110.html#include_parameters) 0

#define [INCLUDE\_pcTaskGetTaskName](http://www.freertos.org/a00110.html#include_parameters) 0

#define [INCLUDE\_eTaskGetState](http://www.freertos.org/a00110.html#include_parameters) 0

#define [INCLUDE\_xEventGroupSetBitFromISR](http://www.freertos.org/a00110.html#include_parameters) 1

#define [INCLUDE\_xTimerPendFunctionCall](http://www.freertos.org/a00110.html#include_parameters) 0

/\* A header file that defines trace macro can be included here. \*/

#endif /\* FREERTOS\_CONFIG\_H \*/

**'config' Parameters**

**configUSE\_PREEMPTION**

Set to 1 to use the preemptive RTOS scheduler, or 0 to use the cooperative RTOS scheduler.

**configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION**

Some FreeRTOS ports have two methods of selecting the next task to execute - a generic method, and a method that is specific to that port.

The Generic method:

* Is used when configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION is set to 0, or when a port specific method is not implemented.
* Can be used with all FreeRTOS ports.
* Is completely written in C, making it less efficient than a port specific method.
* Does not impose a limit on the maximum number of available priorities.

A port specific method:

* Is not available for all ports.
* Is used when configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION is set to 1.
* Relies on one or more architecture specific assembly instructions (typically a Count Leading Zeros [CLZ] of equivalent instruction) so can only be used with the architecture for which it was specifically written.
* Is more efficient than the generic method.
* Typically imposes a limit of 32 on the maximum number of available priorities.

**configUSE\_TICKLESS\_IDLE**

Set configUSE\_TICKLESS\_IDLE to 1 to use the [low power tickless mode](http://www.freertos.org/low-power-tickless-rtos.html), or 0 to keep the tick interrupt running at all times.

**configUSE\_IDLE\_HOOK**

Set to 1 if you wish to use an [idle hook](http://www.freertos.org/RTOS-idle-task.html), or 0 to omit an idle hook.

**configUSE\_MALLOC\_FAILED\_HOOK**

The kernel uses a call to pvPortMalloc() to allocate memory from the heap each time a task, queue or semaphore is created. The official FreeRTOS download includes four sample memory allocation schemes for this purpose. The schemes are implemented in the heap\_1.c, heap\_2.c, heap\_3.c, heap\_4.c and heap\_5.c source files respectively. configUSE\_MALLOC\_FAILED\_HOOK is only relevant when one of these three sample schemes is being used.

The malloc() failed hook function is a hook (or callback) function that, if defined and configured, will be called if pvPortMalloc() ever returns NULL. NULL will be returned only if there is insufficient FreeRTOS heap memory remaining for the requested allocation to succeed.

If configUSE\_MALLOC\_FAILED\_HOOK is set to 1 then the application must define a malloc() failed hook function. If configUSE\_MALLOC\_FAILED\_HOOK is set to 0 then the malloc() failed hook function will not be called, even if one is defined. Malloc() failed hook functions must have the name and prototype shown below.

void vApplicationMallocFailedHook( void );

**configUSE\_TICK\_HOOK**

Set to 1 if you wish to use an [tick hook](http://www.freertos.org/a00016.html#TickHook), or 0 to omit an tick hook.

**configCPU\_CLOCK\_HZ**

Enter the frequency in Hz at which the *internal* clock that driver the peripheral used to generate the tick interrupt will be executing - this is normally the same clock that drives the internal CPU clock. This value is required in order to correctly configure timer peripherals.

**configTICK\_RATE\_HZ**

The frequency of the RTOS tick interrupt.

The tick interrupt is used to measure time. Therefore a higher tick frequency means time can be measured to a higher resolution. However, a high tick frequency also means that the RTOS kernel will use more CPU time so be less efficient. The RTOS demo applications all use a tick rate of 1000Hz. This is used to test the RTOS kernel and is higher than would normally be required.

More than one task can share the same priority. The RTOS scheduler will share processor time between tasks of the same priority by switching between the tasks during each RTOS tick. A high tick rate frequency will therefore also have the effect of reducing the 'time slice' given to each task.

**configMAX\_PRIORITIES**

The number of [priorities](http://www.freertos.org/RTOS-task-priority.html) available to the application tasks. Any number of tasks can share the same priority. Co-routines are prioritised separately - see configMAX\_CO\_ROUTINE\_PRIORITIES.

Each available priority consumes RAM within the RTOS kernel so this value should not be set any higher than actually required by your application.

**configMINIMAL\_STACK\_SIZE**

The size of the stack used by the idle task. Generally this should not be reduced from the value set in the FreeRTOSConfig.h file provided with the demo application for the port you are using.

Like the stack size parameter to the [xTaskCreate()](http://www.freertos.org/a00125.html) function, the stack size is specified in words, not bytes. If each item placed on the stack is 32-bits, then a stack size of 100 means 400 bytes (each 32-bit stack item consuming 4 bytes).

**configTOTAL\_HEAP\_SIZE**

The total amount of RAM available to the RTOS kernel.

This value will only be used if your application makes use of one of the sample memory allocation schemes provided in the FreeRTOS source code download. See the [memory configuration](http://www.freertos.org/a00111.html) section for further details.

**configMAX\_TASK\_NAME\_LEN**

The maximum permissible length of the descriptive name given to a task when the task is created. The length is specified in the number of characters *including* the NULL termination byte.

**configUSE\_TRACE\_FACILITY**

Set to 1 if you wish to include additional structure members and functions to assist with execution visualisation and tracing.

**configUSE\_STATS\_FORMATTING\_FUNCTIONS**

Set configUSE\_TRACE\_FACILITY and configUSE\_STATS\_FORMATTING\_FUNCTIONS to 1 to include the [vTaskList()](http://www.freertos.org/a00021.html#vTaskList) and [vTaskGetRunTimeStats()](http://www.freertos.org/a00021.html#vTaskGetRunTimeStats) functions in the build. Setting either to 0 will omit vTaskList() and vTaskGetRunTimeStates() from the build.

**configUSE\_16\_BIT\_TICKS**

Time is measured in 'ticks' - which is the number of times the tick interrupt has executed since the RTOS kernel was started. The tick count is held in a variable of type TickType\_t.

Defining configUSE\_16\_BIT\_TICKS as 1 causes TickType\_t to be defined (typedef'ed) as an unsigned 16bit type. Defining configUSE\_16\_BIT\_TICKS as 0 causes TickType\_t to be defined (typedef'ed) as an unsigned 32bit type.

Using a 16 bit type will greatly improve performance on 8 and 16 bit architectures, but limits the maximum specifiable time period to 65535 'ticks'. Therefore, assuming a tick frequency of 250Hz, the maximum time a task can delay or block when a 16bit counter is used is 262 seconds, compared to 17179869 seconds when using a 32bit counter.

**configIDLE\_SHOULD\_YIELD**

This parameter controls the behaviour of tasks at the idle priority. It only has an effect if:

1. The preemptive scheduler is being used.
2. The users application creates tasks that run at the idle priority.

Tasks that share the same priority will time slice. Assuming none of the tasks get preempted, it might be assumed that each task of at a given priority will be allocated an equal amount of processing time - and if the shared priority is above the idle priority then this is indeed the case.

When tasks share the idle priority the behaviour can be slightly different. When configIDLE\_SHOULD\_YIELD is set to 1 the idle task will yield immediately should any other task at the idle priority be ready to run. This ensures the minimum amount of time is spent in the idle task when application tasks are available for scheduling. This behaviour can however have undesirable effects (depending on the needs of your application) as depicted below:

http://www.freertos.org/idleyield.gif

This diagram shows the execution pattern of four tasks at the idle priority. Tasks A, B and C are application tasks. Task I is the idle task. A context switch occurs with regular period at times T0, T1, ..., T6. When the idle task yields task A starts to execute - but the idle task has already taken up some of the current time slice. This results in task I and task A effectively sharing a time slice. The application tasks B and C therefore get more processing time than the application task A.

This situation can be avoided by:

* If appropriate, using an [idle hook](http://www.freertos.org/RTOS-idle-task.html) in place of separate tasks at the idle priority.
* Creating all application tasks at a priority greater than the idle priority.
* Setting configIDLE\_SHOULD\_YIELD to 0.

Setting configIDLE\_SHOULD\_YIELD prevents the idle task from yielding processing time until the end of its time slice. This ensure all tasks at the idle priority are allocated an equal amount of processing time - but at the cost of a greater proportion of the total processing time being allocated to the idle task.

**configUSE\_MUTEXES**

Set to 1 to include mutex functionality in the build, or 0 to omit mutex functionality from the build. Readers should familiarise themselves with the differences between mutexes and binary semaphores in relation to the FreeRTOS functionality.

**configUSE\_RECURSIVE\_MUTEXES**

Set to 1 to include recursive mutex functionality in the build, or 0 to omit recursive mutex functionality from the build.

**configUSE\_COUNTING\_SEMAPHORES**

Set to 1 to include counting semaphore functionality in the build, or 0 to omit counting semaphore functionality from the build.

**configUSE\_ALTERNATIVE\_API**

Set to 1 to include the 'alternative' queue functions in the build, or 0 to omit the 'alternative' queue functions from the build. The alternative API is described within the queue.h header file. **The alternative API is deprecated and should not be used in new designs**.

**configCHECK\_FOR\_STACK\_OVERFLOW**

The [stack overflow detection](http://www.freertos.org/Stacks-and-stack-overflow-checking.html) page describes the use of this parameter.

**configQUEUE\_REGISTRY\_SIZE**

The queue registry has two purposes, both of which are associated with RTOS kernel aware debugging:

1. It allows a textual name to be associated with a queue for easy queue identification within a debugging GUI.
2. It contains the information required by a debugger to locate each registered queue and semaphore.

The queue registry has no purpose unless you are using a RTOS kernel aware debugger.

configQUEUE\_REGISTRY\_SIZE defines the maximum number of queues and semaphores that can be registered. Only those queues and semaphores that you want to view using a RTOS kernel aware debugger need be registered. See the API reference documentation for [vQueueAddToRegistry()](http://www.freertos.org/vQueueAddToRegistry.html) and [vQueueUnregisterQueue()](http://www.freertos.org/vQueueUnregisterQueue.html) for more information.

**configUSE\_QUEUE\_SETS**

Set to 1 to include [queue set](http://www.freertos.org/Pend-on-multiple-rtos-objects.html) functionality (the ability to block, or pend, on multiple queues and semaphores), or 0 to omit queue set functionality.

**configUSE\_TIME\_SLICING**

By default (if configUSE\_TIME\_SLICING is not defined, or if configUSE\_TIME\_SLICING is defined as 1) FreeRTOS uses prioritised preemptive scheduling with time slicing. That means the RTOS scheduler will always run the highest priority task that is in the Ready state, and will switch between tasks of equal priority on every RTOS tick interrupt. If configUSE\_TIME\_SLICING is set to 0 then the RTOS scheduler will still run the highest priority task that is in the Ready state, but will not switch between tasks of equal priority just because a tick interrupt has occurred.

**configUSE\_NEWLIB\_REENTRANT**

If configUSE\_NEWLIB\_REENTRANT is set to 1 then a [newlib](http://sourceware.org/newlib/) reent structure will be allocated for each created task.

Note Newlib support has been included by popular demand, but is not used by the FreeRTOS maintainers themselves. FreeRTOS is not responsible for resulting newlib operation. User must be familiar with newlib and must provide system-wide implementations of the necessary stubs. Be warned that (at the time of writing) the current newlib design implements a system-wide malloc() that must be provided with locks.

**configENABLE\_BACKWARD\_COMPATIBILITY**

The FreeRTOS.h header file includes a set of #define macros that map the names of data types used in versions of FreeRTOS prior to version 8.0.0 to the names used in FreeRTOS version 8.0.0. The macros allow application code to update the version of FreeRTOS they are built against from a pre 8.0.0 version to a post 8.0.0 version without modification. Setting configENABLE\_BACKWARD\_COMPATIBILITY to 0 in FreeRTOSConfig.h excludes the macors from the build, and in so doing allowing validation that no pre version 8.0.0 names are being used.

**configGENERATE\_RUN\_TIME\_STATS**

The [Run Time Stats](http://www.freertos.org/rtos-run-time-stats.html) page describes the use of this parameter.

**configUSE\_CO\_ROUTINES**

Set to 1 to include co-routine functionality in the build, or 0 to omit co-routine functionality from the build. To include co-routines croutine.c must be included in the project.

**configMAX\_CO\_ROUTINE\_PRIORITIES**

The number of [priorities](http://www.freertos.org/co-routine-priorities.html) available to the application co-routines. Any number of co-routines can share the same priority. Tasks are prioritised separately - see configMAX\_PRIORITIES.

**configUSE\_TIMERS**

Set to 1 to include software timer functionality, or 0 to omit software timer functionality. See the [FreeRTOS software timers](http://www.freertos.org/RTOS-software-timer.html) page for a full description.

**configTIMER\_TASK\_PRIORITY**

Sets the priority of the software timer service/daemon task. See the [FreeRTOS software timers](http://www.freertos.org/RTOS-software-timer.html) page for a full description.

**configTIMER\_QUEUE\_LENGTH**

Sets the length of the software timer command queue. See the [FreeRTOS software timers](http://www.freertos.org/RTOS-software-timer.html) page for a full description.

**configTIMER\_TASK\_STACK\_DEPTH**

Sets the stack depth allocated to the software timer service/daemon task. See the [FreeRTOS software timers](http://www.freertos.org/RTOS-software-timer.html) page for a full description.

**configKERNEL\_INTERRUPT\_PRIORITY  
configMAX\_SYSCALL\_INTERRUPT\_PRIORITY and  
configMAX\_API\_CALL\_INTERRUPT\_PRIORITY**

Ports that contain a configKERNEL\_INTERRUPT\_PRIORITY setting include ARM Cortex-M3, PIC24, dsPIC, PIC32, SuperH and RX600. Ports that contain a configMAX\_SYSCALL\_INTERRUPT\_PRIORITY setting include PIC32, RX600, ARM Cortex-A and ARM Cortex-M ports.

ARM Cortex-M3 and ARM Cortex-M4 users please take heed of the special note at the end of this section!

configMAX\_API\_CALL\_INTERRUPT\_PRIORITY is a new name for configMAX\_SYSCALL\_INTERRUPT\_PRIORITY that is used by newer ports only. The two are equivalent.

configKERNEL\_INTERRUPT\_PRIORITY should be set to the lowest priority.

Note in the following discussion that only API functions that end in "FromISR" can be called from within an interrupt service routine.

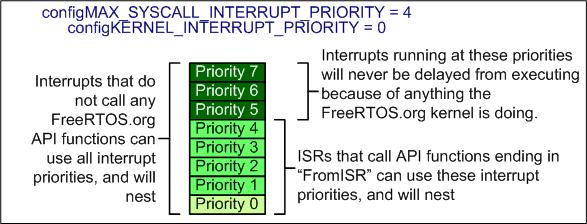
For ports that only implement configKERNEL\_INTERRUPT\_PRIORITY  
configKERNEL\_INTERRUPT\_PRIORITY sets the interrupt priority used by the RTOS kernel itself. Interrupts that call API functions must also execute at this priority. Interrupts that do not call API functions can execute at higher priorities and therefore never have their execution delayed by the RTOS kernel activity (within the limits of the hardware itself).

For ports that implement both configKERNEL\_INTERRUPT\_PRIORITY and configMAX\_SYSCALL\_INTERRUPT\_PRIORITY:  
configKERNEL\_INTERRUPT\_PRIORITY sets the interrupt priority used by the RTOS kernel itself. configMAX\_SYSCALL\_INTERRUPT\_PRIORITY sets the highest interrupt priority from which interrupt safe FreeRTOS API functions can be called.

A full interrupt nesting model is achieved by setting configMAX\_SYSCALL\_INTERRUPT\_PRIORITY above (that is, at a higher priority level) than configKERNEL\_INTERRUPT\_PRIORITY. **This means the FreeRTOS kernel does not completely disable interrupts, even inside critical sections.** Further, this is achieved without the disadvantages of a segmented kernel architecture. Note however, certain microcontroller architectures will (in hardware) disable interrupts when a new interrupt is accepted - meaning interrupts are unavoidably disabled for the short period between the hardware accepting the interrupt, and the FreeRTOS code re-enabling interrupts.

Interrupts that do not call API functions can execute at priorities above configMAX\_SYSCALL\_INTERRUPT\_PRIORITY and therefore never be delayed by the RTOS kernel execution.

For example, imagine a hypothetical microcontroller that has 8 interrupt priority levels - 0 being the lowest and 7 being the highest (see the special note for ARM Cortex-M3 users at the end of this section). The picture below describes what can and cannot be done at each priority level should the two configuration constants be set to 4 and 0 as shown:

  
Example interrupt priority configuration

These configuration parameters allow very flexible interrupt handling:

* Interrupt handling 'tasks' can be written and prioritised as per any other task in the system. These are tasks that are woken by an interrupt. The interrupt service routine (ISR) itself should be written to be as short as it possibly can be - it just grabs the data then wakes the high priority handler task. The ISR then returns directly into the woken handler task - so interrupt processing is contiguous in time just as if it were all done in the ISR itself. The benefit of this is that all interrupts remain enabled while the handler task executes.
* Ports that implement configMAX\_SYSCALL\_INTERRUPT\_PRIORITY take this further - permitting a fully nested model where interrupts between the RTOS kernel interrupt priority and configMAX\_SYSCALL\_INTERRUPT\_PRIORITY can nest and make applicable API calls. Interrupts with priority above configMAX\_SYSCALL\_INTERRUPT\_PRIORITY are never delayed by the RTOS kernel activity.
* ISR's running above the maximum syscall priority are never masked out by the RTOS kernel itself, so their responsiveness is not effected by the RTOS kernel functionality. This is ideal for interrupts that require very high temporal accuracy - for example interrupts that perform motor commutation. However, such ISR's cannot use the FreeRTOS API functions.

To utilize this scheme your application design must adhere to the following rule: **Any interrupt that uses the FreeRTOS API must be set to the same priority as the RTOS kernel (as configured by the configKERNEL\_INTERRUPT\_PRIORITY macro), or at or below configMAX\_SYSCALL\_INTERRUPT\_PRIORITY for ports that include this functionality.**

A special note for ARM Cortex-M3 and ARM Cortex-M4 users: Please read [the page dedicated to interrupt priority settings on ARM Cortex-M devices](http://www.freertos.org/RTOS-Cortex-M3-M4.html). As a minimum, remember that ARM Cortex-M3 cores use numerically low priority numbers to represent HIGH priority interrupts, which can seem counter-intuitive and is easy to forget! If you wish to assign an interrupt a low priority do NOT assign it a priority of 0 (or other low numeric value) as this can result in the interrupt actually having the highest priority in the system - and therefore potentially make your system crash if this priority is above configMAX\_SYSCALL\_INTERRUPT\_PRIORITY.

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

**configASSERT**

The semantics of the configASSERT() macro are the same as the standard C assert() macro. An assertion is triggered if the parameter passed into configASSERT() is zero.

configASSERT() is called throughout the FreeRTOS source files to check how the application is using FreeRTOS. It is highly recommended to develop FreeRTOS applications with configASSERT() defined.

The example definition (shown at the top of the file and replicated below) calls vAssertCalled(), passing in the file name and line number of the triggering configASSERT() call (\_\_FILE\_\_ and \_\_LINE\_\_ are standard macros provided by most compilers). This is just for demonstration as vAssertCalled() is not a FreeRTOS function, configASSERT() can be defined to take whatever action the application writer deems appropriate.

It is normal to define configASSERT() in such a way that it will prevent the application from executing any further. This if for two reasons; stopping the application at the point of the assertion allows the cause of the assertion to be debugged, and executing past a triggered assertion will probably result in a crash anyway.

Note defining configASSERT() will increase both the application code size and execution time. When the application is stable the additional overhead can be removed by simply commenting out the configASSERT() definition in FreeRTOSConfig.h.

/\* Define configASSERT() to call vAssertCalled() if the assertion fails. The assertion

has failed if the value of the parameter passed into configASSERT() equals zero. \*/

#define configASSERT( ( x ) ) if( ( x ) == 0 ) vAssertCalled( \_\_FILE\_\_, \_\_LINE\_\_ )

**configINCLUDE\_APPLICATION\_DEFINED\_PRIVILEGED\_FUNCTIONS**

configINCLUDE\_APPLICATION\_DEFINED\_PRIVILEGED\_FUNCTIONS is only used by FreeRTOS MPU.

If configINCLUDE\_APPLICATION\_DEFINED\_PRIVILEGED\_FUNCTIONS is set to 1 then the application writer must provide a header file called "application\_defined\_privileged\_functions.h", in which functions the application writer needs to execute in privileged mode can be implemented. Note that, despite having a .h extension, the header file should contain the implementation of the C functions, not just the functions' prototypes.

Functions implemented in "application\_defined\_privileged\_functions.h" must save and restore the processor's privilege state using the prvRaisePrivilege() function and portRESET\_PRIVILEGE() macro respectively. For example, if a library provided print function accesses RAM that is outside of the control of the application writer, and therefore cannot be allocated to a memory protected user mode task, then the print function can be encapsulated in a privileged function using the following code:

void MPU\_debug\_printf( const char \*pcMessage )

{

/\* State the privilege level of the processor when the function was called. \*/

BaseType\_t xRunningPrivileged = prvRaisePrivilege();

/\* Call the library function, which now has access to all RAM. \*/

debug\_printf( pcMessage );

/\* Reset the processor privilege level to its original value. \*/

portRESET\_PRIVILEGE( xRunningPrivileged );

}

This technique should only be use during development, and not deployment, as it circumvents the memory protection.

**INCLUDE Parameters**

The macros starting 'INCLUDE' allow those components of the real time kernel not utilized by your application to be excluded from your build. This ensures the RTOS does not use any more ROM or RAM than necessary for your particular embedded application.

Each macro takes the form ...

INCLUDE\_FunctionName

... where FunctionName indicates the API function (or set of functions) that can optionally be excluded. To include the API function set the macro to 1, to exclude the function set the macro to 0. For example, to include the vTaskDelete() API function use:

#define INCLUDE\_vTaskDelete 1

To exclude vTaskDelete() from your build use:

#define INCLUDE\_vTaskDelete 0