Customisation

[Configuration]

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                    1
#define configUSE_PORT_OPTIMISED_TASK_SELECTION 0
#define configUSE_TICKLESS_IDLE                 0
#define configCPU_CLOCK_HZ
60000000
#define configSYSTICK_CLOCK_HZ
1000000
#define configTICK_RATE_HZ                      250
```

```
#define configMAX_PRIORITIES                    5
#define configMINIMAL_STACK_SIZE                128
#define configMAX_TASK_NAME_LEN                 16
#define configUSE_16_BIT_TICKS                  0
#define configIDLE_SHOULD_YIELD                 1
#define configUSE_TASK_NOTIFICATIONS            1
#define configTASK_NOTIFICATION_ARRAY_ENTRIES   3
#define configUSE_MUTEXES                       0
#define configUSE_RECURSIVE_MUTEXES             0
#define configUSE_COUNTING_SEMAPHORES           0
#define configUSE_ALTERNATIVE_API               0
/* Deprecated! */
#define configQUEUE_REGISTRY_SIZE               10
#define configUSE_QUEUE_SETS                    0
#define configUSE_TIME_SLICING                  0
#define configUSE_NEWLIB_REENTRANT              0
#define configENABLE_BACKWARD_COMPATIBILITY     0
#define configNUM_THREAD_LOCAL_STORAGE_POINTERS 5
#define configUSE_MINI_LIST_ITEM                1
#define configSTACK_DEPTH_TYPE
uint16_t
#define configMESSAGE_BUFFER_LENGTH_TYPE
size_t
#define configHEAP_CLEAR_MEMORY_ON_FREE         1

/* Memory allocation related definitions. */
#define configSUPPORT_STATIC_ALLOCATION
1
#define configSUPPORT_DYNAMIC_ALLOCATION
1
#define configTOTAL_HEAP_SIZE
10240
#define configAPPLICATION_ALLOCATED_HEAP
1
#define configSTACK_ALLOCATION_FROM_SEPARATE_HEAP
1

/* Hook function related definitions. */
#define configUSE_IDLE_HOOK                     0
#define configUSE_TICK_HOOK                     0
#define configCHECK_FOR_STACK_OVERFLOW          0
#define configUSE_MALLOC_FAILED_HOOK            0
#define configUSE_DAEMON_TASK_STARTUP_HOOK      0
#define configUSE_SB_COMPLETED_CALLBACK         0

/* Run time and task stats gathering related
definitions. */
#define configGENERATE_RUN_TIME_STATS           0
#define configUSE_TRACE_FACILITY                0
#define configUSE_STATS_FORMATTING_FUNCTIONS    0

/* Co-routine related definitions. */
#define configUSE_CO_ROUTINES                   0
#define configMAX_CO_ROUTINE_PRIORITIES         1

/* Software timer related definitions. */
#define configUSE_TIMERS                        1
#define configTIMER_TASK_PRIORITY               3
#define configTIMER_QUEUE_LENGTH                10
#define configTIMER_TASK_STACK_DEPTH
configMINIMAL_STACK_SIZE

/* Interrupt nesting behaviour configuration. */
#define configKERNEL_INTERRUPT_PRIORITY
```

```
[dependent of processor]
#define configMAX_SYSCALL_INTERRUPT_PRIORITY
[dependent on processor and application]
#define configMAX_API_CALL_INTERRUPT_PRIORITY
[dependent on processor and application]

/* Define to trap errors during development. */
#define configASSERT( ( x ) ) if( ( x ) == 0 )
vAssertCalled( __FILE__, __LINE__ )

/* FreeRTOS MPU specific definitions. */
#define
configINCLUDE_APPLICATION_DEFINED_PRIVILEGED_FUNCTIONS
 0
#define configTOTAL_MPU_REGIONS
8 /* Default value. */
#define configTEX_S_C_B_FLASH
0x07UL /* Default value. */
#define configTEX_S_C_B_SRAM
0x07UL /* Default value. */
#define configENFORCE_SYSTEM_CALLS_FROM_KERNEL_ONLY
1
#define configALLOW_UNPRIVILEGED_CRITICAL_SECTIONS
1
#define configENABLE_ERRATA_837070_WORKAROUND   1

/* ARMv8-M secure side port related definitions. */
#define secureconfigMAX_SECURE_CONTEXTS         5

/* Optional functions - most linkers will remove
unused functions anyway. */
#define INCLUDE_vTaskPrioritySet                1
#define INCLUDE_uxTaskPriorityGet               1
#define INCLUDE_vTaskDelete                     1
#define INCLUDE_vTaskSuspend                    1
#define INCLUDE_xResumeFromISR                  1
#define INCLUDE_vTaskDelayUntil                 1
#define INCLUDE_vTaskDelay                      1
#define INCLUDE_xTaskGetSchedulerState          1
#define INCLUDE_xTaskGetCurrentTaskHandle       1
#define INCLUDE_uxTaskGetStackHighWaterMark     0
#define INCLUDE_uxTaskGetStackHighWaterMark2    0
#define INCLUDE_xTaskGetIdleTaskHandle          0
#define INCLUDE_eTaskGetState                   0
#define INCLUDE_xEventGroupSetBitFromISR        1
#define INCLUDE_xTimerPendFunctionCall          0
#define INCLUDE_xTaskAbortDelay                 0
#define INCLUDE_xTaskGetHandle                  0
#define INCLUDE_xTaskResumeFromISR              1

/* 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] or 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, or 0 to keep the tick interrupt running at all times. Low power tickless implementations are not provided for all FreeRTOS ports.

configUSE_IDLE_HOOK

Set to 1 if you wish to use an idle hook, 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_DAEMON_TASK_STARTUP_HOOK

If configUSE_TIMERS and configUSE_DAEMON_TASK_STARTUP_HOOK are both set to 1 then the application must define a hook function that has the exact name and prototype as shown below. The hook function will be called exactly once when the RTOS daemon task (also known as the timer service task) executes for the first time. Any application initialisation code that needs the RTOS to be running can be placed in the hook function.

```
void void vApplicationDaemonTaskStartupHook( void
);
```

configUSE_SB_COMPLETED_CALLBACK

Setting configUSE_SB_COMPLETED_CALLBACK() includes xStreamBufferCreateWithCallback() and xStreamBufferCreateStaticWithCallback() (and their message buffer equivalents) in the build. Stream and message buffers created with these function can have their own unique send complete and receive complete callbacks, whereas stream and message buffers created with xStreamBufferCreate() and xStreamBufferCreateStatic() (and their message buffer equivalents) all share the callbacks defined by the sbSEND_COMPLETED() and sbRECEVE_COMPLETED() macros. configUSE_SB_COMPLETED_CALLBACK defaults to 0 for backward compatibility.

configUSE_TICK_HOOK

Set to 1 if you wish to use an tick hook, or 0 to omit an tick hook.

configCPU_CLOCK_HZ

Enter the frequency in Hz at which the *internal* clock that drives 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.

configSYSTICK_CLOCK_HZ

Optional parameter for ARM Cortex-M ports only.

By default ARM Cortex-M ports generate the RTOS tick interrupt from the Cortex-M SysTick timer. Most Cortex-M MCUs run the SysTick timer at the same frequency as the MCU itself - when that is the case configSYSTICK_CLOCK_HZ is not needed and should be left undefined. If the SysTick timer is clocked at a different frequency to the MCU core then set configCPU_CLOCK_HZ to the MCU clock frequency, as normal, and configSYSTICK_CLOCK_HZ to the SysTick clock frequency.

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 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 a little RAM within the RTOS kernel so this value should not be set any higher than actually required by your application.

The maximum permissible value will be capped if configUSE_PORT_OPTIMISED_TASK_SELECTION is set to 1.

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() and xTaskCreateStatic() functions, 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).

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() and 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.

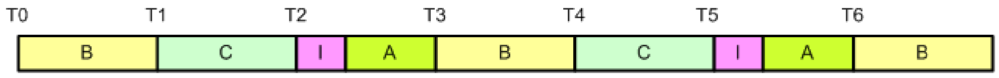
configIDLE_SHOULD_YIELD

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 application creates tasks that run at the idle priority.

If configUSE_TIME_SLICING is set to 1 (or undefined) then tasks that share the same priority will time slice. If none of the tasks get preempted then it might be assumed that each task at a given priority will be allocated an equal amount of processing time - and if the priority is above the idle priority then this is indeed the case.

When tasks share the idle priority the behaviour can be slightly different. If configIDLE_SHOULD_YIELD is set to 1 then the idle task will yield immediately if any other task at the idle priority is 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:



The diagram above shows the execution pattern of four tasks that are all running 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 consumed some of the current time slice. This results in task I and task A effectively sharing the same 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 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 to 0 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 (if none of the tasks get pre-empted) - but at the cost of a greater proportion of the total processing time being allocated to the idle task.

configUSE_TASK_NOTIFICATIONS

Setting configUSE_TASK_NOTIFICATIONS to 1 (or leaving configUSE_TASK_NOTIFICATIONS undefined) will include direct to task notification functionality and its associated API in the build.

Setting configUSE_TASK_NOTIFICATIONS to 0 will exclude direct to task notification functionality and its associated API from the build.

Each task consumes 8 additional bytes of RAM when direct to task notifications are included in the build.

configTASK_NOTIFICATION_ARRAY_ENTRIES

Each RTOS task has an array of task notifications. configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array.

Prior to FreeRTOS V10.4.0 tasks only had a single notification value, not an array of values, so for backward compatibility configTASK_NOTIFICATION_ARRAY_ENTRIES defaults to 1 if it is left undefined.

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 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() and vQueueUnregisterQueue() for more information.

configUSE_QUEUE_SETS

Set to 1 to include queue set 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 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 macros from the build, and in so doing allowing validation that no pre version 8.0.0 names are being used.

configNUM_THREAD_LOCAL_STORAGE_POINTERS

Sets the number of indexes in each task's thread local storage array.

configUSE_MINI_LIST_ITEM

MiniListItem_t is used for start and end marker nodes in a FreeRTOS list and ListItem_t is used for all other nodes in a FreeRTOS list. When configUSE_MINI_LIST_ITEM is set to 0, MiniListItem_t and ListItem_t are both the same. When configUSE_MINI_LIST_ITEM is set to 1, MiniListItem_t contains 3 fewer fields than ListItem_t which saves some RAM at the cost of violating strict aliasing rules which some compilers depend on for optimization. If left undefined, configUSE_MINI_LIST_ITEM defaults to 1 for backward compatibility.

configSTACK_DEPTH_TYPE

Sets the type used to specify the stack depth in calls to xTaskCreate(), and various other places stack sizes are used (for example, when returning the stack high water mark).

Older versions of FreeRTOS specified stack sizes using variables of type UBaseType_t, but that was found to be too restrictive on 8-bit microcontrollers. configSTACK_DEPTH_TYPE removes that restriction by enabling application developers to specify the type to use.

configMESSAGE_BUFFER_LENGTH_TYPE

FreeRTOS Message buffers use variables of type configMESSAGE_BUFFER_LENGTH_TYPE to store the length of each message. If configMESSAGE_BUFFER_LENGTH_TYPE is not defined then it will default to size_t. If the messages stored in a message buffer will never be larger than 255 bytes then defining configMESSAGE_BUFFER_LENGTH_TYPE to uint8_t will save 3 bytes per message on a 32-bit microcontroller. Likewise if the messages stored in a message buffer will never be larger than 65535 bytes then defining configMESSAGE_BUFFER_LENGTH_TYPE to uint16_t will save 2 bytes per message on a 32-bit microcontroller.

configSUPPORT_STATIC_ALLOCATION

If configSUPPORT_STATIC_ALLOCATION is set to 1 then RTOS objects can be created using RAM provided by the application writer.

If configSUPPORT_STATIC_ALLOCATION is set to 0 then RTOS objects can only be created using RAM allocated from the FreeRTOS heap.

If configSUPPORT_STATIC_ALLOCATION is left undefined it will default to 0.

If configSUPPORT_STATIC_ALLOCATION is set to 1 then the application writer must also provide two callback functions: vApplicationGetIdleTaskMemory() to provide the memory for use by the RTOS Idle task, and (if configUSE_TIMERS is set to 1) vApplicationGetTimerTaskMemory() to provide memory for use by the RTOS Daemon/Timer Service task. Examples are provided below.

```
/* configSUPPORT_STATIC_ALLOCATION is set to
1, so the application must provide an
implementation of
vApplicationGetIdleTaskMemory() to provide the
memory that is
used by the Idle task. */
void vApplicationGetIdleTaskMemory(
StaticTask_t **ppxIdleTaskTCBBuffer,

StackType_t **ppxIdleTaskStackBuffer,
                                    uint32_t
*pulIdleTaskStackSize )
{
/* If the buffers to be provided to the Idle
task are declared inside this
function then they must be declared static -
otherwise they will be allocated on
the stack and so not exists after this
function exits. */
static StaticTask_t xIdleTaskTCB;
static StackType_t uxIdleTaskStack[
configMINIMAL_STACK_SIZE ];

    /* Pass out a pointer to the StaticTask_t
structure in which the Idle task's
    state will be stored. */
    *ppxIdleTaskTCBBuffer = &xIdleTaskTCB

    /* Pass out the array that will be used as
the Idle task's stack. */
    *ppxIdleTaskStackBuffer = uxIdleTaskStack;

    /* Pass out the size of the array pointed
to by *ppxIdleTaskStackBuffer.
    Note that, as the array is necessarily of
type StackType_t,
    configMINIMAL_STACK_SIZE is specified in
words, not bytes. */
    *pulIdleTaskStackSize =
configMINIMAL_STACK_SIZE;
}
/*-----------------------------------------------
---------------*/

/* configSUPPORT_STATIC_ALLOCATION and
configUSE_TIMERS are both set to 1, so the
application must provide an implementation of
vApplicationGetTimerTaskMemory()
to provide the memory that is used by the
Timer service task. */
void vApplicationGetTimerTaskMemory(
StaticTask_t **ppxTimerTaskTCBBuffer,

StackType_t **ppxTimerTaskStackBuffer,
                                     uint32_t
*pulTimerTaskStackSize )
{
/* If the buffers to be provided to the Timer
task are declared inside this
function then they must be declared static -
otherwise they will be allocated on
the stack and so not exists after this
function exits. */
static StaticTask_t xTimerTaskTCB;
static StackType_t uxTimerTaskStack[
configTIMER_TASK_STACK_DEPTH ];

    /* Pass out a pointer to the StaticTask_t
structure in which the Timer
    task's state will be stored. */
    *ppxTimerTaskTCBBuffer = &xTimerTaskTCB

    /* Pass out the array that will be used as
the Timer task's stack. */
    *ppxTimerTaskStackBuffer =
uxTimerTaskStack;

    /* Pass out the size of the array pointed
to by *ppxTimerTaskStackBuffer.
    Note that, as the array is necessarily of
type StackType_t,
    configTIMER_TASK_STACK_DEPTH is specified
in words, not bytes. */
    *pulTimerTaskStackSize =
configTIMER_TASK_STACK_DEPTH;
}

Examples of the callback functions that must
be provided by the application to

supply the RAM used by the Idle and Timer
Service tasks if
configSUPPORT_STATIC_ALLOCATION

is set to 1.
```

See the Static Vs Dynamic Memory Allocation page for more information.

configSUPPORT_DYNAMIC_ALLOCATION

If configSUPPORT_DYNAMIC_ALLOCATION is set to 1 then RTOS objects can be created using RAM that is automatically allocated from the FreeRTOS heap.

If configSUPPORT_DYNAMIC_ALLOCATION is set to 0 then RTOS objects can only be created using RAM provided by the application writer.

If configSUPPORT_DYNAMIC_ALLOCATION is left undefined it will default to 1.

See the Static Vs Dynamic Memory Allocation page for more information.

configTOTAL_HEAP_SIZE

The total amount of RAM available in the FreeRTOS heap.

This value will only be used if configSUPPORT_DYNAMIC_ALLOCATION is set to 1 and the application makes use of one of the sample memory allocation schemes provided in the FreeRTOS source code download. See the memory configuration section for further details.

configAPPLICATION_ALLOCATED_HEAP

By default the FreeRTOS heap is declared by FreeRTOS and placed in memory by the linker. Setting configAPPLICATION_ALLOCATED_HEAP to 1 allows the heap to instead be declared by the application writer, which allows the application writer to place the heap wherever they like in memory.

If heap_1.c, heap_2.c or heap_4.c is used, and configAPPLICATION_ALLOCATED_HEAP is set to 1, then the application writer must provide a uint8_t array with the exact name and dimension as shown below. The array will be used as the FreeRTOS heap. How the array is placed at a specific memory location is dependent on the compiler being used - refer to your compiler's documentation.

```
uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];
```

configSTACK_ALLOCATION_FROM_SEPARATE_HEAP

If configSTACK_ALLOCATION_FROM_SEPARATE_HEAP is set to 1 then stack for any task created using xTaskCreate or xTaskCreateRestricted API is allocated using pvPortMallocStack and freed using vPortFreeStack function. The user need to provide thread-safe implementations of pvPortMallocStack and vPortFreeStack functions. This enables the user to allocate stacks for tasks from a separate region of memory (possibly another heap different from the FreeRTOS heap).

If configSTACK_ALLOCATION_FROM_SEPARATE_HEAP is left undefined it will default to 0.

Example implementation of pvPortMallocStack and vPortFreeStack functions is below:

```
void * pvPortMallocStack( size_t xWantedSize
)
{
    /* Allocate a memory block of size
xWantedSize. The function used for
     * allocating memory must be thread safe.
*/
    return MyThreadSafeMalloc( xWantedSize );
}

void vPortFreeStack( void * pv )
{
    /* Free the memory previously allocated
using pvPortMallocStack. The
     * function used for freeing the memory
must be thread safe. */
    MyThreadSafeFree( pv );
}
```

Example implementation of pvPortMallocStack and vPortFreeStack

functions

configGENERATE_RUN_TIME_STATS

The Run Time Stats 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 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 page for a full description.

configTIMER_TASK_PRIORITY

Sets the priority of the software timer service/daemon task. See the FreeRTOS software timers page for a full description.

configTIMER_QUEUE_LENGTH

Sets the length of the software timer command queue. See the FreeRTOS software timers 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 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.

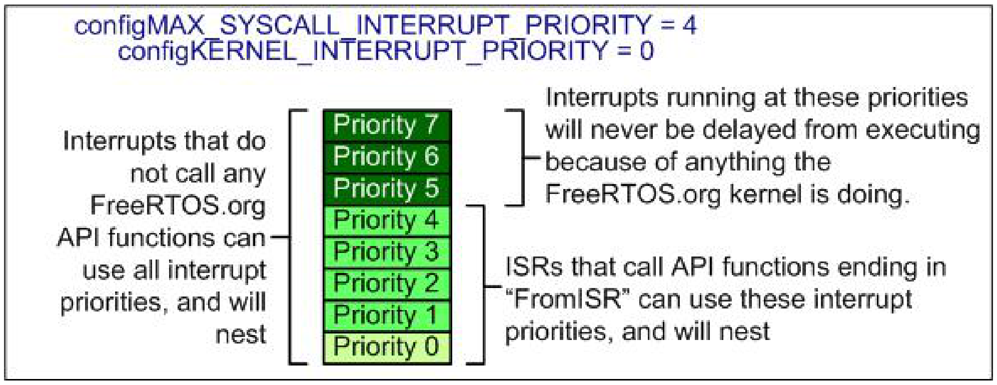
For ports that only implement configKERNEL_INTERRUPT_PRIORITY
configKERNEL_INTERRUPT_PRIORITY sets the interrupt priority used by the RTOS kernel itself. Interrupts that call API functions must also execute at this priority. Interrupts that do not call API functions can execute at higher priorities and 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:


configMAX_SYSCALL_INTERRUPT_PRIORITY = 4
configKERNEL_INTERRUPT_PRIORITY = 0
Interrupts that do not call any FreeRTOS.org API functions can use all interrupt priorities, and will nest
Priority 7
Priority 6
Priority 5
Priority 4
Priority 3
Priority 2
Priority 1
Priority 0
Interrupts running at these priorities will never be delayed from executing because of anything the FreeRTOS.org kernel is doing.
ISRs that call API functions ending in "FromISR" can use these interrupt priorities, and will nest
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. 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 is 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__ )
```

If running FreeRTOS under the control of a debugger, then configASSERT() can be defined to just disable interrupts and sit in a loop, as demonstrated below. That will have the effect of stopping the code on the line that failed the assert test - pausing the debugger will then immediately take you to the offending line so you can see why it failed.

```
/* Define configASSERT() to disable interrupts and
sit in a loop. */
#define configASSERT ( x )     if( ( x ) == 0 ) {
taskDISABLE_INTERRUPTS(); for( ;; ); }
```

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.

configTOTAL_MPU_REGIONS

FreeRTOS MPU (Memory Protection Unit) ports for ARM Cortex-M4 microcontrollers support devices with 16 MPU regions. Set `configTOTAL_MPU_REGIONS` to 16 for devices with 16 MPU regions. If left undefined, it defaults to 8.

configTEX_S_C_B_FLASH

The TEX, Shareable (S), Cacheable (C) and Bufferable (B) bits define the memory type, and where necessary the cacheable and shareable properties of an MPU region. `configTEX_S_C_B_FLASH` allows application writers to override the default values for the for TEX, Shareable (S), Cacheable (C) and Bufferable (B) bits for the MPU region covering Flash. If left undefined, it defaults to 0x07UL which means TEX=000, S=1, C=1, B=1.

configTEX_S_C_B_SRAM

The TEX, Shareable (S), Cacheable (C) and Bufferable (B) bits define the memory type, and where necessary the cacheable and shareable properties of an MPU region. `configTEX_S_C_B_SRAM` allows application writers to override the default values for the for TEX, Shareable (S), Cacheable (C) and Bufferable (B) bits for the

MPU region covering RAM. If left undefined, it defaults to 0x07UL which means TEX=000, S=1, C=1, B=1.

configENFORCE_SYSTEM_CALLS_FROM_KERNEL_ONLY

`configENFORCE_SYSTEM_CALLS_FROM_KERNEL_ONLY` can be defined to 1 to prevent any privilege escalations originating from outside of the kernel code (other than escalations performed by the hardware itself when an interrupt is entered). When `configENFORCE_SYSTEM_CALLS_FROM_KERNEL_ONLY` is set to 1 in `FreeRTOSConfig.h`, variables `__syscalls_flash_start__` and `__syscalls_flash_end__` need to be exported form linker script to indicate the start and end addresses of the system calls memory respectively. It is recommended to define it to 1 for maximum security.

configALLOW_UNPRIVILEGED_CRITICAL_SECTIONS

ARMv7-M MPU ports only (ARM Cortex-M3/4/7).

Set to 0 to prevent unprivileged application tasks from using the `taskENTER_CRITICAL()` macro to create a critical section. Set the constant to 1, or leave it undefined, to allow both privileged and unprivileged tasks to create critical sections. Note: It is recommended to define this constant to 0 for maximum security; because of this, a compiler warning is output if the constant is left undefined.

configENABLE_ERRATA_837070_WORKAROUND

Cortex-M4 MPU ports only.

When using Cortex-M4 MPU ports on a Cortex-M7 r0p0/r0p1 core, set `configENABLE_ERRATA_837070_WORKAROUND` to 1 to ensure that a workaround required for ARM errata 837070 is active.

configHEAP_CLEAR_MEMORY_ON_FREE

If set to 1, then blocks of memory allocated using `pvPortMalloc()` will be cleared when freed using `vPortFree()`. If left undefined, `configHEAP_CLEAR_MEMORY_ON_FREE` defaults to 0 for backward compatibility. We recommend setting `configHEAP_CLEAR_MEMORY_ON_FREE` to 1 for better security.

secureconfigMAX_SECURE_CONTEXTS

ARMv8-M secure side ports only.

Tasks that call secure functions from the non-secure side of an ARMv8-M MCU (ARM Cortex-M23, Cortex-M33 and Cortex-M55) have two contexts – one on the non-secure side and one on the secure-side. Before FreeRTOS v10.4.5, ARMv8-M secure-side ports allocated the structures that reference secure-side contexts at run time. From FreeRTOS V10.4.5 the structures are allocated statically at compile time. secureconfigMAX_SECURE_CONTEXTS sets the number of statically allocated secure contexts. secureconfigMAX_SECURE_CONTEXTS defaults to 8 if left undefined. Applications that only use FreeRTOS code on the non-secure side of ARMv8-M microcontrollers, such as those running third-party code on the secure-side, do not require this constant.

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
```