The FreeRTOS™ Reference Manual

API Functions and Configuration Options

V8.2.3

Contents

1 Task Creation

Modules

- xTaskCreate
- vTaskDelete

TaskHandle_t

task. H

Type by which tasks are referenced. For example, a call to xTaskCreate returns (via a pointer parameter) an TaskHandle_t variable that can then be used as a parameter to vTaskDelete to delete the task.

xTaskCreate

task. h

```
BaseType_t xTaskCreate(
                          TaskFunction_t pvTaskCode,
                          const char * const pcName,
                          unsigned short usStackDepth,
                          void *pvParameters,
                          UBaseType_t uxPriority,
                          TaskHandle_t *pvCreatedTask
                      );
```

Create a new task and add it to the list of tasks that are ready to run.
If you are using FreeRTOS-MPU then it is recommended to use xTaskCreateRestricted() in place of xTaskCreate(). Using xTaskCreate() with FreeRTOS-MPU allows tasks to be created to run in either Privileged or User modes (see the description of uxPriority below). When Privileged mode it used the task will have access to the entire memory map, when User mode is used the task will have access to only its stack. In both cases the MPU will not automatically catch stack overflows, although the standard FreeRTOS stack overflow detection schemes can still be used. xTaskCreateRestricted() permits much greater flexibility.

Parameters:

pvTaskCode	Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).
pcName	A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by configMAX_TASK_NAME_LEN.
usStackDepth	The size of the task stack specified as the number of variables the stack can hold - not the number of bytes. For example, if the stack is 16 bits wide and usStackDepth is defined as 100, 200 bytes will be allocated for stack storage. The stack depth multiplied by the stack width must not exceed the maximum value that can be contained in a variable of type size_t.
pvParameters	Pointer that will be used as the parameter for the task being created.
uxPriority	The priority at which the task should run. Systems that include MPU support can optionally create tasks in a privileged (system) mode by setting bit portPRIVILEGE_BIT of the priority parameter. For example, to create a privileged task at priority 2 the uxPriority parameter should be set to (2 | portPRIVILEGE_BIT).

pvCreatedTask Used to pass back a handle by which the created task can be referenced.

Returns:

pdPASS if the task was successfully created and added to a ready list, otherwise an error code defined in the file projdefs. h

Example usage:

```
/* Task to be created. */
void vTaskCode( void * pvParameters )
{
   for( ;; )
   {
        /* Task code goes here. */
   }
}

/* Function that creates a task. */
void vOtherFunction( void )
{
static unsigned char ucParameterToPass;
TaskHandle_t xHandle = NULL;

   /* Create the task, storing the handle.   Note that the passed parameter
   ucParameterToPass must exist for the lifetime of the task, so in this
   case is declared static.   If it was just an an automatic stack variable
   it might no longer exist, or at least have been corrupted, by the time
   the new task attempts to access it. */
   xTaskCreate( vTaskCode, "NAME", STACK_SIZE, &ucParameterToPass,
tskIDLE_PRIORITY,
                &xHandle );
   configASSERT( xHandle );

   /* Use the handle to delete the task. */
   if( xHandle != NULL )
   {
        vTaskDelete( xHandle );
   }
 }
```

vTaskDelete

task. h

void vTaskDelete(TaskHandle_t xTask);

INCLUDE_vTaskDelete must be defined as 1 for this function to be available. See the configuration section for more information.
Remove a task from the RTOS kernels management. The task being deleted will be removed from all ready, blocked, suspended and event lists.
NOTE: The idle task is responsible for freeing the RTOS kernel allocated memory from tasks that have been deleted. It is therefore important that the idle task is not starved of microcontroller processing time if your application makes any calls to vTaskDelete (). Memory allocated by the task code is not automatically freed, and should be freed before the task is deleted.
See the demo application file death. c for sample code that utilises vTaskDelete ().
Parameters:

xTask The handle of the task to be deleted. Passing NULL will cause the calling task to be deleted.

Example usage:

```
 void vOtherFunction( void )
 {
 TaskHandle_t xHandle = NULL;
      // Create the task, storing the handle.
      xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY,
&xHandle );
      // Use the handle to delete the task.
      if( xHandle != NULL )
      {
            vTaskDelete( xHandle );
      }
 }
```

2 Task Control

Modules

vTaskDelay
vTaskDelayUntil
uxTaskPriorityGet
vTaskPrioritySet
vTaskSuspend
vTaskResume
xTaskResumeFromISR

vTaskDelay

task. H

void vTaskDelay(const TickType_t xTicksToDelay);

INCLUDE_vTaskDelay must be defined as 1 for this function to be available. See the configuration section for more information.
Delay a task for a given number of ticks. The actual time that the task remains blocked depends on the tick rate. The constant portTICK_PERIOD_MS can be used to calculate real time from the tick rate - with the resolution of one tick period.
vTaskDelay() specifies a time at which the task wishes to unblock **relative to** the time at which vTaskDelay() is called. For example, specifying a block period of 100 ticks will cause the task to unblock 100 ticks after vTaskDelay() is called. vTaskDelay() does not therefore provide a good method of controlling the frequency of a periodic task as the path taken through the code, as well as other task and interrupt activity, will effect the frequency at which vTaskDelay() gets called and therefore the time at which the task next executes. See vTaskDelayUntil() for an alternative API function designed to facilitate fixed frequency execution. It does this by specifying an absolute time (rather than a relative time) at which the calling task should unblock.

Parameters:

xTicksToDelay	The amount of time, in tick periods, that the calling task should block.

Example usage:

```
 void vTaskFunction( void * pvParameters )
 {
 /* Block for 500ms. */
 const TickType_t xDelay = 500 / portTICK_PERIOD_MS;

       for( ;; )
       {
            /* Simply toggle the LED every 500ms, blocking between each toggle. */
            vToggleLED();
            vTaskDelay( xDelay );
       }
}
```

vTaskDelayUntil

task. h

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

INCLUDE_vTaskDelayUntil must be defined as 1 for this function to be available. See the configuration section for more information.
Delay a task until a specified time. This function can be used by periodic tasks to ensure a constant execution frequency.
This function differs from vTaskDelay() in one important aspect: vTaskDelay() specifies a time at which the task wishes to unblock *relative* to the time at which vTaskDelay() is called, whereas vTaskDelayUntil() specifies an *absolute* time at which the task wishes to unblock.
vTaskDelay() will cause a task to block for the specified number of ticks from the time vTaskDelay() is called. It is therefore difficult to use vTaskDelay() by itself to generate a fixed execution frequency as the time between a task unblocking following a call to vTaskDelay() and that task next calling vTaskDelay() may not be fixed [the task may take a different path though the code between calls, or may get interrupted or preempted a different number of times each time it executes].
Whereas vTaskDelay() specifies a wake time relative to the time at which the function is called, vTaskDelayUntil() specifies the absolute (exact) time at which it wishes to unblock.
It should be noted that vTaskDelayUntil() will return immediately (without blocking) if it is used to specify a wake time that is already in the past. Therefore a task using vTaskDelayUntil() to execute periodically will have to re-calculate its required wake time if the periodic execution is halted for any reason (for example, the task is temporarily placed into the Suspended state) causing the task to miss one or more periodic executions. This can be detected by checking the variable passed by reference as the pxPreviousWakeTime parameter against the current tick count. This is however not necessary under most usage scenarios.
The constant portTICK_PERIOD_MS can be used to calculate real time from the tick rate - with the resolution of one tick period.
This function must not be called while the RTOS scheduler has been suspended by a call to vTaskSuspendAll().
Parameters:

pxPreviousWakeTime	Pointer to a variable that holds the time at which the task was last unblocked. The variable must be initialised with the current time prior to its first use (see the example below). Following this the variable is automatically updated within vTaskDelayUntil().
xTimeIncrement	The cycle time period. The task will be unblocked at time (*pxPreviousWakeTime + xTimeIncrement). Calling

vTaskDelayUntil with the same xTimeIncrement parameter value will cause the task to execute with a fixed interval period.

Example usage:

```
// Perform an action every 10 ticks.
void vTaskFunction( void * pvParameters )
{
TickType_t xLastWakeTime;
const TickType_t xFrequency = 10;

    // Initialise the xLastWakeTime variable with the current time.
    xLastWakeTime = xTaskGetTickCount();

    for( ;; )
    {
        // Wait for the next cycle.
        vTaskDelayUntil( &xLastWakeTime, xFrequency );

        // Perform action here.
    }
}
```

uxTaskPriorityGet

task. H

UBaseType_t uxTaskPriorityGet(TaskHandle_t xTask);

INCLUDE_vTaskPriorityGet must be defined as 1 for this function to be available. See the configuration section for more information.
Obtain the priority of any task.

Parameters:

xTask Handle of the task to be queried. Passing a NULL handle results in the priority of the calling task being returned.

Returns:

The priority of xTask.

Example usage:

```
 void vAFunction( void )
 {
 TaskHandle_t xHandle;
       // Create a task, storing the handle.
       xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY,
&xHandle );
       // ...
       // Use the handle to obtain the priority of the created task.
       // It was created with tskIDLE_PRIORITY, but may have changed
       // it itself.
       if( uxTaskPriorityGet( xHandle ) != tskIDLE_PRIORITY )
       {
             // The task has changed its priority.
       }
       // ...
       // Is our priority higher than the created task?
       if( uxTaskPriorityGet( xHandle ) < uxTaskPriorityGet( NULL ) )
       {
             // Our priority (obtained using NULL handle) is higher.
       }
 }
```

vTaskPrioritySet

task. h

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

INCLUDE_vTaskPrioritySet must be defined as 1 for this function to be available. See the configuration section for more information.
Set the priority of any task.
A context switch will occur before the function returns if the priority being set is higher than the currently executing task.

Parameters:

xTask	Handle to the task for which the priority is being set. Passing a NULL handle results in the priority of the calling task being set.
uxNewPriority	The priority to which the task will be set.

Example usage:

```
 void vAFunction( void )
 {
 TaskHandle_t xHandle;
       // Create a task, storing the handle.
       xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY,
&xHandle );
       // ...
       // Use the handle to raise the priority of the created task.
       vTaskPrioritySet( xHandle, tskIDLE_PRIORITY + 1 );
       // ...
       // Use a NULL handle to raise our priority to the same value.
       vTaskPrioritySet( NULL, tskIDLE_PRIORITY + 1 );
 }
```

vTaskSuspend

task. H

void vTaskSuspend(TaskHandle_t xTaskToSuspend);

INCLUDE_vTaskSuspend must be defined as 1 for this function to be available. See the configuration section for more information.
Suspend any task. When suspended a task will never get any microcontroller processing time, no matter what its priority.
Calls to vTaskSuspend are not accumulative - i.e. calling vTaskSuspend () twice on the same task still only requires one call to vTaskResume () to ready the suspended task.

Parameters:

xTaskToSuspend	Handle to the task being suspended. Passing a NULL handle will cause the calling task to be suspended.

Example usage:

```
 void vAFunction( void )
 {
 TaskHandle_t xHandle;
       // Create a task, storing the handle.
       xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY,
&xHandle );
       // ...
       // Use the handle to suspend the created task.
       vTaskSuspend( xHandle );
       // ...
       // The created task will not run during this period, unless
       // another task calls vTaskResume( xHandle ).
       //...
       // Suspend ourselves.
       vTaskSuspend( NULL );
       // We cannot get here unless another task calls vTaskResume
       // with our handle as the parameter.
 }
```

vTaskResume

task. H

void vTaskResume(TaskHandle_t xTaskToResume);

INCLUDE_vTaskSuspend must be defined as 1 for this function to be available. See the configuration section for more information.
Resumes a suspended task.
A task that has been suspended by one or more calls to vTaskSuspend () will be made available for running again by a single call to vTaskResume ().

Parameters:

xTaskToResume Handle to the task being readied.

Example usage:

```
 void vAFunction( void )
 {
 TaskHandle_t xHandle;
       // Create a task, storing the handle.
       xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY,
&xHandle );
       // ...
       // Use the handle to suspend the created task.
       vTaskSuspend( xHandle );
       // ...
       // The created task will not run during this period, unless
       // another task calls vTaskResume( xHandle ).
       //...
       // Resume the suspended task ourselves.
       vTaskResume( xHandle );
       // The created task will once again get microcontroller processing
       // time in accordance with its priority within the system.
 }
```

xTaskResumeFromISR

task. H

BaseType_t xTaskResumeFromISR(TaskHandle_t xTaskToResume);

INCLUDE_vTaskSuspend and INCLUDE_xTaskResumeFromISR must be defined as 1 for this function to be available. See the configuration section for more information.
A function to resume a suspended task that can be called from within an ISR.
A task that has been suspended by one of more calls to vTaskSuspend() will be made available for running again by a single call to xTaskResumeFromISR().
xTaskResumeFromISR() should not be used to synchronise a task with an interrupt if there is a chance that the interrupt could arrive prior to the task being suspended - as this can lead to interrupts being missed. Use of a semaphore as a synchronisation mechanism would avoid this eventuality.

Parameters:

xTaskToResume Handle to the task being readied.

Returns:

pdTRUE if resuming the task should result in a context switch, otherwise pdFALSE. This is used by the ISR to determine if a context switch may be required following the ISR.

Example usage:

```
 TaskHandle_t xHandle;

 void vAFunction( void )
 {
      // Create a task, storing the handle.
      xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY,
&xHandle );

      // ... Rest of code.
 }

 void vTaskCode( void *pvParameters )
 {
      // The task being suspended and resumed.
      for( ;; )
      {
           // ... Perform some function here.
```

```
        // The task suspends itself.
        vTaskSuspend( NULL );

        // The task is now suspended, so will not reach here until the ISR resumes it.
    }
}


void vAnExampleISR( void )
{
BaseType_t xYieldRequired;

    // Resume the suspended task.
    xYieldRequired = xTaskResumeFromISR( xHandle );

    if( xYieldRequired == pdTRUE )
    {
        // We should switch context so the ISR returns to a different task.
        // NOTE:  How this is done depends on the port you are using.  Check
        // the documentation and examples for your port.
        portYIELD_FROM_ISR();
    }
}
```

3 Task Utilities

Modules

- uxTaskGetSystemState
- xTaskGetCurrentTaskHandle
- xTaskGetIdleTaskHandle
- uxTaskGetStackHighWaterMark
- eTaskGetState
- pcTaskGetTaskName
- xTaskGetTickCount
- xTaskGetTickCountFromISR
- xTaskGetSchedulerState
- uxTaskGetNumberOfTasks
- vTaskList
- vTaskStartTrace
- ulTaskEndTrace
- vTaskGetRunTimeStats
- vTaskSetApplicationTaskTag
- xTaskGetApplicationTaskTag
- xTaskCallApplicationTaskHook
- pvTaskGetThreadLocalStoragePointer
- vTaskSetThreadLocalStoragePointer
- vTaskSetTimeOutState
- xTaskCheckForTimeOut

uxTaskGetSystemState

task.h

```
UBaseType_t uxTaskGetSystemState(
                        TaskStatus_t * const pxTaskStatusArray,
                        const UBaseType_t uxArraySize,
                        unsigned long * const pulTotalRunTime );
```

configUSE_TRACE_FACILITY must be defined as 1 in FreeRTOSConfig.h for uxTaskGetSystemState() to be available.
uxTaskGetSystemState() populates an TaskStatus_t structure for each task in the system. TaskStatus_t structures contain, among other things, members for the task handle, task name, task priority, task state, and total amount of run time consumed by the task.
NOTE: This function is intended for debugging use only as its use results in the scheduler remaining suspended for an extended period.

Parameters:

pxTaskStatusArray	A pointer to an array of TaskStatus_t structures. The array must contain at least one TaskStatus_t structure for each task that is under the control of the RTOS. The number of tasks under the control of the RTOS can be determined using the uxTaskGetNumberOfTasks() API function.
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 ofTaskStatus_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 *pulTotalRunTime is set by uxTaskGetSystemState() to the total run time (as defined by the run time stats clock) since the target booted. pulTotalRunTime can be set to NULL to omit the total run time value.

Returns:

The number of TaskStatus_t structures that were populated by uxTaskGetSystemState(). This 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.

Example usage:

```
/* This example demonstrates how a human readable table of run time stats
information is generated from raw data provided by uxTaskGetSystemState().
```

```
The human readable table is written to pcWriteBuffer.  (see the vTaskList()
API function which actually does just this). */
void vTaskGetRunTimeStats( signed char *pcWriteBuffer )
{
TaskStatus_t *pxTaskStatusArray;
volatile UBaseType_t uxArraySize, x;
unsigned long ulTotalRunTime, ulStatsAsPercentage;

    /* Make sure the write buffer does not contain a string. */
    *pcWriteBuffer = 0x00;

    /* Take a snapshot of the number of tasks in case it changes while this
    function is executing. */
    uxArraySize = uxCurrentNumberOfTasks();

    /* Allocate a TaskStatus_t structure for each task.  An array could be
    allocated statically at compile time. */
    pxTaskStatusArray = pvPortMalloc( uxArraySize * sizeof( TaskStatus_t ) );

    if( pxTaskStatusArray != NULL )
    {
        /* Generate raw status information about each task. */
        uxArraySize = uxTaskGetSystemState( pxTaskStatusArray,
                                            uxArraySize,
                                            &ulTotalRunTime );

        /* For percentage calculations. */
        ulTotalRunTime /= 100UL;

        /* Avoid divide by zero errors. */
        if( ulTotalRunTime > 0 )
        {
            /* For each populated position in the pxTaskStatusArray array,
            format the raw data as human readable ASCII data. */
            for( x = 0; x < uxArraySize; x++ )
            {
                /* What percentage of the total run time has the task used?
                This will always be rounded down to the nearest integer.
                ulTotalRunTimeDiv100 has already been divided by 100. */
                ulStatsAsPercentage =
                        pxTaskStatusArray[ x ].ulRunTimeCounter / ulTotalRunTime;

                if( ulStatsAsPercentage > 0UL )
                {
```

```
                sprintf( pcWriteBuffer, "%s\t\t%lu\t\t%lu%%\r\n",
                                        pxTaskStatusArray[ x ].pcTaskName,
                                        pxTaskStatusArray[ x ].ulRunTimeCounter,
                                        ulStatsAsPercentage );
            }
            else
            {
                /* If the percentage is zero here then the task has
                consumed less than 1% of the total run time. */
                sprintf( pcWriteBuffer, "%s\t\t%lu\t\t<1%%\r\n",
                                        pxTaskStatusArray[ x ].pcTaskName,
                                        pxTaskStatusArray[ x ].ulRunTimeCounter );
            }

            pcWriteBuffer += strlen( ( char * ) pcWriteBuffer );
        }
    }

    /* The array is no longer needed, free the memory it consumes. */
    vPortFree( pxTaskStatusArray );
  }
}
```

The TaskStatus_t definition

```
typedef struct xTASK_STATUS
{
    /* The handle of the task to which the rest of the information in the
    structure relates. */
    TaskHandle_t xHandle;

    /* A pointer to the task's name.  This value will be invalid if the task was
    deleted since the structure was populated! */
    const signed char *pcTaskName;

    /* A number unique to the task. */
    UBaseType_t xTaskNumber;

    /* The state in which the task existed when the structure was populated. */
    eTaskState eCurrentState;

    /* The priority at which the task was running (may be inherited) when the
    structure was populated. */
    UBaseType_t uxCurrentPriority;

    /* The priority to which the task will return if the task's current priority
    has been inherited to avoid unbounded priority inversion when obtaining a
    mutex.  Only valid if configUSE_MUTEXES is defined as 1 in
    FreeRTOSConfig.h. */
    UBaseType_t uxBasePriority;

    /* The total run time allocated to the task so far, as defined by the run
    time stats clock.  Only valid when configGENERATE_RUN_TIME_STATS is
    defined as 1 in FreeRTOSConfig.h. */
    unsigned long ulRunTimeCounter;

    /* The minimum amount of stack space that has remained for the task since
    the task was created.  The closer this value is to zero the closer the task
    has come to overflowing its stack. */
    unsigned short usStackHighWaterMark;
} TaskStatus_t;
```

xTaskGetCurrentTaskHandle

task.h

TaskHandle_t xTaskGetCurrentTaskHandle(void);

INCLUDE_xTaskGetCurrentTaskHandle must be set to 1 for this function to be available.

Returns:

The handle of the currently running (calling) task.

xTaskGetIdleTaskHandle

task.h

TaskHandle_t xTaskGetIdleTaskHandle(void);

INCLUDE_xTaskGetIdleTaskHandle must be set to 1 for this function to be available.

Returns:

The task handle associated with the Idle task. The Idle task is created automatically when the RTOS scheduler is started.

eTaskGetState

task.h

eTaskState eTaskGetState(TaskHandle_t xTask);

Returns as an enumerated type the state in which a task existed at the time eTaskGetState() was executed.
INCLUDE_eTaskGetState must be set to 1 in FreeRTOSConfig.h for eTaskGetState() to be available.

Parameters:

xTask The handle of the subject task (the task being queried).

Returns:

The table below lists the value that eTaskGetState() will return for each possible state that the task referenced by the xTask parameter can exist in.

State	Returned Value
Ready	eReady
Running	eRunning (the calling task is querying its own priority)
Blocked	eBlocked
Suspended	eSuspended
Deleted	eDeleted (the tasks TCB is waiting to be cleaned up)

pcTaskGetTaskName

task.h

char * pcTaskGetTaskName(TaskHandle_t xTaskToQuery);

INCLUDE_pcTaskGetTaskName must be set to 1 in FreeRTOSConfig.h for pcTaskGetTaskName() to be available.

Parameters:

xTaskToQuery The handle of the task being queried. xTaskToQuery can be set to NULL to query the name of the calling task.

Returns:

A pointer to the subject task's name, which is a standard NULL terminated C string.

xTaskGetTickCount

task.h

volatile TickType_t xTaskGetTickCount(void);
This function cannot be called from an ISR. Use xTaskGetTickCountFromISR() instead.

Returns:

The count of ticks since vTaskStartScheduler was called.

xTaskGetTickCountFromISR

task.h

volatile TickType_t xTaskGetTickCountFromISR(void);
A version of xTaskGetTickCount() that can be called from an ISR.

Returns:

The count of ticks since vTaskStartScheduler was called.

xTaskGetSchedulerState

task.h

BaseType_t xTaskGetSchedulerState(void);

Returns:

One of the following constants (defined within task.h):
taskSCHEDULER_NOT_STARTED, taskSCHEDULER_RUNNING,
taskSCHEDULER_SUSPENDED.
INCLUDE_xTaskGetSchedulerState or configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for this function to be available.

uxTaskGetNumberOfTasks

task.h

UBaseType_t uxTaskGetNumberOfTasks(void);

Returns:

The number of tasks that the RTOS kernel is currently managing. This includes all ready, blocked and suspended tasks. A task that has been deleted but not yet freed by the idle task will also be included in the count.

vTaskList

task.h

void vTaskList(char *pcWriteBuffer);

configUSE_TRACE_FACILITY and configUSE_STATS_FORMATTING_FUNCTIONS must be defined as 1 in FreeRTOSConfig.h for this function to be available. See the configuration section for more information.
NOTE: This function will disable interrupts for its duration. It is not intended for normal application runtime use but as a debug aid.
vTaskList() calls uxTaskGetSystemState(), then formats the raw data generated by uxTaskGetSystemState() into a human readable (ASCII) table that shows the state of each task, including the task's stack high water mark (the smaller the high water mark number the closer the task has come to overflowing its stack). Click here to see an example of the output generated.
In the ASCII table the following letters are used to denote the state of a task:

- 'B' - Blocked
- 'R' - Ready
- 'D' - Deleted (waiting clean up)
- 'S' - Suspended, or Blocked without a timeout

vTaskList() is a utility function provided for convenience only. It is not considered part of the kernel. SeevTaskGetRunTimeStats() for a utility function that generates a similar table of run time task utilisation information.

Parameters:

pcWriteBuffer	A buffer into which the above mentioned details will be written, in ASCII form. This buffer is assumed to be large enough to contain the generated report. Approximately 40 bytes per task should be sufficient.

vTaskStartTrace

task.h

void vTaskStartTrace(char * pcBuffer, unsigned long ulBufferSize);

[The function relates to the legacy trace utility - which was removed in FreeRTOS V7.1.0 - users may find the newer Trace Hook Macros easier and more powerful to use.]
Starts a RTOS kernel activity trace. The trace logs the identity of which task is running when.
The trace file is stored in binary format. A separate DOS utility called convtrce.exe is used to convert this into a tab delimited text file which can be viewed and plotted in a spread sheet.

Parameters:

pcBuffer	The buffer into which the trace will be written.
ulBufferSize	The size of pcBuffer in bytes. The trace will continue until either the buffer in full, or ulTaskEndTrace() is called.

ulTaskEndTrace

task.h

unsigned long ulTaskEndTrace(void);

[The function relates to the legacy trace utility - which was removed in FreeRTOS V7.1.0 - users may find the newer Trace Hook Macros easier and more powerful to use.]
Stops an RTOS kernel activity trace. See vTaskStartTrace().

Returns:

The number of bytes that have been written into the trace buffer.

vTaskGetRunTimeStats

task.h

void vTaskGetRunTimeStats(char *pcWriteBuffer);

See the Run Time Stats page for a full description of this feature.
configGENERATE_RUN_TIME_STATS and configUSE_STATS_FORMATTING_FUNCTIONS must both be defined as 1 for this function to be available. The application must also then provide definitions for portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() and portGET_RUN_TIME_COUNTER_VALUE to configure a peripheral timer/counter and return the timers current count value respectively. The counter should be at least 10 times the frequency of the tick count.
NOTE: This function will disable interrupts for its duration. It is not intended for normal application runtime use but as a debug aid.
vTaskGetRunTimeStats() calls uxTaskGetSystemState(), then formats the raw data generated by uxTaskGetSystemState() into a human readable (ASCII) table that shows the amount of time each task has spent in the Running state (how much CPU time each task has consumed). The data is provided as both an absolute and a percentage value. The resolution of the absolute value is dependent on the frequency of the run time stats clock provided by the application.
vTaskGetRunTimeStats() is a utility function provided for convenience only. It is not considered part of the kernel. See vTaskList() for a utility function that generates information on the state of each task.

Parameters:

pcWriteBuffer	A buffer into which the execution times will be written, in ASCII form. This buffer is assumed to be large enough to contain the generated report. Approximately 40 bytes per task should be sufficient.

vTaskGetRunTimeStats

task.h

void vTaskGetRunTimeStats(char *pcWriteBuffer);

See the Run Time Stats page for a full description of this feature.
configGENERATE_RUN_TIME_STATS and configUSE_STATS_FORMATTING_FUNCTIONS must both be defined as 1 for this function to be available. The application must also then provide definitions for portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() and portGET_RUN_TIME_COUNTER_VALUE to configure a peripheral timer/counter and return the timers current count value respectively. The counter should be at least 10 times the frequency of the tick count.
NOTE: This function will disable interrupts for its duration. It is not intended for normal application runtime use but as a debug aid.
vTaskGetRunTimeStats() calls uxTaskGetSystemState(), then formats the raw data generated by uxTaskGetSystemState() into a human readable (ASCII) table that shows the amount of time each task has spent in the Running state (how much CPU time each task has consumed). The data is provided as both an absolute and a percentage value. The resolution of the absolute value is dependent on the frequency of the run time stats clock provided by the application.
vTaskGetRunTimeStats() is a utility function provided for convenience only. It is not considered part of the kernel. See vTaskList() for a utility function that generates information on the state of each task.

Parameters:

pcWriteBuffer	A buffer into which the execution times will be written, in ASCII form. This buffer is assumed to be large enough to contain the generated report. Approximately 40 bytes per task should be sufficient.

vTaskSetApplicationTaskTag

task. h

```
void vTaskSetApplicationTaskTag(
                                TaskHandle_t xTask,
                                TaskHookFunction_t pxTagValue );
```

configUSE_APPLICATION_TASK_TAG must be defined as 1 for this function to be available. See the configuration section for more information.
A 'tag' value can be assigned to each task. This value is for the use of the application only - the RTOS kernel itself does not make use of it in any way. The RTOS trace macros documentation page provides a good example of how an application might make use of this feature.

Parameters:

xTask	The handle of the task to which a tag value is being assigned. Passing xTask as NULL causes the tag to be assigned to the calling task.
pxTagValue	The value being assigned to the task tag. This is of type TaskHookFunction_t to permit a function pointer to be assigned as the tag, although any value can actually be assigned. See the example below.

Example usage:

```
/* In this example an integer is set as the task tag value.
See the RTOS trace hook macros documentation page for an
example how such an assignment can be used. */
void vATask( void *pvParameters )
{
   /* Assign a tag value of 1 to myself. */
   vTaskSetApplicationTaskTag( NULL, ( void * ) 1 );

   for( ;; )
   {
      /* Rest of task code goes here. */
   }
}
/*-----------------------------------------------*/

/* In this example a callback function is being assigned as the task tag.
First define the callback function - this must have type TaskHookFunction_t
```

```
as per this example. */
static BaseType_t prvExampleTaskHook( void * pvParameter )
{
   /* Perform some action - this could be anything from logging a value,
   updating the task state, outputting a value, etc. */

   return 0;
}

/* Now define the task that sets prvExampleTaskHook as its hook/tag value.
This is in fact registering the task callback, as described on the
xTaskCallApplicationTaskHook() documentation page. */
void vAnotherTask( void *pvParameters )
{
   /* Register our callback function. */
   vTaskSetApplicationTaskTag( NULL, prvExampleTaskHook );

   for( ;; )
   {
      /* Rest of task code goes here. */
   }
}

/* As an example use of the hook (callback) we can get the RTOS kernel to call the
hook function of each task that is being switched out during a reschedule. */
#define traceTASK_SWITCHED_OUT() xTaskCallApplicationTaskHook( pxCurrentTCB,
                                                               0 )
```

xTaskGetApplicationTaskTag

task. h

TaskHookFunction_t xTaskGetApplicationTaskTag(TaskHandle_t xTask);
configUSE_APPLICATION_TASK_TAG must be defined as 1 for this function to be available. See the configuration section for more information.
Returns the 'tag' value associated with a task. The meaning and use of the tag value is defined by the application writer. The RTOS kernel itself will not normally access the tag value.
This function is intended for advanced users only.

Parameters:

xTask The handle of the task being queried. A task can query its own tag value by using NULL as the parameter value.

Returns:

The 'tag' value of the task being queried.

Example usage:

```
/* In this example, an integer is set as the task tag value. */
void vATask( void *pvParameters )
{
    /* Assign a tag value of 1 to the currently executing task.
    The (void *) cast is used to prevent compiler warnings. */
    vTaskSetApplicationTaskTag( NULL, ( void * ) 1 );

    for( ;; )
    {
        /* Rest of task code goes here. */
    }
}

void vAFunction( void )
{
TaskHandle_t xHandle;
int iReturnedTaskHandle;

   /* Create a task from the vATask() function, storing the handle to the
   created task in the xTask variable. */

   /* Create the task. */
   if( xTaskCreate(
```

```
                vATask,             /* Pointer to the function that implements
                                       the task. */
                "Demo task",        /* Text name given to the task. */
                STACK_SIZE,         /* The size of the stack that should be created
                                       for the task.    This is defined in words, not
                                       bytes. */
                NULL,               /* The task does not use the
                                      parameter. */
                TASK_PRIORITY,    /* The priority to assign to the newly created
                                       task. */
                &xHandle            /* The handle to the task being created will be
                                       placed in xHandle. */
                ) == pdPASS )
    {
        /* The task was created successfully.    Delay for a short period to allow
        the task to run. */
        vTaskDelay( 100 );

        /* What tag value is assigned to the task?    The returned tag value is
        stored in an integer, so cast to an integer to prevent compiler
        warnings. */
        iReturnedTaskHandle = ( int ) xTaskGetApplicationTaskTag( xHandle );
    }
}
```

xTaskCallApplicationTaskHook

task. h

```
BaseType_t xTaskCallApplicationTaskHook(
                                        TaskHandle_t xTask,
                                        void *pvParameter );
```

configUSE_APPLICATION_TASK_TAG must be defined as 1 for this function to be available. See the configuration section for more information.
A 'tag' value can be assigned to each task. Normally this value is for the use of the application only and the RTOS kernel does not access it. However, it is possible to use the tag to assign a hook (or callback) function to a task - the hook function being executed by calling xTaskCallApplicationTaskHook(). Each task can define its own callback, or simply not define a callback at all.
Although it is possible to use the first function parameter to call the hook function of any task, the most common use of task hook function is with the trace hook macros, as per the example given below.
Task hook functions must have type TaskHookFunction_t, that is take a void * parameter, and return a value of type BaseType_t. The void * parameter can be used to pass any information into the hook function.

Parameters:

xTask	The handle of the task whose hook function is being called. Passing NULL as xTask will call the hook function assocaited with the currently executing task.
pvParameter	The value to pass to the hook function. This can be a pointer to a structure, or simply a numeric value.

Example usage:

```
/* In this example a callback function is being assigned as the task tag.
First define the callback function - this must have type TaskHookFunction_t
as per this example. */
static BaseType_t prvExampleTaskHook( void * pvParameter )
{
    /* Perform some action - this could be anything from logging a value,
    updating the task state, outputting a value, etc. */

    return 0;
}

/* Now define the task that sets prvExampleTaskHook as its hook/tag value.
```

```
This is in fact registering the task callback, as described on the
xTaskCallApplicationTaskHook() documentation page. */
void vAnotherTask( void *pvParameters )
{
    /* Register our callback function. */
    vTaskSetApplicationTaskTag( NULL, prvExampleTaskHook );

    for( ;; )
    {
        /* Rest of task code goes here. */
    }
}

/* As an example use of the hook (callback) we can get the RTOS kernel to
call the hook function of each task that is being switched out during a
reschedule. */
#define traceTASK_SWITCHED_OUT() xTaskCallApplicationTaskHook( pxCurrentTCB,
0 )
```

pvTaskGetThreadLocalStoragePointer

task. H

```
void *pvTaskGetThreadLocalStoragePointer(
                                        TaskHandle_t xTaskToQuery,
                                        BaseType_t xIndex );
```

Retrieves a value from a task's thread local storage array.
This function is intended for advanced users only.

Parameters:

xTaskToQuery	The handle of the task from which the thread local data is being read. A task can read its own thread local data by using NULL as the parameter value.
xIndex	The index into the thread local storage array from which data is being read. The number of available array indexes is set by theconfigNUM_THREAD_LOCAL_STORAGE_POINTERS comp ile time configuration constant in FreeRTOSConfig.h.

Returns:

The values stored in index position xIndex of the thread local storage array of task xTaskToQuery.

Example usage:
See the examples provided on the thread local storage array documentation page.

vTaskSetThreadLocalStoragePointer

task. H

```
void vTaskSetThreadLocalStoragePointer( TaskHandle_t xTaskToSet,
                                        BaseType_t xIndex,
                                        void *pvValue )
```

Sets a value in a task's thread local storage array.
This function is intended for advanced users only.

Parameters:

xTaskToSet	The handle of the task to which the thread local data is being written. A task can write to its own thread local data by using NULL as the parameter value.
xIndex	The index into the thread local storage array to which data is being written. The number of available array indexes is set by theconfigNUM_THREAD_LOCAL_STORAGE_POINTERS compile time configuration constant in FreeRTOSConfig.h.
pvValue	The value to write into the index specified by the xIndex parameter.

Example usage:
See the examples provided on the thread local storage array documentation page.

vTaskSetTimeOutState

task.h

void vTaskSetTimeOutState(TimeOut_t * const pxTimeOut);

This function is intended for advanced users only.
A task can enter the Blocked state to wait for an event. Typically, the task will not wait in the Blocked state indefinitely, but instead a timeout period will be specified. The task will be removed from the Blocked state if the timeout period expires before the event the task is waiting for occurs.
If a task enters and exits the Blocked state more than once while it is waiting for the event to occur then the timeout used each time the task enters the Blocked state must be adjusted to ensure the total of all the time spent in the Blocked state does not exceed the originally specified timeout period. xTaskCheckForTimeOut() performs the adjustment, taking into account occasional occurrences such as tick count overflows, which would otherwise make a manual adjustment prone to error.
vTaskSetTimeOutState() is used with xTaskCheckForTimeOut(). vTaskSetTimeOutState() is called to set the initial condition, after which xTaskCheckForTimeOut() can be called to check for a timeout condition, and adjust the remaining block time if a timeout has not occurred.

Parameters:

pxTimeOut	A pointer to a structure that will be initialized to hold information necessary to determine if a timeout has occurred.

Example usage:
An example is provided on the xTaskCheckForTimeOut() documentation page.

xTaskCheckForTimeOut

task.h

```
BaseType_t xTaskCheckForTimeOut( TimeOut_t * const pxTimeOut,
                                 TickType_t * const pxTicksToWait );
```

This function is intended for advanced users only.
A task can enter the Blocked state to wait for an event. Typically, the task will not wait in the Blocked state indefinitely, but instead a timeout period will be specified. The task will be removed from the Blocked state if the timeout period expires before the event the task is waiting for occurs.
If a task enters and exits the Blocked state more than once while it is waiting for the event to occur then the timeout used each time the task enters the Blocked state must be adjusted to ensure the total of all the time spent in the Blocked state does not exceed the originally specified timeout period. xTaskCheckForTimeOut() performs the adjustment, taking into account occasional occurrences such as tick count overflows, which would otherwise make a manual adjustment prone to error.
xTaskCheckForTimeOut() is used with vTaskSetTimeOutState(). vTaskSetTimeOutState() is called to set the initial condition, after which xTaskCheckForTimeOut() can be called to check for a timeout condition, and adjust the remaining block time if a timeout has not occurred.

Parameters:

pxTimeOut	A pointer to a structure that holds information necessary to determine if a timeout has occurred. pxTimeOut is initialized using vTaskSetTimeOutState().
pxTicksToWait	Used to pass out an adjusted block time, which is the block time that remains after taking into account the time already spent in the Blocked state.

Returns:

If pdTRUE is returned then no block time remains, and a timeout has occurred.
If pdFALSE is returned then some block time remains, so a timeout has not occurred.

Example usage:

```
/* Driver library function used to receive uxWantedBytes from an Rx buffer that
is filled by a UART interrupt.    If there are not enough bytes in the Rx buffer
then the task enters the Blocked state until it is notified that more data has
been placed into the buffer.    If there is still not enough data then the task
re-enters the Blocked state, and xTaskCheckForTimeOut() is used to re-calculate
the Block time to ensure the total amount of time spent in the Blocked state does
not exceed MAX_TIME_TO_WAIT. This continues until either the buffer contains at
```

```
least uxWantedBytes bytes, or the total amount of time spent in the Blocked state
reaches MAX_TIME_TO_WAIT â€“ at which point the task reads however many bytes are
available up to a maximum of uxWantedBytes. */
size_t xUART_Receive( uint8_t *pucBuffer, size_t uxWantedBytes )
{
size_t uxReceived = 0;
TickType_t xTicksToWait = MAX_TIME_TO_WAIT;
TimeOut_t xTimeOut;

    /* Initialize xTimeOut.  This records the time at which this function was
    entered. */
    vTaskSetTimeOutState( &xTimeOut );

    /* Loop until the buffer contains the wanted number of bytes, or a timeout
    occurs. */
    while( UART_bytes_in_rx_buffer( pxUARTInstance ) < uxWantedBytes )
    {
        /* The buffer didn't contain enough data so this task is going to
        enter the Blocked state.  Adjusting xTicksToWait to account for any time
        that has been spent in the Blocked state within this function so far to
        ensure the total amount of time spent in the Blocked state does not exceed
        MAX_TIME_TO_WAIT. */
        if( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) != pdFALSE )
        {
            /* Timed out before the wanted number of bytes were available, exit the
            loop. */
            break;
        }

        /* Wait for a maximum of xTicksToWait ticks to be notified that the receive
        interrupt has placed more data into the buffer. */
        ulTaskNotifyTake( pdTRUE, xTicksToWait );
    }

    /* 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;
}
```

4 Kernel Control

Modules

- taskYIELD
- taskENTER_CRITICAL
- taskEXIT_CRITICAL
- taskDISABLE_INTERRUPTS
- taskENABLE_INTERRUPTS
- vTaskStartScheduler
- vTaskEndScheduler
- vTaskSuspendAll
- xTaskResumeAll
- vTaskStepTick

taskYIELD

task. H

Macro for forcing a context switch.

taskENTER_CRITICAL

task. H

Macro to mark the start of a critical code region. Preemptive context switches cannot occur when in a critical region.

NOTE: This may alter the stack (depending on the portable implementation) so must be used with care!

taskEXIT_CRITICAL

task. H

Macro to mark the end of a critical code region. Preemptive context switches cannot occur when in a critical region.

NOTE: This may alter the stack (depending on the portable implementation) so must be used with care!

taskDISABLE_INTERRUPTS

task. H

Macro to disable all maskable interrupts.

taskENABLE_INTERRUPTS

task. H

Macro to enable microcontroller interrupts.

vTaskStartScheduler

task. H

void vTaskStartScheduler(void);

Starts the RTOS scheduler. After calling the RTOS kernel has control over which tasks are executed and when.
The idle task and optionally the timer daemon task are created automatically when the RTOS scheduler is started.
vTaskStartScheduler() will only return if there is insufficient RTOS heap available to create the idle or timer daemon tasks.
All the RTOS demo application projects contain examples of using vTaskStartScheduler(), normally in the main() function within main.c.

Example usage:

```
void vAFunction( void )
{
    // Tasks can be created before or after starting the RTOS
    scheduler
    xTaskCreate( vTaskCode,
                 "NAME",
                 STACK_SIZE,
                 NULL,
                 tskIDLE_PRIORITY,
                 NULL );
    // Start the real time scheduler.
    vTaskStartScheduler();
    // Will not get here unless there is insufficient RAM.
}
```

vTaskEndScheduler

task. H

void vTaskEndScheduler(void);

NOTE: This has only been implemented for the x86 Real Mode PC port.
Stops the RTOS kernel tick. All created tasks will be automatically deleted and multitasking (either preemptive or cooperative) will stop. Execution then resumes from the point where vTaskStartScheduler() was called, as if vTaskStartScheduler() had just returned.
See the demo application file main. c in the demo/PC directory for an example that uses vTaskEndScheduler ().
vTaskEndScheduler () requires an exit function to be defined within the portable layer (see vPortEndScheduler () in port. c for the PC port). This performs hardware specific operations such as stopping the RTOS kernel tick.
vTaskEndScheduler () will cause all of the resources allocated by the RTOS kernel to be freed - but will not free resources allocated by application tasks.

Example usage:

```
 void vTaskCode( void * pvParameters )
 {
       for( ;; )
       {
             // Task code goes here.
             // At some point we want to end the real time kernel processing
             // so call ...
             vTaskEndScheduler ();
       }
 }
 void vAFunction( void )
 {
       // Create at least one task before starting the RTOS kernel.
       xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY,
NULL );
       // Start the real time kernel with preemption.
       vTaskStartScheduler();
       // Will only get here when the vTaskCode () task has called
       // vTaskEndScheduler ().    When we get here we are back to single task
       // execution.
 }
```

vTaskSuspendAll

task. H

void vTaskSuspendAll(void);

Suspends the scheduler without disabling interrupts. Context switches will not occur while the scheduler is suspended. RTOS ticks that occur while the scheduler is suspended will be held pending until the scheduler has been unsuspended using a call to xTaskResumeAll().
API functions that have the potential to cause a context switch (for example, vTaskDelayUntil(), xQueueSend(), etc.) must **not** be called while the scheduler is suspended.

Example usage:

```
void vTask1( void * pvParameters )
{
	for( ;; )
	{
		// Task code goes here.
		// ...
		// At some point the task wants to perform a long operation during
		// which it does not want to get swapped out.	It cannot use
		// taskENTER_CRITICAL ()/taskEXIT_CRITICAL () as the length of the
		// operation may cause interrupts to be missed - including the
		// ticks.
		// Prevent the RTOS kernel swapping out the task.
		vTaskSuspendAll ();
		// Perform the operation here.	There is no need to use critical
		// sections as we have all the microcontroller processing time.
		// During this time interrupts will still operate and the RTOS kernel
		// tick count will be maintained.
		// ...
		// The operation is complete.	Restart the RTOS kernel.
		xTaskResumeAll ();
	}
}
```

xTaskResumeAll

task. H

BaseType_t xTaskResumeAll(void);

Resumes the scheduler after it was suspended using a call to vTaskSuspendAll(). xTaskResumeAll() only resumes the scheduler. It does not unsuspend tasks that were previously suspended by a call to vTaskSuspend().

Returns:

If resuming the scheduler caused a context switch then pdTRUE is returned, otherwise pdFALSE is returned.

Example usage:

```
 void vTask1( void * pvParameters )
 {
      for( ;; )
      {
           /* Task code goes here. */
           /* ... */
           /* At some point the task wants to perform a long operation
           during which it does not want to get swapped out.    It cannot
           use taskENTER_CRITICAL ()/taskEXIT_CRITICAL () as the length
           of the operation may cause interrupts to be missed -
           including the ticks.

           Prevent the RTOS kernel swapping out the task. */
           vTaskSuspendAll();

           /* Perform the operation here.    There is no need to use critical
           sections as we have all the microcontroller processing time.
           During this time interrupts will still operate and the real
           time RTOS kernel tick count will be maintained. */
           /* ... */
           /* The operation is complete.    Restart the RTOS kernel.    We want to force
           a context switch - but there is no point if resuming the scheduler
           caused a context switch already. */
           if( !xTaskResumeAll () )
           {
                taskYIELD ();
           }
      }
 }
```

vTaskStepTick

task.h

```
void vTaskStepTick( TickType_t xTicksToJump );
```

If the RTOS is configured to use tickless idle functionality then the tick interrupt will be stopped, and the microcontroller placed into a low power state, whenever the Idle task is the only task able to execute. Upon exiting the low power state the tick count value must be corrected to account for the time that passed while it was stopped.
If a FreeRTOS port includes a
default portSUPPRESS_TICKS_AND_SLEEP() implementation, then vTaskStepTick() is used internally to ensure the correct tick count value is maintained. vTaskStepTick() is a public API function to allow the default portSUPPRESS_TICKS_AND_SLEEP() implementation to be overridden, and for a portSUPPRESS_TICKS_AND_SLEEP() to be provided if the port being used does not provide a default.
The configUSE_TICKLESS_IDLE configuration constant must be set to 1 for vTaskStepTick() to be available.

Parameters:

xTicksToJump	The number of RTOS ticks that have passed since the tick interrupt was stopped. For correct operation the parameter must be less than or equal to the portSUPPRESS_TICKS_AND_SLEEP() parameter.

Returns:

None.

Example usage:

The example shows calls being made to several functions. Only vTaskStepTick() is part of the FreeRTOS API. The other functions are specific to the clocks and power saving modes available on the hardware in use, and as such, must be provided by the application writer.

```
/* First define the portSUPPRESS_TICKS_AND_SLEEP().  The parameter is the time, in ticks, until the kernel next needs to execute. */
#define portSUPPRESS_TICKS_AND_SLEEP( xIdleTime )
vApplicationSleep( xIdleTime )

/* Define the function that is called by portSUPPRESS_TICKS_AND_SLEEP(). */
void vApplicationSleep( TickType_t xExpectedIdleTime )
{
unsigned long ulLowPowerTimeBeforeSleep, ulLowPowerTimeAfterSleep;
```

```
        /* Read the current time from a time source that will remain operational
        while the microcontroller is in a low power state. */
        ulLowPowerTimeBeforeSleep = ulGetExternalTime();

        /* Stop the timer that is generating the tick interrupt. */
        prvStopTickInterruptTimer();

        /* Configure an interrupt to bring the microcontroller out of its low power
        state at the time the kernel next needs to execute.    The interrupt must be
        generated from a source that is remains operational when the microcontroller
        is in a low power state. */
        vSetWakeTimeInterrupt( xExpectedIdleTime );

        /* Enter the low power state. */
        prvSleep();

        /* Determine how long the microcontroller was actually in a low power state
        for, which will be less than xExpectedIdleTime if the microcontroller was
        brought out of low power mode by an interrupt other than that configured by
        the vSetWakeTimeInterrupt() call.    Note that the scheduler is suspended
        before portSUPPRESS_TICKS_AND_SLEEP() is called, and resumed when
        portSUPPRESS_TICKS_AND_SLEEP() returns.    Therefore no other tasks will
        execute until this function completes. */
        ulLowPowerTimeAfterSleep = ulGetExternalTime();

        /* Correct the kernels tick count to account for the time the microcontroller
        spent in its low power state. */
        vTaskStepTick( ulLowPowerTimeAfterSleep – ulLowPowerTimeBeforeSleep );

        /* Restart the timer that is generating the tick interrupt. */
        prvStartTickInterruptTimer();
}
```

5 RTOS Task Notifications

API functions:

- xTaskNotifyGive()
- vTaskNotifyGiveFromISR()
- ulTaskNotifyTake()
- xTaskNotify()
- xTaskNotifyAndQuery()
- xTaskNotifyFromISR()
- xTaskNotifyWait()

xTaskNotifyGive()

v8.2.0
task.h

```
BaseType_t xTaskNotifyGive( TaskHandle_t xTaskToNotify );
```

Each RTOS task has a 32-bit notification value which is initialised to zero when the RTOS task is created. An RTOS task notification is an event sent directly to a task that can unblock the receiving task, and optionally update the receiving task's notification value.
xTaskNotifyGive() is a macro intended for use when an RTOS task notification value is being used as a light weight and faster binary or counting semaphore alternative. FreeRTOS semaphores are given using the xSemaphoreGive() API function, xTaskNotifyGive() is the equivalent that instead uses the receiving RTOS task's notification value.
When a task notification value is being used as a binary or counting semaphore equivalent then the task being notified should wait for the notification using the ulTaskNotifyTake() API function rather than thexTaskNotifyWait() API function.
xTaskNotifyGive() must not be called from an interrupt service routine. Use vTaskNotifyGiveFromISR()instead.

Parameters:

xTaskToNotify The handle of the RTOS task being notified, and having its notification value incremented.
RTOS task handles are obtained using the pvCreatedTask parameter of thexTaskCreate() call used to create the task.
The handle of the currently executing RTOS task is returned by thexTaskGetCurrentTaskHandle() API function.

Returns:

xTaskNotifyGive() is a macro that calls xTaskNotify() with the eAction parameter set to eIncrement resulting in all calls returning pdPASS.

Example usage:

[More examples are referenced from the main RTOS task notifications page]

```
/* Prototypes of the two tasks created by main(). */
static void prvTask1( void *pvParameters );
static void prvTask2( void *pvParameters );

/* Handles for the tasks create by main(). */
static TaskHandle_t xTask1 = NULL, xTask2 = NULL;

/* Create two tasks that send notifications back and forth to each other, then
```

```
start the RTOS scheduler. */
void main( void )
{
	xTaskCreate( prvTask1, "Task1", 200, NULL, tskIDLE_PRIORITY, &xTask1 );
	xTaskCreate( prvTask2, "Task2", 200, NULL, tskIDLE_PRIORITY, &xTask2 );
	vTaskStartScheduler();
}
/*-----------------------------------------------------------*/

static void prvTask1( void *pvParameters )
{
	for( ;; )
	{
		/* Send a notification to prvTask2(), bringing it out of the Blocked
		state. */
		xTaskNotifyGive( xTask2 );

		/* Block to wait for prvTask2() to notify this task. */
		ulTaskNotifyTake( pdTRUE, portMAX_DELAY );
	}
}
/*-----------------------------------------------------------*/

static void prvTask2( void *pvParameters )
{
	for( ;; )
	{
		/* Block to wait for prvTask1() to notify this task. */
		ulTaskNotifyTake( pdTRUE, portMAX_DELAY );

		/* Send a notification to prvTask1(), bringing it out of the Blocked
		state. */
		xTaskNotifyGive( xTask1 );
	}
}
```

vTaskNotifyGiveFromISR()

v8.2.0

task.h

```
void vTaskNotifyGiveFromISR(
                        TaskHandle_t xTaskToNotify,
                        BaseType_t *pxHigherPriorityTaskWoken );
```

A version of xTaskNotifyGive() that can be called from an interrupt service routine (ISR). Each RTOS task has a 32-bit notification value which is initialised to zero when the RTOS task is created. An RTOS task notification is an event sent directly to a task that can unblock the receiving task, and optionally update the receiving task's notification value. vTaskNotifyGiveFromISR() is a function intended for use when an RTOS task notification value is being used as a light weight and faster binary or counting semaphore alternative. FreeRTOS semaphores are given from an interrupt using the xSemaphoreGiveFromISR() API function, vTaskNotifyGiveFromISR() is the equivalent that instead uses the receiving RTOS task's notification value.
When a task notification value is being used as a binary or counting semaphore equivalent then the task being notified should wait for the notification using the ulTaskNotifyTake() API function rather than thexTaskNotifyWait() API function.

Parameters:

xTaskToNotify	The handle of the RTOS task being notified, and having its notification value incremented. RTOS task handles are obtained using the pvCreatedTask parameter of the xTaskCreate() call used to create the task. The handle of the currently executing RTOS task is returned by thexTaskGetCurrentTaskHandle() API function.
pxHigherPriorityTaskWoken	*pxHigherPriorityTaskWoken must be initialised to 0. vTaskNotifyGiveFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if sending the notification caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If vTaskNotifyGiveFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited. See the example below.

pxHigherPriorityTaskWoken is an optional parameter and can be set to NULL.

Example usage:

[More examples are referenced from the main RTOS task notifications page]

```
/* This is an example of a transmit function in a generic peripheral driver.  An
RTOS task calls the transmit function, then waits in the Blocked state (so not
using an CPU time) until it is notified that the transmission is complete.  The
transmission is performed by a DMA, and the DMA end interrupt is used to notify
the task. */

static TaskHandle_t xTaskToNotify = NULL;

/* The peripheral driver's transmit function. */
void StartTransmission( uint8_t *pcData, size_t xDataLength )
{
    /* At this point xTaskToNotify should be NULL as no transmission is in
    progress.  A mutex can be used to guard access to the peripheral if
    necessary. */
    configASSERT( xTaskToNotify == NULL );

    /* Store the handle of the calling task. */
    xTaskToNotify = xTaskGetCurrentTaskHandle();

    /* Start the transmission - an interrupt is generated when the transmission
    is complete. */
    vStartTransmit( pcData, xDatalength );
}
/*-----------------------------------------------------------*/

/* The transmit end interrupt. */
void vTransmitEndISR( void )
{
BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* At this point xTaskToNotify should not be NULL as a transmission was
    in progress. */
    configASSERT( xTaskToNotify != NULL );

    /* Notify the task that the transmission is complete. */
    vTaskNotifyGiveFromISR( xTaskToNotify, &xHigherPriorityTaskWoken );

    /* There are no transmissions in progress, so no tasks to notify. */
    xTaskToNotify = NULL;
```

```
        /* If xHigherPriorityTaskWoken is now set to pdTRUE then a context switch
        should be performed to ensure the interrupt returns directly to the highest
        priority task.    The macro used for this purpose is dependent on the port in
        use and may be called portEND_SWITCHING_ISR(). */
        portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
/*-----------------------------------------------------------*/

/* The task that initiates the transmission, then enters the Blocked state (so
not consuming any CPU time) to wait for it to complete. */
void vAFunctionCalledFromATask( uint8_t ucDataToTransmit, size_t xDataLength )
{
uint32_t ulNotificationValue;
const TickType_t xMaxBlockTime = pdMS_TO_TICKS( 200 );

        /* Start the transmission by calling the function shown above. */
        StartTransmission( ucDataToTransmit, xDataLength );

        /* Wait for the transmission to complete. */
        ulNotificationValue = ulTaskNotifyTake( pdFALSE, xMaxBlockTime );

        if( ulNotificationValue == 1 )
        {
                /* The transmission ended as expected. */
        }
        else
        {
                /* The call to ulTaskNotifyTake() timed out. */
        }
}
```

ulTaskNotifyTake()

v8.2.0

task.h

```
uint32_t ulTaskNotifyTake( BaseType_t xClearCountOnExit,
                           TickType_t xTicksToWait );
```

Each RTOS task has a 32-bit notification value which is initialised to zero when the RTOS task is created. An RTOS task notification is an event sent directly to a task that can unblock the receiving task, and optionally update the receiving task's notification value.
ulTaskNotifyTake() is intended for use when a task notification is used as a faster and lighter weight binary or counting semaphore alternative. FreeRTOS semaphores are taken using the xSemaphoreTake() API function, ulTaskNotifyTake() is the equivalent that instead uses a task notification.
When a task is using its notification value as a binary or counting semaphore other tasks and interrupts should send notifications to it using either the xTaskNotifyGive() macro, or the xTaskNotify() function with the function's eAction parameter set to eIncrement (the two are equivalent).
ulTaskNotifyTake() can either clear the task's notification value to zero on exit, in which case the notification value acts like a binary semaphore, or decrement the task's notification value on exit, in which case the notification value acts more like a counting semaphore.
An RTOS task can use ulTaskNotifyTake() to [optionally] block to wait for a the task's notification value to be non-zero. The task does not consume any CPU time while it is in the Blocked state.
Where as xTaskNotifyWait() will return when a notification is pending, ulTaskNotifyTake() will return when the task's notification value is not zero, decrementing the task's notification value before it returns.

Parameters:

xClearCountOnExit If an RTOS task notification is received and xClearCountOnExit is set to pdFALSE then the RTOS task's notification value is decremented before ulTaskNotifyTake() exits. This is equivalent to the value of a counting semaphore being decremented by a successful call to xSemaphoreTake().
If an RTOS task notification is received and xClearCountOnExit is set to pdTRUE then the RTOS task's notification value is reset to 0 before ulTaskNotifyTake() exits. This is equivalent to the value of a binary semaphore being left at zero (or empty, or 'not available') after a successful call to xSemaphoreTake().

xTicksToWait	The maximum time to wait in the Blocked state for a notification to be received if a notification is not already pending when ulTaskNotifyTake() is called. The RTOS task does not consume any CPU time when it is in the Blocked state. The time is specified in RTOS tick periods. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds into a time specified in ticks.

Returns:

The value of the task's notification value before it is decremented or cleared (see the description of xClearCountOnExit).

Example usage:

[More examples are referenced from the main RTOS task notifications page]

```
/* An interrupt handler.  The interrupt handler does not perform any processing,
instead it unblocks a high priority task in which the event that generated the
interrupt is processed.  If the priority of the task is high enough then the
interrupt will return directly to the task (so it will interrupt one task but
return to a different task), so the processing will occur contiguously in time -
just as if all the processing had been done in the interrupt handler itself. */
void vANInterruptHandler( void )
{
BaseType_t xHigherPriorityTaskWoken;

    /* Clear the interrupt. */
    prvClearInterruptSource();

    /* xHigherPriorityTaskWoken must be initialised to pdFALSE.  If calling
    vTaskNotifyGiveFromISR() unblocks the handling task, and the priority of
    the handling task is higher than the priority of the currently running task,
    then xHigherPriorityTaskWoken will automatically get set to pdTRUE. */
    xHigherPriorityTaskWoken = pdFALSE;

    /* Unblock the handling task so the task can perform any processing necessitated
    by the interrupt.  xHandlingTask is the task's handle, which was obtained
    when the task was created. */
    vTaskNotifyGiveFromISR( xHandlingTask, &xHigherPriorityTaskWoken );

    /* Force a context switch if xHigherPriorityTaskWoken is now set to pdTRUE.
    The macro used to do this is dependent on the port and may be called
    portEND_SWITCHING_ISR. */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

```
/*-----------------------------------------------------------*/

/* A task that blocks waiting to be notified that the peripheral needs servicing,
processing all the events pending in the peripheral each time it is notified to
do so. */
void vHandlingTask( void *pvParameters )
{
BaseType_t xEvent;

    for( ;; )
    {
        /* Block indefinitely (without a timeout, so no need to check the function's
        return value) to wait for a notification.   Here the RTOS task notification
        is being used as a binary semaphore, so the notification value is cleared
        to zero on exit.   NOTE!   Real applications should not block indefinitely,
        but instead time out occasionally in order to handle error conditions
        that may prevent the interrupt from sending any more notifications. */
        ulTaskNotifyTake( pdTRUE,           /* Clear the notification value before
                                               exiting. */
                          portMAX_DELAY ); /* Block indefinitely. */

        /* The RTOS task notification is used as a binary (as opposed to a
        counting) semaphore, so only go back to wait for further notifications
        when all events pending in the peripheral have been processed. */
        do
        {
            xEvent = xQueryPeripheral();

            if( xEvent != NO_MORE_EVENTS )
            {
                vProcessPeripheralEvent( xEvent );
            }

        } while( xEvent != NO_MORE_EVENTS );
    }
}
```

xTaskNotify()

v8.2.0

task.h

```
BaseType_t xTaskNotify( TaskHandle_t xTaskToNotify,
                        uint32_t ulValue,
                        eNotifyAction eAction );
```

[If you are using RTOS task notifications to implement binary or counting semaphore type behaviour then use the simplerxTaskNotifyGive() API function instead of xTaskNotify()]
Each RTOS task has a 32-bit notification value which is initialised to zero when the RTOS task is created. xTaskNotify() is used to send an event directly to and potentially unblock an RTOS task, and optionally update the receiving task's notification value in one of the following ways:

- Write a 32-bit number to the notification value
- Add one (increment) the notification value
- Set one or more bits in the notification value
- Leave the notification value unchanged

This function must not be called from an interrupt service routine (ISR).
Use xTaskNotifyFromISR() instead.

Parameters:

xTaskToNotify	The handle of the RTOS task being notified. This is the *subject* task. RTOS task handles are obtained using the pvCreatedTask parameter of thexTaskCreate() call used to create the task. The handle of the currently executing RTOS task is returned by thexTaskGetCurrentTaskHandle() API function.
ulValue	Used to update the notification value of the subject task. See the description of the eAction parameter below.
eAction	An enumerated type that can take one of the values documented in the table below in order to perform the associated action.

eAction Setting	Action Performed
eNoAction	The subject task receives the event, but its notification value is not updated. In this case ulValue is not used.
eSetBits	The notification value of the subject task will be bitwise ORed with ulValue. For example, if ulValue is set to 0x01, then bit 0 will get set within the subject task's notification value. Likewise if ulValue is 0x04 then bit 2 will get set in the subject

	task's notification value. In this way the RTOS task notification mechanism can be used as a light weight alternative to an event group.
eIncrement	The notification value of the subject task will be incremented by one, making the call to xTaskNotify() equivalent to a call to xTaskNotifyGive(). In this case ulValue is not used.
eSetValueWithOverwrite	The notification value of the subject task is unconditionally set to ulValue. In this way the RTOS task notification mechanism is being used as a light weight alternative to xQueueOverwrite().
eSetValueWithoutOverwrite	If the subject task does not already have a notification pending then its notification value will be set to ulValue. If the subject task already has a notification pending then its notification value is not updated as to do so would overwrite the previous value before it was used. In this case the call to xTaskNotify() fails and pdFALSE is returned. In this way the RTOS task notification mechanism is being used as a light weight alternative to xQueueSend() on a queue of length 1.

Returns:

pdPASS is returned in all cases other than when eAction is set to eSetValueWithoutOverwrite and the subject task's notification value cannot be updated because the subject task already had a notification pending.

Example usage:

[More examples are referenced from the main RTOS task notifications page]

```
/* Set bit 8 in the notification value of the task referenced by xTask1Handle. */
xTaskNotify( xTask1Handle, ( 1UL << 8UL ), eSetBits );

/* Send a notification to the task referenced by xTask2Handle, potentially
removing the task from the Blocked state, but without updating the task's
notification value. */
xTaskNotify( xTask2Handle, 0, eNoAction );

/* Set the notification value of the task referenced by xTask3Handle to 0x50,
even if the task had not read its previous notification value. */
xTaskNotify( xTask3Handle, 0x50, eSetValueWithOverwrite );

/* Set the notification value of the task referenced by xTask4Handle to 0xfff,
but only if to do so would not overwrite the task's existing notification
value before the task had obtained it (by a call to xTaskNotifyWait()
or ulTaskNotifyTake()). */
if( xTaskNotify( xTask4Handle, 0xfff, eSetValueWithoutOverwrite ) == pdPASS )
```

```
{
    /* The task's notification value was updated. */
}
else
{
    /* The task's notification value was not updated. */
}
```

xTaskNotifyAndQuery()

v8.2.1

task.h

```
BaseType_t xTaskNotifyAndQuery( TaskHandle_t xTaskToNotify,
                                uint32_t ulValue,
                                eNotifyAction eAction,
                                uint32_t *pulPreviousNotifyValue );
```

[If you are using RTOS task notifications to implement binary or counting semaphore type behaviour then use the simplerxTaskNotifyGive() API function instead of xTaskNotifyAndQuery().]
xTaskNotifyAndQuery() is similar to xTaskNotify(), but contains an additional parameter in which the subject task's previous notification value is returned.
Each RTOS task has a 32-bit notification value which is initialised to zero when the RTOS task is created. xTaskNotifyAndQuery() is used to send an event directly to and potentially unblock an RTOS task, and optionally update the receiving task's notification value in one of the following ways:

- Write a 32-bit number to the notification value
- Add one (increment) the notification value
- Set one or more bits in the notification value
- Leave the notification value unchanged

This function must not be called from an interrupt service routine (ISR).

Parameters:

xTaskToNotify	The handle of the RTOS task being notified. This is the *subject* task. RTOS task handles are obtained using the pvCreatedTask parameter of thexTaskCreate() call used to create the task. The handle of the currently executing RTOS task is returned by thexTaskGetCurrentTaskHandle() API function.
ulValue	Used to update the notification value of the subject task. See the description of the eAction parameter below.
eAction	An enumerated type that can take one of the values documented in the table below in order to perform the associated action.
pulPreviousNotifyValue	Can be used to pass out the subject task's notification value before any bits are modified by the action of

xTaskNotifyAndQuery().
pulPreviousNotifyValue is an optional parameter, and can be set to NULL if it is not required. If pulPreviousNotifyValue is not used then consider usingxTaskNotify() in place of xTaskNotifyAndQuery().

eAction Setting	Action Performed
eNoAction	The subject task receives the event, but its notification value is not updated. In this case ulValue is not used.
eSetBits	The notification value of the subject task will be bitwise ORed with ulValue. For example, if ulValue is set to 0x01, then bit 0 will get set within the subject task's notification value. Likewise if ulValue is 0x04 then bit 2 will get set in the subject task's notification value. In this way the RTOS task notification mechanism can be used as a light weight alternative to an event group.
eIncrement	The notification value of the subject task will be incremented by one, making the call to xTaskNotify() equivalent to a call to xTaskNotifyGive(). In this case ulValue is not used.
eSetValueWithOverwrite	The notification value of the subject task is unconditionally set to ulValue. In this way the RTOS task notification mechanism is being used as a light weight alternative to xQueueOverwrite().
eSetValueWithoutOverwrite	If the subject task does not already have a notification pending then its notification value will be set to ulValue. If the subject task already has a notification pending then its notification value is not updated as to do so would overwrite the previous value before it was used. In this case the call to xTaskNotify() fails and pdFALSE is returned. In this way the RTOS task notification mechanism is being used as a light weight alternative to xQueueSend() on a queue of length 1.

Returns:

pdPASS is returned in all cases other than when eAction is set to eSetValueWithoutOverwrite and the subject task's notification value cannot be updated because the subject task already had a notification pending.

Example usage:

```
uint32_t ulPreviousValue;

/* Set bit 8 in the notification value of the task referenced by xTask1Handle.
Store the task's previous notification value (before bit 8 is set) in
ulPreviousValue. */
```

```
xTaskNotifyAndQuery( xTask1Handle, ( 1UL << 8UL ), eSetBits, &ulPreviousValue );

/* Send a notification to the task referenced by xTask2Handle, potentially
removing the task from the Blocked state, but without updating the task's
notification value.    Store the tasks notification value in ulPreviousValue. */
xTaskNotifyAndQuery( xTask2Handle, 0, eNoAction, &ulPreviousValue );

/* Set the notification value of the task referenced by xTask3Handle to 0x50,
even if the task had not read its previous notification value. The task's
previous notification value is of no interest so the last parameter is set
to NULL. */
xTaskNotifyAndQuery( xTask3Handle, 0x50, eSetValueWithOverwrite,    NULL );

/* Set the notification value of the task referenced by xTask4Handle to 0xfff,
but only if to do so would not overwrite the task's existing notification
value before the task had obtained it (by a call to xTaskNotifyWait()
or ulTaskNotifyTake()).    The task's previous notification value is saved
in ulPreviousValue. */
if( xTaskNotifyAndQuery( xTask4Handle,
                                0xfff,
                                eSetValueWithoutOverwrite,
                                &ulPreviousValue ) == pdPASS )
{
    /* The task's notification value was updated. */
}
else
{
    /* The task's notification value was not updated. */
}
```

xTaskNotifyFromISR()

v8.2.0

task.h

```
BaseType_t xTaskNotifyFromISR( TaskHandle_t xTaskToNotify,
                               uint32_t ulValue,
                               eNotifyAction eAction,
                               BaseType_t *pxHigherPriorityTaskWoken );
```

[If you are using RTOS task notifications to implement binary or counting semaphore type behaviour then use the simplervTaskNotifyGiveFromISR() API function instead of xTaskNotifyFromISR()]
A version of xTaskNotify() that can be called from an ISR.
Each RTOS task has a 32-bit notification value which is initialised to zero when the RTOS task is created. xTaskNotifyFromISR() is used to send an event directly to and potentially unblock an RTOS task, and optionally update the receiving task's notification value in one of the following ways:

- Write a 32-bit number to the notification value
- Add one (increment) the notification value
- Set one or more bits in the notification value
- Leave the notification value unchanged

Parameters:

xTaskToNotify	The handle of the RTOS task being notified. This is the *subject* task. RTOS task handles are obtained using the pvCreatedTask parameter of the xTaskCreate() call used to create the task. The handle of the currently executing RTOS task is returned by thexTaskGetCurrentTaskHandle() API function.
ulValue	Used to update the notification value of the subject task. See the description of the eAction parameter below.
eAction	An enumerated type that can take one of the values documented in the table below in order to perform the associated action.
pxHigherPriorityTaskWoken	*pxHigherPriorityTaskWoken must be initialised to 0. xTaskNotifyFromISR() will set

*pxHigherPriorityTaskWoken to pdTRUE if sending the notification caused a task to unblock, and the unblocked task has a priority higher than the currently running task.
If xTaskNotifyFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited. See the example below.
pxHigherPriorityTaskWoken is an optional parameter and can be set to NULL.

eAction Setting	Action Performed
eNoAction	The subject task receives the event, but its notification value is not updated. In this case ulValue is not used.
eSetBits	The notification value of the subject task will be bitwise ORed with ulValue. For example, if ulValue is set to 0x01, then bit 0 will get set within the subject task's notification value. Likewise if ulValue is 0x04 then bit 2 will get set in the subject task's notification value. In this way the RTOS task notification mechanism can be used as a light weight alternative to an event group.
eIncrement	The notification value of the subject task will be incremented by one, making the call to xTaskNotifyFromISR() equivalent to a call tovTaskNotifyGiveFromISR(). In this case ulValue is not used.
eSetValueWithOverwrite	The notification value of the subject task is unconditionally set to ulValue. In this way the RTOS task notification mechanism is being used as a light weight alternative to xQueueOverwrite().
eSetValueWithoutOverwrite	If the subject task does not already have a notification pending then its notification value will be set to ulValue. If the subject task already has a notification pending then its notification value is not updated as to do so would overwrite the previous value before it was used. In this case the call to xTaskNotify() fails. In this way the RTOS task notification mechanism is being used as a light weight alternative to xQueueSend() on a queue of length 1.

Returns:

pdPASS is returned in all cases other than when eAction is set to eSetValueWithoutOverwrite and the subject task's notification value cannot be updated because the subject task already had a notification pending.

Example usage:

[More examples are referenced from the main RTOS task notifications page]

This example demonstrates how to use xTaskNotifyFromISR() with the eSetBits action. See the xTaskNotify()API documentation page for examples showing how to use the eNoAction, eSetValueWithOverwrite and eSetValueWithoutOverwrite actions.

```
/* The interrupt handler does not perform any processing itself.    Instead it
it unblocks a high priority task in which the events that generated the
interrupt are processed.    If the priority of the task is high enough then the
interrupt will return directly to the task (so it will interrupt one task but
return to a different task), so the processing will occur contiguously in time -
just as if all the processing had been done in the interrupt handler itself.
The status of the interrupting peripheral is sent to the task using an RTOS task
notification. */
void vANInterruptHandler( void )
{
BaseType_t xHigherPriorityTaskWoken;
uint32_t ulStatusRegister;

    /* Read the interrupt status register which has a bit for each interrupt
    source (for example, maybe an Rx bit, a Tx bit, a buffer overrun bit, etc. */
    ulStatusRegister = ulReadPeripheralInterruptStatus();

    /* Clear the interrupts. */
    vClearPeripheralInterruptStatus( ulStatusRegister );

    /* xHigherPriorityTaskWoken must be initialised to pdFALSE.    If calling
    xTaskNotifyFromISR() unblocks the handling task, and the priority of
    the handling task is higher than the priority of the currently running task,
    then xHigherPriorityTaskWoken will automatically get set to pdTRUE. */
    xHigherPriorityTaskWoken = pdFALSE;

    /* Unblock the handling task so the task can perform any processing necessitated
    by the interrupt.    xHandlingTask is the task's handle, which was obtained
    when the task was created.    The handling task's notification value
    is bitwise ORed with the interrupt status - ensuring bits that are already
    set are not overwritten. */
    xTaskNotifyFromISR( xHandlingTask,
                        ulStatusRegister,
                        eSetBits,
                        &xHigherPriorityTaskWoken );

    /* Force a context switch if xHigherPriorityTaskWoken is now set to pdTRUE.
    The macro used to do this is dependent on the port and may be called
```

```
        portEND_SWITCHING_ISR. */
        portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
/* ---------------------------------------------------------- */

/* A task that blocks waiting to be notified that the peripheral needs servicing,
processing all the events pending in the peripheral each time it is notified to
do so. */
void vHandlingTask( void *pvParameters )
{
uint32_t ulInterruptStatus;

        for( ;; )
        {
                /* Block indefinitely (without a timeout, so no need to check the function's
                return value) to wait for a notification.    NOTE!    Real applications
                should not block indefinitely, but instead time out occasionally in order
                to handle error conditions that may prevent the interrupt from sending
                any more notifications. */
                xTaskNotifyWait( 0x00,                  /* Don't clear any bits on entry. */
                                 ULONG_MAX,             /* Clear all bits on exit. */
                                 &ulInterruptStatus, /* Receives the notification value. */
                                 portMAX_DELAY );       /* Block indefinitely. */

                /* Process any bits set in the received notification value.    This assumes
                the peripheral sets bit 1 for an Rx interrupt, bit 2 for a Tx interrupt,
                and bit 3 for a buffer overrun interrupt. */
                if( ( ulInterruptStatus & 0x01 ) != 0x00 )
                {
                        prvProcessRxInterrupt();
                }

                if( ( ulInterruptStatus & 0x02 ) != 0x00 )
                {
                        prvProcessTxInterrupt();
                }

                if( ( ulInterruptStatus & 0x04 ) != 0x00 )
                {
                        prvClearBufferOverrun();
                }
        }
}
```

xTaskNotifyWait()

v8.2.0

task.h

```
BaseType_t xTaskNotifyWait( uint32_t ulBitsToClearOnEntry,
                            uint32_t ulBitsToClearOnExit,
                            uint32_t *pulNotificationValue,
                            TickType_t xTicksToWait );
```

[If you are using RTOS task notifications to implement binary or counting semaphore type behaviour then use the simplerulTaskNotifyTake() API function instead of xTaskNotifyWait()]
Each RTOS task has a 32-bit notification value which is initialised to zero when the RTOS task is created. An RTOS task notification is an event sent directly to a task that can unblock the receiving task, and optionally update the receiving task's notification value in a number of different ways. For example, a notification may overwrite the receiving task's notification value, or just set one or more bits in the receiving task's notification value. See the RTOS task notifications use case documentation for examples.
xTaskNotifyWait() waits, with an optional timeout, for the calling task to receive a notification.
If the receiving RTOS task was already Blocked waiting for a notification when one arrives the receiving RTOS task will be removed from the Blocked state and the notification cleared.

Parameters:

ulBitsToClearOnEntry Any bits set in ulBitsToClearOnEntry will be cleared in the calling RTOS task's notification value on entry to the xTaskNotifyWait() function (before the task waits for a new notification) provided a notification is not already pending when xTaskNotifyWait() is called.
For example, if ulBitsToClearOnEntry is 0x01, then bit 0 of the task's notification value will be cleared on entry to the function.
Setting ulBitsToClearOnEntry to 0xffffffff (ULONG_MAX) will clear all the bits in the task's notification value, effectively clearing the value to 0.

ulBitsToClearOnExit Any bits set in ulBitsToClearOnExit will be cleared in the calling RTOS task's notification value before xTaskNotifyWait() function exits if a notification was received.
The bits are cleared after the RTOS task's notification

value has been saved in *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 RTOS task's notification value. The value copied to *pulNotificationValue is the RTOS task's notification value as it was before any bits were cleared due to the ulBitsToClearOnExit setting.
If the notification value is not required then set pulNotificationValue to NULL.

xTicksToWait — The maximum time to wait in the Blocked state for a notification to be received if a notification is not already pending when xTaskNotifyWait() is called.
The RTOS task does not consume any CPU time when it is in the Blocked state.
The time is specified in RTOS tick periods. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds into a time specified in ticks.

Returns:

pdTRUE if a notification was received, or a notification was already pending when xTaskNotifyWait() was called.
pdFALSE if the call to xTaskNotifyWait() timed out before a notification was received.

Example usage:

[More examples are referenced from the main RTOS task notifications page]

```
/* This task shows bits within the RTOS task notification value being used to pass
different events to the task in the same way that flags in an event group might
be used for the same purpose. */
void vAnEventProcessingTask( void *pvParameters )
{
uint32_t ulNotifiedValue;

    for( ;; )
    {
        /* Block indefinitely (without a timeout, so no need to check the function's
        return value) to wait for a notification.
```

```
            Bits in this RTOS task's notification value are set by the notifying
            tasks and interrupts to indicate which events have occurred. */
            xTaskNotifyWait( 0x00,          /* Don't clear any notification bits on entry. */
                             ULONG_MAX, /* Reset the notification value to 0 on exit. */
                             &ulNotifiedValue, /* Notified value pass out in
                                                  ulNotifiedValue. */
                             portMAX_DELAY );  /* Block indefinitely. */

            /* Process any events that have been latched in the notified value. */

            if( ( ulNotifiedValue & 0x01 ) != 0 )
            {
                /* Bit 0 was set - process whichever event is represented by bit 0. */
                prvProcessBit0Event();
            }

            if( ( ulNotifiedValue & 0x02 ) != 0 )
            {
                /* Bit 1 was set - process whichever event is represented by bit 1. */
                prvProcessBit1Event();
            }

            if( ( ulNotifiedValue & 0x04 ) != 0 )
            {
                /* Bit 2 was set - process whichever event is represented by bit 2. */
                prvProcessBit2Event();
            }

            /* Etc. */
        }
    }
```

6 FreeRTOS-MPU Specific Functions

Modules

- xTaskCreateRestricted()
- vTaskAllocateMPURegions()
- portSWITCH_TO_USER_MODE()

xTaskCreateRestricted

task. H

```
BaseType_t xTaskCreateRestricted(
                                    TaskParameters_t *pxTaskDefinition,
                                    TaskHandle_t *pxCreatedTask );
```

Create a new Memory Protection Unit (MPU) restricted task and add it to the list of tasks that are ready to run.
xTaskCreateRestricted() is intended for use with FreeRTOS-MPU, the demo applications for which contain comprehensive and documented examples of xTaskCreateRestricted() being used.

Parameters:

pxTaskDefinition	Pointer to a structure that defines the task. The structure is described on this page.
pxCreatedTask	Used to pass back a handle by which the created task can be referenced.

Returns:

pdPASS if the task was successfully created and added to a ready list, otherwise an error code defined in the file projdefs.h

Tasks that include MPU support require even more parameters to create than those that don't. Passing each parameter to xTaskCreateRestricted() individually would be unwieldy so instead the structure TaskParameters_t is used to allow the parameters to be configured statically at compile time. The structure is defined in task.h as:

```
typedef struct xTASK_PARAMETERS
{
    TaskFunction_t pvTaskCode;
    const signed char * const pcName;
    unsigned short usStackDepth;
    void *pvParameters;
    UBaseType_t uxPriority;
    portSTACK_TYPE *puxStackBuffer;
    MemoryRegion_t xRegions[ portNUM_CONFIGURABLE_REGIONS ];
} TaskParameters_t;
```

....where MemoryRegion_t is defined as:

```
typedef struct xMEMORY_REGION
{
	void *pvBaseAddress;
	unsigned long ulLengthInBytes;
	unsigned long ulParameters;
} MemoryRegion_t;
```

Following is a description of each structure member:

- pvTaskCode to uxPriority
  These members are exactly the same as the parameters to xTaskCreate() of the same name. In particular uxPriority is used to set both the priority of the task and the mode in which the task will execute. For example, to create a User mode task at priority 2 simply set uxPriority to 2, to create a Privileged mode task at priority 2 set uxPriority to (2 | portPRIVILEGE_BIT).
- puxStackBuffer
  Each time a task is switched in the MPU is dynamically re-configured to define a region that provides the task read and write access to its own stack. MPU regions must meet a number of constraints - in particular, the size and alignment of each region must both be equal to the same power of two value.
  Standard FreeRTOS ports use pvPortMalloc() to allocate a new stacks each time a task is created. Providing a pvPortMalloc() implementation that took care of the MPU data alignment requirements would be possible but would also be complex and inefficient in its RAM usage. To remove the need for this complexity FreeRTOS-MPU allows stacks to be declared statically at compile time. This allows the alignment to be managed using compiler extensions and RAM usage efficiency to be managed by the linker. For example, if using GCC a stack could be declared and correctly aligned using the following code:

  ```
  char cTaskStack[ 1024 ] __attribute__((align(1024));
  ```

  puxStackBuffer would normally be set to the address of the statically declared stack. As an alternative puxStackBuffer can be set to NULL - in which case pvPortMallocAligned() will be called to allocate the task stack and it is the application writers responsibility to provide an implementation of pvPortMallocAligned() that meets the alignment requirements of the MPU.
- xMemoryRegions
  xRegions is an array of MemoryRegion_t structures, each of which defines a single user definable memory region for use by the task being created. The ARM Cortex-M3 FreeRTOS-MPU port defines portNUM_CONFIGURABLE_REGIONS to be 3.
  The pvBaseAddress and ulLengthInBytes members are self explanatory as the start of the memory region and the length of the memory region respectively.
  ulParameters defines how the task is permitted to access the memory region and can take the bitwise OR of the following values:

portMPU_REGION_READ_WRITE
portMPU_REGION_PRIVILEGED_READ_ONLY
portMPU_REGION_READ_ONLY
portMPU_REGION_PRIVILEGED_READ_WRITE
portMPU_REGION_CACHEABLE_BUFFERABLE
portMPU_REGION_EXECUTE_NEVER

Example usage (please refer to the FreeRTOS-MPU demo applications for a much more complete and comprehensive example):

```
/* Declare the stack that will be used by the task.  The stack alignment must
match its size and be a power of 2, so if 128 words are reserved for the stack
then it must be aligned to ( 128 * 4 ) bytes.  This example used GCC syntax. */
static portSTACK_TYPE xTaskStack[ 128 ] __attribute__((aligned(128*4)));

/* Declare an array that will be accessed by the task.  The task should only
be able to read from the array, and not write to it. */
char cReadOnlyArray[ 512 ] __attribute__((aligned(512)));

/* Fill in a TaskParameters_t structure to define the task - this is the
structure passed to the xTaskCreateRestricted() function. */
static const TaskParameters_t xTaskDefinition =
{
    vTaskFunction,     /* pvTaskCode */
    "A task",          /* pcName */
    128,               /* usStackDepth - defined in words, not bytes. */
    NULL,              /* pvParameters */
    1,                 /* uxPriority - priority 1, start in User mode. */
    xTaskStack,        /* puxStackBuffer - the array to use as the task stack. */

    /* xRegions - In this case only one of the three user definable regions is
    actually used.  The parameters are used to set the region to read only. */
    {
        /* Base address     Length                          Parameters */
        { cReadOnlyArray, mainREAD_ONLY_ALIGN_SIZE,
portMPU_REGION_READ_ONLY },
        { 0,                0,
0                           },
        { 0,                0,
0                           },
    }
};
```

```
void main( void )
{
	/* Create the task defined by xTaskDefinition.	NULL is used as the second
	parameter as a task handle is not required. */
	xTaskCreateRestricted( &xTaskDefinition, NULL );

	/* Start the RTOS scheduler. */
	vTaskStartScheduler();

	/* Should not reach here! */
}
```

vTaskAllocateMPURegions

task. h

```
void vTaskAllocateMPURegions(
                    TaskHandle_t xTaskToModify,
                    const MemoryRegion_t * const xRegions );
```

Memory regions are assigned to a restricted task when the task is created using a call to xTaskCreateRestricted(). The regions can then be modified or redefined at run time using vTaskAllocateMPURegions().
vTaskAllocateMPURegions() is intended for use with FreeRTOS-MPU, the demo applications for which contain an example of vTaskAllocateMPURegions() being used.

Parameters:

xTask	The handle of the task being updated.
xRegions	A pointer to an array of MemoryRegion_t structures, each of which contains a single new memory region definitions. The array should be dimensioned using the constant portNUM_CONFIGURABLE_REGIONS, which on the ARM Cortex-M3 is set to 3.

MemoryRegion_t is defined in task.h as:

```
typedef struct xMEMORY_REGION
{
    void *pvBaseAddress;
    unsigned long ulLengthInBytes;
    unsigned long ulParameters;
} MemoryRegion_t;
```

The pvBaseAddress and ulLengthInBytes members are self explanatory as the start of the memory region and the length of the memory region respectively. It is important to note that MPU regions must meet a number of constraints - in particular, the size and alignment of each region must both be equal to the same power of two value.
ulParameters defines how the task is permitted to access the memory region and can take the bitwise OR of the following values:

portMPU_REGION_READ_WRITE
portMPU_REGION_PRIVILEGED_READ_ONLY
portMPU_REGION_READ_ONLY
portMPU_REGION_PRIVILEGED_READ_WRITE
portMPU_REGION_CACHEABLE_BUFFERABLE

portMPU_REGION_EXECUTE_NEVER

Example usage (please refer to the FreeRTOS-MPU demo applications for a much more complete and comprehensive example):

```
/* Define an array that the task will both read from and write to.  Make sure
the size and alignment are appropriate for an MPU region (note this uses GCC
syntax). */
static unsigned char ucOneKByte[ 1024 ] __attribute__((align( 1024 )));

/* Define an array of MemoryRegion_t structures that configures an MPU region
allowing read/write access for 1024 bytes starting at the beginning of the
ucOneKByte array.  The other two of the maximum 3 definable regions are
unused so set to zero. */
static const MemoryRegion_t xAltRegions[ portNUM_CONFIGURABLE_REGIONS ] =
{
    /* Base address      Length      Parameters */
    { ucOneKByte,        1024,       portMPU_REGION_READ_WRITE },
    { 0,                 0,          0 },
    { 0,                 0,          0 }
};

void vATask( void *pvParameters )
{
    /* This task was created such that it has access to certain regions of
    memory as defined by the MPU configuration.  At some point it is
    desired that these MPU regions are replaced with that defined in the
    xAltRegions const struct above.  Use a call to vTaskAllocateMPURegions()
    for this purpose.  NULL is used as the task handle to indicate that this
    function should modify the MPU regions of the calling task. */
    vTaskAllocateMPURegions( NULL, xAltRegions );

    /* Now the task can continue its function, but from this point on can only
    access its stack and the ucOneKByte array (unless any other statically
    defined or shared regions have been declared elsewhere). */
}
```

portSWITCH_TO_USER_MODE

task. H

void portSWITCH_TO_USER_MODE(void);

Sets the calling task into User mode. Once in User mode a task cannot return to Privileged mode.
portSWITCH_TO_USER_MODE() is intended for use with FreeRTOS-MPU, the demo applications for which contain an example of portSWITCH_TO_USER_MODE() being used.

7 Queue Management

Modules

- uxQueueMessagesWaiting
- uxQueueMessagesWaitingFromISR
- uxQueueSpacesAvailable
- xQueueCreate
- vQueueDelete
- xQueueReset
- xQueueSend
- xQueueSendToBack
- xQueueSendToFront
- xQueueReceive
- xQueuePeek
- xQueuePeekFromISR
- xQueueSendFromISR
- xQueueSendToBackFromISR
- xQueueSendToFrontFromISR
- xQueueReceiveFromISR
- vQueueAddToRegistry
- vQueueUnregisterQueue
- xQueueIsQueueEmptyFromISR
- xQueueIsQueueFullFromISR
- xQueueOverwrite
- xQueueOverwriteFromISR

uxQueueMessagesWaiting

queue.h

UBaseType_t uxQueueMessagesWaiting(QueueHandle_t xQueue);

Return the number of messages stored in a queue.

Parameters:

xQueue A handle to the queue being queried.

Returns:

The number of messages available in the queue.

uxQueueMessagesWaitingFromISR

queue.h

UBaseType_t uxQueueMessagesWaiting(QueueHandle_t xQueue);

A version of uxQueueMessagesWaiting() that can be called from an ISR. Return the number of messages stored in a queue.

Parameters:

xQueue A handle to the queue being queried.

Returns:

The number of messages available in the queue.

uxQueueSpacesAvailable

queue.h

UBaseType_t uxQueueSpacesAvailable(QueueHandle_t xQueue);

Return the number of free spaces in a queue.

Parameters:

xQueue A handle to the queue being queried.

Returns:

The number of free spaces available in the queue.

xQueueCreate

queue. h

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

Creates a new queue instance. This allocates the storage required by the new queue and returns a handle for the queue.

Parameters:

uxQueueLength	The maximum number of items that the queue can contain.
uxItemSize	The number of bytes each item in the queue will require. Items are queued by copy, not by reference, so this is the number of bytes that will be copied for each posted item. Each item on the queue must be the same size.

Returns:

If the queue is successfully create then a handle to the newly created queue is returned. If the queue cannot be created then 0 is returned.

Example usage:

```
struct AMessage
{
    char ucMessageID;
    char ucData[ 20 ];
};
void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;
    // Create a queue capable of containing 10 unsigned long values.
    xQueue1 = xQueueCreate( 10, sizeof( unsigned long ) );
    if( xQueue1 == 0 )
    {
        // Queue was not created and must not be used.
    }
    // Create a queue capable of containing 10 pointers to AMessage structures.
    // These should be passed by pointer as they contain a lot of data.
    xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );
    if( xQueue2 == 0 )
```

```
    {
        // Queue was not created and must not be used.
    }
    // ... Rest of task code.
}
```

vQueueDelete

queue.h

void vQueueDelete(QueueHandle_t xQueue);

Delete a queue - freeing all the memory allocated for storing of items placed on the queue.

Parameters:

xQueue A handle to the queue to be deleted.

xQueueReset

queue.h

BaseType_t xQueueReset(QueueHandle_t xQueue);

Resets a queue to its original empty state.

Parameters:

xQueue The handle of the queue being reset

Returns:

Since FreeRTOS V7.2.0 xQueueReset() always returns pdPASS.

xQueueSend

queue.h

```
BaseType_t xQueueSend(
                          QueueHandle_t xQueue,
                          const void * pvItemToQueue,
                          TickType_t xTicksToWait
                      );
```

This is a macro that calls xQueueGenericSend(). It is included for backward compatibility with versions of FreeRTOS that did not include the xQueueSendToFront() and xQueueSendToBack() macros. It is equivalent to xQueueSendToBack().
Post an item on a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See xQueueSendFromISR() for an alternative which may be used in an ISR.

Parameters:

xQueue	The handle to the queue on which the item is to be posted.
pvItemToQueue	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
xTicksToWait	The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if the queue is full and xTicksToWait is set to 0. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required. If INCLUDE_vTaskSuspend is set to '1' then specifying the block time as portMAX_DELAY will cause the task to block indefinitely (without a timeout).

Returns:pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

Example usage:

```
struct AMessage
 {
     char ucMessageID;
     char ucData[ 20 ];
 } xMessage;

 unsigned long ulVar = 10UL;
```

```
void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;
struct AMessage *pxMessage;

    /* Create a queue capable of containing 10 unsigned long values. */
    xQueue1 = xQueueCreate( 10, sizeof( unsigned long ) );

    /* Create a queue capable of containing 10 pointers to AMessage structures.
    These should be passed by pointer as they contain a lot of data. */
    xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );


    /* ... */


    if( xQueue1 != 0 )
    {
        /* Send an unsigned long.  Wait for 10 ticks for space to become
        available if necessary. */
        if( xQueueSend( xQueue1,
                        ( void * ) &ulVar,
                        ( TickType_t ) 10 ) != pdPASS )
        {
            /* Failed to post the message, even after 10 ticks. */
        }
    }

    if( xQueue2 != 0 )
    {
        /* Send a pointer to a struct AMessage object.  Don't block if the
        queue is already full. */
        pxMessage = & xMessage;
        xQueueSend( xQueue2, ( void * ) &pxMessage, ( TickType_t ) 0 );
    }

        /* ... Rest of task code. */
}
```

xQueueSendToBack

queue.h

```
 BaseType_t xQueueSendToBack(
                                    QueueHandle_t xQueue,
                                    const void * pvItemToQueue,
                                    TickType_t xTicksToWait
                                );
```

This is a macro that calls xQueueGenericSend(). It is equivalent to xQueueSend().
Post an item to the back of a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See xQueueSendToBackFromISR () for an alternative which may be used in an ISR.

Parameters:

xQueue	The handle to the queue on which the item is to be posted.
pvItemToQueue	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
xTicksToWait	The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required. If INCLUDE_vTaskSuspend is set to '1' then specifying the block time as portMAX_DELAY will cause the task to block indefinitely (without a timeout).

Returns:

pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

Example usage:

```
struct AMessage
{
    char ucMessageID;
    char ucData[ 20 ];
} xMessage;

unsigned long ulVar = 10UL;

void vATask( void *pvParameters )
```

```
{
QueueHandle_t xQueue1, xQueue2;
struct AMessage *pxMessage;

    /* Create a queue capable of containing 10 unsigned long values. */
    xQueue1 = xQueueCreate( 10, sizeof( unsigned long ) );

    /* Create a queue capable of containing 10 pointers to AMessage
    structures.  These should be passed by pointer as they contain a lot of
    data. */
    xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

    /* ... */

    if( xQueue1 != 0 )
    {
        /* Send an unsigned long.  Wait for 10 ticks for space to become
        available if necessary. */
        if( xQueueSendToBack( xQueue1,
                              ( void * ) &ulVar,
                              ( TickType_t ) 10 ) != pdPASS )
        {
            /* Failed to post the message, even after 10 ticks. */
        }
    }

    if( xQueue2 != 0 )
    {
        /* Send a pointer to a struct AMessage object.  Don't block if the
        queue is already full. 8/
        pxMessage = & xMessage;
        xQueueSendToBack( xQueue2, ( void * ) &pxMessage, ( TickType_t ) 0 );
    }

     /* ... Rest of task code. */
}
```

xQueueSendToFront

Only available from FreeRTOS V4.5.0 onwards.

queue.h

```
 BaseType_t xQueueSendToFront(
                                    QueueHandle_t xQueue,
                                    const void * pvItemToQueue,
                                    TickType_t xTicksToWait
                                );
```

This is a macro that calls xQueueGenericSend().
Post an item to the front of a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See xQueueSendToFrontFromISR () for an alternative which may be used in an ISR.

Parameters:

xQueue	The handle to the queue on which the item is to be posted.
pvItemToQueue	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
xTicksToWait	The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required. If INCLUDE_vTaskSuspend is set to '1' then specifying the block time as portMAX_DELAY will cause the task to block indefinitely (without a timeout).

Returns:

pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

Example usage:

```
struct AMessage
{
    char ucMessageID;
    char ucData[ 20 ];
} xMessage;

unsigned long ulVar = 10UL;
```

```
void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;
struct AMessage *pxMessage;

    /* Create a queue capable of containing 10 unsigned long values. */
    xQueue1 = xQueueCreate( 10, sizeof( unsigned long ) );

    /* Create a queue capable of containing 10 pointers to AMessage
    structures.  These should be passed by pointer as they contain a lot of
    data. */
    xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

    /* ... */

    if( xQueue1 != 0 )
    {
        /* Send an unsigned long.  Wait for 10 ticks for space to become
        available if necessary. */
        if( xQueueSendToFront( xQueue1,
                               ( void * ) &ulVar,
                               ( TickType_t ) 10 ) != pdPASS )
        {
            /* Failed to post the message, even after 10 ticks. */
        }
    }

    if( xQueue2 != 0 )
    {
        /* Send a pointer to a struct AMessage object.  Don't block if the
        queue is already full. */
        pxMessage = & xMessage;
        xQueueSendToFront( xQueue2, ( void * ) &pxMessage, ( TickType_t ) 0 );
    }

        /* ... Rest of task code. */
}
```

xQueueReceive

queue. h

```
BaseType_t xQueueReceive(
                                    QueueHandle_t xQueue,
                                    void *pvBuffer,
                                    TickType_t xTicksToWait
                                );
```

This is a macro that calls the xQueueGenericReceive() function.
Receive an item from a queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.
This function must not be used in an interrupt service routine. See xQueueReceiveFromISR for an alternative that can.

Parameters:

xQueue	The handle to the queue from which the item is to be received.
pvBuffer	Pointer to the buffer into which the received item will be copied.
xTicksToWait	The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call. Setting xTicksToWait to 0 will cause the function to return immediately if the queue is empty. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required. If INCLUDE_vTaskSuspend is set to '1' then specifying the block time as portMAX_DELAY will cause the task to block indefinitely (without a timeout).

Returns:

pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

Example usage:

```
struct AMessage
{
    char ucMessageID;
    char ucData[ 20 ];
} xMessage;
QueueHandle_t xQueue;
// Task to create a queue and post a value.
```

```
void vATask( void *pvParameters )
{
struct AMessage *pxMessage;
    // Create a queue capable of containing 10 pointers to AMessage structures.
    // These should be passed by pointer as they contain a lot of data.
    xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
    if( xQueue == 0 )
    {
        // Failed to create the queue.
    }
    // ...
    // Send a pointer to a struct AMessage object.    Don't block if the
    // queue is already full.
    pxMessage = & xMessage;
    xQueueSend( xQueue, ( void * ) &pxMessage, ( TickType_t ) 0 );
        // ... Rest of task code.
}
// Task to receive from the queue.
void vADifferentTask( void *pvParameters )
{
struct AMessage *pxRxedMessage;
    if( xQueue != 0 )
    {
        // Receive a message on the created queue.    Block for 10 ticks if a
        // message is not immediately available.
        if( xQueueReceive( xQueue, &( pxRxedMessage ), ( TickType_t ) 10 ) )
        {
            // pcRxedMessage now points to the struct AMessage variable posted
            // by vATask.
        }
    }
        // ... Rest of task code.
}
```

xQueuePeek

queue.h

```
BaseType_t xQueuePeek(
                          QueueHandle_t xQueue,
                          void *pvBuffer,
                          TickType_t xTicksToWait
                      );
```

This is a macro that calls the xQueueGenericReceive() function.
Receive an item from a queue without removing the item from the queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.
Successfully received items remain on the queue so will be returned again by the next call, or a call to xQueueReceive().
This macro must not be used in an interrupt service routine.

Parameters:

xQueue	The handle to the queue from which the item is to be received.
pvBuffer	Pointer to the buffer into which the received item will be copied. This must be at least large enough to hold the size of the queue item defined when the queue was created.
xTicksToWait	The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required. If INCLUDE_vTaskSuspend is set to '1' then specifying the block time as portMAX_DELAY will cause the task to block indefinitely (without a timeout).

Returns:

pdTRUE if an item was successfully received (peeked) from the queue, otherwise pdFALSE.

Example usage:

```
struct AMessage
{
    char ucMessageID;
    char ucData[ 20 ];
} xMessage;
```

```
QueueHandle_t xQueue;

// Task to create a queue and post a value.
void vATask( void *pvParameters )
{
struct AMessage *pxMessage;

    // Create a queue capable of containing 10 pointers to AMessage structures.
    // These should be passed by pointer as they contain a lot of data.
    xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
    if( xQueue == 0 )
    {
        // Failed to create the queue.
    }

    // ...

    // Send a pointer to a struct AMessage object.  Don't block if the
    // queue is already full.
    pxMessage = & xMessage;
    xQueueSend( xQueue, ( void * ) &pxMessage, ( TickType_t ) 0 );

    // ... Rest of task code.
}

// Task to peek the data from the queue.
void vADifferentTask( void *pvParameters )
{
struct AMessage *pxRxedMessage;

    if( xQueue != 0 )
    {
        // Peek a message on the created queue.  Block for 10 ticks if a
        // message is not immediately available.
        if( xQueuePeek( xQueue, &( pxRxedMessage ), ( TickType_t ) 10 ) )
        {
            // pcRxedMessage now points to the struct AMessage variable posted
            // by vATask, but the item still remains on the queue.
        }
    }

    // ... Rest of task code.
}
```

xQueuePeekFromISR

queue.h

```
BaseType_t xQueuePeekFromISR(
                                    QueueHandle_t xQueue,
                                    void *pvBuffer,
                                   );
```

A version of xQueuePeek() that can be used from an interrupt service routine (ISR). Receive an item from a queue without removing the item from the queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.
Successfully received items remain on the queue so will be returned again by the next call, or a call to any queue receive function.

Parameters:

xQueue The handle to the queue from which the item is to be received.

pvBuffer Pointer to the buffer into which the received item will be copied. This must be at least large enough to hold the size of the queue item defined when the queue was created.

Returns:

pdTRUE if an item was successfully received (peeked) from the queue, otherwise pdFALSE.

xQueueSendFromISR

queue.h

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

This is a macro that calls xQueueGenericSendFromISR(). It is included for backward compatibility with versions of FreeRTOS that did not include the xQueueSendToBackFromISR() and xQueueSendToFrontFromISR() macros.
Post an item into the back of a queue. It is safe to use this function from within an interrupt service routine.
Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR. In most cases it would be preferable to store a pointer to the item being queued.

Parameters:

xQueue	The handle to the queue on which the item is to be posted.
pvItemToQueue	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
pxHigherPriorityTaskWoken	xQueueSendFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xQueueSendFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited. From FreeRTOS V7.3.0 pxHigherPriorityTaskWoken is an optional parameter and can be set to NULL.

Returns:

pdTRUE if the data was successfully sent to the queue, otherwise errQUEUE_FULL.

Example usage for buffered IO (where the ISR can obtain more than one value per call):

```
void vBufferISR( void )
{
char cIn;
BaseType_t xHigherPriorityTaskWoken;

    /* We have not woken a task at the start of the ISR. */
    xHigherPriorityTaskWoken = pdFALSE;

    /* Loop until the buffer is empty. */
    do
    {
        /* Obtain a byte from the buffer. */
        cIn = portINPUT_BYTE( RX_REGISTER_ADDRESS );

        /* Post the byte. */
        xQueueSendFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );

    } while( portINPUT_BYTE( BUFFER_COUNT ) );

    /* Now the buffer is empty we can switch context if necessary. */
    if( xHigherPriorityTaskWoken )
    {
        /* Actual macro used here is port specific. */
        taskYIELD_FROM_ISR ();
    }
}
```

xQueueSendToBackFromISR

queue.h

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

This is a macro that calls xQueueGenericSendFromISR().
Post an item to the back of a queue. It is safe to use this function from within an interrupt service routine.
Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR.

Parameters:

xQueue	The handle to the queue on which the item is to be posted.
pvItemToQueue	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
pxHigherPriorityTaskWoken	xQueueSendTobackFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xQueueSendToBackFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited. From FreeRTOS V7.3.0 pxHigherPriorityTaskWoken is an optional parameter and can be set to NULL.

Returns:

pdPASS if sending to the queue was successful, otherwise errQUEUE_FULL.

Example usage for buffered IO (where the ISR can obtain more than one value per call):

```
void vBufferISR( void )
{
```

```
char cIn;
BaseType_t xHigherPriorityTaskWoken;

    /* We have not woken a task at the start of the ISR. */
    xHigherPriorityTaskWoken = pdFALSE;

    /* Loop until the buffer is empty. */
    do
    {
        /* Obtain a byte from the buffer. */
        cIn = portINPUT_BYTE( RX_REGISTER_ADDRESS );

        /* Post the byte. */
        xQueueSendToBackFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );

    } while( portINPUT_BYTE( BUFFER_COUNT ) );

    /* Now the buffer is empty we can switch context if necessary. */
    if( xHigherPriorityTaskWoken )
    {
        /* Actual macro used here is port specific. */
        taskYIELD_FROM_ISR ();
    }
}
```

xQueueSendToFrontFromISR

queue.h

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

This is a macro that calls xQueueGenericSendFromISR().
Post an item to the front of a queue. It is safe to use this function from within an interrupt service routine.
Items are queued by copy not reference so it is preferable to either only send small items, or alternatively send a pointer to the item.

Parameters:

xQueue	The handle to the queue on which the item is to be posted.
pvItemToQueue	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
pxHigherPriorityTaskWoken	xQueueSendToFrontFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xQueueSendToFrontFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited. From FreeRTOS V7.3.0 pxHigherPriorityTaskWoken is an optional parameter and can be set to NULL.

Returns:

pdPass if data was successfully sent to the queue, otherwise errQUEUE_FULL.

Example usage:

```
void vBufferISR( void )
{
char cIn;
BaseType_t xHigherPriorityTaskWoken;

    /* We have not woken a task at the start of the ISR. */
    xHigherPriorityTaskWoken = pdFALSE;

    /* Obtain a byte from the buffer. */
    cIn = portINPUT_BYTE( RX_REGISTER_ADDRESS );

    if( cIn == EMERGENCY_MESSAGE )
    {
        /* Post the byte to the front of the queue. */
        xQueueSendToFrontFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );
    }
    else
    {
        /* Post the byte to the back of the queue. */
        xQueueSendToBackFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );
    }

    /* Did sending to the queue unblock a higher priority task? */
    if( xHigherPriorityTaskWoken )
    {
        /* Actual macro used here is port specific. */
        taskYIELD_FROM_ISR ();
    }
}
```

xQueueReceiveFromISR

queue. h

```
BaseType_t xQueueReceiveFromISR
                    (
                        QueueHandle_t xQueue,
                        void *pvBuffer,
                        BaseType_t *pxHigherPriorityTaskWoken
                    );
```

Receive an item from a queue. It is safe to use this function from within an interrupt service routine.

Parameters:

xQueue	The handle to the queue from which the item is to be received.
pvBuffer	Pointer to the buffer into which the received item will be copied.
pxHigherPriorityTaskWoken	A task may be blocked waiting for space to become available on the queue. If xQueueReceiveFromISR causes such a task to unblock *pxHigherPriorityTaskWoken will get set to pdTRUE, otherwise *pxHigherPriorityTaskWoken will remain unchanged. From FreeRTOS V7.3.0 pxHigherPriorityTaskWoken is an optional parameter and can be set to NULL.

Returns:

pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

Example usage:

```
QueueHandle_t xQueue;

/* Function to create a queue and post some values. */
void vAFunction( void *pvParameters )
{
char cValueToPost;
const TickType_t xTicksToWait = ( TickType_t )0xff;

    /* Create a queue capable of containing 10 characters. */
```

```
    xQueue = xQueueCreate( 10, sizeof( char ) );
    if( xQueue == 0 )
    {
        /* Failed to create the queue. */
    }

    /* ... */

    /* Post some characters that will be used within an ISR.  If the queue
    is full then this task will block for xTicksToWait ticks. */
    cValueToPost = 'a';
    xQueueSend( xQueue, ( void * ) &cValueToPost, xTicksToWait );
    cValueToPost = 'b';
    xQueueSend( xQueue, ( void * ) &cValueToPost, xTicksToWait );

    /* ... keep posting characters ... this task may block when the queue
    becomes full. */

    cValueToPost = 'c';
    xQueueSend( xQueue, ( void * ) &cValueToPost, xTicksToWait );
}

/* ISR that outputs all the characters received on the queue. */
void vISR_Routine( void )
{
BaseType_t xTaskWokenByReceive = pdFALSE;
char cRxedChar;

    while( xQueueReceiveFromISR( xQueue,
                                 ( void * ) &cRxedChar,
                                 &xTaskWokenByReceive) )
    {
        /* A character was received.  Output the character now. */
        vOutputCharacter( cRxedChar );

        /* If removing the character from the queue woke the task that was
        posting onto the queue xTaskWokenByReceive will have been set to
        pdTRUE.  No matter how many times this loop iterates only one
        task will be woken. */
    }

    if( xTaskWokenByReceive != pdFALSE )
    {
        /* We should switch context so the ISR returns to a different task.
```

```
            NOTE:    How this is done depends on the port you are using.    Check
            the documentation and examples for your port. */
            taskYIELD ();
        }
}
```

vQueueAddToRegistry

queue.h

```
 void vQueueAddToRegistry(
                                    QueueHandle_t xQueue,
                                    char *pcQueueName,
                                );
```

Assigns a name to a queue and adds the queue to the registry.

Parameters:

xQueue	The handle of the queue being added to the registry.
pcQueueName	The name to be assigned to the queue. This is just a text string used to facilitate debugging. The queue registry only stores a pointer to the string, so the string must be persistent (a global, or preferably in ROM/Flash), not defined on the stack.

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 the queues and semaphores that you want to view using a RTOS kernel aware debugger need to be registered.
Example:

```
void vAFunction( void )
{
QueueHandle_t xQueue;

    /* Create a queue big enough to hold 10 chars. */
    xQueue = xQueueCreate( 10, sizeof( char ) );

    /* We want this queue to be viewable in a RTOS kernel aware debugger,
    so register it. */
    vQueueAddToRegistry( xQueue, "AMeaningfulName" );
}
```

vQueueUnregisterQueue

queue.h

```
 void vQueueUnregisterQueue(
                                      QueueHandle_t xQueue,
                                    );
```

Removes a queue from the queue registry.

Parameters:

xQueue The handle of the queue being removed from the registry.

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 the queues and semaphores that you want to view using a RTOS kernel aware debugger need to be registered.
Example:

```
void vAFunction( void )
{
QueueHandle_t xQueue;

    /* Create a queue big enough to hold 10 chars. */
    xQueue = xQueueCreate( 10, sizeof( char ) );

    /* We want this queue to be viewable in a RTOS kernel aware debugger,
    so register it. */
    vQueueAddToRegistry( xQueue, "AMeaningfulName" );


    /* The queue gets used here. */

    /* At some later time, the queue is going to be deleted, first
    remove it from the registry. */
    vQueueUnregisterQueue( xQueue );
    vQueueDelete( xQueue );
}
```

xQueueIsQueueEmptyFromISR

queue.h

BaseType_t xQueueIsQueueEmptyFromISR(const QueueHandle_t xQueue);

Queries a queue to determine if the queue is empty. This function should only be used in an ISR.

Parameters:

xQueue The handle of the queue being queried

Returns:

pdFALSE if the queue is not empty, or any other value if the queue is empty.

xQueueIsQueueFullFromISR

queue.h

BaseType_t xQueueIsQueueFullFromISR(const QueueHandle_t xQueue);
Queries a queue to determine if the queue is full. This function should only be used in an ISR.

Parameters:

xQueue The handle of the queue being queried

Returns:

pdFALSE if the queue is not full, or any other value if the queue is full.

8 Queue Set

FreeRTOS Queue Set API Functions

- xQueueCreateSet
- xQueueAddToSet
- xQueueRemoveFromSet
- xQueueSelectFromSet
- xQueueSelectFromSetFromISR

xQueueCreateSet()

queue.h

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

configUSE_QUEUE_SETS must be set to 1 in FreeRTOSConfig.h for the xQueueCreateSet() API function to be available.
Queue sets provide a mechanism to allow an RTOS task to block (pend) on a read operation from multiple RTOS queues or semaphores simultaneously. Note that there are simpler alternatives to using queue sets. See the Blocking on Multiple Objects page for more information.
A queue set must be explicitly created using a call to xQueueCreateSet() before it can be used. Once created, standard FreeRTOS queues and semaphores can be added to the set using calls toxQueueAddToSet(). xQueueSelectFromSet() is then used to determine which, if any, of the queues or semaphores contained in the set is in a state where a queue read or semaphore take operation would be successful.
Notes:

- Queues and semaphores **must be empty** when they are added to a queue set. Take particular care when adding objects such as binary semaphores which are created with the semaphore already available [this is the case if the semaphore is created using the vSemaphoreCreateBinary() macro, but not the case if the semaphore is created using the preferred xSemaphoreCreateBinary() function].
- Blocking on a queue set that contains a mutex will not cause the mutex holder to inherit the priority of the blocked task.
- An additional 4 bytes of RAM are required for each space in every queue added to a queue set. Therefore a counting semaphore that has a high maximum count value should not be added to a queue set.
- A receive (in the case of a queue) or take (in the case of a semaphore) operation must not be performed on a member of a queue set unless a call to xQueueSelectFromSet() has first returned a handle to that set member.

Parameters:

uxEventQueueLength	Queue sets store events that occur on the queues and semaphores contained in the set. uxEventQueueLength specifies the maximum number of events that can be queued at once. To be absolutely certain that events are not lost uxEventQueueLength must be set to the sum of the lengths of the queues added to the set, where binary

semaphores and mutexes have a length of 1, and counting semaphores have a length set by their maximum count value. For example:

- If a queue set is to hold a queue of length 5, another queue of length 12, and a binary semaphore, then uxEventQueueLength should be set to (5 + 12 + 1), or 18.
- If a queue set is to hold three binary semaphores then uxEventQueueLength should be set to (1 + 1 + 1), or 3.
- If a queue set is to hold a counting semaphore that has a maximum count of 5, and a counting semaphore that has a maximum count of 3, then uxEventQueueLength should be set to (5 + 3), or 8.

Returns:

If the queue set is created successfully then a handle to the created queue set is returned. Otherwise NULL is returned.

Example usage:

```
/* Define the lengths of the queues that will be added to the queue set. */
#define QUEUE_LENGTH_1          10
#define QUEUE_LENGTH_2          10

/* Binary semaphores have an effective length of 1. */
#define BINARY_SEMAPHORE_LENGTH         1

/* Define the size of the item to be held by queue 1 and queue 2 respectively.
The values used here are just for demonstration purposes. */
#define ITEM_SIZE_QUEUE_1       sizeof( uint32_t )
#define ITEM_SIZE_QUEUE_2       sizeof( something_else_t )

/* The combined length of the two queues and binary semaphore that will be
added to the queue set. */
#define COMBINED_LENGTH ( QUEUE_LENGTH_1 +
                          QUEUE_LENGTH_2 +
                          BINARY_SEMAPHORE_LENGTH )

void vAFunction( void )
{
static QueueSetHandle_t xQueueSet;
QueueHandle_t xQueue1, xQueue2, xSemaphore;
QueueSetMemberHandle_t xActivatedMember;
```

```
uint32_t xReceivedFromQueue1;
something_else_t xReceivedFromQueue2;

    /* Create the queue set large enough to hold an event for every space in
    every queue and semaphore that is to be added to the set. */
    xQueueSet = xQueueCreateSet( COMBINED_LENGTH );

    /* Create the queues and semaphores that will be contained in the set. */
    xQueue1 = xQueueCreate( QUEUE_LENGTH_1, ITEM_SIZE_QUEUE_1 );
    xQueue2 = xQueueCreate( QUEUE_LENGTH_2, ITEM_SIZE_QUEUE_2 );

    /* Create the semaphore that is being added to the set. */
    xSemaphore = xSemaphoreCreateBinary();

    /* Check everything was created. */
    configASSERT( xQueueSet );
    configASSERT( xQueue1 );
    configASSERT( xQueue2 );
    configASSERT( xSemaphore );

    /* Add the queues and semaphores to the set.  Reading from these queues and
    semaphore can only be performed after a call to xQueueSelectFromSet() has
    returned the queue or semaphore handle from this point on. */
    xQueueAddToSet( xQueue1, xQueueSet );
    xQueueAddToSet( xQueue2, xQueueSet );
    xQueueAddToSet( xSemaphore, xQueueSet );

    for( ;; )
    {
        /* Block to wait for something to be available from the queues or
        semaphore that have been added to the set.  Don't block longer than
        200ms. */
        xActivatedMember = xQueueSelectFromSet( xQueueSet,
                                                200 / portTICK_PERIOD_MS );

        /* Which set member was selected?  Receives/takes can use a block time
        of zero as they are guaranteed to pass because xQueueSelectFromSet()
        would not have returned the handle unless something was available. */
        if( xActivatedMember == xQueue1 )
        {
            xQueueReceive( xActivatedMember, &xReceivedFromQueue1, 0 );
            vProcessValueFromQueue1( xReceivedFromQueue1 );
        }
        else if( xActivatedMember == xQueue2 )
```

```
            {
                xQueueReceive( xActivatedMember, &xReceivedFromQueue2, 0 );
                vProcessValueFromQueue2( &xReceivedFromQueue2 );
            }
            else if( xActivatedMember == xSemaphore )
            {
                /* Take the semaphore to make sure it can be "given" again. */
                xSemaphoreTake( xActivatedMember, 0 );
                vProcessEventNotifiedBySemaphore();
                break;
            }
            else
            {
                /* The 200ms block time expired without an RTOS queue or semaphore
                being ready to process. */
            }
        }
    }
```

xQueueAddToSet()

queue.h

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

configUSE_QUEUE_SETS must be set to 1 in FreeRTOSConfig.h for the xQueueAddToSet() API function to be available.
Adds an RTOS queue or semaphore to a queue set that was previously created by a call toxQueueCreateSet().
A receive (in the case of a queue) or take (in the case of a semaphore) operation must not be performed on a member of a queue set unless a call to xQueueSelectFromSet() has first returned a handle to that set member.

Parameters:

xQueueOrSemaphore	The handle of the queue or semaphore being added to the queue set (cast to an QueueSetMemberHandle_t type).
xQueueSet	The handle of the queue set to which the queue or semaphore is being added.

Returns:

If the queue or semaphore was successfully added to the queue set then pdPASS is returned. If the queue could not be successfully added to the queue set because it is already a member of a different queue set then pdFAIL is returned.

Example usage:
See the example on the xQueueCreateSet() documentation page.

xQueueRemoveFromSet()

queue.h

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

configUSE_QUEUE_SETS must be set to 1 in FreeRTOSConfig.h for the xQueueRemoveFromSet() API function to be available.
Remove an RTOS queue or semaphore from a queue set.
An RTOS queue or semaphore can only be removed from a queue set if the queue or semaphore is empty.

Parameters:

xQueueOrSemaphore	The handle of the queue or semaphore being removed from the queue set (cast to an QueueSetMemberHandle_t type).
xQueueSet	The handle of the queue set in which the queue or semaphore is included.

Returns:

If the queue or semaphore was successfully removed from the queue set then pdPASS is returned. If the queue was not in the queue set, or the queue (or semaphore) was not empty, then pdFAIL is returned.

Example usage:

This example assumes xQueueSet is a queue set that has already been created, and xQueue is a queue that has already been created and added to xQueueSet.

```
if( xQueueRemoveFromSet( xQueue, xQueueSet ) != pdPASS )
{
    /* Either xQueue was not a member of the xQueueSet set, or xQueue is
    not empty and therefore cannot be removed from the set. */
}
else
{
    /* The queue was successfully removed from the set. */
}
```

xQueueSelectFromSet()

queue.h

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

configUSE_QUEUE_SETS must be set to 1 in FreeRTOSConfig.h for the xQueueSelectFromSet() API function to be available.
xQueueSelectFromSet() selects from the members of a queue set a queue or semaphore that either contains data (in the case of a queue) or is available to take (in the case of a semaphore). xQueueSelectFromSet() effectively allows a task to block (pend) on a read operation on all the queues and semaphores in a queue set simultaneously.
Notes:

- There are simpler alternatives to using queue sets. See the Blocking on Multiple Objects page for more information.
- Blocking on a queue set that contains a mutex will not cause the mutex holder to inherit the priority of the blocked task.
- A receive (in the case of a queue) or take (in the case of a semaphore) operation must not be performed on a member of a queue set unless a call to xQueueSelectFromSet() has first returned a handle to that set member.

Parameters:

xQueueSet	The queue set on which the task will (potentially) block.
xTicksToWait	The maximum time, in ticks, that the calling task will remain in the Blocked state (with other tasks executing) to wait for a member of the queue set to be ready for a successful queue read or semaphore take operation.

Returns:

xQueueSelectFromSet() will return the handle of a queue (cast to a QueueSetMemberHandle_t type) contained in the queue set that contains data, or the handle of a semaphore (cast to a QueueSetMemberHandle_t type) contained in the queue set that is available, or NULL if no such queue or semaphore exists before before the specified block time expires.

Example usage:
See the example on the xQueueCreateSet() documentation page.

xQueueSelectFromSetFromISR()

queue.h

```
QueueSetMemberHandle_t xQueueSelectFromSetFromISR
                                (
                                        QueueSetHandle_t xQueueSet
                                 );
```

configUSE_QUEUE_SETS must be set to 1 in FreeRTOSConfig.h for the xQueueSelectFromSetFromISR() API function to be available.
A version of xQueueSelectFromSet() that can be used from an interrupt service routine (ISR).

Parameters:

xQueueSet The queue set being queried. It is not possible to block on a read as this function is designed to be used from an interrupt.

Returns:

xQueueSelectFromSetFromISR() will return the handle of a queue (cast to a QueueSetMemberHandle_t type) contained in the queue set that contains data, or the handle of a semaphore (cast to a QueueSetMemberHandle_t type) contained in the queue set that is available, or NULL if no such queue or semaphore exists.

9 Semaphores

Modules

- xSemaphoreCreateBinary [new]
- vSemaphoreCreateBinary [old]
- xSemaphoreCreateCounting
- xSemaphoreCreateMutex
- xSemaphoreCreateRecursiveMutex
- vSemaphoreDelete
- xSemaphoreGetMutexHolder
- xSemaphoreTake
- xSemaphoreTakeFromISR
- xSemaphoreTakeRecursive
- xSemaphoreGive
- xSemaphoreGiveRecursive
- xSemaphoreGiveFromISR

xSemaphoreCreateBinary

semphr. H

TIP: 'Task Notifications' can provide a light weight alternative to binary semaphores in many situations

SemaphoreHandle_t xSemaphoreCreateBinary(void);

Function that creates a binary semaphore. Binary semaphores are either available, or not available, hence *binary*.
The semaphore is created in the 'empty' state, meaning the semaphore must first be given before it can be taken (obtained) using the xSemaphoreTake() function.
Binary semaphores and mutexes are very similar but have some subtle differences: Mutexes include a priority inheritance mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronisation (between tasks or between tasks and an interrupt), and mutexes the better choice for implementing simple mutual exclusion.
A binary semaphore need not be given back once obtained, so task synchronisation can be implemented by one task/interrupt continuously 'giving' the semaphore while another continuously 'takes' the semaphore. This is demonstrated by the sample code on the xSemaphoreGiveFromISR()documentation page.
The priority of a task that 'takes' a mutex can potentially be raised if another task of higher priority attempts to obtain the same mutex. The task that owns the mutex 'inherits' the priority of the task attempting to 'take' the same mutex. This means the mutex must always be 'given' back - otherwise the higher priority task will never be able to obtain the mutex, and the lower priority task will never 'disinherit' the priority. An example of a mutex being used to implement mutual exclusion is provided on the xSemaphoreTake() documentation page.
Both mutex and binary semaphores are assigned to variables of type SemaphoreHandle_t and can be used in any API function that takes a parameter of this type.

Return values:

NULL	The semaphore could not be created because there was insufficientFreeRTOS heap available.
Any other value	The semaphore was created successfully. The returned value should be stored as the handle to the created semaphore.

Example usage:

```
SemaphoreHandle_t xSemaphore;

void vATask( void * pvParameters )
```

```
{
    /* Attempt to create a semaphore. */
    xSemaphore = xSemaphoreCreateBinary();

    if( xSemaphore == NULL )
    {
        /* There was insufficient OpenRTOS heap available for the semaphore to
        be created successfully. */
    }
    else
    {
        /* The semaphore can now be used. Its handle is stored in the
        xSemahore variable.    Calling xSemaphoreTake() on the semaphore here
        will fail until the semaphore has first been given. */
    }
}
```

vSemaphoreCreateBinary

semphr. H

vSemaphoreCreateBinary(SemaphoreHandle_t xSemaphore)

NOTE: The vSemaphoreCreateBinary() macro remains in the source code to ensure backward compatibility, but it should not be used in new designs. Use the xSemaphoreCreateBinary() function instead.

Macro that creates a semaphore by using the existing queue mechanism. The queue length is 1 as this is a binary semaphore. The data size is 0 as we don't want to actually store any data - we just want to know if the queue is empty or full.
Binary semaphores and mutexes are very similar but have some subtle differences: Mutexes include a priority inheritance mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronisation (between tasks or between tasks and an interrupt), and mutexes the better choice for implementing simple mutual exclusion.
A binary semaphore need not be given back once obtained, so task synchronisation can be implemented by one task/interrupt continuously 'giving' the semaphore while another continuously 'takes' the semaphore. This is demonstrated by the sample code on the xSemaphoreGiveFromISR()documentation page.
The priority of a task that 'takes' a mutex can potentially be raised if another task of higher priority attempts to obtain the same mutex. The task that owns the mutex 'inherits' the priority of the task attempting to 'take' the same mutex. This means the mutex must always be 'given' back - otherwise the higher priority task will never be able to obtain the mutex, and the lower priority task will never 'disinherit' the priority. An example of a mutex being used to implement mutual exclusion is provided on the xSemaphoreTake() documentation page.
Both mutex and binary semaphores are assigned to variables of type SemaphoreHandle_t and can be used in any API function that takes a parameter of this type.

Parameters:

xSemaphore	Handle to the created semaphore. Should be of type SemaphoreHandle_t.

Example usage:

```
SemaphoreHandle_t xSemaphore;
void vATask( void * pvParameters )
{
    // Semaphore cannot be used before a call to vSemaphoreCreateBinary ().
```

```
    // This is a macro so pass the variable in directly.
    vSemaphoreCreateBinary( xSemaphore );
    if( xSemaphore != NULL )
    {
        // The semaphore was created successfully.
        // The semaphore can now be used.
    }
}
```

xSemaphoreCreateCounting

semphr. h

```
SemaphoreHandle_t xSemaphoreCreateCounting
                    (
                        UBaseType_t uxMaxCount,
                        UBaseType_t uxInitialCount
                    )
```

Macro that creates a counting semaphore by using the existing queue mechanism. Counting semaphores are typically used for two things:

1. Counting events.
   In this usage scenario an event handler will 'give' a semaphore each time an event occurs (incrementing the semaphore count value), and a handler task will 'take' a semaphore each time it processes an event (decrementing the semaphore count value). The count value is therefore the difference between the number of events that have occurred and the number that have been processed. In this case it is desirable for the initial count value to be zero.
2. Resource management.
   In this usage scenario the count value indicates the number of resources available. To obtain control of a resource a task must first obtain a semaphore - decrementing the semaphore 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 count value. In this case it is desirable for the initial count value to be equal to the maximum count value, indicating that all resources are free.

Parameters:

uxMaxCount The maximum count value that can be reached. When the semaphore reaches this value it can no longer be 'given'.

uxInitialCount The count value assigned to the semaphore when it is created.

Returns:

Handle to the created semaphore. Of type SemaphoreHandle_t. NULL if the semaphore could not be created.

Example usage:

```
void vATask( void * pvParameters )
{
SemaphoreHandle_t xSemaphore;

    // Semaphore cannot be used before a call to xSemaphoreCreateCounting().
    // The max value to which the semaphore can count shall be 10, and the
    // initial value assigned to the count shall be 0.
```

```
    xSemaphore = xSemaphoreCreateCounting( 10, 0 );

    if( xSemaphore != NULL )
    {
        // The semaphore was created successfully.
        // The semaphore can now be used.
    }
}
```

xSemaphoreCreateMutex

Only available from FreeRTOS V4.5.0 onwards.

semphr. H

SemaphoreHandle_t xSemaphoreCreateMutex(void)

Macro that creates a mutex semaphore by using the existing queue mechanism.
Mutexes created using this macro can be accessed using the xSemaphoreTake() and xSemaphoreGive() macros. The xSemaphoreTakeRecursive() and xSemaphoreGiveRecursive() macros should not be used.
Mutexes and binary semaphores are very similar but have some subtle differences: Mutexes include a priority inheritance mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronisation (between tasks or between tasks and an interrupt), and mutexes the better choice for implementing simple mutual exclusion.
The priority of a task that 'takes' a mutex can potentially be raised if another task of higher priority attempts to obtain the same mutex. The task that owns the mutex 'inherits' the priority of the task attempting to 'take' the same mutex. This means the mutex must always be 'given' back - otherwise the higher priority task will never be able to obtain the mutex, and the lower priority task will never 'disinherit' the priority. An example of a mutex being used to implement mutual exclusion is provided on the xSemaphoreTake() documentation page.
A binary semaphore need not be given back once obtained, so task synchronisation can be implemented by one task/interrupt continuously 'giving' the semaphore while another continuously 'takes' the semaphore. This is demonstrated by the sample code on the xSemaphoreGiveFromISR()documentation page.
Both mutex and binary semaphores are assigned to variables of type SemaphoreHandle_t and can be used in any API function that takes a parameter of this type.

Return:

Handle to the created semaphore. Should be of type SemaphoreHandle_t.

Example usage:

```
SemaphoreHandle_t xSemaphore;
void vATask( void * pvParameters )
{
    // Mutex semaphores cannot be used before a call to
    // xSemaphoreCreateMutex().   The created mutex is returned.
    xSemaphore = xSemaphoreCreateMutex();
```

```
    if( xSemaphore != NULL )
    {
        // The semaphore was created successfully.
        // The semaphore can now be used.
    }
}
```

xSemaphoreCreateRecursiveMutex

semphr. H

SemaphoreHandle_t xSemaphoreCreateRecursiveMutex(void)

Macro that implements a recursive mutex by using the existing queue mechanism.
Mutexes created using this macro can be accessed using the xSemaphoreTakeRecursive() and xSemaphoreGiveRecursive() macros. The xSemaphoreTake() and xSemaphoreGive() macros should not be used.
A mutex used recursively can be 'taken' repeatedly by the owner. The mutex doesn't become available again until the owner has called xSemaphoreGiveRecursive() for each successful 'take' request. For example, if a task successfully 'takes' the same mutex 5 times then the mutex will not be available to any other task until it has also 'given' the mutex back exactly five times.
This type of semaphore uses a priority inheritance mechanism so a task 'taking' a semaphore MUST ALWAYS 'give' the semaphore back once the semaphore it is no longer required.
See vSemaphoreCreateBinary() for an alternative implementation that can be used for pure synchronisation (where one task or interrupt always 'gives' the semaphore and another always 'takes' the semaphore) and from within interrupt service routines.

Returns:

xSemaphore Handle to the created mutex semaphore. Should be of type SemaphoreHandle_t.

Example usage:

```
 SemaphoreHandle_t xMutex;

 void vATask( void * pvParameters )
 {
     // Semaphore cannot be used before a call to xSemaphoreCreateMutex().
     // This is a macro so pass the variable in directly.
     xMutex = xSemaphoreCreateRecursiveMutex();

     if( xMutex != NULL )
     {
         // The mutex type semaphore was created successfully.
         // The mutex can now be used.
     }
 }
```

vSemaphoreDelete

semphr.h

void vSemaphoreDelete(SemaphoreHandle_t xSemaphore);

Deletes a semaphore, including mutex type semaphores and recursive semaphores.
Do not delete a semaphore that has tasks blocked on it (tasks that are in the Blocked state waiting for the semaphore to become available).

Parameters:

xSemaphore The handle of the semaphore being deleted.

xSemaphoreGetMutexHolder

semphr. h

TaskHandle_t xSemaphoreGetMutexHolder(SemaphoreHandle_t xMutex);

Return the handle of the task that holds the mutex specified by the function parameter, if any.
xSemaphoreGetMutexHolder() can be used reliably to determine if the calling task is the mutex holder, but cannot be used reliably if the mutex is held by any task other than the calling task. This is because the mutex holder might change between the calling task calling the function, and the calling task testing the function's return value.
configUSE_MUTEXES must be set to 1 in FreeRTOSConfig.h for xSemaphoreGetMutexHolder() to be available.

Parameters:

xMutex The handle of the mutex being queried.

Returns:

The handle of the task that holds the mutex specified by the xMutex parameter. NULL is returned if the semaphore passed in the xMutex parameter is not a mutex type semaphore, or if the mutex is available and so not held by any task.

xSemaphoreTake

semphr. h

```
xSemaphoreTake(
                    SemaphoreHandle_t xSemaphore,
                    TickType_t xTicksToWait
                )
```

Macro to obtain a semaphore. The semaphore must have previously been created with a call to xSemaphoreCreateBinary(), xSemaphoreCreateMutex() or xSemaphoreCreateCounting().
This macro must not be called from an ISR. xQueueReceiveFromISR() can be used to take a semaphore from within an interrupt if required, although this would not be a normal operation. Semaphores use queues as their underlying mechanism, so functions are to some extent interoperable.

Parameters:

xSemaphore A handle to the semaphore being taken - obtained when the semaphore was created.

xTicksToWait The time in ticks to wait for the semaphore to become available. The macro portTICK_PERIOD_MS can be used to convert this to a real time. A block time of zero can be used to poll the semaphore.
If INCLUDE_vTaskSuspend is set to '1' then specifying the block time as portMAX_DELAY will cause the task to block indefinitely (without a timeout).

Returns:

pdTRUE if the semaphore was obtained. pdFALSE if xTicksToWait expired without the semaphore becoming available.

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;

/* A task that creates a semaphore. */
void vATask( void * pvParameters )
{
    /* Create the semaphore to guard a shared resource.  As we are using
    the semaphore for mutual exclusion we create a mutex semaphore
    rather than a binary semaphore. */
    xSemaphore = xSemaphoreCreateMutex();
}
```

```
/* A task that uses the semaphore. */
void vAnotherTask( void * pvParameters )
{
    /* ... Do other things. */

    if( xSemaphore != NULL )
    {
        /* See if we can obtain the semaphore.  If the semaphore is not
        available wait 10 ticks to see if it becomes free. */
        if( xSemaphoreTake( xSemaphore, ( TickType_t ) 10 ) == pdTRUE )
        {
            /* We were able to obtain the semaphore and can now access the
            shared resource. */

            /* ... */

            /* We have finished accessing the shared resource.  Release the
            semaphore. */
            xSemaphoreGive( xSemaphore );
        }
        else
        {
            /* We could not obtain the semaphore and can therefore not access
            the shared resource safely. */
        }
    }
}
```

xSemaphoreTakeFromISR

semphr. h

```
xSemaphoreTakeFromISR
    (
      SemaphoreHandle_t xSemaphore,
      signed BaseType_t *pxHigherPriorityTaskWoken
    )
```

A version of xSemaphoreTake() that can be called from an ISR. Unlike xSemaphoreTake(), xSemaphoreTakeFromISR() does not permit a block time to be specified.

Parameters:

xSemaphore	The semaphore being 'taken'. A semaphore is referenced by a variable of type SemaphoreHandle_t and must be explicitly created before being used.
pxHigherPriorityTaskWoken	It is possible (although unlikely, and dependent on the semaphore type) that a semaphore will have one or more tasks blocked on it waiting to give the semaphore. Calling xSemaphoreTakeFromISR() will make a task that was blocked waiting to give the semaphore leave the Blocked state. If calling the API function causes a task to leave the Blocked state, and the unblocked task has a priority equal to or higher than the currently executing task (the task that was interrupted), then, internally, the API function will set *pxHigherPriorityTaskWoken to pdTRUE. If xSemaphoreTakeFromISR() sets *pxHigherPriorityTaskWoken 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. The mechanism is identical to that used in the xQueueReceiveFromISR() function, and readers

are referred to
the xQueueReceiveFromISR()documentation for
further explanation.
From FreeRTOS V7.3.0
pxHigherPriorityTaskWoken is an optional
parameter and can be set to NULL.

Returns:

pdTRUE if the semaphore was successfully taken. pdFALSE if the semaphore was not successfully taken because it was not available.

xSemaphoreTakeRecursive

semphr. h

```
xSemaphoreTakeRecursive( SemaphoreHandle_t xMutex,
                         TickType_t xTicksToWait );
```

Macro to recursively obtain, or 'take', a mutex type semaphore. The mutex must have previously been created using a call to xSemaphoreCreateRecursiveMutex(); configUSE_RECURSIVE_MUTEXES must be set to 1 in FreeRTOSConfig.h for this macro to be available.
This macro must not be used on mutexes created using xSemaphoreCreateMutex().
A mutex used recursively can be 'taken' repeatedly by the owner. The mutex doesn't become available again until the owner has called xSemaphoreGiveRecursive() for each successful 'take' request. For example, if a task successfully 'takes' the same mutex 5 times then the mutex will not be available to any other task until it has also 'given' the mutex back exactly five times.

Parameters:

xMutex	A handle to the mutex being obtained. This is the handle returned by xSemaphoreCreateRecursiveMutex().
xTicksToWait	The time in ticks to wait for the semaphore to become available. The macro portTICK_PERIOD_MS can be used to convert this to a real time. A block time of zero can be used to poll the semaphore. If the task already owns the semaphore then xSemaphoreTakeRecursive() will return immediately no matter what the value of xTicksToWait.

Returns:

pdTRUE if the semaphore was obtained. pdFALSE if xTicksToWait expired without the semaphore becoming available.

Example usage:

```
SemaphoreHandle_t xMutex = NULL;

// A task that creates a mutex.
void vATask( void * pvParameters )
{
    // Create the mutex to guard a shared resource.
    xMutex = xSemaphoreCreateRecursiveMutex();
}

// A task that uses the mutex.
```

```
void vAnotherTask( void * pvParameters )
{
    // ... Do other things.

    if( xMutex != NULL )
    {
        // See if we can obtain the mutex.    If the mutex is not available
        // wait 10 ticks to see if it becomes free.
        if( xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 ) == pdTRUE )
        {
            // We were able to obtain the mutex and can now access the
            // shared resource.

            // ...
            // For some reason due to the nature of the code further calls to
            // xSemaphoreTakeRecursive() are made on the same mutex.    In real
            // code these would not be just sequential calls as this would make
            // no sense.    Instead the calls are likely to be buried inside
            // a more complex call structure.
            xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );
            xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );

            // The mutex has now been 'taken' three times, so will not be
            // available to another task until it has also been given back
            // three times.    Again it is unlikely that real code would have
            // these calls sequentially, but instead buried in a more complex
            // call structure.    This is just for illustrative purposes.
            xSemaphoreGiveRecursive( xMutex );
            xSemaphoreGiveRecursive( xMutex );
            xSemaphoreGiveRecursive( xMutex );

            // Now the mutex can be taken by other tasks.
        }
        else
        {
            // We could not obtain the mutex and can therefore not access
            // the shared resource safely.
        }
    }
}
```

xSemaphoreGive

semphr. h

xSemaphoreGive(SemaphoreHandle_t xSemaphore)

Macro to release a semaphore. The semaphore must have previously been created with a call to xSemaphoreCreateBinary(), xSemaphoreCreateMutex() or xSemaphoreCreateCounting(), and obtained using sSemaphoreTake().
This must not be used from an ISR. See xSemaphoreGiveFromISR() for an alternative which can be used from an ISR.
This macro must also not be used on semaphores created using xSemaphoreCreateRecursiveMutex().

Parameters:

xSemaphore A handle to the semaphore being released. This is the handle returned when the semaphore was created.

Returns:

pdTRUE if the semaphore was released. pdFALSE if an error occurred. Semaphores are implemented using queues. An error can occur if there is no space on the queue to post a message - indicating that the semaphore was not first obtained correctly.

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;
void vATask( void * pvParameters )
{
    // Create the semaphore to guard a shared resource.    As we are using
    // the semaphore for mutual exclusion we create a mutex semaphore
    // rather than a binary semaphore.
    xSemaphore = xSemaphoreCreateMutex();

    if( xSemaphore != NULL )
    {
        if( xSemaphoreGive( xSemaphore ) != pdTRUE )
        {
            // We would expect this call to fail because we cannot give
            // a semaphore without first "taking" it!
        }
        // Obtain the semaphore - don't block if the semaphore is not
        // immediately available.
        if( xSemaphoreTake( xSemaphore, ( TickType_t ) 0 ) )
```

```
        {
            // We now have the semaphore and can access the shared resource.
            // ...
            // We have finished accessing the shared resource so can free the
            // semaphore.
            if( xSemaphoreGive( xSemaphore ) != pdTRUE )
            {
                // We would not expect this call to fail because we must have
                // obtained the semaphore to get here.
            }
        }
    }
}
```

xSemaphoreGiveRecursive

semphr. h

xSemaphoreGiveRecursive(SemaphoreHandle_t xMutex)

Macro to recursively release, or 'give', a mutex type semaphore. The mutex must have previously been created using a call to xSemaphoreCreateRecursiveMutex(); configUSE_RECURSIVE_MUTEXES must be set to 1 in FreeRTOSConfig.h for this macro to be available.
This macro must not be used on mutexes created using xSemaphoreCreateMutex().
A mutex used recursively can be 'taken' repeatedly by the owner. The mutex doesn't become available again until the owner has called xSemaphoreGiveRecursive() for each successful 'take' request. For example, if a task successfully 'takes' the same mutex 5 times then the mutex will not be available to any other task until it has also 'given' the mutex back exactly five times.

Parameters:

xMutex A handle to the mutex being released, or 'given'. This is the handle returned by xSemaphoreCreateRecursiveMutex().

Returns:

pdTRUE if the semaphore was successfully given.

Example usage:

```
SemaphoreHandle_t xMutex = NULL;

// A task that creates a mutex.
void vATask( void * pvParameters )
{
    // Create the mutex to guard a shared resource.
    xMutex = xSemaphoreCreateRecursiveMutex();
}

// A task that uses the mutex.
void vAnotherTask( void * pvParameters )
{
    // ... Do other things.

    if( xMutex != NULL )
    {
        // See if we can obtain the mutex.    If the mutex is not available
        // wait 10 ticks to see if it becomes free.
```

```
        if( xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 ) == pdTRUE )
        {
            // We were able to obtain the mutex and can now access the
            // shared resource.

            // ...
            // For some reason due to the nature of the code further calls to
            // xSemaphoreTakeRecursive() are made on the same mutex.  In real
            // code these would not be just sequential calls as this would make
            // no sense.  Instead the calls are likely to be buried inside
            // a more complex call structure.
            xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );
            xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );

            // The mutex has now been 'taken' three times, so will not be
            // available to another task until it has also been given back
            // three times.  Again it is unlikely that real code would have
            // these calls sequentially, it would be more likely that the calls
            // to xSemaphoreGiveRecursive() would be called as a call stack
            // unwound.  This is just for demonstrative purposes.
            xSemaphoreGiveRecursive( xMutex );
            xSemaphoreGiveRecursive( xMutex );
            xSemaphoreGiveRecursive( xMutex );

            // Now the mutex can be taken by other tasks.
        }
        else
        {
            // We could not obtain the mutex and can therefore not access
            // the shared resource safely.
        }
    }
}
```

xSemaphoreGiveFromISR

semphr. h

```
xSemaphoreGiveFromISR
    (
      SemaphoreHandle_t xSemaphore,
      signed BaseType_t *pxHigherPriorityTaskWoken
    )
```

Macro to release a semaphore. The semaphore must have previously been created with a call to xSemaphoreCreateBinary() or xSemaphoreCreateCounting().
Mutex type semaphores (those created using a call to xSemaphoreCreateMutex()) must not be used with this macro.
This macro can be used from an ISR.

Parameters:

xSemaphore	A handle to the semaphore being released. This is the handle returned when the semaphore was created.
pxHigherPriorityTaskWoken	xSemaphoreGiveFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if giving the semaphore caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xSemaphoreGiveFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited. From FreeRTOS V7.3.0 pxHigherPriorityTaskWoken is an optional parameter and can be set to NULL.

Returns:

pdTRUE if the semaphore was successfully given, otherwise errQUEUE_FULL.

Example usage:

```
#define LONG_TIME 0xffff
#define TICKS_TO_WAIT       10

SemaphoreHandle_t xSemaphore = NULL;
```

```
/* Repetitive task. */
void vATask( void * pvParameters )
{
    /* We are using the semaphore for synchronisation so we create a binary
    semaphore rather than a mutex.  We must make sure that the interrupt
    does not attempt to use the semaphore before it is created! */
    xSemaphore = xSemaphoreCreateBinary();

    for( ;; )
    {
        /* We want this task to run every 10 ticks of a timer.  The semaphore
        was created before this task was started.

        Block waiting for the semaphore to become available. */
        if( xSemaphoreTake( xSemaphore, LONG_TIME ) == pdTRUE )
        {
            /* It is time to execute. */

            ...

            /* We have finished our task.  Return to the top of the loop where
            we will block on the semaphore until it is time to execute
            again.  Note when using the semaphore for synchronisation with an
            ISR in this manner there is no need to 'give' the semaphore
            back. */
        }
    }
}

/* Timer ISR */
void vTimerISR( void * pvParameters )
{
static unsigned char ucLocalTickCount = 0;
static signed BaseType_t xHigherPriorityTaskWoken;

    /* A timer tick has occurred. */

    ... Do other time functions.

    /* Is it time for vATask() to run? */
    xHigherPriorityTaskWoken = pdFALSE;
    ucLocalTickCount++;
    if( ucLocalTickCount >= TICKS_TO_WAIT )
    {
```

```
            /* Unblock the task by releasing the semaphore. */
            xSemaphoreGiveFromISR( xSemaphore, &xHigherPriorityTaskWoken );

            /* Reset the count so we release the semaphore again in 10 ticks
            time. */
            ucLocalTickCount = 0;
        }

        /* If xHigherPriorityTaskWoken was set to true you
        we should yield.    The actual macro used here is
        port specific. */
        portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

10 Software Timers

FreeRTOS Software Timer API Functions

- xTimerCreate
- xTimerIsTimerActive
- pvTimerGetTimerID
- pcTimerGetTimerName
- xTimerStart
- xTimerStop
- xTimerChangePeriod
- xTimerDelete
- xTimerReset
- xTimerStartFromISR
- xTimerStopFromISR
- xTimerChangePeriodFromISR
- xTimerResetFromISR
- pvTimerGetTimerID
- vTimerSetTimerID
- xTimerGetTimerDaemonTaskHandle
- xTimerPendFunctionCall
- xTimerPendFunctionCallFromISR

xTimerCreate

timers.h

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

Creates a new software timer instance. This allocates the storage required by the new timer, initialises the new timers internal state, and returns a handle by which the new timer can be referenced.
Timers are created in the dormant state.
The xTimerStart(), xTimerReset(), xTimerStartFromISR(),xTimerResetFromISR(), xTimer ChangePeriod() and xTimerChangePeriodFromISR() API functions can all be used to transition a timer into the active state.

Parameters:

pcTimerName	A text name that is assigned to the timer. This is done purely to assist debugging. The RTOS kernel itself only ever references a timer by its handle, and never by its name.
xTimerPeriod	The timer period. The time is defined in tick periods so the constant portTICK_PERIOD_MS can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then xTimerPeriod should be set to 100. Alternatively, if the timer must expire after 500ms, then xPeriod can be set to (500 / portTICK_PERIOD_MS) provided configTICK_RATE_HZ is less than or equal to 1000.
uxAutoReload	If uxAutoReload is set to pdTRUE, then the timer will expire repeatedly with a frequency set by the xTimerPeriod parameter. If uxAutoReload is set to pdFALSE, then the timer will be a one-shot and enter the dormant state after it expires.
pvTimerID	An identifier that is assigned to the timer being created. Typically this would be used in the timer callback function to identify which timer expired when the same callback

function is assigned to more than one timer, or together with the vTimerSetTimerID() andpvTimerGetTimerID() API functions to save a value between calls to the timer's callback function.

pxCallbackFunction The function to call when the timer expires. Callback functions must have the prototype defined by TimerCallbackFunction_t, which is "void vCallbackFunction(TimerHandle_t xTimer);".

Returns:

If the timer is successfully created then a handle to the newly created timer is returned. If the timer cannot be created (because either there is insufficient FreeRTOS heap remaining to allocate the timer structures, or the timer period was set to 0) then NULL is returned.

Example usage:

```
#define NUM_TIMERS 5

/* An array to hold handles to the created timers. */
TimerHandle_t xTimers[ NUM_TIMERS ];

/* An array to hold a count of the number of times each timer expires. */
long lExpireCounters[ NUM_TIMERS ] = { 0 };

/* Define a callback function that will be used by multiple timer instances.
The callback function does nothing but count the number of times the
associated timer expires, and stop the timer once the timer has expired
10 times. */
void vTimerCallback( TimerHandle_t pxTimer )
{
long lArrayIndex;
const long xMaxExpiryCountBeforeStopping = 10;

    /* Optionally do something if the pxTimer parameter is NULL. */
    configASSERT( pxTimer );

    /* Which timer expired? */
    lArrayIndex = ( long ) pvTimerGetTimerID( pxTimer );

    /* Increment the number of times that pxTimer has expired. */
    lExpireCounters[ lArrayIndex ] += 1;

    /* If the timer has expired 10 times then stop it from running. */
```

```
        if( lExpireCounters[ lArrayIndex ] == xMaxExpiryCountBeforeStopping )
        {
            /* Do not use a block time if calling a timer API function from a
            timer callback function, as doing so could cause a deadlock! */
            xTimerStop( pxTimer, 0 );
        }
}

void main( void )
{
long x;

        /* Create then start some timers.  Starting the timers before the RTOS
        scheduler has been started means the timers will start running
        immediately that the RTOS scheduler starts. */
        for( x = 0; x < NUM_TIMERS; x++ )
        {
            xTimers[ x ] = xTimerCreate
                        (   /* Just a text name, not used by the RTOS kernel. */
                            "Timer",
                            /* The timer period in ticks, must be greater than 0. */
                            ( 100 * x ) + 100,
                            /* The timers will auto-reload themselves when they
                            expire. */
                            pdTRUE,
                            /* Assign each timer a unique id equal to its array
                            index. */
                            ( void * ) x,
                            /* Each timer calls the same callback when it expires. */
                            vTimerCallback
                        );

            if( xTimers[ x ] == NULL )
            {
                /* The timer was not created. */
            }
            else
            {
                /* Start the timer.  No block time is specified, and even if one
                was it would be ignored because the RTOS scheduler has not yet
                been started. */
                if( xTimerStart( xTimers[ x ], 0 ) != pdPASS )
                {
                    /* The timer could not be set into the Active state. */
```

```
            }
        }
    }

    /* ...
    Create tasks here.
    ... */

    /* Starting the RTOS scheduler will start the timers running as they have
    already been set into the active state. */
    vTaskStartScheduler();

    /* Should not reach here. */
    for( ;; );
}
```

xTimerIsTimerActive

timers.h

```
BaseType_t xTimerIsTimerActive( TimerHandle_t xTimer );
```

Queries a timer to see if it is active or dormant.
A timer will be dormant if:

1. It has been created but not started, or
2. It is an expired one-shot timer that has not been restarted.

Timers are created in the dormant state.
The xTimerStart(), xTimerReset(), xTimerStartFromISR(),xTimerResetFromISR(), xTimerChangePeriod() and xTimerChangePeriodFromISR() API functions can all be used to transition a timer into the active state.

Parameters:

xTimer The timer being queried.

Returns:

pdFALSE will be returned if the timer is dormant. A value other than pdFALSE will be returned if the timer is active.

Example usage:

```
/* This function assumes xTimer has already
been created. */
void vAFunction( TimerHandle_t xTimer )
{
    /* or more simply and equivalently
    "if( xTimerIsTimerActive( xTimer ) )" */
    if( xTimerIsTimerActive( xTimer ) != pdFALSE )
    {
        /* xTimer is active, do something. */
    }
    else
    {
        /* xTimer is not active, do something else. */
    }
}
```

xTimerStart

timers.h

```
BaseType_t xTimerStart( TimerHandle_t xTimer,
                        TickType_t xBlockTime );
```

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the RTOS kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.
xTimerStart() starts a timer that was previously created using the xTimerCreate() API function. If the timer had already been started and was already in the active state, then xTimerStart() has equivalent functionality to the xTimerReset() API function.
Starting a timer ensures the timer is in the active state. If the timer is not stopped, deleted, or reset in the mean time, the callback function associated with the timer will get called 'n 'ticks after xTimerStart() was called, where 'n' is the timers defined period.
It is valid to call xTimerStart() before the RTOS scheduler has been started, but when this is done the timer will not actually start until the RTOS scheduler is started, and the timers expiry time will be relative to when the RTOS scheduler is started, not relative to when xTimerStart() was called.
The configUSE_TIMERS configuration constant must be set to 1 for xTimerStart() to be available.

Parameters:

xTimer	The handle of the timer being started/restarted.
xBlockTime	Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the start command to be successfully sent to the timer command queue, should the queue already be full when xTimerStart() was called. xBlockTime is ignored if xTimerStart() is called before the RTOS scheduler is started.

Returns:

pdFAIL will be returned if the start command could not be sent to the timer command queue even after xBlockTime ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when xTimerStart() is actually called. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Example usage:

See the example on the xTimerCreate() documentation page.

xTimerStop

timers.h

```
BaseType_t xTimerStop( TimerHandle_t xTimer,
                       TickType_t xBlockTime );
```

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the RTOS kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.
xTimerStop() stops a timer that was previously started using either of the xTimerStart(),xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod() andxTimerChangePeriodFromISR() API functions.
Stopping a timer ensures the timer is not in the active state.
The configUSE_TIMERS configuration constant must be set to 1 for xTimerStop() to be available.

Parameters:

xTimer	The handle of the timer being stopped.
xBlockTime	Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the stop command to be successfully sent to the timer command queue, should the queue already be full when xTimerStop() was called. xBlockTime is ignored if xTimerStop() is called before the RTOS scheduler is started.

Returns:

pdFAIL will be returned if the stop command could not be sent to the timer command queue even after xBlockTime ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Example usage:
See the example on the xTimerCreate() documentation page.

xTimerChangePeriod

timers.h

```
BaseType_t xTimerChangePeriod( TimerHandle_t xTimer,
                               TickType_t xNewPeriod,
                               TickType_t xBlockTime );
```

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the RTOS kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.
xTimerChangePeriod() changes the period of a timer that was previously created using thexTimerCreate() API function.
xTimerChangePeriod() can be called to change the period of an active or dormant state timer.
The configUSE_TIMERS configuration constant must be set to 1 for xTimerChangePeriod() to be available.

Parameters:

xTimer The handle of the timer that is having its period changed.

xNewPeriod The new period for xTimer. Timer periods are specified in tick periods, so the constant portTICK_PERIOD_MS can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then xNewPeriod should be set to 100. Alternatively, if the timer must expire after 500ms, then xNewPeriod can be set to (500 / portTICK_PERIOD_MS) provided configTICK_RATE_HZ is less than or equal to 1000.

xBlockTime Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the change period command to be successfully sent to the timer command queue, should the queue already be full when xTimerChangePeriod() was called. xBlockTime is ignored if xTimerChangePeriod() is called before the RTOS scheduler is started.

Returns:

pdFAIL will be returned if the change period command could not be sent to the timer command queue even after xBlockTime ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer

service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Example usage:

```
/* This function assumes xTimer has already been created.  If the timer
referenced by xTimer is already active when it is called, then the timer
is deleted.  If the timer referenced by xTimer is not active when it is
called, then the period of the timer is set to 500ms and the timer is
started. */
void vAFunction( TimerHandle_t xTimer )
{
    /* or more simply and equivalently
    "if( xTimerIsTimerActive( xTimer ) )" */
    if( xTimerIsTimerActive( xTimer ) != pdFALSE )
    {
        /* xTimer is already active - delete it. */
        xTimerDelete( xTimer );
    }
    else
    {
        /* xTimer is not active, change its period to 500ms.  This will also
        cause the timer to start.  Block for a maximum of 100 ticks if the
        change period command cannot immediately be sent to the timer
        command queue. */
        if( xTimerChangePeriod( xTimer, 500 / portTICK_PERIOD_MS, 100 )
                                                                == pdPASS )
        {
            /* The command was successfully sent. */
        }
        else
        {
            /* The command could not be sent, even after waiting for 100 ticks
            to pass.  Take appropriate action here. */
        }
    }
}
```

xTimerDelete

timers.h

```
BaseType_t xTimerDelete( TimerHandle_t xTimer,
                         TickType_t xBlockTime );
```

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the RTOS kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.
xTimerDelete() deletes a timer that was previously created using the xTimerCreate() API function.
The configUSE_TIMERS configuration constant must be set to 1 for xTimerDelete() to be available.

Parameters:

xTimer The handle of the timer being deleted.

xBlockTime Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the delete command to be successfully sent to the timer command queue, should the queue already be full when xTimerDelete() was called. xBlockTime is ignored if xTimerDelete() is called before the RTOS scheduler is started.

Returns:

pdFAIL will be returned if the delete command could not be sent to the timer command queue even after xBlockTime ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Example usage:

See the example on the xTimerChangePeriod() documentation page.

xTimerReset

timers.h

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

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the RTOS kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.
xTimerReset() re-starts a timer that was previously created using the xTimerCreate() API function. If the timer had already been started and was already in the active state, then xTimerReset() will cause the timer to re-evaluate its expiry time so that it is relative to when xTimerReset() was called. If the timer was in the dormant state then xTimerReset() has equivalent functionality to thexTimerStart() API function.
Resetting a timer ensures the timer is in the active state. If the timer is not stopped, deleted, or reset in the mean time, the callback function associated with the timer will get called 'n' ticks after xTimerReset() was called, where 'n' is the timers defined period.
It is valid to call xTimerReset() before the RTOS scheduler has been started, but when this is done the timer will not actually start until the RTOS scheduler is started, and the timers expiry time will be relative to when the RTOS scheduler is started, not relative to when xTimerReset() was called.
The configUSE_TIMERS configuration constant must be set to 1 for xTimerReset() to be available.

Parameters:

xTimer	The handle of the timer being reset/started/restarted.
xBlockTime	Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the reset command to be successfully sent to the timer command queue, should the queue already be full when xTimerReset() was called. xBlockTime is ignored if xTimerReset() is called before the RTOS scheduler is started.

Returns:

pdFAIL will be returned if the reset command could not be sent to the timer command queue even after xBlockTime ticks had passed. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when xTimerReset() is actually called. The timer

service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Example usage:

```
/* When a key is pressed, an LCD back-light is switched on.  If 5 seconds pass
without a key being pressed, then the LCD back-light is switched off.  In
this case, the timer is a one-shot timer. */

TimerHandle_t xBacklightTimer = NULL;

/* The callback function assigned to the one-shot timer.  In this case the
parameter is not used. */
void vBacklightTimerCallback( TimerHandle_t pxTimer )
{
    /* The timer expired, therefore 5 seconds must have passed since a key
    was pressed.  Switch off the LCD back-light. */
    vSetBacklightState( BACKLIGHT_OFF );
}

/* The key press event handler. */
void vKeyPressEventHandler( char cKey )
{
    /* Ensure the LCD back-light is on, then reset the timer that is
    responsible for turning the back-light off after 5 seconds of
    key inactivity.  Wait 10 ticks for the command to be successfully sent
    if it cannot be sent immediately. */
    vSetBacklightState( BACKLIGHT_ON );
    if( xTimerReset( xBacklightTimer, 10 ) != pdPASS )
    {
        /* The reset command was not executed successfully.  Take appropriate
        action here. */
    }

    /* Perform the rest of the key processing here. */
}

void main( void )
{
long x;

    /* Create then start the one-shot timer that is responsible for turning
    the back-light off if no keys are pressed within a 5 second period. */
    xBacklightTimer = xTimerCreate
                (
```

```
                        /* Just a text name, not used by the RTOS kernel. */
                        "BacklightTimer",
                        /* The timer period in ticks. */
                        ( 5000 / portTICK_PERIOD_MS),
                        /* The timer is a one-shot timer. */
                        pdFALSE,
                        /* The id is not used by the callback so can take any
                        value. */
                        0,
                        /* The callback function that switches the LCD back-light
                        off. */
                        vBacklightTimerCallback
                );

    if( xBacklightTimer == NULL )
    {
        /* The timer was not created. */
    }
    else
    {
        /* Start the timer.    No block time is specified, and even if one was
        it would be ignored because the RTOS scheduler has not yet been
        started. */
        if( xTimerStart( xBacklightTimer, 0 ) != pdPASS )
        {
            /* The timer could not be set into the Active state. */
        }
    }

    /* ...
      Create tasks here.
    ... */

    /* Starting the RTOS scheduler will start the timer running as it has
    already been set into the active state. */
    vTaskStartScheduler();

    /* Should not reach here. */
    for( ;; );
}
```

xTimerStartFromISR

timers.h

```
BaseType_t xTimerStartFromISR
                    (
                        TimerHandle_t xTimer,
                        BaseType_t *pxHigherPriorityTaskWoken
                    );
```

A version of xTimerStart() that can be called from an interrupt service routine.

Parameters:

xTimer	The handle of the timer being started/restarted.
pxHigherPriorityTaskWoken	The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling xTimerStartFromISR() writes a message to the timer command queue, so has the potential to transition the timer service/daemon task out of the Blocked state. If calling xTimerStartFromISR() causes the timer service/daemon task to leave the Blocked state, and the timer service/ daemon task has a priority equal to or greater than the currently executing task (the task that was interrupted), then *pxHigherPriorityTaskWoken will get set to pdTRUE internally within the xTimerStartFromISR() function. If xTimerStartFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt exits.

Returns:

pdFAIL will be returned if the start command could not be sent to the timer command queue. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when xTimerStartFromISR() is actually called. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Example usage:

```
/* This scenario assumes xBacklightTimer has already been created.  When a
key is pressed, an LCD back-light is switched on.  If 5 seconds pass
without a key being pressed, then the LCD back-light is switched off.  In
this case, the timer is a one-shot timer, and unlike the example given for
the xTimerReset() function, the key press event handler is an interrupt
service routine. */

/* The callback function assigned to the one-shot timer.  In this case the
parameter is not used. */
void vBacklightTimerCallback( TimerHandle_t pxTimer )
{
    /* The timer expired, therefore 5 seconds must have passed since a key
    was pressed.  Switch off the LCD back-light. */
    vSetBacklightState( BACKLIGHT_OFF );
}

/* The key press interrupt service routine. */
void vKeyPressEventInterruptHandler( void )
{
BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Ensure the LCD back-light is on, then restart the timer that is
    responsible for turning the back-light off after 5 seconds of
    key inactivity.  This is an interrupt service routine so can only
    call FreeRTOS API functions that end in "FromISR". */
    vSetBacklightState( BACKLIGHT_ON );

    /* xTimerStartFromISR() or xTimerResetFromISR() could be called here
    as both cause the timer to re-calculate its expiry time.
    xHigherPriorityTaskWoken was initialised to pdFALSE when it was
    declared (in this function). */
    if( xTimerStartFromISR( xBacklightTimer,
        &xHigherPriorityTaskWoken ) != pdPASS )
    {
        /* The start command was not executed successfully.  Take appropriate
        action here. */
    }

    /* Perform the rest of the key processing here. */

    /* If xHigherPriorityTaskWoken equals pdTRUE, then a context switch
    should be performed.  The syntax required to perform a context switch
    from inside an ISR varies from port to port, and from compiler to
```

```
        compiler.    Inspect the demos for the port you are using to find the
        actual syntax required. */
        if( xHigherPriorityTaskWoken != pdFALSE )
        {
            /* Call the interrupt safe yield function here (actual function
            depends on the FreeRTOS port being used). */
        }
}
```

xTimerStopFromISR

timers.h

```
BaseType_t xTimerStopFromISR
                (
                    TimerHandle_t xTimer,
                    BaseType_t *pxHigherPriorityTaskWoken
                );
```

A version of xTimerStop() that can be called from an interrupt service routine.

Parameters:

xTimer	The handle of the timer being stopped.
pxHigherPriorityTaskWoken	The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling xTimerStopFromISR() writes a message to the timer command queue, so has the potential to transition the timer service/daemon task out of the Blocked state. If calling xTimerStopFromISR() causes the timer service/daemon task to leave the Blocked state, and the timer service/ daemon task has a priority equal to or greater than the currently executing task (the task that was interrupted), then *pxHigherPriorityTaskWoken will get set to pdTRUE internally within the xTimerStopFromISR() function. If xTimerStopFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt exits.

Returns:

pdFAIL will be returned if the stop command could not be sent to the timer command queue. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Example usage:

```
/* This scenario assumes xTimer has already been created and started.  When
an interrupt occurs, the timer should be simply stopped. */

/* The interrupt service routine that stops the timer. */
void vAnExampleInterruptServiceRoutine( void )
{
BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* The interrupt has occurred - simply stop the timer.
    xHigherPriorityTaskWoken was set to pdFALSE where it was defined
    (within this function).  As this is an interrupt service routine, only
    FreeRTOS API functions that end in "FromISR" can be used. */
    if( xTimerStopFromISR( xTimer, &xHigherPriorityTaskWoken ) != pdPASS )
    {
        /* The stop command was not executed successfully.  Take appropriate
        action here. */
    }

    /* If xHigherPriorityTaskWoken equals pdTRUE, then a context switch
    should be performed.  The syntax required to perform a context switch
    from inside an ISR varies from port to port, and from compiler to
    compiler.  Inspect the demos for the port you are using to find the
    actual syntax required. */
    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        /* Call the interrupt safe yield function here (actual function
        depends on the FreeRTOS port being used). */
    }
}
```

xTimerChangePeriodFromISR

timers.h

```
BaseType_t xTimerChangePeriodFromISR
        (
            TimerHandle_t xTimer,
            TickType_t xNewPeriod,
            BaseType_t *pxHigherPriorityTaskWoken
        );
```

A version of xTimerChangePeriod() that can be called from an interrupt service routine.

Parameters:

xTimer	The handle of the timer that is having its period changed.
xNewPeriod	The new period for xTimer. Timer periods are specified in tick periods, so the constant portTICK_PERIOD_MS can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then xNewPeriod should be set to 100. Alternatively, if the timer must expire after 500ms, then xNewPeriod can be set to (500 / portTICK_PERIOD_MS) provided configTICK_RATE_HZ is less than or equal to 1000.
pxHigherPriorityTaskWoken	The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling xTimerChangePeriodFromISR() writes a message to the timer command queue, so has the potential to transition the timer service/ daemon task out of the Blocked state. If calling xTimerChangePeriodFromISR() causes the timer service/daemon task to leave the Blocked state, and the timer service/daemon task has a priority equal to or greater than the currently executing task

(the task that was interrupted), then *pxHigherPriorityTaskWoken will get set to pdTRUE internally within the xTimerChangePeriodFromISR() function. If xTimerChangePeriodFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt exits.

Returns:

pdFAIL will be returned if the command to change the timers period could not be sent to the timer command queue. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Example usage:

```
/* This scenario assumes xTimer has already been created and started.  When
an interrupt occurs, the period of xTimer should be changed to 500ms. */

/* The interrupt service routine that changes the period of xTimer. */
void vAnExampleInterruptServiceRoutine( void )
{
BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* The interrupt has occurred - change the period of xTimer to 500ms.
    xHigherPriorityTaskWoken was set to pdFALSE where it was defined
    (within this function).  As this is an interrupt service routine, only
    FreeRTOS API functions that end in "FromISR" can be used. */
    if( xTimerChangePeriodFromISR( xTimer,
                                   &xHigherPriorityTaskWoken ) != pdPASS )
    {
        /* The command to change the timers period was not executed
        successfully.  Take appropriate action here. */
    }

    /* If xHigherPriorityTaskWoken equals pdTRUE, then a context switch
    should be performed.  The syntax required to perform a context switch
    from inside an ISR varies from port to port, and from compiler to
    compiler.  Inspect the demos for the port you are using to find the
    actual syntax required. */
    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        /* Call the interrupt safe yield function here (actual function
```

```
            depends on the FreeRTOS port being used). */
        }
    }
```

xTimerResetFromISR

timers.h

```
BaseType_t xTimerResetFromISR
                    (
                        TimerHandle_t xTimer,
                        BaseType_t *pxHigherPriorityTaskWoken
                    );
```

A version of xTimerReset() that can be called from an interrupt service routine.

Parameters:

xTimer	The handle of the timer that is to be started, reset, or restarted.
pxHigherPriorityTaskWoken	The timer service/daemon task spends most of its time in the Blocked state, waiting for messages to arrive on the timer command queue. Calling xTimerResetFromISR() writes a message to the timer command queue, so has the potential to transition the timer service/daemon task out of the Blocked state. If calling xTimerResetFromISR() causes the timer service/daemon task to leave the Blocked state, and the timer service/ daemon task has a priority equal to or greater than the currently executing task (the task that was interrupted), then *pxHigherPriorityTaskWoken will get set to pdTRUE internally within the xTimerResetFromISR() function. If xTimerResetFromISR() sets this value to pdTRUE then a context switch should be performed before the interrupt exits.

Returns:

pdFAIL will be returned if the reset command could not be sent to the timer command queue. pdPASS will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when xTimerResetFromISR() is actually called. The timer service/daemon task priority is set by the configTIMER_TASK_PRIORITY configuration constant.

Example usage:

```
/* This scenario assumes xBacklightTimer has already been created.  When a
key is pressed, an LCD back-light is switched on.  If 5 seconds pass
without a key being pressed, then the LCD back-light is switched off.  In
this case, the timer is a one-shot timer, and unlike the example given for
the xTimerReset() function, the key press event handler is an interrupt
service routine. */

/* The callback function assigned to the one-shot timer.  In this case the
parameter is not used. */
void vBacklightTimerCallback( TimerHandle_t pxTimer )
{
    /* The timer expired, therefore 5 seconds must have passed since a key
    was pressed.  Switch off the LCD back-light. */
    vSetBacklightState( BACKLIGHT_OFF );
}

/* The key press interrupt service routine. */
void vKeyPressEventInterruptHandler( void )
{
BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Ensure the LCD back-light is on, then reset the timer that is
    responsible for turning the back-light off after 5 seconds of
    key inactivity.  This is an interrupt service routine so can only
    call FreeRTOS API functions that end in "FromISR". */
    vSetBacklightState( BACKLIGHT_ON );

    /* xTimerStartFromISR() or xTimerResetFromISR() could be called here
    as both cause the timer to re-calculate its expiry time.
    xHigherPriorityTaskWoken was initialised to pdFALSE when it was
    declared (in this function). */
    if( xTimerResetFromISR( xBacklightTimer,
                            &xHigherPriorityTaskWoken ) != pdPASS )
    {
        /* The reset command was not executed successfully.  Take appropriate
        action here. */
    }

    /* Perform the rest of the key processing here. */

    /* If xHigherPriorityTaskWoken equals pdTRUE, then a context switch
    should be performed.  The syntax required to perform a context switch
```

```
        from inside an ISR varies from port to port, and from compiler to
        compiler.    Inspect the demos for the port you are using to find the
        actual syntax required. */
        if( xHigherPriorityTaskWoken != pdFALSE )
        {
            /* Call the interrupt safe yield function here (actual function
            depends on the FreeRTOS port being used). */
        }
}
```

pvTimerGetTimerID

timers.h

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

Returns the ID assigned to the timer.
IDs are assigned to timers using the pvTimerID parameter of the call to xTimerCreate() that was used to create the timer.
An identifier (ID) is assigned to a timer when the timer is created, and can be changed at any time using the vTimerSetTimerID() API function.
If the same callback function is assigned to multiple timers, the timer identifier can be inspected inside the callback function to determine which timer actually expired.
The timer identifier can also be used to store data in the timer between calls to the timerâ€™s callback function.

Parameters:

xTimer The timer being queried.

Returns:

The ID assigned to the timer being queried.

Example usage:
See the examples provided on the xTimerCreate() documentation page and thevTimerSetTimerID() documentation page.

vTimerSetTimerID

timers.h

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

An identifier (ID) is assigned to a timer when the timer is created, and can be changed at any time using the vTimerSetTimerID() API function.
If the same callback function is assigned to multiple timers, the timer identifier can be inspected inside the callback function to determine which timer actually expired.
The timer identifier can also be used to store data in the timer between calls to the timerâ€™s callback function.

Parameters:

xTimer	The timer being updated.
pvNewID	The handle to which the timer's identifier will be set.

Example usage:

```
/* A callback function assigned to a timer. */
void TimerCallbackFunction( TimerHandle_t pxExpiredTimer )
{
uint32_t ulCallCount;

    /* A count of the number of times this timer has expired
    and executed its callback function is stored in the
    timer's ID.   Retrieve the count, increment it, then save
    it back into the timer's ID. */
    ulCallCount =
          ( uint32_t ) pvTimerGetTimerID( pxExpiredTimer );
    ulCallCount++;
    vTimerSetTimerID( pxExpiredTimer, ( void * ) ulCallCount );
}
```

xTimerGetTimerDaemonTaskHandle

timers.h

TaskHandle_t xTimerGetTimerDaemonTaskHandle(void);

INCLUDE_xTimerGetTimerDaemonTaskHandle and configUSE_TIMERS must both be set to 1 in FreeRTOSConfig.h for xTimerGetTimerDaemonTaskHandle() to be available.

Returns:

Returns the task handle associated with the software timer daemon (or service) task. If configUSE_TIMERS is set to 1 in FreeRTOSConfig.h, then the timer daemon task is created automatically when the RTOS scheduler is started.

xTimerPendFunctionCall()

v8.0.0

timers.h

```
BaseType_t xTimerPendFunctionCall(
                                  PendedFunction_t xFunctionToPend,
                                  void *pvParameter1,
                                  uint32_t ulParameter2,
                                  TickType_t xTicksToWait );
```

Used to pend the execution of a function to the RTOS daemon task (the timer service task, hence this function is pre-fixed with 'Timer').
Functions that can be deferred to the RTOS daemon task must have the following prototype:
void vPendableFunction(void * pvParameter1, uint32_t ulParameter2);
The pvParameter1 and ulParameter2 are provided for use by the application code.
INCLUDE_xTimerPendFunctionCall() and configUSE_TIMERS must both be set to 1 for xTimerPendFunctionCall() to be available.

Parameters:

xFunctionToPend	The function to execute from the timer service/ daemon task. The function must conform to the PendedFunction_t prototype as shown above.
pvParameter1	The value of the callback function's first parameter. The parameter has a void * type to allow it to be used to pass any type. For example, integer types can be cast to a void *, or the void * can be used to point to a structure.
ulParameter2	The value of the callback function's second parameter.
xTicksToWait	Calling this function will result in a message being sent to the timer daemon task on a queue. xTicksToWait is the amount of time the calling task should remain in the Blocked state (so not using any processing time) for space to become available on the timer queue if the queue is found to be full. The length of the queue is set by the value of configTIMER_QUEUE_LENGTH in FreeRTOSConfig.h.

Returns:

pdPASS is returned if the message was successfully sent to the RTOS timer daemon task, otherwise pdFALSE is returned.

xTimerPendFunctionCallFromISR()

v8.0.0
timers.h

```
BaseType_t xTimerPendFunctionCallFromISR(
                                PendedFunction_t xFunctionToPend,
                                void *pvParameter1,
                                uint32_t ulParameter2,
                                BaseType_t *pxHigherPriorityTaskWoken );
```

Used from application interrupt service routines to defer the execution of a function to the RTOS daemon task (the timer service task, hence this function is implemented in timers.c and is prefixed with 'Timer').
Ideally an interrupt service routine (ISR) is kept as short as possible, but sometimes an ISR either has a lot of processing to do, or needs to perform processing that is not deterministic. In these cases xTimerPendFunctionCallFromISR() can be used to defer processing of a function to the RTOS daemon task.
A mechanism is provided that allows the interrupt to return directly to the task that will subsequently execute the pended function. This allows the callback function to execute contiguously in time with the interrupt - just as if the callback had executed in the interrupt itself.
Functions that can be deferred to the RTOS daemon task must have the following prototype:
void vPendableFunction(void * pvParameter1, uint32_t ulParameter2);
The pvParameter1 and ulParameter2 are provided for use by the application code.
INCLUDE_xTimerPendFunctionCall() and configUSE_TIMERS must both be set to 1 for xTimerPendFunctionCallFromISR() to be available.

Parameters:

xFunctionToPend	The function to execute from the timer service/ daemon task. The function must conform to the PendedFunction_t prototype as shown above.
pvParameter1	The value of the callback function's first parameter. The parameter has a void * type to allow it to be used to pass any type. For example, integer types can be cast to a void *, or the void * can be used to point to a structure.
ulParameter2	The value of the callback function's second parameter.

pxHigherPriorityTaskWoken As mentioned above, calling xTimerPendFunctionCallFromISR() will result in a message being sent to the RTOS timer daemon task. If the priority of the daemon task (which is set usingconfigTIMER_TASK_PRIORITY in FreeRTOSConfig.h) is higher than the priority of the currently running task (the task the interrupt interrupted) then *pxHigherPriorityTaskWoken will be set to pdTRUE within xTimerPendFunctionCallFromISR(), indicating that a context switch should be requested before the interrupt exits. For that reason *pxHigherPriorityTaskWoken must be initialised to pdFALSE. See the example code below.

Returns:

pdPASS is returned if the message was successfully sent to the RTOS timer daemon task, otherwise pdFALSE is returned.

Example usage:

```
/* The callback function that will execute in the context of the daemon task.
Note callback functions must all use this same prototype. */
void vProcessInterface( void *pvParameter1, uint32_t ulParameter2 )
{
BaseType_t xInterfaceToService;

    /* The interface that requires servicing is passed in the second
    parameter.    The first parameter is not used in this case. */
    xInterfaceToService = ( BaseType_t ) ulParameter2;

    /* ...Perform the processing here... */
}

/* An ISR that receives data packets from multiple interfaces */
void vAnISR( void )
{
BaseType_t xInterfaceToService, xHigherPriorityTaskWoken;

    /* Query the hardware to determine which interface needs processing. */
    xInterfaceToService = prvCheckInterfaces();
```

```
        /* The actual processing is to be deferred to a task.  Request the
        vProcessInterface() callback function is executed, passing in the
        number of the interface that needs processing.  The interface to
        service is passed in the second parameter.  The first parameter is
        not used in this case. */
        xHigherPriorityTaskWoken = pdFALSE;
        xTimerPendFunctionCallFromISR( vProcessInterface,
                                       NULL,
                                       ( uint32_t ) xInterfaceToService,
                                       &xHigherPriorityTaskWoken );

        /* If xHigherPriorityTaskWoken is now set to pdTRUE then a context
        switch should be requested.  The macro used is port specific and will
        be either portYIELD_FROM_ISR() or portEND_SWITCHING_ISR() - refer to
        the documentation page for the port being used. */
        portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

11 Event Groups

Event Groups and Event Bits API Functions

- xEventGroupCreate
- xEventGroupWaitBits
- xEventGroupSetBits
- xEventGroupSetBitsFromISR
- xEventGroupClearBits
- xEventGroupClearBitsFromISR
- xEventGroupGetBits
- xEventGroupGetBitsFromISR
- xEventGroupSync
- vEventGroupDelete

xEventGroupCreate()

v8.0.0

event_groups.h

```
EventGroupHandle_t xEventGroupCreate( void );
```

Create a new RTOS event group. This function cannot be called from an interrupt.
Event groups are stored in variables of type EventGroupHandle_t. The number of bits (or flags) implemented within an event group is 8 if configUSE_16_BIT_TICKS is set to 1, or 24 if configUSE_16_BIT_TICKS is set to 0. The dependency on configUSE_16_BIT_TICKS results from the data type used for thread local storage in the internal implementation of RTOS tasks.
The RTOS source file FreeRTOS/source/event_groups.c must be included in the build for the xEventGroupCreate() function to be available.

Parameters:

None

Returns:

If the event group was created then a handle to the event group is returned. If there was insufficient FreeRTOS heap available to create the event group then NULL is returned.

Example usage:

```
/* Declare a variable to hold the created event group. */
EventGroupHandle_t xCreatedEventGroup;

/* Attempt to create the event group. */
xCreatedEventGroup = xEventGroupCreate();

/* Was the event group created successfully? */
if( xCreatedEventGroup == NULL )
{
    /* The event group was not created because there was insufficient
    FreeRTOS heap available. */
}
else
{
    /* The event group was created. */
}
```

xEventGroupDelete()

v8.0.0
event_groups.h

```
void vEventGroupDelete( EventGroupHandle_t xEventGroup );
```

Delete an event group that was previously created using a call to xEventGroupCreate(). Tasks that are blocked on the event group being deleted will be unblocked, and report an event group value of 0.
The RTOS source file FreeRTOS/source/event_groups.c must be included in the build for the vEventGroupDelete() function to be available.
This function cannot be called from an interrupt.

Parameters:

xEventGroup The event group being deleted.

Returns:

None.

xEventGroupWaitBits()

v8.0.0
event_groups.h

```
EventBits_t xEventGroupWaitBits(
                        const EventGroupHandle_t xEventGroup,
                        const EventBits_t uxBitsToWaitFor,
                        const BaseType_t xClearOnExit,
                        const BaseType_t xWaitForAllBits,
                        TickType_t xTicksToWait );
```

Read bits within an RTOS event group, optionally entering the Blocked state (with a timeout) to wait for a bit or group of bits to become set.
This function cannot be called from an interrupt.
The RTOS source file FreeRTOS/source/event_groups.c must be included in the build for the xEventGroupWaitBits() function to be available.

Parameters:

xEventGroup	The event group in which the bits are being tested. The event group must have previously been created using a call to xEventGroupCreate().
uxBitsToWaitFor	A bitwise value that indicates the bit or bits to test inside the event group. For example, to wait for bit 0 and/or bit 2 set uxBitsToWaitFor to 0x05. To wait for bits 0 and/or bit 1 and/or bit 2 set uxBitsToWaitFor to 0x07. Etc. uxBitsToWaitFor must **not** be set to 0.
xClearOnExit	If xClearOnExit is set to pdTRUE then any bits set in the value passed as the uxBitsToWaitFor parameter will be cleared in the event group before xEventGroupWaitBits() returns if xEventGroupWaitBits() returns for any reason other than a timeout. The timeout value is set by the xTicksToWait parameter. If xClearOnExit is set to pdFALSE then the bits set in the event group are not altered when the call to xEventGroupWaitBits() returns.
xWaitForAllBits	xWaitForAllBits is used to create either a logical AND test (where all bits must be set) or a logical OR test (where one or more bits must be set) as follows:

If xWaitForAllBits is set to pdTRUE then xEventGroupWaitBits() will return when either **all** the bits set in the value passed as the uxBitsToWaitFor parameter are set in the event group or the specified block time expires.
If xWaitForAllBits is set to pdFALSE then xEventGroupWaitBits() will return when **any** of the bits set in the value passed as the uxBitsToWaitFor parameter are set in the event group or the specified block time expires.

xTicksToWait The maximum amount of time (specified in 'ticks') to wait for one/all (depending on the xWaitForAllBits value) of the bits specified by uxBitsToWaitFor to become set.

Returns:

The value of the event group at the time either the event bits being waited for became set, or the block time expired. The current value of the event bits in an event group will be different to the returned value if a higher priority task or interrupt changed the value of an event bit between the calling task leaving the Blocked state and exiting the xEventGroupWaitBits() function.
Test the return value to know which bits were set. If xEventGroupWaitBits() returned because its timeout expired then not all the bits being waited for will be set. If xEventGroupWaitBits() returned because the bits it was waiting for were set then the returned value is the event group value before any bits were automatically cleared because the xClearOnExit parameter was set to pdTRUE.

Example usage:

```
#define BIT_0	( 1 << 0 )
#define BIT_4	( 1 << 4 )

void aFunction( EventGroupHandle_t xEventGroup )
{
EventBits_t uxBits;
const TickType_t xTicksToWait = 100 / portTICK_PERIOD_MS;

  /* Wait a maximum of 100ms for either bit 0 or bit 4 to be set within
  the event group.   Clear the bits before exiting. */
  uxBits = xEventGroupWaitBits(
               xEventGroup,     /* The event group being tested. */
               BIT_0 | BIT_4, /* The bits within the event group to wait for. */
```

```
                pdTRUE,             /* BIT_0 & BIT_4 should be cleared before returning. */
                pdFALSE,            /* Don't wait for both bits, either bit will do. */
                xTicksToWait );/* Wait a maximum of 100ms for either bit to be set. */

    if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
    {
        /* xEventGroupWaitBits() returned because both bits were set. */
    }
    else if( ( uxBits & BIT_0 ) != 0 )
    {
        /* xEventGroupWaitBits() returned because just BIT_0 was set. */
    }
    else if( ( uxBits & BIT_4 ) != 0 )
    {
        /* xEventGroupWaitBits() returned because just BIT_4 was set. */
    }
    else
    {
        /* xEventGroupWaitBits() returned because xTicksToWait ticks passed
        without either BIT_0 or BIT_4 becoming set. */
    }
}
```

xEventGroupSetBits()

v8.0.0

event_groups.h

```
EventBits_t xEventGroupSetBits( EventGroupHandle_t xEventGroup,
                                const EventBits_t uxBitsToSet );
```

Set bits (flags) within an RTOS event group. This function cannot be called from an interrupt.xEventGroupSetBitsFromISR() is a version that can be called from an interrupt. Setting bits in an event group will automatically unblock tasks that are blocked waiting for the bits.

The RTOS source file FreeRTOS/source/event_groups.c must be included in the build for the xEventGroupSetBits() function to be available.

Parameters:

xEventGroup The event group in which the bits are to be set. The event group must have previously been created using a call to xEventGroupCreate().

uxBitsToSet A bitwise value that indicates the bit or bits to set in the event group. For example, set uxBitsToSet to 0x08 to set only bit 3. Set uxBitsToSet to 0x09 to set bit 3 and bit 0.

Returns:

The value of the event group **at the time the call to xEventGroupSetBits() returns**.

There are two reasons why the returned value might have the bits specified by the uxBitsToSet parameter cleared:

1. If setting a bit results in a task that was waiting for the bit leaving the blocked state then it is possible the bit will have been cleared automatically (see the xClearBitOnExit parameter ofxEventGroupWaitBits()).
2. Any unblocked (or otherwise Ready state) task that has a priority above that of the task that called xEventGroupSetBits() will execute and may change the event group value before the call to xEventGroupSetBits() returns.

Example usage:

```
#define BIT_0    ( 1 << 0 )
#define BIT_4    ( 1 << 4 )

void aFunction( EventGroupHandle_t xEventGroup )
{
EventBits_t uxBits;
```

```
    /* Set bit 0 and bit 4 in xEventGroup. */
    uxBits = xEventGroupSetBits(
                                        xEventGroup,    /* The event group being updated. */
                                        BIT_0 | BIT_4 );/* The bits being set. */

    if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
    {
        /* Both bit 0 and bit 4 remained set when the function returned. */
    }
    else if( ( uxBits & BIT_0 ) != 0 )
    {
        /* Bit 0 remained set when the function returned, but bit 4 was
        cleared.  It might be that bit 4 was cleared automatically as a
        task that was waiting for bit 4 was removed from the Blocked
        state. */
    }
    else if( ( uxBits & BIT_4 ) != 0 )
    {
        /* Bit 4 remained set when the function returned, but bit 0 was
        cleared.  It might be that bit 0 was cleared automatically as a
        task that was waiting for bit 0 was removed from the Blocked
        state. */
    }
    else
    {
        /* Neither bit 0 nor bit 4 remained set.  It might be that a task
        was waiting for both of the bits to be set, and the bits were cleared
        as the task left the Blocked state. */
    }
}
```

xEventGroupSetBitsFromISR()

event_groups.h

```
BaseType_t xEventGroupSetBitsFromISR(
                                    EventGroupHandle_t xEventGroup,
                                    const EventBits_t uxBitsToSet,
                                    BaseType_t *pxHigherPriorityTaskWoken );
```

Set bits (flags) within an RTOS event group. A version of xEventGroupSetBits() that can be called from an interrupt service routine (ISR).
Setting bits in an event group will automatically unblock tasks that are blocked waiting for the bits.
Setting bits in an event group is not a deterministic operation because there are an unknown number of tasks that may be waiting for the bit or bits being set. FreeRTOS does not allow non-deterministic operations to be performed in interrupts or from critical sections. Therefore xEventGroupSetBitFromISR() sends a message to the RTOS daemon task to have the set operation performed in the context of the daemon task - where a scheduler lock is used in place of a critical section.
NOTE: As mentioned in the paragraph above, setting bits from an ISR will defer the set operation to the RTOS daemon task (also known as the timer service task). The RTOS daemon task is scheduled according to its priority, just like any other RTOS task. Therefore, if it is essential the set operation completes immediately (before a task created by the application executes) then the priority of the RTOS daemon task must be higher than the priority of any application task that uses the event group. The priority of the RTOS daemon task is set by the configTIMER_TASK_PRIORITY definition in FreeRTOSConfig.h.
INCLUDE_xEventGroupSetBitFromISR, configUSE_TIMERS and INCLUDE_xTimerPendFunctionCall must all be set to 1 in FreeRTOSConfig.h for the xEventGroupSetBitsFromISR() function to be available.
The RTOS source file FreeRTOS/source/event_groups.c must be included in the build for the xEventGroupSetBitsFromISR() function to be available.
Parameters:

xEventGroup	The event group in which the bits are to be set. The event group must have previously been created using a call toxEventGroupCreate().
uxBitsToSet	A bitwise value that indicates the bit or bits to set. For example, set uxBitsToSet to 0x08 to set only bit 3. Set uxBitsToSet to 0x09 to set bit 3 and bit 0.
pxHigherPriorityTaskWoken	As mentioned above, calling this function will result in a message being sent to the RTOS daemon task.

If the priority of the daemon task is higher than the priority of the currently running task (the task the interrupt interrupted) then *pxHigherPriorityTaskWoken will be set to pdTRUE by xEventGroupSetBitsFromISR(), indicating that a context switch should be requested before the interrupt exits. For that reason *pxHigherPriorityTaskWoken must be initialised to pdFALSE. See the example code below.

Returns:

If the message was sent to the RTOS daemon task then pdPASS is returned, otherwise pdFAIL is returned. pdFAIL will be returned if the timer service queue was full.

Example usage:

```
#define BIT_0	( 1 << 0 )
#define BIT_4	( 1 << 4 )

/* An event group which it is assumed has already been created by a call to
xEventGroupCreate(). */
EventGroupHandle_t xEventGroup;

void anInterruptHandler( void )
{
BaseType_t xHigherPriorityTaskWoken, xResult;

   /* xHigherPriorityTaskWoken must be initialised to pdFALSE. */
   xHigherPriorityTaskWoken = pdFALSE;

   /* Set bit 0 and bit 4 in xEventGroup. */
   xResult = xEventGroupSetBitsFromISR(
                                       xEventGroup,    /* The event group being updated. */
                                       BIT_0 | BIT_4 /* The bits being set. */
                                       &xHigherPriorityTaskWoken );

   /* Was the message posted successfully? */
   if( xResult != pdFAIL )
   {
        /* If xHigherPriorityTaskWoken is now set to pdTRUE then a context
        switch should be requested.    The macro used is port specific and will
        be either portYIELD_FROM_ISR() or portEND_SWITCHING_ISR() - refer to
```

```
        the documentation page for the port being used. */
        portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
    }
}
```

xEventGroupClearBits()

v8.0.0
event_groups.h

```
EventBits_t xEventGroupClearBits(
                                    EventGroupHandle_t xEventGroup,
                                    const EventBits_t uxBitsToClear );
```

Clear bits (flags) within an RTOS event group. This function cannot be called from an interrupt. SeexEventGroupClearBitsFromISR() for a version that can be called from an interrupt.
The RTOS source file FreeRTOS/source/event_groups.c must be included in the build for the xEventGroupClearBits() function to be available.

Parameters:

xEventGroup The event group in which the bits are to be cleared. The event group must have previously been created using a call to xEventGroupCreate().

uxBitsToClear A bitwise value that indicates the bit or bits to clear in the event group. For example set uxBitsToClear to 0x08 to clear just bit 3. Set uxBitsToClear to 0x09 to clear bit 3 and bit 0.

Returns:

The value of the event group **before** the specified bits were cleared.

Example usage:

```
#define BIT_0        ( 1 << 0 )
#define BIT_4        ( 1 << 4 )

void aFunction( EventGroupHandle_t xEventGroup )
{
EventBits_t uxBits;

  /* Clear bit 0 and bit 4 in xEventGroup. */
  uxBits = xEventGroupClearBits(
                                    xEventGroup,    /* The event group being updated. */
                                    BIT_0 | BIT_4 );/* The bits being cleared. */

  if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
  {
      /* Both bit 0 and bit 4 were set before xEventGroupClearBits()
      was called.    Both will now be clear (not set). */
```

```
    }
    else if( ( uxBits & BIT_0 ) != 0 )
    {
        /* Bit 0 was set before xEventGroupClearBits() was called.  It will
        now be clear. */
    }
    else if( ( uxBits & BIT_4 ) != 0 )
    {
        /* Bit 4 was set before xEventGroupClearBits() was called.  It will
        now be clear. */
    }
    else
    {
        /* Neither bit 0 nor bit 4 were set in the first place. */
    }
}
```

xEventGroupClearBitsFromISR()

v8.0.0
event_groups.h
```
 EventBits_t xEventGroupClearBitsFromISR(
                                        EventGroupHandle_t xEventGroup,
                                        const EventBits_t uxBitsToClear );
```

A version of xEventGroupClearBits() that can be called from an interrupt.
The RTOS source file FreeRTOS/source/event_groups.c must be included in the build for the xEventGroupClearBitsFromISR() function to be available.

Parameters:

xEventGroup The event group in which the bits are to be cleared. The event group must have previously been created using a call to xEventGroupCreate().

uxBitsToClear A bitwise value that indicates the bit or bits to clear in the event group. For example set uxBitsToClear to 0x08 to clear just bit 3. Set uxBitsToClear to 0x09 to clear bit 3 and bit 0.

Returns:

The value of the event group **before** the specified bits were cleared.

Example usage:

```
#define BIT_0        ( 1 << 0 )
#define BIT_4        ( 1 << 4 )

/* This code assumes the event group referenced by the
xEventGroup variable has already been created using a call to
xEventGroupCreate(). */
void anInterruptHandler( void )
{
EventBits_t uxBits;

   /* Clear bit 0 and bit 4 in xEventGroup. */
   uxBits = xEventGroupClearBitsFromISR(
                                        xEventGroup, /* The event group being updated. */
                                        BIT_0 | BIT_4 );/* The bits being cleared. */

   if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
   {
         /* Both bit 0 and bit 4 were set before xEventGroupClearBitsFromISR()
```

```
        was called.    Both will now be clear (not set). */
    }
    else if( ( uxBits & BIT_0 ) != 0 )
    {
        /* Bit 0 was set before xEventGroupClearBitsFromISR() was called.    It will
        now be clear. */
    }
    else if( ( uxBits & BIT_4 ) != 0 )
    {
        /* Bit 4 was set before xEventGroupClearBitsFromISR() was called.    It will
        now be clear. */
    }
    else
    {
        /* Neither bit 0 nor bit 4 were set in the first place. */
    }
}
```

xEventGroupGetBits()

v8.0.0
event_groups.h

```
EventBits_t xEventGroupGetBits( EventGroupHandle_t xEventGroup );
```

Returns the current value of the event bits (event flags) in an RTOS event group. This function cannot be used from an interrupt. See xEventGroupsGetBitsFromISR() for a version that can be used in an interrupt.
The RTOS source file FreeRTOS/source/event_groups.c must be included in the build for the xEventGroupGetBits() function to be available.

Parameters:

xEventGroup The event group being queried. The event group must have previously been created using a call to xEventGroupCreate().

Returns:

The value of the event bits in the event group at the time xEventGroupGetBits() was called.

xEventGroupGetBitsFromISR()

v8.0.0

event_groups.h

```
EventBits_t xEventGroupGetBitsFromISR(
                                    EventGroupHandle_t xEventGroup );
```

A version of xEventGroupGetBits() that can be called from an interrupt.
The RTOS source file FreeRTOS/source/event_groups.c must be included in the build for the xEventGroupGetBitsFrom() function to be available.

Parameters:

xEventGroup The event group being queried. The event group must have previously been created using a call to xEventGroupCreate().

Returns:

The value of the event bits in the event group at the time xEventGroupGetBitsFromISR() was called.

xEventGroupSync()

v8.0.0

event_groups.h

```
EventBits_t xEventGroupSync( EventGroupHandle_t xEventGroup,
                             const EventBits_t uxBitsToSet,
                             const EventBits_t uxBitsToWaitFor,
                             TickType_t xTicksToWait );
```

Atomically set bits (flags) within an RTOS event group, then wait for a combination of bits to be set within the same event group. This functionality is typically used to synchronise multiple tasks (often called a task rendezvous), where each task has to wait for the other tasks to reach a synchronisation point before proceeding.

This function cannot be used from an interrupt.

The function will return before its block time expires if the bits specified by the uxBitsToWait parameter are set, or become set within that time. In this case all the bits specified by uxBitsToWait will be automatically cleared before the function returns.

The RTOS source file FreeRTOS/source/event_groups.c must be included in the build for the xEventGroupSync() function to be available.

Parameters:

xEventGroup	The event group in which the bits are being set and tested. The event group must have previously been created using a call to xEventGroupCreate().
uxBitsToSet	The bit or bits to set in the event group before determining if (and possibly waiting for), all the bits specified by the uxBitsToWait parameter are set. For example, set uxBitsToSet to 0x04 to set bit 2 within the event group.
uxBitsToWaitFor	A bitwise value that indicates the bit or bits to test inside the event group. For example, set uxBitsToWaitFor to 0x05 to wait for bits 0 and bit 2. Set uxBitsToWaitFor to 0x07 to wait for bit 0 and bit 1 and bit 2. Etc.
xTicksToWait	The maximum amount of time (specified in 'ticks') to wait for all the bits specified by the uxBitsToWaitFor parameter value to become set.

Returns:

The value of the event group at the time either the bits being waited for became set, or the block time expired. Test the return value to know which bits were set. If xEventGroupSync() returned because its timeout expired then not all the bits being waited for will be set.

If xEventGroupSync() returned because all the bits it was waiting for were set then the returned value is the event group value **before** any bits were automatically cleared.

Example usage:

```
/* Bits used by the three tasks. */
#define TASK_0_BIT          ( 1 << 0 )
#define TASK_1_BIT          ( 1 << 1 )
#define TASK_2_BIT          ( 1 << 2 )

#define ALL_SYNC_BITS ( TASK_0_BIT | TASK_1_BIT | TASK_2_BIT )

/* Use an event group to synchronise three tasks.  It is assumed this event
group has already been created elsewhere. */
EventGroupHandle_t xEventBits;

void vTask0( void *pvParameters )
{
EventBits_t uxReturn;
TickType_t xTicksToWait = 100 / portTICK_PERIOD_MS;

    for( ;; )
    {
        /* Perform task functionality here. */
        . . .

        /* Set bit 0 in the event group to note this task has reached the
        sync point.  The other two tasks will set the other two bits defined
        by ALL_SYNC_BITS.  All three tasks have reached the synchronisation
        point when all the ALL_SYNC_BITS are set.  Wait a maximum of 100ms
        for this to happen. */
        uxReturn = xEventGroupSync( xEventBits,
                                    TASK_0_BIT,
                                    ALL_SYNC_BITS,
                                    xTicksToWait );

        if( ( uxReturn & ALL_SYNC_BITS ) == ALL_SYNC_BITS )
        {
            /* All three tasks reached the synchronisation point before the call
            to xEventGroupSync() timed out. */
        }
    }
}

void vTask1( void *pvParameters )
```

```
{
    for( ;; )
    {
        /* Perform task functionality here. */
        . . .

        /* Set bit 1 in the event group to note this task has reached the
        synchronisation point.  The other two tasks will set the other two
        bits defined by ALL_SYNC_BITS.  All three tasks have reached the
        synchronisation point when all the ALL_SYNC_BITS are set.  Wait
        indefinitely for this to happen. */
        xEventGroupSync( xEventBits, TASK_1_BIT, ALL_SYNC_BITS,
portMAX_DELAY );

        /* xEventGroupSync() was called with an indefinite block time, so
        this task will only reach here if the syncrhonisation was made by all
        three tasks, so there is no need to test the return value. */
    }
}

void vTask2( void *pvParameters )
{
    for( ;; )
    {
        /* Perform task functionality here. */
        . . .

        /* Set bit 2 in the event group to note this task has reached the
        synchronisation point.  The other two tasks will set the other two
        bits defined by ALL_SYNC_BITS.  All three tasks have reached the
        synchronisation point when all the ALL_SYNC_BITS are set.  Wait
        indefinitely for this to happen. */
        xEventGroupSync( xEventBits, TASK_2_BIT, ALL_SYNC_BITS,
portMAX_DELAY );

        /* xEventGroupSync() was called with an indefinite block time, so
        this task will only reach here if the syncrhonisation was made by all
        three tasks, so there is no need to test the return value. */
    }
}
```

12 Co-routine specific

Modules

- CoRoutineHandle_t
- xCoRoutineCreate
- crDELAY
- crQUEUE_SEND
- crQUEUE_RECEIVE
- crQUEUE_SEND_FROM_ISR
- crQUEUE_RECEIVE_FROM_ISR
- vCoRoutineSchedule

CoRoutineHandle_t

Type by which co-routines are referenced. The co-routine handle is automatically passed into each co-routine function.

xCoRoutineCreate

croutine.h

```
BaseType_t xCoRoutineCreate
                    (
                        crCOROUTINE_CODE pxCoRoutineCode,
                        UBaseType_t uxPriority,
                        UBaseType_t uxIndex
                    );
```

Create a new co-routine and add it to the list of co-routines that are ready to run.

Parameters:

pxCoRoutineCode	Pointer to the co-routine function. Co-routine functions require special syntax - see the co-routine section of the web documentation for more information.
uxPriority	The priority with respect to other co-routines at which the co-routine will run..
uxIndex	Used to distinguish between different co-routines that execute the same function. See the example below and the co-routine section of the web documentation for further information.

Returns:

pdPASS if the co-routine was successfully created and added to a ready list, otherwise an error code defined with ProjDefs.h.

Example usage:

```
// Co-routine to be created.
void vFlashCoRoutine( CoRoutineHandle_t xHandle, UBaseType_t uxIndex )
{
// Variables in co-routines must be declared static if they must
// maintain value across a blocking call. This may not be necessary
// for const variables.
static const char cLedToFlash[ 2 ] = { 5, 6 };
static const TickType_t uxFlashRates[ 2 ] = { 200, 400 };

    // Must start every co-routine with a call to crSTART();
    crSTART( xHandle );
```

```
    for( ;; )
    {
        // This co-routine just delays for a fixed period, then toggles
        // an LED.  Two co-routines are created using this function, so
        // the uxIndex parameter is used to tell the co-routine which
        // LED to flash and how long to delay.  This assumes xQueue has
        // already been created.
        vParTestToggleLED( cLedToFlash[ uxIndex ] );
        crDELAY( xHandle, uxFlashRates[ uxIndex ] );
    }

    // Must end every co-routine with a call to crEND();
    crEND();
}

// Function that creates two co-routines.
void vOtherFunction( void )
{
unsigned char ucParameterToPass;
TaskHandle_t xHandle;

    // Create two co-routines at priority 0.  The first is given index 0
    // so (from the code above) toggles LED 5 every 200 ticks.  The second
    // is given index 1 so toggles LED 6 every 400 ticks.
    for( uxIndex = 0; uxIndex < 2; uxIndex++ )
    {
        xCoRoutineCreate( vFlashCoRoutine, 0, uxIndex );
    }
}
```

crDELAY

croutine.h

```
void crDELAY( CoRoutineHandle_t xHandle,
              TickType_t xTicksToDelay )
```

crDELAY is a macro. The data types in the prototype above are shown for reference only.
Delay a co-routine for a fixed period of time.
crDELAY can only be called from the co-routine function itself - not from within a function called by the co-routine function. This is because co-routines do not maintain their own stack.

Parameters:

xHandle	The handle of the co-routine to delay. This is the xHandle parameter of the co-routine function.
xTickToDelay	The number of ticks that the co-routine should delay for. The actual amount of time this equates to is defined by configTICK_RATE_HZ (set in FreeRTOSConfig.h). The constant portTICK_PERIOD_MS can be used to convert ticks to milliseconds.

Example usage:

```
// Co-routine to be created.
void vACoRoutine( CoRoutineHandle_t xHandle,
UBaseType_t uxIndex )
{
// Variables in co-routines must be declared static if they must maintain
// value across a blocking call. This may not be necessary for const
// variables. We are to delay for 200ms.
static const xTickType xDelayTime = 200 / portTICK_PERIOD_MS;

    // Must start every co-routine with a call to crSTART();
    crSTART( xHandle );

    for( ;; )
    {
        // Delay for 200ms.
        crDELAY( xHandle, xDelayTime );
```

```
        // Do something here.
    }

    // Must end every co-routine with a call to crEND();
    crEND();
}
```

crQUEUE_SEND

croutine.h

```
 crQUEUE_SEND(
                    CoRoutineHandle_t xHandle,
                    QueueHandle_t xQueue,
                    void *pvItemToQueue,
                    TickType_t xTicksToWait,
                    BaseType_t *pxResult
                )
```

crQUEUE_SEND is a macro. The data types are shown in the prototype above for reference only.

The macro's crQUEUE_SEND() and crQUEUE_RECEIVE() are the co-routine equivalent to the xQueueSend() and xQueueReceive() functions used by tasks.

crQUEUE_SEND and crQUEUE_RECEIVE can only be used from a co-routine whereas xQueueSend() and xQueueReceive() can only be used from tasks. **Note** that co-routines can only send data to other co-routines. A co-routine cannot use a queue to send data to a task or vice versa.

crQUEUE_SEND can only be called from the co-routine function itself - not from within a function called by the co-routine function. This is because co-routines do not maintain their own stack.

See the co-routine section of the web documentation for information on passing data between tasks and co-routines and between ISR's and co-routines.

Parameters:

xHandle	The handle of the calling co-routine. This is the xHandle parameter of the co-routine function.
xQueue	The handle of the queue on which the data will be posted. The handle is obtained as the return value when the queue is created using the xQueueCreate() API function.
pvItemToQueue	A pointer to the data being posted onto the queue. The number of bytes of each queued item is specified when the queue is created. This number of bytes is copied from pvItemToQueue into the queue itself.
xTickToDelay	The number of ticks that the co-routine should block to wait for space to become available on the queue, should space not be available immediately. The actual amount of time this equates to is defined by configTICK_RATE_HZ (set in

FreeRTOSConfig.h). The constant portTICK_PERIOD_MS can be used to convert ticks to milliseconds (see example below).

pxResult The variable pointed to by pxResult will be set to pdPASS if data was successfully posted onto the queue, otherwise it will be set to an error defined within ProjDefs.h.

Example usage:

```
// Co-routine function that blocks for a fixed period then posts a number onto
// a queue.
static void prvCoRoutineFlashTask( CoRoutineHandle_t xHandle,
                                   UBaseType_t uxIndex )
{
// Variables in co-routines must be declared static if they must maintain
// value across a blocking call.
static BaseType_t xNumberToPost = 0;
static BaseType_t xResult;

    // Co-routines must begin with a call to crSTART().
    crSTART( xHandle );

    for( ;; )
    {
        // This assumes the queue has already been created.
        crQUEUE_SEND( xHandle,
                      xCoRoutineQueue,
                      &xNumberToPost,
                      NO_DELAY,
                      &xResult );

        if( xResult != pdPASS )
        {
            // The message was not posted!
        }

        // Increment the number to be posted onto the queue.
        xNumberToPost++;

        // Delay for 100 ticks.
        crDELAY( xHandle, 100 );
    }
    // Co-routines must end with a call to crEND().
    crEND();
}
```

crQUEUE_RECEIVE

croutine.h

```
void crQUEUE_RECEIVE(
                        CoRoutineHandle_t xHandle,
                        QueueHandle_t xQueue,
                        void *pvBuffer,
                        TickType_t xTicksToWait,
                        BaseType_t *pxResult
                    )
```

crQUEUE_RECEIVE is a macro. The data types are shown in the prototype above for reference only.
The macro's crQUEUE_SEND() and crQUEUE_RECEIVE() are the co-routine equivalent to the xQueueSend() and xQueueReceive() functions used by tasks.
crQUEUE_SEND and crQUEUE_RECEIVE can only be used from a co-routine whereas xQueueSend() and xQueueReceive() can only be used from tasks. **Note** that co-routines can only send data to other co-routines. A co-routine cannot use a queue to send data to a task or vice versa.
crQUEUE_RECEIVE can only be called from the co-routine function itself - not from within a function called by the co-routine function. This is because co-routines do not maintain their own stack.
See the co-routine section of the web documentation for information on passing data between tasks and co-routines and between ISR's and co-routines.

Parameters:

xHandle The handle of the calling co-routine. This is the xHandle parameter of the co-routine function.

xQueue The handle of the queue from which the data will be received. The handle is obtained as the return value when the queue is created using the xQueueCreate() API function.

pvBuffer The buffer into which the received item is to be copied. The number of bytes of each queued item is specified when the queue is created. This number of bytes is copied into pvBuffer.

xTickToDelay The number of ticks that the co-routine should block to wait for data to become available from the queue, should data not be available immediately. The actual amount of time this equates to is defined by configTICK_RATE_HZ (set in FreeRTOSConfig.h). The

constant portTICK_PERIOD_MS can be used to convert ticks to milliseconds (see the crQUEUE_SEND example).

pxResult The variable pointed to by pxResult will be set to pdPASS if data was successfully retrieved from the queue, otherwise it will be set to an error code as defined within ProjDefs.h.

Example usage:

```
// A co-routine receives the number of an LED to flash from a queue.  It
// blocks on the queue until the number is received.
static void prvCoRoutineFlashWorkTask( CoRoutineHandle_t xHandle,
                                       UBaseType_t uxIndex )
{
// Variables in co-routines must be declared static if they must maintain
// value across a blocking call.
static BaseType_t xResult;
static UBaseType_t uxLEDToFlash;

    // All co-routines must start with a call to crSTART().
    crSTART( xHandle );

    for( ;; )
    {
        // Wait for data to become available on the queue.
        crQUEUE_RECEIVE( xHandle,
                         xCoRoutineQueue,
                         &uxLEDToFlash,
                         portMAX_DELAY,
                         &xResult );

        if( xResult == pdPASS )
        {
            // We received the LED to flash - flash it!
            vParTestToggleLED( uxLEDToFlash );
        }
    }

    crEND();
}
```

crQUEUE_SEND_FROM_ISR

croutine.h

```
BaseType_t crQUEUE_SEND_FROM_ISR
                        (
                            QueueHandle_t xQueue,
                            void *pvItemToQueue,
                            BaseType_t xCoRoutinePreviouslyWoken
                        )
```

crQUEUE_SEND_FROM_ISR() is a macro. The data types are shown in the prototype above for reference only.
The macro's crQUEUE_SEND_FROM_ISR() and crQUEUE_RECEIVE_FROM_ISR() are the co-routine equivalent to the xQueueSendFromISR() and xQueueReceiveFromISR() functions used by tasks.
crQUEUE_SEND_FROM_ISR() and crQUEUE_RECEIVE_FROM_ISR() can only be used to pass data between a co-routine and and ISR, whereas xQueueSendFromISR() and xQueueReceiveFromISR() can only be used to pass data between a task and and ISR.
crQUEUE_SEND_FROM_ISR can only be called from an ISR to send data to a queue that is being used from within a co-routine.
See the co-routine section of the web documentation for information on passing data between tasks and co-routines and between ISR's and co-routines.

Parameters:

xQueue	The handle to the queue on which the item is to be posted.
pvItemToQueue	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
xCoRoutinePreviouslyWoken	This is included so an ISR can post onto the same queue multiple times from a single interrupt. The first call should always pass in pdFALSE. Subsequent calls should pass in the value returned from the previous call.

Returns:

pdTRUE if a co-routine was woken by posting onto the queue. This is used by the ISR to determine if a context switch may be required following the ISR.

Example usage:

```
// A co-routine that blocks on a queue waiting for characters to be received.
static void vReceivingCoRoutine( CoRoutineHandle_t xHandle,
                                 UBaseType_t uxIndex )
{
char cRxedChar;
BaseType_t xResult;

    // All co-routines must start with a call to crSTART().
    crSTART( xHandle );

    for( ;; )
    {
        // Wait for data to become available on the queue.  This assumes the
        // queue xCommsRxQueue has already been created!
        crQUEUE_RECEIVE( xHandle,
                         xCommsRxQueue,
                         &uxLEDToFlash,
                         portMAX_DELAY,
                         &xResult );

        // Was a character received?
        if( xResult == pdPASS )
        {
            // Process the character here.
        }
    }

    // All co-routines must end with a call to crEND().
    crEND();
}

// An ISR that uses a queue to send characters received on a serial port to
// a co-routine.
void vUART_ISR( void )
{
char cRxedChar;
BaseType_t xCRWokenByPost = pdFALSE;
```

```
    // We loop around reading characters until there are none left in the UART.
    while( UART_RX_REG_NOT_EMPTY() )
    {
        // Obtain the character from the UART.
        cRxedChar = UART_RX_REG;

        // Post the character onto a queue.  xCRWokenByPost will be pdFALSE
        // the first time around the loop.  If the post causes a co-routine
        // to be woken (unblocked) then xCRWokenByPost will be set to pdTRUE.
        // In this manner we can ensure that if more than one co-routine is
        // blocked on the queue only one is woken by this ISR no matter how
        // many characters are posted to the queue.
        xCRWokenByPost = crQUEUE_SEND_FROM_ISR( xCommsRxQueue,
                                                &cRxedChar,
                                                xCRWokenByPost );
    }
}
```

crQUEUE_RECEIVE_FROM_ISR

croutine.h

```
BaseType_t crQUEUE_SEND_FROM_ISR
                        (
                            QueueHandle_t xQueue,
                            void *pvBuffer,
                            BaseType_t * pxCoRoutineWoken
                        )
```

The macro's crQUEUE_SEND_FROM_ISR() and crQUEUE_RECEIVE_FROM_ISR() are the co-routine equivalent to the xQueueSendFromISR() and xQueueReceiveFromISR() functions used by tasks.
crQUEUE_SEND_FROM_ISR() and crQUEUE_RECEIVE_FROM_ISR() can only be used to pass data between a co-routine and and ISR, whereas xQueueSendFromISR() and xQueueReceiveFromISR() can only be used to pass data between a task and and ISR.
crQUEUE_RECEIVE_FROM_ISR can only be called from an ISR to receive data from a queue that is being used from within a co-routine (a co-routine posted to the queue).
See the co-routine section of the web documentation for information on passing data between tasks and co-routines and between ISR's and co-routines.

Parameters:

xQueue	The handle to the queue on which the item is to be posted.
pvBuffer	A pointer to a buffer into which the received item will be placed. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from the queue into pvBuffer.
pxCoRoutineWoken	A co-routine may be blocked waiting for space to become available on the queue. If crQUEUE_RECEIVE_FROM_ISR causes such a co-routine to unblock *pxCoRoutineWoken will get set to pdTRUE, otherwise *pxCoRoutineWoken will remain unchanged

Returns:

pdTRUE an item was successfully received from the queue, otherwise pdFALSE.

Example usage:

```
 // A co-routine that posts a character to a queue then blocks for a fixed
```

```
// period.    The character is incremented each time.
static void vSendingCoRoutine( CoRoutineHandle_t xHandle,
                               UBaseType_t uxIndex )
{
// cChar holds its value while this co-routine is blocked and must therefore
// be declared static.
static char cCharToTx = 'a';
BaseType_t xResult;

    // All co-routines must start with a call to crSTART().
    crSTART( xHandle );

    for( ;; )
    {
        // Send the next character to the queue.
        crQUEUE_SEND( xHandle,
                      xCoRoutineQueue,
                      &cCharToTx,
                      NO_DELAY,
                      &xResult );

        if( xResult == pdPASS )
        {
            // The character was successfully posted to the queue.
        }
        else
        {
            // Could not post the character to the queue.
        }

        // Enable the UART Tx interrupt to cause an interrupt in this
        // hypothetical UART.    The interrupt will obtain the character
        // from the queue and send it.
        ENABLE_RX_INTERRUPT();

        // Increment to the next character then block for a fixed period.
        // cCharToTx will maintain its value across the delay as it is
        // declared static.
        cCharToTx++;
        if( cCharToTx > 'x' )
        {
            cCharToTx = 'a';
        }
        crDELAY( 100 );
```

```
        }

        // All co-routines must end with a call to crEND().
        crEND();
    }

    // An ISR that uses a queue to receive characters to send on a UART.
    void vUART_ISR( void )
    {
    char cCharToTx;
    BaseType_t xCRWokenByPost = pdFALSE;

        while( UART_TX_REG_EMPTY() )
        {
            // Are there any characters in the queue waiting to be sent?
            // xCRWokenByPost will automatically be set to pdTRUE if a co-routine
            // is woken by the post - ensuring that only a single co-routine is
            // woken no matter how many times we go around this loop.
            if( crQUEUE_RECEIVE_FROM_ISR( xQueue, &cCharToTx,
&xCRWokenByPost ) )
            {
                SEND_CHARACTER( cCharToTx );
            }
        }
    }
```

vCoRoutineSchedule

croutine.h

void vCoRoutineSchedule(void);

Run a co-routine.
vCoRoutineSchedule() executes the highest priority co-routine that is able to run. The co-routine will execute until it either blocks, yields or is preempted by a task. Co-routines execute cooperatively so one co-routine cannot be preempted by another, but can be preempted by a task.
If an application comprises of both tasks and co-routines then vCoRoutineSchedule should be called from the idle task (in an idle task hook).

Example usage:

```
void vApplicationIdleHook( void )
{
    vCoRoutineSchedule( void );
}
```

Alternatively, if the idle task is not performing any other function it would be more efficient to call vCoRoutineSchedule() from within a loop as:

```
void vApplicationIdleHook( void )
{
    for( ;; )
    {
        vCoRoutineSchedule( void );
    }
}
```