FreeRTOS API Reference

Version 7.0.0 r0

4/19/2011 9:47:00 AM

Table of Contents

Main Page

Module Index

Modules

Here is a list of all modules:

Data Structure Index

Data Structures

Here are the data structures with brief descriptions:

Module Documentation

Task Creation

xTaskHandle

task. h

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

xTaskCreate

task. h

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

Create a new task and add it to the list of tasks that are ready to run.

xTaskCreate() can only be used to create a task that has unrestricted access to the entire microcontroller memory map. Systems that include MPU support can alternatively create an MPU constrained task using xTaskCreateRestricted().

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 tskMAX_TASK_NAME_LEN - default is 16.
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.
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 errors. 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;
xTaskHandle xHandle;

 // 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 );

 // Use the handle to delete the task.
 vTaskDelete( xHandle );
}
```

vTaskDelete

task. h

```
void vTaskDelete( xTaskHandle pxTask );
```

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 real time 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 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:

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

Example usage:

```
void vOtherFunction( void )
{
xTaskHandle xHandle;

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

 // Use the handle to delete the task.
 vTaskDelete( xHandle );
}
```

Task Control

vTaskDelay

task. h

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

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_RATE_MS can be used to calculate real time from the tick rate - with the resolution of one tick period.

INCLUDE_vTaskDelay must be defined as 1 for this function to be available. See the configuration section for more information.

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 cyclical 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 portTickType xDelay = 500 / portTICK_RATE_MS;

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

vTaskDelayUntil

task. h

```
void vTaskDelayUntil( portTickType *pxPreviousWakeTime, portTickType 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 cyclical tasks to ensure a constant execution frequency.

This function differs from vTaskDelay () in one important aspect: 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 starting to execute and that task 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.

The constant portTICK_RATE_MS can be used to calculate real time from the tick rate - with the resolution of one tick period.

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 interface period.

Example usage:

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

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

INCLUDE_xTaskPriorityGet 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:

pxTask	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 pxTask.

Example usage:

```
void vAFunction( void )
{
xTaskHandle 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 it's 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( xTaskHandle pxTask, unsigned portBASE_TYPE 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:

pxTask	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 )
{
xTaskHandle 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( xTaskHandle pxTaskToSuspend );
```

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:

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

Example usage:

```
void vAFunction( void )
{
xTaskHandle 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( xTaskHandle pxTaskToResume );
```

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 of more calls to vTaskSuspend () will be made available for running again by a single call to vTaskResume ().

Parameters:

pxTaskToResume	Handle to the task being readied.

Example usage:

```
void vAFunction( void )
{
xTaskHandle 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 it priority within the system.
}
```

vTaskResumeFromISR

task. h

```
void xTaskResumeFromISR( xTaskHandle pxTaskToResume );
```

INCLUDE_xTaskResumeFromISR must be defined as 1 for this function to be available. See the configuration section for more information.

An implementation of vTaskResume() 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 ().

Parameters:

pxTaskToResume	Handle to the task being readied.

Task Utilities

xTaskGetTickCount

task. h

```
portTickType xTaskGetTickCount( void );
```

Returns:

The count of ticks since vTaskStartScheduler was called.

xTaskGetTickCountFromISR

task. h

```
portTickType xTaskGetTickCountFromISR( void );
```

Returns:

The count of ticks since vTaskStartScheduler was called.

This is a version of xTaskGetTickCount() that is safe to be called from an ISR - provided that portTickType is the natural word size of the microcontroller being used or interrupt nesting is either not supported or not being used.

uxTaskGetNumberOfTasks

task. h

```
unsigned short uxTaskGetNumberOfTasks( void );
```

Returns:

The number of tasks that the real time 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 must be defined as 1 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.

Lists all the current tasks, along with their current state and stack usage high water mark.

Tasks are reported as blocked ('B'), ready ('R'), deleted ('D') or suspended ('S').

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.

vTaskGetRunTimeStats

task. h

```
void vTaskGetRunTimeStats( char *pcWriteBuffer );
```

configGENERATE_RUN_TIME_STATS must 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.

Setting configGENERATE_RUN_TIME_STATS to 1 will result in a total accumulated execution time being stored for each task. The resolution of the accumulated time value depends on the frequency of the timer configured by the portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() macro. Calling vTaskGetRunTimeStats() writes the total execution time of each task into a buffer, both as an absolute count value and as a percentage of the total system execution time.

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.

vTaskStartTrace

task. h

```
void vTaskStartTrace( char * pcBuffer, unsigned portBASE_TYPE uxBufferSize );
```

Starts a real time 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.

usTaskEndTrace

task. h

```
unsigned long ulTaskEndTrace( void );
```

Stops a kernel activity trace. See vTaskStartTrace ().

Returns:

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

uxTaskGetStackHighWaterMark

task.h

```
unsigned portBASE_TYPE uxTaskGetStackHighWaterMark( xTaskHandle xTask );
```

INCLUDE_uxTaskGetStackHighWaterMark must be set to 1 in FreeRTOSConfig.h for this function to be available.

Returns the high water mark of the stack associated with xTask. That is, the minimum free stack space there has been (in words, so on a 32 bit machine a value of 1 means 4 bytes) since the task started. The smaller the returned number the closer the task has come to overflowing its stack.

Parameters:

xTask	Handle of the task associated with the stack to be checked. Set xTask to NULL to check the stack of the calling task.

Returns:

The smallest amount of free stack space there has been (in bytes) since the task referenced by xTask was created.

xTaskCallApplicationTaskHook

task.h

```
portBASE_TYPE xTaskCallApplicationTaskHook( xTaskHandle xTask, pdTASK_HOOK_CODE pxHookFunction );
```

Calls the hook function associated with xTask. Passing xTask as NULL has the effect of calling the Running tasks (the calling task) hook function.

pvParameter is passed to the hook function for the task to interpret as it wants.

Kernel Control

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 real time kernel tick processing. After calling the kernel has control over which tasks are executed and when. This function does not return until an executing task calls vTaskEndScheduler ().

At least one task should be created via a call to xTaskCreate () before calling vTaskStartScheduler (). The idle task is created automatically when the first application task is created.

See the demo application file main.c for an example of creating tasks and starting the kernel.

Example usage:

```
void vAFunction( void )
{
 // Create at least one task before starting the kernel.
 xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, NULL );

 // Start the real time kernel with preemption.
 vTaskStartScheduler ();

 // Will not get here unless a task calls vTaskEndScheduler ()
}
```

vTaskEndScheduler

task. h

```
void vTaskEndScheduler( void );
```

Stops the real time 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 kernel tick.

vTaskEndScheduler () will cause all of the resources allocated by the 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 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 all real time kernel activity while keeping interrupts (including the kernel tick) enabled.

After calling vTaskSuspendAll () the calling task will continue to execute without risk of being swapped out until a call to xTaskResumeAll () has been made.

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 real time 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 kernel
   // tick count will be maintained.

   // ...

   // The operation is complete.  Restart the kernel.
   xTaskResumeAll ();
  }
 }
```

xTaskResumeAll

task. h

```
 char xTaskResumeAll( void );
```

Resumes real time kernel activity following a call to vTaskSuspendAll (). After a call to vTaskSuspendAll () the kernel will take control of which task is executing at any time.

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 real time 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 kernel tick count will be maintained.

  // ...

  // The operation is complete.  Restart the 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 ();
  }
 }
}
```

FreeRTOS-MPU Specific

xTaskCreateRestricted

task. h

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

xTaskCreateRestricted() should only be used in systems that include an MPU implementation.

Create a new task and add it to the list of tasks that are ready to run. The function parameters define the memory regions and associated access permissions allocated to the task.

Parameters:

pxTaskDefinition	Pointer to a structure that contains a member for each of the normal xTaskCreate() parameters (see the xTaskCreate() API documentation) plus an optional stack buffer and the memory region definitions.
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 errors. h

Example usage:

```
// Create an xTaskParameters structure that defines the task to be created.
static const xTaskParameters xCheckTaskParameters =
{
 vATask,  // pvTaskCode - the function that implements the task.
 "ATask", // pcName - just a text name for the task to assist debugging.
 100,  // usStackDepth - the stack size DEFINED IN WORDS.
 NULL,  // pvParameters - passed into the task function as the function parameters.
 ( 1UL | portPRIVILEGE_BIT ),// uxPriority - task priority, set the portPRIVILEGE_BIT if the task should
run in a privileged state.
 cStackBuffer,// puxStackBuffer - the buffer to be used as the task stack.

 // xRegions - Allocate up to three separate memory regions for access by
 // the task, with appropriate access permissions.  Different processors have
 // different memory alignment requirements - refer to the FreeRTOS documentation
 // for full information.
 {
  // Base address     Length Parameters
        { cReadWriteArray,    32,  portMPU_REGION_READ_WRITE },
        { cReadOnlyArray,    32,  portMPU_REGION_READ_ONLY },
        { cPrivilegedOnlyAccessArray, 128, portMPU_REGION_PRIVILEGED_READ_WRITE }
 }
};

int main( void )
{
xTaskHandle xHandle;

 // Create a task from the const structure defined above.  The task handle
 // is requested (the second parameter is not NULL) but in this case just for
 // demonstration purposes as its not actually used.
 xTaskCreateRestricted( &xRegTest1Parameters, &xHandle );

 // Start the scheduler.
 vTaskStartScheduler();

 // Will only get here if there was insufficient memory to create the idle
 // task.
 for( ;; );
}
```

vTaskAllocateMPURegions

task. h

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

Memory regions are assigned to a restricted task when the task is created by a call to xTaskCreateRestricted(). These regions can be redefined using vTaskAllocateMPURegions().

Parameters:

xTask	The handle of the task being updated.
xRegions	A pointer to an xMemoryRegion structure that contains the new memory region definitions.

Example usage:

```
// Define an array of xMemoryRegion 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 xMemoryRegion 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).
}
```

Queues

xQueueCreate

queue. h

```
 xQueueHandle xQueueCreate(
         unsigned portBASE_TYPE uxQueueLength,
         unsigned portBASE_TYPE 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 )
 {
 xQueueHandle 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.
}
```

xQueueSendToFront

queue. h

```
portBASE_TYPE xQueueSendToFront(
          xQueueHandle xQueue,
          const void * pvItemToQueue,
          portTickType 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 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 this is set to 0 and the queue is full. The time is defined in tick periods so the constant portTICK_RATE_MS should be used to convert to real time if this is required.

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 )
{
xQueueHandle 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, ( portTickType ) 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, ( portTickType ) 0 );
}

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

xQueueSendToBack

queue. h

```
portBASE_TYPE xQueueSendToBack(
          xQueueHandle xQueue,
          const void * pvItemToQueue,
          portTickType xTicksToWait
         );
```

This is a macro that calls xQueueGenericSend().

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 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 this is set to 0 and the queue is full. The time is defined in tick periods so the constant portTICK_RATE_MS should be used to convert to real time if this is required.

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 )
{
xQueueHandle 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, ( portTickType ) 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;
 xQueueSendToBack( xQueue2, ( void * ) &pxMessage, ( portTickType ) 0 );
}

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

xQueueSend

queue. h

```
portBASE_TYPE xQueueSend(
        xQueueHandle xQueue,
        const void * pvItemToQueue,
        portTickType xTicksToWait
      );
```

This is a macro that calls xQueueGenericSend(). It is included for backward compatibility with versions of FreeRTOS.org 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 this is set to 0 and the queue is full. The time is defined in tick periods so the constant portTICK_RATE_MS should be used to convert to real time if this is required.

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 )
{
xQueueHandle 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, ( portTickType ) 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, ( portTickType ) 0 );
}

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

xQueueGenericSend

queue. h

```
portBASE_TYPE xQueueGenericSend(
        xQueueHandle xQueue,
        const void * pvItemToQueue,
        portTickType xTicksToWait
        portBASE_TYPE xCopyPosition
       );
```

It is preferred that the macros xQueueSend(), xQueueSendToFront() and xQueueSendToBack() are used in place of calling this function directly.

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 this is set to 0 and the queue is full. The time is defined in tick periods so the constant portTICK_RATE_MS should be used to convert to real time if this is required.
xCopyPosition	Can take the value queueSEND_TO_BACK to place the item at the back of the queue, or queueSEND_TO_FRONT to place the item at the front of the queue (for high priority messages).

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 )
{
xQueueHandle 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( xQueueGenericSend( xQueue1, ( void * ) &ulVar, ( portTickType ) 10, queueSEND_TO_BACK ) != 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;
 xQueueGenericSend( xQueue2, ( void * ) &pxMessage, ( portTickType ) 0, queueSEND_TO_BACK );
}

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

xQueuePeek

queue. h

```
portBASE_TYPE xQueuePeek(
       xQueueHandle xQueue,
       void *pvBuffer,
       portTickType 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:

pxQueue	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. The time is defined in tick periods so the constant portTICK_RATE_MS should be used to convert to real time if this is required. xQueuePeek() will return immediately if xTicksToWait is 0 and the queue is empty.

Returns:

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

Example usage:

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

xQueueHandle 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, ( portTickType ) 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 ), ( portTickType ) 10 ) )
 {
  // pcRxedMessage now points to the struct AMessage variable posted
  // by vATask, but the item still remains on the queue.
 }
}

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

xQueueReceive

queue. h

```
portBASE_TYPE xQueueReceive(
        xQueueHandle xQueue,
        void *pvBuffer,
        portTickType 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.

Successfully received items are removed from the queue.

This function must not be used in an interrupt service routine. See xQueueReceiveFromISR for an alternative that can.

Parameters:

pxQueue	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. xQueueReceive() will return immediately if xTicksToWait is zero and the queue is empty. The time is defined in tick periods so the constant portTICK_RATE_MS should be used to convert to real time if this is required.

Returns:

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

Example usage:

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

xQueueHandle 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, ( portTickType ) 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 ), ( portTickType ) 10 ) )
 {
  // pcRxedMessage now points to the struct AMessage variable posted
  // by vATask.
 }
}
// ... Rest of task code.
}
```

xQueueGenericReceive

queue. h

```
portBASE_TYPE xQueueGenericReceive(
           xQueueHandle xQueue,
           void *pvBuffer,
           portTickType xTicksToWait
           portBASE_TYPE xJustPeek
        );
```

It is preferred that the macro xQueueReceive() be used rather than calling this function directly.

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:

pxQueue	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. The time is defined in tick periods so the constant portTICK_RATE_MS should be used to convert to real time if this is required. xQueueGenericReceive() will return immediately if the queue is empty and xTicksToWait is 0.
xJustPeek	When set to true, the item received from the queue is not actually removed from the queue - meaning a subsequent call to xQueueReceive() will return the same item. When set to false, the item being received from the queue is also removed from the queue.

Returns:

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

Example usage:

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

xQueueHandle 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, ( portTickType ) 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( xQueueGenericReceive( xQueue, &( pxRxedMessage ), ( portTickType ) 10 ) )
 {
  // pcRxedMessage now points to the struct AMessage variable posted
  // by vATask.
 }
}

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

uxQueueMessagesWaiting

queue. h

```
unsigned portBASE_TYPE uxQueueMessagesWaiting( const xQueueHandle 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.

vQueueDelete

queue. h

```
void vQueueDelete( xQueueHandle 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.

xQueueSendToFrontFromISR

queue. h

```
portBASE_TYPE xQueueSendToFrontFromISR(
          xQueueHandle pxQueue,
          const void *pvItemToQueue,
          portBASE_TYPE *pxHigherPriorityTaskWoken
          );
```

This is a macro that calls xQueueGenericSendFromISR().

Post an item to the front of a queue. It is safe to use this macro 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	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 xQueueSendToFromFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.

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;
portBASE_TYPE xHigherPrioritTaskWoken;

// 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.
 xQueueSendToFrontFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );

} while( portINPUT_BYTE( BUFFER_COUNT ) );

// Now the buffer is empty we can switch context if necessary.
if( xHigherPriorityTaskWoken )
{
 taskYIELD ();
}
}
```

xQueueSendToBackFromISR

queue. h

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

This is a macro that calls xQueueGenericSendFromISR().

Post an item to the back of a queue. It is safe to use this macro 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	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.

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;
portBASE_TYPE 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 )
{
 taskYIELD ();
}
}
```

xQueueSendFromISR

queue. h

```
portBASE_TYPE xQueueSendFromISR(
         xQueueHandle pxQueue,
         const void *pvItemToQueue,
         portBASE_TYPE *pxHigherPriorityTaskWoken
       );
```

This is a macro that calls xQueueGenericSendFromISR(). It is included for backward compatibility with versions of FreeRTOS.org that did not include the xQueueSendToBackFromISR() and xQueueSendToFrontFromISR() macros.

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. 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.

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;
portBASE_TYPE 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 ();
}
}
```

xQueueGenericSendFromISR

queue. h

```
portBASE_TYPE xQueueGenericSendFromISR(
            xQueueHandle pxQueue,
            const void *pvItemToQueue,
            portBASE_TYPE *pxHigherPriorityTaskWoken,
            portBASE_TYPE xCopyPosition
           );
```

It is preferred that the macros xQueueSendFromISR(), xQueueSendToFrontFromISR() and xQueueSendToBackFromISR() be used in place of calling this function directly.

Post an item on 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	xQueueGenericSendFromISR() 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 xQueueGenericSendFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited.
xCopyPosition	Can take the value queueSEND_TO_BACK to place the item at the back of

	the queue, or queueSEND_TO_FRONT to place the item at the front of the queue (for high priority messages).

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;
portBASE_TYPE xHigherPriorityTaskWokenByPost;

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

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

 // Post each byte.
 xQueueGenericSendFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWokenByPost, queueSEND_TO_BACK );

} while( portINPUT_BYTE( BUFFER_COUNT ) );

// Now the buffer is empty we can switch context if necessary.  Note that the
// name of the yield function required is port specific.
if( xHigherPriorityTaskWokenByPost )
{
 taskYIELD_YIELD_FROM_ISR();
}
}
```

xQueueReceiveFromISR

queue. h

```
portBASE_TYPE xQueueReceiveFromISR(
           xQueueHandle pxQueue,
           void *pvBuffer,
           portBASE_TYPE *pxTaskWoken
          );
```

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

Parameters:

pxQueue	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.
pxTaskWoken	A task may be blocked waiting for space to become available on the queue. If xQueueReceiveFromISR causes such a task to unblock *pxTaskWoken will get set to pdTRUE, otherwise *pxTaskWoken will remain unchanged.

Returns:

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

Example usage:

```
xQueueHandle xQueue;

// Function to create a queue and post some values.
void vAFunction( void *pvParameters )
{
char cValueToPost;
const portTickType xBlockTime = ( portTickType )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 xBlockTime ticks.
cValueToPost = 'a';
xQueueSend( xQueue, ( void * ) &cValueToPost, xBlockTime );
cValueToPost = 'b';
xQueueSend( xQueue, ( void * ) &cValueToPost, xBlockTime );

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

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

// ISR that outputs all the characters received on the queue.
void vISR_Routine( void )
{
portBASE_TYPE 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 cTaskWokenByReceive will have been set to
 // pdTRUE.  No matter how many times this loop iterates only one
 // task will be woken.
}

if( cTaskWokenByPost != ( char ) pdFALSE;
{
 taskYIELD ();
}
}
```

Semaphore / Mutexes

vSemaphoreCreateBinary

semphr. h

```
vSemaphoreCreateBinary( xSemaphoreHandle xSemaphore )
```

Macro that implements 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.

This type of semaphore can be used for pure synchronisation between tasks or between an interrupt and a task. The semaphore need not be given back once obtained, so one task/interrupt can continuously 'give' the semaphore while another continuously 'takes' the semaphore. For this reason this type of semaphore does not use a priority inheritance mechanism. For an alternative that does use priority inheritance see xSemaphoreCreateMutex().

Parameters:

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

Example usage:

```
xSemaphoreHandle 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.
   }
}
```

xSemaphoreTake

semphr. h

```
xSemaphoreTake(
                  xSemaphoreHandle xSemaphore,
                  portTickType xBlockTime
              )
```

Macro to obtain a semaphore. The semaphore must have previously been created with a call to vSemaphoreCreateBinary(), xSemaphoreCreateMutex() or xSemaphoreCreateCounting().

Parameters:

xSemaphore	A handle to the semaphore being taken - obtained when the semaphore was created.
xBlockTime	The time in ticks to wait for the semaphore to become available. The macro portTICK_RATE_MS can be used to convert this to a real time. A block time of zero can be used to poll the semaphore. A block time of portMAX_DELAY can be used to block indefinitely (provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h).

Returns:

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

Example usage:

```
xSemaphoreHandle xSemaphore = NULL;

// A task that creates a semaphore.
void vATask( void * pvParameters )
{
   // Create the semaphore to guard a shared resource.
   vSemaphoreCreateBinary( xSemaphore );
}

// 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, ( portTickType ) 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.
        }
    }
}
```

xSemaphoreTakeRecursive

semphr. h xSemaphoreTakeRecursive(xSemaphoreHandle xMutex, portTickType xBlockTime)

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();
xBlockTime	The time in ticks to wait for the semaphore to become available. The macro portTICK_RATE_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 xBlockTime.

Returns:

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

Example usage:

```
xSemaphoreHandle 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( xSemaphore, ( portTickType ) 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, ( portTickType ) 10 );
            xSemaphoreTakeRecursive( xMutex, ( portTickType ) 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( xSemaphoreHandle xSemaphore )
```

Macro to release a semaphore. The semaphore must have previously been created with a call to vSemaphoreCreateBinary(), xSemaphoreCreateMutex() or xSemaphoreCreateCounting(). and obtained using sSemaphoreTake().

This macro 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:

```
xSemaphoreHandle xSemaphore = NULL;

void vATask( void * pvParameters )
{
   // Create the semaphore to guard a shared resource.
   vSemaphoreCreateBinary( xSemaphore );

   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, ( portTickType ) 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( xSemaphoreHandle 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 xSemaphoreCreateMutex();

Returns:

pdTRUE if the semaphore was given.

Example usage:

```
xSemaphoreHandle 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, ( portTickType ) 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, ( portTickType ) 10 );
            xSemaphoreTakeRecursive( xMutex, ( portTickType ) 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(
                         xSemaphoreHandle xSemaphore,
                         signed portBASE_TYPE *pxHigherPriorityTaskWoken
                      )
```

Macro to release a semaphore. The semaphore must have previously been created with a call to vSemaphoreCreateBinary() 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.

Returns:

pdTRUE if the semaphore was successfully given, otherwise errQUEUE_FULL.

Example usage:

```
\#define LONG_TIME 0xffff
\#define TICKS_TO_WAIT 10
xSemaphoreHandle xSemaphore = NULL;

// Repetitive task.
void vATask( void * pvParameters )
{
   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 portBASE_TYPE 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 != pdFALSE )
   {
       // We can force a context switch here.  Context switching from an
       // ISR uses port specific syntax.  Check the demo task for your port
       // to find the syntax required.
   }
}
```

vSemaphoreCreateMutex

semphr. h

```
xSemaphoreHandle xSemaphoreCreateMutex( void )
```

Macro that implements 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.

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.

Mutex type semaphores cannot be used from within interrupt service routines.

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 xSemaphoreHandle.

Example usage:

```
xSemaphoreHandle xSemaphore;

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

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

semphr. h

```
xSemaphoreHandle 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.

Mutex type semaphores cannot be used from within interrupt service routines.

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 xSemaphoreHandle.

Example usage:

```
xSemaphoreHandle xSemaphore;

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

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

xSemaphoreCreateCounting

semphr. h

```
 xSemaphoreHandle xSemaphoreCreateCounting( unsigned portBASE_TYPE uxMaxCount, unsigned portBASE_TYPE
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. Null if the semaphore could not be created.

Example usage:

```
 xSemaphoreHandle xSemaphore;

 void vATask( void * pvParameters )
 {
 xSemaphoreHandle xSemaphore = NULL;

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

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

Software Timers

xTimerCreate

```
 xTimerHandle xTimerCreate(  const signed char *pcTimerName,
         portTickType xTimerPeriod,
         unsigned portBASE_TYPE uxAutoReload,
         void * pvTimerID,
         tmrTIMER_CALLBACK 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(), xTimerChangePeriod() 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 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_RATE_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_RATE_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 timer 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.
pxCallbackFunction	The function to call when the timer expires. Callback functions must have the prototype defined by tmrTIMER_CALLBACK, which is "void vCallbackFunction(xTIMER *xTimer);".

Returns:

If the timer is successfully create 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 0 is returned.

Example usage:

```
#define NUM_TIMERS 5

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

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

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

pvTimerGetTimerID

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

Returns the ID assigned to the timer.

IDs are assigned to timers using the pvTimerID parameter of the call to xTimerCreated() that was used to create the timer.

If the same callback function is assigned to multiple timers then the timer ID can be used within the callback function to identify which timer actually expired.

Parameters:

xTimer	The timer being queried.

Returns:

The ID assigned to the timer being queried.

Example usage:

See the xTimerCreate() API function example usage scenario.

xTimerIsTimerActive

```
portBASE_TYPE xTimerIsTimerActive( xTimerHandle 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 on-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( xTimerHandle xTimer )
 {
     if( xTimerIsTimerActive( xTimer ) != pdFALSE ) // or more simply and equivalently "if(
xTimerIsTimerActive( xTimer ) )"
     {
         // xTimer is active, do something.
     }
     else
     {
         // xTimer is not active, do something else.
     }
 }
```

xTimerStart

```
portBASE_TYPE xTimerStart( xTimerHandle xTimer, portTickType 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 though a queue called the timer command queue. The timer command queue is private to the 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 scheduler has been started, but when this is done the timer will not actually start until the scheduler is started, and the timers expiry time will be relative to when the 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 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 xTimerCreate() API function example usage scenario.

xTimerStop

```
portBASE_TYPE xTimerStop( xTimerHandle xTimer, portTickType 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 though a queue called the timer command queue. The timer command queue is private to the 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 The xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod() or xTimerChangePeriodFromISR() 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 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 xTimerCreate() API function example usage scenario.

xTimerChangePeriod

```
portBASE_TYPE xTimerChangePeriod(  xTimerHandle xTimer,
         portTickType xNewPeriod,
         portTickType 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 though a queue called the timer command queue. The timer command queue is private to the 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 the xTimerCreate() 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_RATE_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_RATE_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 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( xTimerHandle xTimer )
 {
     if( xTimerIsTimerActive( xTimer ) != pdFALSE ) // or more simply and equivalently "if(
xTimerIsTimerActive( xTimer ) )"
     {
         // 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_RATE_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

```
portBASE_TYPE xTimerDelete( xTimerHandle xTimer, portTickType 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 though a queue called the timer command queue. The timer command queue is private to the 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 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 xTimerChangePeriod() API function example usage scenario.

xTimerReset

```
portBASE_TYPE xTimerReset( xTimerHandle xTimer, portTickType 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 though a queue called the timer command queue. The timer command queue is private to the 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 the xTimerStart() 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 scheduler has been started, but when this is done the timer will not actually start until the scheduler is started, and the timers expiry time will be relative to when the 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 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 xTimerStart() 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.

xTimerHandle xBacklightTimer = NULL;

// The callback function assigned to the one-shot timer.  In this case the
// parameter is not used.
void vBacklightTimerCallback( xTIMER *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, 100 ) != 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( "BacklightTimer",           // Just a text name, not used by the kernel.
                                ( 5000 / portTICK_RATE_MS), // The timer period in ticks.
```

```
                                     pdFALSE,                    // The timer is a one-shot timer.
                                     0,                          // The id is not used by the callback so
can take any value.
                                     vBacklightTimerCallback     // The callback function that switches
the LCD back-light off.
                                   );

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

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

xTimerStartFromISR

```
 portBASE_TYPE xTimerStartFromISR(  xTimerHandle xTimer,
          portBASE_TYPE *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( xTIMER *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 )
{
portBASE_TYPE 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

```
portBASE_TYPE xTimerStopFromISR(  xTimerHandle xTimer,
         portBASE_TYPE *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 )
{
portBASE_TYPE 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

```
portBASE_TYPE xTimerChangePeriodFromISR( xTimerHandle xTimer,
          portTickType xNewPeriod,
          portBASE_TYPE *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_RATE_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_RATE_MS) provided configTICK_RATE_HZ is less than or equal to 1000.
pxHigherPriorityT	The timer service/daemon task spends most of its time in the Blocked state,

askWoken	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 )
{
portBASE_TYPE 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

```
portBASE_TYPE xTimerResetFromISR(  xTimerHandle xTimer,
         portBASE_TYPE *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( xTIMER *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 )
{
portBASE_TYPE 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.
    }
}
```

Co-routines

xCoRoutineCreate

croutine. h

```
portBASE_TYPE xCoRoutineCreate(
                                crCOROUTINE_CODE pxCoRoutineCode,
                                unsigned portBASE_TYPE uxPriority,
                                unsigned portBASE_TYPE 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( xCoRoutineHandle xHandle, unsigned portBASE_TYPE 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 portTickType 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;
xTaskHandle 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 );
    }
}
```

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:

```
// This idle task hook will schedule a co-routine each time it is called.
// The rest of the idle task will execute between co-routine calls.
void vApplicationIdleHook( void )
{
vCoRoutineSchedule();
}

// Alternatively, if you do not require any other part of the idle task to
// execute, the idle task hook can call vCoRoutineScheduler() within an
// infinite loop.
void vApplicationIdleHook( void )
{
   for( ;; )
   {
       vCoRoutineSchedule();
   }
}
```

crSTART

croutine. h

```
crSTART( xCoRoutineHandle xHandle );
```

This macro MUST always be called at the start of a co-routine function.

Example usage:

```
// Co-routine to be created.
void vACoRoutine( xCoRoutineHandle xHandle, unsigned portBASE_TYPE uxIndex )
{
// Variables in co-routines must be declared static if they must maintain value across a blocking call.
static long ulAVariable;

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

    for( ;; )
    {
        // Co-routine functionality goes here.
    }

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

crEND

croutine. h

```
crEND();
```

This macro MUST always be called at the end of a co-routine function.

Example usage:

```
// Co-routine to be created.
void vACoRoutine( xCoRoutineHandle xHandle, unsigned portBASE_TYPE uxIndex )
{
// Variables in co-routines must be declared static if they must maintain value across a blocking call.
static long ulAVariable;

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

    for( ;; )
    {
         // Co-routine functionality goes here.
    }

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

crDELAY

croutine. h

```
crDELAY( xCoRoutineHandle xHandle, portTickType xTicksToDelay );
```

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_RATE_MS can be used to convert ticks to milliseconds.

Example usage:

```
// Co-routine to be created.
void vACoRoutine( xCoRoutineHandle xHandle, unsigned portBASE_TYPE 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_RATE_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

```
 crQUEUE_SEND(
                  xCoRoutineHandle xHandle,
                  xQueueHandle pxQueue,
                  void *pvItemToQueue,
                  portTickType xTicksToWait,
                  portBASE_TYPE *pxResult
             )
```

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.

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.
pxQueue	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_RATE_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( xCoRoutineHandle xHandle, unsigned portBASE_TYPE uxIndex )
 {
 // Variables in co-routines must be declared static if they must maintain value across a blocking call.
 static portBASE_TYPE xNumberToPost = 0;
 static portBASE_TYPE 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

```
  crQUEUE_RECEIVE(
                     xCoRoutineHandle xHandle,
                     xQueueHandle pxQueue,
                     void *pvBuffer,
                     portTickType xTicksToWait,
                     portBASE_TYPE *pxResult
                 )
```

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.

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.
pxQueue	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_RATE_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( xCoRoutineHandle xHandle, unsigned portBASE_TYPE uxIndex )
{
// Variables in co-routines must be declared static if they must maintain value across a blocking call.
static portBASE_TYPE xResult;
static unsigned portBASE_TYPE 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

```
 crQUEUE_SEND_FROM_ISR(
                          xQueueHandle pxQueue,
                          void *pvItemToQueue,
                          portBASE_TYPE xCoRoutinePreviouslyWoken
                     )
```

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( xCoRoutineHandle xHandle, unsigned portBASE_TYPE uxIndex )
```

```
{
char cRxedChar;
portBASE_TYPE 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;
portBASE_TYPE 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

```
 crQUEUE_RECEIVE_FROM_ISR(
                            xQueueHandle pxQueue,
                            void *pvBuffer,
                            portBASE_TYPE * 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( xCoRoutineHandle xHandle, unsigned portBASE_TYPE uxIndex )
{
// cChar holds its value while this co-routine is blocked and must therefore
// be declared static.
static char cCharToTx = 'a';
portBASE_TYPE 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;
portBASE_TYPE 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( pxQueue, &cCharToTx, &xCRWokenByPost ) )
  {
   SEND_CHARACTER( cCharToTx );
  }
    }
}
```

Data Structure Documentation

corCoRoutineControlBlock Struct Reference

Data Fields

- crCOROUTINE_CODE **pxCoRoutineFunction**
- xListItem **xGenericListItem**
- xListItem **xEventListItem**
- unsigned portBASE_TYPE **uxPriority**
- unsigned portBASE_TYPE **uxIndex**
- unsigned short **uxState**

The documentation for this struct was generated from the following file:

- include/croutine.h

QueueDefinition Struct Reference

Data Fields

- signed char * **pcHead**
- signed char * **pcTail**
- signed char * **pcWriteTo**
- signed char * **pcReadFrom**
- xList **xTasksWaitingToSend**
- xList **xTasksWaitingToReceive**
- volatile unsigned portBASE_TYPE **uxMessagesWaiting**
- unsigned portBASE_TYPE **uxLength**
- unsigned portBASE_TYPE **uxItemSize**
- signed portBASE_TYPE **xRxLock**
- signed portBASE_TYPE **xTxLock**

The documentation for this struct was generated from the following file:

- queue.c

tskTaskControlBlock Struct Reference

Data Fields

- volatile portSTACK_TYPE * **pxTopOfStack**
- xListItem **xGenericListItem**
- xListItem **xEventListItem**
- unsigned portBASE_TYPE **uxPriority**
- portSTACK_TYPE * **pxStack**
- signed char **pcTaskName** [configMAX_TASK_NAME_LEN]

The documentation for this struct was generated from the following file:

- tasks.c

xLIST Struct Reference

Data Fields

- volatile unsigned portBASE_TYPE **uxNumberOfItems**
- volatile xListItem * **pxIndex**
- volatile xMiniListItem **xListEnd**

The documentation for this struct was generated from the following file:

- include/list.h

xLIST_ITEM Struct Reference

Data Fields

- portTickType **xItemValue**
- struct xLIST_ITEM * **pxNext**
- struct xLIST_ITEM * **pxPrevious**
- void * **pvOwner**
- void * **pvContainer**

The documentation for this struct was generated from the following file:

- include/list.h

xMEMORY_REGION Struct Reference

Data Fields

- void * **pvBaseAddress**
- unsigned long **ulLengthInBytes**
- unsigned long **ulParameters**

The documentation for this struct was generated from the following file:

- include/task.h

xMINI_LIST_ITEM Struct Reference

Data Fields

- portTickType **xItemValue**
- struct xLIST_ITEM * **pxNext**
- struct xLIST_ITEM * **pxPrevious**

The documentation for this struct was generated from the following file:

- include/list.h

xTASK_PARAMTERS Struct Reference

Data Fields

- pdTASK_CODE **pvTaskCode**
- const signed char *const **pcName**
- unsigned short **usStackDepth**
- void * **pvParameters**
- unsigned portBASE_TYPE **uxPriority**
- portSTACK_TYPE * **puxStackBuffer**
- xMemoryRegion **xRegions** [portNUM_CONFIGURABLE_REGIONS]

The documentation for this struct was generated from the following file:

- include/task.h

xTIME_OUT Struct Reference

Data Fields

- portBASE_TYPE **xOverflowCount**
- portTickType **xTimeOnEntering**

The documentation for this struct was generated from the following file:

- include/task.h

Index