FreeRTOS
-
API

1.1 Task Creation 2
1.2 Task Control 4
1.3 Task Utilities 12
1.4 Kernel Control 14
1.4 Queue Management 19
1.5 Semaphores 28

Literatur: http://www.freertos.org/index.html?http://www.freertos.org/a00106.html

1.1 Task Creation

Modules

- xTaskCreate
- vTaskDelete

Detailed Description
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 portCHAR * const pcName,
                             unsigned portSHORT 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.

Parameters:	Comment
pvTaskCode	Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).
pcName	A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by configMAX_TASK_NAME_LEN.
usStackDepth	The size of the task stack specified as the number of variables the stack can hold - not the number of bytes. For example, if the stack is 16 bits wide and usStackDepth is defined as 100, 200 bytes will be allocated for stack storage. The stack depth multiplied by the stack width must not exceed the maximum value that can be contained in a variable of type size_t.
pvParameters	Pointer that will be used as the parameter for the task being created.
uxPriority	The priority at which the task should run.
pvCreatedTask	Used to pass back a handle by which the created task can be referenced.

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

```
Example usage:

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

 // Function that creates a task.
 void vOtherFunction( void )
 {
 static unsigned char ucParameterToPass;
 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 time 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:	Comment
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 );
 }
```

1.2 Task Control

Modules

- vTaskDelay
- vTaskDelayUntil
- uxTaskPriorityGet
- vTaskPrioritySet
- vTaskSuspend
- vTaskResume
- xTaskResumeFromISR
- vTaskSetApplicationTag
- xTaskCallApplicationTaskHook

vTaskDelay

```
task. h
void vTaskDelay( portTickType xTicksToDelay );
```

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

Delay a task for a given number of ticks. The actual time that the task remains blocked depends on the tick rate. The constant portTICK_RATE_MS can be used to calculate real time from the tick rate - with the resolution of one tick period.

vTaskDelay() specifies a time at which the task wishes to unblock relative to the time at which vTaskDelay() is called. For example, specifying a block period of 100 ticks will cause the task to unblock 100 ticks after vTaskDelay() is called. vTaskDelay() does not therefore provide a good method of controlling the frequency of a 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:	Comment
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() specifies a time at which the task wishes to unblock relative to the time at which vTaskDelay() is called, whereas vTaskDelayUntil() specifies an absolute time at which the task wishes to unblock.

vTaskDelay() will cause a task to block for the specified number of ticks from the time vTaskDelay() is called. It is therefore difficult to use vTaskDelay() by itself to generate a fixed execution frequency as the time between a task unblocking following a call to vTaskDelay() and that task next calling vTaskDelay() may not be fixed [the task may take a different path though the code between calls, or may get interrupted or preempted a different number of times each time it executes].

Whereas vTaskDelay() specifies a wake time relative to the time at which the function is called, vTaskDelayUntil() specifies the absolute (exact) time at which it wishes to unblock.

It should be noted that vTaskDelayUntil() will return immediately (without blocking) if it is used to specify a wake time that is already in the past. Therefore a task using vTaskDelayUntil() to execute periodically will have to re-calculate its required wake time if the periodic execution is halted for any reason (for example, the task is temporarily placed into the Suspended state) causing the task to miss one or more periodic executions. This can be detected by checking the variable passed by reference as the pxPreviousWakeTime parameter against the current tick count. This is however not necessary under most usage scenarios.

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

This function must not be called while the scheduler has been suspended by a call to vTaskSuspendAll().

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

Example usage:

```
 // Perform an action every 10 ticks.
 void vTaskFunction( void * pvParameters )
 {
 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_vTaskPriorityGet must be defined as 1 for this function to be available. See the configuration section for more information.

Obtain the priority of any task.

Parameters:	Comment
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:	Comment
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:	Comment
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:	Comment
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.
 }
```

xTaskResumeFromISR

task. h

```
portBASE_TYPE xTaskResumeFromISR( xTaskHandle pxTaskToResume );
```

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

A function to resume a suspended task that can be called from within an ISR.

A task that has been suspended by one of more calls to vTaskSuspend() will be made available for running again by a single call to xTaskResumeFromISR().

xTaskResumeFromISR() should not be used to synchronise a task with an interrupt if there is a chance that the interrupt could arrive prior to the task being suspended - as this can lead to interrupts being missed. Use of a semaphore as a synchronisation mechanism would avoid this eventuality.

Parameters:	Comment
pxTaskToResume	Handle to the task being readied.

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

Example usage:

 xTaskHandle xHandle;

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

     // ... Rest of code.
 }

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

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

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

```
void vAnExampleISR( void )
 {
 portBASE_TYPE xYieldRequired;

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

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

1.3 Task Utilities

xTaskGetCurrentTaskHandle
task.h
```
xTaskHandle xTaskGetCurrentTaskHandle( void );
```

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

Returns:
The handle of the currently running (calling) task.

xTaskGetTickCount
task.h
```
volatile portTickType xTaskGetTickCount( void );
```

Returns:
The count of ticks since vTaskStartScheduler was called.

xTaskGetSchedulerState
task.h
```
portBASE_TYPE xTaskGetSchedulerState( void );
```

Returns:
One of the following constants (defined within task.h):
taskSCHEDULER_NOT_STARTED, taskSCHEDULER_RUNNING,
taskSCHEDULER_SUSPENDED.

uxTaskGetNumberOfTasks
task.h
```
unsigned portBASE_TYPE 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( portCHAR *pcWriteBuffer );
```

configUSE_TRACE_FACILITY, INCLUDE_vTaskDelete and INCLUDE_vTaskSuspend must all 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.

vTaskStartTrace
task.h
```
void vTaskStartTrace( portCHAR * pcBuffer, unsigned portLONG
ulBufferSize );
```

[The function relates to the original trace utility - which is still provided - however users may find the newer Trace Hook Macros easier and more powerful to use.]

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.

ulTaskEndTrace
task.h
```
unsigned portLONG ulTaskEndTrace( void );
```

[The function relates to the original trace utility - which is still provided - however users may find the newer Trace Hook Macros easier and more powerful to use.]

Stops a kernel activity trace. See vTaskStartTrace().

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

vTaskGetRunTimeStats
task.h
```
void vTaskGetRunTimeStats( portCHAR *pcWriteBuffer );
```

See the Run Time Stats page for a full description of this feature.

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.

1.4 Kernel Control

Modules

- vTaskStartScheduler
- vTaskEndScheduler
- vTaskSuspendAll
- xTaskResumeAll

Detailed Description

`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.

The idle task is created automatically when vTaskStartScheduler() is called.

If vTaskStartScheduler() is successful the function will not return until an executing task calls vTaskEndScheduler(). The function might fail and return immediately if there is insufficient RAM available for the idle task to be 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

```
portBASE_TYPE xTaskResumeAll( void );
```

Resumes real time kernel activity following a call to vTaskSuspendAll (). After a call to xTaskSuspendAll () 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.
        xTaskSuspendAll ();
        // 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 ();
        }
    }
}
```

1.4 Queue Management

Modules

- xQueueCreate
- xQueueSend
- xQueueReceive
- xQueuePeek
- xQueueSendFromISR
- xQueueSendToBackFromISR
- xQueueSendToFrontFromISR
- xQueueReceiveFromISR
- vQueueAddToRegistry
- vQueueUnregisterQueue

Detailed Description

---

uxQueueMessagesWaiting
queue.h

```
unsigned portBASE_TYPE uxQueueMessagesWaiting( 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.

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
{
   portCHAR ucMessageID;
   portCHAR ucData[ 20 ];
};
void vATask( void *pvParameters )
{
xQueueHandle xQueue1, xQueue2;
   // Create a queue capable of containing 10 unsigned long values.
   xQueue1 = xQueueCreate( 10, sizeof( unsigned portLONG ) );
   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.
}
```

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.

xQueueSend() is part of the fully featured intertask communications API. xQueueAltSend() is the alternative API equivalent. Both versions require the same parameters and return the same values.

Parameters:

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

Returns:

pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

Example usage:

```
 struct AMessage
 {
    portCHAR ucMessageID;
    portCHAR ucData[ 20 ];
 } xMessage;
 unsigned portLONG 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 portLONG ) );
    // 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.
 }
```

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.

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

xQueueReceive() is part of the fully featured intertask communications API. xQueueAltReceive() is the alternative API equivalent. Both versions require the same parameters and return the same values.

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. Setting xTicksToWait to 0 will cause the function to return immediately if the queue is empty. The time is defined in tick periods so the constant portTICK_RATE_MS should be used to convert to real time if this is required. If INCLUDE_vTaskSuspend is set to '1' then specifying the block time as portMAX_DELAY will cause the task to block indefinitely (without a timeout).

Returns:

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

Example usage:

```
struct AMessage
{
   portCHAR ucMessageID;
   portCHAR 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.
}
```

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 into the back of a queue. It is safe to use this function from within an interrupt service routine.

Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR. In most cases it would be preferable to store a pointer to the item being queued.

Parameters:

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

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 )
{
portCHAR 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 ();
    }
}
```

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 )
 {
 portCHAR cValueToPost;
 const portTickType xBlockTime = ( portTickType )0xff;
    // Create a queue capable of containing 10 characters.
    xQueue = xQueueCreate( 10, sizeof( portCHAR ) );
    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;
 portCHAR cRxedChar;
    while( xQueueReceiveFromISR( xQueue, ( void * ) &cRxedChar,
&xTaskWokenByReceive) )
    {
        // A character was received.  Output the character now.
        vOutputCharacter( cRxedChar );
        // If removing the character from the queue woke the task that was
        // posting onto the queue xTaskWokenByReceive will have been set to
        // pdTRUE.  No matter how many times this loop iterates only one
        // task will be woken.
    }
    if( xTaskWokenByPost != pdFALSE )
    {
        // We should switch context so the ISR returns to a different task.
        // NOTE:  How this is done depends on the port you are using.  Check
        // the documentation and examples for your port.
        taskYIELD ();
    }
 }
```

1.5 Semaphores
Modules

- vSemaphoreCreateBinary
- vSemaphoreCreateCounting
- xSemaphoreCreateMutex
- xSemaphoreTake
- xSemaphoreGive
- xSemaphoreGiveFromISR

vSemaphoreCreateBinary

semphr. h
```
vSemaphoreCreateBinary( xSemaphoreHandle xSemaphore )
```

Macro that creates a semaphore by using the existing queue mechanism. The queue length is 1 as this is a binary semaphore. The data size is 0 as we don't want to actually store any data - we just want to know if the queue is empty or full.

Binary semaphores and mutexes are very similar but have some subtle differences: Mutexes include a priority inheritance mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronisation (between tasks or between tasks and an interrupt), and mutexes the better choice for implementing simple mutual exclusion.

A binary semaphore need not be given back once obtained, so task synchronisation can be implemented by one task/interrupt continuously 'giving' the semaphore while another continuously 'takes' the semaphore. This is demonstrated by the sample code on the xSemaphoreGiveFromISR() documentation page.

The priority of a task that 'takes' a mutex can potentially be raised if another task of higher priority attempts to obtain the same mutex. The task that owns the mutex 'inherits' the priority of the task attempting to 'take' the same mutex. This means the mutex must always be 'given' back - otherwise the higher priority task will never be able to obtain the mutex, and the lower priority task will never 'disinherit' the priority. An example of a mutex being used to implement mutual exclusion is provided on the xSemaphoreTake() documentation page.

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.
   }
}
```

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. Of type xSemaphoreHandle. NULL if the semaphore could not be created.

Example usage:

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

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

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

xSemaphoreCreateMutex

Only available from FreeRTOS.org V4.5.0 onwards.

semphr. h

```
xSemaphoreHandle xSemaphoreCreateMutex( void )
```

Macro that creates a mutex semaphore by using the existing queue mechanism.

Mutexes created using this macro can be accessed using the xSemaphoreTake() and xSemaphoreGive() macros. The xSemaphoreTakeRecursive() and xSemaphoreGiveRecursive() macros should not be used.

Mutexes and binary semaphores are very similar but have some subtle differences: Mutexes include a priority inheritance mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronisation (between tasks or between tasks and an interrupt), and mutexes the better choice for implementing simple mutual exclusion.

The priority of a task that 'takes' a mutex can potentially be raised if another task of higher priority attempts to obtain the same mutex. The task that owns the mutex 'inherits' the priority of the task attempting to 'take' the same mutex. This means the mutex must always be 'given' back - otherwise the higher priority task will never be able to obtain the mutex, and the lower priority task will never 'disinherit' the priority. An example of a mutex being used to implement mutual exclusion is provided on the xSemaphoreTake() documentation page.

A binary semaphore need not be given back once obtained, so task synchronisation can be implemented by one task/interrupt continuously 'giving' the semaphore while another continuously 'takes' the semaphore. This is demonstrated by the sample code on the xSemaphoreGiveFromISR() documentation page.

Both mutex and binary semaphores are assigned to variables of type xSemaphoreHandle and can be used in any API function that takes a parameter of this type.

Return:

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

Example usage:

```
xSemaphoreHandle xSemaphore;
void vATask( void * pvParameters )
{
   // Mutex semaphores cannot be used before a call to
   // xSemaphoreCreateMutex().  The created mutex is returned.
   xSemaphore = xSemaphoreCreateMutex();
   if( xSemaphore != NULL )
   {
       // The semaphore was created successfully.
       // The semaphore can now be used.
   }
}
```

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().

This macro must not be called from an ISR. xQueueReceiveFromISR() can be used to take a semaphore from within an interrupt if required, although this would not be a normal operation. Semaphores use queues as their underlying mechanism, so functions are to some extent interoperable.

xSemaphoreTake() is part of the fully featured intertask communications API. xSemaphoreAltTake() is the alternative API equivalent. Both versions require the same parameters and return the same values.

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.

If INCLUDE_vTaskSuspend is set to '1' then specifying the block time as portMAX_DELAY will cause the task to block indefinitely (without a timeout).

Returns:

pdTRUE if the semaphore was obtained. pdFALSE if 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.  As we are using
   // the semaphore for mutual exclusion we create a mutex semaphore
   // rather than a binary semaphore.
   xSemaphore = xSemaphoreCreateMutex();
}
// A task that uses the semaphore.
void vAnotherTask( void * pvParameters )
{
   // ... Do other things.
   if( xSemaphore != NULL )
   {
       // See if we can obtain the semaphore. If the semaphore is not available
       // wait 10 ticks to see if it becomes free.
       if( xSemaphoreTake( xSemaphore, ( 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.
        }
    }
}
```

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 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().

xSemaphoreGive() is part of the fully featured intertask communications API. xSemaphoreAltGive() is the alternative API equivalent. Both versions require the same parameters and return the same values.

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.  As we are using
   // the semaphore for mutual exclusion we create a mutex semaphore
   // rather than a binary semaphore.
   xSemaphore = xSemaphoreCreateMutex();

   if( xSemaphore != NULL )
   {
       if( xSemaphoreGive( xSemaphore ) != pdTRUE )
       {
           // We would expect this call to fail because we cannot give
           // a semaphore without first "taking" it!
       }

       // Obtain the semaphore - don't block if the semaphore is not
       // immediately available.
       if( xSemaphoreTake( xSemaphore, ( 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.
            }
        }
    }
}
```

xSemaphoreGiveFromISR

semphr. h

```
xSemaphoreGiveFromISR(
                          xSemaphoreHandle xSemaphore,
                          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 semaphoree 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 )
{
    /* We are using the semaphore for synchronisation so we create a binary
    semaphore rather than a mutex.  We must make sure that the interrupt
    does not attempt to use the semaphore before it is created! */
    vSemaphoreCreateBinary( xSemaphore );

    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 portCHAR ucLocalTickCount = 0;
static 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 was set to true you
    we should yield.  The actual macro used here is
    port specific. */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```