Home
FreeRTOS Books and Manuals
⊟ FreeRTOS
⊞ About FreeRTOS
⊞ Features / Getting Started...
⊞ More Advanced...
⊞ Demo Projects
⊞ Supported Devices & Demos
⊟ API Reference
PDF Reference Manual
⊞ Task Creation
⊞ Task Control
⊞ Task Utilities
⊟ RTOS Kernel Control
taskYIELD()
taskENTER_CRITICAL()
taskEXIT_CRITICAL()
taskDISABLE_INTERRUPTS()
taskENABLE_INTERRUPTS()
vTaskStartScheduler()
vTaskEndScheduler()
vTaskSuspendAll()
xTaskResumeAll()
vTaskStepTick()
⊞ Direct To Task Notifications
⊞ FreeRTOS-MPU Specific
⊞ Queues
⊞ Queue Sets
⊞ Semaphore / Mutexes
⊞ Software Timers
⊞ Event Groups (or 'flags')
⊞ Co-routines
⊞ Contact, Support, Advertising
⊞ FreeRTOS Interactive!

Quick Start Guide
⇓ Download Source ⇓

vTaskStartScheduler

[RTOS Kernel Control]

task. h

```
void vTaskStartScheduler( void );
```

Starts the RTOS scheduler. After calling the RTOS kernel has control over which tasks are executed and when.

The idle task and optionally the timer daemon task are created automatically when the RTOS scheduler is started.

vTaskStartScheduler() will only return if there is insufficient RTOS heap available to create the idle or timer daemon tasks.

All the RTOS demo application projects contain examples of using vTaskStartScheduler(), normally in the main() function within main.c.

Example usage:

```
void vAFunction( void )
{
    // Tasks can be created before or after starting the RTOS
    scheduler
    xTaskCreate( vTaskCode,
                 "NAME",
                 STACK_SIZE,
                 NULL,
                 tskIDLE_PRIORITY,
                 NULL );

    // Start the real time scheduler.
    vTaskStartScheduler();

    // Will not get here unless there is insufficient RAM.
}
```

Home
FreeRTOS Books and Manuals
FreeRTOS
- About FreeRTOS
- Features / Getting Started...
- More Advanced...
- Demo Projects
- Supported Devices & Demos
- API Reference
  - PDF Reference Manual
  - Task Creation
    - TaskHandle_t (type)
    - xTaskCreate()
    - vTaskDelete()
  - Task Control
  - Task Utilities
  - RTOS Kernel Control
  - Direct To Task Notifications
  - FreeRTOS-MPU Specific
  - Queues
  - Queue Sets
  - Semaphore / Mutexes
  - Software Timers
  - Event Groups (or 'flags')
  - Co-routines
  - Contact, Support, Advertising
FreeRTOS Interactive!

xTaskCreate

[Task Creation]

task. h

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

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

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

Parameters:

pvTaskCode Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).

pcName A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by configMAX_TASK_NAME_LEN.

usStackDepth The size of the task stack specified as the number of variables the stack can hold - not the number of bytes. For example, if the stack is 16 bits wide and usStackDepth is defined as 100, 200 bytes will be allocated for stack storage. The stack depth multiplied by the stack width must not exceed the maximum value that can be contained in a variable of type size_t.

See the FAQ How big should the stack be?

pvParameters Pointer that will be used as the parameter for the task being created.

uxPriority The priority at which the task should run.

Systems that include MPU support can optionally create tasks in a privileged (system) mode by setting bit portPRIVILEGE_BIT of the priority parameter. For example, to create a privileged task at priority 2 the uxPriority parameter should be set to (2 | portPRIVILEGE_BIT).

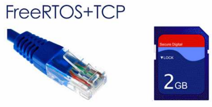
pvCreatedTask Used to pass back a handle by which the created task can be referenced.

Returns:

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

Example usage:

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

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

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

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

vTaskDelete

[Task Creation]

task. h

```
void vTaskDelete( TaskHandle_t xTask );
```

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

Remove a task from the RTOS kernels management. The task being deleted will be removed from all ready, blocked, suspended and event lists.

NOTE: The idle task is responsible for freeing the RTOS kernel allocated memory from tasks that have been deleted. It is therefore important that the idle task is not starved of microcontroller processing time if your application makes any calls to vTaskDelete (). Memory allocated by the task code is not automatically freed, and should be freed before the task is deleted.

See the demo application file death. c for sample code that utilises vTaskDelete ().

Parameters:

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

Example usage:

```
void vOtherFunction( void )
{
TaskHandle_t xHandle = NULL;

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

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

Home
FreeRTOS Books and Manuals
FreeRTOS
- About FreeRTOS
- Features / Getting Started...
- More Advanced...
- Demo Projects
- Supported Devices & Demos
- API Reference
  - PDF Reference Manual
  - Task Creation
  - Task Control
    - vTaskDelay()
    - vTaskDelayUntil()
    - uxTaskPriorityGet()
    - vTaskPrioritySet()
    - vTaskSuspend()
    - vTaskResume()
    - xTaskResumeFromISR()
  - Task Utilities
  - RTOS Kernel Control
  - Direct To Task Notifications
  - FreeRTOS-MPU Specific
  - Queues
  - Queue Sets
  - Semaphore / Mutexes
  - Software Timers
  - Event Groups (or 'flags')
  - Co-routines
  - Contact, Support, Advertising
FreeRTOS Interactive!

vTaskDelay

[Task Control]

task. h

```
void vTaskDelay( const TickType_t xTicksToDelay );
```

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

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

vTaskDelay() specifies a time at which the task wishes to unblock **relative to** the time at which vTaskDelay() is called. For example, specifying a block period of 100 ticks will cause the task to unblock 100 ticks after vTaskDelay() is called. vTaskDelay() does not therefore provide a good method of controlling the frequency of a periodic task as the path taken through the code, as well as other task and interrupt activity, will effect the frequency at which vTaskDelay() gets called and therefore the time at which the task next executes. See vTaskDelayUntil() for an alternative API function designed to facilitate fixed frequency execution. It does this by specifying an absolute time (rather than a relative time) at which the calling task should unblock.

Parameters:

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

Example usage:

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

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

[Back to the top] [About FreeRTOS] [Sitemap] [Report an error on this page]

Quick Start Guide
⇓ Download Source ⇓

Home
FreeRTOS Books and Manuals
⊟ FreeRTOS
⊞ About FreeRTOS
⊞ Features / Getting Started...
⊞ More Advanced...
⊞ Demo Projects
⊞ Supported Devices & Demos
⊟ API Reference
PDF Reference Manual
⊞ Task Creation
⊞ Task Control
⊞ Task Utilities
⊞ RTOS Kernel Control
⊞ Direct To Task Notifications
⊞ FreeRTOS-MPU Specific
⊞ Queues
⊞ Queue Sets
⊞ Semaphore / Mutexes
⊞ Software Timers
⊟ Event Groups (or 'flags')
xEventGroupCreate()
vEventGroupDelete()
xEventGroupWaitBits()
xEventGroupSetBits()
xEventGroupSetBitsFromISR()
xEventGroupClearBits()
xEventGroupClearBitsFromISR()
xEventGroupGetBits()
xEventGroupGetBitsFromISR()
xEventGroupSync()
⊞ Co-routines
⊞ Contact, Support, Advertising
⊞ FreeRTOS Interactive!

xEventGroupCreate()

[Event Group API]

Available From FreeRTOS V8.0.0

event_groups.h

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

Create a new RTOS event group. This function cannot be called from an interrupt.

Event groups are stored in variables of type EventGroupHandle_t. The number of bits (or flags) implemented within an event group is 8 if configUSE_16_BIT_TICKS is set to 1, or 24 if configUSE_16_BIT_TICKS is set to 0. The dependency on configUSE_16_BIT_TICKS results from the data type used for thread local storage in the internal implementation of RTOS tasks.

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

Parameters:

None

Returns:

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

Example usage:

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

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

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

[Back to the top] [About FreeRTOS] [Sitemap] [Report an error on this page]

Home
FreeRTOS Books and Manuals
FreeRTOS
About FreeRTOS
Features / Getting Started...
More Advanced...
Demo Projects
Supported Devices & Demos
API Reference
PDF Reference Manual
Task Creation
Task Control
Task Utilities
RTOS Kernel Control
Direct To Task Notifications
FreeRTOS-MPU Specific
Queues
Queue Sets
Semaphore / Mutexes
Software Timers
Event Groups (or 'flags')
xEventGroupCreate()
vEventGroupDelete()
xEventGroupWaitBits()
xEventGroupSetBits()
xEventGroupSetBitsFromISR()
xEventGroupClearBits()
xEventGroupClearBitsFromISR()
xEventGroupGetBits()
xEventGroupGetBitsFromISR()
xEventGroupSync()
Co-routines
Contact, Support, Advertising
FreeRTOS Interactive!

xEventGroupSetBits()

[Event Group API]

Available From FreeRTOS V8.0.0

event_groups.h

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

Set bits (flags) within an RTOS event group. This function cannot be called from an interrupt. xEventGroupSetBitsFromISR() is a version that can be called from an interrupt.

Setting bits in an event group will automatically unblock tasks that are blocked waiting for the bits.

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

Parameters:

xEventGroup	The event group in which the bits are to be set. The event group must have previously been created using a call to xEventGroupCreate().
uxBitsToSet	A bitwise value that indicates the bit or bits to set in the event group. For example, set uxBitsToSet to 0x08 to set only bit 3. Set uxBitsToSet to 0x09 to set bit 3 and bit 0.

Returns:

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

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

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

Example usage:

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

void aFunction( EventGroupHandle_t xEventGroup )
{
EventBits_t uxBits;

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

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

xEventGroupSetBitsFromISR()

[Event Group API]

Available From FreeRTOS V8.0.0

event_groups.h

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

Set bits (flags) within an RTOS event group. A version of xEventGroupSetBits() that can be called from an interrupt service routine (ISR).

Setting bits in an event group will automatically unblock tasks that are blocked waiting for the bits.

Setting bits in an event group is not a deterministic operation because there are an unknown number of tasks that may be waiting for the bit or bits being set. FreeRTOS does not allow non-deterministic operations to be performed in interrupts or from critical sections. Therefore xEventGroupSetBitFromISR() sends a message to the RTOS daemon task to have the set operation performed in the context of the daemon task - where a scheduler lock is used in place of a critical section.

NOTE: As mentioned in the paragraph above, setting bits from an ISR will defer the set operation to the RTOS daemon task (also known as the timer service task). The RTOS daemon task is scheduled according to its priority, just like any other RTOS task. Therefore, if it is essential the set operation completes immediately (before a task created by the application executes) then the priority of the RTOS daemon task must be higher than the priority of any application task that uses the event group. The priority of the RTOS daemon task is set by the configTIMER_TASK_PRIORITY definition in FreeRTOSConfig.h.

INCLUDE_xEventGroupSetBitFromISR, configUSE_TIMERS and INCLUDE_xTimerPendFunctionCall must all be set to 1 in FreeRTOSConfig.h for the xEventGroupSetBitsFromISR() function to be available.

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

Parameters:

xEventGroup	The event group in which the bits are to be set. The event group must have previously been created using a call to xEventGroupCreate().
uxBitsToSet	A bitwise value that indicates the bit or bits to set. For example, set uxBitsToSet to 0x08 to set only bit 3. Set uxBitsToSet to 0x09 to set bit 3 and bit 0.
pxHigherPriorityTaskWoken	As mentioned above, calling this function will result in a message being sent to the RTOS daemon task. If the priority of the daemon task is higher than the priority of the currently running task (the task the interrupt interrupted) then *pxHigherPriorityTaskWoken will be set to pdTRUE by xEventGroupSetBitsFromISR(), indicating that a context switch should be requested before the interrupt exits. For that reason *pxHigherPriorityTaskWoken must be initialised to pdFALSE. See the example code below.

Returns:

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

Example usage:

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

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

void anInterruptHandler( void )
{
BaseType_t xHigherPriorityTaskWoken, xResult;

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

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

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

[Back to the top]   [About FreeRTOS]   [Sitemap]   [Report an error on this page]

Home
FreeRTOS Books and Manuals
FreeRTOS
About FreeRTOS
Features / Getting Started...
More Advanced...
Demo Projects
Supported Devices & Demos
API Reference
PDF Reference Manual
Task Creation
Task Control
Task Utilities
RTOS Kernel Control
Direct To Task Notifications
FreeRTOS-MPU Specific
Queues
Queue Sets
Semaphore / Mutexes
Software Timers
Event Groups (or 'flags')
xEventGroupCreate()
vEventGroupDelete()
xEventGroupWaitBits()
xEventGroupSetBits()
xEventGroupSetBitsFromISR()
xEventGroupClearBits()
xEventGroupClearBitsFromISR()
xEventGroupGetBits()
xEventGroupGetBitsFromISR()
xEventGroupSync()
Co-routines
Contact, Support, Advertising
FreeRTOS Interactive!

Quick Start Guide
⇓ Download Source ⇓

FreeRTOS+ Lab Projects
FreeRTOS+TCP:
Thread safe TCP/IP stack
FreeRTOS+FAT:
Thread aware file system

FreeRTOS+ Ecosystem
Internet of Things:
Innovative complete solution
Fail Safe File System:
Ensures data integrity
InterNiche TCP/IP:
Low cost pre-ported libraries
FreeRTOS BSPs:
3rd party driver packages
FAT SL File System:
Super lean FAT FS
UDP/IP:
Thread aware UDP stack
Trace & Visualisation:
Tracealyzer for FreeRTOS
CLI:
Command line interface
WolfSSL SSL / TLS:
Networking security protocols
Safety:
TUV certified RTOS
RTOS Training:
Delivered online or on-site
IO:
read(), write(), ioctl() interface

Google Custom Search | Search

xEventGroupWaitBits()

[Event Group API]

Available From FreeRTOS V8.0.0

event_groups.h

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

Read bits within an RTOS event group, optionally entering the Blocked state (with a timeout) to wait for a bit or group of bits to become set.

This function cannot be called from an interrupt.

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

Parameters:

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

uxBitsToWaitFor A bitwise value that indicates the bit or bits to test inside the event group. For example, to wait for bit 0 and/or bit 2 set uxBitsToWaitFor to 0x05. To wait for bits 0 and/or bit 1 and/or bit 2 set uxBitsToWaitFor to 0x07. Etc.

uxBitsToWaitFor must **not** be set to 0.

xClearOnExit If xClearOnExit is set to pdTRUE then any bits set in the value passed as the uxBitsToWaitFor parameter will be cleared in the event group before xEventGroupWaitBits() returns if xEventGroupWaitBits() returns for any reason other than a timeout. The timeout value is set by the xTicksToWait parameter.

If xClearOnExit is set to pdFALSE then the bits set in the event group are not altered when the call to xEventGroupWaitBits() returns.

xWaitForAllBits xWaitForAllBits is used to create either a logical AND test (where all bits must be set) or a logical OR test (where one or more bits must be set) as follows:

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

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

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

Returns:

The value of the event group at the time either the event bits being waited for became set, or the block time expired. The current value of the event bits in an event group will be different to the returned value if a higher priority task or interrupt changed the value of an event bit between the calling task leaving the Blocked state and exiting the xEventGroupWaitBits() function.

Test the return value to know which bits were set. If xEventGroupWaitBits() returned because its timeout expired then not all the bits being waited for will be set. If xEventGroupWaitBits() returned because the bits it was waiting for were set then the returned value is the event group value before any bits were automatically cleared because the xClearOnExit parameter was set to pdTRUE.

Example usage:

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

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

```
const TickType_t xTicksToWait = 100 / portTICK_PERIOD_MS;

  /* Wait a maximum of 100ms for either bit 0 or bit 4 to be set within
  the event group.  Clear the bits before exiting. */
  uxBits = xEventGroupWaitBits(
            xEventGroup,   /* The event group being tested. */
            BIT_0 | BIT_4, /* The bits within the event group to wait for. */
            pdTRUE,        /* BIT_0 & BIT_4 should be cleared before returning. */
            pdFALSE,       /* Don't wait for both bits, either bit will do. */
            xTicksToWait );/* Wait a maximum of 100ms for either bit to be set. */

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

[Back to the top]   [About FreeRTOS]   [Sitemap]   [Report an error on this page]

Home
FreeRTOS Books and Manuals
FreeRTOS
About FreeRTOS
Features / Getting Started...
More Advanced...
Demo Projects
Supported Devices & Demos
API Reference
PDF Reference Manual
Task Creation
Task Control
Task Utilities
RTOS Kernel Control
Direct To Task Notifications
FreeRTOS-MPU Specific
Queues
uxQueueMessagesWaiting()
uxQueueMessagesWaitingFromISR()
uxQueueSpacesAvailable()
xQueueCreate()
vQueueDelete()
xQueueReset()
xQueueSend()
xQueueSendToBack()
xQueueSendToFront()
xQueueReceive()
xQueueOverwrite()
xQueueOverwriteFromISR()
xQueuePeek()
xQueuePeekFromISR()
xQueueSendFromISR()
xQueueSendToBackFromISR()
xQueueSendToFrontFromISR()
xQueueReceiveFromISR()
vQueueAddToRegistry()
vQueueUnregisterQueue()
xQueueIsQueueFullFromISR()
xQueueIsQueueEmptyFromISR()
Queue Sets
Semaphore / Mutexes
Software Timers
Event Groups (or 'flags')
Co-routines
Contact, Support, Advertising
FreeRTOS Interactive!

xQueueCreate

[Queue Management]

queue. h

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

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

Parameters:

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

Returns:

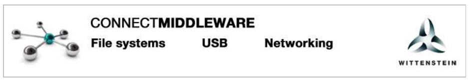
If the queue is successfully create then a handle to the newly created queue is returned. If the queue cannot be created then 0 is returned.

Example usage:

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

void vATask( void *pvParameters )
{
QueueHandle_t xQueue1, xQueue2;

    // Create a queue capable of containing 10 unsigned long values.
    xQueue1 = xQueueCreate( 10, sizeof( unsigned long ) );
    if( xQueue1 == 0 )
    {
        // Queue was not created and must not be used.
    }

    // Create a queue capable of containing 10 pointers to AMessage structures.
    // These should be passed by pointer as they contain a lot of data.
    xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );
    if( xQueue2 == 0 )
    {
        // Queue was not created and must not be used.
    }

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

[Back to the top] [About FreeRTOS] [Sitemap] [Report an error on this page]

Quality RTOS & Embedded Software
About Contact Support FAQ



Quick Start | Supported MCUs | Books & Kits | Trace Tools | Ecosystem | TCP & FAT | Training | Email List

Home
FreeRTOS Books and Manuals
FreeRTOS
About FreeRTOS
Features / Getting Started...
More Advanced...
Demo Projects
Supported Devices & Demos
API Reference
PDF Reference Manual
Task Creation
Task Control
Task Utilities
RTOS Kernel Control
Direct To Task Notifications
FreeRTOS-MPU Specific
Queues
uxQueueMessagesWaiting()
uxQueueMessagesWaitingFromISR()
uxQueueSpacesAvailable()
xQueueCreate()
vQueueDelete()
xQueueReset()
xQueueSend()
xQueueSendToBack()
xQueueSendToFront()
xQueueReceive()
xQueueOverwrite()
xQueueOverwriteFromISR()
xQueuePeek()
xQueuePeekFromISR()
xQueueSendFromISR()
xQueueSendToBackFromISR()
xQueueSendToFrontFromISR()
xQueueReceiveFromISR()
vQueueAddToRegistry()
vQueueUnregisterQueue()
xQueueIsQueueFullFromISR()
xQueueIsQueueEmptyFromISR()
Queue Sets
Semaphore / Mutexes
Software Timers
Event Groups (or 'flags')
Co-routines
Contact, Support, Advertising
FreeRTOS Interactive!

Quick Start Guide
⇓ Download Source ⇓

FreeRTOS+ Lab Projects
FreeRTOS+TCP: Thread safe TCP/IP stack
FreeRTOS+FAT: Thread aware file system

FreeRTOS+ Ecosystem
Internet of Things: Innovative complete solution
Fail Safe File System: Ensures data integrity
InterNiche TCP/IP: Low cost pre-ported libraries
FreeRTOS BSPs: 3rd party driver packages
FAT SL File System: Super lean FAT FS
UDP/IP: Thread aware UDP stack
Trace & Visualisation: Tracealyzer for FreeRTOS
CLI: Command line interface
WolfSSL SSL / TLS: Networking security protocols
Safety: TUV certified RTOS
RTOS Training: Delivered online or on-site
IO: read(), write(), ioctl() interface

xQueueSend

[Queue Management]

queue.h

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

This is a macro that calls xQueueGenericSend(). It is included for backward compatibility with versions of FreeRTOS that did not include the xQueueSendToFront() and xQueueSendToBack() macros. It is equivalent to xQueueSendToBack().

Post an item on a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See xQueueSendFromISR() for an alternative which may be used in an ISR.

Parameters:

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

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

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

 unsigned long ulVar = 10UL;

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

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

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


    /* ... */


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

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

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

Home
FreeRTOS Books and Manuals
FreeRTOS
About FreeRTOS
Features / Getting Started...
More Advanced...
Demo Projects
Supported Devices & Demos
API Reference
PDF Reference Manual
Task Creation
Task Control
Task Utilities
RTOS Kernel Control
Direct To Task Notifications
FreeRTOS-MPU Specific
Queues
uxQueueMessagesWaiting()
uxQueueMessagesWaitingFromISR()
uxQueueSpacesAvailable()
xQueueCreate()
vQueueDelete()
xQueueReset()
xQueueSend()
xQueueSendToBack()
xQueueSendToFront()
xQueueReceive()
xQueueOverwrite()
xQueueOverwriteFromISR()
xQueuePeek()
xQueuePeekFromISR()
xQueueSendFromISR()
xQueueSendToBackFromISR()
xQueueSendToFrontFromISR()
xQueueReceiveFromISR()
vQueueAddToRegistry()
vQueueUnregisterQueue()
xQueueIsQueueFullFromISR()
xQueueIsQueueEmptyFromISR()
Queue Sets
Semaphore / Mutexes
Software Timers
Event Groups (or 'flags')
Co-routines
Contact, Support, Advertising
FreeRTOS Interactive!

xQueueSendFromISR

[Queue Management]

queue.h

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

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

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

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

Parameters:

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

Returns:

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

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

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

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

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

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

    } while( portINPUT_BYTE( BUFFER_COUNT ) );

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

Home
FreeRTOS Books and Manuals
FreeRTOS
About FreeRTOS
Features / Getting Started...
More Advanced...
Demo Projects
Supported Devices & Demos
API Reference
PDF Reference Manual
Task Creation
Task Control
Task Utilities
RTOS Kernel Control
Direct To Task Notifications
FreeRTOS-MPU Specific
Queues
uxQueueMessagesWaiting()
uxQueueMessagesWaitingFromISR()
uxQueueSpacesAvailable()
xQueueCreate()
vQueueDelete()
xQueueReset()
xQueueSend()
xQueueSendToBack()
xQueueSendToFront()
xQueueReceive()
xQueueOverwrite()
xQueueOverwriteFromISR()
xQueuePeek()
xQueuePeekFromISR()
xQueueSendFromISR()
xQueueSendToBackFromISR()
xQueueSendToFrontFromISR()
xQueueReceiveFromISR()
vQueueAddToRegistry()
vQueueUnregisterQueue()
xQueueIsQueueFullFromISR()
xQueueIsQueueEmptyFromISR()
Queue Sets
Semaphore / Mutexes
Software Timers
Event Groups (or 'flags')
Co-routines
Contact, Support, Advertising
FreeRTOS Interactive!

xQueueReceive

[Queue Management]

queue. h

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

This is a macro that calls the xQueueGenericReceive() function.

Receive an item from a queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

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

Parameters:

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

Returns:

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

Example usage:

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

QueueHandle_t xQueue;

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

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

    // ...

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

        // ... Rest of task code.
}

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

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

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

Home
FreeRTOS Books and Manuals
FreeRTOS
About FreeRTOS
Features / Getting Started...
More Advanced...
Demo Projects
Supported Devices & Demos
API Reference
PDF Reference Manual
Task Creation
Task Control
Task Utilities
RTOS Kernel Control
Direct To Task Notifications
FreeRTOS-MPU Specific
Queues
Queue Sets
Semaphore / Mutexes
Software Timers
xTimerCreate()
xTimerIsTimerActive()
xTimerStart()
xTimerStop()
xTimerChangePeriod()
xTimerDelete()
xTimerReset()
xTimerStartFromISR()
xTimerStopFromISR()
xTimerChangePeriodFromISR()
xTimerResetFromISR()
pvTimerGetTimerID()
vTimerSetTimerID()
xTimerGetTimerDaemonTaskHandle()
xTimerPendFunctionCall()
xTimerPendFunctionCallFromISR()
Event Groups (or 'flags')
Co-routines
Contact, Support, Advertising
FreeRTOS Interactive!

xTimerCreate

[Timer API]

timers.h

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

Creates a new software timer instance. This allocates the storage required by the new timer, initialises the new timers internal state, and returns a handle by which the new timer can be referenced.

Timers are created in the dormant state. The xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), 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 RTOS kernel itself only ever references a timer by its handle, and never by its name.
xTimerPeriod	The timer period. The time is defined in tick periods so the constant portTICK_PERIOD_MS can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then xTimerPeriod should be set to 100. Alternatively, if the timer must expire after 500ms, then xPeriod can be set to (500 / portTICK_PERIOD_MS) provided configTICK_RATE_HZ is less than or equal to 1000.
uxAutoReload	If uxAutoReload is set to pdTRUE, then the timer will expire repeatedly with a frequency set by the xTimerPeriod parameter. If uxAutoReload is set to pdFALSE, then the timer will be a one-shot and enter the dormant state after it expires.
pvTimerID	An identifier that is assigned to the timer being created. Typically this would be used in the timer callback function to identify which timer expired when the same callback function is assigned to more than one timer, or together with the vTimerSetTimerID() and pvTimerGetTimerID() API functions to save a value between calls to the timer's callback function.
pxCallbackFunction	The function to call when the timer expires. Callback functions must have the prototype defined by TimerCallbackFunction_t, which is "void vCallbackFunction(TimerHandle_t xTimer);".

Returns:

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

Example usage:

```
#define NUM_TIMERS 5

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

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

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

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

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

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

    /* If the timer has expired 10 times then stop it from running. */
    if( lExpireCounters[ lArrayIndex ] == xMaxExpiryCountBeforeStopping )
    {
        /* Do not use a block time if calling a timer API function from a
```

```
            timer callback function, as doing so could cause a deadlock! */
            xTimerStop( pxTimer, 0 );
        }
}

void main( void )
{
long x;

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

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

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

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

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


[Back to the top] [About FreeRTOS] [Sitemap] [Report an error on this page]

Quality RTOS & Embedded Software
About Contact Support FAQ

Quick Start | Supported MCUs | Books & Kits | Trace Tools | Ecosystem | TCP & FAT | Training | Email List+

Home
FreeRTOS Books and Manuals
FreeRTOS
About FreeRTOS
Features / Getting Started...
More Advanced...
Demo Projects
Supported Devices & Demos
API Reference
PDF Reference Manual
Task Creation
Task Control
Task Utilities
RTOS Kernel Control
Direct To Task Notifications
FreeRTOS-MPU Specific
Queues
Queue Sets
Semaphore / Mutexes
Software Timers
xTimerCreate()
xTimerIsTimerActive()
xTimerStart()
xTimerStop()
xTimerChangePeriod()
xTimerDelete()
xTimerReset()
xTimerStartFromISR()
xTimerStopFromISR()
xTimerChangePeriodFromISR()
xTimerResetFromISR()
pvTimerGetTimerID()
vTimerSetTimerID()
xTimerGetTimerDaemonTaskHandle()
xTimerPendFunctionCall()
xTimerPendFunctionCallFromISR()
Event Groups (or 'flags')
Co-routines
Contact, Support, Advertising
FreeRTOS Interactive!

Quick Start Guide
⇓ Download Source ⇓

FreeRTOS+ Lab Projects
FreeRTOS+TCP:
Thread safe TCP/IP stack
FreeRTOS+FAT:
Thread aware file system

FreeRTOS+ Ecosystem
Internet of Things:
Innovative complete solution
Fail Safe File System:
Ensures data integrity
InterNiche TCP/IP:
Low cost pre-ported libraries
FreeRTOS BSPs:
3rd party driver packages
FAT SL File System:
Super lean FAT FS
UDP/IP:
Thread aware UDP stack
Trace & Visualisation:
Tracealyzer for FreeRTOS

xTimerStart

[Timer API]

timers.h

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

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task through a queue called the timer command queue. The timer command queue is private to the RTOS kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH configuration constant.

xTimerStart() starts a timer that was previously created using the xTimerCreate() API function. If the timer had already been started and was already in the active state, then xTimerStart() has equivalent functionality to the xTimerReset() API function.

Starting a timer ensures the timer is in the active state. If the timer is not stopped, deleted, or reset in the mean time, the callback function associated with the timer will get called 'n 'ticks after xTimerStart() was called, where 'n' is the timers defined period.

It is valid to call xTimerStart() before the RTOS scheduler has been started, but when this is done the timer will not actually start until the RTOS scheduler is started, and the timers expiry time will be relative to when the RTOS scheduler is started, not relative to when xTimerStart() was called.

The configUSE_TIMERS configuration constant must be set to 1 for xTimerStart() to be available.

Parameters:

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

Returns:

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

Example usage:

See the example on the xTimerCreate() documentation page.

[Back to the top] [About FreeRTOS] [Sitemap] [Report an error on this page]

Home
FreeRTOS Books and Manuals
FreeRTOS
About FreeRTOS
Features / Getting Started...
More Advanced...
Demo Projects
Supported Devices & Demos
API Reference
PDF Reference Manual
Task Creation
Task Control
Task Utilities
RTOS Kernel Control
Direct To Task Notifications
FreeRTOS-MPU Specific
Queues
Queue Sets
Semaphore / Mutexes
xSemaphoreCreateBinary()
vSemaphoreCreateBinary()
xSemaphoreCreateCounting()
xSemaphoreCreateMutex()
xSemaphoreCreateRecursiveMutex()
vSemaphoreDelete()
xSemaphoreGetMutexHolder()
xSemaphoreTake()
xSemaphoreTakeFromISR()
xSemaphoreTakeRecursive()
xSemaphoreGive()
xSemaphoreGiveRecursive()
xSemaphoreGiveFromISR()
Software Timers
Event Groups (or 'flags')
Co-routines
Contact, Support, Advertising
FreeRTOS Interactive!

xSemaphoreCreateMutex

[Semaphores]

Only available from FreeRTOS V4.5.0 onwards.

semphr. h

```
SemaphoreHandle_t xSemaphoreCreateMutex( void )
```

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

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

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

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

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

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

Return:

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

Example usage:

```
SemaphoreHandle_t xSemaphore;

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

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

Home
FreeRTOS Books and Manuals
FreeRTOS
About FreeRTOS
Features / Getting Started...
More Advanced...
Demo Projects
Supported Devices & Demos
API Reference
PDF Reference Manual
Task Creation
Task Control
Task Utilities
RTOS Kernel Control
Direct To Task Notifications
FreeRTOS-MPU Specific
Queues
Queue Sets
Semaphore / Mutexes
xSemaphoreCreateBinary()
vSemaphoreCreateBinary()
xSemaphoreCreateCounting()
xSemaphoreCreateMutex()
xSemaphoreCreateRecursiveMutex()
vSemaphoreDelete()
xSemaphoreGetMutexHolder()
xSemaphoreTake()
xSemaphoreTakeFromISR()
xSemaphoreTakeRecursive()
xSemaphoreGive()
xSemaphoreGiveRecursive()
xSemaphoreGiveFromISR()
Software Timers
Event Groups (or 'flags')
Co-routines
Contact, Support, Advertising
FreeRTOS Interactive!

xSemaphoreGive

[Semaphores]

semphr. h

```
xSemaphoreGive( SemaphoreHandle_t xSemaphore )
```

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

This must not be used from an ISR. See xSemaphoreGiveFromISR() for an alternative which can be used from an ISR.

This macro must also not be used on semaphores created using xSemaphoreCreateRecursiveMutex().

Parameters:

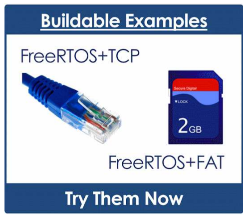
xSemaphore	A handle to the semaphore being released. This is the handle returned when the semaphore was created.

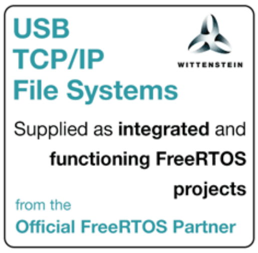
Returns:

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

Example usage:

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

    if( xSemaphore != NULL )
    {
        if( xSemaphoreGive( xSemaphore ) != pdTRUE )
        {
            // We would expect this call to fail because we cannot give
            // a semaphore without first "taking" it!
        }
        // Obtain the semaphore - don't block if the semaphore is not
        // immediately available.
        if( xSemaphoreTake( xSemaphore, ( TickType_t ) 0 ) )
        {
            // We now have the semaphore and can access the shared resource.
            // ...
            // We have finished accessing the shared resource so can free the
            // semaphore.
            if( xSemaphoreGive( xSemaphore ) != pdTRUE )
            {
                // We would not expect this call to fail because we must have
                // obtained the semaphore to get here.
            }
        }
    }
}
```



Quick Start | Supported MCUs | Books & Kits | Trace Tools | Ecosystem | TCP & FAT | Training | Email List+

Home
FreeRTOS Books and Manuals
FreeRTOS
- About FreeRTOS
- Features / Getting Started...
- More Advanced...
- Demo Projects
- Supported Devices & Demos
- API Reference
  - PDF Reference Manual
  - Task Creation
  - Task Control
  - Task Utilities
  - RTOS Kernel Control
  - Direct To Task Notifications
  - FreeRTOS-MPU Specific
  - Queues
  - Queue Sets
  - Semaphore / Mutexes
    - xSemaphoreCreateBinary()
    - vSemaphoreCreateBinary()
    - xSemaphoreCreateCounting()
    - xSemaphoreCreateMutex()
    - xSemaphoreCreateRecursiveMutex()
    - vSemaphoreDelete()
    - xSemaphoreGetMutexHolder()
    - **xSemaphoreTake()**
    - xSemaphoreTakeFromISR()
    - xSemaphoreTakeRecursive()
    - xSemaphoreGive()
    - xSemaphoreGiveRecursive()
    - xSemaphoreGiveFromISR()
  - Software Timers
  - Event Groups (or 'flags')
  - Co-routines
  - Contact, Support, Advertising
- FreeRTOS Interactive!

xSemaphoreTake

[Semaphores]

semphr. h

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

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

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

Parameters:

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

xTicksToWait The time in ticks to wait for the semaphore to become available. The macro portTICK_PERIOD_MS can be used to convert this to a real time. A block time of zero can be used to poll the semaphore.

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

Returns:

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

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;

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

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

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

            /* ... */

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


[Back to the top] [About FreeRTOS] [Sitemap] [Report an error on this page]