# FreeRTOS Kernel

V10.4.3

Generated by Doxygen 1.9.0

# Chapter 1

# LICENSE

# Chapter 2

# Getting started

This repository contains FreeRTOS kernel source/header files and kernel ports only. This repository is referenced as a submodule in `FreeRTOS/FreeRTOS` repository, which contains pre-configured demo application projects under `FreeRTOS/Demo` directory.

The easiest way to use FreeRTOS is to start with one of the pre-configured demo application projects. That way you will have the correct FreeRTOS source files included, and the correct include paths configured. Once a demo application is building and executing you can remove the demo application files, and start to add in your own application source files. See the `FreeRTOS Kernel Quick Start Guide` for detailed instructions and other useful links.

Additionally, for FreeRTOS kernel feature information refer to the `Developer Documentation`, and `API Reference`.

### 2.0.1 Getting help

If you have any questions or need assistance troubleshooting your FreeRTOS project, we have an active community that can help on the `FreeRTOS Community Support Forum`.

## 2.1 Cloning this repository

To clone using HTTPS:

```
git clone https://github.com/FreeRTOS/FreeRTOS-Kernel.git
```

Using SSH:

```
git clone git@github.com:FreeRTOS/FreeRTOS-Kernel.git
```

## 2.2 Repository structure

- The root of this repository contains the three files that are common to every port - list.c, queue.c and tasks.c. The kernel is contained within these three files. croutine.c implements the optional co-routine functionality - which is normally only used on very memory limited systems.
- The `./portable` directory contains the files that are specific to a particular microcontroller and/or compiler. See the readme file in the `./portable` directory for more information.
- The `./include` directory contains the real time kernel header files.

### 2.2.1 Code Formatting

FreeRTOS files are formatted using the "uncrustify" tool. The configuration file used by uncrustify can be found in the `FreeRTOS/FreeRTOS repository`.

### 2.2.2 Spelling

*lexicon.txt* contains words that are not traditionally found in an English dictionary. It is used by the spellchecker to verify the various jargon, variable names, and other odd words used in the FreeRTOS code base. If your pull request fails to pass the spelling and you believe this is a mistake, then add the word to *lexicon.txt*. Note that only the FreeRTOS Kernel source files are checked for proper spelling, the portable section is ignored.

# Chapter 3

# Module Index

## 3.1 Modules

Here is a list of all modules:

# Chapter 4

# Data Structure Index

## 4.1 Data Structures

Here are the data structures with brief descriptions:

# Chapter 5

# Module Documentation

## 5.1 xCoRoutineCreate

croutine. h

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

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

**Parameters**

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

**Returns**

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

Example usage:

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

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

  for( ;; )
  {
      // This co-routine just delays for a fixed period, then toggles
      // an LED.  Two co-routines are created using this function, so
      // the uxIndex parameter is used to tell the co-routine which
      // LED to flash and how int32_t to delay.  This assumes xQueue has
      // already been created.
      vParTestToggleLED( cLedToFlash[ uxIndex ] );
      crDELAY( xHandle, uxFlashRates[ uxIndex ] );
  }

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

// Function that creates two co-routines.
void vOtherFunction( void )
{
uint8_t ucParameterToPass;
TaskHandle_t xHandle;

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

## 5.2 vCoRoutineSchedule

croutine. h

```
void vCoRoutineSchedule( void );
```

Run a co-routine.

vCoRoutineSchedule() executes the highest priority co-routine that is able to run. The co-routine will execute until it either blocks, yields or is preempted by a task. Co-routines execute cooperatively so one co-routine cannot be preempted by another, but can be preempted by a task.

If an application comprises of both tasks and co-routines then vCoRoutineSchedule should be called from the idle task (in an idle task hook).

Example usage:

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

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

## 5.3 crSTART

croutine. h

```
crSTART( CoRoutineHandle_t xHandle );
```

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

Example usage:

```
// Co-routine to be created.
void vACoRoutine( CoRoutineHandle_t xHandle, UBaseType_t uxIndex )
{
// Variables in co-routines must be declared static if they must maintain value across a blocking
static int32_t ulAVariable;

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

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

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

croutine. h

```
crEND();
```

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

Example usage:

```
// Co-routine to be created.
void vACoRoutine( CoRoutineHandle_t xHandle, UBaseType_t uxIndex )
{
// Variables in co-routines must be declared static if they must maintain value across a blocking
static int32_t ulAVariable;

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

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

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

## 5.4 crDELAY

croutine. h

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

Delay a co-routine for a fixed period of time.

crDELAY can only be called from the co-routine function itself - not from within a function called by the co-routine function. This is because co-routines do not maintain their own stack.

**Parameters**

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

Example usage:

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

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

  for( ;; )
  {
     // Delay for 200ms.
     crDELAY( xHandle, xDelayTime );

     // Do something here.
  }

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

## 5.5 crQUEUE_SEND

```
crQUEUE_SEND(
               CoRoutineHandle_t xHandle,
               QueueHandle_t pxQueue,
               void *pvItemToQueue,
               TickType_t xTicksToWait,
               BaseType_t *pxResult
          )
```

The macro's crQUEUE_SEND() and crQUEUE_RECEIVE() are the co-routine equivalent to the xQueueSend() and xQueueReceive() functions used by tasks.

crQUEUE_SEND and crQUEUE_RECEIVE can only be used from a co-routine whereas xQueueSend() and x←QueueReceive() can only be used from tasks.

crQUEUE_SEND can only be called from the co-routine function itself - not from within a function called by the co-routine function. This is because co-routines do not maintain their own stack.

See the co-routine section of the WEB documentation for information on passing data between tasks and co-routines and between ISR's and co-routines.

**Parameters**

| | |
|---|---|
| *xHandle* | The handle of the calling co-routine. This is the xHandle parameter of the co-routine function. |
| *pxQueue* | The handle of the queue on which the data will be posted. The handle is obtained as the return value when the queue is created using the xQueueCreate() API function. |
| *pvItemToQueue* | A pointer to the data being posted onto the queue. The number of bytes of each queued item is specified when the queue is created. This number of bytes is copied from pvItemToQueue into the queue itself. |
| *xTickToDelay* | The number of ticks that the co-routine should block to wait for space to become available on the queue, should space not be available immediately. The actual amount of time this equates to is defined by configTICK_RATE_HZ (set in FreeRTOSConfig.h). The constant portTICK_PERIOD_MS can be used to convert ticks to milliseconds (see example below). |
| *pxResult* | The variable pointed to by pxResult will be set to pdPASS if data was successfully posted onto the queue, otherwise it will be set to an error defined within **ProjDefs.h** (p. **??**). |

Example usage:

```
// Co-routine function that blocks for a fixed period then posts a number onto
// a queue.
static void prvCoRoutineFlashTask( CoRoutineHandle_t xHandle, UBaseType_t uxIndex )
{
// Variables in co-routines must be declared static if they must maintain value across a blocking
static BaseType_t xNumberToPost = 0;
static BaseType_t xResult;

 // Co-routines must begin with a call to crSTART().
 crSTART( xHandle );

 for( ;; )
 {
     // This assumes the queue has already been created.
     crQUEUE_SEND( xHandle, xCoRoutineQueue, &xNumberToPost, NO_DELAY, &xResult );
```

```
        if( xResult != pdPASS )
        {
            // The message was not posted!
        }

        // Increment the number to be posted onto the queue.
        xNumberToPost++;

        // Delay for 100 ticks.
        crDELAY( xHandle, 100 );
    }

    // Co-routines must end with a call to crEND().
    crEND();
}
```

## 5.6 crQUEUE_RECEIVE

croutine. h

```
crQUEUE_RECEIVE(
                  CoRoutineHandle_t xHandle,
                  QueueHandle_t pxQueue,
                  void *pvBuffer,
                  TickType_t xTicksToWait,
                  BaseType_t *pxResult
              )
```

The macro's crQUEUE_SEND() and crQUEUE_RECEIVE() are the co-routine equivalent to the xQueueSend() and xQueueReceive() functions used by tasks.

crQUEUE_SEND and crQUEUE_RECEIVE can only be used from a co-routine whereas xQueueSend() and x↩ QueueReceive() can only be used from tasks.

crQUEUE_RECEIVE can only be called from the co-routine function itself - not from within a function called by the co-routine function. This is because co-routines do not maintain their own stack.

See the co-routine section of the WEB documentation for information on passing data between tasks and co-routines and between ISR's and co-routines.

**Parameters**

| | |
|---|---|
| *xHandle* | The handle of the calling co-routine. This is the xHandle parameter of the co-routine function. |
| *pxQueue* | The handle of the queue from which the data will be received. The handle is obtained as the return value when the queue is created using the xQueueCreate() API function. |
| *pvBuffer* | The buffer into which the received item is to be copied. The number of bytes of each queued item is specified when the queue is created. This number of bytes is copied into pvBuffer. |
| *xTickToDelay* | The number of ticks that the co-routine should block to wait for data to become available from the queue, should data not be available immediately. The actual amount of time this equates to is defined by configTICK_RATE_HZ (set in FreeRTOSConfig.h). The constant portTICK_PERIOD_MS can be used to convert ticks to milliseconds (see the crQUEUE_SEND example). |
| *pxResult* | The variable pointed to by pxResult will be set to pdPASS if data was successfully retrieved from the queue, otherwise it will be set to an error code as defined within **ProjDefs.h** (p. **??**). |

Example usage:

```
// A co-routine receives the number of an LED to flash from a queue.  It
// blocks on the queue until the number is received.
static void prvCoRoutineFlashWorkTask( CoRoutineHandle_t xHandle, UBaseType_t uxIndex )
{
// Variables in co-routines must be declared static if they must maintain value across a blocking
static BaseType_t xResult;
static UBaseType_t uxLEDToFlash;

 // All co-routines must start with a call to crSTART().
 crSTART( xHandle );

 for( ;; )
 {
     // Wait for data to become available on the queue.
```

```
         crQUEUE_RECEIVE( xHandle, xCoRoutineQueue, &uxLEDToFlash, portMAX_DELAY, &xResult );

         if( xResult == pdPASS )
         {
             // We received the LED to flash - flash it!
             vParTestToggleLED( uxLEDToFlash );
         }
     }

     crEND();
 }
```

## 5.7 crQUEUE_SEND_FROM_ISR

croutine. h

```
crQUEUE_SEND_FROM_ISR(
                         QueueHandle_t pxQueue,
                         void *pvItemToQueue,
                         BaseType_t xCoRoutinePreviouslyWoken
                   )
```

The macro's crQUEUE_SEND_FROM_ISR() and crQUEUE_RECEIVE_FROM_ISR() are the co-routine equivalent to the xQueueSendFromISR() and xQueueReceiveFromISR() functions used by tasks.

crQUEUE_SEND_FROM_ISR() and crQUEUE_RECEIVE_FROM_ISR() can only be used to pass data between a co-routine and and ISR, whereas xQueueSendFromISR() and xQueueReceiveFromISR() can only be used to pass data between a task and and ISR.

crQUEUE_SEND_FROM_ISR can only be called from an ISR to send data to a queue that is being used from within a co-routine.

See the co-routine section of the WEB documentation for information on passing data between tasks and co-routines and between ISR's and co-routines.

**Parameters**

| | |
|---|---|
| *xQueue* | The handle to the queue on which the item is to be posted. |
| *pvItemToQueue* | A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area. |
| *xCoRoutinePreviouslyWoken* | This is included so an ISR can post onto the same queue multiple times from a single interrupt. The first call should always pass in pdFALSE. Subsequent calls should pass in the value returned from the previous call. |

**Returns**

pdTRUE if a co-routine was woken by posting onto the queue. This is used by the ISR to determine if a context switch may be required following the ISR.

Example usage:

```
// A co-routine that blocks on a queue waiting for characters to be received.
static void vReceivingCoRoutine( CoRoutineHandle_t xHandle, UBaseType_t uxIndex )
{
char cRxedChar;
BaseType_t xResult;

  // All co-routines must start with a call to crSTART().
  crSTART( xHandle );

  for( ;; )
  {
      // Wait for data to become available on the queue.  This assumes the
      // queue xCommsRxQueue has already been created!
      crQUEUE_RECEIVE( xHandle, xCommsRxQueue, &uxLEDToFlash, portMAX_DELAY, &xResult );
      // Was a character received?
      if( xResult == pdPASS )
      {
          // Process the character here.
      }
  }

  // All co-routines must end with a call to crEND().
  crEND();
}

// An ISR that uses a queue to send characters received on a serial port to
// a co-routine.
void vUART_ISR( void )
{
char cRxedChar;
BaseType_t xCRWokenByPost = pdFALSE;

  // We loop around reading characters until there are none left in the UART.
  while( UART_RX_REG_NOT_EMPTY() )
  {
      // Obtain the character from the UART.
      cRxedChar = UART_RX_REG;

      // Post the character onto a queue.  xCRWokenByPost will be pdFALSE
      // the first time around the loop.  If the post causes a co-routine
      // to be woken (unblocked) then xCRWokenByPost will be set to pdTRUE.
      // In this manner we can ensure that if more than one co-routine is
      // blocked on the queue only one is woken by this ISR no matter how
      // many characters are posted to the queue.
      xCRWokenByPost = crQUEUE_SEND_FROM_ISR( xCommsRxQueue, &cRxedChar, xCRWokenByPost );
  }
}
```

## 5.8 crQUEUE_RECEIVE_FROM_ISR

croutine. h

```
crQUEUE_SEND_FROM_ISR(
                         QueueHandle_t pxQueue,
                         void *pvBuffer,
                         BaseType_t * pxCoRoutineWoken
                    )
```

The macro's crQUEUE_SEND_FROM_ISR() and crQUEUE_RECEIVE_FROM_ISR() are the co-routine equivalent to the xQueueSendFromISR() and xQueueReceiveFromISR() functions used by tasks.

crQUEUE_SEND_FROM_ISR() and crQUEUE_RECEIVE_FROM_ISR() can only be used to pass data between a co-routine and and ISR, whereas xQueueSendFromISR() and xQueueReceiveFromISR() can only be used to pass data between a task and and ISR.

crQUEUE_RECEIVE_FROM_ISR can only be called from an ISR to receive data from a queue that is being used from within a co-routine (a co-routine posted to the queue).

See the co-routine section of the WEB documentation for information on passing data between tasks and co-routines and between ISR's and co-routines.

**Parameters**

| | |
|---|---|
| *xQueue* | The handle to the queue on which the item is to be posted. |
| *pvBuffer* | A pointer to a buffer into which the received item will be placed. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from the queue into pvBuffer. |
| *pxCoRoutineWoken* | A co-routine may be blocked waiting for space to become available on the queue. If crQUEUE_RECEIVE_FROM_ISR causes such a co-routine to unblock ∗pxCoRoutineWoken will get set to pdTRUE, otherwise ∗pxCoRoutineWoken will remain unchanged. |

**Returns**

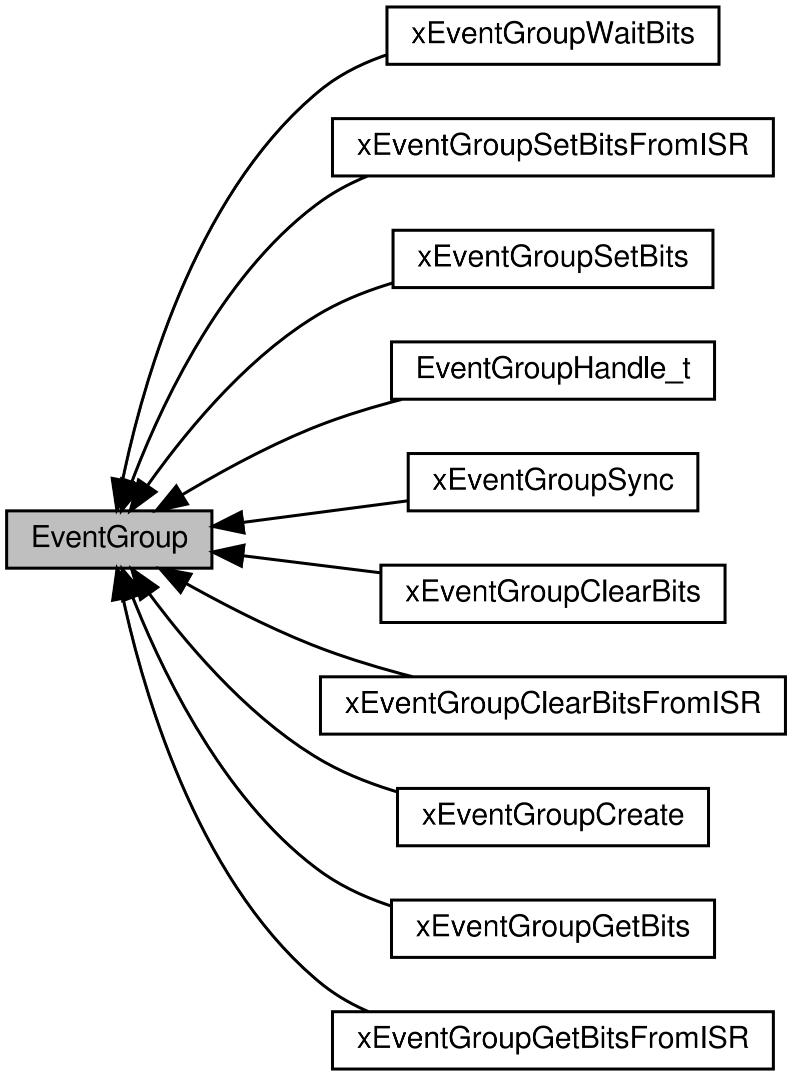
pdTRUE an item was successfully received from the queue, otherwise pdFALSE.

Example usage:

```
// A co-routine that posts a character to a queue then blocks for a fixed
// period.  The character is incremented each time.
static void vSendingCoRoutine( CoRoutineHandle_t xHandle, UBaseType_t uxIndex )
{
// cChar holds its value while this co-routine is blocked and must therefore
// be declared static.
static char cCharToTx = 'a';
BaseType_t xResult;

  // All co-routines must start with a call to crSTART().
  crSTART( xHandle );

  for( ;; )
  {
      // Send the next character to the queue.
      crQUEUE_SEND( xHandle, xCoRoutineQueue, &cCharToTx, NO_DELAY, &xResult );

      if( xResult == pdPASS )
      {
          // The character was successfully posted to the queue.
      }
      else
      {
         // Could not post the character to the queue.
      }

      // Enable the UART Tx interrupt to cause an interrupt in this
      // hypothetical UART.  The interrupt will obtain the character
      // from the queue and send it.
      ENABLE_RX_INTERRUPT();

      // Increment to the next character then block for a fixed period.
      // cCharToTx will maintain its value across the delay as it is
      // declared static.
      cCharToTx++;
      if( cCharToTx > 'x' )
      {
         cCharToTx = 'a';
      }
      crDELAY( 100 );
  }

  // All co-routines must end with a call to crEND().
  crEND();
}

// An ISR that uses a queue to receive characters to send on a UART.
void vUART_ISR( void )
{
char cCharToTx;
BaseType_t xCRWokenByPost = pdFALSE;

  while( UART_TX_REG_EMPTY() )
  {
      // Are there any characters in the queue waiting to be sent?
      // xCRWokenByPost will automatically be set to pdTRUE if a co-routine
      // is woken by the post - ensuring that only a single co-routine is
      // woken no matter how many times we go around this loop.
      if( crQUEUE_RECEIVE_FROM_ISR( pxQueue, &cCharToTx, &xCRWokenByPost ) )
      {
          SEND_CHARACTER( cCharToTx );
      }
  }
}
```

## 5.9 EventGroup

Collaboration diagram for EventGroup:



### Modules

- **EventGroupHandle_t**
- **xEventGroupCreate**
- **xEventGroupWaitBits**
- **xEventGroupClearBits**
- **xEventGroupClearBitsFromISR**
- **xEventGroupSetBits**
- **xEventGroupSetBitsFromISR**
- **xEventGroupSync**
- **xEventGroupGetBits**
- **xEventGroupGetBitsFromISR**

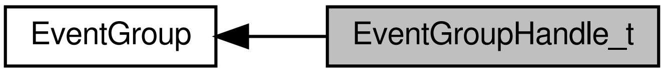
### 5.9.1 Detailed Description

An event group is a collection of bits to which an application can assign a meaning. For example, an application may create an event group to convey the status of various CAN bus related events in which bit 0 might mean "A CAN message has been received and is ready for processing", bit 1 might mean "The application has queued a message that is ready for sending onto the CAN network", and bit 2 might mean "It is time to send a SYNC message onto the CAN network" etc. A task can then test the bit values to see which events are active, and optionally enter the Blocked state to wait for a specified bit or a group of specified bits to be active. To continue the CAN bus example, a CAN controlling task can enter the Blocked state (and therefore not consume any processing time) until either bit 0, bit 1 or bit 2 are active, at which time the bit that was actually active would inform the task which action it had to take (process a received message, send a message, or send a SYNC).

The event groups implementation contains intelligence to avoid race conditions that would otherwise occur were an application to use a simple variable for the same purpose. This is particularly important with respect to when a bit within an event group is to be cleared, and when bits have to be set and then tested atomically - as is the case where event groups are used to create a synchronisation point between multiple tasks (a 'rendezvous').

## 5.10 EventGroupHandle_t

Collaboration diagram for EventGroupHandle_t:

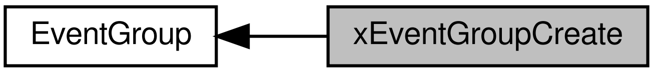


**event_groups.h** (p. **??**)

Type by which event groups are referenced. For example, a call to xEventGroupCreate() returns an EventGroup↩
Handle_t variable that can then be used as a parameter to other event group functions.

## 5.11 xEventGroupCreate

Collaboration diagram for xEventGroupCreate:



**event_groups.h** (p. **??**)

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

Create a new event group.

Internally, within the FreeRTOS implementation, event groups use a [small] block of memory, in which the event group's structure is stored. If an event groups is created using xEventGropuCreate() then the required memory is automatically dynamically allocated inside the xEventGroupCreate() function. (see `https://www.FreeRTOS.org/a00111.html`). If an event group is created using xEventGropuCreateStatic() then the application writer must instead provide the memory that will get used by the event group. xEventGroupCreateStatic() therefore allows an event group to be created without using any dynamic memory allocation.

Although event groups are not related to ticks, for internal implementation reasons the number of bits available for use in an event group is dependent on the configUSE_16_BIT_TICKS setting in FreeRTOSConfig.h. If configUSE_16_BIT_TICKS is 1 then each event group contains 8 usable bits (bit 0 to bit 7). If configUSE_16_BIT_TICKS is set to 0 then each event group has 24 usable bits (bit 0 to bit 23). The EventBits_t type is used to store event bits within an event group.

**Returns**

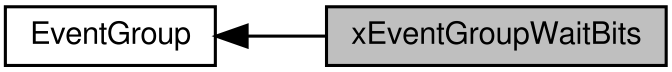
If the event group was created then a handle to the event group is returned. If there was insufficient FreeRTOS heap available to create the event group then NULL is returned. See `https://www.FreeRTOS.org/a00111.html`

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.
}
```

## 5.12 xEventGroupWaitBits

Collaboration diagram for xEventGroupWaitBits:



**event_groups.h** (p. **??**)

```
EventGroupHandle_t xEventGroupCreateStatic( EventGroupHandle_t * pxEventGroupBuffer );
```

Create a new event group.

Internally, within the FreeRTOS implementation, event groups use a [small] block of memory, in which the event group's structure is stored. If an event groups is created using xEventGropuCreate() then the required memory is automatically dynamically allocated inside the xEventGroupCreate() function. (see `https://www.Free←RTOS.org/a00111.html`). If an event group is created using xEventGropuCreateStatic() then the application writer must instead provide the memory that will get used by the event group. xEventGroupCreateStatic() therefore allows an event group to be created without using any dynamic memory allocation.

Although event groups are not related to ticks, for internal implementation reasons the number of bits available for use in an event group is dependent on the configUSE_16_BIT_TICKS setting in FreeRTOSConfig.h. If configUSE←_16_BIT_TICKS is 1 then each event group contains 8 usable bits (bit 0 to bit 7). If configUSE_16_BIT_TICKS is set to 0 then each event group has 24 usable bits (bit 0 to bit 23). The EventBits_t type is used to store event bits within an event group.

**Parameters**

| | |
|---|---|
| *pxEventGroupBuffer* | pxEventGroupBuffer must point to a variable of type StaticEventGroup_t, which will be then be used to hold the event group's data structures, removing the need for the memory to be allocated dynamically. |

**Returns**

If the event group was created then a handle to the event group is returned. If pxEventGroupBuffer was NULL then NULL is returned.

Example usage:

```
// StaticEventGroup_t is a publicly accessible structure that has the same
// size and alignment requirements as the real event group structure.  It is
// provided as a mechanism for applications to know the size of the event
// group (which is dependent on the architecture and configuration file
// settings) without breaking the strict data hiding policy by exposing the
// real event group internals.  This StaticEventGroup_t variable is passed
// into the xSemaphoreCreateEventGroupStatic() function and is used to store
// the event group's data structures
StaticEventGroup_t xEventGroupBuffer;

// Create the event group without dynamically allocating any memory.
xEventGroup = xEventGroupCreateStatic( &xEventGroupBuffer );
```

**event_groups.h** (p. **??**)

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

[Potentially] block to wait for one or more bits to be set within a previously created event group.

This function cannot be called from an interrupt.

**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. |
| *xClearOnExit* | If xClearOnExit is set to pdTRUE then any bits within uxBitsToWaitFor that are set within the event group will be cleared before xEventGroupWaitBits() returns if the wait condition was met (if the function returns for a reason other than a timeout). If xClearOnExit is set to pdFALSE then the bits set in the event group are not altered when the call to xEventGroupWaitBits() returns. |
| *xWaitForAllBits* | If xWaitForAllBits is set to pdTRUE then xEventGroupWaitBits() will return when either all the bits in uxBitsToWaitFor are set or the specified block time expires. If xWaitForAllBits is set to pdFALSE then xEventGroupWaitBits() will return when any one of the bits set in uxBitsToWaitFor is set or the specified block time expires. The block time is specified by the xTicksToWait parameter. |
| *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 bits being waited for became set, or the block time expired. 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 in the case that xClearOnExit parameter was set to pdTRUE.

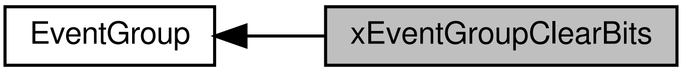
Example usage:

```
#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 and 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.
        }
    }
```

## 5.13 xEventGroupClearBits

Collaboration diagram for xEventGroupClearBits:



**event_groups.h** (p. **??**)

```
   EventBits_t xEventGroupClearBits( EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToCle
```

Clear bits within an event group. This function cannot be called from an interrupt.

**Parameters**

| | |
|---|---|
| *xEventGroup* | The event group in which the bits are to be cleared. |
| *uxBitsToClear* | A bitwise value that indicates the bit or bits to clear in the event group. For example, to clear bit 3 only, set uxBitsToClear to 0x08. To clear bit 3 and bit 0 set uxBitsToClear to 0x09. |

**Returns**

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

Example usage:

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

  void aFunction( EventGroupHandle_t xEventGroup )
  {
  EventBits_t uxBits;

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

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

```
        else
        {
            // Neither bit 0 nor bit 4 were set in the first place.
        }
}
```

## 5.14 xEventGroupClearBitsFromISR

Collaboration diagram for xEventGroupClearBitsFromISR:

**event_groups.h** (p. **??**)

```
   BaseType_t xEventGroupClearBitsFromISR( EventGroupHandle_t xEventGroup, const EventBits_t uxBit
```

A version of xEventGroupClearBits() that can be called from an interrupt.

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 nondeterministic operations to be performed while interrupts are disabled, so protects event groups that are accessed from tasks by suspending the scheduler rather than disabling interrupts. As a result event groups cannot be accessed directly from an interrupt service routine. Therefore xEventGroupClearBitsFromISR() sends a message to the timer task to have the clear operation performed in the context of the timer task.

**Parameters**

| | |
|---|---|
| *xEventGroup* | The event group in which the bits are to be cleared. |
| *uxBitsToClear* | A bitwise value that indicates the bit or bits to clear. For example, to clear bit 3 only, set uxBitsToClear to 0x08. To clear bit 3 and bit 0 set uxBitsToClear to 0x09. |

**Returns**

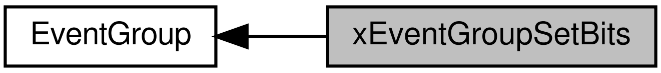
If the request to execute the function was posted successfully then pdPASS is returned, otherwise pdFALSE is returned. pdFALSE will be returned if the timer service queue was full.

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

       if( xResult == pdPASS )
       {
           // The message was posted successfully.
       }
  }
```

## 5.15 xEventGroupSetBits

Collaboration diagram for xEventGroupSetBits:



**event_groups.h** (p. **??**)

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

Set bits within an 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.

**Parameters**

| | |
|---|---|
| *xEventGroup* | The event group in which the bits are to be set. |
| *uxBitsToSet* | A bitwise value that indicates the bit or bits to set. For example, to set bit 3 only, set uxBitsToSet to 0x08. To set bit 3 and bit 0 set uxBitsToSet to 0x09. |

**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. First, 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 be cleared automatically (see the xClearBitOnExit parameter of xEventGroupWaitBits()). Second, 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.
       }
  }
```

## 5.16 xEventGroupSetBitsFromISR

Collaboration diagram for xEventGroupSetBitsFromISR:

**event_groups.h** (p. **??**)

```
   BaseType_t xEventGroupSetBitsFromISR( EventGroupHandle_t xEventGroup, const EventBits_t uxBitsT
```

A version of xEventGroupSetBits() that can be called from an interrupt.

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 nondeterministic operations to be performed in interrupts or from critical sections. Therefore xEventGroupSetBitsFromISR() sends a message to the timer task to have the set operation performed in the context of the timer task - where a scheduler lock is used in place of a critical section.

**Parameters**

| | |
|---|---|
| *xEventGroup* | The event group in which the bits are to be set. |
| *uxBitsToSet* | A bitwise value that indicates the bit or bits to set. For example, to set bit 3 only, set uxBitsToSet to 0x08. To set bit 3 and bit 0 set uxBitsToSet to 0x09. |
| *pxHigherPriorityTaskWoken* | As mentioned above, calling this function will result in a message being sent to the timer daemon task. If the priority of the timer 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 request to execute the function was posted successfully then pdPASS is returned, otherwise pdFALSE is returned. pdFALSE will be returned if the timer service queue was full.

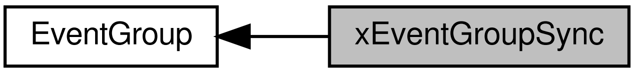
Example usage:

```
#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 == pdPASS )
        {
            // 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 );
        }
   }
```

## 5.17 xEventGroupSync

Collaboration diagram for xEventGroupSync:



**event_groups.h** (p. **??**)

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

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

This function cannot be used from an interrupt.

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

**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(). |
| *uxBitsToSet* | The bits to set in the event group before determining if, and possibly waiting for, all the bits specified by the uxBitsToWait parameter are set. |
| *uxBitsToWaitFor* | A bitwise value that indicates the bit or bits to test inside the event group. For example, to wait for bit 0 and bit 2 set uxBitsToWaitFor to 0x05. To wait for bits 0 and bit 1 and bit 2 set uxBitsToWaitFor to 0x07. Etc. |
| *xTicksToWait* | The maximum amount of time (specified in 'ticks') to wait for all of the bits specified by uxBitsToWaitFor to become set. |

**Returns**

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

Example usage:

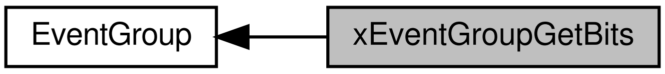
```
  // Bits used by the three tasks.
#define TASK_0_BIT      ( 1 << 0 )
#define TASK_1_BIT      ( 1 << 1 )
#define TASK_2_BIT      ( 1 << 2 )
```

```
#define ALL_SYNC_BITS ( TASK_0_BIT | TASK_1_BIT | TASK_2_BIT )

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

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

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

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

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

  void vTask1( void *pvParameters )
  {
    for( ;; )
    {
       // Perform task functionality here.

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

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

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

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

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

## 5.18 xEventGroupGetBits

Collaboration diagram for xEventGroupGetBits:



**event_groups.h** (p. **??**)

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

Returns the current value of the bits in an event group. This function cannot be used from an interrupt.

**Parameters**

| | |
|---|---|
| *xEventGroup* | The event group being queried. |

**Returns**

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

## 5.19 xEventGroupGetBitsFromISR

Collaboration diagram for xEventGroupGetBitsFromISR:

**event_groups.h** (p. **??**)

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

A version of xEventGroupGetBits() that can be called from an ISR.

**Parameters**

| | |
|---|---|
| *xEventGroup* | The event group being queried. |

**Returns**

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

## 5.20 xMessageBufferCreate

**message_buffer.h** (p. **??**)

```
MessageBufferHandle_t xMessageBufferCreate( size_t xBufferSizeBytes );
```

Creates a new message buffer using dynamically allocated memory. See xMessageBufferCreateStatic() for a version that uses statically allocated memory (memory that is allocated at compile time).

configSUPPORT_DYNAMIC_ALLOCATION must be set to 1 or left undefined in FreeRTOSConfig.h for xMessage↩ BufferCreate() to be available.

**Parameters**

| | |
|---|---|
| *xBufferSizeBytes* | The total number of bytes (not messages) the message buffer will be able to hold at any one time. When a message is written to the message buffer an additional sizeof( size_t ) bytes are also written to store the message's length. sizeof( size_t ) is typically 4 bytes on a 32-bit architecture, so on most 32-bit architectures a 10 byte message will take up 14 bytes of message buffer space. |

**Returns**

If NULL is returned, then the message buffer cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the message buffer data structures and storage area. A non-NULL value being returned indicates that the message buffer has been created successfully - the returned value should be stored as the handle to the created message buffer.

Example use:

```
void vAFunction( void )
{
MessageBufferHandle_t xMessageBuffer;
const size_t xMessageBufferSizeBytes = 100;

 // Create a message buffer that can hold 100 bytes.  The memory used to hold
 // both the message buffer structure and the messages themselves is allocated
 // dynamically.  Each message added to the buffer consumes an additional 4
 // bytes which are used to hold the lengh of the message.
 xMessageBuffer = xMessageBufferCreate( xMessageBufferSizeBytes );

 if( xMessageBuffer == NULL )
 {
     // There was not enough heap memory space available to create the
     // message buffer.
 }
 else
 {
     // The message buffer was created successfully and can now be used.
 }
```

## 5.21 xMessageBufferCreateStatic

**message_buffer.h** (p. **??**)

```
  MessageBufferHandle_t xMessageBufferCreateStatic( size_t xBufferSizeBytes,
                                                    uint8_t *pucMessageBufferStorageArea,
                                                    StaticMessageBuffer_t *pxStaticMessageBuffer );
```

Creates a new message buffer using statically allocated memory. See xMessageBufferCreate() for a version that uses dynamically allocated memory.

**Parameters**

| | |
|---|---|
| *xBufferSizeBytes* | The size, in bytes, of the buffer pointed to by the pucMessageBufferStorageArea parameter. When a message is written to the message buffer an additional sizeof( size_t ) bytes are also written to store the message's length. sizeof( size_t ) is typically 4 bytes on a 32-bit architecture, so on most 32-bit architecture a 10 byte message will take up 14 bytes of message buffer space. The maximum number of bytes that can be stored in the message buffer is actually (xBufferSizeBytes - 1). |
| *pucMessageBufferStorageArea* | Must point to a uint8_t array that is at least xBufferSizeBytes + 1 big. This is the array to which messages are copied when they are written to the message buffer. |
| *pxStaticMessageBuffer* | Must point to a variable of type StaticMessageBuffer_t, which will be used to hold the message buffer's data structure. |

**Returns**

If the message buffer is created successfully then a handle to the created message buffer is returned. If either pucMessageBufferStorageArea or pxStaticmessageBuffer are NULL then NULL is returned.

Example use:

```
  // Used to dimension the array used to hold the messages.  The available space
  // will actually be one less than this, so 999.
#define STORAGE_SIZE_BYTES 1000

  // Defines the memory that will actually hold the messages within the message
  // buffer.
  static uint8_t ucStorageBuffer[ STORAGE_SIZE_BYTES ];

  // The variable used to hold the message buffer structure.
  StaticMessageBuffer_t xMessageBufferStruct;

  void MyFunction( void )
  {
  MessageBufferHandle_t xMessageBuffer;

   xMessageBuffer = xMessageBufferCreateStatic( sizeof( ucBufferStorage ),
                                                ucBufferStorage,
                                                &xMessageBufferStruct );

   // As neither the pucMessageBufferStorageArea or pxStaticMessageBuffer
```

```
    // parameters were NULL, xMessageBuffer will not be NULL, and can be used to
    // reference the created message buffer in other message buffer API calls.

    // Other code that uses the message buffer can go here.
}
```

## 5.22 xMessageBufferSend

**message_buffer.h** (p. **??**)

```
size_t xMessageBufferSend( MessageBufferHandle_t xMessageBuffer,
                           const void *pvTxData,
                           size_t xDataLengthBytes,
                           TickType_t xTicksToWait );
```

Sends a discrete message to the message buffer. The message can be any length that fits within the buffer's free space, and is copied into the buffer.

***NOTE***: Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xMessageBufferSend()) inside a critical section and set the send block time to 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xMessageBufferRead()) inside a critical section and set the receive block time to 0.

Use xMessageBufferSend() to write to a message buffer from a task. Use xMessageBufferSendFromISR() to write to a message buffer from an interrupt service routine (ISR).

**Parameters**

| | |
|---|---|
| *xMessageBuffer* | The handle of the message buffer to which a message is being sent. |
| *pvTxData* | A pointer to the message that is to be copied into the message buffer. |
| *xDataLengthBytes* | The length of the message. That is, the number of bytes to copy from pvTxData into the message buffer. When a message is written to the message buffer an additional sizeof( size_t ) bytes are also written to store the message's length. sizeof( size_t ) is typically 4 bytes on a 32-bit architecture, so on most 32-bit architecture setting xDataLengthBytes to 20 will reduce the free space in the message buffer by 24 bytes (20 bytes of message data and 4 bytes to hold the message length). |
| *xTicksToWait* | The maximum amount of time the calling task should remain in the Blocked state to wait for enough space to become available in the message buffer, should the message buffer have insufficient space when xMessageBufferSend() is called. The calling task will never block if xTicksToWait is zero. The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h. Tasks do not use any CPU time when they are in the Blocked state. |

**Returns**

The number of bytes written to the message buffer. If the call to xMessageBufferSend() times out before there was enough space to write the message into the message buffer then zero is returned. If the call did not time out then xDataLengthBytes is returned.

Example use:

```
void vAFunction( MessageBufferHandle_t xMessageBuffer )
{
size_t xBytesSent;
uint8_t ucArrayToSend[] = { 0, 1, 2, 3 };
char *pcStringToSend = "String to send";
const TickType_t x100ms = pdMS_TO_TICKS( 100 );

 // Send an array to the message buffer, blocking for a maximum of 100ms to
 // wait for enough space to be available in the message buffer.
 xBytesSent = xMessageBufferSend( xMessageBuffer, ( void * ) ucArrayToSend, sizeof( ucArrayToSend

 if( xBytesSent != sizeof( ucArrayToSend ) )
 {
     // The call to xMessageBufferSend() times out before there was enough
     // space in the buffer for the data to be written.
 }

 // Send the string to the message buffer.  Return immediately if there is
 // not enough space in the buffer.
 xBytesSent = xMessageBufferSend( xMessageBuffer, ( void * ) pcStringToSend, strlen( pcStringToSen

 if( xBytesSent != strlen( pcStringToSend ) )
 {
     // The string could not be added to the message buffer because there was
     // not enough free space in the buffer.
 }
}
```

## 5.23 xMessageBufferSendFromISR

**message_buffer.h** (p. **??**)

```
size_t xMessageBufferSendFromISR( MessageBufferHandle_t xMessageBuffer,
                                  const void *pvTxData,
                                  size_t xDataLengthBytes,
                                  BaseType_t *pxHigherPriorityTaskWoken );
```

Interrupt safe version of the API function that sends a discrete message to the message buffer. The message can be any length that fits within the buffer's free space, and is copied into the buffer.

***NOTE***: Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xMessageBufferSend()) inside a critical section and set the send block time to 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xMessageBufferRead()) inside a critical section and set the receive block time to 0.

Use xMessageBufferSend() to write to a message buffer from a task. Use xMessageBufferSendFromISR() to write to a message buffer from an interrupt service routine (ISR).

**Parameters**

| | |
|---|---|
| *xMessageBuffer* | The handle of the message buffer to which a message is being sent. |
| *pvTxData* | A pointer to the message that is to be copied into the message buffer. |
| *xDataLengthBytes* | The length of the message. That is, the number of bytes to copy from pvTxData into the message buffer. When a message is written to the message buffer an additional sizeof( size_t ) bytes are also written to store the message's length. sizeof( size_t ) is typically 4 bytes on a 32-bit architecture, so on most 32-bit architecture setting xDataLengthBytes to 20 will reduce the free space in the message buffer by 24 bytes (20 bytes of message data and 4 bytes to hold the message length). |
| *pxHigherPriorityTaskWoken* | It is possible that a message buffer will have a task blocked on it waiting for data. Calling xMessageBufferSendFromISR() can make data available, and so cause a task that was waiting for data to leave the Blocked state. If calling xMessageBufferSendFromISR() causes a task to leave the Blocked state, and the unblocked task has a priority higher than the currently executing task (the task that was interrupted), then, internally, xMessageBufferSendFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE. If xMessageBufferSendFromISR() sets this value to pdTRUE, then normally a context switch should be performed before the interrupt is exited. This will ensure that the interrupt returns directly to the highest priority Ready state task. *pxHigherPriorityTaskWoken should be set to pdFALSE before it is passed into the function. See the code example below for an example. |

**Returns**

The number of bytes actually written to the message buffer. If the message buffer didn't have enough free space for the message to be stored then 0 is returned, otherwise xDataLengthBytes is returned.

Example use:

```
// A message buffer that has already been created.
MessageBufferHandle_t xMessageBuffer;

void vAnInterruptServiceRoutine( void )
{
size_t xBytesSent;
char *pcStringToSend = "String to send";
BaseType_t xHigherPriorityTaskWoken = pdFALSE; // Initialised to pdFALSE.

 // Attempt to send the string to the message buffer.
 xBytesSent = xMessageBufferSendFromISR( xMessageBuffer,
                                         ( void * ) pcStringToSend,
                                         strlen( pcStringToSend ),
                                         &xHigherPriorityTaskWoken );

 if( xBytesSent != strlen( pcStringToSend ) )
 {
     // The string could not be added to the message buffer because there was
     // not enough free space in the buffer.
 }

 // If xHigherPriorityTaskWoken was set to pdTRUE inside
 // xMessageBufferSendFromISR() then a task that has a priority above the
 // priority of the currently executing task was unblocked and a context
 // switch should be performed to ensure the ISR returns to the unblocked
 // task.  In most FreeRTOS ports this is done by simply passing
 // xHigherPriorityTaskWoken into portYIELD_FROM_ISR(), which will test the
 // variables value, and perform the context switch if necessary.  Check the
 // documentation for the port in use for port specific instructions.
 portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

## 5.24 xMessageBufferReceive

**message_buffer.h** (p. **??**)

```
size_t xMessageBufferReceive( MessageBufferHandle_t xMessageBuffer,
                              void *pvRxData,
                              size_t xBufferLengthBytes,
                              TickType_t xTicksToWait );
```

Receives a discrete message from a message buffer. Messages can be of variable length and are copied out of the buffer.

***NOTE***: Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xMessageBufferSend()) inside a critical section and set the send block time to 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xMessageBufferRead()) inside a critical section and set the receive block time to 0.

Use xMessageBufferReceive() to read from a message buffer from a task. Use xMessageBufferReceiveFromISR() to read from a message buffer from an interrupt service routine (ISR).

**Parameters**

| | |
|---|---|
| *xMessageBuffer* | The handle of the message buffer from which a message is being received. |
| *pvRxData* | A pointer to the buffer into which the received message is to be copied. |
| *xBufferLengthBytes* | The length of the buffer pointed to by the pvRxData parameter. This sets the maximum length of the message that can be received. If xBufferLengthBytes is too small to hold the next message then the message will be left in the message buffer and 0 will be returned. |
| *xTicksToWait* | The maximum amount of time the task should remain in the Blocked state to wait for a message, should the message buffer be empty. xMessageBufferReceive() will return immediately if xTicksToWait is zero and the message buffer is empty. The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h. Tasks do not use any CPU time when they are in the Blocked state. |

**Returns**

The length, in bytes, of the message read from the message buffer, if any. If xMessageBufferReceive() times out before a message became available then zero is returned. If the length of the message is greater than xBufferLengthBytes then the message will be left in the message buffer and zero is returned.

Example use:

```
void vAFunction( MessageBuffer_t xMessageBuffer )
{
```

```
uint8_t ucRxData[ 20 ];
size_t xReceivedBytes;
const TickType_t xBlockTime = pdMS_TO_TICKS( 20 );

 // Receive the next message from the message buffer.  Wait in the Blocked
 // state (so not using any CPU processing time) for a maximum of 100ms for
 // a message to become available.
 xReceivedBytes = xMessageBufferReceive( xMessageBuffer,
                                         ( void * ) ucRxData,
                                         sizeof( ucRxData ),
                                         xBlockTime );

 if( xReceivedBytes > 0 )
 {
     // A ucRxData contains a message that is xReceivedBytes long.  Process
     // the message here....
 }
}
```

## 5.25 xMessageBufferReceiveFromISR

**message_buffer.h** (p. **??**)

```
size_t xMessageBufferReceiveFromISR( MessageBufferHandle_t xMessageBuffer,
                                     void *pvRxData,
                                     size_t xBufferLengthBytes,
                                     BaseType_t *pxHigherPriorityTaskWoken );
```

An interrupt safe version of the API function that receives a discrete message from a message buffer. Messages can be of variable length and are copied out of the buffer.

***NOTE***: Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xMessageBufferSend()) inside a critical section and set the send block time to 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xMessageBufferRead()) inside a critical section and set the receive block time to 0.

Use xMessageBufferReceive() to read from a message buffer from a task. Use xMessageBufferReceiveFromISR() to read from a message buffer from an interrupt service routine (ISR).

**Parameters**

| | |
|---|---|
| *xMessageBuffer* | The handle of the message buffer from which a message is being received. |
| *pvRxData* | A pointer to the buffer into which the received message is to be copied. |
| *xBufferLengthBytes* | The length of the buffer pointed to by the pvRxData parameter. This sets the maximum length of the message that can be received. If xBufferLengthBytes is too small to hold the next message then the message will be left in the message buffer and 0 will be returned. |
| *pxHigherPriorityTaskWoken* | It is possible that a message buffer will have a task blocked on it waiting for space to become available. Calling xMessageBufferReceiveFromISR() can make space available, and so cause a task that is waiting for space to leave the Blocked state. If calling xMessageBufferReceiveFromISR() causes a task to leave the Blocked state, and the unblocked task has a priority higher than the currently executing task (the task that was interrupted), then, internally, xMessageBufferReceiveFromISR() will set ∗pxHigherPriorityTaskWoken to pdTRUE. If xMessageBufferReceiveFromISR() sets this value to pdTRUE, then normally a context switch should be performed before the interrupt is exited. That will ensure the interrupt returns directly to the highest priority Ready state task. ∗pxHigherPriorityTaskWoken should be set to pdFALSE before it is passed into the function. See the code example below for an example. |

**Returns**

The length, in bytes, of the message read from the message buffer, if any.

Example use:

```
// A message buffer that has already been created.
```

```
MessageBuffer_t xMessageBuffer;

void vAnInterruptServiceRoutine( void )
{
uint8_t ucRxData[ 20 ];
size_t xReceivedBytes;
BaseType_t xHigherPriorityTaskWoken = pdFALSE;  // Initialised to pdFALSE.

 // Receive the next message from the message buffer.
 xReceivedBytes = xMessageBufferReceiveFromISR( xMessageBuffer,
                                               ( void * ) ucRxData,
                                               sizeof( ucRxData ),
                                               &xHigherPriorityTaskWoken );

 if( xReceivedBytes > 0 )
 {
     // A ucRxData contains a message that is xReceivedBytes long.  Process
     // the message here....
 }

 // If xHigherPriorityTaskWoken was set to pdTRUE inside
 // xMessageBufferReceiveFromISR() then a task that has a priority above the
 // priority of the currently executing task was unblocked and a context
 // switch should be performed to ensure the ISR returns to the unblocked
 // task.  In most FreeRTOS ports this is done by simply passing
 // xHigherPriorityTaskWoken into portYIELD_FROM_ISR(), which will test the
 // variables value, and perform the context switch if necessary.  Check the
 // documentation for the port in use for port specific instructions.
 portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

## 5.26 xMessageBufferReset

**message_buffer.h** (p. **??**)

```
BaseType_t xMessageBufferReset( MessageBufferHandle_t xMessageBuffer );
```

Resets a message buffer to its initial empty state, discarding any message it contained.

A message buffer can only be reset if there are no tasks blocked on it.

**Parameters**

| | |
|---|---|
| *xMessageBuffer* | The handle of the message buffer being reset. |

**Returns**

If the message buffer was reset then pdPASS is returned. If the message buffer could not be reset because either there was a task blocked on the message queue to wait for space to become available, or to wait for a a message to be available, then pdFAIL is returned.

## 5.27 xMessageBufferSpaceAvailable

**message_buffer.h** (p. **??**)

```
size_t xMessageBufferSpaceAvailable( MessageBufferHandle_t xMessageBuffer ) );
```

Returns the number of bytes of free space in the message buffer.

**Parameters**

| | |
|---|---|
| *xMessageBuffer* | The handle of the message buffer being queried. |

**Returns**

The number of bytes that can be written to the message buffer before the message buffer would be full. When a message is written to the message buffer an additional sizeof( size_t ) bytes are also written to store the message's length. sizeof( size_t ) is typically 4 bytes on a 32-bit architecture, so if xMessageBufferSpaces↩ Available() returns 10, then the size of the largest message that can be written to the message buffer is 6 bytes.

## 5.28 xMessageBufferNextLengthBytes

**message_buffer.h** (p. **??**)

```
size_t xMessageBufferNextLengthBytes( MessageBufferHandle_t xMessageBuffer ) );
```

Returns the length (in bytes) of the next message in a message buffer. Useful if xMessageBufferReceive() returned 0 because the size of the buffer passed into xMessageBufferReceive() was too small to hold the next message.

**Parameters**

| | |
|---|---|
| *xMessageBuffer* | The handle of the message buffer being queried. |

**Returns**

The length (in bytes) of the next message in the message buffer, or 0 if the message buffer is empty.

## 5.29 xMessageBufferSendCompletedFromISR

**message_buffer.h** (p. **??**)

```
BaseType_t xMessageBufferSendCompletedFromISR( MessageBufferHandle_t xStreamBuffer, BaseType_t *px
```

For advanced users only.

The sbSEND_COMPLETED() macro is called from within the FreeRTOS APIs when data is sent to a message buffer or stream buffer. If there was a task that was blocked on the message or stream buffer waiting for data to arrive then the sbSEND_COMPLETED() macro sends a notification to the task to remove it from the Blocked state. xMessageBufferSendCompletedFromISR() does the same thing. It is provided to enable application writers to implement their own version of sbSEND_COMPLETED(), and MUST NOT BE USED AT ANY OTHER TIME.

See the example implemented in FreeRTOS/Demo/Minimal/MessageBufferAMP.c for additional information.

**Parameters**

| | |
|---|---|
| *xStreamBuffer* | The handle of the stream buffer to which data was written. |
| *pxHigherPriorityTaskWoken* | ∗pxHigherPriorityTaskWoken should be initialised to pdFALSE before it is passed into xMessageBufferSendCompletedFromISR(). If calling xMessageBufferSendCompletedFromISR() removes a task from the Blocked state, and the task has a priority above the priority of the currently running task, then ∗pxHigherPriorityTaskWoken will get set to pdTRUE indicating that a context switch should be performed before exiting the ISR. |

**Returns**

If a task was removed from the Blocked state then pdTRUE is returned. Otherwise pdFALSE is returned.

## 5.30 xMessageBufferReceiveCompletedFromISR

**message_buffer.h** (p. **??**)

```
BaseType_t xMessageBufferReceiveCompletedFromISR( MessageBufferHandle_t xStreamBuffer, BaseType_t
```

For advanced users only.

The sbRECEIVE_COMPLETED() macro is called from within the FreeRTOS APIs when data is read out of a message buffer or stream buffer. If there was a task that was blocked on the message or stream buffer waiting for data to arrive then the sbRECEIVE_COMPLETED() macro sends a notification to the task to remove it from the Blocked state. xMessageBufferReceiveCompletedFromISR() does the same thing. It is provided to enable application writers to implement their own version of sbRECEIVE_COMPLETED(), and MUST NOT BE USED AT ANY OTHER TIME.

See the example implemented in FreeRTOS/Demo/Minimal/MessageBufferAMP.c for additional information.

**Parameters**

| | |
|---|---|
| *xStreamBuffer* | The handle of the stream buffer from which data was read. |
| *pxHigherPriorityTaskWoken* | ∗pxHigherPriorityTaskWoken should be initialised to pdFALSE before it is passed into xMessageBufferReceiveCompletedFromISR(). If calling xMessageBufferReceiveCompletedFromISR() removes a task from the Blocked state, and the task has a priority above the priority of the currently running task, then ∗pxHigherPriorityTaskWoken will get set to pdTRUE indicating that a context switch should be performed before exiting the ISR. |

**Returns**

If a task was removed from the Blocked state then pdTRUE is returned. Otherwise pdFALSE is returned.

## 5.31 xQueueCreate

queue. h

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

Creates a new queue instance, and returns a handle by which the new queue can be referenced.

Internally, within the FreeRTOS implementation, queues use two blocks of memory. The first block is used to hold the queue's data structures. The second block is used to hold items placed into the queue. If a queue is created using xQueueCreate() then both blocks of memory are automatically dynamically allocated inside the x↩ QueueCreate() function. (see `https://www.FreeRTOS.org/a00111.html`). If a queue is created using xQueueCreateStatic() then the application writer must provide the memory that will get used by the queue. x↩ QueueCreateStatic() therefore allows a queue to be created without using any dynamic memory allocation.

`https://www.FreeRTOS.org/Embedded-RTOS-Queues.html`

**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 uint32_t values.
 xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );
 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.
}
```

## 5.32 xQueueCreateStatic

queue. h

```
 QueueHandle_t xQueueCreateStatic(
                           UBaseType_t uxQueueLength,
                           UBaseType_t uxItemSize,
                           uint8_t *pucQueueStorageBuffer,
                           StaticQueue_t *pxQueueBuffer
                       );
```

Creates a new queue instance, and returns a handle by which the new queue can be referenced.

Internally, within the FreeRTOS implementation, queues use two blocks of memory. The first block is used to hold the queue's data structures. The second block is used to hold items placed into the queue. If a queue is created using xQueueCreate() then both blocks of memory are automatically dynamically allocated inside the x↩ QueueCreate() function. (see `https://www.FreeRTOS.org/a00111.html`). If a queue is created using xQueueCreateStatic() then the application writer must provide the memory that will get used by the queue. x↩ QueueCreateStatic() therefore allows a queue to be created without using any dynamic memory allocation.

`https://www.FreeRTOS.org/Embedded-RTOS-Queues.html`

**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. |
| *pucQueueStorageBuffer* | If uxItemSize is not zero then pucQueueStorageBuffer must point to a uint8_t array that is at least large enough to hold the maximum number of items that can be in the queue at any one time - which is ( uxQueueLength * uxItemsSize ) bytes. If uxItemSize is zero then pucQueueStorageBuffer can be NULL. |
| *pxQueueBuffer* | Must point to a variable of type StaticQueue_t, which will be used to hold the queue's data structure. |

**Returns**

If the queue is created then a handle to the created queue is returned. If pxQueueBuffer is NULL then NULL is returned.

Example usage:

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

#define QUEUE_LENGTH 10
#define ITEM_SIZE sizeof( uint32_t )

 // xQueueBuffer will hold the queue structure.
 StaticQueue_t xQueueBuffer;
```

```
// ucQueueStorage will hold the items posted to the queue.  Must be at least
// [(queue length) * ( queue item size)] bytes long.
uint8_t ucQueueStorage[ QUEUE_LENGTH * ITEM_SIZE ];

void vATask( void *pvParameters )
{
QueueHandle_t xQueue1;

 // Create a queue capable of containing 10 uint32_t values.
 xQueue1 = xQueueCreate( QUEUE_LENGTH, // The number of items the queue can hold.
                         ITEM_SIZE     // The size of each item in the queue
                         &( ucQueueStorage[ 0 ] ), // The buffer that will hold the items in the
                         &xQueueBuffer ); // The buffer that will hold the queue structure.

 // The queue is guaranteed to be created successfully as no dynamic memory
 // allocation is used.  Therefore xQueue1 is now a handle to a valid queue.

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

## 5.33 xQueueSend

queue. h

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

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

**Parameters**

| | |
|---|---|
| *xQueue* | The handle to the queue on which the item is to be posted. |
| *pvItemToQueue* | A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area. |
| *xTicksToWait* | The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0 and the queue is full. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required. |

**Returns**

pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

Example usage:

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

uint32_t ulVar = 10UL;

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

 // Create a queue capable of containing 10 uint32_t values.
 xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );

 // 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 uint32_t.  Wait for 10 ticks for space to become
     // available if necessary.
     if( xQueueSendToFront( xQueue1, ( void * ) &ulVar, ( TickType_t ) 10 ) != pdPASS )
     {
         // Failed to post the message, even after 10 ticks.
     }
 }

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

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

queue. h

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

This is a macro that calls xQueueGenericSend().

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

**Parameters**

| | |
|---|---|
| *xQueue* | The handle to the queue on which the item is to be posted. |
| *pvItemToQueue* | A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area. |
| *xTicksToWait* | The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0 and the queue is full. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required. |

**Returns**

pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

Example usage:

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

uint32_t ulVar = 10UL;

void vATask( void *pvParameters )
```

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

 // Create a queue capable of containing 10 uint32_t values.
 xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );

 // 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 uint32_t.  Wait for 10 ticks for space to become
     // available if necessary.
     if( xQueueSendToBack( xQueue1, ( void * ) &ulVar, ( TickType_t ) 10 ) != pdPASS )
     {
         // Failed to post the message, even after 10 ticks.
     }
 }

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

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

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 Free↩ RTOS.org that did not include the xQueueSendToFront() and xQueueSendToBack() macros. It is equivalent to xQueueSendToBack().

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

**Parameters**

| | |
|---|---|
| *xQueue* | The handle to the queue on which the item is to be posted. |
| *pvItemToQueue* | A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area. |
| *xTicksToWait* | The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0 and the queue is full. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required. |

**Returns**

pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

Example usage:

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

uint32_t ulVar = 10UL;

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

 // Create a queue capable of containing 10 uint32_t values.
 xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );

 // 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 uint32_t.  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.
}
```

queue. h

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

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

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

**Parameters**

| | |
|---|---|
| *xQueue* | The handle to the queue on which the item is to be posted. |
| *pvItemToQueue* | A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area. |
| *xTicksToWait* | The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0 and the queue is full. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required. |
| *xCopyPosition* | Can take the value queueSEND_TO_BACK to place the item at the back of the queue, or queueSEND_TO_FRONT to place the item at the front of the queue (for high priority messages). |

**Returns**

pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

Example usage:

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

uint32_t ulVar = 10UL;

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

 // Create a queue capable of containing 10 uint32_t values.
 xQueue1 = xQueueCreate( 10, sizeof( uint32_t ) );

 // 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 uint32_t.  Wait for 10 ticks for space to become
     // available if necessary.
     if( xQueueGenericSend( xQueue1, ( void * ) &ulVar, ( TickType_t ) 10, queueSEND_TO_BACK ) !=
     {
         // Failed to post the message, even after 10 ticks.
     }
 }

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

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

## 5.34 xQueueOverwrite

queue. h

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

Only for use with queues that have a length of one - so the queue is either empty or full.

Post an item on a queue. If the queue is already full then overwrite the value held in the queue. The item is queued by copy, not by reference.

This function must not be called from an interrupt service routine. See xQueueOverwriteFromISR () for an alternative which may be used in an ISR.

**Parameters**

| | |
|---|---|
| *xQueue* | The handle of the queue to which the data is being sent. |
| *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. |

**Returns**

xQueueOverwrite() is a macro that calls xQueueGenericSend(), and therefore has the same return values as xQueueSendToFront(). However, pdPASS is the only value that can be returned because xQueueOverwrite() will write to the queue even when the queue is already full.

Example usage:

```
void vFunction( void *pvParameters )
{
QueueHandle_t xQueue;
uint32_t ulVarToSend, ulValReceived;

 // Create a queue to hold one uint32_t value.  It is strongly
 // recommended *not* to use xQueueOverwrite() on queues that can
 // contain more than one value, and doing so will trigger an assertion
 // if configASSERT() is defined.
 xQueue = xQueueCreate( 1, sizeof( uint32_t ) );

 // Write the value 10 to the queue using xQueueOverwrite().
 ulVarToSend = 10;
 xQueueOverwrite( xQueue, &ulVarToSend );

 // Peeking the queue should now return 10, but leave the value 10 in
 // the queue.  A block time of zero is used as it is known that the
 // queue holds a value.
 ulValReceived = 0;
 xQueuePeek( xQueue, &ulValReceived, 0 );

 if( ulValReceived != 10 )
 {
     // Error unless the item was removed by a different task.
 }

 // The queue is still full.  Use xQueueOverwrite() to overwrite the
 // value held in the queue with 100.
 ulVarToSend = 100;
 xQueueOverwrite( xQueue, &ulVarToSend );

 // This time read from the queue, leaving the queue empty once more.
 // A block time of 0 is used again.
 xQueueReceive( xQueue, &ulValReceived, 0 );

 // The value read should be the last value written, even though the
 // queue was already full when the value was written.
 if( ulValReceived != 100 )
 {
     // Error!
 }

 // ...
}
```

## 5.35 xQueuePeek

queue. h

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

Receive an item from a queue without removing the item from the queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

Successfully received items remain on the queue so will be returned again by the next call, or a call to xQueue↩Receive().

This macro must not be used in an interrupt service routine. See xQueuePeekFromISR() for an alternative that can be called from an interrupt service routine.

**Parameters**

| | |
|---|---|
| *xQueue* | The handle to the queue from which the item is to be received. |
| *pvBuffer* | Pointer to the buffer into which the received item will be copied. |
| *xTicksToWait* | The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call. The time is defined in tick periods so the constant portTICK_PERIOD_MS should be used to convert to real time if this is required. xQueuePeek() will return immediately if xTicksToWait is 0 and the queue is empty. |

**Returns**

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

Example usage:

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

QueueHandle_t xQueue;

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

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

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

 // ... Rest of task code.
}

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

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

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

## 5.36 xQueuePeekFromISR

queue. h

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

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

Receive an item from a queue without removing the item from the queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

Successfully received items remain on the queue so will be returned again by the next call, or a call to xQueue←↩ Receive().

**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. |

**Returns**

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

## 5.37 xQueueReceive

queue. h

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

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

Successfully received items are removed from the queue.

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

**Parameters**

| | |
|---|---|
| *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. xQueueReceive() will return immediately if xTicksToWait is zero and 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. |

**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.
}
```

## 5.38 uxQueueMessagesWaiting

queue. h

```
UBaseType_t uxQueueMessagesWaiting( const QueueHandle_t xQueue );
```

Return the number of messages stored in a queue.

**Parameters**

| | |
|---|---|
| *xQueue* | A handle to the queue being queried. |

**Returns**

The number of messages available in the queue.

queue. h

```
UBaseType_t uxQueueSpacesAvailable( const QueueHandle_t xQueue );
```

Return the number of free spaces available in a queue. This is equal to the number of items that can be sent to the queue before the queue becomes full if no items are removed.

**Parameters**

| | |
|---|---|
| *xQueue* | A handle to the queue being queried. |

**Returns**

The number of spaces available in the queue.

## 5.39 vQueueDelete

queue. h

```
void vQueueDelete( QueueHandle_t xQueue );
```

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

**Parameters**

| | |
|---|---|
| *xQueue* | A handle to the queue to be deleted. |

## 5.40 xQueueSendFromISR

queue. h

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

This is a macro that calls xQueueGenericSendFromISR().

Post an item to the front of a queue. It is safe to use this macro from within an interrupt service routine.

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

**Parameters**

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

**Returns**

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

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

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

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

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

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

 } while( portINPUT_BYTE( BUFFER_COUNT ) );

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

queue. h

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

This is a macro that calls xQueueGenericSendFromISR().

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

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

**Parameters**

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

**Returns**

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

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

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

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

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

     // Post the byte.
     xQueueSendToBackFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );
 } while( portINPUT_BYTE( BUFFER_COUNT ) );

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

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.org that did not include the xQueueSendToBackFromISR() and xQueueSendToFrontFromISR() macros.

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

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

**Parameters**

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

**Returns**

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

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

```
void vBufferISR( void )
{
char cIn;
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.
     portYIELD_FROM_ISR ();
 }
}
```

queue. h

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

It is preferred that the macros xQueueSendFromISR(), xQueueSendToFrontFromISR() and xQueueSendToBack←FromISR() be used in place of calling this function directly. xQueueGiveFromISR() is an equivalent for use by semaphores that don't actually copy any data.

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

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

**Parameters**

| | |
|---|---|
| *xQueue* | The handle to the queue on which the item is to be posted. |
| *pvItemToQueue* | A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area. |
| *pxHigherPriorityTaskWoken* | xQueueGenericSendFromISR() will set ∗pxHigherPriorityTaskWoken to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xQueueGenericSendFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited. |
| *xCopyPosition* | Can take the value queueSEND_TO_BACK to place the item at the back of the queue, or queueSEND_TO_FRONT to place the item at the front of the queue (for high priority messages). |

**Returns**

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

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

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

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

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

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

 } while( portINPUT_BYTE( BUFFER_COUNT ) );

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

## 5.41 xQueueOverwriteFromISR

queue. h

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

A version of xQueueOverwrite() that can be used in an interrupt service routine (ISR).

Only for use with queues that can hold a single item - so the queue is either empty or full.

Post an item on a queue. If the queue is already full then overwrite the value held in the queue. The item is queued by copy, not by reference.

**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* | xQueueOverwriteFromISR() 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 xQueueOverwriteFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited. |

**Returns**

xQueueOverwriteFromISR() is a macro that calls xQueueGenericSendFromISR(), and therefore has the same return values as xQueueSendToFrontFromISR(). However, pdPASS is the only value that can be returned because xQueueOverwriteFromISR() will write to the queue even when the queue is already full.

Example usage:

```
QueueHandle_t xQueue;

void vFunction( void *pvParameters )
{
 // Create a queue to hold one uint32_t value.  It is strongly
 // recommended *not* to use xQueueOverwriteFromISR() on queues that can
 // contain more than one value, and doing so will trigger an assertion
 // if configASSERT() is defined.
 xQueue = xQueueCreate( 1, sizeof( uint32_t ) );
}

void vAnInterruptHandler( void )
{
// xHigherPriorityTaskWoken must be set to pdFALSE before it is used.
BaseType_t xHigherPriorityTaskWoken = pdFALSE;
uint32_t ulVarToSend, ulValReceived;
```

```
 // Write the value 10 to the queue using xQueueOverwriteFromISR().
 ulVarToSend = 10;
 xQueueOverwriteFromISR( xQueue, &ulVarToSend, &xHigherPriorityTaskWoken );

 // The queue is full, but calling xQueueOverwriteFromISR() again will still
 // pass because the value held in the queue will be overwritten with the
 // new value.
 ulVarToSend = 100;
 xQueueOverwriteFromISR( xQueue, &ulVarToSend, &xHigherPriorityTaskWoken );

 // Reading from the queue will now return 100.

 // ...

 if( xHigherPrioritytaskWoken == pdTRUE )
 {
     // Writing to the queue caused a task to unblock and the unblocked task
     // has a priority higher than or equal to the priority of the currently
     // executing task (the task this interrupt interrupted).  Perform a context
     // switch so this interrupt returns directly to the unblocked task.
     portYIELD_FROM_ISR(); // or portEND_SWITCHING_ISR() depending on the port.
 }
}
```

## 5.42 xQueueReceiveFromISR

queue. h

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

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

**Parameters**

| | |
|---|---|
| *xQueue* | The handle to the queue from which the item is to be received. |
| *pvBuffer* | Pointer to the buffer into which the received item will be copied. |
| *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:

```
QueueHandle_t xQueue;

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

 // Create a queue capable of containing 10 characters.
 xQueue = xQueueCreate( 10, sizeof( char ) );
 if( xQueue == 0 )
 {
     // Failed to create the queue.
 }

 // ...

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

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

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

```
}

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

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

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

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

## 5.43 vSemaphoreCreateBinary

semphr. h

```
vSemaphoreCreateBinary( SemaphoreHandle_t xSemaphore );
```

In many usage scenarios it is faster and more memory efficient to use a direct to task notification in place of a binary semaphore! `https://www.FreeRTOS.org/RTOS-task-notifications.html`

This old vSemaphoreCreateBinary() macro is now deprecated in favour of the xSemaphoreCreateBinary() function. Note that binary semaphores created using the vSemaphoreCreateBinary() macro are created in a state such that the first call to 'take' the semaphore would pass, whereas binary semaphores created using xSemaphoreCreate↩ Binary() are created in a state such that the the semaphore must first be 'given' before it can be 'taken'.

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

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

**Parameters**

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

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;

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.
 }
}
```

## 5.44 xSemaphoreCreateBinary

semphr. h

```
SemaphoreHandle_t xSemaphoreCreateBinary( void );
```

Creates a new binary semaphore instance, and returns a handle by which the new semaphore can be referenced.

In many usage scenarios it is faster and more memory efficient to use a direct to task notification in place of a binary semaphore! `https://www.FreeRTOS.org/RTOS-task-notifications.html`

Internally, within the FreeRTOS implementation, binary semaphores use a block of memory, in which the semaphore structure is stored. If a binary semaphore is created using xSemaphoreCreateBinary() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateBinary() function. (see `https://www.FreeRTOS.org/a00111.html`). If a binary semaphore is created using xSemaphoreCreateBinaryStatic() then the application writer must provide the memory. xSemaphoreCreateBinaryStatic() therefore allows a binary semaphore to be created without using any dynamic memory allocation.

The old vSemaphoreCreateBinary() macro is now deprecated in favour of this xSemaphoreCreateBinary() function. Note that binary semaphores created using the vSemaphoreCreateBinary() macro are created in a state such that the first call to 'take' the semaphore would pass, whereas binary semaphores created using xSemaphoreCreateBinary() are created in a state such that the the semaphore must first be 'given' before it can be 'taken'.

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

**Returns**

Handle to the created semaphore, or NULL if the memory required to hold the semaphore's data structures could not be allocated.

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;

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

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

## 5.45 xSemaphoreCreateBinaryStatic

semphr. h

```
SemaphoreHandle_t xSemaphoreCreateBinaryStatic( StaticSemaphore_t *pxSemaphoreBuffer );
```

Creates a new binary semaphore instance, and returns a handle by which the new semaphore can be referenced.

NOTE: In many usage scenarios it is faster and more memory efficient to use a direct to task notification in place of a binary semaphore! `https://www.FreeRTOS.org/RTOS-task-notifications.html`

Internally, within the FreeRTOS implementation, binary semaphores use a block of memory, in which the semaphore structure is stored. If a binary semaphore is created using xSemaphoreCreateBinary() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateBinary() function. (see `https://www.↩ FreeRTOS.org/a00111.html`). If a binary semaphore is created using xSemaphoreCreateBinaryStatic() then the application writer must provide the memory. xSemaphoreCreateBinaryStatic() therefore allows a binary semaphore to be created without using any dynamic memory allocation.

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

**Parameters**

| | |
|---|---|
| *pxSemaphoreBuffer* | Must point to a variable of type StaticSemaphore_t, which will then be used to hold the semaphore's data structure, removing the need for the memory to be allocated dynamically. |

**Returns**

If the semaphore is created then a handle to the created semaphore is returned. If pxSemaphoreBuffer is NULL then NULL is returned.

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;
StaticSemaphore_t xSemaphoreBuffer;

void vATask( void * pvParameters )
{
 // Semaphore cannot be used before a call to xSemaphoreCreateBinary().
 // The semaphore's data structures will be placed in the xSemaphoreBuffer
 // variable, the address of which is passed into the function.  The
 // function's parameter is not NULL, so the function will not attempt any
 // dynamic memory allocation, and therefore the function will not return
 // return NULL.
 xSemaphore = xSemaphoreCreateBinary( &xSemaphoreBuffer );

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

## 5.46 xSemaphoreTake

semphr. h

```
xSemaphoreTake(
                  SemaphoreHandle_t xSemaphore,
                  TickType_t xBlockTime
              );
```

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

**Parameters**

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

**Returns**

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

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;

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

// 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.
        }
    }
}
```

## 5.47 xSemaphoreTakeRecursive

semphr. h

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

*Macro* to recursively obtain, or 'take', a mutex type semaphore. The mutex must have previously been created using a call to xSemaphoreCreateRecursiveMutex();

configUSE_RECURSIVE_MUTEXES must be set to 1 in FreeRTOSConfig.h for this macro to be available.

This macro must not be used on mutexes created using xSemaphoreCreateMutex().

A mutex used recursively can be 'taken' repeatedly by the owner. The mutex doesn't become available again until the owner has called xSemaphoreGiveRecursive() for each successful 'take' request. For example, if a task successfully 'takes' the same mutex 5 times then the mutex will not be available to any other task until it has also 'given' the mutex back exactly five times.

**Parameters**

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

**Returns**

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

Example usage:

```
SemaphoreHandle_t xMutex = NULL;

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

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

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

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

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

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

## 5.48 xSemaphoreGive

semphr. h

```
xSemaphoreGive( SemaphoreHandle_t xSemaphore );
```

*Macro* to release a semaphore. The semaphore must have previously been created with a call to xSemaphore↩ CreateBinary(), xSemaphoreCreateMutex() or xSemaphoreCreateCounting(). and obtained using sSemaphore↩ Take().

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

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

**Parameters**

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

**Returns**

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

Example usage:

```
SemaphoreHandle_t xSemaphore = NULL;

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

 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.
            }
        }
    }
}
```

## 5.49 xSemaphoreGiveRecursive

semphr. h

```
xSemaphoreGiveRecursive( SemaphoreHandle_t xMutex );
```

*Macro* to recursively release, or 'give', a mutex type semaphore. The mutex must have previously been created using a call to xSemaphoreCreateRecursiveMutex();

configUSE_RECURSIVE_MUTEXES must be set to 1 in FreeRTOSConfig.h for this macro to be available.

This macro must not be used on mutexes created using xSemaphoreCreateMutex().

A mutex used recursively can be 'taken' repeatedly by the owner. The mutex doesn't become available again until the owner has called xSemaphoreGiveRecursive() for each successful 'take' request. For example, if a task successfully 'takes' the same mutex 5 times then the mutex will not be available to any other task until it has also 'given' the mutex back exactly five times.

**Parameters**

| | |
|---|---|
| *xMutex* | A handle to the mutex being released, or 'given'. This is the handle returned by xSemaphoreCreateMutex(); |

**Returns**

pdTRUE if the semaphore was given.

Example usage:

```
SemaphoreHandle_t xMutex = NULL;

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

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

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

         // ...
         // For some reason due to the nature of the code further calls to
         // xSemaphoreTakeRecursive() are made on the same mutex.  In real
         // code these would not be just sequential calls as this would make
```

```
            // no sense.  Instead the calls are likely to be buried inside
            // a more complex call structure.
            xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );
            xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );

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

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

## 5.50 xSemaphoreGiveFromISR

semphr. h

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

*Macro* to release a semaphore. The semaphore must have previously been created with a call to xSemaphore↩ CreateBinary() or xSemaphoreCreateCounting().

Mutex type semaphores (those created using a call to xSemaphoreCreateMutex()) must not be used with this macro.

This macro can be used from an ISR.

**Parameters**

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

**Returns**

pdTRUE if the semaphore was successfully given, otherwise errQUEUE_FULL.

Example usage:

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

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

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

           // ...

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

```
        }
    }
}

// Timer ISR
void vTimerISR( void * pvParameters )
{
static uint8_t ucLocalTickCount = 0;
static BaseType_t xHigherPriorityTaskWoken;

 // A timer tick has occurred.

 // ... Do other time functions.

 // Is it time for vATask () to run?
 xHigherPriorityTaskWoken = pdFALSE;
 ucLocalTickCount++;
 if( ucLocalTickCount >= TICKS_TO_WAIT )
 {
     // Unblock the task by releasing the semaphore.
     xSemaphoreGiveFromISR( xSemaphore, &xHigherPriorityTaskWoken );

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

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

## 5.51 xSemaphoreCreateMutex

semphr. h

```
SemaphoreHandle_t xSemaphoreCreateMutex( void );
```

Creates a new mutex type semaphore instance, and returns a handle by which the new mutex can be referenced.

Internally, within the FreeRTOS implementation, mutex semaphores use a block of memory, in which the mutex structure is stored. If a mutex is created using xSemaphoreCreateMutex() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateMutex() function. (see `https://www.Free↩ RTOS.org/a00111.html`). If a mutex is created using xSemaphoreCreateMutexStatic() then the application writer must provided the memory. xSemaphoreCreateMutexStatic() therefore allows a mutex to be created without using any dynamic memory allocation.

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

This type of semaphore uses a priority inheritance mechanism so a task 'taking' a semaphore MUST ALWAYS 'give' the semaphore back once the semaphore it is no longer required.

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

See xSemaphoreCreateBinary() for an alternative implementation that can be used for pure synchronisation (where one task or interrupt always 'gives' the semaphore and another always 'takes' the semaphore) and from within interrupt service routines.

**Returns**

If the mutex was successfully created then a handle to the created semaphore is returned. If there was not enough heap to allocate the mutex data structures then NULL is returned.

Example usage:

```
SemaphoreHandle_t xSemaphore;

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

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

## 5.52 xSemaphoreCreateMutexStatic

semphr. h

```
SemaphoreHandle_t xSemaphoreCreateMutexStatic( StaticSemaphore_t *pxMutexBuffer );
```

Creates a new mutex type semaphore instance, and returns a handle by which the new mutex can be referenced.

Internally, within the FreeRTOS implementation, mutex semaphores use a block of memory, in which the mutex structure is stored. If a mutex is created using xSemaphoreCreateMutex() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateMutex() function. (see `https://www.FreeRTOS.org/a00111.html`). If a mutex is created using xSemaphoreCreateMutexStatic() then the application writer must provided the memory. xSemaphoreCreateMutexStatic() therefore allows a mutex to be created without using any dynamic memory allocation.

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

This type of semaphore uses a priority inheritance mechanism so a task 'taking' a semaphore MUST ALWAYS 'give' the semaphore back once the semaphore it is no longer required.

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

See xSemaphoreCreateBinary() for an alternative implementation that can be used for pure synchronisation (where one task or interrupt always 'gives' the semaphore and another always 'takes' the semaphore) and from within interrupt service routines.

**Parameters**

| | |
|---|---|
| *pxMutexBuffer* | Must point to a variable of type StaticSemaphore_t, which will be used to hold the mutex's data structure, removing the need for the memory to be allocated dynamically. |

**Returns**

If the mutex was successfully created then a handle to the created mutex is returned. If pxMutexBuffer was NULL then NULL is returned.

Example usage:

```
SemaphoreHandle_t xSemaphore;
StaticSemaphore_t xMutexBuffer;

void vATask( void * pvParameters )
{
 // A mutex cannot be used before it has been created.  xMutexBuffer is
 // into xSemaphoreCreateMutexStatic() so no dynamic memory allocation is
 // attempted.
 xSemaphore = xSemaphoreCreateMutexStatic( &xMutexBuffer );

 // As no dynamic memory allocation was performed, xSemaphore cannot be NULL,
 // so there is no need to check it.
}
```

## 5.53 xSemaphoreCreateRecursiveMutex

semphr. h

```
SemaphoreHandle_t xSemaphoreCreateRecursiveMutex( void );
```

Creates a new recursive mutex type semaphore instance, and returns a handle by which the new recursive mutex can be referenced.

Internally, within the FreeRTOS implementation, recursive mutexs use a block of memory, in which the mutex structure is stored. If a recursive mutex is created using xSemaphoreCreateRecursiveMutex() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateRecursiveMutex() function. (see `https://www.FreeRTOS.org/a00111.html`). If a recursive mutex is created using xSemaphoreCreate↩ RecursiveMutexStatic() then the application writer must provide the memory that will get used by the mutex. x↩ SemaphoreCreateRecursiveMutexStatic() therefore allows a recursive mutex to be created without using any dynamic memory allocation.

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

A mutex used recursively can be 'taken' repeatedly by the owner. The mutex doesn't become available again until the owner has called xSemaphoreGiveRecursive() for each successful 'take' request. For example, if a task successfully 'takes' the same mutex 5 times then the mutex will not be available to any other task until it has also 'given' the mutex back exactly five times.

This type of semaphore uses a priority inheritance mechanism so a task 'taking' a semaphore MUST ALWAYS 'give' the semaphore back once the semaphore it is no longer required.

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

See xSemaphoreCreateBinary() for an alternative implementation that can be used for pure synchronisation (where one task or interrupt always 'gives' the semaphore and another always 'takes' the semaphore) and from within interrupt service routines.

**Returns**

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

Example usage:

```
SemaphoreHandle_t xSemaphore;

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

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

## 5.54 xSemaphoreCreateRecursiveMutexStatic

semphr. h

```
SemaphoreHandle_t xSemaphoreCreateRecursiveMutexStatic( StaticSemaphore_t *pxMutexBuffer );
```

Creates a new recursive mutex type semaphore instance, and returns a handle by which the new recursive mutex can be referenced.

Internally, within the FreeRTOS implementation, recursive mutexs use a block of memory, in which the mutex structure is stored. If a recursive mutex is created using xSemaphoreCreateRecursiveMutex() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateRecursiveMutex() function. (see `https://www.FreeRTOS.org/a00111.html`). If a recursive mutex is created using xSemaphoreCreate←RecursiveMutexStatic() then the application writer must provide the memory that will get used by the mutex. x←SemaphoreCreateRecursiveMutexStatic() therefore allows a recursive mutex to be created without using any dynamic memory allocation.

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

A mutex used recursively can be 'taken' repeatedly by the owner. The mutex doesn't become available again until the owner has called xSemaphoreGiveRecursive() for each successful 'take' request. For example, if a task successfully 'takes' the same mutex 5 times then the mutex will not be available to any other task until it has also 'given' the mutex back exactly five times.

This type of semaphore uses a priority inheritance mechanism so a task 'taking' a semaphore MUST ALWAYS 'give' the semaphore back once the semaphore it is no longer required.

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

See xSemaphoreCreateBinary() for an alternative implementation that can be used for pure synchronisation (where one task or interrupt always 'gives' the semaphore and another always 'takes' the semaphore) and from within interrupt service routines.

**Parameters**

| | |
|---|---|
| *pxMutexBuffer* | Must point to a variable of type StaticSemaphore_t, which will then be used to hold the recursive mutex's data structure, removing the need for the memory to be allocated dynamically. |

**Returns**

If the recursive mutex was successfully created then a handle to the created recursive mutex is returned. If pxMutexBuffer was NULL then NULL is returned.

Example usage:

```
SemaphoreHandle_t xSemaphore;
StaticSemaphore_t xMutexBuffer;

void vATask( void * pvParameters )
{
 // A recursive semaphore cannot be used before it is created.  Here a
 // recursive mutex is created using xSemaphoreCreateRecursiveMutexStatic().
 // The address of xMutexBuffer is passed into the function, and will hold
 // the mutexes data structures - so no dynamic memory allocation will be
 // attempted.
 xSemaphore = xSemaphoreCreateRecursiveMutexStatic( &xMutexBuffer );

 // As no dynamic memory allocation was performed, xSemaphore cannot be NULL,
 // so there is no need to check it.
}
```

## 5.55 xSemaphoreCreateCounting

semphr. h

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

Creates a new counting semaphore instance, and returns a handle by which the new counting semaphore can be referenced.

In many usage scenarios it is faster and more memory efficient to use a direct to task notification in place of a counting semaphore! `https://www.FreeRTOS.org/RTOS-task-notifications.html`

Internally, within the FreeRTOS implementation, counting semaphores use a block of memory, in which the counting semaphore structure is stored. If a counting semaphore is created using xSemaphoreCreateCounting() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateCounting() function. (see `https://www.FreeRTOS.org/a00111.html`). If a counting semaphore is created using xSemaphore←CreateCountingStatic() then the application writer can instead optionally provide the memory that will get used by the counting semaphore. xSemaphoreCreateCountingStatic() therefore allows a counting semaphore to be created without using any dynamic memory allocation.

Counting semaphores are typically used for two things:

1) Counting events.

In this usage scenario an event handler will 'give' a semaphore each time an event occurs (incrementing the semaphore count value), and a handler task will 'take' a semaphore each time it processes an event (decrementing the semaphore count value). The count value is therefore the difference between the number of events that have occurred and the number that have been processed. In this case it is desirable for the initial count value to be zero.

2) Resource management.

In this usage scenario the count value indicates the number of resources available. To obtain control of a resource a task must first obtain a semaphore - decrementing the semaphore count value. When the count value reaches zero there are no free resources. When a task finishes with the resource it 'gives' the semaphore back - incrementing the semaphore count value. In this case it is desirable for the initial count value to be equal to the maximum count value, indicating that all resources are free.

**Parameters**

| | |
|---|---|
| *uxMaxCount* | The maximum count value that can be reached. When the semaphore reaches this value it can no longer be 'given'. |
| *uxInitialCount* | The count value assigned to the semaphore when it is created. |

**Returns**

Handle to the created semaphore. Null if the semaphore could not be created.

Example usage:

```
SemaphoreHandle_t xSemaphore;

void vATask( void * pvParameters )
{
```

```
SemaphoreHandle_t xSemaphore = NULL;

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

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

## 5.56 xSemaphoreCreateCountingStatic

semphr. h

```
SemaphoreHandle_t xSemaphoreCreateCountingStatic( UBaseType_t uxMaxCount, UBaseType_t uxInitialCou
```

Creates a new counting semaphore instance, and returns a handle by which the new counting semaphore can be referenced.

In many usage scenarios it is faster and more memory efficient to use a direct to task notification in place of a counting semaphore! `https://www.FreeRTOS.org/RTOS-task-notifications.html`

Internally, within the FreeRTOS implementation, counting semaphores use a block of memory, in which the counting semaphore structure is stored. If a counting semaphore is created using xSemaphoreCreateCounting() then the required memory is automatically dynamically allocated inside the xSemaphoreCreateCounting() function. (see `https://www.FreeRTOS.org/a00111.html`). If a counting semaphore is created using xSemaphore←CreateCountingStatic() then the application writer must provide the memory. xSemaphoreCreateCountingStatic() therefore allows a counting semaphore to be created without using any dynamic memory allocation.

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. |
| *pxSemaphoreBuffer* | Must point to a variable of type StaticSemaphore_t, which will then be used to hold the semaphore's data structure, removing the need for the memory to be allocated dynamically. |

**Returns**

If the counting semaphore was successfully created then a handle to the created counting semaphore is returned. If pxSemaphoreBuffer was NULL then NULL is returned.

Example usage:

```
SemaphoreHandle_t xSemaphore;
```

```
StaticSemaphore_t xSemaphoreBuffer;

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

 // Counting semaphore cannot be used before they have been created.  Create
 // a counting semaphore using xSemaphoreCreateCountingStatic().  The max
 // value to which the semaphore can count is 10, and the initial value
 // assigned to the count will be 0.  The address of xSemaphoreBuffer is
 // passed in and will be used to hold the semaphore structure, so no dynamic
 // memory allocation will be used.
 xSemaphore = xSemaphoreCreateCounting( 10, 0, &xSemaphoreBuffer );

 // No memory allocation was attempted so xSemaphore cannot be NULL, so there
 // is no need to check its value.
}
```

## 5.57 vSemaphoreDelete

semphr. h

```
void vSemaphoreDelete( SemaphoreHandle_t xSemaphore );
```

Delete a semaphore. This function must be used with care. For example, do not delete a mutex type semaphore if the mutex is held by a task.

**Parameters**

| | |
|---|---|
| *xSemaphore* | A handle to the semaphore to be deleted. |

## 5.58 xStreamBufferCreate

**message_buffer.h** (p. **??**)

```
StreamBufferHandle_t xStreamBufferCreate( size_t xBufferSizeBytes, size_t xTriggerLevelBytes );
```

Creates a new stream buffer using dynamically allocated memory. See xStreamBufferCreateStatic() for a version that uses statically allocated memory (memory that is allocated at compile time).

configSUPPORT_DYNAMIC_ALLOCATION must be set to 1 or left undefined in FreeRTOSConfig.h for xStream↩ BufferCreate() to be available.

**Parameters**

| | |
|---|---|
| *xBufferSizeBytes* | The total number of bytes the stream buffer will be able to hold at any one time. |
| *xTriggerLevelBytes* | The number of bytes that must be in the stream buffer before a task that is blocked on the stream buffer to wait for data is moved out of the blocked state. For example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 1 then the task will be unblocked when a single byte is written to the buffer or the task's block time expires. As another example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 10 then the task will not be unblocked until the stream buffer contains at least 10 bytes or the task's block time expires. If a reading task's block time expires before the trigger level is reached then the task will still receive however many bytes are actually available. Setting a trigger level of 0 will result in a trigger level of 1 being used. It is not valid to specify a trigger level that is greater than the buffer size. |

**Returns**

If NULL is returned, then the stream buffer cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the stream buffer data structures and storage area. A non-NULL value being returned indicates that the stream buffer has been created successfully - the returned value should be stored as the handle to the created stream buffer.

Example use:

```
void vAFunction( void )
{
StreamBufferHandle_t xStreamBuffer;
const size_t xStreamBufferSizeBytes = 100, xTriggerLevel = 10;

 // Create a stream buffer that can hold 100 bytes.  The memory used to hold
 // both the stream buffer structure and the data in the stream buffer is
 // allocated dynamically.
 xStreamBuffer = xStreamBufferCreate( xStreamBufferSizeBytes, xTriggerLevel );

 if( xStreamBuffer == NULL )
 {
     // There was not enough heap memory space available to create the
     // stream buffer.
 }
 else
 {
     // The stream buffer was created successfully and can now be used.
 }
}
```

## 5.59 xStreamBufferCreateStatic

**stream_buffer.h** (p. **??**)

```
  StreamBufferHandle_t xStreamBufferCreateStatic( size_t xBufferSizeBytes,
                                                 size_t xTriggerLevelBytes,
                                                 uint8_t *pucStreamBufferStorageArea,
                                                 StaticStreamBuffer_t *pxStaticStreamBuffer );
```

Creates a new stream buffer using statically allocated memory. See xStreamBufferCreate() for a version that uses dynamically allocated memory.

configSUPPORT_STATIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h for xStreamBufferCreateStatic() to be available.

**Parameters**

| | |
|---|---|
| *xBufferSizeBytes* | The size, in bytes, of the buffer pointed to by the pucStreamBufferStorageArea parameter. |
| *xTriggerLevelBytes* | The number of bytes that must be in the stream buffer before a task that is blocked on the stream buffer to wait for data is moved out of the blocked state. For example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 1 then the task will be unblocked when a single byte is written to the buffer or the task's block time expires. As another example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 10 then the task will not be unblocked until the stream buffer contains at least 10 bytes or the task's block time expires. If a reading task's block time expires before the trigger level is reached then the task will still receive however many bytes are actually available. Setting a trigger level of 0 will result in a trigger level of 1 being used. It is not valid to specify a trigger level that is greater than the buffer size. |
| *pucStreamBufferStorageArea* | Must point to a uint8_t array that is at least xBufferSizeBytes + 1 big. This is the array to which streams are copied when they are written to the stream buffer. |
| *pxStaticStreamBuffer* | Must point to a variable of type StaticStreamBuffer_t, which will be used to hold the stream buffer's data structure. |

**Returns**

If the stream buffer is created successfully then a handle to the created stream buffer is returned. If either pucStreamBufferStorageArea or pxStaticstreamBuffer are NULL then NULL is returned.

Example use:

```
  // Used to dimension the array used to hold the streams.  The available space
  // will actually be one less than this, so 999.
#define STORAGE_SIZE_BYTES 1000

  // Defines the memory that will actually hold the streams within the stream
  // buffer.
  static uint8_t ucStorageBuffer[ STORAGE_SIZE_BYTES ];
```

```
// The variable used to hold the stream buffer structure.
StaticStreamBuffer_t xStreamBufferStruct;

void MyFunction( void )
{
StreamBufferHandle_t xStreamBuffer;
const size_t xTriggerLevel = 1;

 xStreamBuffer = xStreamBufferCreateStatic( sizeof( ucBufferStorage ),
                                            xTriggerLevel,
                                            ucBufferStorage,
                                            &xStreamBufferStruct );

 // As neither the pucStreamBufferStorageArea or pxStaticStreamBuffer
 // parameters were NULL, xStreamBuffer will not be NULL, and can be used to
 // reference the created stream buffer in other stream buffer API calls.

 // Other code that uses the stream buffer can go here.
}
```

## 5.60 xStreamBufferSend

**stream_buffer.h** (p. **??**)

```
size_t xStreamBufferSend( StreamBufferHandle_t xStreamBuffer,
                          const void *pvTxData,
                          size_t xDataLengthBytes,
                          TickType_t xTicksToWait );
```

Sends bytes to a stream buffer. The bytes are copied into the stream buffer.

***NOTE***: Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xStreamBufferSend()) inside a critical section and set the send block time to 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xStreamBufferReceive()) inside a critical section and set the receive block time to 0.

Use xStreamBufferSend() to write to a stream buffer from a task. Use xStreamBufferSendFromISR() to write to a stream buffer from an interrupt service routine (ISR).

**Parameters**

| | |
|---|---|
| *xStreamBuffer* | The handle of the stream buffer to which a stream is being sent. |
| *pvTxData* | A pointer to the buffer that holds the bytes to be copied into the stream buffer. |
| *xDataLengthBytes* | The maximum number of bytes to copy from pvTxData into the stream buffer. |
| *xTicksToWait* | The maximum amount of time the task should remain in the Blocked state to wait for enough space to become available in the stream buffer, should the stream buffer contain too little space to hold the another xDataLengthBytes bytes. The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h. If a task times out before it can write all xDataLengthBytes into the buffer it will still write as many bytes as possible. A task does not use any CPU time when it is in the blocked state. |

**Returns**

The number of bytes written to the stream buffer. If a task times out before it can write all xDataLengthBytes into the buffer it will still write as many bytes as possible.

Example use:

```
void vAFunction( StreamBufferHandle_t xStreamBuffer )
{
size_t xBytesSent;
uint8_t ucArrayToSend[] = { 0, 1, 2, 3 };
char *pcStringToSend = "String to send";
const TickType_t x100ms = pdMS_TO_TICKS( 100 );
```

```
 // Send an array to the stream buffer, blocking for a maximum of 100ms to
 // wait for enough space to be available in the stream buffer.
 xBytesSent = xStreamBufferSend( xStreamBuffer, ( void * ) ucArrayToSend, sizeof( ucArrayToSend ),

 if( xBytesSent != sizeof( ucArrayToSend ) )
 {
     // The call to xStreamBufferSend() times out before there was enough
     // space in the buffer for the data to be written, but it did
     // successfully write xBytesSent bytes.
 }

 // Send the string to the stream buffer.  Return immediately if there is not
 // enough space in the buffer.
 xBytesSent = xStreamBufferSend( xStreamBuffer, ( void * ) pcStringToSend, strlen( pcStringToSend

 if( xBytesSent != strlen( pcStringToSend ) )
 {
     // The entire string could not be added to the stream buffer because
     // there was not enough free space in the buffer, but xBytesSent bytes
     // were sent.  Could try again to send the remaining bytes.
 }
}
```

## 5.61 xStreamBufferSendFromISR

**stream_buffer.h** (p. **??**)

```
size_t xStreamBufferSendFromISR( StreamBufferHandle_t xStreamBuffer,
                                const void *pvTxData,
                                size_t xDataLengthBytes,
                                BaseType_t *pxHigherPriorityTaskWoken );
```

Interrupt safe version of the API function that sends a stream of bytes to the stream buffer.

***NOTE***: Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xStreamBufferSend()) inside a critical section and set the send block time to 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xStreamBufferReceive()) inside a critical section and set the receive block time to 0.

Use xStreamBufferSend() to write to a stream buffer from a task. Use xStreamBufferSendFromISR() to write to a stream buffer from an interrupt service routine (ISR).

**Parameters**

| | |
|---|---|
| *xStreamBuffer* | The handle of the stream buffer to which a stream is being sent. |
| *pvTxData* | A pointer to the data that is to be copied into the stream buffer. |
| *xDataLengthBytes* | The maximum number of bytes to copy from pvTxData into the stream buffer. |
| *pxHigherPriorityTaskWoken* | It is possible that a stream buffer will have a task blocked on it waiting for data. Calling xStreamBufferSendFromISR() can make data available, and so cause a task that was waiting for data to leave the Blocked state. If calling xStreamBufferSendFromISR() causes a task to leave the Blocked state, and the unblocked task has a priority higher than the currently executing task (the task that was interrupted), then, internally, xStreamBufferSendFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE. If xStreamBufferSendFromISR() sets this value to pdTRUE, then normally a context switch should be performed before the interrupt is exited. This will ensure that the interrupt returns directly to the highest priority Ready state task. *pxHigherPriorityTaskWoken should be set to pdFALSE before it is passed into the function. See the example code below for an example. |

**Returns**

The number of bytes actually written to the stream buffer, which will be less than xDataLengthBytes if the stream buffer didn't have enough free space for all the bytes to be written.

Example use:

```
// A stream buffer that has already been created.
StreamBufferHandle_t xStreamBuffer;

void vAnInterruptServiceRoutine( void )
```

```
{
size_t xBytesSent;
char *pcStringToSend = "String to send";
BaseType_t xHigherPriorityTaskWoken = pdFALSE; // Initialised to pdFALSE.

 // Attempt to send the string to the stream buffer.
 xBytesSent = xStreamBufferSendFromISR( xStreamBuffer,
                                        ( void * ) pcStringToSend,
                                        strlen( pcStringToSend ),
                                        &xHigherPriorityTaskWoken );

 if( xBytesSent != strlen( pcStringToSend ) )
 {
     // There was not enough free space in the stream buffer for the entire
     // string to be written, ut xBytesSent bytes were written.
 }

 // If xHigherPriorityTaskWoken was set to pdTRUE inside
 // xStreamBufferSendFromISR() then a task that has a priority above the
 // priority of the currently executing task was unblocked and a context
 // switch should be performed to ensure the ISR returns to the unblocked
 // task.  In most FreeRTOS ports this is done by simply passing
 // xHigherPriorityTaskWoken into taskYIELD_FROM_ISR(), which will test the
 // variables value, and perform the context switch if necessary.  Check the
 // documentation for the port in use for port specific instructions.
 taskYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

## 5.62 xStreamBufferReceive

**stream_buffer.h** (p. **??**)

```
size_t xStreamBufferReceive( StreamBufferHandle_t xStreamBuffer,
                             void *pvRxData,
                             size_t xBufferLengthBytes,
                             TickType_t xTicksToWait );
```

Receives bytes from a stream buffer.

***NOTE***: Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xStreamBufferSend()) inside a critical section and set the send block time to 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xStreamBufferReceive()) inside a critical section and set the receive block time to 0.

Use xStreamBufferReceive() to read from a stream buffer from a task. Use xStreamBufferReceiveFromISR() to read from a stream buffer from an interrupt service routine (ISR).

**Parameters**

| | |
|---|---|
| *xStreamBuffer* | The handle of the stream buffer from which bytes are to be received. |
| *pvRxData* | A pointer to the buffer into which the received bytes will be copied. |
| *xBufferLengthBytes* | The length of the buffer pointed to by the pvRxData parameter. This sets the maximum number of bytes to receive in one call. xStreamBufferReceive will return as many bytes as possible up to a maximum set by xBufferLengthBytes. |
| *xTicksToWait* | The maximum amount of time the task should remain in the Blocked state to wait for data to become available if the stream buffer is empty. xStreamBufferReceive() will return immediately if xTicksToWait is zero. The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h. A task does not use any CPU time when it is in the Blocked state. |

**Returns**

The number of bytes actually read from the stream buffer, which will be less than xBufferLengthBytes if the call to xStreamBufferReceive() timed out before xBufferLengthBytes were available.

Example use:

```
void vAFunction( StreamBuffer_t xStreamBuffer )
{
uint8_t ucRxData[ 20 ];
size_t xReceivedBytes;
const TickType_t xBlockTime = pdMS_TO_TICKS( 20 );
```

```
 // Receive up to another sizeof( ucRxData ) bytes from the stream buffer.
 // Wait in the Blocked state (so not using any CPU processing time) for a
 // maximum of 100ms for the full sizeof( ucRxData ) number of bytes to be
 // available.
 xReceivedBytes = xStreamBufferReceive( xStreamBuffer,
                                        ( void * ) ucRxData,
                                        sizeof( ucRxData ),
                                        xBlockTime );

 if( xReceivedBytes > 0 )
 {
     // A ucRxData contains another xRecievedBytes bytes of data, which can
     // be processed here....
 }
}
```

## 5.63 xStreamBufferReceiveFromISR

**stream_buffer.h** (p. **??**)

```
size_t xStreamBufferReceiveFromISR( StreamBufferHandle_t xStreamBuffer,
                                    void *pvRxData,
                                    size_t xBufferLengthBytes,
                                    BaseType_t *pxHigherPriorityTaskWoken );
```

An interrupt safe version of the API function that receives bytes from a stream buffer.

Use xStreamBufferReceive() to read bytes from a stream buffer from a task. Use xStreamBufferReceiveFromISR() to read bytes from a stream buffer from an interrupt service routine (ISR).

**Parameters**

| | |
|---|---|
| *xStreamBuffer* | The handle of the stream buffer from which a stream is being received. |
| *pvRxData* | A pointer to the buffer into which the received bytes are copied. |
| *xBufferLengthBytes* | The length of the buffer pointed to by the pvRxData parameter. This sets the maximum number of bytes to receive in one call. xStreamBufferReceive will return as many bytes as possible up to a maximum set by xBufferLengthBytes. |
| *pxHigherPriorityTaskWoken* | It is possible that a stream buffer will have a task blocked on it waiting for space to become available. Calling xStreamBufferReceiveFromISR() can make space available, and so cause a task that is waiting for space to leave the Blocked state. If calling xStreamBufferReceiveFromISR() causes a task to leave the Blocked state, and the unblocked task has a priority higher than the currently executing task (the task that was interrupted), then, internally, xStreamBufferReceiveFromISR() will set ∗pxHigherPriorityTaskWoken to pdTRUE. If xStreamBufferReceiveFromISR() sets this value to pdTRUE, then normally a context switch should be performed before the interrupt is exited. That will ensure the interrupt returns directly to the highest priority Ready state task. ∗pxHigherPriorityTaskWoken should be set to pdFALSE before it is passed into the function. See the code example below for an example. |

**Returns**

The number of bytes read from the stream buffer, if any.

Example use:

```
// A stream buffer that has already been created.
StreamBuffer_t xStreamBuffer;

void vAnInterruptServiceRoutine( void )
{
uint8_t ucRxData[ 20 ];
size_t xReceivedBytes;
BaseType_t xHigherPriorityTaskWoken = pdFALSE;  // Initialised to pdFALSE.

 // Receive the next stream from the stream buffer.
 xReceivedBytes = xStreamBufferReceiveFromISR( xStreamBuffer,
                                               ( void * ) ucRxData,
                                               sizeof( ucRxData ),
                                               &xHigherPriorityTaskWoken );
 if( xReceivedBytes > 0 )
 {
     // ucRxData contains xReceivedBytes read from the stream buffer.
     // Process the stream here....
 }

 // If xHigherPriorityTaskWoken was set to pdTRUE inside
 // xStreamBufferReceiveFromISR() then a task that has a priority above the
 // priority of the currently executing task was unblocked and a context
 // switch should be performed to ensure the ISR returns to the unblocked
 // task.  In most FreeRTOS ports this is done by simply passing
 // xHigherPriorityTaskWoken into taskYIELD_FROM_ISR(), which will test the
 // variables value, and perform the context switch if necessary.  Check the
 // documentation for the port in use for port specific instructions.
 taskYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

## 5.64 vStreamBufferDelete

**stream_buffer.h** (p. **??**)

```
void vStreamBufferDelete( StreamBufferHandle_t xStreamBuffer );
```

Deletes a stream buffer that was previously created using a call to xStreamBufferCreate() or xStreamBufferCreate↩ Static(). If the stream buffer was created using dynamic memory (that is, by xStreamBufferCreate()), then the allocated memory is freed.

A stream buffer handle must not be used after the stream buffer has been deleted.

**Parameters**

| | |
|---|---|
| *xStreamBuffer* | The handle of the stream buffer to be deleted. |

## 5.65 xStreamBufferIsFull

**stream_buffer.h** (p. **??**)

```
BaseType_t xStreamBufferIsFull( StreamBufferHandle_t xStreamBuffer );
```

Queries a stream buffer to see if it is full. A stream buffer is full if it does not have any free space, and therefore cannot accept any more data.

**Parameters**

| | |
|---|---|
| *xStreamBuffer* | The handle of the stream buffer being queried. |

**Returns**

If the stream buffer is full then pdTRUE is returned. Otherwise pdFALSE is returned.

## 5.66 xStreamBufferIsEmpty

**stream_buffer.h** (p. **??**)

```
BaseType_t xStreamBufferIsEmpty( StreamBufferHandle_t xStreamBuffer );
```

Queries a stream buffer to see if it is empty. A stream buffer is empty if it does not contain any data.

**Parameters**

| | |
|---|---|
| *xStreamBuffer* | The handle of the stream buffer being queried. |

**Returns**

If the stream buffer is empty then pdTRUE is returned. Otherwise pdFALSE is returned.

## 5.67 xStreamBufferReset

**stream_buffer.h** (p. **??**)

```
BaseType_t xStreamBufferReset( StreamBufferHandle_t xStreamBuffer );
```

Resets a stream buffer to its initial, empty, state. Any data that was in the stream buffer is discarded. A stream buffer can only be reset if there are no tasks blocked waiting to either send to or receive from the stream buffer.

**Parameters**

| | |
|---|---|
| *xStreamBuffer* | The handle of the stream buffer being reset. |

**Returns**

If the stream buffer is reset then pdPASS is returned. If there was a task blocked waiting to send to or read from the stream buffer then the stream buffer is not reset and pdFAIL is returned.

## 5.68 xStreamBufferSpacesAvailable

**stream_buffer.h** (p. **??**)

```
size_t xStreamBufferSpacesAvailable( StreamBufferHandle_t xStreamBuffer );
```

Queries a stream buffer to see how much free space it contains, which is equal to the amount of data that can be sent to the stream buffer before it is full.

**Parameters**

| | |
|---|---|
| *xStreamBuffer* | The handle of the stream buffer being queried. |

**Returns**

The number of bytes that can be written to the stream buffer before the stream buffer would be full.

## 5.69 xStreamBufferBytesAvailable

**stream_buffer.h** (p. **??**)

```
size_t xStreamBufferBytesAvailable( StreamBufferHandle_t xStreamBuffer );
```

Queries a stream buffer to see how much data it contains, which is equal to the number of bytes that can be read from the stream buffer before the stream buffer would be empty.

**Parameters**

| | |
|---|---|
| *xStreamBuffer* | The handle of the stream buffer being queried. |

**Returns**

The number of bytes that can be read from the stream buffer before the stream buffer would be empty.

## 5.70 xStreamBufferSetTriggerLevel

**stream_buffer.h** (p. **??**)

```
BaseType_t xStreamBufferSetTriggerLevel( StreamBufferHandle_t xStreamBuffer, size_t xTriggerLevel
```

A stream buffer's trigger level is the number of bytes that must be in the stream buffer before a task that is blocked on the stream buffer to wait for data is moved out of the blocked state. For example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 1 then the task will be unblocked when a single byte is written to the buffer or the task's block time expires. As another example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 10 then the task will not be unblocked until the stream buffer contains at least 10 bytes or the task's block time expires. If a reading task's block time expires before the trigger level is reached then the task will still receive however many bytes are actually available. Setting a trigger level of 0 will result in a trigger level of 1 being used. It is not valid to specify a trigger level that is greater than the buffer size.

A trigger level is set when the stream buffer is created, and can be modified using xStreamBufferSetTriggerLevel().

**Parameters**

| | |
|---|---|
| *xStreamBuffer* | The handle of the stream buffer being updated. |
| *xTriggerLevel* | The new trigger level for the stream buffer. |

**Returns**

If xTriggerLevel was less than or equal to the stream buffer's length then the trigger level will be updated and pdTRUE is returned. Otherwise pdFALSE is returned.

## 5.71 xStreamBufferSendCompletedFromISR

**stream_buffer.h** (p. **??**)

```
BaseType_t xStreamBufferSendCompletedFromISR( StreamBufferHandle_t xStreamBuffer, BaseType_t *pxHi
```

For advanced users only.

The sbSEND_COMPLETED() macro is called from within the FreeRTOS APIs when data is sent to a message buffer or stream buffer. If there was a task that was blocked on the message or stream buffer waiting for data to arrive then the sbSEND_COMPLETED() macro sends a notification to the task to remove it from the Blocked state. xStreamBufferSendCompletedFromISR() does the same thing. It is provided to enable application writers to implement their own version of sbSEND_COMPLETED(), and MUST NOT BE USED AT ANY OTHER TIME.

See the example implemented in FreeRTOS/Demo/Minimal/MessageBufferAMP.c for additional information.

**Parameters**

| | |
|---|---|
| *xStreamBuffer* | The handle of the stream buffer to which data was written. |
| *pxHigherPriorityTaskWoken* | ∗pxHigherPriorityTaskWoken should be initialised to pdFALSE before it is passed into xStreamBufferSendCompletedFromISR(). If calling xStreamBufferSendCompletedFromISR() removes a task from the Blocked state, and the task has a priority above the priority of the currently running task, then ∗pxHigherPriorityTaskWoken will get set to pdTRUE indicating that a context switch should be performed before exiting the ISR. |

**Returns**

If a task was removed from the Blocked state then pdTRUE is returned. Otherwise pdFALSE is returned.

## 5.72 xStreamBufferReceiveCompletedFromISR

**stream_buffer.h** (p. **??**)

```
BaseType_t xStreamBufferReceiveCompletedFromISR( StreamBufferHandle_t xStreamBuffer, BaseType_t *p
```

For advanced users only.

The sbRECEIVE_COMPLETED() macro is called from within the FreeRTOS APIs when data is read out of a message buffer or stream buffer. If there was a task that was blocked on the message or stream buffer waiting for data to arrive then the sbRECEIVE_COMPLETED() macro sends a notification to the task to remove it from the Blocked state. xStreamBufferReceiveCompletedFromISR() does the same thing. It is provided to enable application writers to implement their own version of sbRECEIVE_COMPLETED(), and MUST NOT BE USED AT ANY OTHER TIME.

See the example implemented in FreeRTOS/Demo/Minimal/MessageBufferAMP.c for additional information.

**Parameters**

| | |
|---|---|
| *xStreamBuffer* | The handle of the stream buffer from which data was read. |
| *pxHigherPriorityTaskWoken* | ∗pxHigherPriorityTaskWoken should be initialised to pdFALSE before it is passed into xStreamBufferReceiveCompletedFromISR(). If calling xStreamBufferReceiveCompletedFromISR() removes a task from the Blocked state, and the task has a priority above the priority of the currently running task, then ∗pxHigherPriorityTaskWoken will get set to pdTRUE indicating that a context switch should be performed before exiting the ISR. |

**Returns**

If a task was removed from the Blocked state then pdTRUE is returned. Otherwise pdFALSE is returned.

## 5.73 TaskHandle_t

task. h

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

## 5.74 taskYIELD

task. h

Macro for forcing a context switch.

## 5.75 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!

## 5.76 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!

## 5.77 taskDISABLE_INTERRUPTS

task. h

Macro to disable all maskable interrupts.

## 5.78 taskENABLE_INTERRUPTS

task. h

Macro to enable microcontroller interrupts.

## 5.79 xTaskCreate

task. h

```
BaseType_t xTaskCreate(
                              TaskFunction_t pvTaskCode,
                              const char * const pcName,
                              configSTACK_DEPTH_TYPE 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.

Internally, within the FreeRTOS implementation, tasks use two blocks of memory. The first block is used to hold the task's data structures. The second block is used by the task as its stack. If a task is created using xTask↩ Create() then both blocks of memory are automatically dynamically allocated inside the xTaskCreate() function. (see `https://www.FreeRTOS.org/a00111.html`). If a task is created using xTaskCreateStatic() then the application writer must provide the required memory. xTaskCreateStatic() therefore allows a task to be created without using any dynamic memory allocation.

See xTaskCreateStatic() for a version that does not use any dynamic memory allocation.

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

**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 - default is 16. |
| *usStackDepth* | The size of the task stack specified as the number of variables the stack can hold - not the number of bytes. For example, if the stack is 16 bits wide and usStackDepth is defined as 100, 200 bytes will be allocated for stack storage. |
| *pvParameters* | Pointer that will be used as the parameter for the task being created. |
| *uxPriority* | The priority at which the task should run. Systems that include MPU support can optionally create tasks in a privileged (system) mode by setting bit portPRIVILEGE_BIT of the priority parameter. For example, to create a privileged task at priority 2 the uxPriority parameter should be set to ( 2 \| portPRIVILEGE_BIT ). |
| *pvCreatedTask* | Used to pass back a handle by which the created task can be referenced. |

**Returns**

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

Example usage:

```
// Task to be created.
void vTaskCode( void * pvParameters )
```

```
{
  for( ;; )
  {
      // Task code goes here.
  }
}

// Function that creates a task.
void vOtherFunction( void )
{
static uint8_t 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 automatic stack variable it might no longer exist, or at least have been corrupted, by the
  // 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 );
  }
}
```

## 5.80 xTaskCreateStatic

task. h

```
 TaskHandle_t xTaskCreateStatic( TaskFunction_t pvTaskCode,
                                 const char * const pcName,
                                 uint32_t ulStackDepth,
                                 void *pvParameters,
                                 UBaseType_t uxPriority,
                                 StackType_t *pxStackBuffer,
                                 StaticTask_t *pxTaskBuffer );
```

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

Internally, within the FreeRTOS implementation, tasks use two blocks of memory. The first block is used to hold the task's data structures. The second block is used by the task as its stack. If a task is created using xTask↩ Create() then both blocks of memory are automatically dynamically allocated inside the xTaskCreate() function. (see `https://www.FreeRTOS.org/a00111.html`). If a task is created using xTaskCreateStatic() then the application writer must provide the required memory. xTaskCreateStatic() therefore allows a task to be created without using any dynamic memory allocation.

**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. The maximum length of the string is defined by configMAX_TASK_NAME_LEN in FreeRTOSConfig.h. |
| *ulStackDepth* | 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 32-bits wide and ulStackDepth is defined as 100 then 400 bytes will be allocated for stack storage. |
| *pvParameters* | Pointer that will be used as the parameter for the task being created. |
| *uxPriority* | The priority at which the task will run. |
| *pxStackBuffer* | Must point to a StackType_t array that has at least ulStackDepth indexes - the array will then be used as the task's stack, removing the need for the stack to be allocated dynamically. |
| *pxTaskBuffer* | Must point to a variable of type StaticTask_t, which will then be used to hold the task's data structures, removing the need for the memory to be allocated dynamically. |

**Returns**

If neither pxStackBuffer or pxTaskBuffer are NULL, then the task will be created and a handle to the created task is returned. If either pxStackBuffer or pxTaskBuffer are NULL then the task will not be created and NULL is returned.

Example usage:

```
   // Dimensions the buffer that the task being created will use as its stack.
   // NOTE:  This is the number of words the stack will hold, not the number of
   // bytes.  For example, if each stack item is 32-bits, and this is set to 100,
   // then 400 bytes (100 * 32-bits) will be allocated.
#define STACK_SIZE 200
```

```
// Structure that will hold the TCB of the task being created.
StaticTask_t xTaskBuffer;

// Buffer that the task being created will use as its stack.  Note this is
// an array of StackType_t variables.  The size of StackType_t is dependent on
// the RTOS port.
StackType_t xStack[ STACK_SIZE ];

// Function that implements the task being created.
void vTaskCode( void * pvParameters )
{
    // The parameter value is expected to be 1 as 1 is passed in the
    // pvParameters value in the call to xTaskCreateStatic().
    configASSERT( ( uint32_t ) pvParameters == 1UL );

    for( ;; )
    {
        // Task code goes here.
    }
}

// Function that creates a task.
void vOtherFunction( void )
{
    TaskHandle_t xHandle = NULL;

    // Create the task without using any dynamic memory allocation.
    xHandle = xTaskCreateStatic(
                  vTaskCode,       // Function that implements the task.
                  "NAME",          // Text name for the task.
                  STACK_SIZE,      // Stack size in words, not bytes.
                  ( void * ) 1,    // Parameter passed into the task.
                  tskIDLE_PRIORITY,// Priority at which the task is created.
                  xStack,          // Array to use as the task's stack.
                  &xTaskBuffer );  // Variable to hold the task's data structure.

    // puxStackBuffer and pxTaskBuffer were not NULL, so the task will have
    // been created, and xHandle will be the task's handle.  Use the handle
    // to suspend the task.
    vTaskSuspend( xHandle );
}
```

## 5.81 xTaskCreateRestricted

task. h

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

Only available when configSUPPORT_DYNAMIC_ALLOCATION is set to 1.

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

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

See xTaskCreateRestrictedStatic() for a version that does not use any dynamic memory allocation.

**Parameters**

| | |
|---|---|
| *pxTaskDefinition* | Pointer to a structure that contains a member for each of the normal xTaskCreate() parameters (see the xTaskCreate() API documentation) plus an optional stack buffer and the memory region definitions. |
| *pxCreatedTask* | Used to pass back a handle by which the created task can be referenced. |

**Returns**

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

Example usage:

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

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

int main( void )
{
TaskHandle_t xHandle;
```

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

 // Start the scheduler.
 vTaskStartScheduler();

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

task. h

```
void vTaskAllocateMPURegions( TaskHandle_t xTask, const MemoryRegion_t * const pxRegions );
```

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

**Parameters**

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

Example usage:

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

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

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

## 5.82 xTaskCreateRestrictedStatic

task. h

```
BaseType_t xTaskCreateRestrictedStatic( TaskParameters_t *pxTaskDefinition, TaskHandle_t *pxCreate
```

Only available when configSUPPORT_STATIC_ALLOCATION is set to 1.

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

Internally, within the FreeRTOS implementation, tasks use two blocks of memory. The first block is used to hold the task's data structures. The second block is used by the task as its stack. If a task is created using xTask←CreateRestricted() then the stack is provided by the application writer, and the memory used to hold the task's data structure is automatically dynamically allocated inside the xTaskCreateRestricted() function. If a task is created using xTaskCreateRestrictedStatic() then the application writer must provide the memory used to hold the task's data structures too. xTaskCreateRestrictedStatic() therefore allows a memory protected task to be created without using any dynamic memory allocation.

**Parameters**

| | |
|---|---|
| *pxTaskDefinition* | Pointer to a structure that contains a member for each of the normal xTaskCreate() parameters (see the xTaskCreate() API documentation) plus an optional stack buffer and the memory region definitions. If configSUPPORT_STATIC_ALLOCATION is set to 1 the structure contains an additional member, which is used to point to a variable of type StaticTask_t - which is then used to hold the task's data structure. |
| *pxCreatedTask* | Used to pass back a handle by which the created task can be referenced. |

**Returns**

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

Example usage:

```
// Create an TaskParameters_t structure that defines the task to be created.
// The StaticTask_t variable is only included in the structure when
// configSUPPORT_STATIC_ALLOCATION is set to 1.  The PRIVILEGED_DATA macro can
// be used to force the variable into the RTOS kernel's privileged data area.
static PRIVILEGED_DATA StaticTask_t xTaskBuffer;
static const TaskParameters_t xCheckTaskParameters =
{
 vATask,     // pvTaskCode - the function that implements the task.
 "ATask",    // pcName - just a text name for the task to assist debugging.
 100,        // usStackDepth - the stack size DEFINED IN WORDS.
 NULL,       // pvParameters - passed into the task function as the function parameters.
 ( 1UL | portPRIVILEGE_BIT ),// uxPriority - task priority, set the portPRIVILEGE_BIT if the task
 cStackBuffer,// puxStackBuffer - the buffer to be used as the task stack.

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

  // Holds the task's data structure.
};

int main( void )
{
TaskHandle_t xHandle;

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

 // Start the scheduler.
 vTaskStartScheduler();

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

## 5.83 vTaskDelete

task. h

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

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

Remove a task from the RTOS real time kernel's 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**

| | |
|---|---|
| *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;

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

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

## 5.84 vTaskDelay

task. h

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

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

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

vTaskDelay() specifies a time at which the task wishes to unblock relative to the time at which vTaskDelay() is called. For example, specifying a block period of 100 ticks will cause the task to unblock 100 ticks after vTask↩Delay() 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 xTaskDelayUntil() 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 ); } }

## 5.85 xTaskDelayUntil

task. h

```
BaseType_t xTaskDelayUntil( TickType_t *pxPreviousWakeTime, const TickType_t xTimeIncrement );
```

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

Delay a task until a specified time. This function can be used by periodic tasks to ensure a constant execution frequency.

This function differs from vTaskDelay () in one important aspect: vTaskDelay () will cause a task to block for the specified number of ticks from the time vTaskDelay () is called. It is therefore difficult to use vTaskDelay () by itself to generate a fixed execution frequency as the time between a task starting to execute and that task calling vTask↩ Delay () 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, xTaskDelayUntil () specifies the absolute (exact) time at which it wishes to unblock.

The macro pdMS_TO_TICKS() can be used to calculate the number of ticks from a time specified in milliseconds with a resolution of one tick period.

**Parameters**

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

**Returns**

Value which can be used to check whether the task was actually delayed. Will be pdTRUE if the task way delayed and pdFALSE otherwise. A task will not be delayed if the next expected wake time is in the past.

Example usage:

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

    // Initialise the xLastWakeTime variable with the current time.
    xLastWakeTime = xTaskGetTickCount ();
    for( ;; )
    {
        // Wait for the next cycle.
        xWasDelayed = xTaskDelayUntil( &xLastWakeTime, xFrequency );
        // Perform action here. xWasDelayed value can be used to determine
        // whether a deadline was missed if the code here took too long.
    }
}
```

## 5.86 xTaskAbortDelay

task. h

```
BaseType_t xTaskAbortDelay( TaskHandle_t xTask );
```

INCLUDE_xTaskAbortDelay must be defined as 1 in FreeRTOSConfig.h for this function to be available.

A task will enter the Blocked state when it is waiting for an event. The event it is waiting for can be a temporal event (waiting for a time), such as when vTaskDelay() is called, or an event on an object, such as when xQueueReceive() or ulTaskNotifyTake() is called. If the handle of a task that is in the Blocked state is used in a call to xTaskAbort←Delay() then the task will leave the Blocked state, and return from whichever function call placed the task into the Blocked state.

There is no 'FromISR' version of this function as an interrupt would need to know which object a task was blocked on in order to know which actions to take. For example, if the task was blocked on a queue the interrupt handler would then need to know if the queue was locked.

**Parameters**

| | |
|---|---|
| *xTask* | The handle of the task to remove from the Blocked state. |

**Returns**

If the task referenced by xTask was not in the Blocked state then pdFAIL is returned. Otherwise pdPASS is returned.

## 5.87 uxTaskPriorityGet

task. h

```
UBaseType_t uxTaskPriorityGet( const TaskHandle_t xTask );
```

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

Obtain the priority of any task.

**Parameters**

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

**Returns**

The priority of xTask.

Example usage:

```
void vAFunction( void )
{
TaskHandle_t xHandle;

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

  // ...

  // Use the handle to obtain the priority of the created task.
  // It was created with tskIDLE_PRIORITY, but may have changed
  // it itself.
  if( uxTaskPriorityGet( xHandle ) != tskIDLE_PRIORITY )
  {
      // The task has changed it's priority.
  }

  // ...

  // Is our priority higher than the created task?
  if( uxTaskPriorityGet( xHandle ) < uxTaskPriorityGet( NULL ) )
  {
      // Our priority (obtained using NULL handle) is higher.
  }
}
```

## 5.88 vTaskGetInfo

task. h

```
void vTaskGetInfo( TaskHandle_t xTask, TaskStatus_t *pxTaskStatus, BaseType_t xGetFreeStackSpace,
```

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

Populates a TaskStatus_t structure with information about a task.

**Parameters**

| | |
|---|---|
| *xTask* | Handle of the task being queried. If xTask is NULL then information will be returned about the calling task. |
| *pxTaskStatus* | A pointer to the TaskStatus_t structure that will be filled with information about the task referenced by the handle passed using the xTask parameter. |

@xGetFreeStackSpace The TaskStatus_t structure contains a member to report the stack high water mark of the task being queried. Calculating the stack high water mark takes a relatively long time, and can make the system temporarily unresponsive - so the xGetFreeStackSpace parameter is provided to allow the high water mark checking to be skipped. The high watermark value will only be written to the TaskStatus_t structure if xGetFreeStackSpace is not set to pdFALSE;

**Parameters**

| | |
|---|---|
| *eState* | The TaskStatus_t structure contains a member to report the state of the task being queried. Obtaining the task state is not as fast as a simple assignment - so the eState parameter is provided to allow the state information to be omitted from the TaskStatus_t structure. To obtain state information then set eState to eInvalid - otherwise the value passed in eState will be reported as the task state in the TaskStatus_t structure. |

Example usage:

```
void vAFunction( void )
{
TaskHandle_t xHandle;
TaskStatus_t xTaskDetails;

 // Obtain the handle of a task from its name.
 xHandle = xTaskGetHandle( "Task_Name" );

 // Check the handle is not NULL.
 configASSERT( xHandle );

 // Use the handle to obtain further information about the task.
 vTaskGetInfo( xHandle,
               &xTaskDetails,
               pdTRUE, // Include the high water mark in xTaskDetails.
               eInvalid ); // Include the task state in xTaskDetails.
}
```

## 5.89 vTaskPrioritySet

task. h

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

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

Set the priority of any task.

A context switch will occur before the function returns if the priority being set is higher than the currently executing task.

**Parameters**

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

Example usage:

```
void vAFunction( void )
{
TaskHandle_t xHandle;

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

  // ...

  // Use the handle to raise the priority of the created task.
  vTaskPrioritySet( xHandle, tskIDLE_PRIORITY + 1 );

  // ...

  // Use a NULL handle to raise our priority to the same value.
  vTaskPrioritySet( NULL, tskIDLE_PRIORITY + 1 );
}
```

## 5.90 vTaskSuspend

task. h

```
void vTaskSuspend( TaskHandle_t xTaskToSuspend );
```

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

Suspend any task. When suspended a task will never get any microcontroller processing time, no matter what its priority.

Calls to vTaskSuspend are not accumulative - i.e. calling vTaskSuspend () twice on the same task still only requires one call to vTaskResume () to ready the suspended task.

**Parameters**

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

Example usage:

```
void vAFunction( void )
{
TaskHandle_t xHandle;

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

  // ...

  // Use the handle to suspend the created task.
  vTaskSuspend( xHandle );

  // ...

  // The created task will not run during this period, unless
  // another task calls vTaskResume( xHandle ).

  //...


  // Suspend ourselves.
  vTaskSuspend( NULL );

  // We cannot get here unless another task calls vTaskResume
  // with our handle as the parameter.
}
```

## 5.91 vTaskResume

task. h

```
void vTaskResume( TaskHandle_t xTaskToResume );
```

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

Resumes a suspended task.

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

**Parameters**

| | |
|---|---|
| *xTaskToResume* | Handle to the task being readied. |

Example usage:

```
void vAFunction( void )
{
TaskHandle_t xHandle;

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

  // ...

  // Use the handle to suspend the created task.
  vTaskSuspend( xHandle );

  // ...

  // The created task will not run during this period, unless
  // another task calls vTaskResume( xHandle ).

  //...


  // Resume the suspended task ourselves.
  vTaskResume( xHandle );

  // The created task will once again get microcontroller processing
  // time in accordance with its priority within the system.
}
```

## 5.92 vTaskResumeFromISR

task. h

```
void xTaskResumeFromISR( TaskHandle_t xTaskToResume );
```

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

An implementation of vTaskResume() that can be called from within an ISR.

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

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

**Parameters**

| | |
|---|---|
| *xTaskToResume* | Handle to the task being readied. |

**Returns**

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

## 5.93 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.

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 ()
}
```

## 5.94 vTaskEndScheduler

task. h

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

NOTE: At the time of writing only the x86 real mode port, which runs on a PC in place of DOS, implements this function.

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.
}
```

## 5.95 vTaskSuspendAll

task. h

```
void vTaskSuspendAll( void );
```

Suspends the scheduler without disabling interrupts. Context switches will not occur while the scheduler is suspended.

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, xTaskDelayUntil(), 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 ();
  }
}
```

## 5.96 xTaskResumeAll

task. h

```
BaseType_t xTaskResumeAll( void );
```

Resumes scheduler activity after it was suspended by a call to vTaskSuspendAll().

xTaskResumeAll() only resumes the scheduler. It does not unsuspend tasks that were previously suspended by a call to vTaskSuspend().

**Returns**

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

Example usage:

```
void vTask1( void * pvParameters )
{
  for( ;; )
  {
      // Task code goes here.

      // ...

      // At some point the task wants to perform a long operation during
      // which it does not want to get swapped out.  It cannot use
      // taskENTER_CRITICAL ()/taskEXIT_CRITICAL () as the length of the
      // operation may cause interrupts to be missed - including the
      // ticks.

      // Prevent the real time kernel swapping out the task.
      vTaskSuspendAll ();

      // Perform the operation here.  There is no need to use critical
      // sections as we have all the microcontroller processing time.
      // During this time interrupts will still operate and the real
      // time kernel tick count will be maintained.

      // ...

      // The operation is complete.  Restart the kernel.  We want to force
      // a context switch - but there is no point if resuming the scheduler
      // caused a context switch already.
      if( !xTaskResumeAll () )
      {
           taskYIELD ();
      }
  }
}
```

## 5.97 xTaskGetTickCount

task. h

```
TickType_t xTaskGetTickCount( void );
```

**Returns**

The count of ticks since vTaskStartScheduler was called.

## 5.98 xTaskGetTickCountFromISR

task. h

```
TickType_t xTaskGetTickCountFromISR( void );
```

**Returns**

The count of ticks since vTaskStartScheduler was called.

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

## 5.99 uxTaskGetNumberOfTasks

task. h

```
uint16_t 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.

## 5.100 pcTaskGetName

task. h

```
char *pcTaskGetName( TaskHandle_t xTaskToQuery );
```

**Returns**

The text (human readable) name of the task referenced by the handle xTaskToQuery. A task can query its own name by either passing in its own handle, or by setting xTaskToQuery to NULL.

## 5.101 pcTaskGetHandle

task. h

```
TaskHandle_t xTaskGetHandle( const char *pcNameToQuery );
```

NOTE: This function takes a relatively long time to complete and should be used sparingly.

**Returns**

The handle of the task that has the human readable name pcNameToQuery. NULL is returned if no matching name is found. INCLUDE_xTaskGetHandle must be set to 1 in FreeRTOSConfig.h for pcTaskGetHandle() to be available.

## 5.102 vTaskList

task. h

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

configUSE_TRACE_FACILITY and configUSE_STATS_FORMATTING_FUNCTIONS must both be defined as 1 for this function to be available. See the configuration section of the FreeRTOS.org website for more information.

NOTE 1: 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').

PLEASE NOTE:

This function is provided for convenience only, and is used by many of the demo applications. Do not consider it to be part of the scheduler.

vTaskList() calls uxTaskGetSystemState(), then formats part of the uxTaskGetSystemState() output into a human readable table that displays task names, states and stack usage.

vTaskList() has a dependency on the sprintf() C library function that might bloat the code size, use a lot of stack, and provide different results on different platforms. An alternative, tiny, third party, and limited functionality implementation of sprintf() is provided in many of the FreeRTOS/Demo sub-directories in a file called printf-stdarg.c (note printf-stdarg.c does not provide a full snprintf() implementation!).

It is recommended that production systems call uxTaskGetSystemState() directly to get access to raw stats data, rather than indirectly through a call to vTaskList().

**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. |

## 5.103 vTaskGetRunTimeStats

task. h

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

configGENERATE_RUN_TIME_STATS and configUSE_STATS_FORMATTING_FUNCTIONS must both be defined as 1 for this function to be available. The application must also then provide definitions for portCONFIGURE↩ _TIMER_FOR_RUN_TIME_STATS() and portGET_RUN_TIME_COUNTER_VALUE() to configure a peripheral timer/counter and return the timers current count value respectively. The counter should be at least 10 times the frequency of the tick count.

NOTE 1: 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.

NOTE 2:

This function is provided for convenience only, and is used by many of the demo applications. Do not consider it to be part of the scheduler.

vTaskGetRunTimeStats() calls uxTaskGetSystemState(), then formats part of the uxTaskGetSystemState() output into a human readable table that displays the amount of time each task has spent in the Running state in both absolute and percentage terms.

vTaskGetRunTimeStats() has a dependency on the sprintf() C library function that might bloat the code size, use a lot of stack, and provide different results on different platforms. An alternative, tiny, third party, and limited functionality implementation of sprintf() is provided in many of the FreeRTOS/Demo sub-directories in a file called printf-stdarg.c (note printf-stdarg.c does not provide a full snprintf() implementation!).

It is recommended that production systems call uxTaskGetSystemState() directly to get access to raw stats data, rather than indirectly through a call to vTaskGetRunTimeStats().

**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. |

## 5.104 ulTaskGetIdleRunTimeCounter

task. h

```
uint32_t ulTaskGetIdleRunTimeCounter( void );
```

configGENERATE_RUN_TIME_STATS and configUSE_STATS_FORMATTING_FUNCTIONS must both be defined as 1 for this function to be available. The application must also then provide definitions for portCONFIGURE↩
_TIMER_FOR_RUN_TIME_STATS() and portGET_RUN_TIME_COUNTER_VALUE() to configure a peripheral timer/counter and return the timers current count value respectively. The counter should be at least 10 times the frequency of the tick count.

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. While uxTaskGetSystemState() and vTaskGetRun↩
TimeStats() writes the total execution time of each task into a buffer, ulTaskGetIdleRunTimeCounter() returns the total execution time of just the idle task.

**Returns**

The total run time of the idle task. This is the amount of time the idle task has actually been executing. The unit of time is dependent on the frequency configured using the portCONFIGURE_TIMER_FOR_RUN_TIME_↩
STATS() and portGET_RUN_TIME_COUNTER_VALUE() macros.

## 5.105 xTaskNotifyIndexed

task. h

```
BaseType_t xTaskNotifyIndexed( TaskHandle_t xTaskToNotify, UBaseType_t uxIndexToNotify, uint32_t u
```

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

See `https://www.FreeRTOS.org/RTOS-task-notifications.html` for details.

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for these functions to be available.

Sends a direct to task notification to a task, with an optional value and action.

Each task has a private array of "notification values" (or 'notifications'), each of which is a 32-bit unsigned integer (uint32_t). The constant configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment one of the task's notification values. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

A task can use xTaskNotifyWaitIndexed() to [optionally] block to wait for a notification to be pending, or ulTask↩ NotifyTakeIndexed() to [optionally] block to wait for a notification value to have a non-zero value. The task does not consume any CPU time while it is in the Blocked state.

A notification sent to a task will remain pending until it is cleared by the task calling xTaskNotifyWaitIndexed() or ulTaskNotifyTakeIndexed() (or their un-indexed equivalents). If the task was already in the Blocked state to wait for a notification when the notification arrives then the task will automatically be removed from the Blocked state (unblocked) and the notification cleared.

**NOTE** Each notification within the array operates independently - a task can only block on one notification within the array at a time and will not be unblocked by a notification sent to any other array index.

Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single "notification value", and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. xTaskNotify() is the original API function, and remains backward compatible by always operating on the notification value at index 0 in the array. Calling xTaskNotify() is equivalent to calling xTaskNotifyIndexed() with the uxIndex↩ ToNotify parameter set to 0.

**Parameters**

| | |
|---|---|
| *xTaskToNotify* | The handle of the task being notified. The handle to a task can be returned from the xTaskCreate() API function used to create the task, and the handle of the currently running task can be obtained by calling xTaskGetCurrentTaskHandle(). |
| *uxIndexToNotify* | The index within the target task's array of notification values to which the notification is to be sent. uxIndexToNotify must be less than configTASK_NOTIFICATION_ARRAY_ENTRIES. xTaskNotify() does not have this parameter and always sends notifications to index 0. |
| *ulValue* | Data that can be sent with the notification. How the data is used depends on the value of the eAction parameter. |
| *eAction* | Specifies how the notification updates the task's notification value, if at all. Valid values for eAction are as follows: |

eSetBits - The target notification value is bitwise ORed with ulValue. xTaskNofifyIndexed() always returns pdPASS in this case.

eIncrement - The target notification value is incremented. ulValue is not used and xTaskNotifyIndexed() always returns pdPASS in this case.

eSetValueWithOverwrite - The target notification value is set to the value of ulValue, even if the task being notified had not yet processed the previous notification at the same array index (the task already had a notification pending at that index). xTaskNotifyIndexed() always returns pdPASS in this case.

eSetValueWithoutOverwrite - If the task being notified did not already have a notification pending at the same array index then the target notification value is set to ulValue and xTaskNotifyIndexed() will return pdPASS. If the task being notified already had a notification pending at the same array index then no action is performed and pdFAIL is returned.

eNoAction - The task receives a notification at the specified array index without the notification value at that index being updated. ulValue is not used and xTaskNotifyIndexed() always returns pdPASS in this case.

pulPreviousNotificationValue - Can be used to pass out the subject task's notification value before any bits are modified by the notify function.

**Returns**

Dependent on the value of eAction. See the description of the eAction parameter.

## 5.106 xTaskNotifyAndQueryIndexed

task. h

```
BaseType_t xTaskNotifyAndQueryIndexed( TaskHandle_t xTaskToNotify, UBaseType_t uxIndexToNotify, ui
```

```
BaseType_t xTaskNotifyAndQuery( TaskHandle_t xTaskToNotify, uint32_t ulValue, eNotifyAction eActio
```

See `https://www.FreeRTOS.org/RTOS-task-notifications.html` for details.

xTaskNotifyAndQueryIndexed() performs the same operation as xTaskNotifyIndexed() with the addition that it also returns the subject task's prior notification value (the notification value at the time the function is called rather than when the function returns) in the additional pulPreviousNotifyValue parameter.

xTaskNotifyAndQuery() performs the same operation as xTaskNotify() with the addition that it also returns the subject task's prior notification value (the notification value as it was at the time the function is called, rather than when the function returns) in the additional pulPreviousNotifyValue parameter.

## 5.107 xTaskNotifyIndexedFromISR

task. h

```
BaseType_t xTaskNotifyIndexedFromISR( TaskHandle_t xTaskToNotify, UBaseType_t uxIndexToNotify, uin
```

```
BaseType_t xTaskNotifyFromISR( TaskHandle_t xTaskToNotify, uint32_t ulValue, eNotifyAction eAction
```

See `https://www.FreeRTOS.org/RTOS-task-notifications.html` for details.

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for these functions to be available.

A version of xTaskNotifyIndexed() that can be used from an interrupt service routine (ISR).

Each task has a private array of "notification values" (or 'notifications'), each of which is a 32-bit unsigned integer (uint32_t). The constant configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment one of the task's notification values. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

A task can use xTaskNotifyWaitIndexed() to [optionally] block to wait for a notification to be pending, or ulTask↩NotifyTakeIndexed() to [optionally] block to wait for a notification value to have a non-zero value. The task does not consume any CPU time while it is in the Blocked state.

A notification sent to a task will remain pending until it is cleared by the task calling xTaskNotifyWaitIndexed() or ulTaskNotifyTakeIndexed() (or their un-indexed equivalents). If the task was already in the Blocked state to wait for a notification when the notification arrives then the task will automatically be removed from the Blocked state (unblocked) and the notification cleared.

**NOTE** Each notification within the array operates independently - a task can only block on one notification within the array at a time and will not be unblocked by a notification sent to any other array index.

Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single "notification value", and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. xTaskNotifyFromISR() is the original API function, and remains backward compatible by always operating on the notification value at index 0 within the array. Calling xTaskNotifyFromISR() is equivalent to calling xTaskNotify↩IndexedFromISR() with the uxIndexToNotify parameter set to 0.

**Parameters**

| | |
|---|---|
| *uxIndexToNotify* | The index within the target task's array of notification values to which the notification is to be sent. uxIndexToNotify must be less than configTASK_NOTIFICATION_ARRAY_ENTRIES. xTaskNotifyFromISR() does not have this parameter and always sends notifications to index 0. |
| *xTaskToNotify* | The handle of the task being notified. The handle to a task can be returned from the xTaskCreate() API function used to create the task, and the handle of the currently running task can be obtained by calling xTaskGetCurrentTaskHandle(). |
| *ulValue* | Data that can be sent with the notification. How the data is used depends on the value of the eAction parameter. |
| *eAction* | Specifies how the notification updates the task's notification value, if at all. Valid values for eAction are as follows: |

eSetBits - The task's notification value is bitwise ORed with ulValue. xTaskNofify() always returns pdPASS in this case.

eIncrement - The task's notification value is incremented. ulValue is not used and xTaskNotify() always returns pdPASS in this case.

eSetValueWithOverwrite - The task's notification value is set to the value of ulValue, even if the task being notified had not yet processed the previous notification (the task already had a notification pending). xTaskNotify() always returns pdPASS in this case.

eSetValueWithoutOverwrite - If the task being notified did not already have a notification pending then the task's notification value is set to ulValue and xTaskNotify() will return pdPASS. If the task being notified already had a notification pending then no action is performed and pdFAIL is returned.

eNoAction - The task receives a notification without its notification value being updated. ulValue is not used and xTaskNotify() always returns pdPASS in this case.

**Parameters**

| | |
|---|---|
| *pxHigherPriorityTaskWoken* | xTaskNotifyFromISR() will set ∗pxHigherPriorityTaskWoken to pdTRUE if sending the notification caused the task to which the notification was sent to leave the Blocked state, and the unblocked task has a priority higher than the currently running task. If xTaskNotifyFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited. How a context switch is requested from an ISR is dependent on the port - see the documentation page for the port in use. |

**Returns**

Dependent on the value of eAction. See the description of the eAction parameter.

## 5.108 xTaskNotifyAndQueryIndexedFromISR

task. h

```
BaseType_t xTaskNotifyAndQueryIndexedFromISR( TaskHandle_t xTaskToNotify, UBaseType_t uxIndexToNot
```

```
BaseType_t xTaskNotifyAndQueryFromISR( TaskHandle_t xTaskToNotify, uint32_t ulValue, eNotifyActior
```

See `https://www.FreeRTOS.org/RTOS-task-notifications.html` for details.

xTaskNotifyAndQueryIndexedFromISR() performs the same operation as xTaskNotifyIndexedFromISR() with the addition that it also returns the subject task's prior notification value (the notification value at the time the function is called rather than at the time the function returns) in the additional pulPreviousNotifyValue parameter.

xTaskNotifyAndQueryFromISR() performs the same operation as xTaskNotifyFromISR() with the addition that it also returns the subject task's prior notification value (the notification value at the time the function is called rather than at the time the function returns) in the additional pulPreviousNotifyValue parameter.

## 5.109 xTaskNotifyWaitIndexed

task. h

```
BaseType_t xTaskNotifyWaitIndexed( UBaseType_t uxIndexToWaitOn, uint32_t ulBitsToClearOnEntry, uin

BaseType_t xTaskNotifyWait( uint32_t ulBitsToClearOnEntry, uint32_t ulBitsToClearOnExit, uint32_t
```

Waits for a direct to task notification to be pending at a given index within an array of direct to task notifications.

See `https://www.FreeRTOS.org/RTOS-task-notifications.html` for details.

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for this function to be available.

Each task has a private array of "notification values" (or 'notifications'), each of which is a 32-bit unsigned integer (uint32_t). The constant configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment one of the task's notification values. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

A notification sent to a task will remain pending until it is cleared by the task calling xTaskNotifyWaitIndexed() or ulTaskNotifyTakeIndexed() (or their un-indexed equivalents). If the task was already in the Blocked state to wait for a notification when the notification arrives then the task will automatically be removed from the Blocked state (unblocked) and the notification cleared.

A task can use xTaskNotifyWaitIndexed() to [optionally] block to wait for a notification to be pending, or ulTask↩ NotifyTakeIndexed() to [optionally] block to wait for a notification value to have a non-zero value. The task does not consume any CPU time while it is in the Blocked state.

**NOTE** Each notification within the array operates independently - a task can only block on one notification within the array at a time and will not be unblocked by a notification sent to any other array index.

Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single "notification value", and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. xTaskNotifyWait() is the original API function, and remains backward compatible by always operating on the notification value at index 0 in the array. Calling xTaskNotifyWait() is equivalent to calling xTaskNotifyWaitIndexed() with the uxIndexToWaitOn parameter set to 0.

**Parameters**

| | |
|---|---|
| *uxIndexToWaitOn* | The index within the calling task's array of notification values on which the calling task will wait for a notification to be received. uxIndexToWaitOn must be less than configTASK_NOTIFICATION_ARRAY_ENTRIES. xTaskNotifyWait() does not have this parameter and always waits for notifications on index 0. |
| *ulBitsToClearOnEntry* | Bits that are set in ulBitsToClearOnEntry value will be cleared in the calling task's notification value before the task checks to see if any notifications are pending, and optionally blocks if no notifications are pending. Setting ulBitsToClearOnEntry to ULONG_MAX (if limits.h is included) or 0xffffffffUL (if limits.h is not included) will have the effect of resetting the task's notification value to 0. Setting ulBitsToClearOnEntry to 0 will leave the task's notification value unchanged. |

**Parameters**

| | |
|---|---|
| *ulBitsToClearOnExit* | If a notification is pending or received before the calling task exits the xTaskNotifyWait() function then the task's notification value (see the xTaskNotify() API function) is passed out using the pulNotificationValue parameter. Then any bits that are set in ulBitsToClearOnExit will be cleared in the task's notification value (note ∗pulNotificationValue is set before any bits are cleared). Setting ulBitsToClearOnExit to ULONG_MAX (if limits.h is included) or 0xffffffffUL (if limits.h is not included) will have the effect of resetting the task's notification value to 0 before the function exits. Setting ulBitsToClearOnExit to 0 will leave the task's notification value unchanged when the function exits (in which case the value passed out in pulNotificationValue will match the task's notification value). |
| *pulNotificationValue* | Used to pass the task's notification value out of the function. Note the value passed out will not be effected by the clearing of any bits caused by ulBitsToClearOnExit being non-zero. |
| *xTicksToWait* | The maximum amount of time that the task should wait in the Blocked state for a notification to be received, should a notification not already be pending when xTaskNotifyWait() was called. The task will not consume any processing time while it is in the Blocked state. This is specified in kernel ticks, the macro pdMS_TO_TICSK( value_in_ms ) can be used to convert a time specified in milliseconds to a time specified in ticks. |

**Returns**

If a notification was received (including notifications that were already pending when xTaskNotifyWait was called) then pdPASS is returned. Otherwise pdFAIL is returned.

## 5.110 xTaskNotifyGiveIndexed

task. h

```
BaseType_t xTaskNotifyGiveIndexed( TaskHandle_t xTaskToNotify, UBaseType_t uxIndexToNotify );
```

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

Sends a direct to task notification to a particular index in the target task's notification array in a manner similar to giving a counting semaphore.

See `https://www.FreeRTOS.org/RTOS-task-notifications.html` for more details.

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for these macros to be available.

Each task has a private array of "notification values" (or 'notifications'), each of which is a 32-bit unsigned integer (uint32_t). The constant configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment one of the task's notification values. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

xTaskNotifyGiveIndexed() is a helper macro intended for use when task notifications are used as light weight and faster binary or counting semaphore equivalents. Actual FreeRTOS semaphores are given using the xSemaphore↩ Give() API function, the equivalent action that instead uses a task notification is xTaskNotifyGiveIndexed().

When task notifications are being used as a binary or counting semaphore equivalent then the task being notified should wait for the notification using the ulTaskNotificationTakeIndexed() API function rather than the xTaskNotify↩ WaitIndexed() API function.

**NOTE** Each notification within the array operates independently - a task can only block on one notification within the array at a time and will not be unblocked by a notification sent to any other array index.

Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single "notification value", and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. xTaskNotifyGive() is the original API function, and remains backward compatible by always operating on the notification value at index 0 in the array. Calling xTaskNotifyGive() is equivalent to calling xTaskNotifyGiveIndexed() with the uxIndexToNotify parameter set to 0.

**Parameters**

| | |
|---|---|
| *xTaskToNotify* | The handle of the task being notified. The handle to a task can be returned from the xTaskCreate() API function used to create the task, and the handle of the currently running task can be obtained by calling xTaskGetCurrentTaskHandle(). |
| *uxIndexToNotify* | The index within the target task's array of notification values to which the notification is to be sent. uxIndexToNotify must be less than configTASK_NOTIFICATION_ARRAY_ENTRIES. xTaskNotifyGive() does not have this parameter and always sends notifications to index 0. |

**Returns**

xTaskNotifyGive() is a macro that calls xTaskNotify() with the eAction parameter set to eIncrement - so pdPASS is always returned.

## 5.111 vTaskNotifyGiveIndexedFromISR

task. h

```
void vTaskNotifyGiveIndexedFromISR( TaskHandle_t xTaskHandle, UBaseType_t uxIndexToNotify, BaseTyp
```

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

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

See `https://www.FreeRTOS.org/RTOS-task-notifications.html` for more details.

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for this macro to be available.

Each task has a private array of "notification values" (or 'notifications'), each of which is a 32-bit unsigned integer (uint32_t). The constant configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment one of the task's notification values. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

vTaskNotifyGiveIndexedFromISR() is intended for use when task notifications are used as light weight and faster binary or counting semaphore equivalents. Actual FreeRTOS semaphores are given from an ISR using the x↩SemaphoreGiveFromISR() API function, the equivalent action that instead uses a task notification is vTaskNotify↩GiveIndexedFromISR().

When task notifications are being used as a binary or counting semaphore equivalent then the task being notified should wait for the notification using the ulTaskNotificationTakeIndexed() API function rather than the xTaskNotify↩WaitIndexed() API function.

**NOTE** Each notification within the array operates independently - a task can only block on one notification within the array at a time and will not be unblocked by a notification sent to any other array index.

Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single "notification value", and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. xTaskNotifyFromISR() is the original API function, and remains backward compatible by always operating on the notification value at index 0 within the array. Calling xTaskNotifyGiveFromISR() is equivalent to calling xTask↩NotifyGiveIndexedFromISR() with the uxIndexToNotify parameter set to 0.

**Parameters**

| | |
|---|---|
| *xTaskToNotify* | The handle of the task being notified. The handle to a task can be returned from the xTaskCreate() API function used to create the task, and the handle of the currently running task can be obtained by calling xTaskGetCurrentTaskHandle(). |
| *uxIndexToNotify* | The index within the target task's array of notification values to which the notification is to be sent. uxIndexToNotify must be less than configTASK_NOTIFICATION_ARRAY_ENTRIES. xTaskNotifyGiveFromISR() does not have this parameter and always sends notifications to index 0. |
| *pxHigherPriorityTaskWoken* | vTaskNotifyGiveFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if sending the notification caused the task to which the notification was sent to leave the Blocked state, and the unblocked task has a priority higher than the currently running task. If vTaskNotifyGiveFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited. How a context switch is requested from an ISR is dependent on the port - see the |

## 5.112 ulTaskNotifyTakeIndexed

task. h

```
uint32_t ulTaskNotifyTakeIndexed( UBaseType_t uxIndexToWaitOn, BaseType_t xClearCountOnExit, TickT
```

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

Waits for a direct to task notification on a particular index in the calling task's notification array in a manner similar to taking a counting semaphore.

See `https://www.FreeRTOS.org/RTOS-task-notifications.html` for details.

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for this function to be available.

Each task has a private array of "notification values" (or 'notifications'), each of which is a 32-bit unsigned integer (uint32_t). The constant configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.

Events can be sent to a task using an intermediary object. Examples of such objects are queues, semaphores, mutexes and event groups. Task notifications are a method of sending an event directly to a task without the need for such an intermediary object.

A notification sent to a task can optionally perform an action, such as update, overwrite or increment one of the task's notification values. In that way task notifications can be used to send data to a task, or be used as light weight and fast binary or counting semaphores.

ulTaskNotifyTakeIndexed() is intended for use when a task notification is used as a faster and lighter weight binary or counting semaphore alternative. Actual FreeRTOS semaphores are taken using the xSemaphoreTake() API function, the equivalent action that instead uses a task notification is ulTaskNotifyTakeIndexed().

When a task is using its notification value as a binary or counting semaphore other tasks should send notifications to it using the xTaskNotifyGiveIndexed() macro, or xTaskNotifyIndex() function with the eAction parameter set to eIncrement.

ulTaskNotifyTakeIndexed() can either clear the task's notification value at the array index specified by the uxIndex↩ ToWaitOn parameter to zero on exit, in which case the notification value acts like a binary semaphore, or decrement the notification value on exit, in which case the notification value acts like a counting semaphore.

A task can use ulTaskNotifyTakeIndexed() to [optionally] block to wait for the task's notification value to be non-zero. The task does not consume any CPU time while it is in the Blocked state.

Where as xTaskNotifyWaitIndexed() will return when a notification is pending, ulTaskNotifyTakeIndexed() will return when the task's notification value is not zero.

**NOTE** Each notification within the array operates independently - a task can only block on one notification within the array at a time and will not be unblocked by a notification sent to any other array index.

Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single "notification value", and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. ulTaskNotifyTake() is the original API function, and remains backward compatible by always operating on the notification value at index 0 in the array. Calling ulTaskNotifyTake() is equivalent to calling ulTaskNotifyTakeIndexed() with the uxIndexToWaitOn parameter set to 0.

**Parameters**

| | |
|---|---|
| *uxIndexToWaitOn* | The index within the calling task's array of notification values on which the calling task will wait for a notification to be non-zero. uxIndexToWaitOn must be less than configTASK_NOTIFICATION_ARRAY_ENTRIES. xTaskNotifyTake() does not have this parameter and always waits for notifications on index 0. |
| *xClearCountOnExit* | if xClearCountOnExit is pdFALSE then the task's notification value is decremented when the function exits. In this way the notification value acts like a counting semaphore. If xClearCountOnExit is not pdFALSE then the task's notification value is cleared to zero when the function exits. In this way the notification value acts like a binary semaphore. |
| *xTicksToWait* | The maximum amount of time that the task should wait in the Blocked state for the task's notification value to be greater than zero, should the count not already be greater than zero when ulTaskNotifyTake() was called. The task will not consume any processing time while it is in the Blocked state. This is specified in kernel ticks, the macro pdMS_TO_TICSK( value_in_ms ) can be used to convert a time specified in milliseconds to a time specified in ticks. |

**Returns**

The task's notification count before it is either cleared to zero or decremented (see the xClearCountOnExit parameter).

## 5.113 xTaskNotifyStateClearIndexed

task. h

```
BaseType_t xTaskNotifyStateClearIndexed( TaskHandle_t xTask, UBaseType_t uxIndexToCLear );

BaseType_t xTaskNotifyStateClear( TaskHandle_t xTask );
```

See `https://www.FreeRTOS.org/RTOS-task-notifications.html` for details.

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for these functions to be available.

Each task has a private array of "notification values" (or 'notifications'), each of which is a 32-bit unsigned integer (uint32_t). The constant configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.

If a notification is sent to an index within the array of notifications then the notification at that index is said to be 'pending' until it is read or explicitly cleared by the receiving task. xTaskNotifyStateClearIndexed() is the function that clears a pending notification without reading the notification value. The notification value at the same array index is not altered. Set xTask to NULL to clear the notification state of the calling task.

Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single "notification value", and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. xTaskNotifyStateClear() is the original API function, and remains backward compatible by always operating on the notification value at index 0 within the array. Calling xTaskNotifyStateClear() is equivalent to calling xTaskNotify↩ StateClearIndexed() with the uxIndexToNotify parameter set to 0.

**Parameters**

| | |
|---|---|
| *xTask* | The handle of the RTOS task that will have a notification state cleared. Set xTask to NULL to clear a notification state in the calling task. To obtain a task's handle create the task using xTaskCreate() and make use of the pxCreatedTask parameter, or create the task using xTaskCreateStatic() and store the returned value, or use the task's name in a call to xTaskGetHandle(). |
| *uxIndexToClear* | The index within the target task's array of notification values to act upon. For example, setting uxIndexToClear to 1 will clear the state of the notification at index 1 within the array. uxIndexToClear must be less than configTASK_NOTIFICATION_ARRAY_ENTRIES. ulTaskNotifyStateClear() does not have this parameter and always acts on the notification at index 0. |

**Returns**

pdTRUE if the task's notification state was set to eNotWaitingNotification, otherwise pdFALSE.

## 5.114 ulTaskNotifyValueClear

task. h

```
uint32_t ulTaskNotifyValueClearIndexed( TaskHandle_t xTask, UBaseType_t uxIndexToClear, uint32_t u

uint32_t ulTaskNotifyValueClear( TaskHandle_t xTask, uint32_t ulBitsToClear );
```

See `https://www.FreeRTOS.org/RTOS-task-notifications.html` for details.

configUSE_TASK_NOTIFICATIONS must be undefined or defined as 1 for these functions to be available.

Each task has a private array of "notification values" (or 'notifications'), each of which is a 32-bit unsigned integer (uint32_t). The constant configTASK_NOTIFICATION_ARRAY_ENTRIES sets the number of indexes in the array, and (for backward compatibility) defaults to 1 if left undefined. Prior to FreeRTOS V10.4.0 there was only one notification value per task.

ulTaskNotifyValueClearIndexed() clears the bits specified by the ulBitsToClear bit mask in the notification value at array index uxIndexToClear of the task referenced by xTask.

Backward compatibility information: Prior to FreeRTOS V10.4.0 each task had a single "notification value", and all task notification API functions operated on that value. Replacing the single notification value with an array of notification values necessitated a new set of API functions that could address specific notifications within the array. ulTaskNotifyValueClear() is the original API function, and remains backward compatible by always operating on the notification value at index 0 within the array. Calling ulTaskNotifyValueClear() is equivalent to calling ulTaskNotify↵
ValueClearIndexed() with the uxIndexToClear parameter set to 0.

**Parameters**

| | |
|---|---|
| *xTask* | The handle of the RTOS task that will have bits in one of its notification values cleared. Set xTask to NULL to clear bits in a notification value of the calling task. To obtain a task's handle create the task using xTaskCreate() and make use of the pxCreatedTask parameter, or create the task using xTaskCreateStatic() and store the returned value, or use the task's name in a call to xTaskGetHandle(). |
| *uxIndexToClear* | The index within the target task's array of notification values in which to clear the bits. uxIndexToClear must be less than configTASK_NOTIFICATION_ARRAY_ENTRIES. ulTaskNotifyValueClear() does not have this parameter and always clears bits in the notification value at index 0. |
| *ulBitsToClear* | Bit mask of the bits to clear in the notification value of xTask. Set a bit to 1 to clear the corresponding bits in the task's notification value. Set ulBitsToClear to 0xffffffff (UINT_MAX on 32-bit architectures) to clear the notification value to 0. Set ulBitsToClear to 0 to query the task's notification value without clearing any bits. |

**Returns**

The value of the target task's notification value before the bits specified by ulBitsToClear were cleared.

## 5.115 vTaskSetTimeOutState

**task.h** (p. **??**)

```
void vTaskSetTimeOutState( TimeOut_t * const pxTimeOut );
```

Capture the current time for future use with xTaskCheckForTimeOut().

**Parameters**

| | |
|---|---|
| *pxTimeOut* | Pointer to a timeout object into which the current time is to be captured. The captured time includes the tick count and the number of times the tick count has overflowed since the system first booted. |

## 5.116 xTaskCheckForTimeOut

**task.h** (p. **??**)

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

Determines if pxTicksToWait ticks has passed since a time was captured using a call to vTaskSetTimeOutState(). The captured time includes the tick count and the number of times the tick count has overflowed.

**Parameters**

| | |
|---|---|
| *pxTimeOut* | The time status as captured previously using vTaskSetTimeOutState. If the timeout has not yet occurred, it is updated to reflect the current time status. |
| *pxTicksToWait* | The number of ticks to check for timeout i.e. if pxTicksToWait ticks have passed since pxTimeOut was last updated (either by vTaskSetTimeOutState() or xTaskCheckForTimeOut()), the timeout has occurred. If the timeout has not occurred, pxTIcksToWait is updated to reflect the number of remaining ticks. |

**Returns**

If timeout has occurred, pdTRUE is returned. Otherwise pdFALSE is returned and pxTicksToWait is updated to reflect the number of remaining ticks.

**See also**

`https://www.FreeRTOS.org/xTaskCheckForTimeOut.html`

Example Usage:

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

size_t xUART_Receive( uint8_t *pucBuffer, size_t uxWantedBytes )
{
size_t uxReceived = 0;
TickType_t xTicksToWait = MAX_TIME_TO_WAIT;
TimeOut_t xTimeOut;

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

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

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

    // Attempt to read uxWantedBytes from the receive buffer into pucBuffer.
    // The actual number of bytes read (which might be less than
    // uxWantedBytes) is returned.
    uxReceived = UART_read_from_receive_buffer( pxUARTInstance,
                                                pucBuffer,
                                                uxWantedBytes );

    return uxReceived;
}
```

## 5.117 xTaskCatchUpTicks

**task.h** (p. **??**)

```
BaseType_t xTaskCatchUpTicks( TickType_t xTicksToCatchUp );
```

This function corrects the tick count value after the application code has held interrupts disabled for an extended period resulting in tick interrupts having been missed.

This function is similar to vTaskStepTick(), however, unlike vTaskStepTick(), xTaskCatchUpTicks() may move the tick count forward past a time at which a task should be removed from the blocked state. That means tasks may have to be removed from the blocked state as the tick count is moved.

**Parameters**

| | |
|---|---|
| *xTicksToCatchUp* | The number of tick interrupts that have been missed due to interrupts being disabled. Its value is not computed automatically, so must be computed by the application writer. |

**Returns**

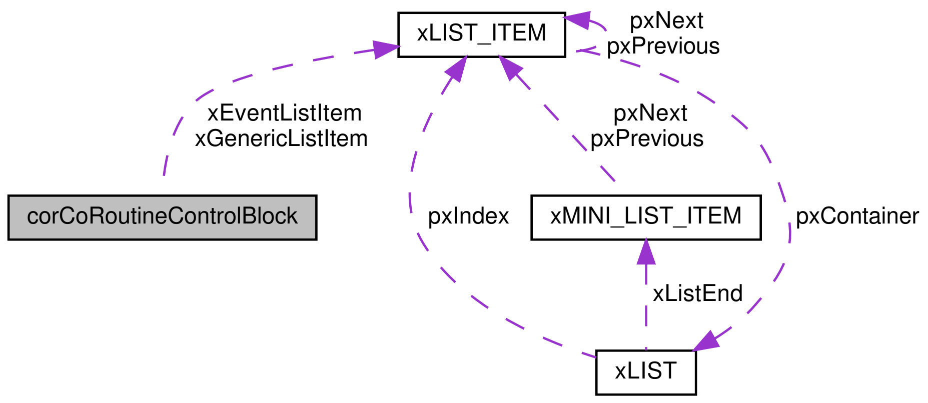
pdTRUE if moving the tick count forward resulted in a task leaving the blocked state and a context switch being performed. Otherwise pdFALSE.

# Chapter 6

# Data Structure Documentation

## 6.1 corCoRoutineControlBlock Struct Reference

Collaboration diagram for corCoRoutineControlBlock:

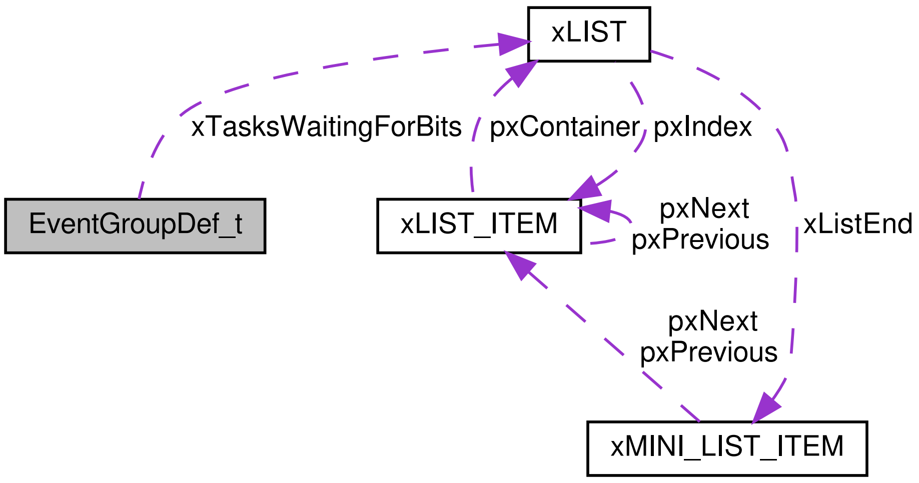


### Data Fields

- crCOROUTINE_CODE **pxCoRoutineFunction**
- **ListItem_t xGenericListItem**
- **ListItem_t xEventListItem**
- UBaseType_t **uxPriority**
- UBaseType_t **uxIndex**
- uint16_t **uxState**

## 6.2 EventGroupDef_t Struct Reference

Collaboration diagram for EventGroupDef_t:



### Data Fields

- EventBits_t **uxEventBits**
- **List_t xTasksWaitingForBits**

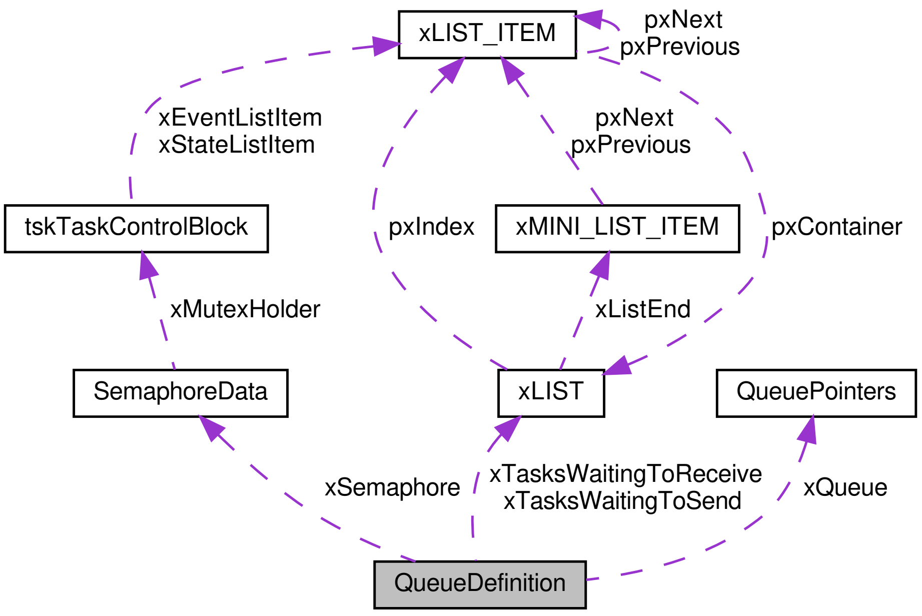
## 6.3 HeapRegion Struct Reference

### Data Fields

- uint8_t ∗ **pucStartAddress**
- size_t **xSizeInBytes**

## 6.4 QueueDefinition Struct Reference

Collaboration diagram for QueueDefinition:



### Data Fields

- int8_t ∗ **pcHead**
- int8_t ∗ **pcWriteTo**
-

  union {
      **QueuePointers_t xQueue**
      **SemaphoreData_t xSemaphore**
  } **u**

- **List_t xTasksWaitingToSend**
- **List_t xTasksWaitingToReceive**
- volatile UBaseType_t **uxMessagesWaiting**
- UBaseType_t **uxLength**
- UBaseType_t **uxItemSize**
- volatile int8_t **cRxLock**
- volatile int8_t **cTxLock**

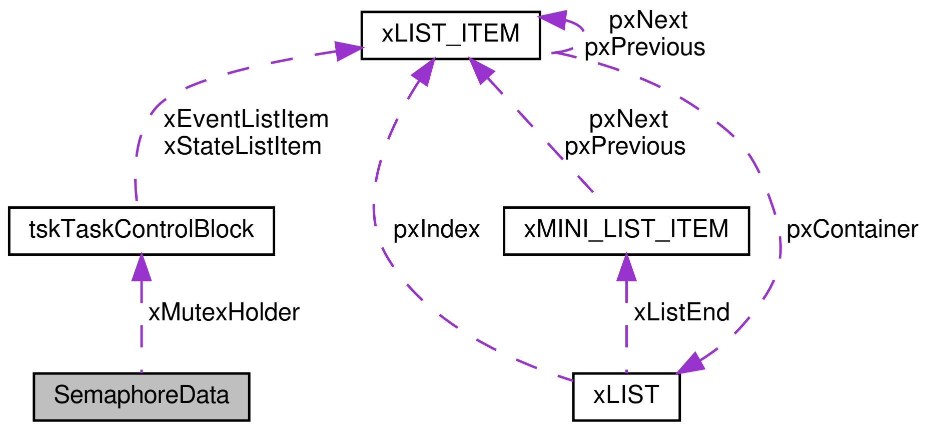
## 6.5 QueuePointers Struct Reference

### Data Fields

- int8_t ∗ **pcTail**
- int8_t ∗ **pcReadFrom**

## 6.6 SemaphoreData Struct Reference

Collaboration diagram for SemaphoreData:

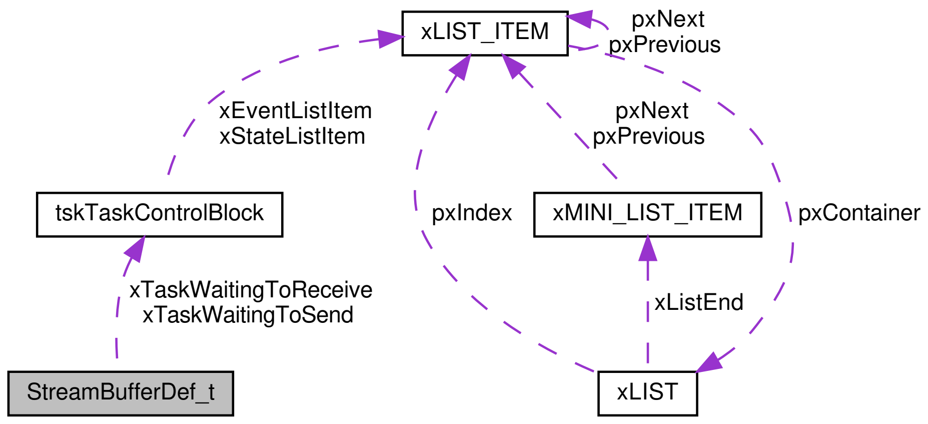


### Data Fields

- **TaskHandle_t xMutexHolder**
- UBaseType_t **uxRecursiveCallCount**

## 6.7 StreamBufferDef_t Struct Reference

Collaboration diagram for StreamBufferDef_t:

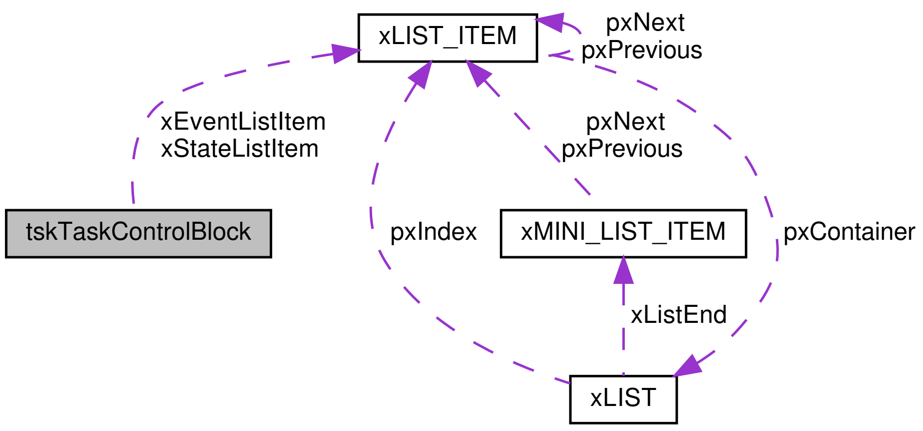


### Data Fields

- volatile size_t **xTail**
- volatile size_t **xHead**
- size_t **xLength**
- size_t **xTriggerLevelBytes**
- volatile **TaskHandle_t xTaskWaitingToReceive**
- volatile **TaskHandle_t xTaskWaitingToSend**
- uint8_t ∗ **pucBuffer**
- uint8_t **ucFlags**

## 6.8 tskTaskControlBlock Struct Reference

Collaboration diagram for tskTaskControlBlock:



### Data Fields

- volatile StackType_t ∗ **pxTopOfStack**
- **ListItem_t xStateListItem**
- **ListItem_t xEventListItem**
- UBaseType_t **uxPriority**
- StackType_t ∗ **pxStack**
- char **pcTaskName** [configMAX_TASK_NAME_LEN]

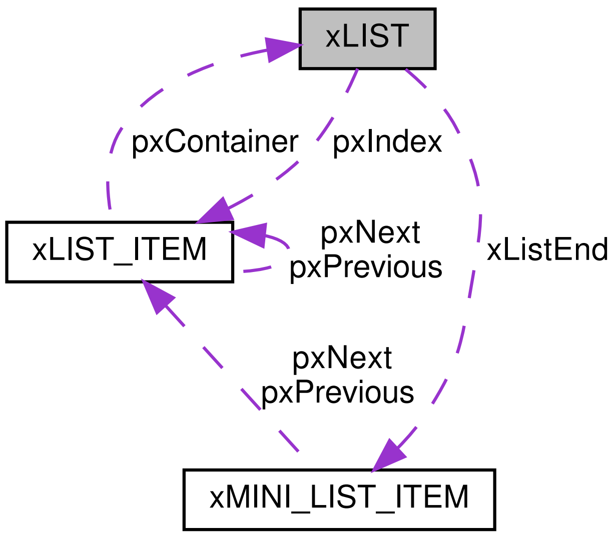
## 6.9 xHeapStats Struct Reference

### Data Fields

- size_t **xAvailableHeapSpaceInBytes**
- size_t **xSizeOfLargestFreeBlockInBytes**
- size_t **xSizeOfSmallestFreeBlockInBytes**
- size_t **xNumberOfFreeBlocks**
- size_t **xMinimumEverFreeBytesRemaining**
- size_t **xNumberOfSuccessfulAllocations**
- size_t **xNumberOfSuccessfulFrees**

## 6.10 xLIST Struct Reference

Collaboration diagram for xLIST:

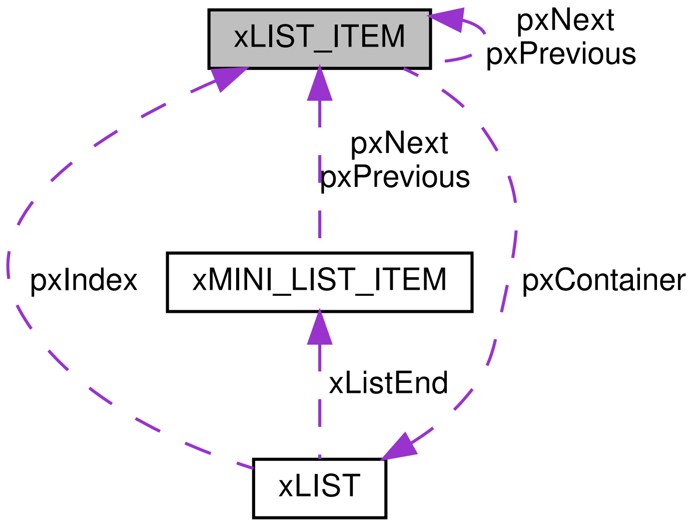


### Data Fields

- listFIRST_LIST_INTEGRITY_CHECK_VALUE volatile UBaseType_t **uxNumberOfItems**
- **ListItem_t** ∗configLIST_VOLATILE **pxIndex**
- **MiniListItem_t xListEnd**

## 6.11 xLIST_ITEM Struct Reference

Collaboration diagram for xLIST_ITEM:



### Data Fields

- listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE configLIST_VOLATILE TickType_t **xItemValue**
- struct **xLIST_ITEM** ∗configLIST_VOLATILE **pxNext**
- struct **xLIST_ITEM** ∗configLIST_VOLATILE **pxPrevious**
- void ∗ **pvOwner**
- struct **xLIST** ∗configLIST_VOLATILE **pxContainer**

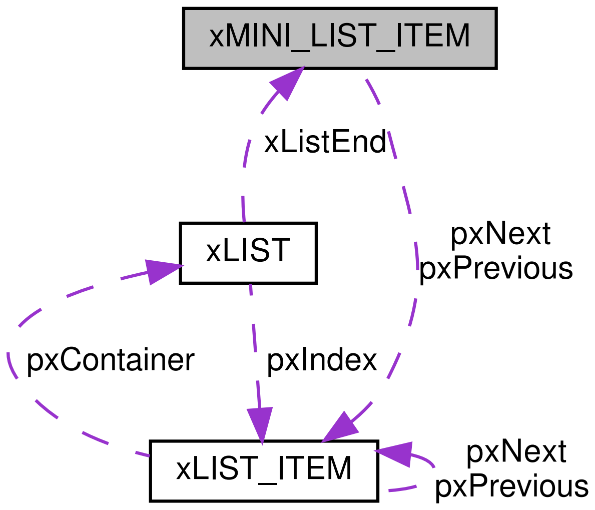
## 6.12 xMEMORY_REGION Struct Reference

### Data Fields

- void ∗ **pvBaseAddress**
- uint32_t **ulLengthInBytes**
- uint32_t **ulParameters**

## 6.13 xMINI_LIST_ITEM Struct Reference

Collaboration diagram for xMINI_LIST_ITEM:

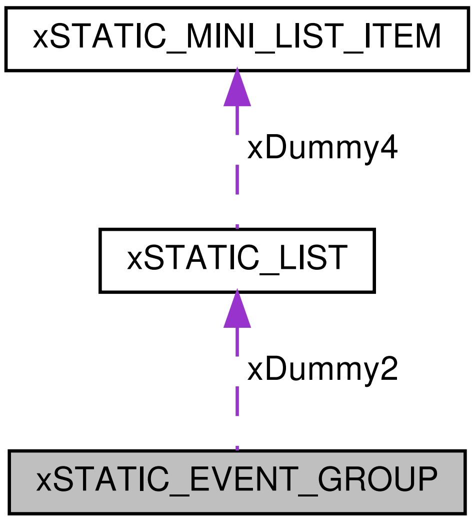


### Data Fields

- listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE configLIST_VOLATILE TickType_t **xItemValue**
- struct **xLIST_ITEM** ∗configLIST_VOLATILE **pxNext**
- struct **xLIST_ITEM** ∗configLIST_VOLATILE **pxPrevious**

## 6.14 xSTATIC_EVENT_GROUP Struct Reference

Collaboration diagram for xSTATIC_EVENT_GROUP:

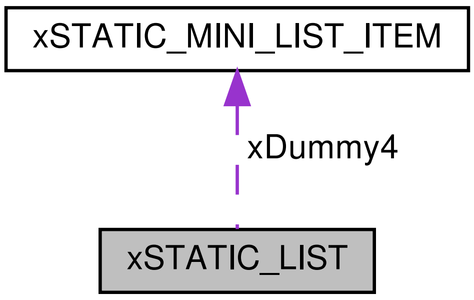


### Data Fields

- TickType_t **xDummy1**
- **StaticList_t xDummy2**

## 6.15 xSTATIC_LIST Struct Reference

Collaboration diagram for xSTATIC_LIST:



### Data Fields

- UBaseType_t **uxDummy2**
- void ∗ **pvDummy3**
- **StaticMiniListItem_t xDummy4**

## 6.16 xSTATIC_LIST_ITEM Struct Reference

### Data Fields

- TickType_t **xDummy2**
- void ∗ **pvDummy3** [4]

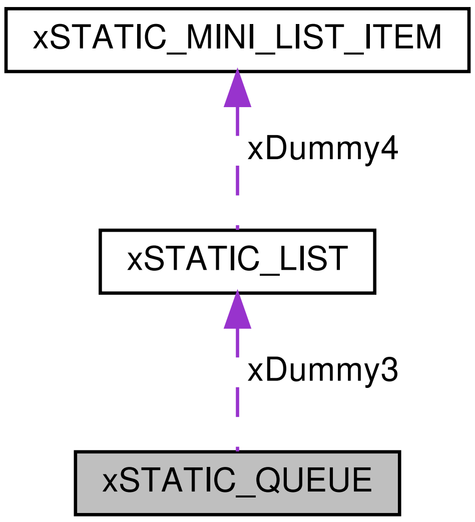
## 6.17 xSTATIC_MINI_LIST_ITEM Struct Reference

### Data Fields

- TickType_t **xDummy2**
- void ∗ **pvDummy3** [2]

## 6.18 xSTATIC_QUEUE Struct Reference

Collaboration diagram for xSTATIC_QUEUE:



### Data Fields

- void ∗ **pvDummy1** [3]
- 
  union {
    void ∗ **pvDummy2**
    UBaseType_t **uxDummy2**
  } **u**

- **StaticList_t xDummy3** [2]
- UBaseType_t **uxDummy4** [3]
- uint8_t **ucDummy5** [2]

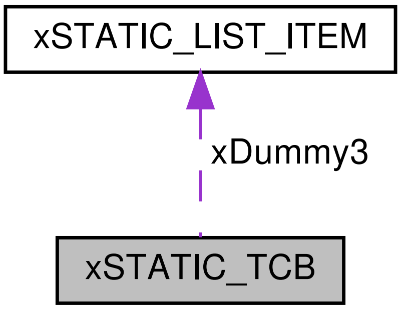
## 6.19 xSTATIC_STREAM_BUFFER Struct Reference

### Data Fields

- size_t **uxDummy1** [4]
- void ∗ **pvDummy2** [3]
- uint8_t **ucDummy3**

## 6.20 xSTATIC_TCB Struct Reference

Collaboration diagram for xSTATIC_TCB:

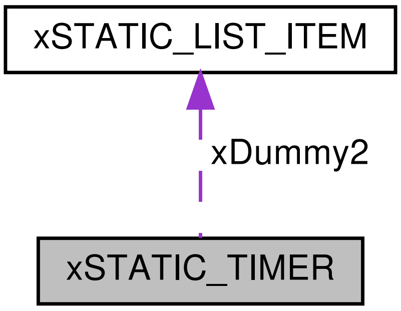


### Data Fields

- void ∗ **pxDummy1**
- **StaticListItem_t xDummy3** [2]
- UBaseType_t **uxDummy5**
- void ∗ **pxDummy6**
- uint8_t **ucDummy7** [configMAX_TASK_NAME_LEN]
- uint32_t **ulDummy18** [configTASK_NOTIFICATION_ARRAY_ENTRIES]
- uint8_t **ucDummy19** [configTASK_NOTIFICATION_ARRAY_ENTRIES]

## 6.21 xSTATIC_TIMER Struct Reference

Collaboration diagram for xSTATIC_TIMER:

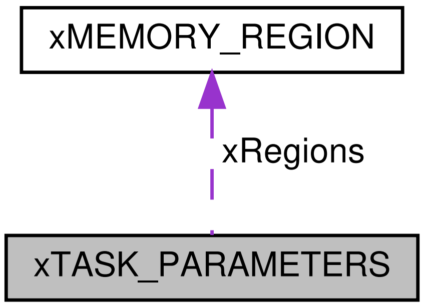


### Data Fields

- void ∗ **pvDummy1**
- **StaticListItem_t xDummy2**
- TickType_t **xDummy3**
- void ∗ **pvDummy5**
- TaskFunction_t **pvDummy6**
- uint8_t **ucDummy8**

## 6.22 xTASK_PARAMETERS Struct Reference

Collaboration diagram for xTASK_PARAMETERS:

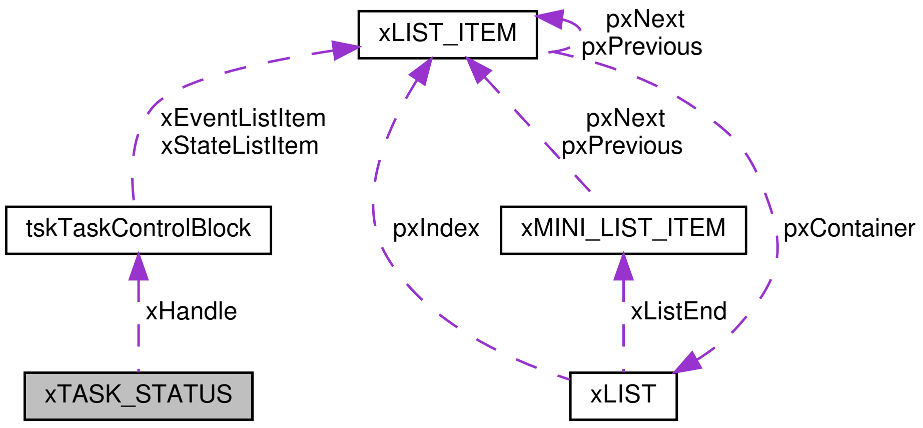


### Data Fields

- TaskFunction_t **pvTaskCode**
- const char ∗ **pcName**
- configSTACK_DEPTH_TYPE **usStackDepth**
- void ∗ **pvParameters**
- UBaseType_t **uxPriority**
- StackType_t ∗ **puxStackBuffer**
- **MemoryRegion_t xRegions** [portNUM_CONFIGURABLE_REGIONS]

## 6.23 xTASK_STATUS Struct Reference

Collaboration diagram for xTASK_STATUS:



### Data Fields

- **TaskHandle_t xHandle**
- const char ∗ **pcTaskName**
- UBaseType_t **xTaskNumber**
- eTaskState **eCurrentState**
- UBaseType_t **uxCurrentPriority**
- UBaseType_t **uxBasePriority**
- uint32_t **ulRunTimeCounter**
- StackType_t ∗ **pxStackBase**
- configSTACK_DEPTH_TYPE **usStackHighWaterMark**

## 6.24 xTIME_OUT Struct Reference

### Data Fields

- BaseType_t **xOverflowCount**
- TickType_t **xTimeOnEntering**