FreeRTOS Kernel

Reference Manual

FreeRTOS Kernel: Reference Manual

Table of Contents

Welcome

This guide provides a technical reference to both the primary FreeRTOS API and the FreeRTOS kernel configuration options. It is assumed that the reader is already familiar with the concepts of writing multitasking applications, and the primitives provided by real-time kernels. Readers who are not familiar with these fundamental concepts can read the FreeRTOS Kernel Developer Guide for a more descriptive, tutorial-style text.

Within this document, the API functions have been split into seven groups:

Other topics are covered in:

API Usage Restrictions

The following rules apply when using the FreeRTOS API:

1. API functions that do not end in “FromISR” must not be used in an interrupt service routine (ISR). Some FreeRTOS ports make a further restriction that even API functions that do end in “FromISR” cannot be used in an interrupt service routine that has a (hardware) priority above the priority set by the configMAX_SYSCALL_INTERRUPT_PRIORITY (or configMAX_API_CALL_INTERRUPT_PRIORITY, depending on the port) kernel configuration constant, which is described in Kernel Configuration (p. 289). The second restriction is to ensure that the timing, determinism, and latency of interrupts that have a priority above that set by configMAX_SYSCALL_INTERRUPT_PRIORITY are not affected by FreeRTOS.
2. API functions that can potentially cause a context switch must not be called while the scheduler is suspended.
3. API functions that can potentially cause a context switch must not be called from within a critical section.

Task and Scheduler API

portSWITCH_TO_USER_MODE()

```
#include "FreeRTOS.h"

#include "task.h"

void portSWITCH_TO_USER_MODE( void );
```

Summary

This function is intended for advanced users only. It is only relevant to FreeRTOS ports that make use of a Memory Protection Unit (MPU)).

MPU-restricted tasks are created using xTaskCreateRestricted(). The parameters supplied to xTaskCreateRestricted() specify whether the task being created should be a User (unprivileged) mode task or a Supervisor (privileged) mode task. A Supervisor mode task can call portSWITCH_TO_USER_MODE() to convert itself from a Supervisor mode task into a User mode task.

Parameters

None.

Return Values

None.

Notes

There is no reciprocal equivalent to portSWITCH_TO_USER_MODE() that permits a task to convert itself from a User mode into a Supervisor mode task.

vTaskAllocateMPURegions()

```
#include "FreeRTOS.h"

#include "task.h"

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

Summary

Define a set of Memory Protection Unit (MPU) regions for use by an MPU restricted task.

This function is intended for advanced users only. It is only relevant to FreeRTOS MPU ports.

MPU-controlled memory regions can be assigned to an MPU restricted task when the task is created using the xTaskCreateRestricted() function. They can then be redefined (or reassigned) at runtime using the vTaskAllocateMPURegions() function.

Parameters

xTaskToModify	The handle of the restricted task being modified (the task that is being given access to the memory regions defined by the xRegions parameter). The handle of a task is obtained using the pxCreatedTask parameter of the xTaskCreateRestricted() API function. A task can modify its own memory region access definitions by passing NULL in place of a valid task handle.
xRegions	An array of MemoryRegion_t structures. The number of positions in the array is defined by the port-specific portNUM_CONFIGURABLE_REGIONS constant. On a Cortex-M3, portNUM_CONFIGURABLE_REGIONS is defined as three. Each MemoryRegion_t structure in the array defines a single MPU memory region for use by the task referenced by the xTaskToModify parameter.

Notes

MPU memory regions are defined using the MemoryRegion_t structure shown in the following code.

```
typedef struct xMEMORY_REGION

{

    void *pvBaseAddress;

    unsigned long ulLengthInBytes;

    unsigned long ulParameters;

} MemoryRegion_t;
```

The pvBaseAddress and ulLengthInBytes members are self-explanatory as the start of the memory region and the length of the memory region respectively. These must comply with the size and alignment restrictions imposed by the MPU. In particular, the size and alignment of each region must be equal to the same power of two value.

ulParameters defines how the task is permitted to access the memory region being defined and can take the bitwise OR of the following values:

- portMPU_REGION_READ_WRITE
- portMPU_REGION_PRIVILEGED_READ_ONLY
- portMPU_REGION_READ_ONLY
- portMPU_REGION_PRIVILEGED_READ_WRITE
- portMPU_REGION_CACHEABLE_BUFFERABLE
- portMPU_REGION_EXECUTE_NEVER

Example

```
/* Define an array that the task will both read from and write to. Make sure the size and
 alignment are appropriate for an MPU region. (Note this uses GCC syntax.) */

static unsigned char ucOneKByte[1024] __attribute__ ((align (1024)));

/* Define an array of MemoryRegion_t structures that configures an MPU region allowing
 read/write access for 1024 bytes starting at the beginning of the ucOneKByte array. */
/* The other two of the maximum three 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 using xTaskCreateRestricted() to have access to a maximum of
 three MPU-controlled memory regions. */
    /* At some point it is required that these MPU regions are replaced with those defined
 in the xAltRegions const structure defined above.*/
    /* Use a call to vTaskAllocateMPURegions() for this purpose. */
```

```
    /* NULL is used as the task handle to indicate that the change should be applied to 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). */

}
```

xTaskAbortDelay ()

```
#include "FreeRTOS.h"
#include "task.h"
BaseType_t xTaskAbortDelay (TaskHandle_t xTask);
```

Summary

Calling an API function that includes a timeout parameter can result in the calling task entering the Blocked state. A task that is in the Blocked state is either waiting for a timeout period to elapse or waiting with a timeout for an event to occur, after which the task will automatically leave the Blocked state and enter the Ready state. Here are two examples of this behavior:

- If a task calls vTaskDelay(), it will enter the Blocked state until the timeout specified by the function's parameter has elapsed, at which time the task will automatically leave the Blocked state and enter the Ready state.
- If a task calls ulTaskNotifyTake() when its notification value is zero, it will enter the Blocked state until either it receives a notification or the timeout specified by one of the function's parameters has elapsed, at which time the task will automatically leave the Blocked state and enter the Ready state.

xTaskAbortDelay() will move a task from the Blocked state to the Ready state even if the event the task is waiting for has not occurred, and the timeout specified when the task entered the Blocked state has not elapsed.

While a task is in the Blocked state, it is not available to the scheduler and will not consume any processing time.

Parameters

xTask	The handle of the task that will be moved out of the Blocked state. 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().
Returned value	If the task referenced by xTask was removed from the Blocked state, then pdPASS is returned. If the task referenced by xTask was not removed from the Blocked state because it was not in the Blocked state, then pdFAIL is returned.

Notes

INCLUDE_xTaskAbortDelay must be set to 1 in FreeRTOSConfig.h for xTaskAbortDelay() to be available.

Example

```
void vAFunction( TaskHandle_t xTask )

{

    /* The task referenced by xTask is blocked to wait for something that the task calling
 this function has determined will never happen. Force the task referenced by xTask out of
 the Blocked state. */

    if( xTaskAbortDelay( xTask ) == pdFAIL )

    {

        /* The task referenced by xTask was not in the Blocked state anyway. */

    }

    else

    {

        /* The task referenced by xTask was in the Blocked state, but is not now. */

    }

}
```

xTaskCallApplicationTaskHook()

```
#include "FreeRTOS.h"

#include "task.h"

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

Summary

This function is intended for advanced users only.

The vTaskSetApplicationTaskTag() function can be used to assign a 'tag' value to a task. The meaning and use of the tag value is defined by the application writer. The kernel will not normally access the tag value.

As a special case, the tag value can be used to associate a task hook (or callback) function to a task. When this is done, the hook function is called using xTaskCallApplicationTaskHook().

Task hook functions can be used for any purpose. The example shown in this section demonstrates a task hook used to output debug trace information.

The following shows the prototype to which all task hook functions must conform:

```
BaseType_t xAnExampleTaskHookFunction( void *pvParameters );
```

xTaskCallApplicationTaskHook() is only available when configUSE_APPLICATION_TASK_TAG is set to 1 in FreeRTOSConfig.h.

Parameters

xTask	The handle of the task whose associated hook function is being called. 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(). A task can call its own hook function by passing NULL in place of a valid task handle.
pvParameters	The value used as the parameter to the task hook function itself. This parameter has the type 'pointer to void' to allow the task hook function parameter to

	effectively (and indirectly by means of casting) receive a parameter of any type. For example, integer types can be passed into a hook function by casting the integer to a void pointer at the point the hook function is called, then by casting the void pointer parameter back to an integer within the hook function itself.

Example

```
/* Define a hook (callback) function using the required prototype as demonstrated by
 Listing 8 */

static BaseType_t prvExampleTaskHook( void * pvParameter )

{

    /* Perform an action. This could be anything. In this example, the hook is used to
 output debug trace information. pxCurrentTCB is the handle of the currently executing
 task. (vWriteTrace() is not an API function. It's just used as an example.) */

    vWriteTrace( pxCurrentTCB );

    /* This example does not make use of the hook return value so just returns 0 in every
 case. */

    return 0;

}

/* Define an example task that makes use of its tag value. */

void vAnotherTask( void *pvParameters )

{

    /* vTaskSetApplicationTaskTag() sets the 'tag' value associated with a task. NULL is
 used in place of a valid task handle to indicate that it should be the tag value of the
 calling task that gets set. In this example, the 'value' being set is the hook function.
 */

    vTaskSetApplicationTaskTag( NULL, prvExampleTaskHook );

    for( ;; )

    {

        /* The rest of the task code goes here. */

    }

}

/* Define the traceTASK_SWITCHED_OUT() macro to call the hook function of each task that is
 switched out. pxCurrentTCB points to the handle of the currently running task. */

#define traceTASK_SWITCHED_OUT() xTaskCallApplicationTaskHook( pxCurrentTCB, 0 )
```

xTaskCheckForTimeOut()

```
#include "FreeRTOS.h"

#include "task.h"

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

Summary

This function is intended for advanced users only.

A task can enter the Blocked state to wait for an event. Typically, the task will not wait in the Blocked state indefinitely. Instead, a timeout period will be specified. The task will be removed from the Blocked state if the timeout period expires before the event the task is waiting for occurs.

If a task enters and exits the Blocked state more than once while it is waiting for the event to occur, then the timeout used each time the task enters the Blocked state must be adjusted to ensure the total of all the time spent in the Blocked state does not exceed the originally specified timeout period. xTaskCheckForTimeOut() performs the adjustment, taking into account occasional occurrences such as tick count overflows, which would otherwise make a manual adjustment prone to error.

xTaskCheckForTimeOut() is used with vTaskSetTimeOutState(). vTaskSetTimeOutState() is called to set the initial condition, after which xTaskCheckForTimeOut() can be called to check for a timeout condition and adjust the remaining block time if a timeout has not occurred.

Parameters

pxTimeOut	A pointer to a structure that holds information required to determine if a timeout has occurred. pxTimeOut is initialized using vTaskSetTimeOutState().
pxTicksToWait	Used to pass out an adjusted block time, which is the block time that remains after taking into account the time already spent in the Blocked state.
Returned value	If pdTRUE is returned, then no block time remains, and a timeout has occurred. If pdFALSE is returned, then some block time remains, so a timeout has not occurred.

Example

```
/* 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 recalculate 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 the bytes 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;

}
```

xTaskCreate()

```
#include "FreeRTOS.h"

#include "task.h"

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

Summary

Creates a new instance of a task.

Each task requires RAM that is used to hold the task state (the task control block or TCB) and used by the task as its stack. If a task is created using xTaskCreate(), then the required RAM is automatically allocated from the FreeRTOS heap. If a task is created using xTaskCreateStatic(), then the RAM is provided by the application writer, which results in two additional function parameters, but allows the RAM to be statically allocated at compile time.

Newly created tasks are initially placed in the Ready state, but will immediately become the Running state task if there are no higher priority tasks that are able to run.

Tasks can be created both before and after the scheduler has been started.

Parameters

pvTaskCode	Tasks are simply C functions that never exit and, as such, are normally implemented as an infinite loop. The pvTaskCode parameter is simply a pointer to the function (in effect, just the function name) that implements the task.
pcName	A descriptive name for the task. This is mainly used to facilitate debugging, but can also be used in a call to xTaskGetHandle() to obtain a task handle. The application-defined constant configMAX_TASK_NAME_LEN defines the maximum length of the name in characters, including the NULL terminator. Supplying a string longer than this maximum will result in the string being silently truncated.
usStackDepth	Each task has its own unique stack that is allocated by the kernel to the task when the task is created. The usStackDepth value tells the kernel how large to make the stack.

	The value specifies the number of words the stack can hold, not the number of bytes. For example, on an architecture with a 4-byte stack width, if usStackDepth is passed in as 100, then 400 bytes of stack space will be allocated (100 * 4 bytes). The stack depth multiplied by the stack width must not exceed the maximum value that can be contained in a variable of type size_t. The size of the stack used by the idle task is defined by the application-defined constant configMINIMAL_STACK_SIZE. The value assigned to this constant in the demo application provided for the chosen microcontroller architecture is the minimum recommended for any task on that architecture. If your task uses a lot of stack space, then you must assign a larger value.
pvParameters	Task functions accept a parameter of type 'pointer to void' (void*). The value assigned to pvParameters will be the value passed into the task. This parameter has the type 'pointer to void' to allow the task parameter to effectively (and indirectly by means of casting) receive a parameter of any type. For example, integer types can be passed into a task function by casting the integer to a void pointer at the point the task is created, and then casting the void pointer parameter back to an integer in the task function definition itself.
uxPriority	Defines the priority at which the task will execute. Priorities can be assigned from 0, which is the lowest priority, to (configMAX_PRIORITIES - 1), which is the highest priority. configMAX_PRIORITIES is a user-defined constant. If configUSE_PORT_OPTIMISED_TASK_SELECTION is set to 0, then there is no upper limit to the number of priorities that can be available (other than the limit of the data types used and the RAM available in your microcontroller). To avoid wasting RAM, use the lowest number of priorities required. Passing a uxPriority value above (configMAX_PRIORITIES - 1) will result in the priority assigned to the task being capped silently to the maximum legitimate value.

pxCreatedTask	pxCreatedTask can be used to pass out a handle to the task being created. This handle can then be used to reference the task in API calls that, for example, change the task priority or delete the task. If your application has no use for the task handle, then pxCreatedTask can be set to NULL.

Return Values

pdPASS	Indicates that the task has been created successfully.
errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY	Indicates that the task could not be created because there was insufficient heap memory available for FreeRTOS to allocate the task data structures and stack. If heap_1.c, heap_2.c, or heap_4.c are included in the project, then the total amount of heap available is defined by configTOTAL_HEAP_SIZE in FreeRTOSConfig.h, and failure to allocate memory can be trapped using the vApplicationMallocFailedHook() callback (or 'hook') function, and the amount of free heap memory remaining can be queried using the xPortGetFreeHeapSize() API function. If heap_3.c is included in the project, then the total heap size is defined by the linker configuration.

Notes

configSUPPORT_DYNAMIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h, or simply left undefined, for this function to be available.

Example

```
/* Define a structure called xStruct and a variable of type xStruct. These are just used to
 demonstrate a parameter being passed into a task function. */

typedef struct A_STRUCT

{

    char cStructMember1;

    char cStructMember2;
```

```
} xStruct;

/* Define a variable of the type xStruct to pass as the task parameter. */

xStruct xParameter = { 1, 2 };

/* Define the task that will be created. Note the name of the function that implements the
 task is used as the first parameter in the call to xTaskCreate() below. */

void vTaskCode( void * pvParameters )

{

    xStruct *pxParameters;

    /* Cast the void * parameter back to the required type. */

    pxParameters = ( xStruct * ) pvParameters;

    /* The parameter can now be accessed as expected. */

    if( pxParameters->cStructMember1 != 1 )

    {

        /* Etc. */

    }

    /* Enter an infinite loop to perform the task processing. */

    for( ;; )

    {

        /* Task code goes here. */

    }

}

/* Define a function that creates a task. This could be called either
before or after the scheduler has been started. */

void vAnotherFunction( void )

{

    TaskHandle_t xHandle;

    /* Create the task. */

    if( xTaskCreate(vTaskCode, /* Pointer to the function that implements the task. */
                Demo task, /* Text name given to the task. */ STACK_SIZE, /* The size of
 the stack that should be created for the task. This is defined in words, not bytes. */
                    (void*) &xParameter,/* A reference to xParameters is used as the task
 parameter. This is cast to a void * to prevent compiler warnings. */
         TASK_PRIORITY, /* The priority to assign to the newly created task.*/ &xHandle /*
 The handle to the task being created will be placed in xHandle. */) != pdPASS )

    {

       /* The task could not be created because there was insufficient heap memory
 remaining. If heap_1.c, heap_2.c, or heap_4.c are included in the project, then this
 situation can be trapped using the vApplicationMallocFailedHook() callback (or 'hook')
```

```
 function, and the amount of FreeRTOS heap memory that remains unallocated can be queried
 using the xPortGetFreeHeapSize() API function.*/

    }

    else

    {

        /* The task was created successfully. The handle can now be used in other API
 functions (for example, to change the priority of the task).*/

        vTaskPrioritySet( xHandle, 2 );

    }

}
```

xTaskCreateStatic()

```
#include "FreeRTOS.h"

#include "task.h"

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

Summary

Creates a new instance of a task.

Each task requires RAM that is used to hold the task state (the task control block or TCB) and used by the task as its stack. If a task is created using xTaskCreate(), then the required RAM is automatically allocated from the FreeRTOS heap. If a task is created using xTaskCreateStatic(), then the RAM is provided by the application writer, which results in two additional function parameters, but allows the RAM to be statically allocated at compile time.

Newly created tasks are initially placed in the Ready state, but will immediately become the Running state task if there are no higher priority tasks that are able to run.

Tasks can be created both before and after the scheduler has been started.

Parameters

pvTaskCode	Tasks are simply C functions that never exit and, as such, are normally implemented as an infinite loop. The pvTaskCode parameter is simply a pointer to the function (in effect, just the function name) that implements the task.
pcName	A descriptive name for the task. This is mainly used to facilitate debugging, but can also be used in a call to xTaskGetHandle() to obtain a task handle. The application-defined constant configMAX_TASK_NAME_LEN defines the maximum length of the name in characters, including the NULL terminator. Supplying a string longer than this maximum will result in the string being silently truncated.
ulStackDepth	The puxStackBuffer parameter is used to pass an array of StackType_t variables into xTaskCreateStatic(). ulStackDepth must be set to the number of indexes in the array.

pvParameters	Task functions accept a parameter of type 'pointer to void' (void*). The value assigned to pvParameters will be the value passed into the task. This parameter has the type 'pointer to void' to allow the task parameter to effectively (and indirectly by means of casting) receive a parameter of any type. For example, integer types can be passed into a task function by casting the integer to a void pointer at the point the task is created, then by casting the void pointer parameter back to an integer in the task function definition itself.
uxPriority	Defines the priority at which the task will execute. Priorities can be assigned from 0, which is the lowest priority, to (configMAX_PRIORITIES - 1), which is the highest priority. configMAX_PRIORITIES is a user-defined constant. If configUSE_PORT_OPTIMISED_TASK_SELECTION is set to 0, then there is no upper limit to the number of priorities that can be available (other than the limit of the data types used and the RAM available in your microcontroller). To avoid wasting RAM, use the lowest number of priorities required. Passing a uxPriority value above (configMAX_PRIORITIES - 1) will result in the priority assigned to the task being capped silently to the maximum legitimate value.
puxStackBuffer	Must point to an array of StackType_t variables that has at least ulStackDepth indexes (see the ulStackDepth parameter above). The array will be used as the created task's stack, so must be persistent (not declared within the stack frame created by a function, or in any other memory that can legitimately be overwritten as the application executes).
pxTaskBuffer	Must point to a variable of type StaticTask_t. The variable will be used to hold the created task's data structures (TCB), so it must be persistent (not declared within the stack frame created by a function, or in any other memory that can legitimately be overwritten as the application executes).

Return Values

NULL	The task could not be created because puxStackBuffer or pxTaskBuffer was NULL.

Any other value	If a non-NULL value is returned, then the task was created and the returned value is the handle of the created task.

Notes

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

Example

```
/* 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 depends 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 because 1 is passed in the pvParameters
 parameter in the call to xTaskCreateStatic(). */

    configASSERT( ( uint32_t ) pvParameters == 1UL );

    for( ;; )

    {

        /* Task code goes here. */

    }

}

/* Function that creates a task. */

void vFunction( 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, /* The number of indexes in the xStack
array. */ ( 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 );

}
```

xTaskCreateRestricted()

```
#include "FreeRTOS.h"

#include "task.h"

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

Summary

This function is intended for advanced users only. It is only relevant to FreeRTOS ports that make use of a Memory Protection Unit (MPU).

Create an MPU restricted task.

Newly created tasks are initially placed in the Ready state, but will immediately become the Running state task if there are no higher priority tasks that are able to run.

Tasks can be created both before and after the scheduler has been started.

Parameters

pxTaskDefinition	Pointer to a structure that defines the task. The structure is described in the notes.
pxCreatedTask	pxCreatedTask can be used to pass out a handle to the task being created. This handle can then be used to reference the task in API calls that, for example, change the task priority or delete the task. If your application has no use for the task handle, then you can set pxCreatedTask to NULL.

Return Values

pdPASS	Indicates that the task has been created successfully.
Any other value	Indicates that the task could not be created as specified, probably because there is insufficient FreeRTOS heap memory available to allocate the task data structures. If heap_1.c, heap_2.c, or heap_4.c are included in the project, then the total

	amount of heap available is defined by configTOTAL_HEAP_SIZE in FreeRTOSConfig.h, and failure to allocate memory can be trapped using the vApplicationMallocFailedHook() callback (or 'hook') function, and the amount of free heap memory remaining can be queried using the xPortGetFreeHeapSize() API function. If heap_3.c is included in the project, then the total heap size is defined by the linker configuration.

```
typedef struct xTASK_PARAMTERS

{

    TaskFunction_t pvTaskCode;

    const signed char * const pcName;

    unsigned short usStackDepth;

    void *pvParameters;

    UBaseType_t uxPriority;

    portSTACK_TYPE *puxStackBuffer;

    MemoryRegion_t xRegions[ portNUM_CONFIGURABLE_REGIONS ];

} TaskParameters_t;

/* ....where MemoryRegion_t is defined as: */

typedef struct xMEMORY_REGION

{

    void *pvBaseAddress;

    unsigned long ulLengthInBytes;

    unsigned long ulParameters;

} MemoryRegion_t;
```

Notes

Here is a description of the structure members.

pvTaskCode, portPRIVILEGE_BIT, uxPriority

These structure members are equivalent to the xTaskCreate() API function parameters that have the same names. Unlike standard FreeRTOS tasks, protected tasks can be created in either User (unprivileged) or Supervisor (privileged) modes, and the uxPriority structure member is used to control this option. To create a task in User mode, uxPriority is set to equal the priority at which the task is to be created. To create a task in Supervisor mode, uxPriority is set to equal the priority at which the task is to be created and to have its most significant bit set. The macro is provided for this purpose. For example,

to create a User mode task at priority three, set uxPriority to equal 3. To create a Supervisor mode task at priority three, set uxPriority to equal (3 | portPRIVILEGE_BIT).

puxStackBuffer

The xTaskCreate() API function will automatically allocate a stack for use by the task being created. The restrictions imposed by using an MPU mean that the xTaskCreateRestricted() function cannot do the same. Instead, the stack used by the task being created must be statically allocated and passed into the xTaskCreateRestricted() function using the puxStackBuffer parameter. Each time a restricted task is switched in (transitioned to the Running state), the MPU is dynamically reconfigured to define an MPU region that provides the task read and write access to its own stack. Therefore, the statically allocated task stack must comply with the size and alignment restrictions imposed by the MPU. In particular, the size and alignment of each region must both be equal to the same power of two value. Statically declaring a stack buffer allows the alignment to be managed using compiler extensions, and allows the linker to take care of stack placement, which it will do as efficiently as possible. For example, if using GCC, a stack can be declared and correctly aligned using the following syntax:

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

MemoryRegion_t

An array of MemoryRegion_t structures. Each MemoryRegion_t structure defines a single MPU memory region for use by the task being created. The Cortex-M3 FreeRTOS-MPU port defines portNUM_CONFIGURABLE_REGIONS to be 3. Three regions can be defined when the task is created. The regions can be redefined at runtime by using the vTaskAllocateMPURegions() function. The pvBaseAddress and ulLengthInBytes members are self-explanatory as the start of the memory region and the length of the memory region, respectively. ulParameters defines how the task is permitted to access the memory region being defined. It can take the bitwise OR of the following values:

- portMPU_REGION_READ_WRITE - portMPU_REGION_PRIVILEGED_READ_ONLY - portMPU_REGION_READ_ONLY - portMPU_REGION_PRIVILEGED_READ_WRITE - portMPU_REGION_CACHEABLE_BUFFERABLE - portMPU_REGION_EXECUTE_NEVER

Example

```
/* Declare the stack that will be used by the protected task being created. The stack
 alignment must match its size and be a power of 2. So, if 128 words are reserved for
 the stack, then it must be aligned on a ( 128 * 4 ) byte boundary. This example uses GCC
 syntax. */

static portSTACK_TYPE xTaskStack[ 128 ]  __attribute__((aligned(128*4)));

/* Declare an array that will be accessed by the protected task being created. The task
 should only be able to read from the array, not write to it. */

char cReadOnlyArray[ 512 ] __attribute__((aligned(512)));

/* Fill in a TaskParameters_t structure to define the task. This is the structure passed to
 the xTaskCreateRestricted() function. */

static const TaskParameters_t xTaskDefinition =

{

    vTaskFunction, /* pvTaskCode */

    "A task", /* pcName */

    128, /* usStackDepth - defined in words, not bytes. */
```

```
    NULL, /* pvParameters */

    1, /* uxPriority - priority 1, start in User mode. */

    xTaskStack, /* puxStackBuffer - the array to use as the task stack. */

    /* xRegions - In this case, only one of the three user-definable regions is actually
 used. The parameters are used to set the region to read-only. */

    {

        /* Base address Length Parameters */

        { cReadOnlyArray, 512, portMPU_REGION_READ_ONLY },

        { 0, 0, 0 },

        { 0, 0, 0 },

    }

};

void main( void )

{

    /* Create the task defined by xTaskDefinition. NULL is used as the second parameter
 because a task handle is not required. */

    xTaskCreateRestricted( &xTaskDefinition, NULL );

    /* Start the scheduler. */

    vTaskStartScheduler();

    /* Should not reach here! */

}
```

vTaskDelay()

```
#include "FreeRTOS.h"

#include "task.h"

void vTaskDelay( TickType_t xTicksToDelay );
```

Summary

Places the task that calls vTaskDelay() into the Blocked state for a fixed number of tick interrupts.

Specifying a delay period of zero ticks will not result in the calling task being placed into the Blocked state, but will result in the calling task yielding to any Ready state tasks that share its priority. Calling vTaskDelay(0) is equivalent to calling taskYIELD().

Parameters

xTicksToDelay	The number of tick interrupts that the calling task will remain in the Blocked state before being transitioned back into the Ready state. For example, if a task called vTaskDelay(100) when the tick count was 10,000, then it would immediately enter the Blocked state and remain in the Blocked state until the tick count reached 10,100. Any time that remains between vTaskDelay() being called and the next tick interrupt occurring counts as one complete tick period. Therefore, the highest time resolution that can be achieved when specifying a delay period is, in the worst case, equal to one complete tick interrupt period. The macro pdMS_TO_TICKS() can be used to convert milliseconds into ticks, as shown in the example in this section.

Return Values

None.

Notes

INCLUDE_vTaskDelay must be set to 1 in FreeRTOSConfig.h for the vTaskDelay() API function to be available.

Example

```
void vAnotherTask( void * pvParameters )

{

    for( ;; )

    {

        /* Perform some processing here. */

        …

        /* Enter the Blocked state for 20 tick interrupts. The time spent in the Blocked
 state depends on the tick frequency. */

        vTaskDelay( 20 );

        /* 20 ticks will have passed since the first call to vTaskDelay() was executed. */

        /* Enter the Blocked state for 20 milliseconds. Using the pdMS_TO_TICKS() macro
 means the tick frequency can change without affecting the time spent in the blocked state
 (other than due to the resolution of the tick frequency). */

        vTaskDelay( pdMS_TO_TICKS( 20 ) );

    }

}
```

vTaskDelayUntil()

```
#include "FreeRTOS.h"

#include "task.h"

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

Summary

Places the task that calls vTaskDelayUntil() into the Blocked state until an absolute time is reached.

Periodic tasks can use vTaskDelayUntil() to achieve a constant execution frequency.

Differences Between vTaskDelay() and vTaskDelayUntil()

vTaskDelay() results in the calling task entering and remaining in the Blocked state for the specified number of ticks from the time vTaskDelay() was called. The time at which the task that called vTaskDelay() exits the Blocked state is relative to when vTaskDelay() was called.

vTaskDelayUntil() results in the calling task entering and then remaining in the Blocked state until an absolute time has been reached. The task that called vTaskDelayUntil() exits the Blocked state exactly at the specified time, not at a time that is relative to when vTaskDelayUntil() was called.

Parameters

pxPreviousWakeTime	This parameter is named on the assumption that vTaskDelayUntil() is being used to implement a task that executes periodically and with a fixed frequency. In this case, pxPreviousWakeTime holds the time at which the task last left the Blocked state (was woken up). This time is used as a reference point to calculate the time at which the task should next leave the Blocked state. The variable pointed to by pxPreviousWakeTime is updated automatically within the vTaskDelayUntil() function. It would not normally be modified by the application code, other than when the variable is first initialized. The example in this section demonstrates how the initialization is performed.

xTimeIncrement	This parameter is also named on the assumption that vTaskDelayUntil() is being used to implement a task that executes periodically and with a fixed frequency (the frequency being set by the xTimeIncrement value). xTimeIncrement is specified in ticks. The pdMS_TO_TICKS() macro can be used to convert milliseconds to ticks.

Return Values

None.

Notes

INCLUDE_vTaskDelayUntil must be set to 1 in FreeRTOSConfig.h for the vTaskDelay() API function to be available.

Example

```
/* Define a task that performs an action every 50 milliseconds. */

void vCyclicTaskFunction( void * pvParameters )

{

    TickType_t xLastWakeTime;

    const TickType_t xPeriod = pdMS_TO_TICKS( 50 );

    /* The xLastWakeTime variable needs to be initialized with the current tick count. Note
 that this is the only time the variable is written to explicitly. After this assignment,
 xLastWakeTime is updated automatically internally within vTaskDelayUntil(). */

    xLastWakeTime = xTaskGetTickCount();

    /* Enter the loop that defines the task behavior. */

    for( ;; )

    {

        /* This task should execute every 50 milliseconds. Time is measured in ticks.
 The pdMS_TO_TICKS macro is used to convert milliseconds into ticks. xLastWakeTime is
 automatically updated within vTaskDelayUntil() so is not explicitly updated by the task.
 */

        vTaskDelayUntil( &xLastWakeTime, xPeriod );

        /* Perform the periodic actions here. */

     }
```

```
}
```

vTaskDelete()

The following code listing shows the vTaskDelete() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

void vTaskDelete( TaskHandle_t pxTask );
```

Summary

Deletes an instance of a task that was previously created using a call to xTaskCreate() or xTaskCreateStatic().

Deleted tasks no longer exist so they cannot enter the Running state.

Do not attempt to use a task handle to reference a task that has been deleted.

When a task is deleted, it is the responsibility of the idle task to free the memory that had been used to hold the deleted task's stack and data structures (task control block). Therefore, if an application makes use of the vTaskDelete() API function, it is vital that the application also ensures the idle task is not starved of processing time. (The idle task must be allocated time in the Running state.)

Only memory that is allocated to a task by the kernel itself is automatically freed when a task is deleted. Memory or any other resource that the application (rather than the kernel) allocates to a task must be explicitly freed by the application when the task is deleted.

Parameters

pxTask	The handle of the task being deleted (the subject 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(). A task can delete itself by passing NULL in place of a valid task handle.

Return Values

None.

Example

```
void vAnotherFunction( void )

{

    TaskHandle_t xHandle;

    /* Create a task, storing the handle to the created task in xHandle.*/

    if(xTaskCreate(vTaskCode, "Demo task", STACK_SIZE, NULL, PRIORITY, &xHandle /* The
 address of xHandle is passed in as the last parameter to xTaskCreate() to obtain a handle
 to the task being created. */ ) != pdPASS )

    {

        /* The task could not be created because there was not enough FreeRTOS heap memory
 available for the task data structures and stack to be allocated. */

    }

    else

    {

        /* Delete the task just created. Use the handle passed out of  xTaskCreate() to
 reference the subject task. */

        vTaskDelete( xHandle );

    }

    /* Delete the task that called this function by passing NULL in as the vTaskDelete()
 parameter. The same task (this task) could also be deleted by passing in a valid handle to
 itself. */

    vTaskDelete( NULL );

}
```

taskDISABLE_INTERRUPTS()

The following listing shows the taskDISABLE_INTERRUPTS() macro prototype.

```
#include "FreeRTOS.h"

#include "task.h"

void taskDISABLE_INTERRUPTS( void );
```

Summary

If the FreeRTOS port being used does not make use of the configMAX_SYSCALL_INTERRUPT_PRIORITY (or configMAX_API_CALL_INTERRUPT_PRIORITY, depending on the port) kernel configuration constant, then calling taskDISABLE_INTERRUPTS() will leave interrupts globally disabled.

If the FreeRTOS port being used does make use of the configMAX_SYSCALL_INTERRUPT_PRIORITY kernel configuration constant, then calling taskDISABLE_INTERRUPTS() will leave interrupts at and below the interrupt priority set by configMAX_SYSCALL_INTERRUPT_PRIORITY disabled, and all higher priority interrupt enabled.

configMAX_SYSCALL_INTERRUPT_PRIORITY is normally defined in FreeRTOSConfig.h.

Calls to taskDISABLE_INTERRUPTS() and taskENABLE_INTERRUPTS() are not designed to nest. For example, if taskDISABLE_INTERRUPTS() is called twice, a single call to taskENABLE_INTERRUPTS() will still result in interrupts becoming enabled. If nesting is required, then use taskENTER_CRITICAL() and taskEXIT_CRITICAL() in place of taskDISABLE_INTERRUPTS() and taskENABLE_INTERRUPTS(), respectively.

Some FreeRTOS API functions use critical sections that will re-enable interrupts if the critical section nesting count is zero, even if interrupts were disabled by a call to taskDISABLE_INTERRUPTS() before the API function was called. We do not recommend calling FreeRTOS API functions when interrupts have already been disabled.

Parameters

None.

Return Values

None.

taskENABLE_INTERRUPTS()

The following listing shows the taskENABLE_INTERRUPTS() macro prototype.

```
#include "FreeRTOS.h"

#include "task.h"

void taskENABLE_INTERRUPTS( void );
```

Summary

Calling taskENABLE_INTERRUPTS() will result in all interrupt priorities being enabled.

Calls to taskDISABLE_INTERRUPTS() and taskENABLE_INTERRUPTS() are not designed to nest. For example, if taskDISABLE_INTERRUPTS() is called twice, a single call to taskENABLE_INTERRUPTS() will still result in interrupts becoming enabled. If nesting is required, then use taskENTER_CRITICAL() and taskEXIT_CRITICAL() in place of taskDISABLE_INTERRUPTS() and taskENABLE_INTERRUPTS(), respectively.

Some FreeRTOS API functions use critical sections that will re-enable interrupts if the critical section nesting count is zero, even if interrupts were disabled by a call to taskDISABLE_INTERRUPTS() before the API function was called. We do not recommend calling FreeRTOS API functions when interrupts have already been disabled.

Parameters

None.

Return Values

None.

taskENTER_CRITICAL()

The following listing shows the taskENTER_CRITICAL macro prototype.

```
#include "FreeRTOS.h"

#include "task.h"

void taskENTER_CRITICAL( void );
```

Summary

Critical sections are entered by calling taskENTER_CRITICAL() and exited by calling taskEXIT_CRITICAL().

taskENTER_CRITICAL() must not be called from an interrupt service routine. See taskENTER_CRITICAL_FROM_ISR() for an interrupt-safe equivalent.

The taskENTER_CRITICAL() and taskEXIT_CRITICAL() macros provide a basic critical section implementation that works by simply disabling interrupts, either globally or up to a specific interrupt priority level. For information about creating a critical section without disabling interrupts, see the vTaskSuspendAll() API function.

If the FreeRTOS port being used does not make use of the configMAX_SYSCALL_INTERRUPT_PRIORITY kernel configuration constant, then calling taskENTER_CRITICAL() will leave interrupts globally disabled.

If the FreeRTOS port being used does make use of the configMAX_SYSCALL_INTERRUPT_PRIORITY (or configMAX_API_CALL_INTERRUPT_PRIORITY, depending on the port) kernel configuration constant, then calling taskENTER_CRITICAL() will leave interrupts at and below the interrupt priority set by configMAX_SYSCALL_INTERRUPT_PRIORITY disabled and all higher priority interrupt enabled.

Preemptive context switches only occur inside an interrupt, so will not occur when interrupts are disabled. Therefore, the task that called taskENTER_CRITICAL() is guaranteed to remain in the Running state until the critical section is exited, unless the task explicitly attempts to block or yield (which it should not do from inside a critical section).

Calls to taskENTER_CRITICAL() and taskEXIT_CRITICAL() are designed to nest. Therefore, a critical section will only be exited when one call to taskEXIT_CRITICAL() has been executed for every preceding call to taskENTER_CRITICAL().

Critical sections must be kept very short. Otherwise, they will adversely affect interrupt response times. Every call to taskENTER_CRITICAL() must be closely paired with a call to taskEXIT_CRITICAL().

FreeRTOS API functions must not be called from within a critical section.

Parameters

None.

Return Values

None.

Example

The following listing shows the use of taskENTER_CRITICAL() and taskEXIT_CRITICAL().

```
/* A function that makes use of a critical section. */

void vDemoFunction( void )

{

    /* Enter the critical section. In this example, this function is itself called  from
 within a critical section, so entering this critical section will result in a nesting
 depth of 2. */

    taskENTER_CRITICAL();

    /* Perform the action that is being protected by the critical section here. */

    /* Exit the critical section. In this example, this function is itself called from a
 critical section, so this call to taskEXIT_CRITICAL() will decrement the nesting count by
 one, but not result in interrupts becoming enabled. */

    taskEXIT_CRITICAL();

}

/* A task that calls vDemoFunction() from within a critical section. */

void vTask1( void * pvParameters )

{

    for( ;; )

    {

        /* Perform some functionality here. */

        /* Call taskENTER_CRITICAL() to create a critical section. */

        taskENTER_CRITICAL();

        /* Execute the code that requires the critical section here. */

        /* Calls to taskENTER_CRITICAL() can be nested so it is safe to call a function
 that includes its own calls to taskENTER_CRITICAL() and taskEXIT_CRITICAL(). */

        vDemoFunction();

        /* The operation that required the critical section is complete so exit the
 critical section. After this call to taskEXIT_CRITICAL(), the nesting depth will be zero,
 so interrupts will have been re-enabled. */

        taskEXIT_CRITICAL();

    }

}
```

taskENTER_CRITICAL_FROM_ISR()

The following listing shows the taskENTER_CRITICAL_FROM_ISR() macro prototype.

```
#include "FreeRTOS.h"

#include "task.h"

UBaseType_t taskENTER_CRITICAL_FROM_ISR( void );
```

Summary

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

In an ISR, critical sections are entered by calling taskENTER_CRITICAL_FROM_ISR() and exited by calling taskEXIT_CRITICAL_FROM_ISR().

The taskENTER_CRITICAL_FROM_ISR() and taskEXIT_CRITICAL_FROM_ISR() macros provide a basic critical section implementation that works by simply disabling interrupts, either globally or up to a specific interrupt priority level.

If the FreeRTOS port being used supports interrupt nesting, then calling taskENTER_CRITICAL_FROM_ISR() will disable interrupts at and below the interrupt priority set by the configMAX_SYSCALL_INTERRUPT_PRIORITY (or configMAX_API_CALL_INTERRUPT_PRIORITY) kernel configuration constant, and leave all other interrupt priorities enabled. If the FreeRTOS port being used does not support interrupt nesting, then taskENTER_CRITICAL_FROM_ISR() and taskEXIT_CRITICAL_FROM_ISR() will have no effect.

Calls to taskENTER_CRITICAL_FROM_ISR() and taskEXIT_CRITICAL_FROM_ISR() are designed to nest, but the semantics of how the macros are used is different from the taskENTER_CRITICAL() and taskEXIT_CRITICAL() equivalents.

Critical sections must be kept very short. Otherwise, they will adversely affect the response times of higher priority interrupts that would otherwise nest. Every call to taskENTER_CRITICAL_FROM_ISR() must be closely paired with a call to taskEXIT_CRITICAL_FROM_ISR().

FreeRTOS API functions must not be called from within a critical section.

Parameters

None.

Return Values

The interrupt mask state at the time taskENTER_CRITICAL_FROM_ISR() is called is returned. The return value must be saved so it can be passed into the matching call to taskEXIT_CRITICAL_FROM_ISR().

Example

The following listing shows the use of taskENTER_CRITICAL_FROM_ISR() and taskEXIT_CRITICAL_FROM_ISR().

```
/* A function called from an ISR. */

void vDemoFunction( void )

{

    UBaseType_t uxSavedInterruptStatus;

    /* Enter the critical section. In this example, this function is itself called from
 within a critical section, so entering this critical section will result in a nesting
 depth of 2. Save the value returned by taskENTER_CRITICAL_FROM_ISR() into a local stack
 variable so it can be passed into taskEXIT_CRITICAL_FROM_ISR(). */
    uxSavedInterruptStatus = taskENTER_CRITICAL_FROM_ISR();

    /* Perform the action that is being protected by the critical section here. */

    /* Exit the critical section. In this example, this function is itself called
 from a critical section, so interrupts will have already been disabled before a value
 was stored in uxSavedInterruptStatus. Therefore, passing uxSavedInterruptStatus into
 taskEXIT_CRITICAL_FROM_ISR() will not result in interrupts being re-enabled. */

    taskEXIT_CRITICAL_FROM_ISR( uxSavedInterruptStatus );

}

/* A task that calls vDemoFunction() from within an interrupt service routine. */

void vDemoISR( void )

{

    UBaseType_t uxSavedInterruptStatus;

    /* Call taskENTER_CRITICAL_FROM_ISR() to create a critical section, saving the returned
 value into a local stack. */

    uxSavedInterruptStatus = taskENTER_CRITICAL_FROM_ISR();

    /* Execute the code that requires the critical section here. */

    /* Calls to taskENTER_CRITICAL_FROM_ISR() can be nested so it is safe to
 call a function that includes its own calls to taskENTER_CRITICAL_FROM_ISR() and
 taskEXIT_CRITICAL_FROM_ISR(). */

    vDemoFunction();

    /* The operation that required the critical section is complete, so exit the critical
 section. Assuming interrupts were enabled on entry to this ISR, the value saved in
 uxSavedInterruptStatus will result in interrupts being re-enabled.*/

    taskEXIT_CRITICAL_FROM_ISR( uxSavedInterruptStatus );

}
```

taskEXIT_CRITICAL()

The following listing shows the taskEXIT_CRITICAL() macro prototype.

```
#include "FreeRTOS.h"

#include "task.h"

void taskEXIT_CRITICAL( void );
```

Summary

Critical sections are entered by calling taskENTER_CRITICAL() and exited by calling taskEXIT_CRITICAL().

taskEXIT_CRITICAL() must not be called from an interrupt service routine. See taskEXIT_CRITICAL_FROM_ISR() for an interrupt-safe equivalent.

The taskENTER_CRITICAL() and taskEXIT_CRITICAL() macros provide a basic critical section implementation that works by simply disabling interrupts, either globally or up to a specific interrupt priority level.

If the FreeRTOS port being used does not make use of the configMAX_SYSCALL_INTERRUPT_PRIORITY kernel configuration constant, then calling taskENTER_CRITICAL() will leave interrupts globally disabled.

If the FreeRTOS port being used does make use of the configMAX_SYSCALL_INTERRUPT_PRIORITY kernel configuration constant, then calling taskENTER_CRITICAL() will leave interrupts at and below the interrupt priority set by configMAX_SYSCALL_INTERRUPT_PRIORITY disabled, and all higher priority interrupt enabled.

Preemptive context switches only occur inside an interrupt, so will not occur when interrupts are disabled. Therefore, the task that called taskENTER_CRITICAL() is guaranteed to remain in the Running state until the critical section is exited, unless the task explicitly attempts to block or yield (which it should not do from inside a critical section).

Calls to taskENTER_CRITICAL() and taskEXIT_CRITICAL() are designed to nest. Therefore, a critical section will only be exited when one call to taskEXIT_CRITICAL() has been executed for every preceding call to taskENTER_CRITICAL().

Critical sections must be kept very short. Otherwise, they will adversely affect interrupt response times. Every call to taskENTER_CRITICAL() must be closely paired with a call to taskEXIT_CRITICAL().

FreeRTOS API functions must not be called from within a critical section.

Parameters

None.

Return Values

None.

taskEXIT_CRITICAL_FROM_ISR()

The following listing shows the taskEXIT_CRITICAL_FROM_ISR() macro prototype.

```
#include "FreeRTOS.h"

#include "task.h"

void taskENTER_CRITICAL_FROM_ISR( UBaseType_t uxSavedInterruptStatus);
```

Summary

Exits a critical section that was entered by calling taskENTER_CRITICAL_FROM_ISR().

In an ISR, critical sections are entered by calling taskENTER_CRITICAL_FROM_ISR() and exited by calling taskEXIT_CRITICAL_FROM_ISR().

The taskENTER_CRITICAL_FROM_ISR() and taskEXIT_CRITICAL_FROM_ISR() macros provide a basic critical section implementation that works by simply disabling interrupts, either globally or up to a specific interrupt priority level.

If the FreeRTOS port being used supports interrupt nesting, then calling taskENTER_CRITICAL_FROM_ISR() will disable interrupts at and below the interrupt priority set by the configMAX_SYSCALL_INTERRUPT_PRIORITY (or configMAX_API_CALL_INTERRUPT_PRIORITY) kernel configuration constant and leave all other interrupt priorities enabled. If the FreeRTOS port being used does not support interrupt nesting, then taskENTER_CRITICAL_FROM_ISR() and taskEXIT_CRITICAL_FROM_ISR() will have no effect.

Calls to taskENTER_CRITICAL_FROM_ISR() and taskEXIT_CRITICAL_FROM_ISR() are designed to nest, but the semantics of how the macros are used is different to the taskENTER_CRITICAL() and taskEXIT_CRITICAL() equivalents.

Critical sections must be kept very short. Otherwise, they will adversely affect the response times of higher priority interrupts that would otherwise nest. Every call to taskENTER_CRITICAL_FROM_ISR() must be closely paired with a call to taskEXIT_CRITICAL_FROM_ISR().

FreeRTOS API functions must not be called from within a critical section.

Parameters

uxSavedInterruptStatus	The value returned from the matching call to taskENTER_CRITICAL_FROM_ISR() must be used as the uxSavedInterruptStatus value.

Return Values

None.

xTaskGetApplicationTaskTag()

The following listing shows the xTaskGetApplicationTaskTag() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

TaskHookFunction_t xTaskGetApplicationTaskTag( TaskHandle_t xTask );
```

Summary

This function is intended for advanced users only.

Returns the 'tag' value associated with a task. The meaning and use of the tag value is defined by the application writer. The kernel itself will not normally access the tag value.

Parameters

xTask	The handle of the task being queried. This is the subject task. A task can obtain its own tag value by either using its own task handle or by using NULL in place of a valid task handle.

Return Values

The 'tag' value of the task being queried.

Notes

The tag value can be used to hold a function pointer. When this is done, the function assigned to the tag value can be called using the xTaskCallApplicationTaskHook() API function. This technique is in effect assigning a callback function to the task. It is common for such a callback to be used in combination with the traceTASK_SWITCHED_IN() macro to implement an execution trace feature.

configUSE_APPLICATION_TASK_TAG must be set to 1 in FreeRTOSConfig.h for xTaskGetApplicationTaskTag() to be available.

Example

```
/* In this example, an integer is set as the task tag value. */
```

```
void vATask( void *pvParameters )

{

    /* Assign a tag value of 1 to the currently executing task. The (void *) cast is used
 to prevent compiler warnings. */

    vTaskSetApplicationTaskTag( NULL, ( void * ) 1 );

    for( ;; )

    {

        /* Rest of task code goes here. */

    }

}

void vAFunction( void )

{

    TaskHandle_t xHandle;

    long lReturnedTaskHandle;

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

    /* Create the task. */

    if( xTaskCreate(vATask, /* Pointer to the function that implements the task. */ "Demo
 task", /* Text name given to the task. */ STACK_SIZE, /* The size of the stack that should
 be created for the task. This is defined in words, not bytes. */ NULL, /* The task does
 not use the parameter. */ TASK_PRIORITY, /* The priority to assign to the newly created
 task. */ &xHandle /* The handle to the task being created will be placed in xHandle. */ )
 == pdPASS )

    {

        /* The task was created successfully. Delay for a short period to allow the task to
 run. */

        vTaskDelay( 100 );

        /* What tag value is assigned to the task? The returned tag value is stored in an
 integer, so cast to an integer to prevent compiler warnings. */

       lReturnedTaskHandle = ( long ) xTaskGetApplicationTaskTag( xHandle );

    }

}
```

xTaskGetCurrentTaskHandle()

The following listing shows the xTaskGetCurrentTaskHandle() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

TaskHandle_t xTaskGetCurrentTaskHandle( void );
```

Summary

Returns the handle of the task that is in the Running state. This will be the handle of the task that called xTaskGetCurrentTaskHandle().

Parameters

None.

Return Values

The handle of the task that called xTaskGetCurrentTaskHandle().

Notes

INCLUDE_xTaskGetCurrentTaskHandle must be set to 1 in FreeRTOSConfig.h for xTaskGetCurrentTaskHandle() to be available.

xTaskGetIdleTaskHandle()

The following listing shows the xTaskGetIdleTaskHandle() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

TaskHandle_t xTaskGetIdleTaskHandle( void );
```

Summary

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

Parameters

None.

Return Values

The handle of the Idle task.

Notes

INCLUDE_xTaskGetIdleTaskHandle must be set to 1 in FreeRTOSConfig.h for xTaskGetIdleTaskHandle() to be available.

xTaskGetHandle()

The following listing shows the xTaskGetHandle() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

TaskHandle_t xTaskGetHandle( const char *pcNameToQuery );
```

Summary

Tasks are created using xTaskCreate() or xTaskCreateStatic(). Both functions have a parameter called pcName that is used to assign a human-readable text name to the task being created. xTaskGetHandle() looks up and returns a task's handle from the task's human-readable text name.

Parameters

pcNameToQuery	The name of the task being queried. The name is specified as a standard NULL-terminated C string.

Return Values

If a task has the exact same name as specified by the pcNameToQuery parameter, then the handle of the task will be returned. If no tasks have the name specified by the pcNameToQuery parameter, then NULL is returned.

Notes

xTaskGetHandle() can take a relatively long time to complete. For this reason, we recommend that xTaskGetHandle() is only used once for each task name. The task handle returned by xTaskGetHandle() can then be stored for later reuse.

The behavior of xTaskGetHandle() is undefined if there is more than one task that has the same name.

INCLUDE_xTaskGetHandle must be set to 1 in FreeRTOSConfig.h for xTaskGetHandle() to be available.

Example

```
void vATask( void *pvParameters )

{
```

```
    const char *pcNameToLookup = "MyTask";

    TaskHandle_t xHandle;

    /* Find the handle of the task that has the name MyTask, storing the returned handle
 locally so it can be reused later. */

    xHandle = xTaskGetHandle( pcNameToLookup );

    if( xHandle != NULL )

    {

        /* The handle of the task was found and can now be used in any other FreeRTOS API
 function that takes a TaskHandle_t parameter. */

    }

    for( ;; )

    {

        /* The rest of the task code goes here. */

    }

}
```

uxTaskGetNumberOfTasks()

The following listing shows the uxTaskGetNumberOfTasks() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

UBaseType_t uxTaskGetNumberOfTasks( void );
```

Summary

Returns the total number of tasks that exist at the time uxTaskGetNumberOfTasks() is called.

Parameters

None.

Return Values

The value returned is the total number of tasks that are under the control of the FreeRTOS kernel at the time uxTaskGetNumberOfTasks() is called. This is the number of Suspended state tasks, plus the number of Blocked state tasks, plus the number of Ready state tasks, plus the idle task, plus the Running state task.

vTaskGetRunTimeStats()

The following listing shows the vTaskGetRunTimeStats() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

void vTaskGetRunTimeStats( char *pcWriteBuffer );
```

Summary

FreeRTOS can be configured to collect task runtime statistics. Task runtime statistics provide information about the amount of processing time each task has received. Figures are provided as both an absolute time and a percentage of the total application runtime. The vTaskGetRunTimeStats() API function formats the collected runtime statistics into a human-readable table. Columns are generated for the task name, the absolute time allocated to that task, and the percentage of the total application run time allocated to that task. A row is generated for each task in the system, including the Idle task.

Parameters

pcWriteBuffer	A pointer to a character buffer into which the formatted and human-readable table is written. The buffer must be large enough to hold the entire table because no boundary checking is performed.

Return Values

None.

Notes

vTaskGetRunTimeStats() is a utility function that is provided for convenience only. It is not considered part of the kernel. vTaskGetRunTimeStats() obtains its raw data using the xTaskGetSystemState() API function.

configGENERATE_RUN_TIME_STATS and configUSE_STATS_FORMATTING_FUNCTIONS must both be set to 1 in FreeRTOSConfig.h for vTaskGetRunTimeStats() to be available. Setting configGENERATE_RUN_TIME_STATS will also require the application to define the following macros:

portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()	This macro must be provided to initialize whichever peripheral is used to generate the time base. The time base used by the runtime stats must have a higher resolution than the tick interrupt. Otherwise, the gathered statistics might be too inaccurate to be truly useful. We recommend that you make the time base between 10 and 20 times faster than the tick interrupt.
portGET_RUN_TIME_COUNTER_VALUE(), or portALT_GET_RUN_TIME_COUNTER_VALUE(Time)	One of these two macros must be provided to return the current time base value, which is the total time that the application has been running in the chosen time base units. If the first macro is used, it must be defined to evaluate to the current time base value. If the second macro is used, it must be defined to set its 'Time' parameter to the current time base value.

These macros can be defined in FreeRTOSConfig.h.

Example

```
/* The LM3Sxxxx Eclipse demo application already includes a 20KHz timer interrupt.
 The interrupt handler was updated to simply increment a variable called
 ulHighFrequencyTimerTicks each time it executed. portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()
 then sets this variable to 0 and */
/* portGET_RUN_TIME_COUNTER_VALUE() returns its value. To implement this, the following few
 lines are added to FreeRTOSConfig.h. */

extern volatile unsigned long ulHighFrequencyTimerTicks;

/* ulHighFrequencyTimerTicks is already being incremented at 20KHz. Just set its value back
 to 0. */

#define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() (ulHighFrequencyTimerTicks = 0UL )

/* Simply return the high frequency counter value. */

#define portGET_RUN_TIME_COUNTER_VALUE() ulHighFrequencyTimerTicks



 /* The LPC17xx demo application does not include the high frequency interrupt test, so
 portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() is used to configure the timer 0 */

 /* peripheral to generate the time base. portGET_RUN_TIME_COUNTER_VALUE() simply returns
 the current timer 0 counter value. This was implemented using the following functions and
 macros. */

/* Defined in main.c. */

void vConfigureTimerForRunTimeStats( void )

{

    const unsigned long TCR_COUNT_RESET = 2,

    CTCR_CTM_TIMER = 0x00,
```

```
    TCR_COUNT_ENABLE = 0x01;

    /* Power up and feed the timer with a clock. */

    PCONP |= 0x02UL;

    PCLKSEL0 = (PCLKSEL0 & (~(0x3<<2))) | (0x01 << 2);

    /* Reset Timer 0 */

    T0TCR = TCR_COUNT_RESET;

    /* Just count up. */

    T0CTCR = CTCR_CTM_TIMER;

    /* Prescale to a frequency that is good enough to get a decent resolution, but not too
 fast so as to overflow all the time. */

    T0PR = ( configCPU_CLOCK_HZ / 10000UL ) - 1UL;

    /* Start the counter. */

    T0TCR = TCR_COUNT_ENABLE;

}

/* Defined in FreeRTOSConfig.h. */

extern void vConfigureTimerForRunTimeStats( void );

#define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()
vConfigureTimerForRunTimeStats()

 #define portGET_RUN_TIME_COUNTER_VALUE() T0TC



 void vAFunction( void )

 {

    /* Define a buffer that is large enough to hold the generated table. In most cases, the
 buffer will be too large to allocate on the stack. In this example, it is declared static.
 */

    static char cBuffer[ BUFFER_SIZE ];

    /* Pass the buffer into vTaskGetRunTimeStats() to generate the table of data. */

    vTaskGetRunTimeStats( cBuffer );

    /* The generated information can be saved or viewed here. */

}
```

xTaskGetSchedulerState()

The following shows the xTaskGetSchedulerState() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

BaseType_t xTaskGetSchedulerState( void );
```

Summary

Returns a value that indicates the state the scheduler is in at the time xTaskGetSchedulerState() is called.

Parameters

None.

Return Values

taskSCHEDULER_NOT_STARTED	This value will only be returned when xTaskGetSchedulerState() is called before vTaskStartScheduler() has been called.
taskSCHEDULER_RUNNING	Returned if vTaskStartScheduler() has already been called, provided the scheduler is not in the Suspended state.
taskSCHEDULER_SUSPENDED	Returned when the scheduler is in the Suspended state because vTaskSuspendAll() was called.

Notes

INCLUDE_xTaskGetSchedulerState must be set to 1 in FreeRTOSConfig.h for xTaskGetSchedulerState() to be available.

uxTaskGetStackHighWaterMark()

```
#include "FreeRTOS.h"

#include "task.h"

UBaseType_t uxTaskGetStackHighWaterMark( TaskHandle_t xTask );
```

Summary

Each task maintains its own stack, the total size of which is specified when the task is created. uxTaskGetStackHighWaterMark() is used to query how close a task has come to overflowing the stack space allocated to it. This value is called the stack *high water mark*.

Parameters

xTask	The handle of the task whose stack high water mark is being queried (the subject 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(). A task can query its own stack high water mark by passing NULL in place of a valid task handle.

Return Values

The amount of stack used by a task grows and shrinks as the task executes and interrupts are processed. uxTaskGetStackHighWaterMark() returns the minimum amount of remaining stack space that has been available since the task started executing. This is the amount of stack that remained unused when stack usage was at its greatest (or deepest) value. The closer the high water mark is to zero, the closer the task has come to overflowing its stack.

Notes

uxTaskGetStackHighWaterMark() can take a relatively long time to execute. Therefore, we recommend that you limit its use to test and debug builds.

INCLUDE_uxTaskGetStackHighWaterMark must be set to 1 in FreeRTOSConfig.h for uxTaskGetStackHighWaterMark() to be available.

Example

```
void vTask1( void * pvParameters )

{


    UBaseType_t uxHighWaterMark;

    /* Inspect the high water mark of the calling task when the task starts to execute. */

    uxHighWaterMark = uxTaskGetStackHighWaterMark( NULL );

    for( ;; )

    {

        /* Call any function. */

        vTaskDelay( 1000 );

        /* Calling a function will have used some stack space, so it will be */

        /* expected that uxTaskGetStackHighWaterMark() will return a lower value  at this
 point than when it was called on entry to the task function. */

        uxHighWaterMark = uxTaskGetStackHighWaterMark( NULL );

    }

}
```

eTaskGetState()

The following listing shows the eTaskGetState() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

eTaskState eTaskGetState( TaskHandle_t pxTask );
```

Summary

Returns as an enumerated type the state in which a task existed at the time eTaskGetState() was executed.

Parameters

pxTask	The handle of the subject 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().

Return Values

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

State	Return Value
Running	eRunning (the task is querying its own state)
Ready	eReady
Blocked	eBlocked
Suspended	eSuspended
Deleted	eDeleted (the task's structures are waiting to be cleaned up)

Notes

INCLUDE_eTaskGetState must be set to 1 in FreeRTOSConfig.h for the eTaskGetState() API function to be available.

uxTaskGetSystemState()

The following listing shows the uxTaskGetSystemState() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

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

Summary

uxTaskGetSystemState() populates a TaskStatus_t structure for each task in the system. The TaskStatus_t structure contains, among other things, the task's handle, name, priority, state, and total amount of runtime consumed.

Parameters

pxTaskStatusArray	A pointer to an array of TaskStatus_t structures. The array must contain at least one TaskStatus_t structure for each task that is under the control of the RTOS. The number of tasks under the control of the RTOS can be determined using the uxTaskGetNumberOfTasks() API function.
uxArraySize	The size of the array pointed to by the pxTaskStatusArray parameter. The size is specified as the number of indexes in the array (the number of TaskStatus_t structures contained in the array), not by the number of bytes in the array.
pulTotalRunTime	If configGENERATE_RUN_TIME_STATS is set to 1 in FreeRTOSConfig.h, then `*pulTotalRunTime` is set by uxTaskGetSystemState() to the total runtime (as defined by the runtime stats clock) since the target booted. pulTotalRunTime can be set to NULL to omit the total runtime value.

Return Values

The number of TaskStatus_t structures that were populated by uxTaskGetSystemState(). This should equal the number returned by the uxTaskGetNumberOfTasks() API function, but will be zero if the value passed in the uxArraySize parameter was too small.

Notes

This function is intended for debugging use only because its use results in the scheduler remaining suspended for an extended period.

To obtain information about a single task rather than all the tasks in the system, use vTaskGetTaskInfo() instead of uxTaskGetSystemState().

configUSE_TRACE_FACILITY must be defined as 1 in FreeRTOSConfig.h for uxTaskGetSystemState() to be available.

Example

```
/* This example demonstrates how a human-readable table of runtime stats information is
 generated from raw data provided by uxTaskGetSystemState().  The human-readable table
 is written to pcWriteBuffer. (See the vTaskList() API function which actually does just
 this.) */

void vTaskGetRunTimeStats( signed char *pcWriteBuffer )

{

    TaskStatus_t *pxTaskStatusArray;

    volatile UBaseType_t uxArraySize, x;

    unsigned long ulTotalRunTime, ulStatsAsPercentage;

    /* Make sure the write buffer does not contain a string. */

    *pcWriteBuffer = 0x00;

    /* Take a snapshot of the number of tasks in case it changes while this function is
 executing. */

    uxArraySize = uxTaskGetNumberOfTasks();

    /* Allocate a TaskStatus_t structure for each task. An array could be allocated
 statically at compile time. */

    pxTaskStatusArray = pvPortMalloc( uxArraySize * sizeof( TaskStatus_t ));

    if( pxTaskStatusArray != NULL )

    {

          /* Generate raw status information about each task. */

          uxArraySize = uxTaskGetSystemState( pxTaskStatusArray, uxArraySize,
 &ulTotalRunTime );

          /* For percentage calculations. */

          ulTotalRunTime /= 100UL;

          /* Avoid divide by zero errors. */

          if( ulTotalRunTime > 0 )

          {
```

```
              /* For each populated position in the pxTaskStatusArray array, format the
 raw data as human-readable ASCII data. */

              for( x = 0; x < uxArraySize; x++ )

              {

                   /* What percentage of the total runtime has the task used? This will
 always be rounded down to the nearest integer. ulTotalRunTimeDiv100 has already been
 divided by 100. */

                   ulStatsAsPercentage = pxTaskStatusArray[ x ].ulRunTimeCounter /
 ulTotalRunTime;

                   if( ulStatsAsPercentage > 0UL )

                   {

                        sprintf( pcWriteBuffer, "%s\t\t%lu\t\t%lu%%\r\n",
 pxTaskStatusArray[ x ].pcTaskName, pxTaskStatusArray[ x ].ulRunTimeCounter,
 ulStatsAsPercentage );

                   }

                   else

                   {

                       /* If the percentage is zero here, then the task has consumed less
 than 1% of the total runtime. */

                       sprintf( pcWriteBuffer, "%s\t\t%lu\t\t<1%%\r
\n",pxTaskStatusArray[ x ].pcTaskName, pxTaskStatusArray[ x ].ulRunTimeCounter );

                   }

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

               }

             }

             /* The array is no longer needed. Free the memory it consumes. */

             vPortFree( pxTaskStatusArray );

    }

}
```

The following listing shows the use of uxTaskGetSystemState().

```
typedef struct xTASK_STATUS

{

   /* The handle of the task to which the rest of the information in the structure
 relates. */

   TaskHandle_t xHandle;

   /* A pointer to the task's name. This value will be invalid if the task was deleted
 since the structure was populated! */
```

```
   const signed char *pcTaskName;

   /* A number unique to the task. */

   UBaseType_t xTaskNumber;

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

   eTaskState eCurrentState;

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

   UBaseType_t uxCurrentPriority;

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

   UBaseType_t uxBasePriority;

   /* The total runtime allocated to the task so far, as defined by the runtime stats
clock. Only valid when configGENERATE_RUN_TIME_STATS is defined as 1 in FreeRTOSConfig.h.
*/

   unsigned long ulRunTimeCounter;

   /* Points to the lowest address of the task's stack area. */

   StackType_t *pxStackBase;

   /* The minimum amount of stack space that has remained for the task since the task was
created. The closer this value is to zero, the closer the task has come to overflowing its
stack. */

   unsigned short usStackHighWaterMark;

} TaskStatus_t;
```

vTaskGetTaskInfo()

The following shows the vTaskGetTaskInfo() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

void vTaskGetTaskInfo( TaskHandle_t xTask,TaskStatus_t *pxTaskStatus, BaseType_t
 xGetFreeStackSpace, eTaskState eState );
```

Summary

vTaskGetTaskInfo() populates a TaskStatus_t structure for a single task. The TaskStatus_t structure contains, among other things, the task's handle, name, priority, state, and total amount of runtime consumed.

The TaskStatus_t structure is defined in the example.

Parameters

xTask

The handle of the task being queried. 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().

pxTaskStatus

Must point to a variable of type TaskStatus_t, which will contain information about the task being queried.

xGetFreeStackSpace

The TaskStatus_t structure contains a member to report the stack high water mark of the task being queried. The stack high water mark is the minimum amount of stack space that has ever existed for the task, so the closer the number is to zero, the closer the task has come to overflowing its stack. 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.

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, set eState to eInvalid. Otherwise, the value passed in eState will be reported as the task state in the TaskStatus_t structure.

Notes

This function is intended for debugging use only because its use can potentially result in the scheduler remaining suspended for an extended period.

To obtain a TaskStatus_t structure for all the tasks in the system, use uxTaskGetSystemState() in place of vTaskGetTaskInfo().

configUSE_TRACE_FACILITY must bet defined as 1 in FreeRTOSConfig.h for uxTaskGetSystemState() to be available.

Example

```
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 more information about the task. */

    vTaskGetTaskInfo( /* The handle of the task being queried. */ xHandle, /* The
 TaskStatus_t structure to complete with information on xHandle. */ &xTaskDetails, /*
 Include the stack high water mark value in the TaskStatus_t structure. */ pdTRUE, /*
 Include the task state in the TaskStatus_t structure. */ eInvalid );

}
```

pvTaskGetThreadLocalStoragePointer()

The following listing shows the pvTaskGetThreadLocalStoragePointer() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

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

Summary

Thread local storage (TLS) allows the application writer to store values inside a task's control block, making the value specific to (local to) the task itself and allowing each task to have its own unique value.

Each task has its own array of pointers that can be used as thread local storage. The number of indexes in the array is set by the configNUM_THREAD_LOCAL_STORAGE_POINTERS compile time configuration constant in FreeRTOSConfig.h.

pvTaskGetThreadLocalStoragePointer() reads a value from an index in the array, effectively retrieving a thread local value.

Parameters

xTaskToQuery	The handle of the task from which the thread local data is being read. A task can read its own thread local data by using NULL as the parameter value..
xIndex	The index into the thread local storage array from which data is being read.

Return Values

The value read from the task's thread local storage array at index xIndex.

Example

```
uint32_t ulVariable;

/* Read the value stored in index 5 of the calling task's thread local storage array into
 ulVariable. */

ulVariable = ( uint32_t ) pvTaskGetThreadLocalStoragePointer( NULL, 5 );
```

pcTaskGetName()

The following listing shows the pcTaskGetName() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

char * pcTaskGetName( TaskHandle_t xTaskToQuery );
```

Summary

Queries the human-readable text name of a task. A text name is assigned to a task using the pcName parameter of the xTaskCreate() or xTaskCreateStatic() API function call used to create the task.

Parameters

xTaskToQuery	The handle of the task being queried (the subject 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(). A task may query its own name by passing NULL in place of a valid task handle.

Return Values

Task names are standard NULL-terminated C strings. The value returned is a pointer to the subject task's name.

xTaskGetTickCount()

The following listing shows the xTaskGetTickCount() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

TickType_t xTaskGetTickCount( void );
```

Summary

The tick count is the total number of tick interrupts that have occurred since the scheduler was started. xTaskGetTickCount() returns the current tick count value.

Parameters

None.

Return Values

xTaskGetTickCount() always returns the tick count value at the time that xTaskGetTickCount() was called.

Notes

The actual time represented by one tick period depends on the value assigned to configTICK_RATE_HZ in FreeRTOSConfig.h. The pdMS_TO_TICKS() macro can be used to convert a time in milliseconds to ticks.

The tick count will eventually overflow and return to zero. This will not affect the internal operation of the kernel. For example, tasks will always block for the specified period even if the tick count overflows while the task is in the Blocked state. Overflows must be considered by host applications if the application makes direct use of the tick count value.

The frequency at which the tick count overflows depends on both the tick frequency and the data type used to hold the count value. If configUSE_16_BIT_TICKS is set to 1, then the tick count will be held in a 16-bit variable. If configUSE_16_BIT_TICKS is set to 0, then the tick count will be held in a 32-bit variable.

Example

```
void vAFunction( void )

{
```

```
    TickType_t xTime1, xTime2, xExecutionTime;

    /* Get the time the function started. */

    xTime1 = xTaskGetTickCount();

    /* Perform some operation. */

    /* Get the time following the execution of the operation. */

    xTime2 = xTaskGetTickCount();

    /* Approximately how long did the operation take? */

    xExecutionTime = xTime2 - xTime1;

}
```

xTaskGetTickCountFromISR()

The following listing shows the xTaskGetTickCountFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

TickType_t xTaskGetTickCountFromISR( void );
```

Summary

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

The tick count is the total number of tick interrupts that have occurred since the scheduler was started.

Parameters

None.

Return Values

xTaskGetTickCountFromISR() always returns the tick count value at the time xTaskGetTickCountFromISR() is called.

Notes

The actual time represented by one tick period depends on the value assigned to configTICK_RATE_HZ in FreeRTOSConfig.h. The pdMS_TO_TICKS() macro can be used to convert a time in milliseconds to ticks.

The tick count will eventually overflow and return to zero. This will not affect the internal operation of the kernel. For example, tasks will always block for the specified period even if the tick count overflows while the task is in the Blocked state. Overflows must be considered by host applications if the application makes direct use of the tick count value.

The frequency at which the tick count overflows depends on both the tick frequency and the data type used to hold the count value. If configUSE_16_BIT_TICKS is set to 1, then the tick count will be held in a 16-bit variable. If configUSE_16_BIT_TICKS is set to 0, then the tick count will be held in a 32-bit variable.

Example

```
void vAnISR( void )
```

```
{

    static TickType_t xTimeISRLastExecuted = 0;

    TickType_t xTimeNow, xTimeBetweenInterrupts;

    /* Store the time at which this interrupt was entered. */

    xTimeNow = xTaskGetTickCountFromISR();

    /* Perform some operation. */

    /* How many ticks occurred between this and the previous interrupt? */

    xTimeBetweenInterrupts = xTimeISRLastExecuted - xTimeNow;

    /* If more than 200 ticks occurred between this and the previous interrupt, then do
 something. */

    if( xTimeBetweenInterrupts > 200 )

    {

        /* Take appropriate action here. */

    }

    /* Remember the time at which this interrupt was entered. */

    xTimeISRLastExecuted = xTimeNow;

}
```

vTaskList()

The following listing shows the vTaskList() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

void vTaskList( char *pcWriteBuffer );
```

Summary

Creates a human-readable table in a character buffer that describes the state of each task at the time vTaskList() was called.

Parameters

pcWriteBuffer	The buffer into which the table text is written. This must be large enough to hold the entire table becaus no boundary checking is performed.

Return Values

None.

Notes

vTaskList() is a utility function that is provided for convenience only. It is not considered part of the kernel. vTaskList() obtains its raw data using the xTaskGetSystemState() API function.

vTaskList() will disable interrupts for the duration of its execution. This might not be acceptable for applications that include hard real time functionality.

configUSE_TRACE_FACILITY and configUSE_STATS_FORMATTING_FUNCTIONS must both be set to 1 in FreeRTOSConfig.h for vTaskList() to be available.

By default, vTaskList() makes use of the standard library sprintf() function. This can result in a marked increase in the compiled image size and in stack usage. The FreeRTOS download includes an open source cut-down version of sprintf() in a file called printf-stdarg.c. This can be used in place of the standard library sprintf() to help minimize the code size impact. Note that printf-stdarg.c is licensed separately from FreeRTOS. Its license terms are contained in the file itself.

Example

```
void vAFunction( void )
```

```
{

    /* Define a buffer that is large enough to hold the generated table. In most cases, the
 buffer will be too large to allocate on the stack. In this example, it is declared static.
 */

    static char cBuffer[ BUFFER_SIZE ];

    /* Pass the buffer into vTaskList() to generate the table of information. */

     vTaskList( cBuffer );

    /* The generated information can be saved or viewed here. */

}
```

xTaskNotify()

The following listing shows the xTaskNotify() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

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

Summary

Each task has a 32-bit notification value that is initialized to zero when the task is created. xTaskNotify() is used to send an event directly to and potentially unblock a task and optionally update the receiving task's notification value in one of the following ways:

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

Parameters

xTaskToNotify	The handle of the RTOS task being notified. 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().
ulValue	Used to update the notification value of the task being notified. How ulValue is interpreted depends on the value of the eAction parameter.
eAction	The action to perform when notifying the task. eAction is an enumerated type and can take one of the following values: • eNoAction - The task is notified, but its notification value is not changed. In this case, ulValue is not used. • eSetBits - The task's notification value is bitwise ORed with ulValue. For example, if ulValue is set to 0x01, then bit 0 will be set within the task's notification value. If ulValue is 0x04, then bit 2

	will be set in the task's notification value. Using eSetBits allows task notifications to be used as a faster and lightweight alternative to an event group. • eIncrement - The task's notification value is incremented by one. In this case, ulValue is not used. • eSetValueWithOverwrite - The task's notification value is unconditionally set to the value of ulValue, even if the task already had a notification pending when xTaskNotify() was called. • eSetValueWithoutOverwrite - If the task already has a notification pending, then its notification value is not changed and xTaskNotify() returns pdFAIL. If the task did not already have a notification pending, then its notification value is set to ulValue.

Return Values

If eAction is set to eSetValueWithoutOverwrite and the task's notification value is not updated, then pdFAIL is returned. In all other cases, pdPASS is returned.

Notes

If the task's notification value is being used as a lightweight and faster alternative to a binary or counting semaphore, then use the simpler xTaskNotifyGive() API function instead of xTaskNotify().

RTOS task notification functionality is enabled by default and can be excluded from a build (saving 8 bytes per task) by setting configUSE_TASK_NOTIFICATIONS to 0 in FreeRTOSConfig.h.

Example

The following listing demonstrates the use of xTaskNotify():

```
/* Set bit 8 in the notification value of the task referenced by xTask1Handle. */

xTaskNotify( xTask1Handle, ( 1UL << 8UL ), eSetBits );

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

xTaskNotify( xTask2Handle, 0, eNoAction );

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

xTaskNotify( xTask3Handle, 0x50, eSetValueWithOverwrite );

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

```
if( xTaskNotify( xTask4Handle, 0xfff, eSetValueWithoutOverwrite ) == pdPASS )
{
    /* The task's notification value was updated. */
}
else
{
    /* The task's notification value was not updated. */
}
```

xTaskNotifyAndQuery()

The following listing shows the xTaskNotifyAndQuery() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

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

Summary

xTaskNotifyAndQuery() is similar to xTaskNotify(), but includes an additional parameter in which the subject task's previous notification value is returned.

Each task has a 32-bit notification value that is initialized to zero when the task is created. xTaskNotifyAndQuery() is used to send an event directly to and potentially unblock a task, and optionally update the receiving task's notification value in one of the following ways:

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

Parameters

xTaskToNotify	The handle of the RTOS task being notified. 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().
ulValue	Used to update the notification value of the task being notified. How ulValue is interpreted depends on the value of the eAction parameter.
eAction	The action to perform when notifying the task. eAction is an enumerated type and can take one of the following values: • eNoAction - The task is notified, but its notification value is not changed. In this case, ulValue is not used.

	• eSetBits - The task's notification value is bitwise ORed with ulValue. For example, if ulValue is set to 0x01, then bit 0 will be set within the task's notification value. If ulValue is 0x04, then bit 2 will be set in the task's notification value. Using eSetBits allows task notifications to be used as a faster and lightweight alternative to an event group. • eIncrement - The task's notification value is incremented by one. In this case, ulValue is not used. • eSetValueWithOverwrite - The task's notification value is unconditionally set to the value of ulValue, even if the task already had a notification pending when xTaskNotify() was called. • eSetValueWithoutOverwrite - If the task already has a notification pending, then its notification value is not changed and xTaskNotify() returns pdFAIL. If the task did not already have a notification pending, then its notification value is set to ulValue.
pulPreviousNotifyValue	Used to pass out the subject task's notification value before any bits are modified by the action of xTaskNotifyAndQuery(). pulPreviousNotifyValue is an optional parameter and can be set to NULL if it is not required. If pulPreviousNotifyValue is not used, then consider using xTaskNotify() in place of xTaskNotifyAndQuery().

Return Values

If eAction is set to eSetValueWithoutOverwrite and the task's notification value is not updated, then pdFAIL is returned. In all other cases, pdPASS is returned.

Notes

If the task's notification value is being used as a lightweight and faster alternative to a binary or counting semaphore, then use the simpler xTaskNotifyGive() API function instead of xTaskNotify().

RTOS task notification functionality is enabled by default and can be excluded from a build (saving 8 bytes per task) by setting configUSE_TASK_NOTIFICATIONS to 0 in FreeRTOSConfig.h.

Example

The following listing demonstrates the use of xTaskNotifyAndQuery().

```
uint32_t ulPreviousValue;
```

```
/* Set bit 8 in the notification value of the task referenced by xTask1Handle. The task's
 previous notification value is not needed, so the last pulPreviousNotifyValue parameter is
 set to NULL. */

xTaskNotifyAndQuery( xTask1Handle, ( 1UL << 8UL ), eSetBits, NULL );

/* Send a notification to the task referenced by xTask2Handle, potentially removing the
 task from the Blocked state, but without updating the task's notification value. The
 task's current notification value is saved in ulPreviousValue. */

xTaskNotifyAndQuery( xTask2Handle, 0, eNoAction, &ulPreviousValue );

/* Set the notification value of the task referenced by xTask3Handle to 0x50, even if the
 task had not read its previous notification value. Save the task's previous notification
 value (before it was set to 0x50) in ulPreviousValue. */

xTaskNotifyAndQuery( xTask3Handle, 0x50, eSetValueWithOverwrite, &ulPreviousValue );

/* Set the notification value of the task referenced by xTask4Handle to 0xfff, but only
 if to do so would not overwrite the task's existing notification value before the task
 had obtained it (by a call to xTaskNotifyWait() or ulTaskNotifyTake()). Save the task's
 previous notification value (before it was set to 0xfff) in ulPreviousValue. */

if( xTaskNotifyAndQuery( xTask4Handle, 0xfff, eSetValueWithoutOverwrite, &ulPreviousValue )
 == pdPASS )

{

    /* The task's notification value was updated. */

}

else

{

    /* The task's notification value was not updated. */

}
```

xTaskNotifyAndQueryFromISR()

The following listing shows the xTaskNotifyAndQueryFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

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

Summary

xTaskNotifyAndQuery() is similar to xTaskNotify(), but includes an additional parameter in which the subject task's previous notification value is returned. xTaskNotifyAndQueryFromISR() is a version of xTaskNotifyAndQuery() that can be called from an interrupt service routine (ISR).

Each task has a 32-bit notification value that is initialized to zero when the task is created. xTaskNotifyAndQueryFromISR() is used to send an event directly to and potentially unblock a task, and optionally update the receiving task's notification value in one of the following ways:

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

Parameters

xTaskToNotify	The handle of the RTOS task being notified. 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().
ulValue	Used to update the notification value of the task being notified. How ulValue is interpreted depends on the value of the eAction parameter.
eAction	The action to perform when notifying the task. eAction is an enumerated type and can take one of the following values: • eNoAction - The task is notified, but its notification value is not changed. In this case, ulValue is not used.

	• eSetBits - The task's notification value is bitwise ORed with ulValue. For example, if ulValue is set to 0x01, then bit 0 will be set within the task's notification value. If ulValue is 0x04, then bit 2 will be set in the task's notification value. Using eSetBits allows task notifications to be used as a faster and lightweight alternative to an event group. • eIncrement - The task's notification value is incremented by one. In this case, ulValue is not used. • eSetValueWithOverwrite - The task's notification value is unconditionally set to the value of ulValue, even if the task already had a notification pending when xTaskNotify() was called. • eSetValueWithoutOverwrite - If the task already has a notification pending, then its notification value is not changed and xTaskNotify() returns pdFAIL. If the task did not already have a notification pending, then its notification value is set to ulValue.
pulPreviousNotifyValue	Used to pass out the subject task's notification value before any bits are modified by the action of xTaskNotifyAndQuery(). pulPreviousNotifyValue is an optional parameter and can be set to NULL if it is not required. If pulPreviousNotifyValue is not used, then consider using xTaskNotifyFromISR() in place of xTaskNotifyAndQueryFromISR().
pxHigherPriorityTaskWoken	**pxHigherPriorityTaskWoken* must be initialized to pdFALSE. xTaskNotifyAndQueryFromISR() will set **pxHigherPriorityTaskWoken* to pdTRUE if sending the notification caused the task being notified to leave the Blocked state, and the task being notified has a priority above that of the currently running task. If xTaskNotifyAndQueryFromISR() sets this value to pdTRUE, then a context switch should be requested before the interrupt is exited. See the example. pxHigherPriorityTaskWoken is an optional parameter and can be set to NULL.

Return Values

If eAction is set to eSetValueWithoutOverwrite and the task's notification value is not updated, then pdFAIL is returned. In all other cases, pdPASS is returned.

Notes

RTOS task notification functionality is enabled by default and can be excluded from a build (saving 8 bytes per task) by setting configUSE_TASK_NOTIFICATIONS to 0 in FreeRTOSConfig.h.

Example

The following listing demonstrates the use of xTaskNotifyAndQueryFromISR() to perform various actions from inside an ISR.

```
uint32_t ulPreviousValue;

/* xHigherPriorityTaskWoken must be set to pdFALSE so it can later be detected if it was
 set to pdTRUE by any of the functions called within the interrupt. */

BaseType_t xHigherPriorityTaskWoken = pdFALSE;

/* Set bit 8 in the notification value of the task referenced by xTask1Handle. The task's
 previous notification value is not needed, so the last pulPreviousNotifyValue parameter is
 set to NULL. */

xTaskNotifyAndQueryFromISR( xTask1Handle, ( 1UL << 8UL ), eSetBits, NULL,
 &xHigherPriorityTaskWoken );

/* Send a notification to the task referenced by xTask2Handle, potentially removing the
 task from the Blocked state, but without updating the task's notification value. The
 task's current notification value is saved in ulPreviousValue. */

xTaskNotifyAndQueryFromISR( xTask2Handle, 0, eNoAction, &ulPreviousValue,
 &xHigherPriorityTaskWoken );

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

portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
```

xTaskNotifyFromISR()

The following listing shows the xTaskNotifyFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

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

Summary

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

Each task has a 32-bit notification value that is initialized to zero when the task is created. xTaskNotifyFromISR() is used to send an event directly to and potentially unblock a task, and optionally update the receiving task's notification value in one of the following ways:

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

Parameters

xTaskToNotify	The handle of the RTOS task being notified. 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().
ulValue	Used to update the notification value of the task being notified. How ulValue is interpreted depends on the value of the eAction parameter.
eAction	The action to perform when notifying the task. eAction is an enumerated type and can take one of the following values: • eNoAction - The task is notified, but its notification value is not changed. In this case, ulValue is not used. • eSetBits - The task's notification value is bitwise ORed with ulValue. For example, if ulValue is set to 0x01, then bit 0 will be set within the task's notification value. If ulValue is 0x04, then bit 2

	will be set in the task's notification value. Using eSetBits allows task notifications to be used as a faster and lightweight alternative to an event group. • eIncrement - The task's notification value is incremented by one. In this case, ulValue is not used. • eSetValueWithOverwrite - The task's notification value is unconditionally set to the value of ulValue, even if the task already had a notification pending when xTaskNotify() was called. • eSetValueWithoutOverwrite - If the task already has a notification pending, then its notification value is not changed and xTaskNotify() returns pdFAIL. If the task did not already have a notification pending, then its notification value is set to ulValue.
pxHigherPriorityTaskWoken	**pxHigherPriorityTaskWoken* must be initialized to pdFALSE. xTaskNotifyFromISR() will then set **pxHigherPriorityTaskWoken* to pdTRUE if sending the notification caused the task being notified to leave the Blocked state, and the task being notified has a priority above that of the currently running task. If xTaskNotifyFromISR() sets this value to pdTRUE, then a context switch should be requested before the interrupt is exited. See the example. pxHigherPriorityTaskWoken is an optional parameter and can be set to NULL.

Return Values

If eAction is set to eSetValueWithoutOverwrite and the task's notification value is not updated, then pdFAIL is returned. In all other cases, pdPASS is returned.

Notes

If the task's notification value is being used as a lightweight and faster alternative to a binary or counting semaphore, then use the simpler vTaskNotifyGiveFromISR() API function instead of xTaskNotifyFromISR().

RTOS task notification functionality is enabled by default and can be excluded from a build (saving 8 bytes per task) by setting configUSE_TASK_NOTIFICATIONS to 0 in FreeRTOSConfig.h.

Example

This example demonstrates a single RTOS task being used to process events that originate from two separate interrupt service routines: a transmit interrupt and a receive interrupt. Many peripherals will use

the same handler for both, in which case the peripheral's interrupt status register can simply be bitwise ORed with the receiving task's notification value.

```
/* First bits are defined to represent each interrupt source. */

#define TX_BIT 0x01

#define RX_BIT 0x02

/* The handle of the task that will receive notifications from the interrupts. The handle
 was obtained when the task was created. */

static TaskHandle_t xHandlingTask;

/* The implementation of the transmit interrupt service routine. */

void vTxISR(void)

{

  BaseType_t xHigherPriorityTaskWoken = pdFALSE;

  /* Clear the interrupt source. */

  prvClearInterrupt();

  /* Notify the task that the transmission is complete by setting the  TX_BIT in the
 task's notification value. */

  xTaskNotifyFromISR(xHandlingTask, TX_BIT, eSetBits, & xHigherPriorityTaskWoken);

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

  portYIELD_FROM_ISR(xHigherPriorityTaskWoken);

}

/*-----------------------------------------------------------*/

/* The implementation of the receive interrupt service routine is identical except for the
 bit that gets set in the receiving task's notification value. */

void vRxISR(void)

{

  BaseType_t xHigherPriorityTaskWoken = pdFALSE;

  /* Clear the interrupt source. */

  prvClearInterrupt();

  /* Notify the task that the reception is complete by setting the RX_BIT in the task's
 notification value. */

  xTaskNotifyFromISR(xHandlingTask, RX_BIT, eSetBits, & xHigherPriorityTaskWoken);

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

```
  portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
/*-----------------------------------------------------------*/
/* The implementation of the task that is notified by the interrupt service routines. */
static void prvHandlingTask(void * pvParameter)
{
  const TickType_t xMaxBlockTime = pdMS_TO_TICKS(500);
  BaseType_t xResult;
  for (;;)
  {
    /* Wait to be notified of an interrupt. */
    xResult = xTaskNotifyWait(pdFALSE, /* Don't clear bits on entry. */
      ULONG_MAX, /* Clear all bits on exit. */
      & ulNotifiedValue, /* Stores the notified value. */
      xMaxBlockTime);
    if (xResult == pdPASS)
    {
      /* A notification was received. See which bits were set. */
      if ((ulNotifiedValue & TX_BIT) != 0)
      {
        /* The TX ISR has set a bit. */
        prvProcessTx();
      }
      if ((ulNotifiedValue & RX_BIT) != 0)
      {
        /* The RX ISR has set a bit. */
        prvProcessRx();
      }
    } else
    {
      /* Did not receive a notification within the expected time. */
      prvCheckForErrors();
    }
```

```
    }

}
```

xTaskNotifyGive()

The following listing show the xTaskNotifyGive() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

BaseType_t xTaskNotifyGive( TaskHandle_t xTaskToNotify );
```

Summary

Each task has a 32-bit notification value that is initialized to zero when the task is created. A task notification is an event sent directly to a task that can unblock the receiving task and optionally update the receiving task's notification value.

xTaskNotifyGive() is a macro intended for use when a task notification value is being used as a lightweight and faster alternative to a binary semaphore or a counting semaphore. FreeRTOS semaphores are given using the xSemaphoreGive() API function, and xTaskNotifyGive() is the equivalent that uses the receiving task's notification value instead of a separate semaphore object.

Parameters

xTaskToNotify	The handle of the task being notified and having its notification value incremented. 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().

Return Values

xTaskNotifyGive() is a macro that calls xTaskNotify() with the eAction parameter set to eIncrement. Therefore, pdPASS is always returned.

Notes

When a task notification value is being used as a binary or counting semaphore, then the task being notified should wait for the notification using the simpler ulTaskNotifyTake() API function rather than the xTaskNotifyWait() API function.

RTOS task notification functionality is enabled by default and can be excluded from a build (saving 8 bytes per task) by setting configUSE_TASK_NOTIFICATIONS to 0 in FreeRTOSConfig.h.

Example

```
/* Prototypes of the two tasks created by main(). */

static void prvTask1( void *pvParameters );

static void prvTask2( void *pvParameters );

/* Handles for the tasks create by main(). */

static TaskHandle_t xTask1 = NULL, xTask2 = NULL;

/* Create two tasks that send notifications back and forth to each other, and then start
 the RTOS scheduler. */

void main( void )

{

    xTaskCreate( prvTask1, "Task1", 200, NULL, tskIDLE_PRIORITY, &xTask1 );

    xTaskCreate( prvTask2, "Task2", 200, NULL, tskIDLE_PRIORITY, &xTask2 );

    vTaskStartScheduler();

}

/*-----------------------------------------------------------*/

static void prvTask1( void *pvParameters )

{

    for( ;; )

    {

        /* Send a notification to prvTask2(), bringing it out of the Blocked state. */

        xTaskNotifyGive( xTask2 );

        /* Block to wait for prvTask2() to notify this task. */

        ulTaskNotifyTake( pdTRUE, portMAX_DELAY );

    }

}

/*-----------------------------------------------------------*/

static void prvTask2( void *pvParameters )

{

    for( ;; )

    {
```

```
        /* Block to wait for prvTask1() to notify this task. */
        ulTaskNotifyTake( pdTRUE, portMAX_DELAY );
        /* Send a notification to prvTask1(), bringing it out of the Blocked state. */
        xTaskNotifyGive( xTask1 );
    }
}
```

vTaskNotifyGiveFromISR()

The following listing shows the vTaskNotifyGiveFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

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

Summary

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

Each task has a 32-bit notification value that is initialized to zero when the task is created. A task notification is an event sent directly to a task that can unblock the receiving task and optionally update the receiving task's notification value.

vTaskNotifyGiveFromISR() is intended for use when a task notification value is being used as a lightweight and faster alternative to a binary semaphore or a counting semaphore. FreeRTOS semaphores are given using the xSemaphoreGiveFromISR() API function, and vTaskNotifyGiveFromISR() is the equivalent that uses the receiving task's notification value instead of a separate semaphore object.

Parameters

xTaskToNotify	The handle of the RTOS task being notified and having its notification value incremented. 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().
pxHigherPriorityTaskWoken	`*pxHigherPriorityTaskWoken` must be initialized to pdFALSE. vTaskNotifyGiveFromISR() will then set `*pxHigherPriorityTaskWoken` to pdTRUE if sending the notification caused the task being notified to leave the Blocked state, and the unblocked task has a priority above that of the currently running task. If vTaskNotifyGiveFromISR() sets this value to pdTRUE, then a context switch should be requested before the interrupt is exited. See the example. pxHigherPriorityTaskWoken is an optional parameter and can be set to NULL.

Notes

When a task notification value is being used as a binary or counting semaphore, then the task being notified should wait for the notification using the ulTaskNotifyTake() API function rather than the xTaskNotifyWait() API function.

RTOS task notification functionality is enabled by default and can be excluded from a build (saving 8 bytes per task) by setting configUSE_TASK_NOTIFICATIONS to 0 in FreeRTOSConfig.h.

Example

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

```
static TaskHandle_t xTaskToNotify = NULL;


/* The peripheral driver's transmit function. */


void StartTransmission( uint8_t *pcData, size_t xDataLength )
{
    /* At this point xTaskToNotify should be NULL because no transmission is in progress. A
 mutex can be used to guard access to the peripheral if necessary. */

    configASSERT( xTaskToNotify == NULL );

    /* Store the handle of the calling task. */

    xTaskToNotify = xTaskGetCurrentTaskHandle();

    /* Start the transmission. An interrupt is generated when the transmission is complete.
 */

    vStartTransmit( pcData, xDataLength );

}

/*-----------------------------------------------------------*/

/* The transmit end interrupt. */

void vTransmitEndISR( void )

{

    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* At this point xTaskToNotify should not be NULL because a transmission was in
 progress. */

    configASSERT( xTaskToNotify != NULL );

    /* Notify the task that the transmission is complete. */

    vTaskNotifyGiveFromISR( xTaskToNotify, &xHigherPriorityTaskWoken );

    /* There are no transmissions in progress, so no tasks to notify. */
```

```
    xTaskToNotify = NULL;


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

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}

/*-----------------------------------------------------------*/

/* The task that initiates the transmission, and then enters the Blocked state (so not
 consuming any CPU time) to wait for it to complete. */

void vAFunctionCalledFromATask( uint8_t ucDataToTransmit, size_t xDataLength )

{

    uint32_t ulNotificationValue;

    const TickType_t xMaxBlockTime = pdMS_TO_TICKS( 200 );

    /* Start the transmission by calling the function shown above. */

    StartTransmission( ucDataToTransmit, xDataLength );

    /* Wait for the transmission to complete. */

    ulNotificationValue = ulTaskNotifyTake( pdFALSE, xMaxBlockTime );

    if( ulNotificationValue == 1 )

    {

        /* The transmission ended as expected. */

    }

    else

    {

      /* The call to ulTaskNotifyTake() timed out. */

    }

}
```

xTaskNotifyStateClear()

The following shows the xTaskNotifyStateClear() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

BaseType_t xTaskNotifyStateClear( TaskHandle_t xTask )
```

Summary

Each task has a 32-bit notification value that is initialized to zero when the task is created. A task notification is an event sent directly to a task that can unblock the receiving task and optionally update the receiving task's notification value.

If a task is in the Blocked state to wait for a notification when the notification arrives, then the task immediately exits the Blocked state and the notification does not remain pending. If a task is not waiting for a notification when a notification arrives, then the notification will remain pending until either:

- The receiving task reads its notification value.
- The receiving task is the subject task in a call to xTaskNotifyStateClear().

  xTaskNotifyStateClear() will clear a pending notification, but does not change the notification value.

Parameters

xTask	The handle of the task that will have a pending notification cleared. Setting xTask to NULL will clear a pending notification in the task that called xTaskNotifyStateClear().

Return Values

If the task referenced by xTask had a notification pending, then pdPASS is returned. If the task referenced by xTask did not have a notification pending, then pdFAIL is returned.

Example

The following shows the use of xTaskNotifyStateClear().

```
/* An example UART transmit function. The function starts a UART
transmission, and then waits to be notified that the transmission is
complete. The transmission complete notification is sent from the UART
```

```
interrupt. The calling task's notification state is cleared before the
transmission is started to ensure it is not coincidentally already
pending before the task attempts to block on its notification state. */

void vSerialPutString( const signed char * const pcStringToSend, uint16_t usStringLength )

{

    const TickType_t xMaxBlockTime = pdMS_TO_TICKS( 5000 );

    /* xSendingTask holds the handle of the task waiting for the transmission to complete.
 If xSendingTask is NULL, then a transmission is not in progress. Don't start to send a new
 string unless transmission of the previous string is complete. */

    if( ( xSendingTask == NULL ) && ( usStringLength > 0 ) )

    {

            /* Ensure the calling task's notification state is not already pending. */

            xTaskNotifyStateClear( NULL );

            /* Store the handle of the transmitting task. This is used to unblock the task
 when the transmission has completed. */

            xSendingTask = xTaskGetCurrentTaskHandle();

            /* Start sending the string. The transmission is then controlled by an
 interrupt. */

            UARTSendString( pcStringToSend, usStringLength );

            /* Wait in the Blocked state (so not using any CPU time) until the UART
 ISR sends a notification to xSendingTask to notify (and unblock) the task when the
 transmission is complete. */

            ulTaskNotifyTake( pdTRUE, xMaxBlockTime );

    }

}
```

ulTaskNotifyTake()

The following shows the ulTaskNotifyTake() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

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

Summary

Each task has a 32-bit notification value that is initialized to zero when the task is created. A task notification is an event sent directly to a task that can unblock the receiving task and optionally update the receiving task's notification value.

ulTaskNotifyTake() is intended for use when a task notification is used as a faster and lightweight alternative to a binary semaphore or a counting semaphore. FreeRTOS semaphores are taken using the xSemaphoreTake() API function. ulTaskNotifyTake() is the equivalent and uses a task notification value instead of a separate semaphore object.

Where xTaskNotifyWait() will return when a notification is pending, ulTaskNotifyTake() will return when the task's notification value is not zero, decrementing the task's notification value before it returns.

A task can use ulTaskNotifyTake() to optionally block to wait for a task's notification value to be non-zero. The task does not consume any CPU time while it is in the Blocked state.

ulTaskNotifyTake() can either clear the task's notification value to zero on exit, in which case the notification value acts like a binary semaphore, or decrement the task's notification value on exit, in which case the notification value acts more like a counting semaphore.

Parameters

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

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

Return Values

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

Notes

When a task is using its notification value as a binary or counting semaphore, other tasks and interrupts should send notifications to it using either the xTaskNotifyGive() macro or the xTaskNotify() function with the function's eAction parameter set to eIncrement. (The two are equivalent.)

RTOS task notification functionality is enabled by default and can be excluded from a build (saving 8 bytes per task) by setting configUSE_TASK_NOTIFICATIONS to 0 in FreeRTOSConfig.h.

Example

Example use of ulTaskNotifyTake()

```
/* An interrupt handler that unblocks a high priority task in which the
event that generated the interrupt is processed. If the priority of the
task is high enough, then the interrupt will return directly to the task
(so it will interrupt one task and then return to a different task), so the
processing will occur contiguously in time, just as if all the
processing had been done in the interrupt handler itself. */

void vANInterruptHandler( void )

{

    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Clear the interrupt. */

    prvClearInterruptSource();

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

    vTaskNotifyGiveFromISR( xHandlingTask, &xHigherPriorityTaskWoken );

    /* Force a context switch if xHigherPriorityTaskWoken is now set to pdTRUE.
```

```
    The macro used to do this depends on the port and may be called
 portEND_SWITCHING_ISR(). */

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}

/*-----------------------------------------------------------*/

/* Task that blocks waiting to be notified that the peripheral needs servicing. */

void vHandlingTask( void *pvParameters )

{

    BaseType_t xEvent;

    for( ;; )

    {

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

        ulTaskNotifyTake( pdTRUE, /* Clear the notification value on exit. */
                          portMAX_DELAY );/* Block indefinitely. */

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

        do

        {

            xEvent = xQueryPeripheral();

            if( xEvent != NO_MORE_EVENTS )

            {

                    vProcessPeripheralEvent( xEvent );

            }

        } while( xEvent != NO_MORE_EVENTS );

    }

}
```

xTaskNotifyWait()

The following shows the xTaskNotifyWait() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

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

Summary

Each task has a 32-bit notification value that is initialized to zero when the task is created. A task notification is an event sent directly to a task that can unblock the receiving task and optionally update the receiving task's notification value in a number of ways. For example, a notification can overwrite the receiving task's notification value or just set one or more bits in the receiving task's notification value. See the xTaskNotify() API documentation for examples.

xTaskNotifyWait() waits, with an optional timeout, for the calling task to receive a notification.

If the receiving task was already Blocked waiting for a notification, when one arrives, the receiving task will be removed from the Blocked state and the notification cleared.

Parameters

ulBitsToClearOnEntry	Any bits set in ulBitsToClearOnEntry will be cleared in the calling task's notification value on entry to the xTaskNotifyWait() function (before the task waits for a new notification) provided a notification is not already pending when xTaskNotifyWait() is called. For example, if ulBitsToClearOnEntry is 0x01, then bit 0 of the task's notification value will be cleared on entry to the function. Setting ulBitsToClearOnEntry to 0xffffffff (ULONG_MAX) will clear all the bits in the task's notification value, effectively clearing the value to 0.
ulBitsToClearOnExit	Any bits set in ulBitsToClearOnExit will be cleared in the calling task's notification value before xTaskNotifyWait() function exits if a notification was received. The bits are cleared after the task's notification value has been saved in `*pulNotificationValue`.

	For example, if ulBitsToClearOnExit is 0x03, then bit 0 and bit 1 of the task's notification value will be cleared before the function exits. Setting ulBitsToClearOnExit to 0xffffffff (ULONG_MAX) will clear all the bits in the task's notification value, effectively clearing the value to 0.
pulNotificationValue	Used to pass out the task's notification value. The value copied to `*pulNotificationValue` is the task's notification value as it was before any bits were cleared due to the ulBitsToClearOnExit setting. pulNotificationValue is an optional parameter and can be set to NULL if it is not required.
xTicksToWait	The maximum time to wait in the Blocked state for a notification to be received if a notification is not already pending when xTaskNotifyWait() is called. The task does not consume any CPU time when it is in the Blocked state. The time is specified in RTOS tick periods. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds into ticks.

Return Values

pdTRUE is returned if a notification was received or if a notification was already pending when xTaskNotifyWait() was called.

pdFALSE is returned if the call to xTaskNotifyWait() timed out before a notification was received.

Notes

If you are using task notifications to implement binary or counting semaphore type behavior, then use the simpler ulTaskNotifyTake() API function instead of xTaskNotifyWait().

RTOS task notification functionality is enabled by default and can be excluded from a build (saving 8 bytes per task) by setting configUSE_TASK_NOTIFICATIONS to 0 in FreeRTOSConfig.h.

Example

The following shows the use of xTaskNotifyWait().

```
/* This task shows bits within the RTOS task notification value being
used to pass different events to the task in the same way that flags in
an event group might be used for the same purpose. */
```

```
void vAnEventProcessingTask( void *pvParameters )

{

    uint32_t ulNotifiedValue;

    for( ;; )

    {

          /* Block indefinitely (without a timeout, so no need to check the function's
 return value) to wait for a notification.
           Bits in this RTOS task's notification value are set by the notifying tasks and
 interrupts to indicate which events have occurred. */

          xTaskNotifyWait( 0x00, /* Don't clear any notification bits on entry.*/
                           ULONG_MAX, /* Reset the notification value to 0 on exit. */
                           &ulNotifiedValue, /* Notified value pass out in ulNotifiedValue.
 */
                           portMAX_DELAY ); /* Block indefinitely. */

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

          if( ( ulNotifiedValue & 0x01 ) != 0 )
          {

              /* Bit 0 was set. Process the event represented by bit 0. */

              prvProcessBit0Event();

          }

          if( ( ulNotifiedValue & 0x02 ) != 0 )

          {

                 /* Bit 1 was set. Process the event represented by bit 1. */

                 prvProcessBit1Event();

          }

          if( ( ulNotifiedValue & 0x04 ) != 0 )

          {

               /* Bit 2 was set. Process the event represented by bit 2. */

               prvProcessBit2Event();

          }

          /* Etc. */

      }

 }
```

uxTaskPriorityGet()

The following listing shows the uxTaskPriorityGet() function prototype.

Summary

Queries the priority assigned to a task at the time uxTaskPriorityGet() is called.

Parameters

pxTask	The handle of the task being queried (the subject 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(). A task can query its own priority by passing NULL in place of a valid task handle.

Return Values

The value returned is the priority of the task being queried at the time uxTaskPriorityGet() is called.

Example

Example use of uxTaskPriorityGet()

```
void vAFunction( void )

{

    TaskHandle_t xHandle;

    UBaseType_t uxCreatedPriority, uxOurPriority;

    /* Create a task, storing the handle of the created task in xHandle.*/

    if( xTaskCreate( vTaskCode, "Demo task", STACK_SIZE, NULL, PRIORITY, &xHandle ) !=
 pdPASS )

    {
```

```
        /* The task was not created successfully. */

    }

    else

    {

        /* Use the handle to query the priority of the created task. */

        uxCreatedPriority = uxTaskPriorityGet( xHandle );

        /* Query the priority of the calling task by using NULL in place of a valid task
 handle. */

        uxOurPriority = uxTaskPriorityGet( NULL );

        /* Is the priority of this task higher than the priority of the task just created?
 */

       if( uxOurPriority > uxCreatedPriority )

       {

          /* Yes. */

       }

    }

}
```

vTaskPrioritySet()

The following shows the vTaskPrioritySet() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

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

Summary

Changes the priority of a task.

Parameters

pxTask	The handle of the task being modified (the subject 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(). A task can change its own priority by passing NULL in place of a valid task handle.
uxNewPriority	The priority to which the subject task will be set. Priorities can be assigned from 0, which is the lowest priority, to (configMAX_PRIORITIES - 1), which is the highest priority. configMAX_PRIORITIES is defined in FreeRTOSConfig.h. Passing a value above (configMAX_PRIORITIES - 1) will result in the priority assigned to the task being capped to the maximum legitimate value.

Return Values

None.

Notes

vTaskPrioritySet() must only be called from an executing task, and therefore must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

It is possible to have a set of tasks that are all blocked waiting for the same queue or semaphore event. These tasks will be ordered according to their priority. For example, the first event will unblock the highest priority task that was waiting for the event. The second event will unblock the second highest priority task that was originally waiting for the event, and so on. Using vTaskPrioritySet() to change the priority of such a blocked task will not cause the order in which the blocked tasks are assessed to be reevaluated.

Example

Example use of vTaskPrioritySet()

```
void vAFunction( void )

{

    TaskHandle_t xHandle;

    /* Create a task, storing the handle of the created task in xHandle.*/

    if( xTaskCreate( vTaskCode,"Demo task",STACK_SIZE,NULL,PRIORITY,&xHandle) != pdPASS )

    {

          /* The task was not created successfully. */

    }

    else

    {

         /* Use the handle to raise the priority of the created task. */

         vTaskPrioritySet( xHandle, PRIORITY + 1 );

         /* Use NULL in place of a valid task handle to set the priority of the calling
 task to 1. */

         vTaskPrioritySet( NULL, 1 );

    }

}
```

vTaskResume()

The following shows the vTaskResume() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

void vTaskResume( TaskHandle_t pxTaskToResume );
```

Summary

Transition a task from the Suspended state to the Ready state. The task must have previously been placed into the Suspended state using a call to vTaskSuspend().

Parameters

pxTaskToResume	The handle of the task being resumed (transitioned out of the Suspended state). This is the subject 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().

Return Values

None.

Notes

A task can be blocked to wait for a queue event, specifying a timeout period. It is legitimate to move such a Blocked task into the Suspended state using a call to vTaskSuspend(), and then out of the Suspended state and into the Ready state using a call to vTaskResume(). Following this scenario, the next time the task enters the Running state, it will check whether or not its timeout period has (in the meantime) expired. If the timeout period has not expired, the task will once again enter the Blocked state to wait for the queue event for the remainder of the originally specified timeout period.

A task can also be blocked to wait for a temporal event using the vTaskDelay() or vTaskDelayUntil() API functions. It is legitimate to move such a Blocked task into the Suspended state using a call to vTaskSuspend(), then out of the Suspended state and into the Ready state using a call to vTaskResume().

Following this scenario, the next time the task enters the Running state it will exit the vTaskDelay() or vTaskDelayUntil() function as if the specified delay period had expired, even if this is not actually the case.

vTaskResume() must only be called from an executing task and therefore must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

Example

The following shows the use of vTaskResume().

```
void vAFunction( void )

{

     TaskHandle_t xHandle;

     /* Create a task, storing the handle to the created task in xHandle.*/

     if( xTaskCreate( vTaskCode,"Demo task", STACK_SIZE, NULL, PRIORITY, &xHandle ) !=
 pdPASS )

     {

        /* The task was not created successfully. */

     }

     else

     {

             /* Use the handle to suspend the created task. */

             vTaskSuspend( xHandle );

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

             /* Resume the suspended task again. */

             vTaskResume( xHandle );

             /* The created task is again available to the scheduler and can enter The
 Running state. */

       }

}
```

xTaskResumeAll()

The following shows the xTaskResumeAll() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

BaseType_t xTaskResumeAll( void );
```

Summary

Resumes scheduler activity, following a previous call to vTaskSuspendAll(), by transitioning the scheduler into the Active state from the Suspended state.

Parameters

None.

Return Values

pdTRUE	The scheduler was transitioned into the Active state. The transition caused a pending context switch to occur.
pdFALSE	Either the scheduler was transitioned into the Active state and the transition did not cause a context switch to occur, or the scheduler was left in the Suspended state due to nested calls to vTaskSuspendAll().

Notes

The scheduler can be suspended by calling vTaskSuspendAll(). When the scheduler is suspended, interrupts remain enabled, but a context switch will not occur. If a context switch is requested while the scheduler is suspended, then the request will be held pending until such time that the scheduler is resumed (unsuspended).

Calls to vTaskSuspendAll() can be nested. The same number of calls must be made to xTaskResumeAll() as have previously been made to vTaskSuspendAll() before the scheduler will leave the Suspended state and re-enter the Active state.

xTaskResumeAll() must only be called from an executing task and therefore must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

Other FreeRTOS API functions should not be called while the scheduler is suspended.

Example

The following shows the use of xTaskResumeAll().

```
/* A function that suspends then resumes the scheduler. */

void vDemoFunction( void )

{

    /* This function suspends the scheduler. When it is called from vTask1, the scheduler
 is already suspended, so this call creates a nesting depth of 2. */

    vTaskSuspendAll();

    /* Perform an action here. */

    /* As calls to vTaskSuspendAll() are now nested, resuming the scheduler here does not
 cause the scheduler to re-enter the active state. */

    xTaskResumeAll();

}

void vTask1( void * pvParameters )

{

    for( ;; )

    {

         /* Perform some actions here. */

         /* At some point the task wants to perform an operation during which it does not
 want to get swapped out, or it wants to access data that is
         also accessed from another task (but not from an interrupt). It cannot use
 taskENTER_CRITICAL()/taskEXIT_CRITICAL() as the length of the
         operation might cause interrupts to be missed. */

         /* Prevent the scheduler from performing a context switch. */

         vTaskSuspendAll();

         /* Perform the operation here. There is no need to use critical sections because
 the task has all the processing time other than that used by
         interrupt service routines.*/

         /* Calls to vTaskSuspendAll() can be nested, so it is safe to call a (non-API)
 function that also calls vTaskSuspendAll(). API functions should
         not be called while the scheduler is suspended. */

         vDemoFunction();

         /* The operation is complete. Set the scheduler back into the Active state. */

         if( xTaskResumeAll() == pdTRUE )

         {
```

```
                /* A context switch occurred within xTaskResumeAll(). */

        }

        else

        {

                /* A context switch did not occur within xTaskResumeAll(). */

        }

    }

}
```

xTaskResumeFromISR()

The following shows the xTaskResumeFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

BaseType_t xTaskResumeFromISR( TaskHandle_t pxTaskToResume );
```

Summary

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

Parameters

pxTaskToResume	The handle of the task being resumed (transitioned out of the Suspended state). This is the subject 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().

Return Values

pdTRUE	Returned if the task being resumed (unblocked) has a priority equal to or higher than the currently executing task (the task that was interrupted), meaning a context switch should be performed before exiting the interrupt.
pdFALSE	Returned if the task being resumed has a priority lower than the currently executing task (the task that was interrupted), meaning it is not necessary to perform a context switch before exiting the interrupt.

Notes

A task can be suspended by calling vTaskSuspend(). While in the Suspended state, the task will not be selected to enter the Running state. vTaskResume() and xTaskResumeFromISR() can be used to

resume (unsuspend) a suspended task. xTaskResumeFromISR() can be called from an interrupt, but vTaskResume() cannot.

Calls to vTaskSuspend() do not maintain a nesting count. A task that has been suspended by one or more calls to vTaskSuspend() will always be unsuspended by a single call to vTaskResume() or xTaskResumeFromISR().

xTaskResumeFromISR() must not be used to synchronize a task with an interrupt. Doing so will result in interrupt events being missed if the interrupt events occur faster than the execution of its associated task level handling functions. Task and interrupt synchronization can be achieved safely using a binary or counting semaphore because the semaphore will latch events.

Example

Example use of xTaskResumeFromISR()

```
TaskHandle_t xHandle;

void vAFunction( void )

{

      /* Create a task, storing the handle of the created task in xHandle.*/

      xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );

      /* ... Rest of code. */

 }

 void vTaskCode( void *pvParameters )

 {

       /* The task being suspended and resumed. */

       for( ;; )

       {

          /* ... Perform some function here. */

          /* The task suspends itself by using NULL as the parameter to vTaskSuspend() in
 place of a valid task handle. */

          vTaskSuspend( NULL );

          /* The task is now suspended, so will not reach here until the ISR resumes
 (unsuspends) it. */

       }

  }

  void vAnExampleISR( void )

  {

      BaseType_t xYieldRequired;

      /* Resume the suspended task. */
```

```
     xYieldRequired = xTaskResumeFromISR( xHandle );

     if( xYieldRequired == pdTRUE )

     {

          /* A context switch should now be performed so the ISR returns directly to
          the resumed task. This is because the resumed task had a priority that
          was equal to or higher than the task that is currently in the Running state.
          NOTE: The syntax required to perform a context switch from an ISR varies
          from port to port and from compiler to compiler. Check the
          documentation and examples for the port being used to find the syntax required
by your
          application. It is likely that this if() statement can be replaced by a
          single call to portYIELD_FROM_ISR() [or portEND_SWITCHING_ISR()]
          using xYieldRequired as the macro parameter:
portYIELD_FROM_ISR( xYieldRequired );*/

          portYIELD_FROM_ISR();

      }

}
```

vTaskSetApplicationTaskTag()

The following shows the vTaskSetApplicationTaskTag() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

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

Summary

This function is intended for advanced users only.

The vTaskSetApplicationTaskTag() API function can be used to assign a 'tag' value to a task. The meaning and use of the tag value is defined by the application writer. The kernel will not normally access the tag value.

Parameters

xTask	The handle of the task to which a tag value is being assigned. This is the subject task. A task can assign a tag value to itself by either using its own task handle or by using NULL in place of a valid task handle.
pxTagValue	The value being assigned as the tag value of the subject task. This is of type TaskHookFunction_t to permit a function pointer to be assigned to the tag, even though indirectly by casting, tag values can be of any type.

Return Values

None.

Notes

The tag value can be used to hold a function pointer. When this is done, the function assigned to the tag value can be called using the xTaskCallApplicationTaskHook() API function. This technique is in effect assigning a callback function to the task. It is common for such a callback to be used in combination with the traceTASK_SWITCHED_IN() macro to implement an execution trace feature.

configUSE_APPLICATION_TASK_TAG must be set to 1 in FreeRTOSConfig.h for vTaskSetApplicationTaskTag() to be available.

Example

The following shows the use of vTaskSetApplicationTaskTag().

```
/* In this example, an integer is set as the task tag value. */

void vATask( void *pvParameters )

{

    /* Assign a tag value of 1 to the currently executing task. The (void *) cast is used
 to prevent compiler warnings. */

    vTaskSetApplicationTaskTag( NULL, ( void * ) 1 );

    for( ;; )

    {

         /* Rest of task code goes here. */

    }

 }

 /* In this example, a callback function is assigned as the task tag. First, define the
 callback function. It must have type TaskHookFunction_t, as shown in this example. */

 static BaseType_t prvExampleTaskHook( void * pvParameter )

 {

    /* Perform some action. This can be anything from logging a value, updating the task
 state, outputting a value, and so on. */

    return 0;

 }

 /* Now define the task that sets prvExampleTaskHook() as its hook/tag value. This is in
 effect registering the task callback function. */

 void vAnotherTask( void *pvParameters )

 {

     /* Register a callback function for the currently running (calling) task. */

     vTaskSetApplicationTaskTag( NULL, prvExampleTaskHook );

     for( ;; )

     {

         /* Rest of task code goes here. */

     }

 }
```

```
/* [As an example use of the hook (callback)] Define the traceTASK_SWITCHED_OUT() macro to
call the hook function. The kernel will then automatically call
the task hook each time the task is switched out. This technique can be used to generate
an execution trace. pxCurrentTCB references the currently executing
task. */

#define traceTASK_SWITCHED_OUT() xTaskCallApplicationTaskHook(pxCurrentTCB, 0 )
```

vTaskSetThreadLocalStoragePointer()

The following shows the vTaskSetThreadLocalStoragePointer() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

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

Summary

Thread local storage (TLS) allows the application writer to store values inside a task's control block, making the value specific to (local to) the task itself and allowing each task to have its own unique value.

Each task has its own array of pointers that can be used as thread local storage. The number of indexes in the array is set by the configNUM_THREAD_LOCAL_STORAGE_POINTERS compile time configuration constant in FreeRTOSConfig.h.

vTaskSetThreadLocalStoragePointer() sets the value of an index in the array, effectively storing a thread local value.

Parameters

xTaskToSet	The handle of the task to which the thread local data is being written. A task can write to its own thread local data by using NULL as the parameter value..
xIndex	The index into the thread local storage array to which data is being written.
pvValue	The value to write into the index specified by xIndex.

Return Values

None.

Example

The following shows the use of vTaskSetThreadLocalStoragePointer().

```
uint32_t ulVariable;

/* Write the 32-bit 0x12345678 value directly into index 1 of the thread local
storage array. Passing NULL as the task handle has the effect of writing
to the calling task's thread local storage array. */

vTaskSetThreadLocalStoragePointer( NULL, /* Task handle. */ 1, /* Index into the array. */
 ( void * ) 0x12345678 );

/* Store the value of the 32-bit variable ulVariable to index 0 of the calling task's
 thread local storage array. */

ulVariable = ERROR_CODE;

vTaskSetThreadLocalStoragePointer( NULL, /* Task handle. */ 0, /* Index into the array. */
 ( void * ) &ulVariable );
```

vTaskSetTimeOutState()

The following shows the vTaskSetTimeOutState() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

void vTaskSetTimeOutState( TimeOut_t * const pxTimeOut );
```

Summary

This function is intended for advanced users only.

A task can enter the Blocked state to wait for an event. Typically, the task will not wait in the Blocked state indefinitely. Instead, a timeout period will be specified. The task will be removed from the Blocked state if the timeout period expires before the event the task is waiting for occurs.

If a task enters and exits the Blocked state more than once while it is waiting for the event to occur, then the timeout used each time the task enters the Blocked state must be adjusted to ensure the total of all the time spent in the Blocked state does not exceed the originally specified timeout period. xTaskCheckForTimeOut() performs the adjustment, taking into account occasional occurrences such as tick count overflows, which would otherwise make a manual adjustment prone to error.

vTaskSetTimeOutState() is used with xTaskCheckForTimeOut(). vTaskSetTimeOutState() is called to set the initial condition, after which xTaskCheckForTimeOut() can be called to check for a timeout condition and adjust the remaining block time if a timeout has not occurred.

Parameters

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

Example

The following shows the use of vTaskSetTimeOutState() and xTaskCheckForTimeOut().

```
/* 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 recalculate 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 the number of available bytes, 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;

}
```

vTaskStartScheduler()

The following shows the vTaskStartScheduler() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

void vTaskStartScheduler( void );
```

Summary

Starts the FreeRTOS scheduler running.

Typically, before the scheduler has been started, main() (or a function called by main()) will be executing. After the scheduler has been started, only tasks and interrupts will execute.

Starting the scheduler causes the highest priority task that was created while the scheduler was in the Initialization state to enter the Running state.

Parameters

None.

Return Values

The Idle task is created automatically when the scheduler is started. vTaskStartScheduler() will only return if there is not enough FreeRTOS heap memory available for the Idle task to be created.

Notes

Ports that execute on ARM7 and ARM9 microcontrollers require the processor to be in Supervisor mode before vTaskStartScheduler() is called.

Example

The following shows the use of vTaskStartScheduler().

```
TaskHandle_t xHandle;

/* Define a task function. */

void vATask( void )
```

```
{
    for( ;; )
    {
        /* Task code goes here. */
    }
}
void main( void )
{
    /* Create at least one task. In this case, the task function defined
    above is created. Calling vTaskStartScheduler() before any tasks have been
    created will cause the idle task to enter the Running state. */
    xTaskCreate( vTaskCode, "task name", STACK_SIZE, NULL, TASK_PRIORITY, NULL );
    /* Start the scheduler. */
    vTaskStartScheduler();
    /* This code will only be reached if the idle task could not be created inside
    vTaskStartScheduler(). An infinite loop is used to assist debugging by
    ensuring this scenario does not result in main() exiting. */
    for( ;; );
}
```

vTaskStepTick()

```
#include "FreeRTOS.h"

#include "task.h"

void vTaskStepTick( TickType_t xTicksToJump );
```

Summary

If the RTOS is configured to use tickless idle functionality, then the tick interrupt will be stopped and the microcontroller placed into a low power state whenever the Idle task is the only task able to execute. Upon exiting the low power state, the tick count value must be corrected to account for the time that passed while it was stopped.

If a FreeRTOS port includes a default portSUPPRESS_TICKS_AND_SLEEP() implementation, then vTaskStepTick() is used internally to ensure the correct tick count value is maintained. vTaskStepTick() is a public API function to allow the default portSUPPRESS_TICKS_AND_SLEEP() implementation to be overridden and for a portSUPPRESS_TICKS_AND_SLEEP() to be provided if the port being used does not provide a default.

Parameters

xTicksToJump	The number of RTOS tick periods that passed between the tick interrupt being stopped and restarted (how long the tick interrupt was suppressed for). For correct operation, the parameter must be less than or equal to the portSUPPRESS_TICKS_AND_SLEEP() parameter.

Return Values

None.

Notes

configUSE_TICKLESS_IDLE must be set to 1 in FreeRTOSConfig.h for vTaskStepTick() to be available.

Example

Example use of vTaskStepTick()

```
/* This is an example of how portSUPPRESS_TICKS_AND_SLEEP() might be
implemented by an application writer. This basic implementation will
introduce inaccuracies in the tracking of the time maintained by the
kernel in relation to calendar time. Official FreeRTOS implementations
account for these inaccuracies as much as possible.
Only vTaskStepTick() is part of the FreeRTOS API. The other function
calls are for demonstration only. */

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

#define portSUPPRESS_TICKS_AND_SLEEP( xIdleTime ) vApplicationSleep(xIdleTime )

/* Define the function that is called by portSUPPRESS_TICKS_AND_SLEEP(). */

void vApplicationSleep( TickType_t xExpectedIdleTime )

{

    unsigned long ulLowPowerTimeBeforeSleep, ulLowPowerTimeAfterSleep;

    /* Read the current time from a time source that will remain operational when
    the microcontroller is in a low power state. */

    ulLowPowerTimeBeforeSleep = ulGetExternalTime();

    /* Stop the timer that is generating the tick interrupt. */

    prvStopTickInterruptTimer();

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

    vSetWakeTimeInterrupt( xExpectedIdleTime );

    /* Enter the low power state. */

    prvSleep();

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

    ulLowPowerTimeAfterSleep = ulGetExternalTime();

    /* Correct the kernel's tick count to account for the time the microcontroller spent in
 its low power state. */

    vTaskStepTick( ulLowPowerTimeAfterSleep - ulLowPowerTimeBeforeSleep );

    /* Restart the timer that is generating the tick interrupt. */

    prvStartTickInterruptTimer();

}
```

vTaskSuspend()

The following shows the vTaskSuspend() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

void vTaskSuspend( TaskHandle_t pxTaskToSuspend );
```

Summary

Places a task into the Suspended state. A task in the Suspended state will never be selected to enter the Running state.

The only way to remove a task from the Suspended state is to make it the subject of a call to vTaskResume().

Parameters

pxTaskToSuspend	The handle of the task being suspended. 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(). A task can suspend itself by passing NULL in place of a valid task handle.

Return Values

None.

Notes

If you are using FreeRTOS version 6.1.0 or later, then vTaskSuspend() can be called to place a task into the Suspended state before the scheduler has been started (before vTaskStartScheduler() has been called). This will result in the task (effectively) starting in the Suspended state.

Example

Example use of vTaskSuspend()

```
void vAFunction( void )

{

    TaskHandle_t xHandle;

    /* Create a task, storing the handle of the created task in xHandle.*/

    if( xTaskCreate( vTaskCode, "Demo task", STACK_SIZE, NULL, PRIORITY, &xHandle ) !=
 pdPASS )

    {

        /* The task was not created successfully. */

    }

    else

    {

        /* Use the handle of the created task to place the task in the Suspended state.
 From FreeRTOS version 6.1.0, this can be done before the
        Scheduler has been started. */

        vTaskSuspend( xHandle );

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

        /* Use a NULL parameter to suspend the calling task. */

        vTaskSuspend( NULL );

        /* This task can only execute past the call to vTaskSuspend( NULL ) if another task
 has resumed (unsuspended) it using a call to vTaskResume(). */

    }

}
```

vTaskSuspendAll()

The following shows the vTaskSuspendAll() function prototype.

```
#include "FreeRTOS.h"

#include "task.h"

void vTaskSuspendAll( void );
```

Summary

Suspends the scheduler. Suspending the scheduler prevents a context switch from occurring, but leaves interrupts enabled. If an interrupt requests a context switch while the scheduler is suspended, then the request is held pending and is performed only when the scheduler is resumed (unsuspended).

Parameters

None.

Return Values

None.

Notes

Calls to xTaskResumeAll() transition the scheduler out of the Suspended state following a previous call to vTaskSuspendAll().

Calls to vTaskSuspendAll() can be nested. The same number of calls must be made to xTaskResumeAll() as have previously been made to vTaskSuspendAll() before the scheduler will leave the Suspended state and re-enter the Active state.

xTaskResumeAll() must only be called from an executing task and therefore must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

Other FreeRTOS API functions must not be called while the scheduler is suspended.

Example

Example use of vTaskSuspendAll()

```
/* A function that suspends then resumes the scheduler. */
```

```
void vDemoFunction( void )

{

    /* This function suspends the scheduler. When it is called from vTask1,
    the scheduler is already suspended, so this call creates a nesting depth of 2. */

    vTaskSuspendAll();

    /* Perform an action here. */

    /* As calls to vTaskSuspendAll() are nested, resuming the scheduler
    here will not cause the scheduler to re-enter the active state. */

    xTaskResumeAll();

}

void vTask1( void * pvParameters )

{

    for( ;; )

    {

        /* Perform some actions here. */

        /* At some point the task wants to perform an operation during which it
        does not want to get swapped out, or it wants to access data which is also
        accessed from another task (but not from an interrupt). It cannot use
        taskENTER_CRITICAL()/taskEXIT_CRITICAL() because the length of the
        operation might cause interrupts to be missed. */

        /* Prevent the scheduler from performing a context switch. */

        vTaskSuspendAll();

        /* Perform the operation here. There is no need to use critical
        sections because the task has all the processing time other than that used by
        interrupt service routines.*/

        /* Calls to vTaskSuspendAll() can be nested so it is safe to call a
        (non-API) function that also contains calls to vTaskSuspendAll(). API functions
        should not be called while the scheduler is suspended. */

        vDemoFunction();

        /* The operation is complete. Set the scheduler back into the Active state. */

        if( xTaskResumeAll() == pdTRUE )

        {

            /* A context switch occurred within xTaskResumeAll(). */

        }

        else

        {

            /* A context switch did not occur within xTaskResumeAll(). */

        }
```

```
    }

}
```

taskYIELD()

The following shows the taskYIELD() macro prototype.

```
#include "FreeRTOS.h"

#include "task.h"

void taskYIELD( void );
```

Summary

Yield to another task of equal priority.

Yielding occurs when a task volunteers to leave the Running state without being preempted and before its time slice has been fully used.

Parameters

None.

Return Values

None.

Notes

taskYIELD() must only be called from an executing task and therefore must not be called while the scheduler is in the Initialization state (before the scheduler is started).

When a task calls taskYIELD(), the scheduler will select another Ready state task of equal priority to enter the Running state in its place. If there are no other Ready state tasks of equal priority, then the task that called taskYIELD() will be transitioned into the Running state.

The scheduler will only ever select a task of equal priority to the task that called taskYIELD() because, if there were any tasks of higher priority that were in the Ready state, the task that called taskYIELD() would not have been executing in the first place.

Example

The following shows the use of taskYIELD().

```
void vATask( void * pvParameters)
```

```
{

    for( ;; )

    {

        /* Perform some actions. */
        /* If there are any tasks of equal priority to this task that are in the
           Ready state, then let them execute now even though this task has not
           used all of its time slice. */

        taskYIELD();

        /* If there were any tasks of equal priority to this task in the Ready state,
        then they will have executed before this task reaches here. */

    }

}
```

Queue API

vQueueAddToRegistry()

The following shows the vQueueAddToRegistry() function prototype.

```
#include "FreeRTOS.h"

#include "queue.h"

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

Summary

Assigns a human-readable name to a queue and adds the queue to the queue registry.

Parameters

xQueue	The handle of the queue that will be added to the registry. Semaphore handles can also be used.
pcQueueName	A descriptive name for the queue or semaphore. This is not used by FreeRTOS in any way. It is included as a debugging aid only. Identifying a queue or semaphore by a human-readable name is much simpler than attempting to identify it by its handle.

Return Values

None.

Notes

The queue registry is used by kernel-aware debuggers:

1. It allows a text name to be associated with a queue or semaphore for easy queue and semaphore identification in a debugging interface.
2. It provides a means for a debugger to locate queue and semaphore structures.

The configQUEUE_REGISTRY_SIZE kernel configuration constant defines the maximum number of queues and semaphores that can be registered at any one time. Only queues and semaphores that need to be viewed in a kernel-aware debugging interface need to be registered.

The queue registry is required only when a kernel-aware debugger is being used. At all other times, it has no purpose and can be omitted by setting configQUEUE_REGISTRY_SIZE to 0 or by omitting the configQUEUE_REGISTRY_SIZE configuration constant definition altogether.

Deleting a registered queue will remove it from the registry automatically.

Example

The following shows the use of vQueueAddToRegistry().

```
void vAFunction( void )

{

    QueueHandle_t xQueue;

    /* Create a queue big enough to hold 10 chars. */

    xQueue = xQueueCreate( 10, sizeof( char ) );

    /* The created queue must be viewable in a kernel-aware debugger, so add it to the
 registry. */

    vQueueAddToRegistry( xQueue, "AMeaningfulName" );
}
```

xQueueAddToSet()

The following shows the xQueueAddToSet() function prototype.

```
#include "FreeRTOS.h"

#include "queue.h"

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

Summary

Adds a queue or semaphore to a queue set that was previously created by a call to xQueueCreateSet().

A receive (in the case of a queue) or take (in the case of a semaphore) operation must not be performed on a member of a queue set unless a call to xQueueSelectFromSet() has first returned a handle to that set member.

Parameters

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

Return Values

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

Notes

configUSE_QUEUE_SETS must be set to 1 in FreeRTOSConfig.h for the xQueueAddToSet() API function to be available.

Example

See the example for the xQueueCreateSet() function.

xQueueCreate()

The following shows the xQueueCreate() function prototype.

```
#include "FreeRTOS.h"

#include "queue.h"

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

Summary

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

Each queue requires RAM that is used to hold the queue state and items that are contained in the queue (the queue storage area). If a queue is created using xQueueCreate(), then the required RAM is automatically allocated from the FreeRTOS heap. If a queue is created using xQueueCreateStatic(), then the RAM is provided by the application writer, which results in a higher number of parameters, but allows the RAM to be statically allocated at compile time.

Parameters

uxQueueLength	The maximum number of items that the queue being created can hold at any one time.
uxItemSize	The size, in bytes, of each data item that can be stored in the queue.

Return Values

NULL	The queue cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the queue data structures and storage area.
Any other value	The queue was created successfully. The returned value is a handle by which the created queue can be referenced.

Notes

Queues are used to pass data between tasks, and between tasks and interrupts.

Queues can be created before or after the scheduler has been started.

configSUPPORT_DYNAMIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h, or simply left undefined, for this function to be available.

Example

The following shows the use of xQueueCreate().

```
/* Define the data type that will be queued. */

typedef struct A_Message

{

    char ucMessageID;

    char ucData[ 20 ];

} AMessage;

/* Define the queue parameters. */

#define QUEUE_LENGTH 5

#define QUEUE_ITEM_SIZE sizeof( AMessage )

int main( void )

{

    QueueHandle_t xQueue;

    /* Create the queue, storing the returned handle in the xQueue variable. */

    xQueue = xQueueCreate( QUEUE_LENGTH, QUEUE_ITEM_SIZE );

    if( xQueue == NULL )

    {

        /* The queue could not be created. */

    }

    /* Rest of code goes here. */

}
```

xQueueCreateSet()

The following shows the xQueueCreateSet() function prototype.

```
#include "FreeRTOS.h"

#include "queue.h"

QueueSetHandle_t xQueueCreateSet( const UBaseType_t uxEventQueueLength );
```

Summary

Queue sets provide a mechanism to allow an RTOS task to block (pend) on a read operation from multiple RTOS queues or semaphores simultaneously. Note that there are simpler alternatives to using queue sets. For more information, see Blocking on Multiple Objects on the FreeRTOS.org website.

A queue set must be explicitly created using a call to xQueueCreateSet() before it can be used. Then standard FreeRTOS queues and semaphores can be added to the set using calls to xQueueAddToSet(). xQueueSelectFromSet() is then used to determine which, if any, of the queues or semaphores contained in the set is in a state where a queue read or semaphore take operation would be successful.

Parameters

uxEventQueueLength	Queue sets store events that occur on the queues and semaphores contained in the set. uxEventQueueLength specifies the maximum number of events that can be queued at once. To be absolutely certain that events are not lost, uxEventQueueLength must be set to the sum of the lengths of the queues added to the set, where binary semaphores and mutexes have a length of 1, and counting semaphores have a length set by their maximum count value. For example: • If a queue set is to hold a queue of length 5, another queue of length 12, and a binary semaphore, then uxEventQueueLength should be set to (5 + 12 + 1), or 18. • If a queue set is to hold three binary semaphores, then uxEventQueueLength should be set to (1 + 1 + 1), or 3. • If a queue set is to hold a counting semaphore that has a maximum count of 5, and a counting semaphore that has a maximum count of 3, then uxEventQueueLength should be set to (5 + 3), or 8.

Return Values

NULL	The queue set could not be created.
Any other value	The queue set was created successfully. The returned value is a handle by which the created queue set can be referenced.

Notes

Blocking on a queue set that contains a mutex will not cause the mutex holder to inherit the priority of the blocked task.

An additional 4 bytes of RAM are required for each space in every queue added to a queue set. Therefore, a counting semaphore that has a high maximum count value should not be added to a queue set.

A receive (in the case of a queue) or take (in the case of a semaphore) operation must not be performed on a member of a queue set unless a call to xQueueSelectFromSet() has first returned a handle to that set member.

configUSE_QUEUE_SETS must be set to 1 in FreeRTOSConfig.h for the xQueueCreateSet() API function to be available.

Example

The following shows the use of xQueueCreateSet() and other queue set API functions.

```
/* Define the lengths of the queues that will be added to the queue set. */

#define QUEUE_LENGTH_1 10

#define QUEUE_LENGTH_2 10

/* Binary semaphores have an effective length of 1. */

#define BINARY_SEMAPHORE_LENGTH 1

/* Define the size of the item to be held by queue 1 and queue 2,
respectively. The values used here are just for demonstration purposes.
*/

#define ITEM_SIZE_QUEUE_1 sizeof( uint32_t )

#define ITEM_SIZE_QUEUE_2 sizeof( something_else_t )

/* The combined length of the two queues and binary semaphore that will
be added to the queue set. */

#define COMBINED_LENGTH ( QUEUE_LENGTH_1 + QUEUE_LENGTH_2 + BINARY_SEMAPHORE_LENGTH )

void vAFunction( void )

{

    static QueueSetHandle_t xQueueSet;
```

```
QueueHandle_t xQueue1, xQueue2, xSemaphore;

QueueSetMemberHandle_t xActivatedMember;

uint32_t xReceivedFromQueue1;

something_else_t xReceivedFromQueue2;

/* Create a queue set large enough to hold an event for every space in every
queue and semaphore that is to be added to the set. */

xQueueSet = xQueueCreateSet( COMBINED_LENGTH );

/* Create the queues and semaphores that will be contained in the set.*/

xQueue1 = xQueueCreate( QUEUE_LENGTH_1, ITEM_SIZE_QUEUE_1 );

xQueue2 = xQueueCreate( QUEUE_LENGTH_2, ITEM_SIZE_QUEUE_2 );

/* Create the semaphore that is being added to the set. */

xSemaphore = xSemaphoreCreateBinary();

/* Take the semaphore, so it starts empty. A block time of zero can be used
   because the semaphore is guaranteed to be available. It has just been created. */

xSemaphoreTake( xSemaphore, 0 );

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

xQueueAddToSet( xQueue1, xQueueSet );

xQueueAddToSet( xQueue2, xQueueSet );

xQueueAddToSet( xSemaphore, xQueueSet );

for( ;; )

{

    /* Block to wait for something to be available from the queues or semaphore
    that have been added to the set. Don't block longer than 200ms. */

    xActivatedMember = xQueueSelectFromSet( xQueueSet, pdMS_TO_TICKS( 200 ) );

    /* Which set member was selected? Receives/takes can use a block time of
    zero because they are guaranteed to pass. xQueueSelectFromSet() would not
    have returned the handle unless something was available. */

    if( xActivatedMember == xQueue1 )

    {

        xQueueReceive( xActivatedMember, &xReceivedFromQueue1, 0 );

        vProcessValueFromQueue1( xReceivedFromQueue1 );

    }

    else if( xActivatedQueue == xQueue2 )

    {
```

```
            xQueueReceive( xActivatedMember, &xReceivedFromQueue2, 0 );

            vProcessValueFromQueue2( &xReceivedFromQueue2 );

        }

        else if( xActivatedQueue == xSemaphore )

        {

            /* Take the semaphore to make sure it can be "given" again. */

            xSemaphoreTake( xActivatedMember, 0 );

            vProcessEventNotifiedBySemaphore();

            break;

        }

        else

        {

            /* The 200ms block time expired without an RTOS queue or semaphore being ready
 to process. */

        }

    }

}
```

xQueueCreateStatic()

The following shows the xQueueCreateStatic() function prototype.

```
#include "FreeRTOS.h"

#include "queue.h"

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

Summary

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

Each queue requires RAM that is used to hold the queue state and items that are contained in the queue (the queue storage area). If a queue is created using xQueueCreate(), then the required RAM is automatically allocated from the FreeRTOS heap. If a queue is created using xQueueCreateStatic(), then the RAM is provided by the application writer, which results in a greater number of parameters, but allows the RAM to be statically allocated at compile time.

Parameters

uxQueueLength	The maximum number of items that the queue being created can hold at any one time.
uxItemSize	The size, in bytes, of each data item that can be stored in the queue.
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 * uxItemSize) bytes. If uxItemSize is zero, then pucQueueStorageBuffer can be NULL because no date will be copied into the queue storage area.
pxQueueBuffer	Must point to a variable of type StaticQueue_t, which will be used to hold the queue's data structure.

Return Values

NULL	The queue was not created because pxQueueBuffer was NULL.

Any other value	The queue was created and the value returned is the handle of the created queue.

Notes

Queues are used to pass data between tasks, and between tasks and interrupts.

Queues can be created before or after the scheduler has been started.

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

Example

The following shows the use of xQueueCreateStatic().

```
/* The queue is to be created to hold a maximum of 10 uint64_t variables. */

#define QUEUE_LENGTH 10

#define ITEM_SIZE sizeof( uint64_t )

/* The variable used to hold the queue's data structure. */

static StaticQueue_t xStaticQueue;

/* The array to use as the queue's storage area. This must be at least (uxQueueLength *
 uxItemSize) bytes. */

uint8_t ucQueueStorageArea[ QUEUE_LENGTH * ITEM_SIZE ];

void vATask( void *pvParameters )

{

    QueueHandle_t xQueue;

    /* Create a queue capable of containing 10 uint64_t values. */

    xQueue = xQueueCreateStatic( QUEUE_LENGTH, ITEM_SIZE,  ucQueueStorageArea,
 &xStaticQueue );

    /* pxQueueBuffer was not NULL so xQueue should not be NULL. */

    configASSERT( xQueue );

}
```

vQueueDelete()

The following shows the vQueueDelete() function prototype.

```
#include "FreeRTOS.h"

#include "queue.h"

void vQueueDelete( TaskHandle_t pxQueueToDelete );
```

Summary

Deletes a queue that was previously created using a call to xQueueCreate() or xQueueCreateStatic(). vQueueDelete() can also be used to delete a semaphore.

Parameters

pxQueueToDelete	The handle of the queue being deleted. Semaphore handles can also be used.

Return Values

None

Notes

Queues are used to pass data between tasks and between tasks and interrupts.

Tasks can opt to block on a queue/semaphore (with an optional timeout) if they attempt to send data to the queue/semaphore and the queue/semaphore is already full, or they attempt to receive data from a queue/semaphore and the queue/semaphore is already empty. A queue/semaphore must not be deleted if there are any tasks currently blocked on it.

Example

The following shows the use of vQueueDelete().

```
/* Define the data type that will be queued. */

typedef struct A_Message

{
```

```
    char ucMessageID;
    char ucData[ 20 ];
 } AMessage;
/* Define the queue parameters. */
#define QUEUE_LENGTH 5
#define QUEUE_ITEM_SIZE sizeof( AMessage )
int main( void )
{
    QueueHandle_t xQueue;
    /* Create the queue, storing the returned handle in the xQueue variable. */
    xQueue = xQueueCreate( QUEUE_LENGTH, QUEUE_ITEM_SIZE );
    if( xQueue == NULL )
    {
        /* The queue could not be created. */
    }
    else
    {
        /* Delete the queue again by passing xQueue to vQueueDelete(). */
        vQueueDelete( xQueue );
    }
}
```

pcQueueGetName()

The following shows the pcQueueGetName() function prototype.

```
#include "FreeRTOS.h"

#include "queue.h"

const char *pcQueueGetName( QueueHandle_t xQueue );
```

Summary

Queries the human-readable text name of a queue.

A queue will only have a text name if it has been added to the queue registry. See the vQueueAddToRegistry() API function.

Parameters

xQueue	The handle of the queue being queried.

Return Values

Queue names are standard NULL-terminated C strings. The value returned is a pointer to the name of the queue being queried.

xQueueIsQueueEmptyFromISR()

The following shows the xQueueIsQueueEmptyFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "queue.h"

BaseType_t xQueueIsQueueEmptyFromISR( const QueueHandle_t pxQueue );
```

Summary

Queries a queue to see if it contains items or if it is already empty. Items cannot be received from a queue if the queue is empty.

This function should only be used from an ISR.

Parameters

pxQueue	The queue being queried.

Return Values

pdFALSE	The queue being queried is empty (does not contain any data items) at the time xQueueIsQueueEmptyFromISR() was called.
Any other value	The queue being queried was not empty (contained data items) at the time xQueueIsQueueEmptyFromISR() was called.

Notes

None.

xQueueIsQueueFullFromISR()

The following shows the xQueueIsQueueFullFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "queue.h"

BaseType_t xQueueIsQueueFullFromISR( const QueueHandle_t pxQueue );
```

Summary

Queries a queue to see if it is already full or if it has space to receive a new item. A queue can only successfully receive new items when it is not full.

This function should only be used from an ISR.

Parameters

pxQueue	The queue being queried.

Return Values

pdFALSE	The queue being queried is not full at the time xQueueIsQueueFullFromISR() was called.
Any other value	The queue being queried was full at the time xQueueIsQueueFullFromISR() was called.

Notes

None.

uxQueueMessagesWaiting()

The following shows the uxQueueMessagesWaiting() function prototype.

```
#include "FreeRTOS.h"

#include "queue.h"

UBaseType_t uxQueueMessagesWaiting( const QueueHandle_t xQueue );
```

Summary

Returns the number of items that are currently held in a queue.

Parameters

xQueue	The handle of the queue being queried.

Returned Value

The number of items that are held in the queue being queried at the time that uxQueueMessagesWaiting() is called.

Example

Example use of uxQueueMessagesWaiting()

```
void vAFunction( QueueHandle_t xQueue )

{

   UBaseType_t uxNumberOfItems;

   /* How many items are currently in the queue referenced by the xQueue handle? */

   uxNumberOfItems = uxQueueMessagesWaiting( xQueue );

}
```

uxQueueMessagesWaitingFromISR()

The following shows the uxQueueMessagesWaitingFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "queue.h"

UBaseType_t uxQueueMessagesWaitingFromISR( const QueueHandle_t xQueue);
```

Summary

A version of uxQueueMessagesWaiting() that can be used from inside an interrupt service routine.

Parameters

xQueue	The handle of the queue being queried.

Returned Value

The number of items that are contained in the queue being queried at the time that uxQueueMessagesWaitingFromISR() is called.

Example

The following shows the use of uxQueueMessagesWaitingFromISR().

```
void vAnInterruptHandler( void )

{

    UBaseType_t uxNumberOfItems;

    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Check the status of the queue. If it contains more than 10 items, then wake the task
 that will drain the queue. */

    /* How many items are currently in the queue referenced by the xQueue handle? */

    uxNumberOfItems = uxQueueMessagesWaitingFromISR( xQueue );

    if( uxNumberOfItems > 10 )

    {
```

```
        /* The task being woken is currently blocked on xSemaphore. Giving the
        semaphore will unblock the task. */

        xSemaphoreGiveFromISR( xSemaphore, &xHigherPriorityTaskWoken );

    }

    /* If xHigherPriorityTaskWoken is equal to pdTRUE at this point, then
    the task that was unblocked by the call to xSemaphoreGiveFromISR() had a priority
    either equal to or greater than the currently executing task (the task that was
    in the Running state when this interrupt occurred). In that case, a context
    switch should be performed before leaving this interrupt service routine to
    ensure the interrupt returns to the highest priority ready state task (the task
    that was unblocked). The syntax required to perform a context switch from inside
    an interrupt varies from port to port and from compiler to compiler. Check
    the web documentation and examples for the port in use to find the correct
    syntax for your application. */

}
```

xQueueOverwrite()

The following shows the xQueueOverwrite() function prototype.

```
#include "FreeRTOS.h"

#include "queue.h"

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

Summary

A version of xQueueSendToBack() that will write to the queue even if the queue is full, overwriting data that is already held in the queue.

xQueueOverwrite() is intended for use with queues that have a length of one, meaning the queue is either empty or full.

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

Parameters

xQueue	The handle of the queue to which the data is to be sent.
pvItemToQueue	A pointer to the item that is to be placed in the queue. The size of each item the queue can hold is set when the queue is created, and that many bytes will be copied from pvItemToQueue into the queue storage area.

Returned Value

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

The following shows the use of xQueueOverwrite().

```
void vFunction( void *pvParameters )
```

```
{

    QueueHandle_t xQueue;

    unsigned long ulVarToSend, ulValReceived;

    /* Create a queue to hold one unsigned long value. We strongly
    recommended that you do not use xQueueOverwrite() on queues that can
    contain more than one value. Doing so will trigger an assertion
    if configASSERT() is defined. */

    xQueue = xQueueCreate( 1, sizeof( unsigned long ) );

    /* 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 another task removed the value. */

    }

    /* 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 unless another task is using the same queue. */

    }

    /* ... */

}
```

xQueueOverwriteFromISR()

The following shows the xQueueOverwriteFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "queue.h"

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

Summary

A version of xQueueOverwrite() that can be used in an ISR. xQueueOverwriteFromISR() is similar to xQueueSendToBackFromISR(), but will write to the queue even if the queue is full, overwriting data that is already held in the queue.

xQueueOverwriteFromISR() is intended for use with queues that have a length of one, meaning the queue is either empty or full.

Parameters

xQueue	The handle of the queue to which the data is to be sent.
pvItemToQueue	A pointer to the item that is to be placed in the queue. The size of each item the queue can hold is set when the queue is created, and that many bytes will be copied from pvItemToQueue into the queue storage area.
pxHigherPriorityTaskWoken	xQueueOverwriteFromISR() will set <problematic>*</problematic> 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. For information, see the Interrupt Service Routines section of the documentation.

Returned Value

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

The following shows the use of xQueueOverwriteFromISR().

```
QueueHandle_t xQueue;

void vFunction( void *pvParameters )

{

    /* Create a queue to hold one unsigned long value. We strongly
    recommend that you do not use xQueueOverwriteFromISR() on queues that can
    contain more than one value. Doing so will trigger an assertion
    if configASSERT() is defined. */
    xQueue = xQueueCreate( 1, sizeof( unsigned long ) );

}

void vAnInterruptHandler( void )

{

    /* xHigherPriorityTaskWoken must be set to pdFALSE before it is used.*/

    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    unsigned long 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.*/

    }

}
```

xQueuePeek()

The following shows the xQueuePeek() function prototype.

```
#include "FreeRTOS.h"

#include "queue.h"

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

Summary

Reads an item from a queue, but without removing the item from the queue. The same item will be returned the next time xQueueReceive() or xQueuePeek() is used to obtain an item from the same queue.

Parameters

xQueue	The handle of the queue from which data is to be read.
pvBuffer	A pointer to the memory into which the data read from the queue will be copied. The length of the buffer must be at least equal to the queue item size. The item size will have been set by the uxItemSize parameter of the call to xQueueCreate() or xQueueCreateStatic() used to create the queue.
xTicksToWait	The maximum amount of time the task should remain in the Blocked state to wait for data to become available on the queue, should the queue already be empty. If xTicksToWait is zero, then xQueuePeek() will return immediately if the queue is already empty. The block time is specified in tick periods, so the absolute time it represents depends on the tick frequency. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds to 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.

Return Values

pdPASS	Returned if data was successfully read from the queue. If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state, to wait for data to become available on the queue, but data was successfully read from the queue before the block time expired.
errQUEUE_EMPTY	Returned if data cannot be read from the queue because the queue is already empty. If a block time was specified (xTicksToWait was not zero), then the calling task will have been placed into the Blocked state to wait for another task or interrupt to send data to the queue, but the block time expired before this happened.

Notes

None.

Example

Example use of xQueuePeek()

```
struct AMessage

{

    char ucMessageID;

    char ucData[ 20 ];

} xMessage;

QueueHandle_t xQueue;

/* Task that creates a queue and posts a value. */

void vATask( void *pvParameters )

{

    struct AMessage *pxMessage;

    /* Create a queue capable of containing 10 pointers to AMessage
    structures. Store the handle to the created queue in the xQueue variable. */

    xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
```

```
    if( xQueue == 0 )

    {
        /* The queue was not created because there was not enough FreeRTOS heap
        memory available to allocate the queues data structures or storage area.*/
    }

    else

    {

        /* ... */

        /* Send a pointer to a struct AMessage object to the queue referenced by
        the xQueue variable. Don't block if the queue is already full (the third
        parameter to xQueueSend() is zero, so not block time is specified). */

        pxMessage = &xMessage;

        xQueueSend( xQueue, ( void * ) &pxMessage, 0 );

    }

    /* ... Rest of the task code. */

    for( ;; )

    {

    }

}

/* 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
 available immediately. */

        if( xQueuePeek( xQueue, &( pxRxedMessage ), 10 ) == pdPASS )

        {

                /* pxRxedMessage now points to the struct AMessage variable posted by
                vATask, but the item still remains on the queue. */

        }

    }

    else

    {

        /* The queue could not or has not been created. */

    }
```

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

    for( ;; )

    {

    }

}
```

xQueuePeekFromISR()

The following shows the xQueuePeekFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "queue.h"

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

Summary

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

Reads an item from a queue, but without removing the item from the queue. The same item will be returned the next time xQueueReceive() or xQueuePeek() is used to obtain an item from the same queue.

Parameters

xQueue	The handle of the queue from which data is to be read.
pvBuffer	A pointer to the memory into which the data read from the queue will be copied. The length of the buffer must be at least equal to the queue item size. The item size will have been set by the uxItemSize parameter of the call to xQueueCreate() or xQueueCreateStatic() used to create the queue.

Return Values

pdPASS	Returned if data was successfully read from the queue.
errQUEUE_EMPTY	Returned if data cannot be read from the queue because the queue is already empty.

Notes

None.

xQueueReceive()

The following shows the xQueueReceive() function prototype.

```
#include "FreeRTOS.h"

#include "queue.h"

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

Summary

Receive (read) an item from a queue.

Parameters

xQueue	The handle of the queue from which the data is being received (read). The queue handle will have been returned from the call to xQueueCreate() or xQueueCreateStatic() used to create the queue.
pvBuffer	A pointer to the memory into which the received data will be copied. The length of the buffer must be at least equal to the queue item size. The item size will have been set by the uxItemSize parameter of the call to xQueueCreate() or xQueueCreateStatic() used to create the queue.
xTicksToWait	The maximum amount of time the task should remain in the Blocked state to wait for data to become available on the queue, should the queue already be empty. If xTicksToWait is zero, then xQueueReceive() will return immediately if the queue is already empty. The block time is specified in tick periods, so the absolute time it represents is depends on the tick frequency. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds to 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.

Return Values

pdPASS	Returned if data was successfully read from the queue. If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state, to wait for data to become available on the queue, but data was successfully read from the queue before the block time expired.
errQUEUE_EMPTY	Returned if data cannot be read from the queue because the queue is already empty. If a block time was specified (xTicksToWait was not zero), then the calling task will have been placed into the Blocked state to wait for another task or interrupt to send data to the queue, but the block time expired before this happened.

Notes

None.

Example

The following shows the use of xQueueReceive().

```
/* Define the data type that will be queued. */

typedef struct A_Message

{

    char ucMessageID;

    char ucData[ 20 ];

} AMessage;

/* Define the queue parameters. */

#define QUEUE_LENGTH 5

#define QUEUE_ITEM_SIZE sizeof( AMessage )

int main( void )

{

    QueueHandle_t xQueue;

    /* Create the queue, storing the returned handle in the xQueue variable. */
```

```
    xQueue = xQueueCreate( QUEUE_LENGTH, QUEUE_ITEM_SIZE );

    if( xQueue == NULL )

    {

        /* The queue could not be created. Do something. */

    }

    /* Create a task, passing in the queue handle as the task parameter.*/

    xTaskCreate( vAnotherTask, "Task", STACK_SIZE, ( void * ) xQueue, /* The queue handle
 is used as the task parameter. */ TASK_PRIORITY, NULL );

    /* Start the task executing. */

    vTaskStartScheduler();

    /* Execution will only reach here if there was not enough FreeRTOS heap memory
 remaining for the idle task to be created. */

    for( ;; );

}

void vAnotherTask( void *pvParameters )

{

    QueueHandle_t xQueue;

    AMessage xMessage;

    /* The queue handle is passed into this task as the task parameter. Cast the
    void * parameter back to a queue handle. */

    xQueue = ( QueueHandle_t ) pvParameters;

    for( ;; )

    {

        /* Wait for the maximum period for data to become available on the queue.
        The period will be indefinite if INCLUDE_vTaskSuspend is set to 1 in
        FreeRTOSConfig.h. */

        if( xQueueReceive( xQueue, &xMessage, portMAX_DELAY ) != pdPASS )

        {

            /* Nothing was received from the queue even after blocking to wait for data to
 arrive. */

        }

        else

        {

            /* xMessage now contains the received data. */

        }

    }
```

```
}
```

xQueueReceiveFromISR()

The following shows the xQueueReceiveFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "queue.h"

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

Summary

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

Parameters

xQueue	The handle of the queue from which the data is being received (read). The queue handle will have been returned from the call to xQueueCreate() or xQueueCreateStatic() used to create the queue.
pvBuffer	A pointer to the memory into which the received data will be copied. The length of the buffer must be at least equal to the queue item size. The item size will have been set by the uxItemSize parameter of the call to xQueueCreate() or xQueueCreateStatic() used to create the queue.
pxHigherPriorityTaskWoken	It is possible that a single queue will have one or more tasks blocked on it waiting for space to become available on the queue. Calling xQueueReceiveFromISR() can make space available, and so cause such a task to leave the Blocked state. If calling the API function causes a task to leave the Blocked state, and the unblocked task has a priority equal to or higher than the currently executing task (the task that was interrupted), then, internally, the API function will set <problematic>*</problematic> pxHigherPriorityTaskWoken to pdTRUE. If xQueueReceiveFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt is exited. This will

	ensure that the interrupt returns directly to the highest priority Ready state task. From FreeRTOS V7.3.0 pxHigherPriorityTaskWoken is an optional parameter and can be set to NULL.

Return Values

pdPASS	Data was successfully received from the queue.
pdFAIL	Data was not received from the queue because the queue was already empty.

Notes

Calling xQueueReceiveFromISR() within an interrupt service routine can potentially cause a task that was blocked on a queue to leave the Blocked state. A context switch should be performed if such an unblocked task has a priority higher than or equal to the currently executing task (the task that was interrupted). The context switch will ensure that the interrupt returns directly to the highest priority Ready state task. Unlike the xQueueReceive() API function, xQueueReceiveFromISR() will not perform a context switch. Instead, it will just indicate whether a context switch is required.

xQueueReceiveFromISR() must not be called before the scheduler is started. Therefore, an interrupt that calls xQueueReceiveFromISR() must not be allowed to execute before the scheduler is started.

Example

For clarity of demonstration, the example in this section makes multiple calls to xQueueReceiveFromISR() to receive multiple small data items. This is inefficient and therefore not recommended for most applications. A better approachis to send the multiple data items in a structure to the queue in a single post, allowing xQueueReceiveFromISR() to be called only once. Alternatively, and preferably, processing can be deferred to the task level.

```
/* vISR is an interrupt service routine that empties a queue of values,
sending each to a peripheral. It might be that there are multiple tasks blocked on
the queue waiting for space to write more data to the queue. */

void vISR( void )

{

    char cByte;

    BaseType_t xHigherPriorityTaskWoken;

    /* No tasks have yet been unblocked. */

    xHigherPriorityTaskWoken = pdFALSE;

    /* Loop until the queue is empty. xHigherPriorityTaskWoken will get set to pdTRUE
 internally within
```

```
    xQueueReceiveFromISR() if calling xQueueReceiveFromISR()caused a task to
    leave the Blocked state, and the unblocked task has a priority equal to or
    greater than the task currently in the Running state (the task this ISR interrupted).
    */

    while( xQueueReceiveFromISR( xQueue, &cByte, &xHigherPriorityTaskWoken ) == pdPASS )

    {

        /* Write the received byte to the peripheral. */

        OUTPUT_BYTE( TX_REGISTER_ADDRESS, cByte );

    }

    /* Clear the interrupt source. */

    /* Now that the queue is empty and we have cleared the interrupt, we can
    perform a context switch if one is required (if xHigherPriorityTaskWoken has been
    set to pdTRUE).
    NOTE: The syntax required to perform a context switch from an ISR varies
    from port to port and from compiler to compiler. Check the web documentation and
    examples for the port being used to find the correct syntax required for
    your application. */

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

xQueueRemoveFromSet()

The following shows the xQueueRemoveFromSet() function prototype.

```
#include "FreeRTOS.h"

#include "queue.h"

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

Summary

Remove a queue or semaphore from a queue set.

A queue or semaphore can only be removed from a queue set if the queue or semaphore is empty.

Parameters

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

Return Values

pdPASS	The queue or semaphore was successfully removed from the queue set.
pdFAIL	The queue or semaphore was not removed from the queue set because either the queue or semaphore was not in the queue set, or the queue or semaphore was not empty.

Notes

configUSE_QUEUE_SETS must be set to 1 in FreeRTOSConfig.h for the xQueueRemoveFromSet() API function to be available.

Example

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

```
if( xQueueRemoveFromSet( xQueue, xQueueSet ) != pdPASS )

{

    /* Either xQueue was not a member of the xQueueSet set, or xQueue is not empty and
 therefore cannot be removed from the set. */

}

else

{

    /* The queue was successfully removed from the set. */

}
```

xQueueReset()

The following shows the xQueueReset() function prototype. .. code:

```
#include "FreeRTOS.h"

#include "queue.h"

BaseType_t xQueueReset( QueueHandle_t xQueue );
```

Summary

Resets a queue to its original empty state. Any data contained in the queue at the time it is reset is discarded.

Parameters

xQueue	The handle of the queue that is being reset. The queue handle will have been returned from the call to xQueueCreate() or xQueueCreateStatic() used to create the queue.

Return Values

Original versions of xQueueReset() returned pdPASS or pdFAIL. Since FreeRTOS V7.2.0, xQueueReset() always returns pdPASS.

xQueueSelectFromSet()

The following shows the xQueueSelectFromSet() function prototype.

```
#include "FreeRTOS.h"

#include "queue.h"

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

Summary

xQueueSelectFromSet() selects from the members of a queue set a queue or semaphore that either contains data (in the case of a queue) or is available to take (in the case of a semaphore). xQueueSelectFromSet() effectively allows a task to block (pend) on a read operation on all the queues and semaphores in a queue set simultaneously.

Parameters

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

Return Values

NULL	A queue or semaphore contained in the set did not become available before the block time specified by the xTicksToWait parameter expired.
Any other value	The handle of a queue (cast to a QueueSetMemberHandle_t type) contained in the queue set that contains data, or the handle of a semaphore (cast to a QueueSetMemberHandle_t type) contained in the queue set that is available.

Notes

configUSE_QUEUE_SETS must be set to 1 in FreeRTOSConfig.h for the xQueueSelectFromSet() API function to be available.

There are simpler alternatives to using queue sets. For more information, see Blocking on Multiple Objects on the FreeRTOS.org website.

Blocking on a queue set that contains a mutex will not cause the mutex holder to inherit the priority of the blocked task.

A receive (in the case of a queue) or take (in the case of a semaphore) operation must not be performed on a member of a queue set unless a call to xQueueSelectFromSet() has first returned a handle to that set member.

Example

See the example provided for the xQueueCreateSet() function.

xQueueSelectFromSetFromISR()

The following shows the xQueueSelectFromSetFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "queue.h"

QueueSetMemberHandle_t xQueueSelectFromSetFromISR( QueueSetHandle_t xQueueSet );
```

Summary

A version of xQueueSelectFromSet() that can be used from an interrupt service routine.

Parameters

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

Return Values

NULL	No members of the queue set were available.
Any other value	The handle of a queue (cast to a QueueSetMemberHandle_t type) contained in the queue set that contains data, or the handle of a semaphore (cast to a QueueSetMemberHandle_t type) contained in the queue set that is available.

Notes

configUSE_QUEUE_SETS must be set to 1 in FreeRTOSConfig.h for the xQueueSelectFromSetFromISR() API function to be available.

Example

The following shows the use of xQueueSelectFromSetFromISR().

```
void vReceiveFromQueueInSetFromISR( void )
```

```
{
    QueueSetMemberHandle_t xActivatedQueue;

    unsigned long ulReceived;

    /* See if any of the queues in the set contain data. */

    xActivatedQueue = xQueueSelectFromSetFromISR( xQueueSet );

    if( xActivatedQueue != NULL )

    {

        /* Reading from the queue returned by xQueueSelectFromSetFormISR(). */

        if( xQueueReceiveFromISR( xActivatedQueue, &ulReceived, NULL ) != pdPASS)

        {

             /* Data should have been available as the handle was returned from
 xQueueSelectFromSetFromISR(). */

        }

    }

}
```

xQueueSend(), xQueueSendToFront(), xQueueSendToBack()

The following shows the xQueueSend(), xQueueSendToFront() and xQueueSendToBack() function prototypes.

```
#include "FreeRTOS.h"

#include "queue.h"

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

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

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

Summary

Sends (writes) an item to the front or the back of a queue.

xQueueSend() and xQueueSendToBack() perform the same operation so are equivalent. Both send data to the back of a queue. xQueueSend() was the original version, and it is now recommended to use xQueueSendToBack() in its place.

Parameters

xQueue	The handle of the queue to which the data is being sent (written). The queue handle will have been returned from the call to xQueueCreate() or xQueueCreateStatic() used to create the queue.
pvItemToQueue	A pointer to the data to be copied into the queue. The size of each item the queue can hold is set when the queue is created, and that many bytes will be copied from pvItemToQueue into the queue storage area.

xTicksToWait	The maximum amount of time the task should remain in the Blocked state to wait for space to become available on the queue, should the queue already be full. xQueueSend(), xQueueSendToFront() and xQueueSendToBack() will return immediately if xTicksToWait is zero and the queue is already full. The block time is specified in tick periods, so the absolute time it represents is depends on the tick frequency. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds to 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.

Return Values

pdPASS	Returned if data was successfully sent to the queue. If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state, to wait for space to become available in the queue before the function returned, but data was successfully written to the queue before the block time expired.
errQUEUE_FULL	Returned if data could not be written to the queue because the queue was already full. If a block time was specified (xTicksToWait was not zero), then the calling task will have been placed into the Blocked state to wait for another task or interrupt to make room in the queue, but the specified block time expired before that happened.

Notes

None.

Example

The following shows the use of xQueueSendToBack().

```
/* Define the data type that will be queued. */
```

```
typedef struct A_Message {

        char ucMessageID;

        char ucData[ 20 ];

} AMessage;

/* Define the queue parameters. */

#define QUEUE_LENGTH 5

#define QUEUE_ITEM_SIZE sizeof( AMessage )

int main( void )

{

    QueueHandle_t xQueue;

    /* Create the queue, storing the returned handle in the xQueue variable. */

    xQueue = xQueueCreate( QUEUE_LENGTH, QUEUE_ITEM_SIZE );

    if( xQueue == NULL )

    {

        /* The queue could not be created. Do something. */

    }

    /* Create a task, passing in the queue handle as the task parameter. */

    xTaskCreate( vAnotherTask, "Task", STACK_SIZE, ( void * ) xQueue, /* xQueue is used as
 the task parameter. */ TASK_PRIORITY, NULL );

    /* Start the task executing. */

    vTaskStartScheduler();

    /* Execution will only reach here if there was not enough FreeRTOS heap memory
 remaining for the idle task to be created. */

    for( ;; );

}

void vATask( void *pvParameters )

{

    QueueHandle_t xQueue;

    AMessage xMessage;

    /* The queue handle is passed into this task as the task parameter. Cast the parameter
 back to a queue handle. */

    xQueue = ( QueueHandle_t ) pvParameters;

    for( ;; )

    {
```

```
        /* Create a message to send on the queue. */

        xMessage.ucMessageID = SEND_EXAMPLE;

        /* Send the message to the queue, waiting for 10 ticks for space to become
 available if the queue is already full. */

        if( xQueueSendToBack( xQueue, &xMessage, 10 ) != pdPASS )

        {

            /* Data could not be sent to the queue even after waiting 10 ticks. */

        }

    }

}
```

xQueueSendFromISR(), xQueueSendToBackFromISR(), xQueueSendToFrontFromISR()

The following shows the listings for the xQueueSendFromISR(), xQueueSendToBackFromISR(), and xQueueSendToFrontFromISR() function prototypes.

```
#include "FreeRTOS.h"

#include "queue.h"

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

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

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

Summary

Versions of the xQueueSend(), xQueueSendToFront(), and xQueueSendToBack() API functions that can be called from an ISR. Unlike xQueueSend(), xQueueSendToFront(), and xQueueSendToBack(), the ISR-safe versions do not permit a block time to be specified.

xQueueSendFromISR() and xQueueSendToBackFromISR() perform the same operation, so they are equivalent. Both send data to the back of a queue. xQueueSendFromISR() was the original version. We recommend you use xQueueSendToBackFromISR() instead.

Parameters

xQueue	The handle of the queue to which the data is being sent (written). The queue handle will have been returned from the call to xQueueCreate() or xQueueCreateStatic() used to create the queue.
pvItemToQueue	A pointer to the data to be copied into the queue. The size of each item the queue can hold is set when the queue is created, and that many bytes will be copied from pvItemToQueue into the queue storage area.
pxHigherPriorityTaskWoken	It is possible that a single queue will have one or more tasks blocked on it waiting for data to

	become available. Calling xQueueSendFromISR(), xQueueSendToFrontFromISR() or xQueueSendToBackFromISR() can make data available, and so cause such a task to leave the Blocked state. If calling the API function causes a task to leave the Blocked state, and the unblocked task has a priority equal to or higher than the currently executing task (the task that was interrupted), then, internally, the API function will set <problematic>*</problematic> pxHigherPriorityTaskWoken to pdTRUE. If xQueueSendFromISR(), xQueueSendToFrontFromISR() or xQueueSendToBackFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt is exited. This will ensure that the interrupt returns directly to the highest priority Ready state task. From FreeRTOS V7.3.0 pxHigherPriorityTaskWoken is an optional parameter and can be set to NULL.

Return Values

pdTRUE	Data was successfully sent to the queue.
errQUEUE_FULL	Data could not be sent to the queue because the queue was already full.

Notes

Calling xQueueSendFromISR(), xQueueSendToBackFromISR(), or xQueueSendToFrontFromISR() within an interrupt service routine can potentially cause a task that was blocked on a queue to leave the Blocked state. A context switch should be performed if such an unblocked task has a priority higher than or equal to the currently executing task (the task that was interrupted). The context switch will ensure that the interrupt returns directly to the highest priority Ready state task. Unlike the xQueueSend(), xQueueSendToBack(), and xQueueSendToFront() API functions, xQueueSendFromISR(), xQueueSendToBackFromISR(), and xQueueSendToFrontFromISR() will not perform a context switch. Instead, they just indicate whether a context switch is required.

xQueueSendFromISR(), xQueueSendToBackFromISR(), and xQueueSendToFrontFromISR() must not be called before the scheduler is started. Therefore, an interrupt that calls any of these functions must not be allowed to execute before the scheduler is started.

Example

For clarity of demonstration, the following example makes multiple calls to xQueueSendToBackFromISR() to send multiple small data items. This is inefficient and therefore not recommended. Better approaches include:

1. Packing the multiple data items into a structure, and then using a single call to xQueueSendToBackFromISR() to send the entire structure to the queue. This approach is appropriate only if the number of data items is small.
2. Writing the data items into a circular RAM buffer, and then using a single call to xQueueSendToBackFromISR() to let a task know how many new data items the buffer contains.

```
/* vBufferISR() is an interrupt service routine that empties a buffer
of values, writing each value to a queue. It might be that there are multiple tasks
blocked on the queue waiting for the data. */

void vBufferISR( void )

{

    char cIn;

    BaseType_t xHigherPriorityTaskWoken;

    /* No tasks have yet been unblocked. */

    xHigherPriorityTaskWoken = pdFALSE;

    /* Loop until the buffer is empty. */

    do

    {

        /* Obtain a byte from the buffer. */

        cIn = INPUT_BYTE( RX_REGISTER_ADDRESS );

        /* Write the byte to the queue. xHigherPriorityTaskWoken will get set
        to pdTRUE if writing to the queue causes a task to leave the Blocked state,
        and the task leaving the Blocked state has a priority higher than the
        currently executing task (the task that was interrupted). */

        xQueueSendToBackFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );

    } while( INPUT_BYTE( BUFFER_COUNT ) );

    /* Clear the interrupt source here. */

    /* Now that the buffer is empty and the interrupt source has been cleared,
    a context switch should be performed if xHigherPriorityTaskWoken is equal to pdTRUE.
    NOTE: The syntax required to perform a context switch from an ISR varies from
    port to port and from compiler to compiler. Check the web documentation and
    examples for the port being used to find the syntax required for your application. */

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

uxQueueSpacesAvailable()

The following shows the uxQueueSpacesAvailable() function prototype.

```
#include "FreeRTOS.h"

#include "queue.h"

UBaseType_t uxQueueSpacesAvailable( const QueueHandle_t xQueue );
```

Summary

Returns the number of free spaces that are available in a queue (that is, the number of items that can be posted to the queue before the queue becomes full).

Parameters

xQueue	The handle of the queue being queried.

Returned Value

The number of free spaces that are available in the queue being queried at the time uxQueueSpacesAvailable() is called.

Example

The following shows the use of uxQueueSpacesAvailable().

```
void vAFunction( QueueHandle_t xQueue )

{

    UBaseType_t uxNumberOfFreeSpaces;

    /* How many free spaces are currently available in the queue referenced by the xQueue
 handle? */

    uxNumberOfFreeSpaces = uxQueueSpacesAvailable( xQueue );

}
```

Semaphore API

vSemaphoreCreateBinary()

The following shows the vSemaphoreCreateBinary() macro prototype.

```
#include "FreeRTOS.h"

#include "semphr.h"

void vSemaphoreCreateBinary( SemaphoreHandle_t xSemaphore );
```

Summary

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

A macro that creates a binary semaphore. A semaphore must be explicitly created before it can be used.

Parameters

xSemaphore	Variable of type SemaphoreHandle_t that will store the handle of the semaphore being created.

Return Values

None.

If, following a call to vSemaphoreCreateBinary(), xSemaphore is equal to NULL, then the semaphore cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the semaphore data structures. In all other cases, xSemaphore will hold the handle of the created semaphore.

Notes

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

Binary semaphores A binary semaphore used for synchronization does not need to be 'given' back after it has been successfully 'taken' (obtained). Task synchronization is implemented by having one task or interrupt 'give' the semaphore and another task 'take' the semaphore. (See the xSemaphoreGiveFromISR() documentation.)

Mutexes The priority of a task that holds a mutex will be raised if another task of higher priority attempts to obtain the same mutex. The task that already holds the mutex is said to 'inherit' the priority

of the task that is attempting to 'take' the same mutex. The inherited priority will be 'disinherited' when the mutex is returned (the task that inherited a higher priority while it held a mutex will return to its original priority when the mutex is returned).

A task that obtains a mutex that is used for mutual exclusion must always give the mutex back. Otherwise, no other task will ever be able to obtain the same mutex. The xSemaphoreTake() section includes an example of a mutex being used to implement mutual exclusion.

Mutexes and binary semaphores are both referenced using variables that have an SemaphoreHandle_t type and can be used in any API function that takes a parameter of that type.

Mutexes and binary semaphores that were created using the old vSemaphoreCreateBinary() macro, as opposed to the preferred xSemaphoreCreateBinary() function, are both created such that the first call to xSemaphoreTake() on the semaphore or mutex will pass. Note vSemaphoreCreateBinary() is deprecated and must not be used in new applications. Binary semaphores created using the xSemaphoreCreateBinary() function are created 'empty', so the semaphore must first be given before the semaphore can be taken (obtained) using a call to xSemaphoreTake().

Example

Example use of vSemaphoreCreateBinary()

```
SemaphoreHandle_t xSemaphore;

void vATask( void * pvParameters )

{

    /* Attempt to create a semaphore.
    *NOTE*: New designs should use the xSemaphoreCreateBinary() function,
    not the vSemaphoreCreateBinary() macro. */

    vSemaphoreCreateBinary( xSemaphore );

    if( xSemaphore == NULL )

    {

        /* There was insufficient FreeRTOS heap available for the semaphore to be created
 successfully. */

    }

    else

    {

        /* The semaphore can now be used. Its handle is stored in the xSemaphore variable.
 */

    }

}
```

xSemaphoreCreateBinary()

The following shows the xSemaphoreCreateBinary() function prototype.

Summary

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

Each binary semaphore requires a small amount of RAM that is used to hold the semaphore's state. If a binary semaphore is created using xSemaphoreCreateBinary() then the required RAM is automatically allocated from the FreeRTOS heap. If a binary semaphore is created using xSemaphoreCreateBinaryStatic() then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

The semaphore is created in the 'empty' state, meaning the semaphore must first be given before it can be taken (obtained) using the xSemaphoreTake() function.

Parameters

None.

Return Values

NULL	The semaphore could not be created because there was insufficient heap memory available for FreeRTOS to allocate the semaphore data structures.
Any other value	The semaphore was created successfully. The returned value is a handle by which the created semaphore can be referenced.

Notes

Direct-to-task notifications normally provide a lightweight and faster alternative to binary semaphores.

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

Binary semaphores A binary semaphore used for synchronization does not need to be 'given' back after it has been successfully 'taken' (obtained). Task synchronization is implemented by having one task or interrupt 'give' the semaphore, and another task 'take' the semaphore. (See xSemaphoreGiveFromISR().) The same functionality can often be achieved in a more efficient way by using a direct-to-task notification.

Mutexes The priority of a task that holds a mutex will be raised if another task of higher priority attempts to obtain the same mutex. The task that already holds the mutex is said to 'inherit' the priority of the task that is attempting to 'take' the same mutex. The inherited priority will be 'disinherited' when the mutex is returned (the task that inherited a higher priority while it held a mutex will return to its original priority when the mutex is returned).

A task that obtains a mutex that is used for mutual exclusion must always give the mutex back. Otherwise, no other task will ever be able to obtain the same mutex. An example of a mutex being used to implement mutual exclusion is provided in the xSemaphoreTake() section.

Mutexes and binary semaphores are both referenced by using variables that have an SemaphoreHandle_t type. They can be used in any API function that takes a parameter of that type.

Mutexes and binary semaphores that were created by using the vSemaphoreCreateBinary() macro (as opposed to the preferred xSemaphoreCreateBinary() function) are created such that the first call to xSemaphoreTake() on the semaphore or mutex will pass. vSemaphoreCreateBinary() is deprecated. Do not use it in new applications. Binary semaphores created by using the xSemaphoreCreateBinary() function are created empty, so the semaphore must first be given before the semaphore can be taken (obtained) using a call to xSemaphoreTake().

configSUPPORT_DYNAMIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h, or simply left undefined, for this function to be available.

Example

The following shows the use of xSemaphoreCreateBinary().

```
SemaphoreHandle_t xSemaphore;

void vATask( void * pvParameters )

{

    /* Attempt to create a semaphore. */

    xSemaphore = xSemaphoreCreateBinary();

    if( xSemaphore == NULL )

    {

            /* There was insufficient FreeRTOS heap available for the semaphore to be
 created successfully. */

    }

    else

    {

            /* The semaphore can now be used. Its handle is stored in the xSemaphore
            variable. Calling xSemaphoreTake() on the semaphore here will fail until
            the semaphore has first been given. */

    }

}
```

xSemaphoreCreateBinaryStatic()

The following shows the xSemaphoreCreateBinaryStatic() function prototype.

```
#include "FreeRTOS.h"

#include "semphr.h"

SemaphoreHandle_t xSemaphoreCreateBinaryStatic( StaticSemaphore_t *pxSemaphoreBuffer );
```

Summary

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

Each binary semaphore requires a small amount of RAM that is used to hold the semaphore's state. If a binary semaphore is created using xSemaphoreCreateBinary(), then the required RAM is allocated from the FreeRTOS heap automatically. If a binary semaphore is created by using xSemaphoreCreateBinaryStatic(), then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

The semaphore is created in the 'empty' state, which means the semaphore must first be given before it can be taken (obtained) by using the xSemaphoreTake() function.

Parameters

pxSemaphoreBuffer	Must point to a variable of type StaticSemaphore_t, which will be used to hold the semaphore's state.

Return Values

NULL	The semaphore could not be created because pxSemaphoreBuffer was NULL.
Any other value	The semaphore was created successfully. The returned value is a handle by which the created semaphore can be referenced.

Notes

Direct-to-task notifications normally provide a lightweight and faster alternative to binary semaphores.

Binary semaphores and mutexes are very similar, but do have some subtle differences. Mutexes include a priority inheritance mechanism. Binary semaphores do not. This makes binary semaphores the better

choice for implementing synchronization (between tasks or between an interrupt and a task), and mutexes the better choice for implementing simple mutual exclusion.

Binary semaphores A binary semaphore used for synchronization does not need to be 'given' back after it has been successfully 'taken' (obtained). Task synchronization is implemented by having one task or interrupt 'give' the semaphore, and another task 'take' the semaphore. (See the xSemaphoreGiveFromISR() documentation.) The same functionality can often be achieved in a more efficient way by using a direct-to-task notification.

Mutexes The priority of a task that holds a mutex will be raised if another task of higher priority attempts to obtain the same mutex. The task that already holds the mutex is said to 'inherit' the priority of the task that is attempting to 'take' the same mutex. The inherited priority will be 'disinherited' when the mutex is returned (the task that inherited a higher priority while it held a mutex will return to its original priority when the mutex is returned).

A task that obtains a mutex that is used for mutual exclusion must always give the mutex back. Otherwise, no other task will ever be able to obtain the same mutex. An example of a mutex used to implement mutual exclusion is provided in the xSemaphoreTake() section.

Mutexes and binary semaphores are both referenced using variables that have an SemaphoreHandle_t type, and can be used in any API function that takes a parameter of that type.

Mutexes and binary semaphores that were created by using the vSemaphoreCreateBinary() macro (as opposed to the preferred xSemaphoreCreateBinary() function) are both created such that the first call to xSemaphoreTake() on the semaphore or mutex will pass. vSemaphoreCreateBinary() is deprecated. Do not use it in new applications. Binary semaphores created by using the xSemaphoreCreateBinary() function are created 'empty', so the semaphore must first be given before the semaphore can be taken (obtained) using a call to xSemaphoreTake().

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

Example

The following shows the use of xSemaphoreCreateBinaryStatic().

```
SemaphoreHandle_t xSemaphoreHandle;

StaticSemaphore_t xSemaphoreBuffer;

void vATask( void * pvParameters )

{

    /* Create a binary semaphore without using any dynamic memory allocation. */

    xSemaphoreHandle = xSemaphoreCreateBinaryStatic( &xSemaphoreBuffer );

    /* pxSemaphoreBuffer was not NULL so the binary semaphore will have
    been created, and xSemaphoreHandle will be a valid handle. The rest of the task code
 goes here. */

}
```

xSemaphoreCreateCounting()

The following shows the xSemaphoreCreateCounting() function prototype.

```
#include "FreeRTOS.h"

#include "semphr.h"

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

Summary

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

Each counting semaphore requires a small amount of RAM that is used to hold the semaphore's state. If a counting semaphore is created using xSemaphoreCreateCounting(), then the required RAM is allocated from the FreeRTOS heap automatically. If a counting semaphore is created using xSemaphoreCreateCountingStatic(), then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

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.

Return Values

NULL	Returned if the semaphore cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the semaphore data structures.
Any other value	The semaphore was created successfully. The returned value is a handle by which the created semaphore can be referenced.

Notes

Direct-to-task notifications normally provide a lightweight and faster alternative to counting semaphores.

Counting semaphores are typically used for two things:

1. Counting events.

In this usage scenario, an event handler will 'give' the semaphore each time an event occurs, and a handler task will 'take' the semaphore each time it processes an event.

The semaphore's count value will be incremented each time it is 'given' and decremented each time it is 'taken'. The count value is therefore the difference between the number of events that have occurred and the number of events that have been processed.

Semaphores created to count events should be created with an initial count value of zero because no events will have been counted before the semaphore was created.

1. Resource management.

In this usage scenario, the count value of the semaphore represents the number of available resources.

To obtain control of a resource, a task must first successfully 'take' the semaphore. The action of 'taking' the semaphore will decrement the semaphore's count value. When the count value reaches zero, no more resources are available, and attempts to 'take' the semaphore will fail.

When a task finishes with a resource, it must 'give' the semaphore. The action of 'giving' the semaphore will increment the semaphore's count value, indicating that a resource is available, and allowing future attempts to 'take' the semaphore to be successful.

Semaphores created to manage resources should be created with an initial count value equal to the number of available resources.

configSUPPORT_DYNAMIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h, or simply left undefined, for this function to be available.

Example

The following shows the use of xSemaphoreCreateCounting().

```
void vATask( void * pvParameters )

{

    SemaphoreHandle_t xSemaphore;

    /* The semaphore cannot be used before it is created using a call to

    xSemaphoreCreateCounting(). The maximum value to which the semaphore can

    count in this example is set to 10, and the initial value assigned to

    the count is set to 0. */

    xSemaphore = xSemaphoreCreateCounting( 10, 0 );

    if( xSemaphore != NULL )

    {

        /* The semaphore was created successfully. The semaphore can now be used. */
```

```
    }

}
```

xSemaphoreCreateCountingStatic()

The following shows the xSemaphoreCreateCountingStatic() function prototype.

```
#include "FreeRTOS.h"

#include "semphr.h"

SemaphoreHandle_t xSemaphoreCreateCountingStatic( UBaseType_t uxMaxCount, UBaseType_t
 uxInitialCount, StaticSemaphore_t pxSempahoreBuffer );
```

Summary

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

Each counting semaphore requires a small amount of RAM that is used to hold the semaphore's state. If a counting semaphore is created using xSemaphoreCreateCounting(), then the required RAM is automatically allocated from the FreeRTOS heap. If a counting semaphore is created using xSemaphoreCreateCountingStatic(), then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

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 be used to hold the semaphore's state.

Return Values

NULL	The semaphore could not be created because pxSemaphoreBuffer was NULL.
Any other value	The semaphore was created successfully. The returned value is a handle by which the created semaphore can be referenced.

Notes

Direct-to-task notifications normally provide a lightweight and faster alternative to counting semaphores.

Counting semaphores are typically used for two things:

1. Counting events.

In this usage scenario, an event handler will 'give' the semaphore each time an event occurs, and a handler task will 'take' the semaphore each time it processes an event.

The semaphore's count value will be incremented each time it is 'given' and decremented each time it is 'taken'. The count value is therefore the difference between the number of events that have occurred and the number of events that have been processed.

Semaphores created to count events should be created with an initial count value of zero, because no events will have been counted before the semaphore was created.

1. Resource management.

In this usage scenario, the count value of the semaphore represents the number of available resources.

To obtain control of a resource, a task must first successfully 'take' the semaphore. The action of 'taking' the semaphore will decrement the semaphore's count value. When the count value reaches zero, no more resources are available, and attempts to 'take' the semaphore will fail.

When a task finishes with a resource, it must 'give' the semaphore. The action of 'giving' the semaphore will increment the semaphore's count value, indicating that a resource is available, and allowing future attempts to 'take' the semaphore to be successful.

Semaphores created to manage resources should be created with an initial count value equal to the number of available resources.

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

Example

The following shows the use of xSemaphoreCreateCountingStatic().

```
void vATask( void * pvParameters )

{

    SemaphoreHandle_t xSemaphoreHandle;

    StaticSemaphore_t xSemaphoreBuffer;

    /* Create a counting semaphore without using dynamic memory allocation.
    The maximum value to which the semaphore can count in this example case is
    set to 10, and the initial value assigned to the count is set to 0. */

    xSemaphoreHandle = xSemaphoreCreateCountingStatic( 10, 0, &xSemaphoreBuffer );

    /* The pxSemaphoreBuffer parameter was not NULL, so the semaphore will
```

```
    have been created and is now ready for use. */

}
```

xSemaphoreCreateMutex()

The following shows the xSemaphoreCreateMutex() function prototype

```
#include "FreeRTOS.h"

#include "semphr.h"

SemaphoreHandle_t xSemaphoreCreateMutex( void );
```

Summary

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

Each mutex type semaphore requires a small amount of RAM that is used to hold the semaphore's state. If a mutex is created using xSemaphoreCreateMutex() then the required RAM is automatically allocated from the FreeRTOS heap. If a mutex is created using xSemaphoreCreateMutexStatic() then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

Parameters

None

Return Values

NULL	Returned if the semaphore cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the semaphore data structures.
Any other value	The semaphore was created successfully. The returned value is a handle by which the created semaphore can be referenced.

Notes

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

Binary Semaphores - A binary semaphore used for synchronization does not need to be 'given' back after it has been successfully 'taken' (obtained). Task synchronization is implemented by

having one task or interrupt 'give' the semaphore, and another task 'take' the semaphore (see the xSemaphoreGiveFromISR() documentation).

Mutexes - The priority of a task that holds a mutex will be raised if another task of higher priority attempts to obtain the same mutex. The task that already holds the mutex is said to 'inherit' the priority of the task that is attempting to 'take' the same mutex. The inherited priority will be 'disinherited' when the mutex is returned (the task that inherited a higher priority while it held a mutex will return to its original priority when the mutex is returned).

A task that obtains a mutex that is used for mutual exclusion must always give the mutex back - otherwise no other task will ever be able to obtain the same mutex. An example of a mutex being used to implement mutual exclusion is provided in the xSemaphoreTake() section of this manual.

Mutexes and binary semaphores are both referenced using variables that have an SemaphoreHandle_t type, and can be used in any API function that takes a parameter of that type.

configSUPPORT_DYNAMIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h, or simply left undefined, for this function to be available.

Example

Example use of xSemaphoreCreateMutex()

```
SemaphoreHandle_t xSemaphore;

void vATask( void * pvParameters )

{

    /* Attempt to create a mutex type semaphore. */

    xSemaphore = xSemaphoreCreateMutex();

    if( xSemaphore == NULL )

    {

        /* There was insufficient heap memory available for the mutex to be created. */

    }

    else

    {

        /* The mutex can now be used. The handle of the created mutex will be stored in the
 xSemaphore variable. */

    }

}
```

xSemaphoreCreateMutexStatic()

The following shows the xSemaphoreCreateMutexStatic() function prototype.

```
#include "FreeRTOS.h"

#include "semphr.h"

SemaphoreHandle_t xSemaphoreCreateMutexStatic( StaticSemaphore_t *pxMutexBuffer );
```

Summary

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

Each mutex type semaphore requires a small amount of RAM that is used to hold the semaphore's state. If a mutex is created using xSemaphoreCreateMutex(), then the required RAM is allocated from the FreeRTOS heap automatically. If a mutex is created using xSemaphoreCreateMutexStatic(), then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

Parameters

pxMutexBuffer	Must point to a variable of type StaticSemaphore_t, which will be used to hold the mutex's state.

Return Values

NULL	The mutex could not be created because pxMutexBuffer was NULL.
Any other value	The mutex was created successfully. The returned value is a handle by which the created mutex can be referenced.

Notes

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

Binary semaphores A binary semaphore used for synchronization does not need to be 'given' back after it has been successfully 'taken' (obtained). Task synchronization is implemented by having one task or interrupt 'give' the semaphore, and another task 'take' the semaphore. (See xSemaphoreGiveFromISR().)

Mutexes The priority of a task that holds a mutex will be raised if another task of higher priority attempts to obtain the same mutex. The task that already holds the mutex is said to 'inherit' the priority of the task that is attempting to 'take' the same mutex. The inherited priority will be 'disinherited' when the mutex is returned (the task that inherited a higher priority while it held a mutex will return to its original priority when the mutex is returned).

A task that obtains a mutex that is used for mutual exclusion must always give the mutex back. Otherwise, no other task will ever be able to obtain the same mutex. An example of a mutex being used to implement mutual exclusion is provided in the xSemaphoreTake() section.

Mutexes and binary semaphores are both referenced using variables that have an SemaphoreHandle_t type, and can be used in any API function that takes a parameter of that type.

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

Example

The following shows the use of xSemaphoreCreateMutexStatic().

```
SemaphoreHandle_t xSemaphoreHandle;

StaticSemaphore_t xSemaphoreBuffer;

void vATask( void * pvParameters )

{

    /* Create a mutex without using any dynamic memory allocation. */

    xSemaphoreHandle = xSemaphoreCreateMutexStatic( &xSemaphoreBuffer );

    /* The pxMutexBuffer parameter was not NULL so the mutex will have been created and is
 now ready for use. */

}
```

xSemaphoreCreateRecursiveMutex()

The following shows the xSemaphoreCreateRecursiveMutex() function prototype.

```
#include "FreeRTOS.h"

#include "semphr.h"

SemaphoreHandle_t xSemaphoreCreateRecursiveMutex( void );
```

Summary

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

Each recursive mutex requires a small amount of RAM that is used to hold the mutex's state. If a recursive mutex is created using xSemaphoreCreateRecursiveMutex(), then the required RAM is allocated from the FreeRTOS heap automatically. If a recursive mutex is created using xSemaphoreCreateRecursiveMutexStatic(), then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

Parameters

None.

Return Values

NULL	Returned if the semaphore cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the mutex data structures.
Any other value	The mutex was created successfully. The returned value is a handle by which the created mutex can be referenced.

Notes

configUSE_RECURSIVE_MUTEXES must be set to 1 in FreeRTOSConfig.h for the xSemaphoreCreateRecursiveMutex() API function to be available.

A recursive mutex is 'taken' using the xSemaphoreTakeRecursive() function, and 'given' using the xSemaphoreGiveRecursive() function. The xSemaphoreTake() and xSemaphoreGive() functions must not be used with recursive mutexes.

Calls to xSemaphoreTakeRecursive() can be nested. Therefore, after a recursive mutex has been successfully 'taken' by a task, further calls to xSemaphoreTakeRecursive() made by the same task will also be successful. The same number of calls must be made to xSemaphoreGiveRecursive() as have previously been made to xSemaphoreTakeRecursive() before the mutex becomes available to any other task. For example, if a task successfully and recursively 'takes' the same mutex five times, then the mutex will not be available to any other task until the task that successfully obtained the mutex has also 'given' the mutex back exactly five times.

As with standard mutexes, a recursive mutex can only be held/obtained by a single task at any one time.

The priority of a task that holds a recursive mutex will be raised if another task of higher priority attempts to obtain the same mutex. The task that already holds the recursive mutex is said to 'inherit' the priority of the task that is attempting to 'take' the same mutex. The inherited priority will be 'disinherited' when the mutex is returned (the task that inherited a higher priority while it held a mutex will return to its original priority when the mutex is returned).

configSUPPORT_DYNAMIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h, or simply left undefined, for this function to be available.

Example

The following shows the use of xSemaphoreCreateRecursiveMutex().

```
void vATask( void * pvParameters )

{

    SemaphoreHandle_t xSemaphore;

    /* Recursive semaphores cannot be used before being explicitly created using a call to
 xSemaphoreCreateRecursiveMutex(). */

    xSemaphore = xSemaphoreCreateRecursiveMutex();

    if( xSemaphore != NULL )

    {

        /* The recursive mutex semaphore was created successfully and its handle will be
 stored in xSemaphore variable. The recursive mutex can now be used. */

    }

}
```

xSemaphoreCreateRecursiveMutexStatic()

The following shows the xSemaphoreCreateRecursiveMutexStatic() function prototype.

```
#include "FreeRTOS.h"

#include "semphr.h"

SemaphoreHandle_t xSemaphoreCreateRecursiveMutex( StaticSemaphore_t pxMutexBuffer );
```

Summary

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

Each recursive mutex requires a small amount of RAM that is used to hold the mutex's state. If a recursive mutex is created using xSemaphoreCreateRecursiveMutex(), then the required RAM is allocated from the FreeRTOS heap automatically. If a recursive mutex is created using xSemaphoreCreateRecursiveMutexStatic(), then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

Parameters

pxMutexBuffer	Must point to a variable of type StaticSemaphore_t, which will be used to hold the mutex's state.

Return Values

NULL	The semaphore could not be created because pxMutexBuffer was NULL.
Any other value	The mutex was created successfully. The returned value is a handle by which the created mutex can be referenced.

Notes

configUSE_RECURSIVE_MUTEXES must be set to 1 in FreeRTOSConfig.h for the xSemaphoreCreateRecursiveMutexStatic() API function to be available.

A recursive mutex is 'taken' using the xSemaphoreTakeRecursive() function, and 'given' using the xSemaphoreGiveRecursive() function. The xSemaphoreTake() and xSemaphoreGive() functions must not be used with recursive mutexes.

Calls to xSemaphoreTakeRecursive() can be nested. Therefore, once a recursive mutex has been successfully 'taken' by a task, further calls to xSemaphoreTakeRecursive() made by the same task will also be successful. The same number of calls must be made to xSemaphoreGiveRecursive() as have previously been made to xSemaphoreTakeRecursive() before the mutex becomes available to any other task. For example, if a task successfully and recursively 'takes' the same mutex five times, then the mutex will not be available to any other task until the task that successfully obtained the mutex has also 'given' the mutex back exactly five times.

As with standard mutexes, a recursive mutex can only be held/obtained by a single task at any one time.

The priority of a task that holds a recursive mutex will be raised if another task of higher priority attempts to obtain the same mutex. The task that already holds the recursive mutex is said to 'inherit' the priority of the task that is attempting to 'take' the same mutex. The inherited priority will be 'disinherited' when the mutex is returned (the task that inherited a higher priority while it held a mutex will return to its original priority when the mutex is returned).

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

Example

The following shows the use of xSemaphoreCreateRecursiveMutexStatic().

```
void vATask( void * pvParameters )

{

    SemaphoreHandle_t xSemaphoreHandle;

    StaticSemaphore_t xSemaphoreBuffer;

    /* Create a recursive mutex without using any dynamic memory allocation. */

    xSemaphoreHandle = xSemaphoreCreateRecursiveMutexStatic(&xSemaphoreBuffer );

    /* The pxMutexBuffer parameter was not NULL so the recursive mutex will have been
 created and is now ready for use. */

 }
```

vSemaphoreDelete()

The following shows the vSemaphoreDelete() function prototype.

```
#include "FreeRTOS.h"

#include "semphr.h"

void vSemaphoreDelete( SemaphoreHandle_t xSemaphore );
```

Summary

Deletes a semaphore that was previously created using a call to vSemaphoreCreateBinary(), xSemaphoreCreateCounting(), xSemaphoreCreateRecursiveMutex(), or xSemaphoreCreateMutex().

Parameters

xSemaphore	The handle of the semaphore being deleted.

Return Values

None.

Notes

Tasks can opt to block on a semaphore (with an optional timeout) if they attempt to obtain a semaphore that is not available. A semaphore must not be deleted if there are any tasks currently blocked on it.

uxSemaphoreGetCount()

The following shows the uxSemaphoreGetCount() function prototype.

```
#include "FreeRTOS.h"

#include "semphr.h"

UBaseType_t uxSemaphoreGetCount( SemaphoreHandle_t xSemaphore );
```

Summary

Returns the count of a semaphore.

Binary semaphores can only have a count of zero or one. Counting semaphores can have a count between zero and the maximum count specified when the counting semaphore was created.

Parameters

xSemaphore	The handle of the semaphore being queried.

Return Values

The count of the semaphore referenced by the handle passed in the xSemaphore parameter.

xSemaphoreGetMutexHolder()

The following shows the xSemaphoreGetMutexHolder() function prototype.

```
#include "FreeRTOS.h"

#include "semphr.h"

TaskHandle_t xSemaphoreGetMutexHolder( SemaphoreHandle_t xMutex );
```

Summary

Returns the handle of the task that holds the mutex specified by the function parameter, if any.

Parameters

xMutex	The handle of the mutex being queried.

Return Values

NULL	Either: • The semaphore specified by the xMutex parameter is not a mutex type semaphore, or • The semaphore is available, and not held by any task.
Any other value	The handle of the task that holds the semaphore specified by the xMutex parameter.

Notes

xSemaphoreGetMutexHolder() can be used to determine if the calling task is the mutex holder, but cannot be used reliably if the mutex is held by any task other than the calling task. This is because the mutex holder might change between the calling task calling the function and the calling task testing the function's return value.

configUSE_MUTEXES and *INCLUDE_xSemaphoreGetMutexHolder* must both be set to 1 in FreeRTOSConfig.h for xSemaphoreGetMutexHolder() to be available.

xSemaphoreGive()

```
#include "FreeRTOS.h"

#include "semphr.h"

BaseType_t xSemaphoreGive( SemaphoreHandle_t xSemaphore );
```

Listing xSemaphoreGive() function prototype

Summary

'Gives' (or releases) a semaphore that has previously been created using a call to vSemaphoreCreateBinary(), xSemaphoreCreateCounting(), or xSemaphoreCreateMutex() and has also been successfully 'taken'.

Parameters

xSemaphore	The semaphore being 'given'. A semaphore is referenced by a variable of type SemaphoreHandle_t and must be explicitly created before being used.

Return Values

pdPASS	The semaphore 'give' operation was successful.
pdFAIL	The semaphore 'give' operation was not successful because the task calling xSemaphoreGive() is not the semaphore holder. A task must successfully 'take' a semaphore before it can successfully 'give' it back.

Notes

None.

Example

Example use of xSemaphoreGive()

```
SemaphoreHandle_t xSemaphore = NULL;
```

```
void vATask( void * pvParameters )

{

    /* A semaphore is going to be used to guard a shared resource. In this case, a mutex
 type semaphore is created because it includes priority inheritance functionality. */

    xSemaphore = xSemaphoreCreateMutex();

    for( ;; )

    {

        if( xSemaphore != NULL )

        {

            if( xSemaphoreGive( xSemaphore ) != pdTRUE )

            {

                    /* This call should fail because the semaphore has not yet been
 'taken'. */

            }

            /* Obtain the semaphore. Don't block if the semaphore is not immediately
 available (the specified block time is zero). */

            if( xSemaphoreTake( xSemaphore, 0 ) == pdPASS )

            {

                /* The semaphore was 'taken' successfully, so the resource it is guarding
 can be accessed safely. */

                /* ... */

                /* Access to the resource the semaphore is guarding is complete, so the
 semaphore must be 'given' back. */

                if( xSemaphoreGive( xSemaphore ) != pdPASS )

                {

                        /* This call should not fail because the calling task has already
 successfully 'taken' the semaphore. */

                }

            }

        }

        else

        {

                /* The semaphore was not created successfully because there is not enough
 FreeRTOS heap remaining for the semaphore data structures to be allocated. */

        }

    }
```

```
}
```

xSemaphoreGiveFromISR()

The following shows the xSemaphoreGiveFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "semphr.h"

BaseType_t xSemaphoreGiveFromISR( SemaphoreHandle_t xSemaphore,

BaseType_t *pxHigherPriorityTaskWoken );
```

Summary

A version of xSemaphoreGive() that can be used in an ISR. Unlike xSemaphoreGive(), xSemaphoreGiveFromISR() does not permit a block time to be specified.

Parameters

Return Values

pdTRUE	The call to xSemaphoreGiveFromISR() was successful.
errQUEUE_FULL	If a semaphore is already available, it cannot be given, and xSemaphoreGiveFromISR() will return errQUEUE_FULL.

Notes

Calling xSemaphoreGiveFromISR() within an interrupt service routine can potentially cause a task that was blocked waiting to take the semaphore to leave the Blocked state. A context switch should be performed if such an unblocked task has a priority higher than or equal to the currently executing task (the task that was interrupted). The context switch will ensure that the interrupt returns directly to the highest priority Ready state task.

Unlike the xSemaphoreGive() API function, xSemaphoreGiveFromISR() will not itself perform a context switch. It will instead just indicate whether a context switch is required.

xSemaphoreGiveFromISR() must not be called before the scheduler is started. Therefore, an interrupt that calls xSemaphoreGiveFromISR() must not be allowed to execute before the scheduler is started.

Example

The following shows the use of xSemaphoreGiveFromISR().

#define LONG_TIME 0xffff .. code:

```
#define TICKS_TO_WAIT 10

SemaphoreHandle_t xSemaphore = NULL;

/* Define a task that performs an action each time an interrupt occurs. The interrupt
 processing is deferred to this task. The task is synchronized with the interrupt using a
 semaphore. */

void vATask( void * pvParameters )

{

        /* It is assumed the semaphore has already been created outside of this task. */

        for( ;; )

        {

                /* Wait for the next event. */

                if( xSemaphoreTake( xSemaphore, portMAX_DELAY ) == pdTRUE )

                {

                        /* The event has occurred. Process it here. */

                            ...

                        /* Processing is complete. Return to wait for the next event. */

                }

          }

  }

  /* An ISR that defers its processing to a task by using a semaphore to indicate when
 events that require processing have occurred. */

  void vISR( void * pvParameters )

  {

        BaseType_t xHigherPriorityTaskWoken = pdFALSE;

        /* The event has occurred. Use the semaphore to unblock the task so the task can
 process the event. */

        xSemaphoreGiveFromISR( xSemaphore, &xHigherPriorityTaskWoken );

        /* Clear the interrupt here. */

        /* Now that the task has been unblocked, a context switch should be performed
 if xHigherPriorityTaskWoken is equal to pdTRUE. NOTE: The syntax required to perform a
 context switch from an ISR varies from port to port and from compiler to compiler. Check
 the web documentation and examples for the port being used to find the syntax required for
 your application. */

        portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

   }
```

xSemaphoreGiveRecursive()

The following shows the xSemaphoreGiveRecursive() function prototype.

```
#include "FreeRTOS.h"

#include "semphr.h"

BaseType_t xSemaphoreGiveRecursive( SemaphoreHandle_t xMutex );
```

Summary

'Gives' (or releases) a recursive mutex type semaphore that has previously been created by using xSemaphoreCreateRecursiveMutex().

Parameters

xMutex	The semaphore being 'given'. A semaphore is referenced by a variable of type SemaphoreHandle_t and must be explicitly created before being used.

Return Values

pdPASS	The call to xSemaphoreGiveRecursive() was successful.
pdFAIL	The call to xSemaphoreGiveRecursive() failed because the calling task is not the mutex holder.

Notes

A recursive mutex is 'taken' using the xSemaphoreTakeRecursive() function and 'given' using the xSemaphoreGiveRecursive() function. The xSemaphoreTake() and xSemaphoreGive() functions must not be used with recursive mutexes.

Calls to xSemaphoreTakeRecursive() can be nested. Therefore, after a recursive mutex has been successfully 'taken' by a task, further calls to xSemaphoreTakeRecursive() made by the same task will also be successful. The same number of calls must be made to xSemaphoreGiveRecursive() as have previously been made to xSemaphoreTakeRecursive() before the mutex becomes available to any other task. For example, if a task successfully and recursively 'takes' the same mutex five times, then the mutex will not be available to any other task until the task that successfully obtained the mutex has also 'given' the mutex back exactly five times.

xSemaphoreGiveRecursive() must only be called from an executing task and therefore must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

xSemaphoreGiveRecursive() must not be called from within a critical section or while the scheduler is suspended.

Example

The following shows the use of xSemaphoreGiveRecursive().

```
/* A task that creates a recursive mutex. */

void vATask( void * pvParameters )

{

    /* Recursive mutexes cannot be used before being explicitly created using a call to
 xSemaphoreCreateRecursiveMutex(). */

    xMutex = xSemaphoreCreateRecursiveMutex();

    /* Rest of task code goes here. */

    for( ;; )

    {

    }

}

/* A function (called by a task) that uses the mutex. */

void vAFunction( void )

{

    /* ... Do other things. */

    if( xMutex != NULL )

    {

        /* See if the mutex can be obtained. If the mutex is not available, wait 10 ticks
 to see if it becomes free. */

        if( xSemaphoreTakeRecursive( xMutex, 10 ) == pdTRUE )

        {

            /* The mutex was successfully 'taken'. */

            ...

            /* 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 because that would serve no purpose.
            Instead, the calls are likely to be buried inside a more complex call
            structure (for example, in a TCP/IP stack).*/

            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. They would
            be buried in a more complex call structure instead. This is just for
            illustrative purposes. */

            xSemaphoreGiveRecursive( xMutex );

            xSemaphoreGiveRecursive( xMutex );

            xSemaphoreGiveRecursive( xMutex );

            /* Now the mutex can be taken by other tasks. */

        }

        else

        {

            /* The mutex was not successfully 'taken'. */

        }

    }

}
```

xSemaphoreTake()

The following shows the xSemaphoreTake() function prototype.

```
#include "FreeRTOS.h"

#include "semphr.h"

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

Summary

'Takes' (obtains) a semaphore that has previously been created by using a call to vSemaphoreCreateBinary(), xSemaphoreCreateCounting(), or xSemaphoreCreateMutex().

Parameters

xSemaphore	The semaphore being 'taken'. A semaphore is referenced by a variable of type SemaphoreHandle_t and must be explicitly created before being used.
xTicksToWait	The maximum amount of time the task should remain in the Blocked state to wait for the semaphore to become available, if the semaphore is not available immediately. If xTicksToWait is zero, then xSemaphoreTake() will return immediately if the semaphore is not available. The block time is specified in tick periods, so the absolute time it represents depends on the tick frequency. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds to 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.

Return Values

pdPASS	Returned only if the call to xSemaphoreTake() was successful in obtaining the semaphore.

	If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state to wait for the semaphore if it was not immediately available, but the semaphore became available before the block time expired.
pdFAIL	Returned if the call to xSemaphoreTake() did not successfully obtain the semaphore. If a block time was specified (xTicksToWait was not zero), then the calling task will have been placed into the Blocked state to wait for the semaphore to become available, but the block time expired before this happened.

Notes

xSemaphoreTake() must only be called from an executing task. Therefore, it must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

xSemaphoreTake() must not be called from within a critical section or while the scheduler is suspended.

Example

The following shows the use of xSemaphoreTake().

```
SemaphoreHandle_t xSemaphore = NULL;

/* A task that creates a mutex type semaphore. */

void vATask( void * pvParameters )

{

    /* A semaphore is going to be used to guard a shared resource. In this case, a mutex
 type semaphore is created because it includes priority inheritance functionality. */

    xSemaphore = xSemaphoreCreateMutex();

    /* The rest of the task code goes here. */

    for( ;; )

    {

        /* ... */

    }

}

/* A task that uses the mutex. */

void vAnotherTask( void * pvParameters )

{
```

```
   for( ;; )
   {
       /* ... Do other things. */
       if( xSemaphore != NULL )
       {
           /* See if the mutex can be obtained. If the mutex is not available, wait 10
ticks to see if it becomes free. */
           if( xSemaphoreTake( xSemaphore, 10 ) == pdTRUE )
           {
                   /* The mutex was successfully obtained so the shared resource can be
accessed safely. */
                   /* ... */
                   /* Access to the shared resource is complete, so the mutex is returned.
*/
                   xSemaphoreGive( xSemaphore );
           }
           else
           {
                   /* The mutex could not be obtained even after waiting 10 ticks, so the
shared resource cannot be accessed. */
           }
       }
   }
}
```

xSemaphoreTakeFromISR()

The following shows the xSemaphoreTakeFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "queue.h"

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

Summary

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

Parameters

Return Values

pdPASS	The semaphore was successfully taken (acquired).
pdFAIL	The semaphore was not successfully taken because it was not available.

xSemaphoreTakeRecursive()

The following shows the xSemaphoreTakeRecursive() function prototype.

```
#include "FreeRTOS.h"

#include "semphr.h"

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

Summary

'Takes' (obtains) a recursive mutex type semaphore that has previously been created by using xSemaphoreCreateRecursiveMutex().

Parameters

xMutex	The semaphore being 'taken'. A semaphore is referenced by a variable of type SemaphoreHandle_t and must be explicitly created before being used.
xTicksToWait	The maximum amount of time the task should remain in the Blocked state to wait for the semaphore to become available, if the semaphore is not available immediately. If xTicksToWait is zero, then xSemaphoreTakeRecursive() will return immediately if the semaphore is not available. The block time is specified in tick periods, so the absolute time it represents depends on the tick frequency. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds to 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.

Return Values

pdPASS	Returned only if the call to xSemaphoreTakeRecursive() was successful in obtaining the semaphore.

	If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state to wait for the semaphore if it was not immediately available, but the semaphore became available before the block time expired.
pdFAIL	Returned if the call to xSemaphoreTakeRecursive() did not successfully obtain the semaphore. If a block time was specified (xTicksToWait was not zero), then the calling task will have been placed into the Blocked state to wait for the semaphore to become available, but the block time expired before this happened.

Notes

A recursive mutex is 'taken' by using the xSemaphoreTakeRecursive() function and 'given' by using the xSemaphoreGiveRecursive() function. The xSemaphoreTake() and xSemaphoreGive() functions must not be used with recursive mutexes.

Calls to xSemaphoreTakeRecursive() can be nested. Therefore, after a recursive mutex has been successfully 'taken' by a task, further calls to xSemaphoreTakeRecursive() made by the same task will also be successful. The same number of calls must be made to xSemaphoreGiveRecursive() as have previously been made to xSemaphoreTakeRecursive() before the mutex becomes available to any other task. For example, if a task successfully and recursively 'takes' the same mutex five times, then the mutex will not be available to any other task until the task that successfully obtained the mutex has also 'given' the mutex back exactly five times.

xSemaphoreTakeRecursive() must only be called from an executing task. Therefore, it must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

xSemaphoreTakeRecursive() must not be called from within a critical section or while the scheduler is suspended.

Example

The following shows the use of xSemaphoreTakeRecursive().

```
/* A task that creates a recursive mutex. */

void vATask( void * pvParameters )

{

    /* Recursive mutexes cannot be used before being explicitly created using a call to
 xSemaphoreCreateRecursiveMutex(). */

    xMutex = xSemaphoreCreateRecursiveMutex();

    /* Rest of task code goes here. */

    for( ;; )

    {
```

```
    }

}

/* A function (called by a task) that uses the mutex. */

void vAFunction( void )

{

    /* ... Do other things. */

    if( xMutex != NULL )

    {

        /* See if the mutex can be obtained. If the mutex is not available, wait 10 ticks
 to see if it becomes free. */

        if( xSemaphoreTakeRecursive( xMutex, 10 ) == pdTRUE )

        {

            /* The mutex was successfully 'taken'. */

            ...

            /* 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 because that would serve no purpose.
 Instead, the calls are likely to be buried inside a more complex call
            structure (for example, in a TCP/IP stack).*/

            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. They would be
 buried in a more complex call structure instead. This is just for illustrative purposes.
 */

            xSemaphoreGiveRecursive( xMutex );

            xSemaphoreGiveRecursive( xMutex );

            xSemaphoreGiveRecursive( xMutex );

            /* Now the mutex can be taken by other tasks. */

        }

        else

        {

            /* The mutex was not successfully 'taken'. */

        }

   }

}
```

Software Timer API

xTimerChangePeriod()

The following shows the xTimerChangePeriod() function prototype.

```
#include "FreeRTOS.h"

#include "timers.h"

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

Summary

Changes the period of a timer. xTimerChangePeriodFormISR() is an equivalent function that can be called from an interrupt service routine.

If xTimerChangePeriod() is used to change the period of a timer that is already running, then the timer will use the new period value to recalculate its expiry time. The recalculated expiry time will then be relative to when xTimerChangePeriod() was called, and not relative to when the timer was originally started.

If xTimerChangePeriod() is used to change the period of a timer that is not already running, then the timer will use the new period value to calculate an expiry time, and the timer will start running.

Parameters

xTimer	The timer to which the new period is being assigned.
xNewPeriod	The new period for the timer referenced by the xTimer parameter. Timer periods are specified in multiples of tick periods. The pdMS_TO_TICKS() macro can be used to convert a time in milliseconds to ticks. For example, if the timer must expire after 100 ticks, then xNewPeriod can be set directly to 100. Alternatively, if the timer must expire after 500 ms, then xNewPeriod can be set to pdMS_TO_TICKS(500), provided configTICK_RATE_HZ is less than or equal to 1000.
xTicksToWait	Timer functionality is not provided by the core FreeRTOS code, but by a timer service (or daemon) task. The FreeRTOS timer API sends commands to the timer service task on a queue called the timer command queue. xTicksToWait specifies the maximum amount of time the task

	should remain in the Blocked state to wait for space to become available on the timer command queue, should the queue already be full. The block time is specified in tick periods, so the absolute time it represents depends on the tick frequency. As with the xNewPeriod parameter, the pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds to 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. xTicksToWait is ignored if xTimerChangePeriod() is called before the scheduler is started.

Return Values

pdPASS	The change period command was successfully sent to the timer command queue. If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state to wait for space to become available on the timer command queue before the function returned, but data was successfully written to the queue before the block time expired. When the command is actually processed will depend on the priority of the timer service task relative to other tasks in the system, although the timer's expiry time is relative to when xTimerChangePeriod() is actually called. The priority of the timer service task is set by the configTIMER_TASK_PRIORITY configuration constant.
pdFAIL	The change period command was not sent to the timer command queue because the queue was already full. If a block time was specified (xTicksToWait was not zero), then the calling task will have been placed into the Blocked state to wait for the timer service task to make room in the queue, but the specified block time expired before that happened.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerChangePeriod() to be available.

Example

The following shows the use of xTimerChangePeriod().

..code:

```
/* This function assumes xTimer has already been created. If the timer
referenced by xTimer is already active when it is called, then the timer
is deleted. If the timer referenced by xTimer is not active when it is
called, then the period of the timer is set to 500 ms, and the timer is
started. */

void vAFunction( TimerHandle_t xTimer )

{

    if( xTimerIsTimerActive( xTimer ) != pdFALSE )

    {

        /* xTimer is already active - delete it. */

        xTimerDelete( xTimer );

    }

    else

    {

        /* xTimer is not active, change its period to 500 ms. This will also cause the
 timer to start. Block for a maximum of 100 ticks if the change period command cannot
 immediately be sent to the timer command queue. */

        if( xTimerChangePeriod( xTimer, pdMS_TO_TICKS( 500 ), 100 ) == pdPASS)

        {

            /* The command was successfully sent. */

        }

        else

        {

                /* The command could not be sent, even after waiting for 100 ticks to pass.
 Take appropriate action here. */

        }

    }

}
```

xTimerChangePeriodFromISR()

The following shows the xTimerChangePeriodFromISR() function prototype. .. code:

```
#include "FreeRTOS.h"

#include "timers.h"

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

Summary

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

Parameters

xTimer	The timer to which the new period is being assigned.
xNewPeriod	The new period for the timer referenced by the xTimer parameter. Timer periods are specified in multiples of tick periods. The pdMS_TO_TICKS() macro can be used to convert a time in milliseconds to a time in ticks. For example, if the timer must expire after 100 ticks, then xNewPeriod can be set directly to 100. Alternatively, if the timer must expire after 500ms, then xNewPeriod can be set to pdMS_TO_TICKS(500), provided configTICK_RATE_HZ is less than or equal to 1000.
pxHigherPriorityTaskWoken	xTimerChangePeriodFromISR() writes a command to the timer command queue. If writing to the timer command queue causes the timer service task to leave the Blocked state, and the timer service task has a priority equal to or greater than the currently executing task (the task that was interrupted), then <problematic>*</problematic> pxHigherPriorityTaskWoken will be set to pdTRUE internally within the xTimerChangePeriodFromISR() function. If xTimerChangePeriodFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt exits.

Return Values

pdPASS	The change period command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service task relative to other tasks in the system, although the timer's expiry time is relative to when xTimerChangePeriodFromISR() is actually called. The priority of the timer service task is set by the configTIMER_TASK_PRIORITY configuration constant.
pdFAIL	The change period command was not sent to the timer command queue because the queue was already full.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerChangePeriodFromISR() to be available.

Example

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

/* The interrupt service routine that changes the period of xTimer. */

void vAnExampleInterruptServiceRoutine( void )

{

    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* The interrupt has occurred. Change the period of xTimer to 500 ms.
    xHigherPriorityTaskWoken was set to pdFALSE where it was defined (within this
    function). Because this is an interrupt service routine, only FreeRTOS API
    functions that end in "FromISR" can be used. */

    if( xTimerChangePeriodFromISR( xTimer, &xHigherPriorityTaskWoken ) != pdPASS )

    {

        /* The command to change the timer's period was not executed successfully. Take
 appropriate action here. */

    }

    /* If xHigherPriorityTaskWoken equals pdTRUE, then a context switch should be
 performed. The syntax required to perform a context switch from inside an ISR varies from
 port to port and from compiler to compiler. Inspect the demos for the port you are using
 to find the syntax required. */
```

```
    if( xHigherPriorityTaskWoken != pdFALSE )

    {

        /* Call the interrupt safe yield function here (function depends on the FreeRTOS
 port being used). */

    }

}
```

xTimerCreate()

The following shows the xTimerCreate() function prototype.

```
#include "FreeRTOS.h"

#include "timers.h"

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

Summary

Creates a new software timer and returns a handle by which the created software timer can be referenced.

Each software timer requires a small amount of RAM that is used to hold the timer's state. If a software timer is created using xTimerCreate(), then this RAM is allocated from the FreeRTOS heap automatically. If a software timer is created using xTimerCreateStatic(), then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

Creating a timer does not start the timer running. The xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod(), and xTimerChangePeriodFromISR() API functions can all be used to start the timer running.

Parameters

pcTimerName	A plain text name that is assigned to the timer only to assist in debugging.
xTimerPeriod	The timer period. Timer periods are specified in multiples of tick periods. The pdMS_TO_TICKS() macro can be used to convert a time in milliseconds to a time in ticks. For example, if the timer must expire after 100 ticks, then xNewPeriod can be set directly to 100. Alternatively, if the timer must expire after 500ms, then xNewPeriod can be set to pdMS_TO_TICKS(500), provided configTICK_RATE_HZ is less than or equal to 1000.
uxAutoReload	Set to pdTRUE to create an autoreload timer. Set to pdFALSE to create a one-shot timer. After it's started, an autoreload timer will expire repeatedly with a frequency set by the xTimerPeriod parameter.

	After it's started, a one-shot timer will expire only once. A one-shot timer can be manually restarted after it has expired.
pvTimerID	An identifier that is assigned to the timer being created. The identifier can be updated by using the vTimerSetTimerID() API function. If the same callback function is assigned to multiple timers, then the timer identifier can be inspected inside the callback function to determine which timer actually expired. In addition, the timer identifier can be used to store a value in between calls to the timer's callback function.
pxCallbackFunction	The function to call when the timer expires. Callback functions must have the prototype defined by the TimerCallbackFunction_t typedef. void vCallbackFunctionExample(TimerHandle_t xTimer);

Return Values

NULL	The software timer could not be created because there was insufficient FreeRTOS heap memory available to successfully allocate the timer data structures.
Any other value	The software timer was created successfully and the returned value is the handle by which the created software timer can be referenced.

Notes

configUSE_TIMERS and configSUPPORT_DYNAMIC_ALLOCATION must both be set to 1 in FreeRTOSConfig.h for xTimerCreate() to be available. configSUPPORT_DYNAMIC_ALLOCATION will default to 1 if it is left undefined.

Example

```
/* Define a callback function that will be used by multiple timer
instances. The callback function does nothing but count the number of
times the associated timer expires and stop the timer after the timer
has expired 10 times. The count is saved as the ID of the timer. */

void vTimerCallback( TimerHandle_t xTimer )

{
```

```
    const uint32_t ulMaxExpiryCountBeforeStopping = 10;

    uint32_t ulCount;

    /* The number of times this timer has expired is saved as the timer's ID. Obtain the
 count. */

    ulCount = ( uint32_t ) pvTimerGetTimerID( xTimer );

    /* Increment the count, and then test to see if the timer has expired
 ulMaxExpiryCountBeforeStopping yet. */

    ulCount++;

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

    if( ulCount >= xMaxExpiryCountBeforeStopping )

    {

            /* Do not use a block time if calling a timer API function from a timer
 callback function. Doing so can cause a deadlock! */

            xTimerStop( pxTimer, 0 );

    }

    else

    {

        /* Store the incremented count back into the timer's ID field so it can be read
 back again the next time this software timer expires. */

        vTimerSetTimerID( xTimer, ( void * ) ulCount );

    }

}
```

Example use of xTimerCreate()

```
#define NUM_TIMERS 5

/* An array to hold handles to the created timers. */

TimerHandle_t xTimers[ NUM_TIMERS ];

void main( void )

{

    long x;

     /* Create and then start some timers. Starting the timers before the RTOS scheduler
 has been started means the timers will start running immediately when the RTOS scheduler
 starts. */

    for( x = 0; x < NUM_TIMERS; x++ )

    {

        xTimers[ x ] = xTimerCreate( /* Just a text name. Not used by the RTOS kernel. */

                                     "Timer",
```

```
                                    /* The timer period, in ticks. Must be greater than 0.
*/
                                    ( 100 * x ) + 100,
                                    /* The timers will auto-reload themselves when they
expire. */
                                    pdTRUE,
                                    /* The ID is used to store a count of the number of
times the timer has expired, which is initialized to 0. */
                                    ( void * ) 0,
                                    /* Each timer calls the same callback when it expires.
*/
                                    vTimerCallback );
            if( xTimers[ x ] == NULL )
            {
                    /* The timer was not created. */
            }
            else
            {
                    /* Start the timer. No block time is specified. Even if one were
specified, it would be ignored because the RTOS scheduler has not yet been started. */
                    if( xTimerStart( xTimers[ x ], 0 ) != pdPASS )
                    {
                        /* The timer could not be set into the Active state. */
                    }
            }
    }
    /* ...
            Create tasks here.
    ... */
    /* Starting the RTOS scheduler will start the timers running because they have already
been set into the Active state. */
    vTaskStartScheduler();
    /* Should not reach here. */
    for( ;; );
}
```

xTimerCreateStatic()

The following shows the xTimerCreateStatic() function prototype.

```
#include "FreeRTOS.h"

#include "timers.h"

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

Summary

Creates a software timer and returns a handle by which the created software timer can be referenced.

Each software timer requires a small amount of RAM that is used to hold the timer's state. If a software timer is created by using xTimerCreate(), then this RAM is allocated from the FreeRTOS heap automatically. If a software timer is created using xTimerCreateStatic(), then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

Creating a timer does not start the timer running. The xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod(), and xTimerChangePeriodFromISR() API functions can all be used to start the timer running.

Parameters

pcTimerName	A plain text name that is assigned to the timer only to assist in debugging.
xTimerPeriod	The timer period. Timer periods are specified in multiples of tick periods. The pdMS_TO_TICKS() macro can be used to convert a time in milliseconds to a time in ticks. For example, if the timer must expire after 100 ticks, then xNewPeriod can be set directly to 100. Alternatively, if the timer must expire after 500ms, then xNewPeriod can be set to pdMS_TO_TICKS(500), provided configTICK_RATE_HZ is less than or equal to 1000.
uxAutoReload	Set to pdTRUE to create an autoreload timer. Set to pdFALSE to create a one-shot timer. After it's started, an autoreload timer will expire repeatedly with a frequency set by the xTimerPeriod parameter. After it's started, a one-shot timer will expire only once. A one-shot timer can be manually restarted after it has expired.

pvTimerID	An identifier that is assigned to the timer being created. The identifier can be updated later by using the vTimerSetTimerID() API function. If the same callback function is assigned to multiple timers, then the timer identifier can be inspected inside the callback function to determine which timer actually expired. In addition, the timer identifier can be used to store a value in between calls to the timer's callback function.
pxCallbackFunction	The function to call when the timer expires. Callback functions must have the prototype defined by the TimerCallbackFunction_t typedef. void vCallbackFunctionExample(TimerHandle_t xTimer);
pxTimerBuffer	Must point to a variable of type StaticTimer_t, which is then used to hold the timer's state.

Return Values

NULL	The software timer could not be created because pxTimerBuffer was NULL.
Any other value	The software timer was created successfully and the returned value is the handle by which the created software timer can be referenced.

Notes

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

Example

The following shows the definition of the callback function used in the calls to xTimerCreate().

```
/* Define a callback function that will be used by multiple timer instances. The callback
 function does nothing but count the number of times the associated timer expires and stops
 the timer after it has expired 10 times. The count is saved as the ID of the timer. */

void vTimerCallback( TimerHandle_t xTimer )

{

    const uint32_t ulMaxExpiryCountBeforeStopping = 10;

    uint32_t ulCount;
```

```
    /* The number of times this timer has expired is saved as the timer's ID. Obtain the
 count. */

    ulCount = ( uint32_t ) pvTimerGetTimerID( xTimer );

    /* Increment the count, and then test to see if the timer has expired

    ulMaxExpiryCountBeforeStopping yet. */

    ulCount++;

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

    if( ulCount >= xMaxExpiryCountBeforeStopping )

    {

       /* Do not use a block time if calling a timer API function from a timer callback
 function. Doing so can cause a deadlock! */

       xTimerStop( pxTimer, 0 );

    }

    else

    {

      /* Store the incremented count back into the timer's ID field so it can be read back
 again the next time this software timer expires. */

      vTimerSetTimerID( xTimer, ( void * ) ulCount );

    }

}
```

The following shows the use of xTimerCreateStatic().

```
#define NUM_TIMERS 5

/* An array to hold handles to the created timers. */

TimerHandle_t xTimers[ NUM_TIMERS ];

/* An array of StaticTimer_t structures, which are used to store the state of each created
 timer. */

StaticTimer_t xTimerBuffers[ NUM_TIMERS ];

void main( void )

{

    long x;

    /* Create and then start some timers. Starting the timers before the RTOS scheduler
 has been started means the timers will start running immediately when the RTOS scheduler
 starts. */

    for( x = 0; x < NUM_TIMERS; x++ )

    {
```

```
       xTimers[ x ] = xTimerCreateStatic( /* Just a text name. Not used by the RTOS
kernel. */
                                          "Timer",
                                          /* The timer period, in ticks. Must be greater
than 0. */
                                          ( 100 * x ) + 100,
                                          /* The timers will auto-reload themselves when
they expire. */
                                          pdTRUE,
                                          /* The ID is used to store a count of the number
of times the timer has expired, which is initialized to 0. */
                                          ( void * ) 0,
                                          /* Each timer calls the same callback when it
expires. */
                                          vTimerCallback,
                                          /* Pass in the address of a StaticTimer_t
variable, which will hold the data associated with the timer being created. */
                                          &( xTimerBuffers[ x ] ); );

         if( xTimers[ x ] == NULL )

         {

             /* The timer was not created. */

         }

         else

         {

                 /* Start the timer. No block time is specified. Even if one were, it
would be ignored because the RTOS scheduler has not yet been started. */

                 if( xTimerStart( xTimers[ x ], 0 ) != pdPASS )

                 {

                       /* The timer could not be set into the Active state. */

                 }

          }

    }

    /* ...

       Create tasks here.

    ... */

    /* Starting the RTOS scheduler will start the timers running as they have already been
set into the Active state. */
```

```
    vTaskStartScheduler();

    /* Should not reach here. */

    for( ;; );

}
```

xTimerDelete()

The following shows the xTimerDelete() macro prototype.

```
#include "FreeRTOS.h"

#include "timers.h"

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

Summary

Deletes a timer. The timer must first have been created by using the xTimerCreate() API function.

Parameters

xTimer	The handle of the timer being deleted.
xTicksToWait	Timer functionality is not provided by the core FreeRTOS code, but by a timer service (or daemon) task. The FreeRTOS timer API sends commands to the timer service task on a queue called the timer command queue. xTicksToWait specifies the maximum amount of time the task should remain in the Blocked state to wait for space to become available on the timer command queue, should the queue already be full. The block time is specified in tick periods, so the absolute time it represents depends on the tick frequency. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds to 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. xTicksToWait is ignored if xTimerDelete() is called before the scheduler is started.

Return Values

pdPASS	The delete command was successfully sent to the timer command queue.

	If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state to wait for space to become available on the timer command queue before the function returned, but data was successfully written to the queue before the block time expired. When the command is actually processed will depend on the priority of the timer service task relative to other tasks in the system. The priority of the timer service task is set by the configTIMER_TASK_PRIORITY configuration constant.
pdFAIL	The delete command was not sent to the timer command queue because the queue was already full. If a block time was specified (xTicksToWait was not zero), then the calling task will have been placed into the Blocked state to wait for the timer service task to make room in the queue, but the specified block time expired before that happened.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerDelete() to be available.

Example

See the example provided for the xTimerChangePeriod() API function.

xTimerGetExpiryTime()

The following shows the xTimerGetExpiryTime() function prototype.

```
#include "FreeRTOS.h"

#include "timers.h"

TickType_t xTimerGetExpiryTime( TimerHandle_t xTimer );
```

Summary

Returns the time at which a software timer will expire, which is the time the software timer's callback function will execute.

Parameters

xTimer	The handle of the timer being queried.

Return Values

If the timer referenced by xTimer is active, then the time at which the timer's callback function will next execute is returned. The time is specified in RTOS ticks.

The return value is undefined if the timer referenced by xTimer is not active. The xTimerIsTimerActive() API function can be used to determine if a timer is active.

Notes

If the value returned by xTimerGetExpiryTime() is less than the current tick count, then the timer will not expire until after the tick count has overflowed and wrapped back to 0. Overflows are handled in the RTOS implementation, so a timer's callback function will execute at the correct time whether it is before or after the tick count overflows.

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerGetExpiryTime() to be available.

Example

The following shows the use of xTimerGetExpiryTime().

..code:

```
static void vAFunction( TimerHandle_t xTimer )
```

```
{

    TickType_t xRemainingTime;

    /* Calculate the time that remains before the timer referenced by xTimer expires and
 executes its callback function. TickType_t is an unsigned type, so the subtraction will
 result in the correct answer even if the timer will not expire until after the tick count
 has overflowed. */

    xRemainingTime = xTimerGetExpiryTime( xTimer ) - xTaskGetTickCount();

}
```

pcTimerGetName()

The following shows the pcTimerGetName() function prototype. .. code:

```
#include "FreeRTOS.h"

#include "timers.h"

const char * pcTimerGetName( TimerHandle_t xTimer );
```

Summary

Returns the human-readable text name assigned to the timer when the timer was created. For more information, see the xTimerCreate() API function.

Parameters

xTimer	The timer being queried.

Return Values

Timer names are standard NULL-terminated C strings. The value returned is a pointer to the subject timer's name.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for pcTimerGetName() to be available.

xTimerGetPeriod()

The following shows the xTimerGetPeriod() function prototype.

```
#include "FreeRTOS.h"

#include "timers.h"

TickType_t xTimerGetPeriod( TimerHandle_t xTimer );
```

Summary

Returns the period of a software timer. The period is specified in RTOS ticks.

The period of a software timer is initially specified by the xTimerPeriod parameter of the call to xTimerCreate() used to create the timer. It can subsequently be changed by using the xTimerChangePeriod() and xTimerChangePeriodFromISR() API functions.

Parameters

xTimer	The handle of the timer being queried.

Return Values

The period of the timer, specified in ticks.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerGetPeriod() to be available.

Example

```
/* A callback function assigned to a software timer. */

static void prvTimerCallback( TimerHandle_t xTimer )

{

       TickType_t xTimerPeriod;

       /* Query the period of the timer that expired. */

       xTimerPeriod = xTimerGetPeriod( xTimer );
```

```
}
```

xTimerGetTimerDaemonTaskHandle()

The following shows the xTimerGetTimerDaemonTaskHandle() function prototype.

```
#include "FreeRTOS.h"

#include "timers.h"

TaskHandle_t xTimerGetTimerDaemonTaskHandle( void );
```

Summary

Returns the task handle associated with the software timer daemon (or service) task. If configUSE_TIMERS is set to 1 in FreeRTOSConfig.h, then the timer daemon task is created automatically when the scheduler is started. All FreeRTOS software timer callback functions run in the context of the timer daemon task.

Parameters

None.

Return Values

The handle of the timer daemon task. FreeRTOS software timer callback functions run in the context of the software daemon task.

Notes

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

pvTimerGetTimerID()

The following shows the pvTimerGetTimerID() function prototype.

```
#include "FreeRTOS.h"

#include "timers.h"

void *pvTimerGetTimerID( TimerHandle_t xTimer );
```

Summary

Returns the identifier (ID) assigned to the timer. An identifier is assigned to the timer when the timer is created, and can be updated using the vTimerSetTimerID() API function. See the xTimerCreate() API function for more information.

If the same callback function is assigned to multiple timers, the timer identifier can be inspected inside the callback function to determine which timer actually expired. This is demonstrated in the example code provided for the xTimerCreate() API function.

In addition the timer's identifier can be used to store values in between calls to the timer's callback function.

Parameters

xTimer	The timer being queried.

Return Values

The identifier assigned to the timer being queried.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for pvTimerGetTimerID() to be available.

Example

```
/* A callback function assigned to a timer. */

void TimerCallbackFunction( TimerHandle_t pxExpiredTimer )

{
```

```
    uint32_t ulCallCount;

    /* A count of the number of times this timer has expired and executed its callback
 function is stored in the timer's ID. Retrieve the count, increment it, and then save it
 back into the timer's ID. */

    ulCallCount = ( uint32_t ) pvTimerGetTimerID( pxExpiredTimer );

    ulCallCount++;

    vTimerSetTimerID( pxExpiredTimer, ( void * ) ulCallCount );

}
```

xTimerIsTimerActive()

The following shows the xTimerIsTimerActive() function prototype. .. code:

```
#include "FreeRTOS.h"

#include "timers.h"

BaseType_t xTimerIsTimerActive( TimerHandle_t xTimer );
```

Summary

Queries a timer to determine if the timer is running.

A timer will not be running if:

1. The timer has been created, but not started.
2. The timer is a one-shot timer that has not been restarted since it expired.

The xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod(), and xTimerChangePeriodFromISR() API functions can all be used to start a timer running.

Parameters

xTimer	The timer being queried.

Return Values

pdFALSE	The timer is not running.
Any other value	The timer is running.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerIsTimerActive() to be available.

Example

```
/* This function assumes xTimer has already been created. */
```

```
void vAFunction( TimerHandle_t xTimer )

{

    /* The following line could equivalently be written as:
 "if( xTimerIsTimerActive( xTimer ) )" */

    if( xTimerIsTimerActive( xTimer ) != pdFALSE )

    {

        /* xTimer is active, do something. */

    }

    else

    {

        /* xTimer is not active, do something else. */

    }

}
```

xTimerPendFunctionCall()

The following shows the xTimerPendFunctionCall() function prototype.

```
#include "FreeRTOS.h"

#include "timers.h"

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

Summary

Used to defer the execution of a function to the RTOS daemon task also known as the timer service task. (It is for this reason the function is implemented in timers.c and prefixed with 'Timer'.)

This function must not be called from an interrupt service routine. See xTimerPendFunctionCallFromISR() for a version that can be called from an interrupt service routine.

Functions that can be deferred to the RTOS daemon task must have the prototype shown here.

```
void vPendableFunction( void *pvParameter1, uint32_t ulParameter2 );
```

The pvParameter1 and ulParameter2 parameters are provided for use by the application code.

Parameters

xFunctionToPend	The function to execute from the timer service/ daemon task. The function must conform to the PendedFunction_t prototype.
pvParameter1	The value to pass into the callback function as the function's first parameter. The parameter has a void * type to allow it to be used to pass any type. For example, integer types can be cast to a void <problematic>*</problematic> , or the void * can be used to point to a structure.
ulParameter2	The value to pass into the callback function as the function's second parameter.
xTicksToWait	Calling xTimerPendFunctionCall() will result in a message being sent on a queue to the timer daemon task (also known as the timer service task). xTicksToWait specifies the amount of time the calling task should wait in the Blocked state (so not consuming any processing time) for space to come available on the queue if the queue is full.

Return Values

pdPASS	The message was successfully sent to the RTOS daemon task.
Any other value	The message was not sent to the RTOS daemon task because the message queue was already full. The length of the queue is set by the value of configTIMER_QUEUE_LENGTH in FreeRTOSConfig.h.

Notes

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

xTimerPendFunctionCallFromISR()

The following shows the xTimerPendFunctionCallFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "timers.h"

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

Summary

Used from application interrupt service routines to defer the execution of a function to the RTOS daemon task, also known as the timer service task. (It is for this reason the function is implemented in timers.c and prefixed with 'Timer'.)

Ideally, an interrupt service routine (ISR) is kept as short as possible, but sometimes an ISR either has a lot of processing to do or needs to perform processing that is not deterministic. In these cases, xTimerPendFunctionCallFromISR() can be used to defer processing of a function to the RTOS daemon task.

A mechanism is provided that allows the interrupt to return directly to the task that will subsequently execute the pended function. This allows the callback function to execute contiguously in time with the interrupt, just as if the callback had executed in the interrupt itself.

Functions that can be deferred to the RTOS daemon task must have the prototype demonstrated here.

```
void vPendableFunction( void *pvParameter1, uint32_t ulParameter2 );
```

The pvParameter1 and ulParameter2 parameters are provided for use by the application code.

Parameters

xFunctionToPend

The function to execute from the timer service/daemon task. The function must conform to the PendedFunction_t prototype shown here.

pvParameter1

The value that will be passed into the callback function as the function's first parameter. The parameter has a void * type to allow it to be used to pass any type. For example, integer types can be cast to a void *, or the void * can be used to point to a structure.

ulParameter2

The value that will be passed into the callback function as the function's second parameter.

pxHigherPriorityTaskWoken

Calling xTimerPendFunctionCallFromISR() will result in a message being sent on a queue to the RTOS timer daemon task. If the priority of the daemon task (which is set by the value of configTIMER_TASK_PRIORITY in FreeRTOSConfig.h) is higher than the priority of the currently running task (the task the interrupt interrupted) then `*pxHigherPriorityTaskWoken` will be set to pdTRUE within xTimerPendFunctionCallFromISR(), indicating that a context switch should be requested before the interrupt exits. For that reason, `*pxHigherPriorityTaskWoken` must be initialized to pdFALSE.

Return Values

pdPASS	The message was successfully sent to the RTOS daemon task.
Any other value	The message was not sent to the RTOS daemon task because the message queue was already full. The length of the queue is set by the value of configTIMER_QUEUE_LENGTH in FreeRTOSConfig.h.

Notes

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

Example

```
/* The callback function that will execute in the context of the daemon task.    All
 callback functions must use this same prototype. */

void vProcessInterface( void *pvParameter1, uint32_t ulParameter2 )

{

    BaseType_t xInterfaceToService;

    /* The interface that requires servicing is passed in the second parameter. The first
 parameter is not used in this case. */

    xInterfaceToService = ( BaseType_t ) ulParameter2;

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

}

/* An ISR that receives data packets from multiple interfaces */

void vAnISR( void )

{

        BaseType_t xInterfaceToService, xHigherPriorityTaskWoken;
```

```
        /* Query the hardware to determine which interface needs processing. */

        xInterfaceToService = prvCheckInterfaces();

        /* The actual processing is to be deferred to a task. Request the
    vProcessInterface() callback function is executed, passing in the number of
               the interface that needs processing. The interface to service is passed in the
    second parameter. The first parameter is not used in this case. */

        xHigherPriorityTaskWoken = pdFALSE;

        xTimerPendFunctionCallFromISR( vProcessInterface,

                                       NULL,
                                       ( uint32_t ) xInterfaceToService,
                                       &xHigherPriorityTaskWoken );

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

        portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

xTimerReset()

The following shows the xTimerReset() function prototype.

```
#include "FreeRTOS.h"

#include "timers.h"

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

Summary

Restarts a timer. xTimerResetFromISR() is an equivalent function that can be called from an interrupt service routine.

If the timer is already running, then the timer will recalculate its expiry time to be relative to when xTimerReset() was called.

If the timer was not running, then the timer will calculate an expiry time relative to when xTimerReset() was called, and the timer will start running. In this case, xTimerReset() is functionally equivalent to xTimerStart().

Resetting a timer ensures the timer is running. If the timer is not stopped, deleted, or reset in the meantime, the callback function associated with the timer will get called *n* ticks after xTimerReset() was called, where *n* is the timer's defined period.

If xTimerReset() is called before the scheduler is started, then the timer will not start running until the scheduler has been started, and the timer's expiry time will be relative to when the scheduler started.

Parameters

xTimer	The timer being reset, started, or restarted.
xTicksToWait	Timer functionality is not provided by the core FreeRTOS code, but by a timer service (or daemon) task. The FreeRTOS timer API sends commands to the timer service task on a queue called the timer command queue. xTicksToWait specifies the maximum amount of time the task should remain in the Blocked state to wait for space to become available on the timer command queue, should the queue already be full. The block time is specified in tick periods, so the absolute time it represents depends on the tick frequency. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds to 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. xTicksToWait is ignored if xTimerReset() is called before the scheduler is started.

Return Values

pdPASS	The reset command was successfully sent to the timer command queue. If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state to wait for space to become available on the timer command queue before the function returned, but data was successfully written to the queue before the block time expired. When the command is actually processed will depend on the priority of the timer service task relative to other tasks in the system, but the timer's expiry time is relative to when xTimerReset() is actually called. The priority of the timer service task is set by the configTIMER_TASK_PRIORITY configuration constant.
pdFAIL	The reset command was not sent to the timer command queue because the queue was already full. If a block time was specified (xTicksToWait was not zero), then the calling task will have been placed into the Blocked state to wait for the timer service task to make room in the queue, but the specified block time expired before that happened.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerReset() to be available.

Example

```
/* In this example, when a key is pressed, an LCD backlight is
switched on. If 5 seconds pass without a key being pressed, then the LCD
backlight is switched off by a one-shot timer. */
```

```
TimerHandle_t xBacklightTimer = NULL;

/* The callback function assigned to the one-shot timer. In this case,
the parameter is not used. */

void vBacklightTimerCallback( TimerHandle_t pxTimer )

{

    /* The timer expired, so 5 seconds must have passed since a key
    was pressed. Switch off the LCD backlight. */

    vSetBacklightState( BACKLIGHT_OFF );

}

/* The key press event handler. */

void vKeyPressEventHandler( char cKey )

{

    /* Ensure the LCD backlight is on, and then reset the timer that is responsible for
 turning the
    backlight off after 5 seconds of key inactivity. Wait 10 ticks for the reset command to
 be
    successfully sent if it cannot be sent immediately. */

    vSetBacklightState( BACKLIGHT_ON );

    if( xTimerReset( xBacklightTimer, 10 ) != pdPASS )

    {

        /* The reset command was not executed successfully. Take appropriate
        action here. */

    }

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

}

void main( void )

{

    /* Create and then start the one-shot timer that is responsible for turning the
 backlight off
    if no keys are pressed within a 5-second period. */

    xBacklightTimer = xTimerCreate( "BcklghtTmr" /* Just a text name. Not used by the
 kernel. */

                                    pdMS_TO_TICKS( 5000 ), /* The timer period, in ticks. */

                                    pdFALSE, /* It is a one-shot timer. */

                                    0, /* ID not used by the callback so can take any value. */

                                    vBacklightTimerCallback /* The callback function that
 switches the LCD backlight off. */

                                );
```

```
    if( xBacklightTimer == NULL )

    {

            /* The timer was not created. */

    }

    else

    {

        /* Start the timer. No block time is specified. Even if one were, it would be
 ignored because the scheduler has not yet been started. */

        if( xTimerStart( xBacklightTimer, 0 ) != pdPASS )


        {
             /* The timer could not be set into the Active state. */

        }

   }

   /* Create tasks here. */

   /* Starting the scheduler will start the timer running because xTimerStart has already
 been called. */

   xTaskStartScheduler();

}
```

xTimerResetFromISR()

The following shows the xTimerResetFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "timers.h"

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

Summary

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

Parameters

xTimer	The handle of the timer that is being started, reset, or restarted.
pxHigherPriorityTaskWoken	xTimerResetFromISR() writes a command to the timer command queue. If writing to the timer command queue causes the timer service task to leave the Blocked state, and the timer service task has a priority equal to or greater than the currently executing task (the task that was interrupted), then <problematic>*</problematic> pxHigherPriorityTaskWoken will be set to pdTRUE internally within the xTimerResetFromISR() function. If xTimerResetFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt exits.

Return Values

pdPASS	The reset command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service task relative to other tasks in the system, although the timer's expiry time is relative to when xTimerResetFromISR() is actually called. The priority of the timer service task is set by the configTIMER_TASK_PRIORITY configuration constant.

pdFAIL	The reset command was not sent to the timer command queue because the queue was already full.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerResetFromISR() to be available.

Example

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

/* The callback function assigned to the one-shot timer. In this case,
the parameter is not used. */

void vBacklightTimerCallback( TimerHandle_t pxTimer )

{

    /* The timer expired, so 5 seconds must have passed since a key was pressed. Switch off
 the LCD backlight. */

    vSetBacklightState( BACKLIGHT_OFF );

}

/* The key press interrupt service routine. */

void vKeyPressEventInterruptHandler( void )

{

    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Ensure the LCD backlight is on, and then reset the timer that is responsible for
 turning the backlight off after 5 seconds of key inactivity. This is an interrupt service
 routine so can only call FreeRTOS API functions that end in "FromISR". */

    vSetBacklightState( BACKLIGHT_ON );

    /* xTimerStartFromISR() or xTimerResetFromISR() could be called here because both cause
 the timer to recalculate its expiry time. xHigherPriorityTaskWoken was initialized to
 pdFALSE when it was declared (in this function). */

    if( xTimerResetFromISR( xBacklightTimer, &xHigherPriorityTaskWoken ) != pdPASS )

    {

        /* The reset command was not executed successfully. Take appropriate action here.
 */

    }
```

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

    /* If xHigherPriorityTaskWoken equals pdTRUE, then a context switch should be
 performed. The syntax required to perform a context switch from inside an ISR varies from
 port to port and from compiler to compiler. Inspect the demos for the port you are using
 to find the actual syntax required. */

    if( xHigherPriorityTaskWoken != pdFALSE )

    {

        /* Call the interrupt-safe yield function here. (The function depends on the
 FreeRTOS port being used.) */

    }

}
```

vTimerSetTimerID()

The following shows the vTimerSetTimerID() function prototype.

```
#include "FreeRTOS.h"

#include "timers.h"

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

Summary

An identifier (ID) is assigned to a timer when the timer is created. It can be changed at any time by using the vTimerSetTimerID() API function.

If the same callback function is assigned to multiple timers, the timer identifier can be inspected inside the callback function to determine which timer expired.

The timer identifier can also be used to store data in the timer between calls to the timer's callback function.

Parameters

xTimer	The handle of the timer being updated with a new identifier.
pvNewID	The value to which the timer's identifier will be set.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerSetTimerID() to be available.

Example

The following example shows the use of vTimerSetTimerID().

```
/* A callback function assigned to a timer. */

void TimerCallbackFunction( TimerHandle_t pxExpiredTimer )

{

    uint32_t ulCallCount;
```

```
    /* A count of the number of times this timer has expired and executed its callback
 function is stored in the timer's ID. Retrieve the count, increment it, and then save it
 back into the timer's ID. */

    ulCallCount = ( uint32_t ) pvTimerGetTimerID( pxExpiredTimer );

    ulCallCount++;

    vTimerSetTimerID( pxExpiredTimer, ( void * ) ulCallCount );

}
```

xTimerStart()

The following shows the xTimerStart() function prototype.

```
#include "FreeRTOS.h"

#include "timers.h"

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

Summary

Starts a timer running. xTimerStartFromISR() is an equivalent function that can be called from an interrupt service routine.

If the timer was not already running, then the timer will calculate an expiry time relative to when xTimerStart() was called.

If the timer was already running, then xTimerStart() is functionally equivalent to xTimerReset().

If the timer is not stopped, deleted, or reset in the meantime, the callback function associated with the timer will get called *n* ticks after xTimerStart() was called, where *n* is the timer's defined period.

Parameters

xTimer	The timer to be reset, started, or restarted.
xTicksToWait	Timer functionality is not provided by the core FreeRTOS code, but by a timer service (or daemon) task. The FreeRTOS timer API sends commands to the timer service task on a queue called the timer command queue. xTicksToWait specifies the maximum amount of time the task should remain in the Blocked state to wait for space to become available on the timer command queue, should the queue already be full. The block time is specified in tick periods, so the absolute time it represents depends on the tick frequency. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds to 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. xTicksToWait is ignored if xTimerStart() is called before the scheduler is started.

Return Values

pdPASS	The start command was successfully sent to the timer command queue. If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state to wait for space to become available on the timer command queue before the function returned, but data was successfully written to the queue before the block time expired. When the command is actually processed will depend on the priority of the timer service task relative to other tasks in the system, although the timer's expiry time is relative to when xTimerStart() is actually called. The priority of the timer service task is set by the configTIMER_TASK_PRIORITY configuration constant.
pdFAIL	The start command was not sent to the timer command queue because the queue was already full. If a block time was specified (xTicksToWait was not zero), then the calling task will have been placed into the Blocked state to wait for the timer service task to make room in the queue, but the specified block time expired before that happened.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerStart() to be available.

Example

See the example provided for the xTimerCreate() API function.

xTimerStartFromISR()

The following shows the xTimerStartFromISR() macro prototype.

```
#include "FreeRTOS.h"

#include "timers.h"

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

Summary

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

Parameters

xTimer	The handle of the timer that is being started, reset, or restarted.
pxHigherPriorityTaskWoken	xTimerStartFromISR() writes a command to the timer command queue. If writing to the timer command queue causes the timer service task to leave the Blocked state, and the timer service task has a priority equal to or greater than the currently executing task (the task that was interrupted), then <problematic>*</problematic> pxHigherPriorityTaskWoken will be set to pdTRUE internally within the xTimerStartFromISR() function. If xTimerStartFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt exits.

Return Values

pdPASS	The start command was successfully sent to the timer command queue. The processing of the command depends on the priority of the timer service task relative to other tasks in the system. The timer's expiry time is relative to when xTimerStartFromISR() is actually called. The priority of the timer service task is set by the configTIMER_TASK_PRIORITY configuration constant.

pdFAIL	The start command was not sent to the timer command queue because the queue was already full.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerStartFromISR() to be available.

Example

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

/* The callback function assigned to the one-shot timer. In this case, the parameter is not
 used. */

void vBacklightTimerCallback( TimerHandle_t pxTimer )

{

    /* The timer expired. Therefore, 5 seconds must have passed since a key was pressed.
 Switch off the LCD backlight. */

    vSetBacklightState( BACKLIGHT_OFF );

 }

 /* The key press interrupt service routine. */

 void vKeyPressEventInterruptHandler( void )

 {

    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Ensure the LCD backlight is on, and then restart the timer that is responsible for
 turning the backlight off after 5 seconds of key inactivity. This is an interrupt service
 routine, so can only call FreeRTOS API functions that end in "FromISR". */

    vSetBacklightState( BACKLIGHT_ON );

    /* xTimerStartFromISR() or xTimerResetFromISR() could be called here because both cause
 the timer to recalculate its expiry time. xHigherPriorityTaskWoken was initialized to
 pdFALSE when it was declared (in this function). */

    if( xTimerStartFromISR( xBacklightTimer, &xHigherPriorityTaskWoken ) != pdPASS )

    {

        /* The start command was not executed successfully. Take appropriate action here.
 */

    }

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

```
   /* If xHigherPriorityTaskWoken equals pdTRUE, then a context switch should be
performed. The syntax required to perform a context switch from inside an ISR varies from
port to port and from compiler to compiler. Inspect the demos for the port you are using
to find the syntax required. */

   if( xHigherPriorityTaskWoken != pdFALSE )

   {

      /* Call the interrupt-safe yield function here. (The function depends on the
FreeRTOS port being used.) */

   }

}
```

xTimerStop()

The following shows the xTimerStop() function prototype.

```
#include "FreeRTOS.h"

#include "timers.h"

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

Summary

Stops a timer running. xTimerStopFromISR() is an equivalent function that can be called from an interrupt service routine.

Parameters

xTimer	The timer to be stopped.
xTicksToWait	Timer functionality is not provided by the core FreeRTOS code, but by a timer service (or daemon) task. The FreeRTOS timer API sends commands to the timer service task on a queue called the timer command queue. xTicksToWait specifies the maximum amount of time the task should remain in the Blocked state to wait for space to become available on the timer command queue, should the queue already be full. The block time is specified in tick periods, so the absolute time it represents depends on the tick frequency. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds to 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. xTicksToWait is ignored if xTimerStop() is called before the scheduler is started.

Return Values

pdPASS	The stop command was successfully sent to the timer command queue.

	If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state to wait for space to become available on the timer command queue before the function returned, but data was successfully written to the queue before the block time expired. When the command is actually processed will depend on the priority of the timer service task relative to other tasks in the system. The priority of the timer service task is set by the configTIMER_TASK_PRIORITY configuration constant.
pdFAIL	The stop command was not sent to the timer command queue because the queue was already full. If a block time was specified (xTicksToWait was not zero), then the calling task will have been placed into the Blocked state to wait for the timer service task to make room in the queue, but the specified block time expired before that happened.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerStop() to be available.

Example

See the example provided for the xTimerCreate() API function.

xTimerStopFromISR()

The following shows the xTimerStopFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "timers.h"

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

Summary

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

Parameters

xTimer	The handle of the timer that is being stopped.
pxHigherPriorityTaskWoken	xTimerStopFromISR() writes a command to the timer command queue. If writing to the timer command queue causes the timer service task to leave the Blocked state, and the timer service task has a priority equal to or greater than the currently executing task (the task that was interrupted), then <problematic>*</problematic> pxHigherPriorityTaskWoken will be set to pdTRUE internally within the xTimerStopFromISR() function. If xTimerStopFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt exits.

Return Values

pdPASS	The stop command was successfully sent to the timer command queue. The time at which the command is processed depends on the priority of the timer service task relative to other tasks in the system. The priority of the timer service task is set by the configTIMER_TASK_PRIORITY configuration constant.
pdFAIL	The stop command was not sent to the timer command queue because the queue was already full.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerStopFromISR() to be available.

Example

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

/* The interrupt service routine that stops the timer. */

void vAnExampleInterruptServiceRoutine( void )

{

    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* The interrupt has occurred. Simply stop the timer. xHigherPriorityTaskWoken

     was set to pdFALSE where it was defined (within this function). Because this is an

     interrupt service routine, only FreeRTOS API functions that end in "FromISR" can be
 used. */

     if( xTimerStopFromISR( xTimer, &xHigherPriorityTaskWoken ) != pdPASS )

     {

         /* The stop command was not executed successfully. Take appropriate action here.
 */

     }

     /* If xHigherPriorityTaskWoken equals pdTRUE, then a context switch should be
 performed. The syntax required to perform a context switch from inside an ISR
     varies from port to port and from compiler to compiler. Inspect the demos for the port
 you are using to find the syntax required. */

    if( xHigherPriorityTaskWoken != pdFALSE )

    {

         /* Call the interrupt-safe yield function here. (The function depends on the
 FreeRTOS port being used.) */

    }

}
```

Event Groups API

xEventGroupClearBits()

The following shows the xEventGroupClearBits() function prototype.

```
#include "FreeRTOS.h"

#include "event_groups.h"

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

Summary

Clear bits (flags) within an RTOS event group. This function cannot be called from an interrupt. See xEventGroupClearBitsFromISR() for a version that can be called from an interrupt.

Parameters

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

Return Values

All values	The value of the bits in the event group before any bits were cleared.

Notes

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

Example

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

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

```
void aFunction( EventGroupHandle_t xEventGroup )

{

     EventBits_t uxBits;

     /* Clear bit 0 and bit 4 in xEventGroup. */

     uxBits = xEventGroupClearBits(xEventGroup, /* The event group being updated. */ BIT_0
 | BIT_4 );/* The bits being cleared. */

     if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
     {

        /* Both bit 0 and bit 4 were set before xEventGroupClearBits() was called. Both
 will now be clear (not set). */

    }

    else if( ( uxBits & BIT_0 ) != 0 )

    {

           /* Bit 0 was set before xEventGroupClearBits() was called. It will now be clear.
 */

    }

    else if( ( uxBits & BIT_4 ) != 0 )

    {

          /* Bit 4 was set before xEventGroupClearBits() was called. It will now be clear.
 */

    }

    else

    {

         /* Neither bit 0 nor bit 4 were set in the first place. */

    }

}
```

xEventGroupClearBitsFromISR()

The following shows the xEventGroupClearBitsFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "event_groups.h"

BaseType_t xEventGroupClearBitsFromISR( EventGroupHandle_t xEventGroup, const EventBits_t
 uxBitsToClear );
```

Summary

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

xEventGroupClearBitsFromISR() sends a message to the RTOS daemon task to have the clear operation performed in the context of the daemon task. The priority of the daemon task is set by configTIMER_TASK_PRIORITY in FreeRTOSConfig.h.

Parameters

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

Return Values

pdPASS	The message was sent to the RTOS daemon task.
pdFAIL	The message could not be sent to the RTOS daemon task (also known as the timer service task) because the timer command queue was full. The length of the queue is set by the configTIMER_QUEUE_LENGTH setting in FreeRTOSConfig.h.

Notes

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

Example

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

#define BIT_4 ( 1 << 4 )

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

void anInterruptHandler( void )

{

    BaseType_t xSuccess;

    /* Clear bit 0 and bit 4 in xEventGroup. */

    xSuccess = xEventGroupClearBitsFromISR(xEventGroup, /* The event group being updated.
 */ BIT_0 | BIT_4 );/* The bits being cleared. */

    if( xSuccess == pdPASS )

    {

        /* The clear bits message was sent to the daemon task. */

    }

    else

    {

        /* The clear bits message was not sent to the daemon task. */

    }

}
```

xEventGroupCreate()

The following shows the xEventGroupCreate() function prototype.

```
#include "FreeRTOS.h"

#include "event_groups.h"

EventGroupHandle_t xEventGroupCreate( void );
```

Summary

Creates a new event group and returns a handle by which the created event group can be referenced.

Each event group requires a very small amount of RAM that is used to hold the event group's state. If an event group is created by using xEventGroupCreate(), then this RAM is allocated from the FreeRTOS heap automatically. If an event group is created by using xEventGroupCreateStatic(), then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

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

This function cannot be called from an interrupt.

Parameters

None.

Return Values

NULL	The event group could not be created because there was insufficient FreeRTOS heap available.
Any other value	The event group was created and the value returned is the handle of the created event group.

Notes

configSUPPORT_DYNAMIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h (or left undefined, in which case it will default to 1) and the RTOS source file FreeRTOS/source/event_groups.c must be included in the build for the xEventGroupCreate() function to be available.

Example

```
/* 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. */

}
```

xEventGroupCreateStatic()

The following shows the xEventGroupCreateStatic() function prototype.

```
#include "FreeRTOS.h"

#include "event_groups.h"

EventGroupHandle_t xEventGroupCreateStatic( StaticEventGroup_t *pxEventGroupBuffer );
```

Summary

Creates a new event group and returns a handle by which the created event group can be referenced.

Each event group requires a very small amount of RAM that is used to hold the event group's state. If an event group is created using xEventGroupCreate(), then this RAM is allocated from the FreeRTOS heap automatically. If an event group is created using xEventGroupCreateStatic(), then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

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

Parameters

pxEventGroupBuffer	Must point to a variable of type StaticEventGroup_t, in which the event group's data structure will be stored.

Return Values

NULL	The event group could not be created because pxEventGroupBuffer was NULL.
Any other value	The event group was created and the value returned is the handle of the created event group.

Notes

configSUPPORT_STATIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h, and the RTOS source file FreeRTOS/source/event_groups.c must be included in the build, for the xEventGroupCreateStatic() function to be available.

Example

```
/* Declare a variable to hold the handle of the created event group.*/

EventGroupHandle_t xEventGroupHandle;

/* Declare a variable to hold the data associated with the created event group. */

StaticEventGroup_t xCreatedEventGroup;

void vAFunction( void )

{

    /* Attempt to create the event group. */

    xEventGroupHandle = xEventGroupCreate( &xCreatedEventGroup );

    /* pxEventGroupBuffer was not null so expect the event group to have been created. */

    configASSERT( xEventGroupHandle );

}
```

vEventGroupDelete()

The following shows the vEventGroupDelete() function prototype.

```
#include "FreeRTOS.h"

#include "event_groups.h"

void vEventGroupDelete( EventGroupHandle_t xEventGroup );
```

Summary

Delete an event group that was previously created using a call to xEventGroupCreate().

Tasks that are blocked on the event group being deleted will be unblocked and report an event group value of 0.

This function must not be called from an interrupt.

Parameters

xEventGroup	The event group to delete.

Return Values

None.

Notes

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

xEventGroupGetBits()

The following shows the xEventGroupGetBits() function prototype.

```
#include "FreeRTOS.h"

#include "event_groups.h"

EventBits_t xEventGroupGetBits( EventGroupHandle_t xEventGroup );
```

Summary

Returns the current value of the event bits (event flags) in an event group. This function cannot be used from an interrupt. See xEventGroupGetBitsFromISR() for a version that can be used in an interrupt.

Parameters

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

Return Values

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

Notes

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

xEventGroupGetBitsFromISR()

The following shows the xEventGroupGetBitsFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "event_groups.h"

EventBits_t xEventGroupGetBitsFromISR( EventGroupHandle_t xEventGroup);
```

Summary

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

Parameters

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

Return Values

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

Notes

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

xEventGroupSetBits()

The following shows the xEventGroupSetBits() function prototype.

```
#include "FreeRTOS.h"

#include "event_groups.h"

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

Summary

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

Setting bits in an event group will automatically unblock any tasks that were blocked waiting for the bits to be set.

Parameters

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

Return Values

Any Value	The value of the bits in the event group at the time the call to xEventGroupSetBits() returned. There are two reasons why the returned value might have the bits specified by the uxBitsToSet parameter cleared: 1. If setting a bit results in a task that was waiting for the bit leaving the blocked state, then it is possible the bit will have been cleared automatically (see the xClearBitsOnExit parameter of xEventGroupWaitBits()). 2. Any task that leaves the blocked state as a result of the bits being set (or otherwise any

	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.

Notes

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

Example

```
#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 because 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 because a task that was waiting for bit 0 was
 removed from the Blocked state. */

    }

    else

    {
```

```
        /* Neither bit 0 nor bit 4 remained set. It might be that a task was waiting for
 both of the bits to be set, and the bits were cleared as the task left the Blocked state.
 */

    }

}
```

xEventGroupSetBitsFromISR()

The following shows the xEventGroupSetBitsFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "event_groups.h"

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

Summary

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

Setting bits in an event group will automatically unblock any tasks that were blocked waiting for the bits to be set.

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

Parameters

Return Values

pdPASS	The message was sent to the RTOS daemon task.
pdFAIL	The message could not be sent to the RTOS daemon task (also known as the timer service task) because the timer command queue was full. The length of the queue is set by the configTIMER_QUEUE_LENGTH setting in FreeRTOSConfig.h.

Notes

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

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

Example

```
#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 initialized to pdFALSE. */

    xHigherPriorityTaskWoken = pdFALSE;

    /* Set bit 0 and bit 4 in xEventGroup. */

    xResult = xEventGroupSetBitsFromISR(xEventGroup, /* The event group being updated. */
 BIT_0 | BIT_4 /* The bits being set. */ &xHigherPriorityTaskWoken );

    /* Was the message posted successfully? */

    if( xResult != pdFAIL )

    {

         /* If xHigherPriorityTaskWoken is now set to pdTRUE, then a context switch should
 be requested. The macro used is port-specific and will

         be either portYIELD_FROM_ISR() or portEND_SWITCHING_ISR(). See the documentation
 page for the port being used. */

         portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

    }

}
```

xEventGroupSync()

The following shows the xEventGroupSync() function prototype.

```
#include "FreeRTOS.h"

#include "event_groups.h"

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

Summary

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

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

This function cannot be used from an interrupt.

Parameters

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

Return Values

All values	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 in the returned value. 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.

Notes

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

Example

```
/* 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 synchronize 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 = pdMS_TO_TICKS( 100 );

    for( ;; )

    {

        /* Perform task functionality here. */

        . . .

        /* Set bit 0 in the event group to note this task has reached the
```

```
         sync point. The other two tasks will set the other two bits defined
         by ALL_SYNC_BITS. All three tasks have reached the synchronization
         point when all the ALL_SYNC_BITS bits are set. Wait a maximum of 100ms
         for this to happen. */
        uxReturn = xEventGroupSync( xEventBits,
             TASK_0_BIT, /* The bit to set. */
             ALL_SYNC_BITS, /* The bits to wait for. */
             xTicksToWait );/* Timeout value. */
        if( ( uxReturn & ALL_SYNC_BITS ) == ALL_SYNC_BITS )
        {
                /* All three tasks reached the synchronization point before the call to
 xEventGroupSync() timed out. */
        }
    }
}
void vTask1( void *pvParameters )
{
    for( ;; )
    {
            /* Perform task functionality here. */
              . . .
            /* Set bit 1 in the event group to note this task has reached the
                  synchronization point. The other two tasks will set the other two
                  bits defined by ALL_SYNC_BITS. All three tasks have reached the
                  synchronization 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 synchronization was made by all
            three tasks, so there is no need to test the return value. */
    }
}
 void vTask2( void *pvParameters )
```

```
{
      for( ;; )
      {
            /* Perform task functionality here. */
              . . .
            /* Set bit 2 in the event group to note this task has reached the
            synchronization point. The other two tasks will set the other two
            bits defined by ALL_SYNC_BITS. All three tasks have reached the
            synchronization 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 synchronization was made by all
            three tasks, so there is no need to test the return value. */
       }
}
```

xEventGroupWaitBits()

The following shows the xEventGroupWaitBits() function prototype.

```
#include "event_groups.h"

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

Summary

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

This function cannot be called from an interrupt.

Parameters

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

	all the bits set in the value passed as the uxBitsToWaitFor parameter are set in the event group or the specified block time expires. If xWaitForAllBits is set to pdFALSE, then xEventGroupWaitBits() will return when any of the bits set in the value passed as the uxBitsToWaitFor parameter are set in the event group or the specified block time expires.
xTicksToWait	The maximum amount of time (specified in ticks) to wait for one/all (depending on the xWaitForAllBits value) of the bits specified by uxBitsToWaitFor to become set.

Return Values

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

Notes

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

Example

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

#define BIT_4 ( 1 << 4 )

void aFunction( EventGroupHandle_t xEventGroup )

{

    EventBits_t uxBits;
```

```
   const TickType_t xTicksToWait = pdMS_TO_TICKS( 100 );

   /* 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. */

   }

}
```

Kernel Configuration

FreeRTOSConfig.h

Kernel configuration is achieved by setting #define constants in FreeRTOSConfig.h. Each application that uses FreeRTOS must provide a FreeRTOSConfig.h header file.

All the demo application projects included in the FreeRTOS download contain a predefined FreeRTOSConfig.h that can be used as a reference or simply copied. Some of the demo projects were generated before all the options documented here were available, so the FreeRTOSConfig.h header files do not include all the constants and options that are documented in the following sections.

Constants That Start with *INCLUDE_*

Constants that start with the text *INCLUDE_* are used to include or exclude FreeRTOS API functions from the application. For example, setting INCLUDE_vTaskPrioritySet to 0 will exclude the vTaskPrioritySet() API function from the build, meaning the application cannot call vTaskPrioritySet(). Setting INCLUDE_vTaskPrioritySet to 1 will include the vTaskPrioritySet() API function in the build, so the application can call vTaskPrioritySet().

In some cases, a single *INCLUDE_* configuration constant will include or exclude multiple API functions.

The *INCLUDE_* constants are provided to permit the code size to be reduced by removing FreeRTOS functions and features that are not required. However, by default, most linkers will remove unreferenced code unless optimization is turned completely off. Linkers that do not have this default behavior can be configured to remove unreferenced code. Therefore, in most cases, the *INCLUDE_* configuration constants will have little, if any, impact on the executable code size.

It is possible that excluding an API function from an application will also reduce the amount of RAM used by the FreeRTOS kernel. For example, removing the vTaskSuspend() API function will also prevent the structures that would otherwise reference Suspended tasks from ever being allocated.

INCLUDE_xEventGroupSetBitsFromISR

configUSE_TIMERS, INCLUDE_xTimerPendFunctionCall, and INCLUDE_xEventGroupSetBitsFromISR must all be set to 1 for the xEventGroupSetBitsFromISR () API function to be available.

INCLUDE_xSemaphoreGetMutexHolder

INCLUDE_xSemaphoreGetMutexHolder must be set to 1 for the xSemaphoreGetMutexHolder() API function to be available.

INCLUDE_xTaskAbortDelay

INCLUDE_xTaskAbortDelay must be set to 1 for the xTaskAbortDelay() API function to be available.

INCLUDE_vTaskDelay

INCLUDE_vTaskDelay must be set to 1 for the vTaskDelay() API function to be available.

INCLUDE_vTaskDelayUntil

INCLUDE_vTaskDelayUntil must be set to 1 for the vTaskDelayUntil() API function to be available.

INCLUDE_vTaskDelete

INCLUDE_vTaskDelete must be set to 1 for the vTaskDelete() API function to be available.

INCLUDE_xTaskGetCurrentTaskHandle

INCLUDE_xTaskGetCurrentTaskHandle must be set to 1 for the xTaskGetCurrentTaskHandle() API function to be available.

INCLUDE_xTaskGetHandle

INCLUDE_xTaskGetHandle must be set to 1 for the xTaskGetHandle() API function to be available.

INCLUDE_xTaskGetIdleTaskHandle

INCLUDE_xTaskGetIdleTaskHandle must be set to 1 for the xTaskGetIdleTaskHandle() API function to be available.

INCLUDE_xTaskGetSchedulerState

INCLUDE_xTaskGetSchedulerState must be set to 1 for the xTaskGetSchedulerState() API function to be available.

INCLUDE_uxTaskGetStackHighWaterMark

INCLUDE_uxTaskGetStackHighWaterMark must be set to 1 for the uxTaskGetStackHighWaterMark() API function to be available.

INCLUDE_uxTaskPriorityGet

INCLUDE_uxTaskPriorityGet must be set to 1 for the uxTaskPriorityGet() API function to be available.

INCLUDE_vTaskPrioritySet

INCLUDE_vTaskPrioritySet must be set to 1 for the vTaskPrioritySet() API function to be available.

INCLUDE_xTaskResumeFromISR

INCLUDE_xTaskResumeFromISR and INCLUDE_vTaskSuspend must both be set to 1 for the xTaskResumeFromISR() API function to be available.

INCLUDE_eTaskGetState

INCLUDE_eTaskGetState must be set to 1 for the eTaskGetState() API function to be available.

INCLUDE_vTaskSuspend

INCLUDE_vTaskSuspend must be set to 1 for the vTaskSuspend(), vTaskResume(), and xTaskIsTaskSuspended() API functions to be available.

INCLUDE_vTaskSuspend and INCLUDE_xTaskResumeFromISR must both be set to 1 for the xTaskResumeFromISR() API function to be available.

Some queue and semaphore API functions allow the calling task to opt to be placed into the Blocked state to wait for a queue or semaphore event to occur. These API functions require that a maximum block period or timeout is specified. The calling task will then be held in the Blocked state until the queue or semaphore event occurs or the block period expires. The maximum block period that can be specified is defined by portMAX_DELAY. If INCLUDE_vTaskSuspend is set to 0, then specifying a block period of portMAX_DELAY will result in the calling task being placed into the Blocked state for a maximum of portMAX_DELAY ticks. If INCLUDE_vTaskSuspend is set to 1, then specifying a block period of portMAX_DELAY will result in the calling task being placed into the Blocked state indefinitely (without a timeout). In the second case, the block period is indefinite, so the only way out of the Blocked state is for the queue or semaphore event to occur.

INCLUDE_xTimerPendFunctionCall

configUSE_TIMERS and INCLUDE_xTimerPendFunctionCall must both be set to 1 for the xTimerPendFunctionCall() and xTimerPendFunctionCallFromISR () API functions to be available.

Constants That Start with config

Constants that start with "config" define attributes of the kernel or include or exclude features of the kernel.

configAPPLICATION_ALLOCATED_HEAP

By default, the FreeRTOS heap is declared by FreeRTOS and placed in memory by the linker. Setting configAPPLICATION_ALLOCATED_HEAP to 1 allows the heap to be declared by the application writer instead, which allows the application writer to place the heap wherever they like in memory.

If heap_1.c, heap_2.c, or heap_4.c is used, and configAPPLICATION_ALLOCATED_HEAP is set to 1, then the application writer must provide a uint8_t array with the exact name and dimension as shown here. The array will be used as the FreeRTOS heap. How the array is placed at a memory location depends on the compiler in use. See the documentation for your compiler.

configASSERT

Calls to configASSERT(x) exist at key points in the FreeRTOS kernel code.

If FreeRTOS is functioning and being used correctly, then the configASSERT() parameter will be non-zero. If the parameter is found to equal zero, then an error has occurred.

Most errors trapped by configASSERT() are a result of an invalid parameter being passed into a FreeRTOS API function. configASSERT() can therefore assist in runtime debugging. However, defining configASSERT() also increases the application code size and slows down its execution.

configASSERT() is equivalent to the standard C assert() macro. It is used in place of the standard C assert() macro because not all the compilers that can be used to build FreeRTOS provide an assert.h header file.

configASSERT() should be defined in FreeRTOSConfig.h.

Here is an example configASSERT() definition that assumed vAssertCalled() is defined elsewhere by the application.

configCHECK_FOR_STACK_OVERFLOW

Each task has a unique stack. If a task is created using the xTaskCreate() API function, then the stack is automatically allocated from the FreeRTOS heap, and the size of the stack is specified by the xTaskCreate() usStackDepth parameter. If a task is created using the xTaskCreateStatic() API function, then the stack is preallocated by the application writer.

Stack overflow is a very common cause of application instability. FreeRTOS provides two optional methods that can be used to assist in stack overflow detection and debugging. You use the configCHECK_FOR_STACK_OVERFLOW configuration constant to configure which method is used.

If configCHECK_FOR_STACK_OVERFLOW is not set to 0, then the application must also provide a stack overflow hook (or callback) function. The kernel will call the stack overflow hook whenever a stack overflow is detected.

The stack overflow hook function must be called vApplicationStackOverflowHook() and have the prototype shown here.

```
void vApplicationStackOverflowHook( TaskHandle_t *pxTask,signed char *pcTaskName );
```

The name and handle of the task that has exceeded its stack space are passed into the stack overflow hook function using the pcTaskName and pxTask parameters, respectively. A stack overflow can potentially corrupt these parameters, in which case the pxCurrentTCB variable can be inspected to determine which task caused the stack overflow hook function to be called.

Stack overflow checking can only be used on architectures that have a linear (rather than segmented) memory map.

Some processors will generate a fault exception in response to a stack corruption before the stack overflow callback function can be called.

Stack overflow checking increases the time taken to perform a context switch.

Stack overflow detection method one	Method one is selected by setting configCHECK_FOR_STACK_OVERFLOW to 1. It is likely that task stack utilization will reach its maximum when the task's context is saved to the stack during a context switch. Stack overflow detection method one checks the stack utilization at that time to ensure the task stack pointer remains within the valid stack area. The stack overflow hook function will be called if the stack pointer contains an invalid value (a value that references memory outside of the valid stack area). Method one is quick, but will not necessarily catch all stack overflow occurrences.
Stack overflow detection method two	Method two is selected by setting configCHECK_FOR_STACK_OVERFLOW to 2. Method two includes the checks performed by method one. In addition, method two will also verify that the limit of the valid stack region has not been overwritten. The stack allocated to a task is filled with a known pattern at the time the task is created. Method two checks the last *n* bytes within the valid stack range to ensure this pattern remains unmodified (has not been overwritten). The stack overflow hook function is called if any of these *n* bytes have changed from their original values. Method two is less efficient than method one, but still fast. It will catch most stack overflow occurrences, although it is conceivable that some could be missed (for example, where a stack overflow occurs without the last *n* bytes being written to).

configCPU_CLOCK_HZ

This must be set to the frequency of the clock that drives the peripheral used to generate the kernel's periodic tick interrupt. This is very often, but not always, equal to the main system clock frequency.

configSUPPORT_DYNAMIC_ALLOCATION

If configSUPPORT_DYNAMIC_ALLOCATION is set to 1, then RTOS objects can be created using RAM that is automatically allocated from the FreeRTOS heap. If configSUPPORT_DYNAMIC_ALLOCATION is set to 0, then RTOS objects can only be created using RAM provided by the application writer. See also configSUPPORT_STATIC_ALLOCATION.

If configSUPPORT_DYNAMIC_ALLOCATION is not defined, then it will default to 1.

configENABLE_BACKWARD_COMPATIBILITY

The FreeRTOS.h header file includes a set of #define macros that map the names of data types used in versions of FreeRTOS earlier than 8.0.0 to the names used in FreeRTOS version 8.0.0. The macros allow application code to update the version of FreeRTOS they are built against from a pre-8.0.0 version to a post-8.0.0 version without modification. Setting configENABLE_BACKWARD_COMPATIBILITY to 0 in FreeRTOSConfig.h excludes the macros from the build, and in so doing allowing validation that no pre-8.0.0 version names are being used.

configGENERATE_RUN_TIME_STATS

The task runtime statistics feature collects information about the amount of processing time each task is receiving. The feature requires the application to configure a statistics time base. The frequency of the runtime statistics time base must be at least ten times greater than the frequency of the tick interrupt.

Setting configGENERATE_RUN_TIME_STATS to 1 will include the runtime statistics gathering functionality and associated API in the build. Setting configGENERATE_RUN_TIME_STATS to 0 will exclude the runtime statistics gathering functionality and associated API from the build.

If configGENERATE_RUN_TIME_STATS is set to 1, then the application must also provide definitions for the macros described in the following table. If configGENERATE_RUN_TIME_STATS is set to 0, then the application must not define any of the macros described in the following table. Otherwise, there is a risk that the application will not compiler and/or link.

Macro	Description
portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()	This macro must be provided to initialize the peripheral used to generate the runtime statistics time base.
portGET_RUN_TIME_COUNTER_VALUE() or portALT_GET_RUN_TIME_COUNTER_VALUE(Time)	One of these two macros must be provided to return the current time base value. This is the total time that the application has been running in the chosen time base units. If the first macro is used, it must be defined to evaluate to the current time base value. If the second macro is used, it must be defined to set its 'Time' parameter to the

	current time base value. ('ALT' in the macro name is an abbreviation for alternative).

configIDLE_SHOULD_YIELD

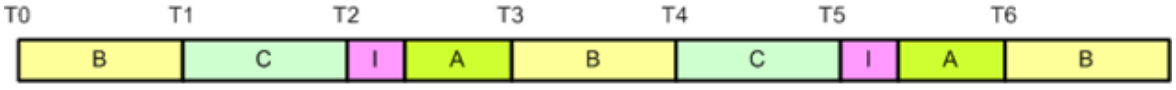
configIDLE_SHOULD_YIELD controls the behavior of the idle task if there are application tasks that also run at the idle priority. It only has an effect if the preemptive scheduler is being used.

Tasks that share a priority are scheduled using a round robin, time-sliced algorithm. Each task will be selected in turn to enter the running state, but might not remain in the running state for an entire tick period. For example, a task might be preempted, choose to yield, or choose to enter the Blocked state before the next tick interrupt.

If configIDLE_SHOULD_YIELD is set to 0, then the idle task will never yield to another task and will only leave the Running state when it is preempted.

If configIDLE_SHOULD_YIELD is set to 1, then idle task will never perform more than one iteration of its defined functionality without yielding to another task if there is another Idle priority task that is in the Ready state. This ensures a minimum amount of time is spent in the idle task when application tasks are available to run.

The following figure shows the side effect of an Idle task consistently yielding to another Idle priority Ready state task.



This figure shows the execution pattern of four tasks that all run at the idle priority. Tasks A, B, and C are application tasks. Task I is the idle task. The tick interrupt initiates a context switch at regular intervals, shown at times T0, T1, T2, and so on. You'll see that the Idle task starts to execute at time T2. It executes for part of a time slice, and then yields to Task A. Task A executes for the remainder of the same time slice, and then gets preempted at time T3. Task I and task A effectively share a single time slice, resulting in task B and task C consistently using more processing time than task A.

Setting configIDLE_SHOULD_YIELD to 0 prevents this behavior by ensuring the Idle task remains in the Running state for an entire tick period (unless preempted by an interrupt other than the tick interrupt). When this is the case, averaged over time, the other tasks that share the idle priority will get an equal share of the processing time, but more time will also be spent executing the idle task. Using an Idle task hook function can ensure the time spent executing the Idle task is used productively.

configINCLUDE_APPLICATION_DEFINED_PRIVILEGED_FU

configINCLUDE_APPLICATION_DEFINED_PRIVILEGED_FUNCTIONS is used by FreeRTOS MPU only.

If configINCLUDE_APPLICATION_DEFINED_PRIVILEGED_FUNCTIONS is set to 1, then the application writer must provide a header file called application_defined_privileged_functions.h in which functions the application writer needs to execute in privileged mode can be implemented. Despite having a .h extension, the header file should contain the implementation of the C functions, not just the functions' prototypes.

Functions implemented in application_defined_privileged_functions.h must save and restore the processor's privilege state using the prvRaisePrivilege() function and portRESET_PRIVILEGE() macro, respectively. For example, if a library-provided print function accesses RAM that is outside of the control of the application writer, and therefore cannot be allocated to a memory protected user mode task, then the print function can be encapsulated in a privileged function using the following code:

This technique should be used during development only, not deployment, because it circumvents the memory protection.

configKERNEL_INTERRUPT_PRIORITY, configMAX_SYSCALL_INTERRUPT_PRIORITY, configMAX_API_CALL_INTERRUPT_PRIORITY

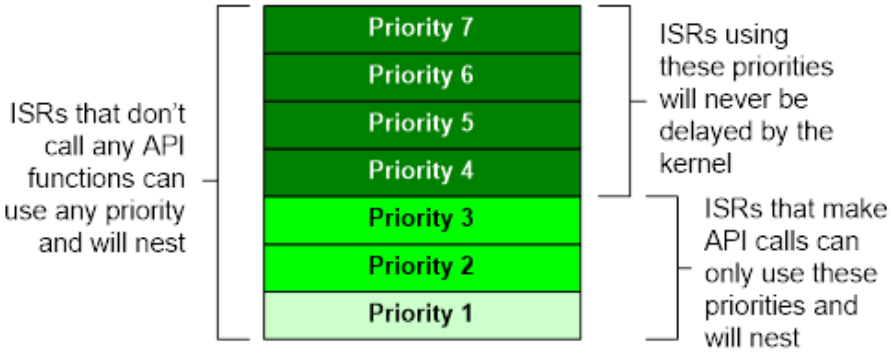
configMAX_API_CALL_INTERRUPT_PRIORITY is a new name for configMAX_SYSCALL_INTERRUPT_PRIORITY. It is used by newer ports only. The two are equivalent.

configKERNEL_INTERRUPT_PRIORITY and configMAX_SYSCALL_INTERRUPT_PRIORITY are only relevant to ports that implement interrupt nesting.

If a port only implements the configKERNEL_INTERRUPT_PRIORITY configuration constant, then configKERNEL_INTERRUPT_PRIORITY sets the priority of interrupts that are used by the kernel itself. In this case, ISR-safe FreeRTOS API functions (those that end in "FromISR") must not be called from any interrupt that has been assigned a priority above that set by configKERNEL_INTERRUPT_PRIORITY. Interrupts that do not call API functions can execute at higher priorities to ensure the interrupt timing, determinism, and latency are not adversely affected by anything the kernel is executing.

If a port implements both the configKERNEL_INTERRUPT_PRIORITY and the configMAX_SYSCALL_INTERRUPT_PRIORITY configuration constants, then configKERNEL_INTERRUPT_PRIORITY sets the interrupt priority of interrupts that are used by the kernel itself, and configMAX_SYSCALL_INTERRUPT_PRIORITY sets the maximum priority of interrupts from which ISR-safe FreeRTOS API functions (those that end in "FromISR") can be called. A full interrupt nesting model is achieved by setting configMAX_SYSCALL_INTERRUPT_PRIORITY above (that is, at a higher priority level) than configKERNEL_INTERRUPT_PRIORITY. Interrupts that do not call API functions can execute at priorities above configMAX_SYSCALL_INTERRUPT_PRIORITY to ensure the interrupt timing, determinism, and latency are not adversely affected by anything the kernel is executing.

As an example, imagine a hypothetical microcontroller that has seven interrupt priority levels. In this case, one is the lowest interrupt priority and seven is the highest interrupt priority. The following figure describes what can and cannot be done at each priority level when configKERNEL_INTERRUPT_PRIORITY and configMAX_SYSCALL_INTERRUPT_PRIORITY are set to one and three, respectively.



ISRs running above the configMAX_SYSCALL_INTERRUPT_PRIORITY are never masked by the kernel itself, so their responsiveness is not affected by the kernel functionality. This is ideal for interrupts that require very high temporal accuracy (for example, interrupts that perform motor commutation). However, interrupts that have a priority above configMAX_SYSCALL_INTERRUPT_PRIORITY cannot call any FreeRTOS API functions. Even those that end in "FromISR" cannot be used.

configKERNEL_INTERRUPT_PRIORITY will nearly always be set to the lowest available interrupt priority.

configMAX_CO_ROUTINE_PRIORITIES

Sets the maximum priority that can be assigned to a coroutine. Coroutines can be assigned a priority from zero, which is the lowest priority, to (configMAX_CO_ROUTINE_PRIORITIES - 1), which is the highest priority.

configMAX_PRIORITIES

Sets the maximum priority that can be assigned to a task. Tasks can be assigned a priority from zero, which is the lowest priority, to (configMAX_PRIORITIES - 1), which is the highest priority.

configMAX_TASK_NAME_LEN

Sets the maximum number of characters that can be used for the name of a task. The NULL terminator is included in the count of characters.

configMAX_SYSCALL_INTERRUPT_PRIORITY

See the description for the configKERNEL_INTERRUPT_PRIORITY configuration constant.

configMINIMAL_STACK_SIZE

Sets the size of the stack allocated to the Idle task. The value is specified in words, not bytes.

The kernel does not use configMINIMAL_STACK_SIZE for any other purpose, but the constant is used extensively by the standard demo tasks.

A demo application is provided for every official FreeRTOS port. The value of configMINIMAL_STACK_SIZE used in such a port-specific demo application is the minimum recommended stack size for any task created using that port.

configNUM_THREAD_LOCAL_STORAGE_POINTERS

Thread local storage (TLS) allows the application writer to store values inside a task's control block, making the value specific to (local to) the task itself and allowing each task to have its own unique value.

Each task has its own array of pointers that can be used as thread local storage. The number of indexes in the array is set by configNUM_THREAD_LOCAL_STORAGE_POINTERS.

configQUEUE_REGISTRY_SIZE

Sets the maximum number of queues and semaphores that can be referenced from the queue registry at any one time. Only queues and semaphores that need to be viewed in a kernel-aware debugging interface need to be registered.

The queue registry is required only when a kernel-aware debugger is being used. At all other times, it has no purpose and can be omitted by setting configQUEUE_REGISTRY_SIZE to 0 or by omitting the configQUEUE_REGISTRY_SIZE configuration constant definition altogether.

configSUPPORT_STATIC_ALLOCATION

If configSUPPORT_STATIC_ALLOCATION is set to 1, then RTOS objects can be created using RAM provided by the application writer. If configSUPPORT_STATIC_ALLOCATION is set to 0, then RTOS objects can only be created using RAM allocated from the FreeRTOS heap. See also configSUPPORT_DYNAMIC_ALLOCATION.

If configSUPPORT_STATIC_ALLOCATION is not defined, then it will default to 0.

configTICK_RATE_HZ

Sets the tick interrupt frequency. The value is specified in Hz.

The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds to ticks. Block times specified this way will remain constant even when the configTICK_RATE_HZ definition is changed. pdMS_TO_TICKS() can only be used when configTICK_RATE_HZ is less than or equal to 1000. The standard demo tasks make extensive use of pdMS_TO_TICKS(), so they too can only be used when configTICK_RATE_HZ is less than or equal to 1000.

configTIMER_QUEUE_LENGTH

Timer functionality is not provided by the core FreeRTOS code, but by a timer service (or daemon) task. The FreeRTOS timer API sends commands to the timer service task on a queue called the timer command queue. configTIMER_QUEUE_LENGTH sets the maximum number of unprocessed commands that the timer command queue can hold at any one time.

The following are some reasons the timer command queue might fill up:

- Multiple timer API function calls being made before the scheduler has been started, and therefore before the timer service task has been created.
- Multiple (interrupt-safe) timer API function calls being made from an interrupt service routine (ISR), and therefore not allowing the timer service task to process the commands.
- Multiple timer API function calls being made from a task that has a priority above that of the timer service task.

configTIMER_TASK_PRIORITY

Timer functionality is not provided by the core FreeRTOS code, but by a timer service (or daemon) task. The FreeRTOS timer API sends commands to the timer service task on a queue called the timer command

queue. configTIMER_TASK_PRIORITY sets the priority of the timer service task. Like all tasks, the timer service task can run at any priority between 0 and (configMAX_PRIORITIES - 1).

This value needs to be chosen carefully to meet the requirements of the application. For example, if the timer service task is made the highest priority task in the system, then commands sent to the timer service task (when a timer API function is called) and expired timers will both get processed immediately. Conversely, if the timer service task is given a low priority, then commands sent to the timer service task and expired timers will not be processed until the timer service task is the highest priority task that is able to run. Timer expiry times are calculated relative to when a command is sent, not to when a command is processed.

configTIMER_TASK_STACK_DEPTH

Timer functionality is not provided by the core FreeRTOS code, but by a timer service (or daemon) task. The FreeRTOS timer API sends commands to the timer service task on a queue called the timer command queue. configTIMER_TASK_STACK_DEPTH sets the size of the stack (in words, not bytes) allocated to the timer service task.

Timer callback functions execute in the context of the timer service task. The stack requirement of the timer service task therefore depends on the stack requirements of the timer callback functions.

configTOTAL_HEAP_SIZE

The kernel allocates memory from the heap each time a task, queue, or semaphore is created. The official FreeRTOS download includes three sample memory allocation schemes for this purpose. The schemes are implemented in the heap_1.c, heap_2.c, heap_3.c, and heap_4.c source files, respectively. The schemes defined by heap_1.c, heap_2.c, and heap_4.c allocate memory from a statically allocated array, known as the FreeRTOS heap. configTOTAL_HEAP_SIZE sets the size of this array. The size is specified in bytes.

The configTOTAL_HEAP_SIZE setting has no effect unless heap_1.c, heap_2.c, or heap_4.c are being used by the application.

configUSE_16_BIT_TICKS

The tick count is held in a variable of type TickType_t. When configUSE_16_BIT_TICKS is set to 1, TickType_t is defined to be an unsigned 16-bit type. When configUSE_16_BIT_TICKS is set to 0, TickType_t is defined to be an unsigned 32-bit type.

Using a 16-bit type can greatly improve efficiency on 8-bit and 16-bit microcontrollers, but at the cost of limiting the maximum block time that can be specified.

configUSE_ALTERNATIVE_API

Two sets of API functions are provided to send to, and receive from, queues: the standard API and the alternative API. Only the standard API is documented here because the use of the alternative API is not recommended.

Setting configUSE_ALTERNATIVE_API to 1 will include the alternative API functions in the build. Setting configUSE_ALTERNATIVE_API to 0 will exclude the alternative API functions from the build.

configUSE_APPLICATION_TASK_TAG

Setting configUSE_APPLICATION_TASK_TAG to 1 will include both the vTaskSetApplicationTaskTag() and xTaskCallApplicationTaskHook() API functions in the build. Setting configUSE_APPLICATION_TASK_TAG to 0 will exclude both the vTaskSetApplicationTaskTag() and the xTaskCallApplicationTaskHook() API functions from the build.

configUSE_CO_ROUTINES

Coroutines are lightweight tasks that save RAM by sharing a stack, but have limited functionality. Their use is omitted from this reference.

Setting configUSE_CO_ROUTINES to 1 will include all coroutine functionality and its associated API functions in the build. Setting configUSE_CO_ROUTINES to 0 will exclude all coroutine functionality and its associated API functions from the build.

configUSE_COUNTING_SEMAPHORES

Setting configUSE_COUNTING_SEMAPHORES to 1 will include the counting semaphore functionality and its associated API in the build. Setting configUSE_COUNTING_SEMAPHORES to 0 will exclude the counting semaphore functionality and its associated API from the build.

configUSE_DAEMON_TASK_STARTUP_HOOK

If configUSE_TIMERS and configUSE_DAEMON_TASK_STARTUP_HOOK are both set to 1, then the application must define a hook function that has the exact name and prototype as shown in the following code. The hook function will be called exactly once when the RTOS daemon task (also known as the timer service) executes for the first time. Any application initialization code that needs the RTOS to be running can be placed in the hook function.

```
void vApplicationDaemonTaskStartupHook( void );
```

configUSE_IDLE_HOOK

The idle task hook function is a hook (or callback) function that, if defined and configured, will be called by the Idle task on each iteration of its implementation.

If configUSE_IDLE_HOOK is set to 1, then the application must define an idle task hook function. If configUSE_IDLE_HOOK is set to 0, then the idle task hook function will not be called, even if one is defined.

Idle task hook functions must have the name and prototype shown in the following code.

void vApplicationIdleHook(void);

configUSE_MALLOC_FAILED_HOOK

The kernel uses a call to pvPortMalloc() to allocate memory from the heap each time a task, queue, or semaphore is created. The official FreeRTOS download includes three sample memory allocation

schemes for this purpose. The schemes are implemented in the heap_1.c, heap_2.c, heap_3.c, and heap_4.c source files, respectively. configUSE_MALLOC_FAILED_HOOK is only relevant when one of the three sample schemes is being used.

The malloc() failed hook function is a hook (or callback) function that, if defined and configured, will be called if pvPortMalloc() ever returns NULL. NULL will be returned only if there is insufficient FreeRTOS heap memory remaining for the requested allocation to succeed.

If configUSE_MALLOC_FAILED_HOOK is set to 1, then the application must define a malloc() failed hook function. If configUSE_MALLOC_FAILED_HOOK is set to 0, then the malloc() failed hook function will not be called, even if one is defined.

Malloc() failed hook functions must have the name and prototype shown in the following code.

```
void vApplicationMallocFailedHook( void );
```

configUSE_MUTEXES

Setting configUSE_MUTEXES to 1 will include the mutex functionality and its associated API in the build. Setting configUSE_MUTEXES to 0 will exclude the mutex functionality and its associated API from the build.

configUSE_NEWLIB_REENTRANT

If configUSE_NEWLIB_REENTRANT is set to 1, then a newlib reent structure will be allocated for each created task.

Newlib support has been included by popular demand, but is not used by the FreeRTOS maintainers themselves. FreeRTOS is not responsible for the resulting newlib operation. Users must be familiar with newlib and must provide system-wide implementations of the required stubs. The current newlib design implements a system-wide malloc() that must be provided with locks.

configUSE_PORT_OPTIMISED_TASK_SELECTION

Some FreeRTOS ports have two methods for selecting the next task to execute: a generic method and a port-specific method.

Generic method:

- Is used when configUSE_PORT_OPTIMISED_TASK_SELECTION is set to 0 or when a port specific method is not implemented.
- Can be used with all FreeRTOS ports.
- Is completely written in C, making it less efficient than a port-specific method.
- Does not impose a limit on the maximum number of available priorities.

Port-specific method:

- Is not available for all ports.
- Is used when configUSE_PORT_OPTIMISED_TASK_SELECTION is set to 1.

- Relies on one or more architecture-specific assembly instructions, typically a count leading zeros (CLZ) of equivalent instruction, so can only be used with the architecture for which it was specifically written.
- Is more efficient than the generic method.
- Typically imposes a limit of 32 on the maximum number of available priorities.

configUSE_PREEMPTION

Setting configUSE_PREEMPTION to 1 will cause the preemptive scheduler to be used. Setting configUSE_PREEMPTION to 0 will cause the cooperative scheduler to be used.

When the preemptive scheduler is used, the kernel will execute during each tick interrupt, which can result in a context switch occurring in the tick interrupt.

When the cooperative scheduler is used, a context switch will only occur when either:

1. A task explicitly calls taskYIELD().
2. A task explicitly calls an API function that results in it entering the Blocked state.
3. An application-defined interrupt explicitly performs a context switch.

configUSE_QUEUE_SETS

Setting configUSE_QUEUE_SETS to 1 will include queue set functionality (the ability to block on multiple queues at the same time) and its associated API in the build. Setting configUSE_QUEUE_SETS to 0 will exclude queue set functionality and its associated API from the build.

configUSE_RECURSIVE_MUTEXES

Setting configUSE_RECURSIVE_MUTEXES to 1 will cause the recursive mutex functionality and its associated API to be included in the build. Setting configUSE_RECURSIVE_MUTEXES to 0 will cause the recursive mutex functionality and its associated API to be excluded from the build.

configUSE_STATS_FORMATTING_FUNCTIONS

Set configUSE_TRACE_FACILITY and configUSE_STATS_FORMATTING_FUNCTIONS to 1 to include the vTaskList() and vTaskGetRunTimeStats() functions in the build. Setting either to 0 will omit vTaskList() and vTaskGetRunTimeStates() from the build.

configUSE_TASK_NOTIFICATIONS

Setting configUSE_TASK_NOTIFICATIONS to 1 (or leaving configUSE_TASK_NOTIFICATIONS undefined) will include direct-to-task notification functionality and its associated API in the build. Setting configUSE_TASK_NOTIFICATIONS to 0 will exclude direct-to-task notification functionality and its associated API from the build.

Each task consumes 8 additional bytes of RAM when direct-to-task notifications are included in the build.

configUSE_TICK_HOOK

The tick hook function is a hook (or callback) function that, if defined and configured, will be called during each tick interrupt.

If configUSE_TICK_HOOK is set to 1, then the application must define a tick hook function. If configUSE_TICK_HOOK is set to 0, then the tick hook function will not be called, even if one is defined.

Tick hook functions must have the name and prototype shown in the following code.

```
void vApplicationTickHook( void );
```

configUSE_TICKLESS_IDLE

Set configUSE_TICKLESS_IDLE to 1 to use the low-power tickless mode, or 0 to keep the tick interrupt running at all times. Low-power tickless implementations are not provided for all FreeRTOS ports.

configUSE_TIMERS

Setting configUSE_TIMERS to 1 will include software timer functionality and its associated API in the build. Setting configUSE_TIMERS to 0 will exclude software timer functionality and its associated API from the build.

If configUSE_TIMERS is set to 1, then configTIMER_TASK_PRIORITY, configTIMER_QUEUE_LENGTH, and configTIMER_TASK_STACK_DEPTH must also be defined.

configUSE_TIME_SLICING

By default, if configUSE_TIME_SLICING is not defined or if configUSE_TIME_SLICING is defined as 1, FreeRTOS uses prioritized preemptive scheduling with time slicing. That means the RTOS scheduler will always run the highest priority task that is in the Ready state, and will switch between tasks of equal priority on every RTOS tick interrupt. If configUSE_TIME_SLICING is set to 0, then the RTOS scheduler will still run the highest priority task that is in the Ready state, but will not switch between tasks of equal priority just because a tick interrupt executed.

configUSE_TRACE_FACILITY

Setting configUSE_TRACE_FACILITY to 1 will result in the inclusion in the build of additional structure members and functions that assist with execution visualization and tracing.

Data Types and Coding Style Guide

Data Types

Each port of FreeRTOS has a unique portmacro.h header file that contains definitions for two special data types: TickType_t and BaseType_t. These data types are described in the following table.

Macro or typedef used	Actual type
TickType_t	This is used to store the tick count value, and by variables that specify block times. TickType_t can be either an unsigned 16-bit type or an unsigned 32-bit type, depending on the setting of configUSE_16_BIT_TICKS within FreeRTOSConfig.h. Using a 16-bit type can greatly improve efficiency on 8-bit and 16-bit architectures, but severely limits the maximum block period that can be specified. There is no reason to use a 16-bit type on a 32-bit architecture.
BaseType_t	This is always defined to be the most efficient data type for the architecture. Typically, this is a 32-bit type on a 32-bit architecture, a 16-bit type on a 16-bit architecture, and an 8-bit type on an 8-bit architecture. BaseType_t is generally used for Booleans and for variables that can take only a very limited range of values.

Standard data types other than 'char' are not used. Instead, type names defined in the compiler's stdint.h header file are used. 'char' types are only permitted to point to ASCII strings or reference single-ASCII characters.

Variable Names

Variables are prefixed with their type: 'c' for char, 's' for short, 'l' for long, and 'x' for BaseType_t and any other types (structures, task handles, queue handles, and so on).

If a variable is unsigned, it is also prefixed with a 'u'. If a variable is a pointer, it is also prefixed with a 'p'. Therefore, a variable of type unsigned char will be prefixed with 'uc'. A variable of type pointer to char will be prefixed with 'pc'.

Function Names

Functions are prefixed with both the type they return and the file in which they are defined. For example: - vTaskPrioritySet() returns a void and is defined within task.c. - xQueueReceive() returns a variable of

type BaseType_t and is defined within queue.c. - vSemaphoreCreateBinary() returns a void and is defined within semphr.h. File scope (private) functions are prefixed with prv.

Formatting

One tab is always set to equal four spaces.

Macro Names

Most macros are written in uppercase and prefixed with lowercase letters that indicate where the macro is defined. The following table provides a list of prefixes.

Prefix	Location of macro definition
port (for example, portMAX_DELAY)	portable.h
task (for example, taskENTER_CRITICAL())	task.h
pd (for example, pdTRUE)	projdefs.h
config (for example, configUSE_PREEMPTION)	FreeRTOSConfig.h
err (for example, errQUEUE_FULL)	projdefs.h

The semaphore API is written almost entirely as a set of macros, but follows the function naming convention rather than the macro naming convention.

The macros defined in the following table are used throughout the FreeRTOS source code.

Macro	Value
pdTRUE	1
pdFALSE	0
pdPASS	1
pdFAIL	0

Rationale for Excessive Type Casting

The FreeRTOS source code can be compiled with many different compilers, all of which differ in how and when they generate warnings. In particular, different compilers want casting to be used in different ways. As a result, the FreeRTOS source code contains more type casting than would normally be warranted.

Stream Buffer API

Stream buffers are used to send a continuous stream of data from one task or interrupt to another. Their implementation is light weight, making them particularly suited for interrupt to task and core to core communication scenarios.

NOTE: Other FreeRTOS objects, such as queues, semaphores and event groups, are written to allow multiple writers and multiple readers. Uniquely among FreeRTOS objects, the implementation of stream buffers (message buffers are built on top of stream buffers) assumes there is only one writer and only one reader. THEREFORE, UNLIKE OTHER FREERTOS OBJECTS, THE IMPLEMENTATION OF THE STREAM AND MESSAGE BUFFER API FUNCTIONS DO NOT CONTAIN CRITICAL SECTIONS TO PROTECT AGAINST CONCURRENT ACCESS. If multiple tasks or interrupts write to the same stream buffer then the application writer must place the API calls that write to the stream buffer inside a critical section. Likewise, if multiple tasks or interrupts read from the same stream buffer then the application writer must place the API calls that read from the stream buffer inside a critical section. Critical sections are not required if there is only a single reader and a single writer even if the reader and writer are different tasks or interrupts.

xStreamBufferCreate()

The following shows the xStreamBufferCreate() function prototype.

```
#include "FreeRTOS.h"

#include "stream_buffer.h"

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

Summary

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).

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.

Return Value

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

```
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.
    }
```

xStreamBufferCreateStatic()

The following shows the xStreamBufferCreateStatic() function prototype.

```
#include "FreeRTOS.h"

#include "stream_buffer.h"

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

Summary

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

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.

Return Value

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.

```
// 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.
}
```

xStreamBufferSend()

The following shows the xStreamBufferSend() function prototype.

```
#include "FreeRTOS.h"

#include "stream_buffer.h"

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

Summary

Sends a bytes to the stream buffer. The data is copied into the stream buffer.

NOTE: Uniquely among FreeRTOS objects, the implementation of stream buffers (message buffers are built on top of stream buffers) assumes there is only one writer and only one reader. THEREFORE, UNLIKE OTHER FREERTOS OBJECTS, THE IMPLEMENTATIONS OF THE STREAM AND MESSAGE BUFFER API FUNCTIONS DO NOT CONTAIN CRITICAL SECTIONS TO PROTECT AGAINST CONCURRENT ACCESS. If multiple tasks or interrupts write to the same stream buffer then the application writer must place the API calls that write to the stream buffer inside a critical section.

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 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.

Return Value

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.

```
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 ), x100ms );

    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 ), 0 );

    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.
    }
```

xStreamBufferSendFromISR()

The following shows the xStreamBufferSendFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "stream_buffer.h"

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

Summary

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

NOTE: Uniquely among FreeRTOS objects, the implementation of stream buffers (message buffers are built on top of stream buffers) assumes there is only one writer and only one reader. THEREFORE, UNLIKE OTHER FREERTOS OBJECTS, THE IMPLEMENTATIONS OF THE STREAM AND MESSAGE BUFFER API FUNCTIONS DO NOT CONTAIN CRITICAL SECTIONS TO PROTECT AGAINST CONCURRENT ACCESS. If multiple tasks or interrupts write to the same stream buffer then the application writer must place the API calls that write to the stream buffer inside a critical section.

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.

Return Value

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

```
// 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 );
}
```

xStreamBufferReceive()

The following shows the xStreamBufferReceive() function prototype.

```
#include "FreeRTOS.h"

#include "stream_buffer.h"

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

Summary

Receives bytes from a stream buffer.

NOTE: Uniquely among FreeRTOS objects, the implementation of stream buffers (messages buffers are built on top of stream buffers) assumes there is only one writer and only one reader. THEREFORE, UNLIKE OTHER FREERTOS OBJECTS, THE IMPLEMENTATIONS OF THE STREAM AND MESSAGE BUFFER API FUNCTIONS DO NOT CONTAIN CRITICAL SECTIONS TO PROTECT AGAINST CONCURRENT ACCESS. If multiple tasks or interrupts write to the same stream buffer then the application writer must place the API calls that write to the stream buffer inside a critical section.

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 pucRxData 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.

Return Value

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

```
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....
    }
}
```

xStreamBufferReceiveFromISR()

The following shows the xStreamBufferReceiveFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "stream_buffer.h"

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

Summary

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 pucRxData 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.

Return Value

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

Example

```
// 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 );
}
```

vStreamBufferDelete()

The following shows the vStreamBufferDelete() function prototype.

```
#include "FreeRTOS.h"

#include "stream_buffer.h"

void vStreamBufferDelete( StreamBufferHandle_t xStreamBuffer );
```

Summary

Deletes a stream buffer that was previously created using a call to xStreamBufferCreate() or xStreamBufferCreateStatic(). 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.

xStreamBufferIsFull()

The following shows the xStreamBufferIsFull() function prototype.

```
#include "FreeRTOS.h"

#include "stream_buffer.h"

BaseType_t xStreamBufferIsFull( StreamBufferHandle_t xStreamBuffer );
```

Summary

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.

Return Value

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

xStreamBufferIsEmpty()

The following shows the xStreamBufferIsEmpty() function prototype.

```
#include "FreeRTOS.h"

#include "stream_buffer.h"

BaseType_t xStreamBufferIsEmpty( StreamBufferHandle_t xStreamBuffer );
```

Summary

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.

Return Value

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

xStreamBufferReset()

The following shows the xStreamBufferReset() function prototype.

```
#include "FreeRTOS.h"

#include "stream_buffer.h"

BaseType_t xStreamBufferReset( StreamBufferHandle_t xStreamBuffer );
```

Summary

Resets a stream buffer to its initial, empty, state. Any data that was int 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.

Return Value

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.

xStreamBufferSpacesAvailable()

The following shows the xStreamBufferSpacesAvailable() function prototype.

```
#include "FreeRTOS.h"

#include "stream_buffer.h"

BaseType_t xStreamBufferSpacesAvailable( StreamBufferHandle_t xStreamBuffer );
```

Summary

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.

Return Value

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

xStreamBufferBytesAvailable()

The following shows the xStreamBufferBytesAvailable() function prototype.

```
#include "FreeRTOS.h"

#include "stream_buffer.h"

BaseType_t xStreamBufferBytesAvailable( StreamBufferHandle_t xStreamBuffer );
```

Summary

Queries a stream buffer to see how much data it contains, which is equal to 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.

Return Value

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

xStreamBufferSetTriggerLevel()

The following shows the xStreamBufferSetTriggerLevel() function prototype.

```
#include "FreeRTOS.h"

#include "stream_buffer.h"

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

Summary

A stream buffer's trigger level is the number of bytes that must be in the stream buffer 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 queried.

Return Value

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

Message Buffer API

Message buffers build functionality on top of FreeRTOS stream buffers. Whereas stream buffers are used to send a continuous stream of data from one task or interrupt to another, message buffers are used to send variable length discrete messages from one task or interrupt to another. Their implementation is light weight, making them particularly suited for interrupt to task and core to core communication scenarios.

NOTE Other FreeRTOS objects, such as queues, semaphores and event groups, are written to allow multiple writers and multiple readers. Uniquely among FreeRTOS objects, the implementation of stream buffers (message buffers are built on top of stream buffers) assumes there is only one writer and only one reader. THEREFORE, UNLIKE OTHER FREERTOS OBJECTS, THE IMPLEMENTATION OF THE STREAM AND MESSAGE BUFFER API FUNCTIONS DO NOT CONTAIN CRITICAL SECTIONS TO PROTECT AGAINST CONCURRENT ACCESS. If multiple tasks or interrupts write to the same message buffer then the application writer must place the API calls that write to the message buffer inside a critical section. Likewise, if multiple tasks or interrupts read from the same message buffer then the application writer must place the API calls that read read from the message buffer inside a critical section. Critical sections are not required if there is only a single reader and a single writer even if the reader and writer are different tasks or interrupts.

Message buffers hold variable length messages. To enable that, when a message is written to the message buffer an additional sizeof(size_t) bytes are also written to store the message's length (that happens internally, with the API function). sizeof(size_t) is typically 4 bytes on a 32-bit architecture, so writing a 10 byte message to a message buffer on a 32-bit architecture will actually reduce the available space in the message buffer by 14 bytes (10 byte are used by the message, and 4 bytes to hold the length of the message).

xMessageBufferCreate()

The following shows the xMessageBufferCreate() function prototype.

```
#include "FreeRTOS.h"

#include "message_buffer.h"

MessageBufferHandle_t xMessageBufferCreate( size_t xBufferSizeBytes );
```

Summary

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).

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.

Return Value

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

```
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.
    }
```

xMessageBufferCreateStatic()

The following shows the xMessageBufferCreateStatic() function prototype.

```
#include "FreeRTOS.h"

#include "message_buffer.h"

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

Summary

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.

Return Value

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

```
// 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.
}
```

xMessageBufferSend()

The following shows the xMessageBufferSend() function prototype.

```
#include "FreeRTOS.h"

#include "message_buffer.h"

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

Summary

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 implementation of stream buffers (message buffers are built on top of stream buffers) assumes there is only one writer and only one reader. THEREFORE, UNLIKE OTHER FREERTOS OBJECTS, THE IMPLEMENTATION OF THE STREAM AND MESSAGE BUFFER API FUNCTIONS DO NOT CONTAIN CRITICAL SECTIONS TO PROTECT AGAINST CONCURRENT ACCESS. If multiple tasks or interrupts write to the same message buffer then the application writer must place the API calls that write to the message buffer inside a critical section.

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 task should remain in the Blocked state to wait for enough space to become available in the message buffer, should the message buffer contain too little space to hold

the another xDataLengthBytes bytes. xMessageBufferSend() will return immediately if xTicksToWait is zero and the amount of space in the message buffer is less than xDataLengthBytes. 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.

Return Value

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 will be returned.

Example

```
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 ), x100ms );

    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( pcStringToSend ), 0 );

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

xMessageBufferSendFromISR()

The following shows the xMessageBufferSendFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "message_buffer.h"

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

Summary

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 implementation of stream buffers (message buffers are built on top of stream buffers) assumes there is only one writer and only one reader. THEREFORE, UNLIKE OTHER FREERTOS OBJECTS, THE IMPLEMENTATION OF THE STREAM AND MESSAGE BUFFER API FUNCTIONS DO NOT CONTAIN CRITICAL SECTIONS TO PROTECT AGAINST CONCURRENT ACCESS. If multiple tasks or interrupts write to the same message buffer then the application writer must place the API calls that write to the message buffer inside a critical section.

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.

Return Value

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 will returned, otherwise xDataLengthBytes will be returned.

Example

```
// 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 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 );
}
```

xMessageBufferReceive()

The following shows the xMessageBufferReceive() function prototype.

```
#include "FreeRTOS.h"

#include "message_buffer.h"

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

Summary

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 implementation of stream buffers (message buffers are built on top of stream buffers) assumes there is only one writer and only one reader. THEREFORE, UNLIKE OTHER FREERTOS OBJECTS, THE IMPLEMENTATION OF THE STREAM AND MESSAGE BUFFER API FUNCTIONS DO NOT CONTAIN CRITICAL SECTIONS TO PROTECT AGAINST CONCURRENT ACCESS. If multiple tasks or interrupts write to the same message buffer then the application writer must place the API calls that write to the message buffer inside a critical section.

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 pucRxData 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.

Return Value

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

```
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....
    }
}
```

xMessageBufferReceiveFromISR()

The following shows the xMessageBufferReceiveFromISR() function prototype.

```
#include "FreeRTOS.h"

#include "message_buffer.h"

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

Summary

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.

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 pucRxData 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.

Return Value

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

Example

```
// 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 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 );
}
```

vMessageBufferDelete()

The following shows the vMessageBufferDelete() function prototype.

```
#include "FreeRTOS.h"

#include "message_buffer.h"

void vMessageBufferDelete( MessageBufferHandle_t xMessageBuffer );
```

Summary

Deletes a message buffer that was previously created using a call to xMessageBufferCreate() or xMessageBufferCreateStatic(). If the message buffer was created using dynamic memory (that is, by xMessageBufferCreate()), then the allocated memory is freed.

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

Parameters

xMessageBuffer

The handle of the message buffer to be deleted.

xMessageBufferIsFull()

The following shows the xMessageBufferIsFull() function prototype.

```
#include "FreeRTOS.h"

#include "message_buffer.h"

BaseType_t xMessageBufferIsFull( MessageBufferHandle_t xMessageBuffer ) );
```

Summary

Tests to see if a message buffer is full. A message buffer is full if it cannot accept any more messages, of any size, until space is made available by a message being removed from the message buffer.

Parameters

xMessageBuffer

The handle of the message buffer being queried.

Return Value

If the message buffer referenced by xMessageBuffer is full then pdTRUE is returned. Otherwise pdFALSE is returned.

xMessageBufferIsEmpty()

The following shows the xMessageBufferIsEmpty() function prototype.

```
#include "FreeRTOS.h"

#include "message_buffer.h"

BaseType_t xMessageBufferIsEmpty( MessageBufferHandle_t xMessageBuffer ) );
```

Summary

Tests to see if a message buffer is empty (does not contain any messages).

Parameters

xMessageBuffer

The handle of the message buffer being queried.

Return Value

If the message buffer referenced by xMessageBuffer is empty then pdTRUE is returned. Otherwise pdFALSE is returned.

xMessageBufferReset()

The following shows the xMessageBufferReset() function prototype.

```
#include "FreeRTOS.h"

#include "message_buffer.h"

BaseType_t xMessageBufferReset( MessageBufferHandle_t xMessageBuffer );
```

Summary

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.

Return Value

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 message to be available, then pdFAIL is returned.

xMessageBufferSpaceAvailable()

The following shows the xMessageBufferSpaceAvailable() function prototype.

```
#include "FreeRTOS.h"

#include "message_buffer.h"

size_t xMessageBufferSpaceAvailable( MessageBufferHandle_t xMessageBuffer ) );
```

Summary

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

Parameters

xMessageBuffer

The handle of the message buffer being queried.

Return Value

The free space in the message buffer, in bytes.