The FreeRTOS™ Reference Manual

The FreeRTOS™ Reference Manual

API Functions and Configuration Options

Richard Barry

Version 1.2.1.

http://www.FreeRTOS.org

Contents

List of Figures

List of Code Listings

List of Tables

List of Notation

API	Application Programming Interface
ISR	Interrupt Service Routine
MPU	Memory Protection Unit
RTOS	Real-time Operating System

Chapter 1

About This Manual

1.1 Scope

This document provides a technical reference to both the primary FreeRTOS API[1], and the FreeRTOS kernel configuration options. It is assumed the reader is already familiar with the concepts of writing multi tasking applications, and the primitives provided by real time kernels. Readers that are not familiar with these fundamental concepts are recommended to read the book **"Using the FreeRTOS Real Time Kernel – A Practical Guide"** for a much more descriptive, hands on, and tutorial style text. The book can be obtained from http://www.FreeRTOS.org/Documentation.

The Order in Which Functions Appear in This Manual

Within this document, the API functions have been split into four groups – task and scheduler related functions, queue related functions, semaphore related functions, and software timer related functions. Each group is documented in its own chapter, and within each chapter, the API functions are listed in alphabetical order. Note however that the name of each API function is prefixed with one or more letters that specify the function's return type, and the alphabetical ordering of API functions within each chapter ignores the function return type prefix. APPENDIX 1: describes the prefixes in more detail.

As an example, consider the API function that is used to create a FreeRTOS task. Its name is vTaskCreate(). The 'v' prefix specifies that vTaskCreate() is a void function (it does not return a value). The secondary 'Task' prefix specifies that the function is a task related function, and, as such, will be documented in the chapter that contains task and scheduler related functions. The 'v' is not considered in the alphabetical ordering, so vTaskCreate() will appear in the task and scheduler chapter ordered as if its name was just TaskCreate().

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

[1] The 'alternative' API is not included as its use is no longer recommended. The co-routine API is also omitted as co-routines are only useful to a small subset of applications.

(hardware) priority above the priority set by the configMAX_SYSCALL_INTERRUPT_PRIORITY kernel configuration constant (described in section 6.1 of this document). 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.

Chapter 2

Task and Scheduler API

2.1 portSWITCH_TO_USER_MODE()

```
#include “FreeRTOS.h”
#include “task.h”

void portSWITCH_TO_USER_MODE( void );
```

Listing 1 portSWITCH_TO_USER_MODE() macro prototype

Summary

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

MPU restricted tasks are created using xTaskCreateRestricted(). The parameters supplied to xTaskCreateRestricted() specify whether the task being created should be a User (un-privileged) 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 portSWTICH_TO_USER_MODE() that permits a task to convert itself from a User mode into a Supervisor mode task.

2.2 xTaskAllocateMPURegions()

```
#include "FreeRTOS.h"
#include "task.h"

void vTaskAllocateMPURegions( xTaskHandle xTaskToModify,
                              const xMemoryRegion * const xRegions );
```

Listing 2 vTaskAllocateMPURegions() function prototype

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 and is only relevant to FreeRTOS MPU ports (FreeRTOS ports that make use of a Memory Protection Unit).

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 run time using the xTaskAllocateMPURegions() 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 xMemoryRegion 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 xMemoryRegion 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 xMemoryRegion structure shown in Listing 3.

```
typedef struct xMEMORY_REGION
{
    void *pvBaseAddress;
    unsigned long ulLengthInBytes;
    unsigned long ulParameters;
} xMemoryRegion;
```

Listing 3 The data structures used by xTaskCreateRestricted()

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

void vATask( void *pvParameters )
{
    /* This task was created 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). */
}
```

Listing 4 Example use of xTaskCallApplicationTaskHook()

2.3 xTaskCallApplicationHook()

```
#include "FreeRTOS.h"
#include "task.h"

portBASE_TYPE xTaskCallApplicationTaskHook( xTaskHandle xTask, void *pvParameters );
```

Listing 5 xTaskCallApplicationTaskHook() function prototype

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 itself 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 being used to output debug trace information.

Task hook functions must have the prototype demonstrated by Listing 6.

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

Listing 6 The prototype to which all task hook functions must conform

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. See the pxCreatedTask parameter of the xTaskCreate() API function for information on obtaining handles to tasks. 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 6 */
static portBASE_TYPE 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,
    its 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 )
```

Listing 7 Example use of xTaskCallApplicationTaskHook()

2.4 xTaskCreate()

```
#include "FreeRTOS.h"
#include "task.h"

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

Listing 8 xTaskCreate() function prototype

Summary

Creates a new instance of a 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

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 not used by FreeRTOS in any way. It is included purely as a debugging aid. Identifying a task by a human readable name is much simpler than attempting to identify it by its handle. The application-defined constant configMAX_TASK_NAME_LEN defines the maximum length a task name can take – 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, 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. 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), but you should use the lowest number of priorities required, to avoid wasting RAM. 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 or heap_2.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.

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. */
typdef 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 )
{
xTaskHandle 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 as there was insufficient heap memory remaining. If
        heap_1.c or heap_2.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 );
    }
}
```

Listing 9 Example use of xTaskCreate()

2.5 xTaskCreateRestricted()

```
#include "FreeRTOS.h"
#include "task.h"

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

Listing 10 xTaskCreateRestricted() function prototype

Summary

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

Create a new Memory Protection Unit (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 under the notes heading in this section of the reference manual
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.
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 or heap_2.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

xTaskCreateRestricted() makes use of the two data structures shown in Listing 11.

```
typedef struct xTASK_PARAMTERS
{
    pdTASK_CODE pvTaskCode;
    const signed char * const pcName;
    unsigned short usStackDepth;
    void *pvParameters;
    unsigned portBASE_TYPE uxPriority;
    portSTACK_TYPE *puxStackBuffer;
    xMemoryRegion xRegions[ portNUM_CONFIGURABLE_REGIONS ];
} xTaskParameters;

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

typedef struct xMEMORY_REGION
{
    void *pvBaseAddress;
    unsigned long ulLengthInBytes;
    unsigned long ulParameters;
} xMemoryRegion;
```

Listing 11 The data structures used by xTaskCreateRestricted()

A description of the Listing 11 structure members is given below.

pvTaskCode to 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 (un-privileged) 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 portPRIVILEGE_BIT 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 means that the xTaskCreateRestricted() function cannot do the same, and 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 re-configured 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));
```

Listing 12 Statically declaring a correctly aligned stack for use by a restricted task

xMemoryRegions	An array of xMemoryRegion structures. Each xMemoryRegion 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 run time 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, 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

```
/* 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, and not write to it. */
char cReadOnlyArray[ 512 ] __attribute__((aligned(512)));

/* Fill in a xTaskParameters structure to define the task - this is the structure passed to the
xTaskCreateRestricted() function. */
static const xTaskParameters 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 as a
    task handle is not required. */
    xTaskCreateRestricted( &xTaskDefinition, NULL );

    /* Start the scheduler. */
    vTaskStartScheduler();

    /* Should not reach here! */
}
```

Listing 13 Example use of xTaskCreateRestricted()

2.6 vTaskDelay()

```
#include "FreeRTOS.h"
#include "task.h"

void vTaskDelay( portTickType xTicksToDelay );
```

Listing 14 vTaskDelay() function prototype

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 constant portTICK_RATE_MS can be used to convert milliseconds into ticks. This is demonstrated in the example in this section.

Return Values

None.

Example

```
void vAnotherTask( void * pvParameters )
{
    for( ;; )
    {
        /* Perform some processing here. */

        …

        /* Enter the Blocked state for 20 tick interrupts – the actual time spent
        in the Blocked state is dependent 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
        portTICK_RATE_MS constant means the tick frequency can change without
        effecting the time spent in the blocked state (other than due to the
        resolution of the tick frequency). */
        vTaskDelay( 20 / portTICK_RATE_MS );
    }
}
```

Listing 15 Example use of vTaskDelay()

2.7 vTaskDelayUntil()

```
#include "FreeRTOS.h"
#include "task.h"

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

Listing 16 vTaskDelayUntil() function prototype

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 into the Blocked state, and then 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 into the Blocked state, 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 constant portTICK_RATE_MS can be used to convert milliseconds to ticks.

Return Values

None.

Example

```
/* Define a task that performs an action every 50 milliseconds. */
void vCyclicTaskFunction( void * pvParameters )
{
portTickType xLastWakeTime;
const portTickType xPeriod = ( 50 / portTICK_RATE_MS );

    /* 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 portTICK_RATE_MS constant 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. */
    }
}
```

Listing 17 Example use of vTaskDelayUntil()

2.8 vTaskDelete()

```
#include "FreeRTOS.h"
#include "task.h"

void vTaskDelete( xTaskHandle pxTask );
```

Listing 18 vTaskDelete() function prototype

Summary

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

Deleted tasks no longer exist so 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) – see the pxCreatedTask parameter of the xTaskCreate() API function for information on obtaining handles to tasks. A task can delete itself by passing NULL in place of a valid task handle.

Return Values

None.

Example

```
void vAnotherFunction( void )
{
xTaskHandle 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 );
}
```

Listing 19 Example use of the vTaskDelete()

2.9 taskDISABLE_INTERRUPTS()

```
#include "FreeRTOS.h"
#include "task.h"

void taskDISABLE_INTERRUPTS( void );
```

Listing 20 taskDISABLE_INTERRUPTS() macro prototype

Summary

If the FreeRTOS port being used does not make use of the configMAX_SYSCALL_INTERRUPT_PRIORITY 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. It is not recommended to call FreeRTOS API functions when interrupts have already been disabled.

Parameters

None.

Return Values

None.

2.10 taskENABLE_INTERRUPTS()

```
#include “FreeRTOS.h”
#include “task.h”

void taskENABLE_INTERRUPTS( void );
```

Listing 21 taskENABLE_INTERRUPTS() macro prototype

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. It is not recommended to call FreeRTOS API functions when interrupts have already been disabled.

Parameters

None.

Return Values

None.

2.11 vTaskEndScheduler()

```
#include "FreeRTOS.h"
#include "task.h"

void vTaskEndScheduler( void );
```

Listing 22 vTaskEndScheduler() function prototype

Summary

At the time of writing, vTaskEndScheduler() is only implemented for the real mode x86 FreeRTOS port. It stops the kernel tick interrupt, deletes all the tasks, then resumes from the point where vTaskStartScheduler() was called.

vTaskEndScheduler() requires an exit function to be defined in the FreeRTOS portable layer (see vPortEndScheduler() in port.c for the real mode x86 PC port) to perform hardware specific operations such as stopping the tick interrupt.

Parameters

None.

Return Values

None.

2.12 ulTaskEndTrace()

```
#include "FreeRTOS.h"
#include "task.h"

unsigned long ulTaskEndTrace( void );
```

Listing 23 ulTaskEndTrace() function prototype

Summary

ulTaskEndTrace() relates to a deprecated kernel trace feature. The deprecated trace feature has been replaced by the trace macros.

FreeRTOS contains a simple, and now deprecated, execution trace facility. This writes the number of the task selected to enter the Running state and a time stamp to a RAM buffer each time a context switch occurs. The RAM buffer can then be converted, offline, to a plain text file suitable for viewing in a spread sheet program.

A trace log is started using vTaskStartTrace() and ended using ulTaskEndTrace().

Parameters

None.

Return Values

ulTaskEndTrace() returns the number of bytes that were written into the RAM buffer while the trace was running.

2.13 taskENTER_CRITICAL()

```
#include "FreeRTOS.h"
#include "task.h"

void taskENTER_CRITICAL( void );
```

Listing 24 taskENTER_CRITICAL macro prototype

Summary

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

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.

Example

```
/* 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();


        /* Perform whatever functionality required 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();
    }
}
```

Listing 25 Example use of taskENTER_CRITICAL() and taskEXIT_CRITICAL()

2.14 taskEXIT_CRITICAL()

```
#include "FreeRTOS.h"
#include "task.h"

void taskEXIT_CRITICAL( void );
```

Listing 26 taskEXIT_CRITICAL() macro prototype

Summary

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

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.

Example

See Listing 25.

2.15 xTaskGetApplicationTaskTag()

```
#include "FreeRTOS.h"
#include "task.h"

pdTASK_HOOK_CODE xTaskGetApplicationTaskTag( xTaskHandle xTask );
```

Listing 27 xTaskGetApplicationTaskTag() function prototype

Summary

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.

This function is intended for advanced users only.

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 vTaskGetApplicationTaskTag() 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 )
{
xTaskHandle xHandle;
int iReturnedTaskHandle;

    /* 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. */
        iReturnedTaskHandle = ( int ) xTaskGetApplicationTaskTag();
    }
}
```

Listing 28 Example use of xTaskGetApplicationTaskTag()

2.16 xTaskGetCurrentTaskHandle()

```
#include "FreeRTOS.h"
#include "task.h"

xTaskHandle xTaskGetCurrentTaskHandle( void );
```

Listing 29 xTaskGetCurrentTaskHandle() function prototype

Summary

Returns the handle of the task that is in the Running state – which 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.

2.17 uxTaskGetHighWaterMark()

```
#include "FreeRTOS.h"
#include "task.h"

unsigned portBASE_TYPE uxTaskGetStackHighWaterMark( xTaskHandle 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). See the pxCreatedTask parameter of the xTaskCreate() API function for information on obtaining handles to tasks. 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. It is therefore recommended that its use is limited 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 )
{
unsigned portBASE_TYPE 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 );
    }
}
```

Listing 30 Example use of uxTaskGetStackHighWaterMark()

2.18 xTaskGetIdleTaskHandle()

```
#include "FreeRTOS.h"
#include "task.h"

xTaskHandle xTaskGetIdleTaskHandle( void );
```

Listing 31 xTaskGetIdleTaskHandle() function prototype

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.

2.19 uxTaskGetNumberOfTasks()

```
#include "FreeRTOS.h"
#include "task.h"

unsigned portBASE_TYPE uxTaskGetNumberOfTasks( void );
```

Listing 32 uxTaskGetNumberOfTasks() function prototype

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.

2.20 vTaskGetRunTimeStats()

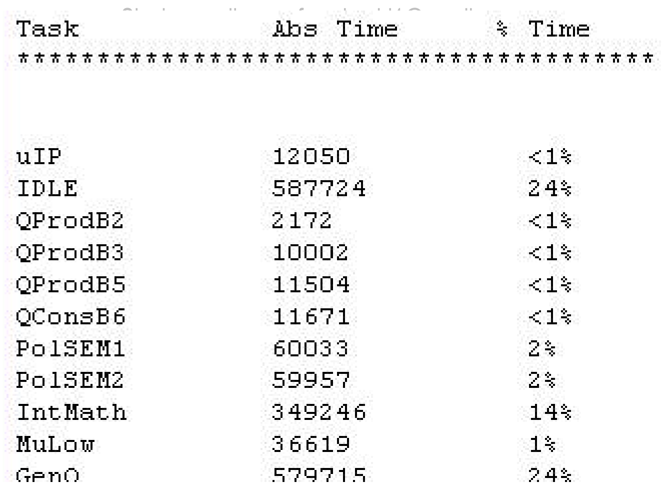
```
#include "FreeRTOS.h"
#include "task.h"

void vTaskGetRunTimeStats( char *pcWriteBuffer );
```

Listing 33 vTaskGetRunTimeStats() function prototype

Summary

FreeRTOS can be configured to collect task run time statistics. Task run time statistics provide information on the amount of processing time each task has received. Figures are provided as both an absolute time and a percentage of the total application run time. The vTaskGetRunTimeStats() API function formats the collected run time 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. An example output is shown in Figure 1.



```
Task            Abs Time       % Time
****************************************


uIP             12050          <1%
IDLE            587724         24%
QProdB2         2172           <1%
QProdB3         10002          <1%
QProdB5         11504          <1%
QConsB6         11671          <1%
PolSEM1         60033          2%
PolSEM2         59957          2%
IntMath         349246         14%
MuLow           36619          1%
GenQ            579715         24%
```

Figure 1 An example of the table produced by calling vTaskGetRunTimeStats()

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, as no boundary checking is performed.

Return Values

None.

Notes

configGENERATE_RUN_TIME_STATS must 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 run time stats must have a higher resolution than the tick interrupt, otherwise the gathered statistics may be too inaccurate to be truly useful. It is recommended to 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
```

Listing 34 Example macro definitions, taken from the LM3Sxxx Eclipse Demo

```
/* 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
```

Listing 35 Example macro definitions, taken from the LPC17xx Eclipse Demo

```
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, hence 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. */
}
```

Listing 36 Example use of vTaskGetRunTimeStats()

2.21 xTaskGetSchedulerState()

```
#include "FreeRTOS.h"
#include "task.h"

portBASE_TYPE xTaskGetSchedulerState( void );
```

Listing 37 xTaskGetSchedulerState() function prototype

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 sate because vTaskSuspendAll() was called.

Notes

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

2.22 pcTaskGetTaskName()

```
#include "FreeRTOS.h"
#include "task.h"

signed char * pcTaskGetTaskName( xTaskHandle xTaskToQuery );
```

Listing 38 pcTaskGetTaskName() function prototype

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() API function call used to create the task.

Parameters

xTaskToQuery	The handle of the task being queried (the subject task). The handle to a task is obtained using the pxCreatedTask parameter of the xTaskCreate() API function call used to create the task. 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 tasks name.

Notes

INCLUDE_pcTaskGetTaskName must be set to 1 in FreeRTOSConfig.h for pcTaskGetTaskName() to be available.

2.23 xTaskGetTickCount()

```
#include "FreeRTOS.h"
#include "task.h"

portTickType xTaskGetTickCount( void );
```

Listing 39 xTaskGetTickCount() function prototype

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 one tick period represents depends on the value assigned to configTICK_RATE_HZ within FreeRTOSConifg.h. The constant portTICK_RATE_MS can be used to convert a time in milliseconds to a time in 'ticks'.

The tick count will eventually overflow and return to zero. This will not effect 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 however 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 )
{
portTickType 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? */
    xExectutionTime = xTime2 - xTime1;
}
```

Listing 40 Example use of xTaskGetTickCount()

2.24 xTaskGetTickCountFromISR()

```
#include "FreeRTOS.h"
#include "task.h"

portTickType xTaskGetTickCountFromISR( void );
```

Listing 41 xTaskGetTickCountFromISR() function prototype

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 one tick period represents depends on the value assigned to configTICK_RATE_HZ within FreeRTOSConifg.h. The constant portTICK_RATE_MS can be used to convert a time in milliseconds to a time in 'ticks'.

The tick count will eventually overflow and return to zero. This will not effect 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 however 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 portTickType xTimeISRLastExecuted = 0;
portTickType 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;
}
```

Listing 42 Example use of xTaskGetTickCountFromISR()

2.25 xTaskIsTaskSuspended()

```
#include "FreeRTOS.h"
#include "task.h"

portBASE_TYPE xTaskIsTaskSuspended( xTaskHandle xTask );
```

Listing 43 xTaskIsTaskSuspended() function prototype

Summary

Determines whether a task is currently in the Suspended state.

Parameters

xTask	The handle of the task being queried (the subject task). The handle to a task is obtained using the pxCreatedTask parameter to the xTaskCreate() API function when the task is created.

Return Values

pdTRUE	The task being queried was in the Suspended state at the time xTaskIsTaskSuspended() was called.
pdFALSE	The task being queried was not in the Suspended state at the time xTaskIsTaskSuspended() was called.

Notes

INCLUDE_vTaskSuspend must be set to 1 in FreeRTOSConfig.h for xTaskIsTaskSuspended() to be available.

2.26 vTaskList()

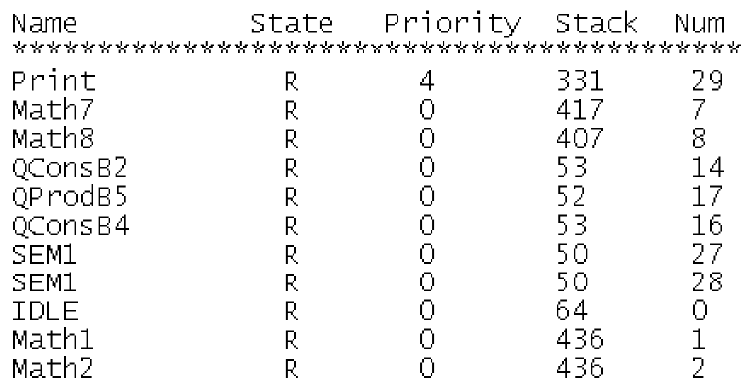
```
#include “FreeRTOS.h”
#include “task.h”

void vTaskList( signed char *pcWriteBuffer );
```

Listing 44 vTaskList() function prototype

Summary

Creates a human readable table in a character buffer that describes the state of each task at the time vTaskList() was called. An example is shown in Figure 2.



```
Name          State   Priority  Stack  Num
*******************************************
Print           R       4        331    29
Math7           R       0        417    7
Math8           R       0        407    8
QConsB2         R       0        53     14
QProdB5         R       0        52     17
QConsB4         R       0        53     16
SEM1            R       0        50     27
SEM1            R       0        50     28
IDLE            R       0        64     0
Math1           R       0        436    1
Math2           R       0        436    2
```

Figure 2 An example of the table produced by calling vTaskList()

The table includes the following information:

- Name – This is the name given to the task when the task was created.
- State – The state of the task at the time vTaskList() was called, as follows:
    - ‘B’ if the task is in the Blocked state.
    - ‘R’ if the task is in the Ready state.
    - ‘S’ if the task is in the Suspended state.
    - ‘D’ if the task has been deleted, but the idle task has not yet freed the memory that was being used by the task to hold its data structures and stack.
- Priority – The priority assigned to the task at the time vTaskList() was called.

- Stack – Shows the ‘high water mark’ of the task’s stack. This is the minimum amount of free stack that has been available during the lifetime of the task. The closer this value is to zero, the closer the task has come to overflowing its stack.

- Num – This is a unique number that is assigned to each task. It has no purpose other than to help identify tasks when more than one task has been assigned the same name.

Parameters

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

Return Values

None.

Notes

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, INCLUDE_vTaskDelete and INCLUDE_vTaskSuspend must all be set to 1 within 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 minimise the code size impact. Note that printf-stdarg.c is licensed separately to FreeRTOS. Its license terms are contained in the file itself.

Tasks that are in the Blocked state, but did not specifying a time out, (they have blocked indefinately to wait for an event) will be shown as being in the Suspended state.

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, hence 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. */
}
```

Listing 45 Example use of vTaskList()

2.27 uxTaskPriorityGet()

```
#include "FreeRTOS.h"
#include "task.h"

unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle pxTask );
```

Listing 46 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). The handle to a task is obtained using the pxCreatedTask parameter to the xTaskCreate() API function when the task is created. A task may 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

```
void vAFunction( void )
{
xTaskHandle xHandle;
unsigned portBASE_TYPE 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. */
        }
    }
}
```

Listing 47 Example use of uxTaskPriorityGet()

2.28 vTaskPrioritySet()

```
#include "FreeRTOS.h"
#include "task.h"

void vTaskPrioritySet( xTaskHandle pxTask, unsigned portBASE_TYPE uxNewPriority );
```

Listing 48 vTaskPrioritySet() function prototype

Summary

Changes the priority of a task.

Parameters

pxTask	The handle of the task being modified (the subject task). The handle of a task is obtained using the pxCreatedTask parameter of the xTaskCreate() API function. 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, etc. 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 re-evaluated.

Example

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

Listing 49 Example use of vTaskPrioritySet()

2.29 vTaskResume()

```
#include "FreeRTOS.h"
#include "task.h"

void vTaskResume( xTaskHandle pxTaskToResume );
```

Listing 50 vTaskResume() function prototype

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. The handle of a task is obtained using the pxCreatedTask parameter of the xTaskCreate() API function.

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(), 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 mean time) 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

```
void vAFunction( void )
{
xTaskHandle 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. */
    }
}
```

Listing 51 Example use of vTaskResume()

2.30 xTaskResumeAll()

```
#include "FreeRTOS.h"
#include "task.h"

portBASE_TYPE xTaskResumeAll( void );
```

Listing 52 xTaskResumeAll() function prototype

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 (un-suspended).

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

```
/* 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 which is also
        accessed from another task (but not from an interrupt).  It cannot use
        taskENTER_CRITICAL()/taskEXIT_CRITICAL() as the length of the operation may
        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
        as the task has all the processing time other than that utilized 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(). */
        }
    }
}
```

Listing 53 Example use of xTaskResumeAll()

2.31 xTaskResumeFromISR()

```
#include "FreeRTOS.h"
#include "task.h"

portBASE_TYPE xTaskResumeFromISR( xTaskHandle pxTaskToResume );
```

Listing 54 xTaskResumeFromISR() function prototype

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. The handle of a task is obtained using the pxCreatedTask parameter of the xTaskCreate() API function.

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 that 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 (un-suspend) 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 of more calls to vTaskSuspend() will always be un-suspended 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

```
xTaskHandle 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
        (un-suspends) it. */
    }
}

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

Listing 55 Example use of vTaskResumeFromISR()

2.32 vTaskSetApplicationTaskTag()

```
#include "FreeRTOS.h"
#include "task.h"

void vTaskSetApplicationTaskTag( xTaskHandle xTask, pdTASK_HOOK_CODE pxTagValue );
```

Listing 56 vTaskSetApplicationTaskTag() function prototype

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 itself 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 pdTASK_HOOK_CODE to permit a function pointer to be assigned to the tag, although, 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

```
/* 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 - this must have type pdTASK_HOOK_CODE, as per this example. */
static portBASE_TYPE prvExampleTaskHook( void * pvParameter )
{
    /* Perform some action - this could be anything from logging a value, updating
    the task state, outputting a value, etc. */

    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 )
```

Listing 57 Example use of vTaskSetApplicationTaskTag()

2.33 vTaskStartScheduler()

```
#include "FreeRTOS.h"
#include "task.h"

void vTaskStartScheduler( void );
```

Listing 58 vTaskStartScheduler() function prototype

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 ever 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

```
xTaskHandle 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( ;; );
}
```

Listing 59 Example use of vTaskStartScheduler()

2.34 vTaskSuspend()

```
#include "FreeRTOS.h"
#include "task.h"

void vTaskSuspend( xTaskHandle pxTaskToSuspend );
```

Listing 60 vTaskSuspend() function prototype

Summary

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

The only way of removing 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. The handle of a task is obtained using the pxCreatedTask parameter of the xTaskCreate() API function. A task may suspend itself by passing NULL in place of a valid task handle.

Return Values

None.

Notes

If FreeRTOS version 6.1.0 or later is being used, 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

```
void vAFunction( void )
{
xTaskHandle 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 (un-suspended) it using a call to vTaskResume(). */
    }
}
```

Listing 61 Example use of vTaskSuspend()

2.35 vTaskSuspendAll()

```
#include "FreeRTOS.h"
#include "task.h"

void vTaskSuspendAll( void );
```

Listing 62 vTaskSuspendAll() function prototype

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 (un-suspended).

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

```
/* 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() as the length of the operation may
        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 as
        the task has all the processing time other than that utilized by interrupt
        service routines.*/


        /* Calls to vTaskSuspendAll() can be nested so it is safe to call a (non API)
        function which 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(). */
        }
    }
}
```

Listing 63 Example use of vTaskSuspendAll()

2.36 taskYIELD()

```
#include “FreeRTOS.h”
#include “task.h”

void taskYIELD( void );
```

Listing 64 taskYIELD() macro prototype

Summary

Yield to another task of equal priority.

Yielding is where a task volunteers to leave the Running state, without being pre-empted, and before its time slice has been fully utilized.

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 (prior to the scheduler being 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 itself be transitioned straight back 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

```
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. */
    }
}
```

Listing 65 Example use of taskYIELD()

Chapter 3

Queue API

3.1 vQueueAddToRegistry()

```
#include "FreeRTOS.h"
#include "queue.h"

void vQueueAddToRegistry( xQueueHandle xQueue, signed char *pcQueueName );
```

Listing 66 vQueueAddToRegistry() function prototype

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 purely as a debugging aid. 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 the queues and semaphores that need to be viewed in a kernel aware debugging interface need to be registered.

The queue registry is only required 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 automatically remove it from the registry.

Example

```
void vAFunction( void )
{
xQueueHandle xQueue;

    /* Create a queue big enough to hold 10 chars. */
    xQueue = xQueueCreate( 10, sizeof( char ) );

    /* The created queue needs to be viewable in a kernel aware debugger, so
    add it to the registry. */
    vQueueAddToRegistry( xQueue, "AMeaningfulName" );
 }
```

Listing 67 Example use of vQueueAddToRegistry()

3.2 xQueueCreate()

```
#include "FreeRTOS.h"
#include "queue.h"

xQueueHandle xQueueCreate( unsigned portBASE_TYPE uxQueueLength,
                           unsigned portBASE_TYPE uxItemSize );
```

Listing 68 xQueueCreate() function prototype

Summary

Creates a queue. A queue must be explicitly created before it can be used.

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 should be stored as the handle to 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.

Example

```c
/* 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 )
{
xQueueHandle 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. */
}
```

Listing 69 Example use of xQueueCreate()

3.3 vQueueDelete()

```
#include "FreeRTOS.h"
#include "queue.h"

void vQueueDelete( xTaskHandle pxQueueToDelete );
```

Listing 70 vQueueDelete() function prototype

Summary

Deletes a queue that was previously created using a call to xQueueCreate(). 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

```
/* 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 )
{
xQueueHandle 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 );
    }
}
```

Listing 71 Example use of vQueueDelete()

3.4 xQueueIsQueueEmptyFromISR()

```
#include "FreeRTOS.h"
#include "queue.h"

portBASE_TYPE xQueueIsQueueEmptyFromISR( const xQueueHandle pxQueue );
```

Listing 72 xQueueIsQueueEmptyFromISR() function prototype

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 xQueueIsQueueEmtpyFromISR() was called.

Notes

None.

3.5 xQueueIsQueueFullFromISR()

```
#include "FreeRTOS.h"
#include "queue.h"

portBASE_TYPE xQueueIsQueueFullFromISR( const xQueueHandle pxQueue );
```

Listing 73 xQueueIsQueueFullFromISR() function prototype

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.

3.6 uxQueueMessagesWaiting()

```
#include "FreeRTOS.h"
#include "queue.h"

unsigned portBASE_TYPE uxQueueMessagesWaiting( const xQueueHandle xQueue );
```

Listing 74 uxQueueMessagesWaiting() function prototype

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

```
void vAFunction( xQueueHandle xQueue )
{
unsigned portBASE_TYPE uxNumberOfItems;

    /* How many items are currently in the queue referenced by the xQueue handle? */
    uxNumberOfItems = uxQueueMessagesWaiting( xQueue );
}
```

Listing 75 Example use of uxQueueMessagesWaiting()

3.7 uxQueueMessagesWaitingFromISR()

```
#include “FreeRTOS.h”
#include “queue.h”

unsigned portBASE_TYPE uxQueueMessagesWaitingFromISR( const xQueueHandle xQueue );
```

Listing 76 uxQueueMessagesWaitingFromISR() function prototype

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

```
void vAnInterruptHandler( void )
{
unsigned portBASE_TYPE uxNumberOfItems;
portBASE_TYPE 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. */
}
```

Listing 77 Example use of uxQueueMessagesWaitingFromISR()

3.8 xQueuePeek()

```
#include "FreeRTOS.h"
#include "queue.h"

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

Listing 78 xQueuePeek() function prototype

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() 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 is dependent on the tick frequency. The constant portTICK_RATE_MS can be used to convert a time specified in milliseconds to 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.

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

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

xQueueHandle 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( ;; )
    {
    }
}
```

Listing 79 Example use of xQueuePeek()

3.9 xQueueReceive()

```
#include "FreeRTOS.h"
#include "queue.h"

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

Listing 80 xQueueReceive() function prototype

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() 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() 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 dependent on the tick frequency. The constant portTICK_RATE_MS can be used to convert a time specified in milliseconds to 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.

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

```
/* 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 )
{
xQueueHandle 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 )
{
xQueueHandle 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 = ( xQueueHandle ) 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. */
        }
    }
}
```

Listing 81 Example use of xQueueReceive()

3.10 xQueueReceiveFromISR()

```
#include "FreeRTOS.h"
#include "queue.h"

portBASE_TYPE xQueueReceiveFromISR(  xQueueHandle xQueue,
                                     void *pvBuffer,
                                     portBASE_TYPE *pxHigherPriorityTaskWoken
                                   );
```

Listing 82 xQueueReceiveFromISR() function prototype

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() 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() 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 *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.

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 itself perform a context switch. It will instead just indicate whether or not a context switch is required.

xQueueReceiveFromISR() must not be called prior to the scheduler being started. Therefore an interrupt that calls xQueueReceiveFromISR() must not be allowed to execute prior to the scheduler being 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 preferable approach would be 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;
portBASE_TYPE 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 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. */
    taskYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Listing 83 Example use of xQueueReceiveFromISR()

3.11 xQueueSend(), xQueueSendToFront(), xQueueSendToBack()

```
#include "FreeRTOS.h"
#include "queue.h"

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

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

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

Listing 84 xQueueSend(), xQueueSendToFront() and xQueueSendToBack() function prototypes

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() used to create the queue
pvItemToQueue	A pointer to the data to be copied into the queue. The size of each item that the queue can hold is set when the queue is created, so this number of 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 dependent on the tick frequency. The constant portTICK_RATE_MS can be used to convert a time specified in milliseconds to 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.

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

```
/* 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 )
{
xQueueHandle 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 )
{
xQueueHandle xQueue;
AMessage xMessage;

    /* The queue handle is passed into this task as the task parameter.  Cast
    the parameter back to a queue handle. */
    xQueue = ( xQueueHandle ) 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. */
        }
    }
}
```

Listing 85 Example use of xQueueSendToBack()

3.12 xQueueSendFromISR(), xQueueSendToBackFromISR(), xQueueSendToFrontFromISR()

```
#include "FreeRTOS.h"
#include "queue.h"

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

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

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

Listing 86 xQueueSendFromISR(), xQueueSendToBackFromISR() and xQueueSendToFrontFromISR() function prototypes

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 are equivalent. Both send data to the back of a queue. xQueueSendFromISR() was the original version and it is now recommended to use xQueueSendToBackFromISR() 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() used to create the queue.
pvItemToQueue	A pointer to the data to be copied into the queue. The size of each item that the queue can hold is set when the queue is created, so this number of 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 *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.

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 themselves perform a context switch. They will instead just indicate whether or not a context switch is required.

xQueueSendFromISR(), xQueueSendToBackFromISR() and xQueueSendToFrontFromISR() must not be called prior to the scheduler being started. Therefore an interrupt that calls any of these functions must not be allowed to execute prior to the scheduler being 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. Preferable approaches include:

1. Packing the multiple data items into a structure, then using a single call to xQueueSendToBackFromISR() to send the entire structure to the queue. This approach is only appropriate if the number of data items is small.

2. Writing the data items into a circular RAM buffer, 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;
portBASE_TYPE 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 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. */
    taskYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Listing 87 Example use of xQueueSendToBackFromISR()

Chapter 4

Semaphore API

4.1 vSemaphoreCreateBinary()

```
#include "FreeRTOS.h"
#include "semphr.h"

void vSemaphoreCreateBinary( xSemaphoreHandle xSemaphore );
```

Listing 88 vSemaphoreCreateBinary() macro prototype

Summary

A macro that creates a binary semaphore. A semaphore must be explicitly created before it can be used.

Parameters

xSemaphore	Variable of type xSemaphoreHandle 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), 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. A fast, small, and simple priority inheritance mechanism is implemented that assumes a task will never hold more than one mutex at a time.

Mutexes and binary semaphores are both referenced using variables that have an xSemaphoreHandle type, and can be used in any API function that takes a parameter of that type.

Mutexes and binary semaphores are both created such that the first call to xSemaphoreTake() on the semaphore or mutex will pass.

Example

```
xSemaphoreHandle xSemaphore;

void vATask( void * pvParameters )
{
    /* Attempt to create a semaphore. */
    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 xSemahore
        variable. */
    }
}
```

Listing 89 Example use of vSemaphoreCreateBinary()

4.2 xSemaphoreCreateCounting()

```
#include "FreeRTOS.h"
#include "semphr.h"

xSemaphoreHandle xSemaphoreCreateCounting(    unsigned portBASE_TYPE uxMaxCount,
                                              unsigned portBASE_TYPE uxInitialCount
                                          );
```

Listing 90 xSemaphoreCreateCounting() function prototype

Summary

Creates a counting semaphore. A semaphore must be explicitly created before it can be used.

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 should be stored as the handle to the created semaphore.

Notes

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 prior to the semaphore being created.

2. Resource management.

In this usage scenario, the count value of the semaphore represents the number of resources that are available.

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 further 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 resource that are available.

Example

```
void vATask( void * pvParameters )
{
xSemaphoreHandle 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 case 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. */
    }
}
```

Listing 91 Example use of xSemaphoreCreateCounting()

4.3 xSemaphoreCreateMutex()

```
#include "FreeRTOS.h"
#include "semphr.h"

xSemaphoreHandle xSemaphoreCreateMutex( void );
```

Listing 92 xSemaphoreCreateMutex() function prototype

Summary

Creates a mutex type semaphore. A semaphore must be explicitly created before it can be used.

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 should be stored as the handle to 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), 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. A fast, small, and simple priority inheritance mechanism is implemented that assumes a task will never hold more than one mutex at a time.

Mutexes and binary semaphores are both referenced using variables that have an xSemaphoreHandle type, and can be used in any API function that takes a parameter of that type.

Mutexes and binary semaphores are both created such that the first call to xSemaphoreTake() on the semaphore or mutex will pass.

Mutex type semaphores cannot be used in an interrupt service routine.

Example

```
xSemaphoreHandle 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. */
    }
}
```

Listing 93 Example use of xSemaphoreCreateMutex()

4.4 xSemaphoreCreateRecursiveMutex()

```
#include "FreeRTOS.h"
#include "semphr.h"

xSemaphoreHandle xSemaphoreCreateRecursiveMutex( void );
```

Listing 94 xSemaphoreCreateRecursiveMutex() function prototype

Summary

Creates a recursive mutex type semaphore. A semaphore must be explicitly created before it can be used.

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 should be stored as the handle to the created semaphore.

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, 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).

Recursive mutex type semaphores cannot be used from within interrupt service routines.

Example

```
void vATask( void * pvParameters )
{
xSemaphoreHandle 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. */
    }
}
```

Listing 95 Example use of xSemaphoreCreateRecursiveMutex()

4.5 vSemaphoreDelete()

```
#include "FreeRTOS.h"
#include "semphr.h"

void vSemaphoreDelete( xSemaphoreHandle xSemaphore );
```

Listing 96 vSemaphoreDelete() function prototype

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.

4.6 xSemaphoreGive()

```
#include "FreeRTOS.h"
#include "semphr.h"

portBASE_TYPE xSemaphoreGive( xSemaphoreHandle xSemaphore );
```

Listing 97 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 xSemaphoreHandle 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

```
xSemaphoreHandle 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. */
        }
    }
}
```

Listing 98 Example use of xSemaphoreGive()

4.7 xSemaphoreGiveFromISR()

```
#include "FreeRTOS.h"
#include "semphr.h"

portBASE_TYPE xSemaphoreGiveFromISR(    xSemaphoreHandle xSemaphore,
                                        portBASE_TYPE *pxHigherPriorityTaskWoken
                                    );
```

Listing 99 xSemaphoreGiveFromISR() function prototype

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

xSemaphore	The semaphore being 'given'. A semaphore is referenced by a variable of type xSemaphoreHandle and must be explicitly created before being used.
*pxHigherPriorityTaskWoken	It is possible that a single semaphore will have one or more tasks blocked on it waiting for the semaphore to become available. Calling xSemaphoreGiveFromISR() can make the semaphore available, and so cause such a task to leave the Blocked state. If calling xSemaphoreGiveFromISR() causes a task to leave the Blocked state, and the unblocked task has a priority higher than or equal to the currently executing task (the task that was interrupted), then, internally, xSemaphoreGiveFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE. If xSemaphoreGiveFromISR() 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.

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 or not a context switch is required.

xSemaphoreGiveFromISR() must not be called prior to the scheduler being started. Therefore an interrupt that calls xSemaphoreGiveFromISR() must not be allowed to execute prior to the scheduler being started.

Example

```
#define LONG_TIME 0xffff
#define TICKS_TO_WAIT    10
xSemaphoreHandle 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 )
{
portBASE_TYPE 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 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. */
    taskYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Listing 100 Example use of xSemaphoreGiveFromISR()

4.8 xSemaphoreGiveRecursive()

```
#include "FreeRTOS.h"
#include "semphr.h"

portBASE_TYPE xSemaphoreGiveRecursive( xSemaphoreHandle xMutex );
```

Listing 101 xSemaphoreGiveRecursive() function prototype

Summary

'Gives' (or releases) a recursive mutex type semaphore that has previously been created using xSemaphoreCreateRecursiveMutex().

Parameters

xMutex	The semaphore being 'given'. A semaphore is referenced by a variable of type xSemaphoreHandle 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, 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.

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

```
/* 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, as 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, ( portTickType ) 10 );
            xSemaphoreTakeRecursive( xMutex, ( portTickType ) 10 );

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

            /* Now the mutex can be taken by other tasks. */
        }
        else
        {
            /* The mutex was not successfully 'taken'. */
        }
    }
}
```

Listing 102 Example use of xSemaphoreGiveRecursive()

4.9 xSemaphoreTake()

```
#include "FreeRTOS.h"
#include "semphr.h"

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

Listing 103 xSemaphoreTake() function prototype

Summary

'Takes' (or obtains) a semaphore that has previously been created using a call to vSemaphoreCreateBinary(), xSemaphoreCreateCounting() or xSemaphoreCreateMutex().

Parameters

xSemaphore	The semaphore being 'taken'. A semaphore is referenced by a variable of type xSemaphoreHandle and must be explicitly created before being used.
xBlockTime	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 xBlockTime 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 is dependent on the tick frequency. The constant portTICK_RATE_MS can be used to convert a time specified in milliseconds to a time specified in ticks. Setting xBlockTime 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 (xBlockTime 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 (xBlockTime 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 and therefore 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

```
xSemaphoreHandle 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. */
            }
        }
    }
}
```

Listing 104 Example use of xSemaphoreTake()

4.10 xSemaphoreTakeRecursive()

```
#include "FreeRTOS.h"
#include "semphr.h"

portBASE_TYPE xSemaphoreTakeRecursive( xSemaphoreHandle xMutex, portTickType
xBlockTime );
```

Listing 105 xSemaphoreTakeRecursive() function prototype

Summary

'Takes' (or obtains) a recursive mutex type semaphore that has previously been created using xSemaphoreCreateRecursiveMutex().

Parameters

xMutex	The semaphore being 'taken'. A semaphore is referenced by a variable of type xSemaphoreHandle and must be explicitly created before being used.
xBlockTime	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 xBlockTime 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 is dependent on the tick frequency. The constant portTICK_RATE_MS can be used to convert a time specified in milliseconds to a time specified in ticks. Setting xBlockTime 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 (xBlockTime 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 (xBlockTime 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' 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.

xSemaphoreTakeRecursive() 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).

xSemaphoreTakeRecursive() must not be called from within a critical section or while the scheduler is suspended.

Example

```
/* 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, as 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, ( portTickType ) 10 );
            xSemaphoreTakeRecursive( xMutex, ( portTickType ) 10 );

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

            /* Now the mutex can be taken by other tasks. */
        }
        else
        {
            /* The mutex was not successfully 'taken'. */
        }
    }
}
```

Listing 106 Example use of xSemaphoreTakeRecursive()

Chapter 5

Software Timer API

5.1 xTimerChangePeriod()

```
#include "FreeRTOS.h"
#include "timers.h"

portBASE_TYPE xTimerChangePeriod( xTimerHandle xTimer,
                                  portTickType xNewPeriod,
                                  portTickType xBlockTime );
```

Listing 107 xTimerChangePeriod() function prototype

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. portTICK_RATE_MS 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 (500 / portTICK_RATE_MS), provided configTICK_RATE_HZ is less than or equal to 1000.
xBlockTime	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. xBlockTime 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 is dependent on the tick frequency. As with the xNewPeriod parameter, the constant portTICK_RATE_MS can be used to convert a time specified in milliseconds to a time specified in ticks.

Setting xBlockTime to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

xBlockTime 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 (xBlockTime 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 (xBlockTime 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

```c
/* This function assumes xTimer has already been created.  If the timer referenced by
xTimer is already active when it is called, then the timer is deleted.  If the timer
referenced by xTimer is not active when it is called, then the period of the timer is
set to 500ms, and the timer is started. */
void vAFunction( xTimerHandle xTimer )
{
    if( xTimerIsTimerActive( xTimer ) != pdFALSE )
    {
        /* xTimer is already active - delete it. */
        xTimerDelete( xTimer );
    }
    else
    {
        /* xTimer is not active, change its period to 500ms.  This will also cause
        the timer to start.  Block for a maximum of 100 ticks if the change period
        command cannot immediately be sent to the timer command queue. */
        if( xTimerChangePeriod( xTimer, 500 / portTICK_RATE_MS, 100 ) == pdPASS )
        {
            /* The command was successfully sent. */
        }
        else
        {
            /* The command could not be sent, even after waiting for 100 ticks to
            pass.  Take appropriate action here. */
        }
    }
}
```

Listing 108 Example use of xTimerChangePeriod()

5.2 xTimerChangePeriodFromISR()

```
#include "FreeRTOS.h"
#include "timers.h"

portBASE_TYPE xTimerChangePeriodFromISR( xTimerHandle xTimer,
                                         portTickType xNewPeriod,
                                         portBASE_TYPE *pxHigherPriorityTaskWoken );
```

Listing 109 xTimerChangePeriodFromISR() function prototype

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. portTICK_RATE_MS 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 (500 / portTICK_RATE_MS), 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 *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 500ms. */

/* The interrupt service routine that changes the period of xTimer. */
void vAnExampleInterruptServiceRoutine( void )
{
portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

    /* The interrupt has occurred - change the period of xTimer to 500ms.
    xHigherPriorityTaskWoken was set to pdFALSE where it was defined (within this
    function).  As this is an interrupt service routine, only FreeRTOS API functions
    that end in "FromISR" can be used. */
    if( xTimerChangePeriodFromISR( xTimer, &xHigherPriorityTaskWoken ) != pdPASS )
    {
        /* The command to change the 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 actual syntax required. */
    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        /* Call the interrupt safe yield function here (actual function depends on
        the FreeRTOS port being used). */
    }
}
```

Listing 110 Example use of xTimerChangePeriodFromISR()

5.3 xTimerCreate()

```
#include "FreeRTOS.h"
#include "timers.h"

xTimerHandle xTimerCreate( const signed char *pcTimerName,
                           portTickType xTimerPeriod,
                           unsigned portBASE_TYPE uxAutoReload,
                           void * pvTimerID,
                           tmrTIMER_CALLBACK pxCallbackFunction );
```

Listing 111 xTimerCreate() function prototype

Summary

Creates and initializes a new instance of a software timer.

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, purely to assist debugging.
xTimerPeriod	The timer period. Timer periods are specified in multiples of tick periods. portTICK_RATE_MS 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 (500 / portTICK_RATE_MS), provided configTICK_RATE_HZ is less than or equal to 1000.
uxAutoReload	Set to pdTRUE to create an autoreaload timer. Set to pdFALSE to create a one-shot timer. Once started, an autoreload timer will expire repeatedly with a frequency set by the xTimerPeriod parameter.

	Once 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. 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.
pxCallbackFunction	The function to call when the timer expires. Callback functions must have the prototype defined by the tmrTIMER_CALLBACK typedef. The required prototype is shown in Listing 112.

```
void vCallbackFunctionExample( xTimerHandle xTimer );
```

Listing 112 The timer callback function prototype

Return Values

NULL	The timer could not be created because there was insufficient FreeRTOS heap memory available to successfully allocate the timer data structures.
Non-NULL	A new timer was created, and the handle of the timer was returned.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerCreate() to be available.

Example

```
#define NUM_TIMERS 5

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

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

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

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

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

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

    /* If the timer has expired 10 times, then stop it from running. */
    if( lExpireCounters[ lArrayIndex ] == xMaxExpiryCountBeforeStopping )
    {
        /* Do not use a block time if calling a timer API function from a timer callback
        function, as doing so could cause a deadlock! */
        xTimerStop( pxTimer, 0 );
    }
}
```

Listing 113 Definition of the callback function used in the calls to xTimerCreate() in Listing 114

```
void main( void )
{
long x;

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

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

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

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

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

Listing 114 Example use of xTimerCreate()

5.4 xTimerDelete()

```
#include "FreeRTOS.h"
#include "timers.h"

portBASE_TYPE xTimerDelete( xTimerHandle xTimer, portTickType xBlockTime );
```

Listing 115 xTimerDelete() macro prototype

Summary

Deletes a timer. The timer must first have been created using the xTimerCreate() API function.

Parameters

xTimer	The handle of the timer being deleted.
xBlockTime	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. xBlockTime 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 is dependent on the tick frequency. The constant portTICK_RATE_MS can be used to convert a time specified in milliseconds to a time specified in ticks.

Setting xBlockTime to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

xBlockTime 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 (xBlockTime 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 (xBlockTime 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.

5.1 xTimerGetTimerDaemonTaskHandle()

```
#include "FreeRTOS.h"
#include "task.h"

xTaskHandle xTimerGetTimerDaemonTaskHandle( void );
```

Listing 116 xTimerGetTimerDaemonTaskHandle() function prototype

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

INCLUDE_xTimerGetTimerDaemonTaskHandle and configUSE_TIMERS must both be set to 1 in FreeRTOSConfig.h for xTimerGetTimerDaemonTaskHandle() to be available.

5.2 pvTimerGetTimerID()

```
#include "FreeRTOS.h"
#include "timers.h"

void *pvTimerGetTimerID( xTimerHandle xTimer );
```

Listing 117 pvTimerGetTimerID() function prototype

Summary

Returns the identifier (ID) assigned to the timer when the timer was created. 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.

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

See the example provided for the xTimerCreate() API function.

5.3 xTimerIsTimerActive()

```
#include “FreeRTOS.h”
#include “timers.h”

portBASE_TYPE xTimerIsTimerActive( xTimerHandle xTimer );
```

Listing 118 xTimerIsTimerActive() function prototype

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 restrated 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( xTimerHandle 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. */
    }
}
```

Listing 119 Example use of xTimerIsTimerActive()

5.4 xTimerReset()

```
#include "FreeRTOS.h"
#include "timers.h"

portBASE_TYPE xTimerReset( xTimerHandle xTimer, portTickType xBlockTime );
```

Listing 120 xTimerReset() function prototype

Summary

Re-starts 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 mean time, 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.
xBlockTime	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. xBlockTime 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 is dependent on the tick frequency. The constant portTICK_RATE_MS can be used to convert a time specified in milliseconds to a time specified in ticks.

Setting xBlockTime to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

xBlockTime 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 (xBlockTime 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 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 (xBlockTime 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 back-light is switched on.  If 5 seconds pass
without a key being pressed, then the LCD back-light is switched off by a one-shot timer. */

xTimerHandle xBacklightTimer = NULL;

/* The callback function assigned to the one-shot timer.  In this case the parameter is not
used. */
void vBacklightTimerCallback( xTimerHandle pxTimer )
{
    /* The timer expired, therefore 5 seconds must have passed since a key was pressed.  Switch
    off the LCD back-light.
    vSetBacklightState( BACKLIGHT_OFF );
}

/* The key press event handler. */
void vKeyPressEventHandler( char cKey )
{
    /* Ensure the LCD back-light is on, then reset the timer that is responsible for turning the
    back-light off after 5 seconds of key inactivity.  Wait 10 ticks for the 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 then start the one-shot timer that is responsible for turning the back-light off
    if no keys are pressed within a 5 second period. */
    xBacklightTimer = xTimerCreate( "BcklghtTmr" /* Just a text name, not used by the kernel. */
                                    ( 5000 / portTICK_RATE_MS), /* 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 back-light off. */
                                  );

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

    /* Create tasks here. */

    /* Starting the scheduler will start the timer running as xTimerStart has already been
    called. */
    xTaskStartScheduler();
}
```

Listing 121 Example use of xTimerReset()

5.5 xTimerResetFromISR()

```
#include "FreeRTOS.h"
#include "timers.h"

portBASE_TYPE xTimerResetFromISR( xTimerHandle xTimer,
                                  portBASE_TYPE *pxHigherPriorityTaskWoken );
```

Listing 122 xTimerResetFromISR() function prototype

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 *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

```c
/* This scenario assumes xBacklightTimer has already been created.  When a key is
pressed, an LCD back-light is switched on.  If 5 seconds pass without a key being
pressed, then the LCD back-light is switched off 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( xTimerHandle pxTimer )
{
    /* The timer expired, therefore 5 seconds must have passed since a key was
    pressed.  Switch off the LCD back-light. */
    vSetBacklightState( BACKLIGHT_OFF );
}

/* The key press interrupt service routine. */
void vKeyPressEventInterruptHandler( void )
{
portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

    /* Ensure the LCD back-light is on, then reset the timer that is responsible for
    turning the back-light off after 5 seconds of key inactivity.  This is an
    interrupt service routine so can only call FreeRTOS API functions that end in
    "FromISR". */
    vSetBacklightState( BACKLIGHT_ON );

    /* xTimerStartFromISR() or xTimerResetFromISR() could be called here as both
    cause the timer to re-calculate its expiry time.  xHigherPriorityTaskWoken was
    initialised to pdFALSE when it was declared (in this function). */
    if( xTimerResetFromISR( xBacklightTimer, &xHigherPriorityTaskWoken ) != pdPASS )
    {
        /* The reset command was not executed successfully.  Take appropriate action
        here. */
    }

    /* Perform the rest of the key processing here. */

    /* If xHigherPriorityTaskWoken equals pdTRUE, then a context switch should be
    performed.  The syntax required to perform a context switch from inside an ISR
    varies from port to port, and from compiler to compiler.  Inspect the demos for
    the port you are using to find the actual syntax required. */
    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        /* Call the interrupt safe yield function here (actual function depends on
        the FreeRTOS port being used). */
    }
}
```

Listing 123 Example use of xTimerResetFromISR()

5.6 xTimerStart()

```
#include "FreeRTOS.h"
#include "timers.h"

portBASE_TYPE xTimerStart( xTimerHandle xTimer, portTickType xBlockTime );
```

Listing 124 xTimerStart() function prototype

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 mean time, 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.
xBlockTime	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. xBlockTime 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 is dependent on the tick frequency. The constant portTICK_RATE_MS can be used to convert a time specified in milliseconds to a time specified in ticks. Setting xBlockTime to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in

FreeRTOSConfig.h.

xBlockTime 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 (xBlockTime 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 (xBlockTime 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.

5.7 xTimerStartFromISR()

```
#include "FreeRTOS.h"
#include "timers.h"

portBASE_TYPE xTimerStartFromISR( xTimerHandle xTimer,
                                  portBASE_TYPE *pxHigherPriorityTaskWoken );
```

Listing 125 xTimerStartFromISR() macro prototype

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 *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. 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 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 back-light is switched on.  If 5 seconds pass without a key being
pressed, then the LCD back-light is switched off by a on-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( xTimerHandle pxTimer )
{
    /* The timer expired, therefore 5 seconds must have passed since a key was
    pressed.  Switch off the LCD back-light. */
    vSetBacklightState( BACKLIGHT_OFF );
}

/* The key press interrupt service routine. */
void vKeyPressEventInterruptHandler( void )
{
portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

    /* Ensure the LCD back-light is on, then restart the timer that is responsible
    for turning the back-light off after 5 seconds of key inactivity.  This is an
    interrupt service routine so can only call FreeRTOS API functions that end in
    "FromISR". */
    vSetBacklightState( BACKLIGHT_ON );

    /* xTimerStartFromISR() or xTimerResetFromISR() could be called here as both
    cause the timer to re-calculate its expiry time.  xHigherPriorityTaskWoken was
    initialised to pdFALSE when it was declared (in this function). */
    if( xTimerStartFromISR( xBacklightTimer, &xHigherPriorityTaskWoken ) != pdPASS )
    {
        /* The start command was not executed successfully.  Take appropriate action
        here. */
    }

    /* Perform the rest of the key processing here. */

    /* If xHigherPriorityTaskWoken equals pdTRUE, then a context switch should be
    performed.  The syntax required to perform a context switch from inside an ISR
    varies from port to port, and from compiler to compiler.  Inspect the demos for
    the port you are using to find the actual syntax required. */
    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        /* Call the interrupt safe yield function here (actual function depends on
        the FreeRTOS port being used). */
    }
}
```

Listing 126 Example use of xTimerStartFromISR()

5.8 xTimerStop()

```
#include "FreeRTOS.h"
#include "timers.h"

portBASE_TYPE xTimerStop( xTimerHandle xTimer, portTickType xBlockTime );
```

Listing 127 xTimerStop() function prototype

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.
xBlockTime	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. xBlockTime 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 is dependent on the tick frequency. The constant portTICK_RATE_MS can be used to convert a time specified in milliseconds to a time specified in ticks. Setting xBlockTime to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h. xBlockTime 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 (xBlockTime 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 (xBlockTime 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.

5.9 xTimerStopFromISR()

```
#include "FreeRTOS.h"
#include "timers.h"

portBASE_TYPE xTimerStopFromISR( xTimerHandle xTimer,
                                 portBASE_TYPE *pxHigherPriorityTaskWoken );
```

Listing 128 xTimerStopFromISR() function prototype

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 *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. 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.

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 )
{
portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

    /* The interrupt has occurred - simply stop the timer. xHigherPriorityTaskWoken
    was set to pdFALSE where it was defined (within this function).  As this is an
    interrupt service routine, only FreeRTOS API functions that end in "FromISR" can
    be used. */
    if( xTimerStopFromISR( xTimer, &xHigherPriorityTaskWoken ) != pdPASS )
    {
        /* The stop command was not executed successfully.  Take appropriate action
        here. */
    }

    /* If xHigherPriorityTaskWoken equals pdTRUE, then a context switch should be
    performed.  The syntax required to perform a context switch from inside an ISR
    varies from port to port, and from compiler to compiler.  Inspect the demos for
    the port you are using to find the actual syntax required. */
    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        /* Call the interrupt safe yield function here (actual function depends on
        the FreeRTOS port being used). */
    }
}
```

Listing 129 Example use of xTimerStopFromISR()

Chapter 6

Kernel Configuration

6.1 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 contains a pre-defined FreeRTOSConfig.h that can be used as a reference or simply copied. Note, however, that some of the demo projects were generated before all the options documented in this chapter were available, so the FreeRTOSConfig.h header files they contain will not include all the constants and options that are documented in the following sub-sections.

6.2 Constants that Start “INCLUDE_”

Constants that start with the text “INCLUDE_” are used to included or excluded 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 vTaskPriotitySet(). 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, most linkers will, by default, automatically remove unreferenced code unless optimization is turned completely off. Linkers that do not have this default behavior can normally be configured to remove unreferenced code. Therefore, in most practical 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_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_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_pcTaskGetTaskName

INCLUDE_pcTaskGetTaskName must be set to 1 for the pcTaskGetTaskName() 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_vTaskSuspend

INCLUDE_vTaskSuspend must be set to 1 for the vTaskSuspend(), xTaskResume(), 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 time out, is specified. The calling task will then be held in the Blocked state until either 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 time out). 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_xTimerGetTimerDaemonTaskHandle

configUSE_TIMERS and INCLUDE_xTimerGetTimerDaemonTaskHandle must both be set to 1 for the xTimerGetTimerDaemonTaskHandle() API function to be available.

6.3 Constants that Start “config”

Constants that start with the text “config” define attributes of the kernel, or include or exclude features of the kernel.

configASSERT

Calls to configASSERT(x) exist at key points in the FreeRTOS kernel code.

If FreeRTOS is functioning correctly, and is being used correctly, then the configASSERT() parameter will be non-zero. If the parameter is found to equal zero, then an error has occurred.

It is likely that most errors trapped by configASSERT() will be a result of an invalid parameter being passed into a FreeRTOS API function. configASSERT() can therefore assist in run time debugging. However, defining configASSERT() will also increase the application code size, and slow 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. Listing 130 shows an example configASSERT() definition that assumed vAssertCalled() is defined elsewhere by the application.

```
#define configASSERT( ( x ) )  if( ( x ) == 0 ) vAsserCalled( __FILE__, __LINE__ )
```

Listing 130 An example configASSERT() definition

configCHECK_FOR_STACK_OVERFLOW

Each task has a unique stack that is automatically allocated from the FreeRTOS heap when the task is created. The size of the stack is set by the usStackDepth parameter of the xTaskCreate() call used to create the task.

Stack overflow is a very common cause of application instability. FreeRTOS provides two optional mechanisms that can be used to assist in stack overflow detection and debugging.

Which (if any) option is used is configured by the configCHECK_FOR_STACK_OVERFLOW configuration constant.

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 in Listing 131.

```
void vApplicationStackOverflowHook( xTaskHandle *pxTask,
                                    signed char *pcTaskName );
```

Listing 131 The stack overflow hook function prototype

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. It should be noted that a stack overflow can potentially corrupted 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 kernels periodic tick interrupt. This is very often, but not always, equal to the main system clock frequency.

configGENERATE_RUN_TIME_STATS

The task run time statistics feature collects information on the amount of processing time each task is receiving. The feature requires the application to configure a run time statistics time base. The frequency of the run time 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 run time statistics gathering functionality and associated API in the build. Setting configGENERATE_RUN_TIME_STATS to 0 will exclude the run time 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 Table 1. If configGENERATE_RUN_TIME_STATS is set to 0 then the application must not define any of the macros described in Table 1, otherwise there is a risk that the application will not compiler and/or link.

Table 1. Additional macros that are required if configGENERATE_RUN_TIME_STATS is set to 1

Macro	Description
portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()	This macro must be provided to initialize whichever peripheral is used to generate the run time 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 of ‘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 may not remain in the running state for an entire tick period. For example, a task may 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 pre-empted.

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 Idle task consistently yielding to another Idle priority Ready state tasks has the side effect shown in Figure 3.

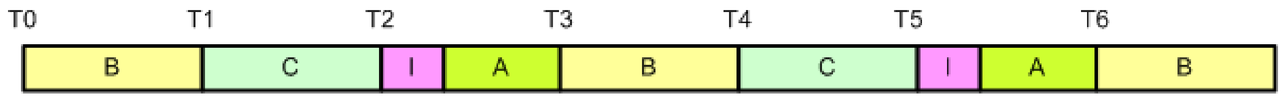


Figure 3 Time line showing the execution of 4 tasks, all of which run at the idle priority

Figure 3 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, etc. It can be seen that the Idle task starts to execute at time T2. It executes for part of a time slice, then yields to Task A. Task A executes for the remainder of the same time slice, then gets pre-empted at time T3. Task I and task A effectively share a single time slice, resulting in task B and task C consistently utilizing 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 pre-empted 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.

configKERNEL_INTERRUPT_PRIORITY, configMAX_SYSCALL_INTERRUPT_PRIORITY

configKERNEL_INTERRUPT_PRIORITY and configMAX_SYSCALL_INTERRUPT_PRIORITY are only relevant to ports that implement interrupt nesting.

If a port only implements the configKERENL_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 is 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 is not adversely affected by anything the kernel is executing.

As an example – imagine a hypothetical microcontroller that has seven interrupt priority levels. In this hypothetical case, one is the lowest interrupt priority and seven is the highest interrupt priority[2]. Figure 4 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.

[2] Note care must be taken when assigning values to configKERNEL_INTERRUPT_PRIORITY and configMAX_SYCALL_INTERRUPT_PRIORITY as some microcontrollers use zero or one to mean the *lowest* priority, while others use zero or one to mean the *highest* priority.

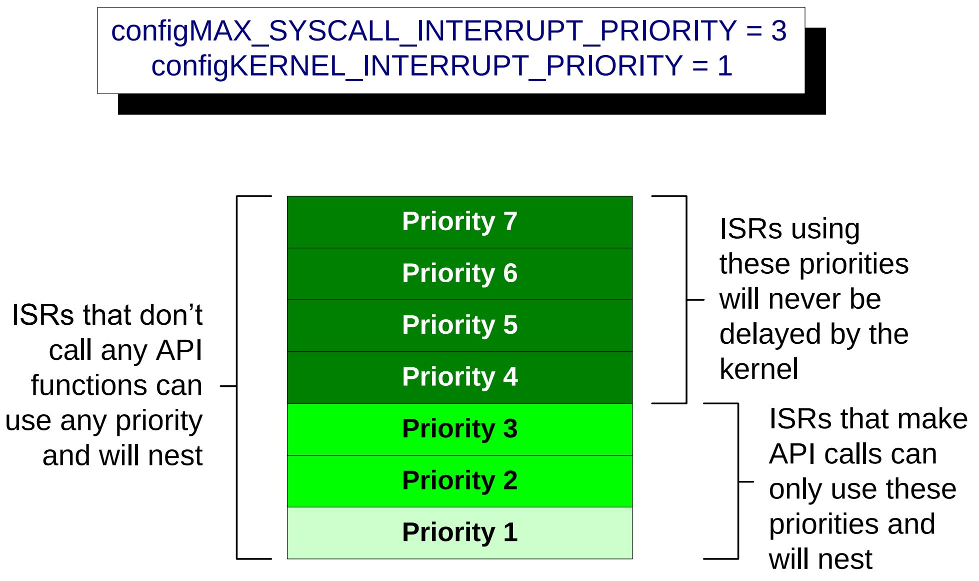

configMAX_SYSCALL_INTERRUPT_PRIORITY = 3
configKERNEL_INTERRUPT_PRIORITY = 1
Priority 7
Priority 6
Priority 5
Priority 4
Priority 3
Priority 2
Priority 1
ISRs that don't call any API functions can use any priority and will nest
ISRs using these priorities will never be delayed by the kernel
ISRs that make API calls can only use these priorities and will nest

Figure 4 An example interrupt priority configuration

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, in not always, be set to the lowest available interrupt priority.

configMAX_CO_ROUTINE_PRIORITIES

Sets the maximum priority that can be assigned to a co-routine. Co-routines 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 of 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 itself does not use configMINIMAL_STACK_SIZE for any other purpose, although 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.

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 only required 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.

configTICK_RATE_HZ

Sets the tick interrupt frequency. The value is specified in Hz.

The constant portTICK_RATE_MS is determined by the configTICK_RATE_HZ definition, and is provided to allow block times to be specified in milliseconds rather than ticks. Block times specified this way will remain constant even when the configTICK_RATE_HZ definition is changed. portTICK_RATE_MS can only be used when configTICK_RATE_HZ is less than or equal to 1000. The standard demo tasks make extensive use of portTICK_RATE_MS, 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.

Reasons the timer command queue might fill up include:

- 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. It is worth noting however, that timer expiry times are calculated relative to when a command is sent, and not relative 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 and heap_3.c source files respectively. The schemes defined by heap_1.c and heap_2.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 or heap_2.c are being used by the application.

configUSE_16_BIT_TICKS

The tick count is held in a variable of type portTickType. When configUSE_16_BIT_TICKS is set to 1, portTickType is defined to be an unsigned 16-bit type. When configUSE_16_BIT_TICKS is set to 0, portTickType 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 in this manual. Use of the alternative API is no longer 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

Co-routines are light weight tasks that save RAM by sharing a stack, but have limited functionality. Their use is omitted from this manual.

Setting configUSE_CO_ROUTINES to 1 will include all co-routine functionality and its associated API functions in the build. Setting configUSE_CO_ROUTINES to 0 will exclude all co-routine 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_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 Listing 132.

```
void vApplicationIdleHook( void );
```

Listing 132 The idle task hook function name and prototype.

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 and heap_3.c source files respectively. configUSE_MALLOC_FAILED_HOOK is only relevant when one of these 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 Listing 133.

```
void vApplicationMallocFailedHook( void );
```

Listing 133 The malloc() failed hook function name and prototype.

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_PREEMPTION

Setting configUSE_PREEMPTION to 1 will cause the pre-emptive scheduler to be used. Setting configUSE_PREEMPTION to 0 will cause the co-operative scheduler to be used.

When the pre-emptive 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 co-operative 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_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_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 Listing 134.

```
void vApplicationTickHook( void );
```

Listing 134 The tick hook function name and prototype.

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_TRACE_FACILITY

The configUSE_TRACE_FACILITY constant relates to a legacy trace facility that is no longer recommended for use. It is recommended to use the trace hook macros in its place.

Setting configUSE_TRACE_FACILITY to 1 will cause the legacy trace functionality and its associated API to be included in the build. Setting configUSE_TRACE_FACILITY to 0 will cause the legacy trace functionality and its associated API to be excluded from the build.

APPENDIX 1: Data Types and Coding Style Guide

Data Types

Each port of FreeRTOS has a unique portmacro.h header file that contains (amongst other things) definitions for two special data types, portTickType and portBASE_TYPE. These data types are described in Table 2.

Table 2. Special data types used by FreeRTOS

Macro or typedef used	Actual type
portTickType	This is used to store the tick count value, and by variables that specify block times. portTickType 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.
portBASE_TYPE	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. portBASE_TYPE is generally used for variables that can take only a very limited range of values, and for Booleans.

Some compilers make all unqualified char variables unsigned, while others make them signed. For this reason, the FreeRTOS source code explicitly qualifies every use of char with either 'signed' or 'unsigned'.

Plain int types are never used—only long and short.

Variable Names

Variables are prefixed with their type: 'c' for char, 's' for short, 'l' for long, and 'x' for portBASE_TYPE and any other types (structures, task handles, queue handles, etc.).

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', and 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 they are defined in. For example:

- v**Task**PrioritySet() returns a void and is defined within **task**.c.
- x**Queue**Receive() returns a variable of type portBASE_TYPE and is defined within **queue**.c.
- v**Semaphore**CreateBinary() 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 upper case and prefixed with lower case letters that indicate where the macro is defined. Table 3 provides a list of prefixes.

Table 3. Macro 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

Note that 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 Table 4 are used throughout the FreeRTOS source code.

Table 4. Common macro definitions

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.

APPENDIX 2: Licensing Information

FreeRTOS is licensed under a modified version of the GNU General Public License (GPL) and *can* be used in commercial applications under that license. An alternative and optional commercial license is also available if:

- You cannot fulfill the requirements stated in the 'Open source modified GPL license' column of Table 5.
- You wish to receive direct technical support.
- You wish to have assistance with your development.
- You require guarantees and indemnification.

Table 5. Comparing the open source license with the commercial license

	Open source modified GPL license	Commercial license
Is it free?	Yes	No
Can I use it in a commercial application?	Yes	Yes
Is it royalty free?	Yes	Yes
Do I have to open source my application code?	No	No
Do I have to open source my changes to the FreeRTOS kernel?	Yes	No
Do I have to document that my product uses FreeRTOS.	Yes	No
Do I have to offer to provide the FreeRTOS source code to users of my application?	Yes (a WEB link to the FreeRTOS.org site is normally sufficient)	No
Can I receive support on a commercial basis?	No	Yes
Are any legal guarantees provided?	No	Yes

Open Source License Details

The FreeRTOS source code is licensed under version 2 of the GNU General Public License (GPL) *modified by an exception*.

The full text of the GPL is available at http://www.freertos.org/license.txt. The text of the exception that modified the GPL license is provided below.

The exception permits the source code of applications that use FreeRTOS solely through the API published on the FreeRTOS.org website to remain closed source, thus permitting the use of FreeRTOS in commercial applications without necessitating that the entire application be open sourced. The exception can be used only if you wish to combine FreeRTOS with a proprietary product and you comply with the terms stated in the exception itself.

GPL Exception Text

Note that the exception text is subject to change. Consult the FreeRTOS.org website for the most recent version.

Clause 1

Linking FreeRTOS statically or dynamically with other modules is making a combined work based on FreeRTOS. Thus, the terms and conditions of the GNU General Public License cover the whole combination.

As a special exception, the copyright holder of FreeRTOS gives you permission to link FreeRTOS with independent modules that communicate with FreeRTOS solely through the FreeRTOS API interface, regardless of the license terms of these independent modules, and to copy and distribute the resulting combined work under terms of your choice, provided that:

1. *Every copy of the combined work is accompanied by a written statement that details to the recipient the version of FreeRTOS used and an offer by yourself to provide the FreeRTOS source code (including any modifications you may have made) should the recipient request it.*

2. *The combined work is not itself an RTOS, scheduler, kernel or related product.*

3. *The independent modules add significant and primary functionality to FreeRTOS and do not merely extend the existing functionality already present in FreeRTOS.*

An independent module is a module which is not derived from or based on FreeRTOS.

Clause 2

FreeRTOS may not be used for any competitive or comparative purpose, including the publication of any form of run time or compile time metric, without the express permission of Real Time Engineers ltd. (this is the norm within the industry and is intended to ensure information accuracy).

INDEX