WITTENSTEIN

WITTENSTEIN **High**Integrity**Systems**

SAFE**RTOS** USER MANUAL

WITTENSTEIN

CONTENTS

LIST OF FIGURES

LIST OF TABLES

WITTENSTEIN

LIST OF CODE LISTINGS

LIST OF NOTATION

API	Application Program Interface
FIFO	First In First Out
ISR	Interrupt Service Routine
Ltd	Limited
RTOS	Real Time Operating System
WHIS	WITTENSTEIN high integrity systems

REFERENCED DOCUMENTS

Ref #	Document	Description
1	34-172-MAN-2	SafeRTOS Safety Manual
2	IEC 61508	Functional Safety of Electrical/Electronic/Programmable Electronic Safety Related Systems.
3	34-172-MAN-3-ccc-aaa	SafeRTOS Product Variant User Manual

WITTENSTEIN

CHAPTER 1 INTRODUCTION

WITTENSTEIN

1.1 ABOUT THIS MANUAL

1.1.1 Identification

This is the user manual for the SafeRTOS^{TM} pre-emptive real time scheduler. SafeRTOS is either supplied as C and assembler code, as a C linkable library or, depending on the processor, pre-programmed in to the processor ROM.

Incorporating SafeRTOS in to an embedded software application permits that application to be structured as a set of autonomous tasks. The scheduler selects which task to execute at any point in time in accordance with the state and relative priority of each created task. CHAPTER 2 elaborates the states in which a task can exist.

SafeRTOS is based on the OPEN**RTOS**^{TM} code base.

1.1.2 Use in Safety Related Systems

SafeRTOS was developed using a formalized process. The initial version was independently certified by TÜV SÜD to confirm that the development processes used were as expected when implementing an IEC 61508 [Reference 2] part 3, SIL 3 project. The same processes have been used to create all subsequent 'Product Variants' of SafeRTOS that are specific to a particular processor and development environment. The requirements used for this development and the evidence of conformance are contained in the Design Assurance Package for SafeRTOS. The Design Assurance Package is specific to each 'Product Variant' of SafeRTOS.

Any use of SafeRTOS in any application cannot make any claim related to the conformance of SafeRTOS to any requirements or process specification (including IEC 61508 [Reference 2]) without first following a recognized system wide conformance verification process. This conformance evidence must then be presented audited and accepted by a recognized and relevant independent assessment organization. Without undergoing this process of due diligence **no claim can be made** as to the suitability of SafeRTOS to be used in any safety or otherwise commercially critical application.

1.1.3 Document Overview

1.1.3.1 Scope

It is assumed that system developers are adequately trained or already experienced in their field of involvement. It is therefore assumed that readers are familiar with the concepts of multitasking embedded systems (such as multiple tasks, reentrancy and mutual exclusion) and are proficient in the C programming language. This manual is limited to technical aspects specific to SafeRTOS.

Please refer to the SafeRTOS Safety Manual [Reference 1] for information on integrating SafeRTOS into safety related applications. The Safety Manual is available as part of the Design Assurance Package.

The '⚠' symbol is used to emphasize instruction or information to which compliance is deemed to be essential for the correct and safe integration of SAFERTOS into an application.

1.1.3.2 Following Chapters

CHAPTER 2 provides an overview of SafeRTOS and the description of the SafeRTOS task, queue and scheduling mechanisms.

CHAPTER 3 describes the installation and setup required to use SafeRTOS in your application. Information specific to the SafeRTOS port being used is deferred to a separate technical note.

CHAPTER 4 provides an API reference.

⚠ SAFERTOS users must not call functions within the SAFERTOS code base that are not documented within CHAPTER 4.

CHAPTER 2 SYSTEM DESCRIPTION

2.1 SYSTEM OVERVIEW

2.1.1 Summary of the SafeRTOS Scheduler

The SafeRTOS pre-emptive real time scheduler has the following characteristics:

- Any number of tasks can be created - system RAM constraints are the limiting factor.
- Each task is assigned a priority - any number of priorities can be used (source code versions only).
- Any number of tasks can share the same priority - allowing for maximum application design flexibility.
- The highest priority task that is able to execute (i.e. that is not blocked or suspended) will be the task selected by the scheduler to execute.
- Tasks of equal priority will each get a share of the processing time available to tasks of that priority. A time sliced round robin policy is used (see the Section 'The Scheduling Policy').
- Queues can be used to send data between tasks, and to send data between tasks and interrupt service routines (ISR).
- Tasks can block for a fixed period.
- Tasks can block to wait for a specified time.
- Tasks can block with a specified timeout period to wait for queue events (either data being written to or read from the queue).

2.1.2 Differences Between SafeRTOS and OPENRTOS

While SafeRTOS and OPEN**RTOS** share many attributes the development process has necessitated some notable differences. These are summarized below:

- SafeRTOS does not dynamically allocate any memory. All the memory required for the creation of tasks and queues must be provided by the host application. This has necessitated some changes to the OPEN**RTOS** API.
- SafeRTOS performs validity checks on all parameters passed into its API and on some internal data values. As a result more SafeRTOS API functions return a status value than their OPEN**RTOS** counterparts.
- The scheduler will not permit a task stack to overflow during the task context switch process.
- The detection of an error within the scheduler's internal data or the detection of a potential stack overflow (during a context switch) will result in the execution of an application defined callback function. This permits application-specific fail-safe processing to be performed.
- OPEN**RTOS** implemented binary and counting semaphores through the provision of a set of macros. These macros did nothing other than use the existing queue implementation and have been removed. CHAPTER 4 provides information on how the documented API can be manually used to obtain the same functionality.

- SafeRTOS does not provide recursive semaphores, or mutexes with priority inheritance. This functionality can be added if required.
- SafeRTOS does not support co-routines. OPEN**RTOS** co-routines are light weight tasks that utilize the same stack.
- OPEN**RTOS** allows components to be optionally excluded through the use of preprocessor directives. SafeRTOS does not include any conditional compilation options. In most cases the linker can be used to achieve the same code size reduction.
- OPEN**RTOS** permits the scheduling policy to be optionally set to 'cooperative'. SafeRTOS only permits the policy to be 'preemptive'.
- SafeRTOS does not provide a tick hook function.
- OPEN**RTOS** defines stack sizes in terms of the number of data items the stack can hold whereas SafeRTOS defines stack sizes in bytes. See the documentation for the ulStackSizeBytes parameter to the xTaskCreate() function for further information.

From these points it can be seen that for reasons of certification SafeRTOS is a statically configured subset of OPEN**RTOS**. This is to maintain control over the scope of code test and analysis that must be performed whilst also reducing reliance upon preprocessor compilation carried out by the chosen development tools. It is technically very simple to add back in the same configurability and feature set provided by OPEN**RTOS** - please contact WITTENSTEIN high integrity systems should this be required for your application.

2.1.3 Design Goals

The design goal of SafeRTOS is to achieve its stated functionality using a small, simple and robust implementation.

2.2 CODING CONVENTIONS

2.2.1 Project Definitions

Each C file that utilizes the SafeRTOS API must include the SafeRTOS.h header file before the header file that contains the prototype of the API function itself. SafeRTOS.h includes ProjDefs.h, which contains the definitions detailed in the Table 'Project Definitions' and the Table 'Port dependent definitions'.

Table 2-1 Project Definitions

Definition	Value
pdTRUE	1
pdFALSE	0
pdPASS	1
pdFAIL	0

The ‘pd’ prefix denotes that the constant is defined within the ProjDefs.h header file. ProjDefs.h also contains error code definitions which are prefixed ‘err’.

Table 2-2 Port dependent definitions

Definition	Value
portCHAR	char (type)
portLONG	long (type)
portSHORT	short (type)
portBASE_TYPE	Port dependent - defined to be the most efficient data type for the architecture
portMAX_DELAY	Port dependent
portTickType	Port dependent

Values in the Table 'Port dependent definitions' described as ‘Port dependent’ are defined in the port specific documentation.

2.2.2 Naming Conventions

The following conventions are used throughout the code:

- Parameter names are prefixed with their type as follows:
  - Variables of type portCHAR are prefixed c
  - Variables of type portSHORT are prefixed s
  - Variables of type portLONG are prefixed l
  - Variables of type portBASE_TYPE are prefixed x
  - Other types (e.g. structures) are also prefixed x
  - Items of type void are also prefixed v (pointers to void and void functions)
  - Pointers have an additional prefixed p, for example a pointer to a short will have prefix ps, a pointer to void will have the prefix pv, etc..
  - Unsigned variables have an additional prefixed u, for example an unsigned short will have prefix us
- Function names are also prefixed with their return type using the same convention.
- API functions for which the function prototype is contained in the file 'task.h' start with the word 'Task'. For example, the prototype for the API function xTaskGetTickCount() is contained in 'task.h' and the function returns a value of type portTickType.
- API functions for which the function prototype is contained in the file 'queue.h' start with the word 'Queue'. For example, the prototype for the API function xQueueSend() is contained in 'queue.h' and the function returns a value of portBASE_TYPE.
- Macro names are written in all upper case other than a lower case prefix that indicates in which header file the macro is defined. The exception to this rule are the error codes which are prefixed 'err' but contained in the ProjDefs.h header file.

2.3 SYSTEM COMPONENTS

2.3.1 Tasks

Including SafeRTOS in your application allows the application to be structured as a set of autonomous tasks - the resultant system functionality being the sum of the functionality of the multiple tasks that make up the application.

Each task executes within its own context with no coincidental dependency on other tasks within the system or the scheduler itself.

2.3.1.1 Task Functions

Functions that implement a task must be of type pdTASK_CODE, where pdTASK_CODE is defined as shown by the Listing 'The pdTASK_CODE definition' with an example of such a function shown in the Listing 'The typical structure of a task'.

A task will typically execute indefinitely and as such be written as an infinite loop, also demonstrated by the Listing 'The typical structure of a task'.

```
typedef void (*pdTASK_CODE)( void * pvParameters );
```

Listing 1 The pdTASK_CODE definition

```
void vATaskFunction( void *pvParameters )
{
        /* The function executes indefinitely so enter an infinite loop. */
        for( ;; )
        {
                /* -- Task application code goes here. -- */
        }
}
```

Listing 2 The typical structure of a task

A task is created using the xTaskCreate() API function.

A task is deleted using the xTaskDelete() API function.

⚠ A task function must never terminate by attempting to return to its caller (or by calling exit()) as doing so will result in undefined behavior. If required a task can delete itself prior to reaching the function end as illustrated by the Listing 'A task deleting itself prior to the function terminating'.

```
void vATaskFunction( void *pvParameters )
{
       for( ;; )
       {
              /* -- Task application code here. -- */
       }

       /* The task deletes itself (indicated by the NULL parameter)
          before reaching the end of the task function. */
       xTaskDelete( NULL );
}
```

Listing 3 A task deleting itself prior to the function terminating

The void* function parameter permits a reference to any type to be passed into the task when the task is created. Where more than one parameter is required a pointer to a structure can be used. See the API documentation for the xTaskCreate() function for further information.

2.3.1.2 Task States

Only one task can actually be executing at any one time. The scheduler is responsible for selecting the task to execute in accordance with each task's relative priority and state.

A task can exist in one of the states described by the Table 'Task States', with valid transitions between states depicted by the Figure 'Valid task state transitions'.

Table 2-3 Task States

Task State	Description
Running	When a task is actually executing it is said to be in the Running state. It is the task selected by the scheduler to execute and is currently utilizing the processor. Only one task can be in the Running state at any given time.
Blocked	A task is in the Blocked state if it is waiting for an event. It cannot continue until the event occurs and until that time cannot be selected by the scheduler as the task to enter the Running state. Tasks in the Blocked state always have a timeout period, after which the task will become unblocked.
Suspended	A task will enter the Suspended state when it is the subject of a call to the xTaskSuspend() API function, and remain in the Suspended state until unsuspended by a call to the xTaskResume() API function. A timeout period cannot be specified. Suspended state tasks cannot be selected by the scheduler as the task to enter the Running state.
Ready	A task is in the Ready state if it is able to enter the Running state (it is not in the Blocked or Suspended state) but is not currently the task that is selected to execute. The only tasks that are available to the scheduler for selection as the task to enter the Running state are those that are in the Ready state. Ready is the initial state when a task is created.

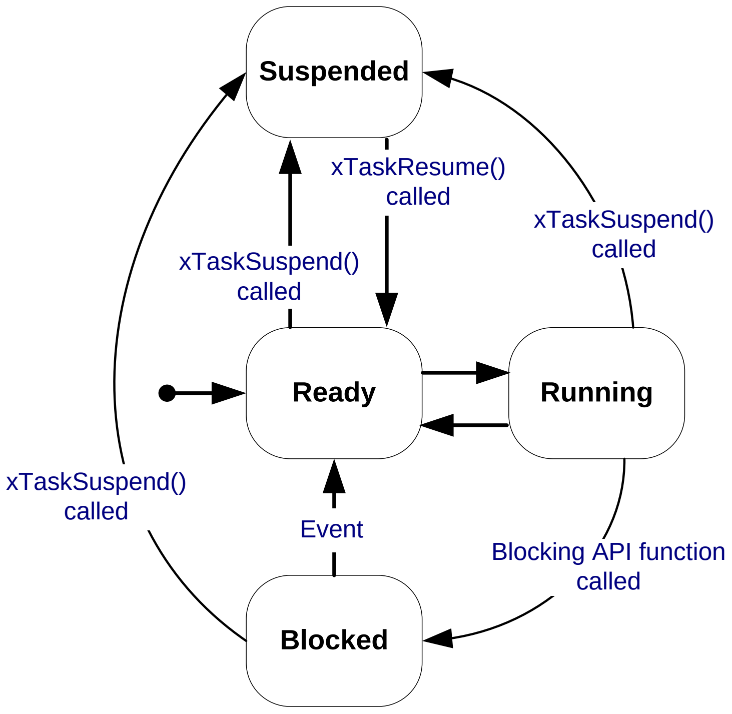

Figure 1 Valid task state transitions

Each task executes within its own context. The process of transitioning one task out of the Running state while transitioning another task into the Running state is called 'context switching'.

A call to the xTaskSuspend() API function can cause a task in the Running state, Blocked state or Ready state to enter the Suspended state.

Calls to the xTaskDelay() and xTaskDelayUntil() API functions can cause a task in the Running state to enter the Blocked state to wait for a temporal event - the event being the expiration of the requested delay period.

Calls to the xQueueSend() and xQueueReceive() API functions can cause a task in the Running state to enter the Blocked state to wait for a queue event - the event being either data being added to or removed from a queue. Section 'Intertask Communication' provides more information on using Queues.

2.3.1.3 Task Priorities

A priority is assigned to each task when the task is created.

The priority of a task can be queried using the xTaskPriorityGet() API function and changed by using the xTaskPrioritySet() API function.

Low numeric values denote low priority tasks. The lowest priority value that can be assigned to a task is 0.

High numeric values denote high priority tasks. The maximum priority that can be assigned to a task is (configMAX_PRIORITIES - 1), where configMAX_PRIORITIES is a user specified value as described in CHAPTER 3 (applies only when source code is supplied).

2.3.2 The Scheduler

The ‘scheduler’ has responsibility for:

- Deciding which task will be the task selected to enter the Running state, and performing the context switching accordingly.
- Measuring the passage of time.
- Transitioning tasks from the Blocked state into the Ready state upon the expiration of a timeout period.

2.3.2.1 Measuring Time

A periodic (tick) timer interrupt is used to measure time. Refer to the port-specific documentation for information on the timer peripheral utilized by your port.

The time between two consecutive timer interrupts is defined to be one ‘tick’ period. Times are therefore measured and specified in ‘tick’ units.

The number of milliseconds between each tick is defined by the constant configTICK_RATE_HZ. Refer to CHAPTER 3 for further information.

2.3.2.2 The Scheduling Policy

The scheduler selects as the task to be in the Running state the highest priority task that would otherwise be in the Ready state. In other words, the task chosen to execute is the highest priority task that is able to execute. Tasks in the Blocked or Suspended state are not able to execute.

Different tasks can be assigned the same priority. When this is the case the tasks of equal priority are selected to enter the Running state in turn. Each task will execute for a maximum of one tick period before the scheduler selects another task of equal priority to enter the Running state.

⚠ While the scheduler will ensure that tasks of equal priority will be selected to enter the Running state in turn, it is not guaranteed that each such task will get an equal share of processing time.

Contact WITTENSTEIN high integrity systems if your application requires a different scheduling policy to that described here.

2.3.2.3 Starting the Scheduler

The scheduler is started using the xTaskStartScheduler() API function. See the Listing 'Using a gatekeeper task to control access to a resource' for an example usage scenario.

At least one task must be created prior to xTaskStartScheduler() being called.

Calling xTaskStartScheduler() causes the creation of the Idle task. The Idle task never enters the Blocked or Suspended state. It is created to ensure there is always at least one task that is able to enter the Running state. The idle task hook (callback) function can be utilized to execute application specific code within the idle task.

2.3.2.4 Yielding

Yielding is where a task volunteers to leave the Running state by re-entering the ready state. When a task yields the schedule re-evaluates which task should be in the Running state. If no tasks of higher or equal priority to the yielding task are in the Ready state then the yielding task shall again be selected as the task to enter the Running state.

A task can yield by explicitly calling the taskYIELD() macro, or by calling an API function that changes the state or priority of another task within the application.

2.3.2.5 Scheduler States

The scheduler can exist in one of the states described by the Table 'Scheduler States', with valid transitions between states depicted by the Figure 'Valid scheduler state transitions'.

Table 2-4 Scheduler States

Scheduler State	Description
Initialization	This is the initial state, prior to the scheduler being started. While in the Initialization state the scheduler has no control over the application execution. Tasks and queues can be created while the scheduler is in the Initialization state.
Active	While in the Active state the scheduler controls the application execution by selecting the task that is in the Running state as described in the Section 'The Scheduling Policy'.
Suspended	The Scheduler does not perform any context switching while in the Suspended state. The task that was in the Running state when the scheduler entered the Suspended state will remain in the Running state until the scheduler returns to the Active state.

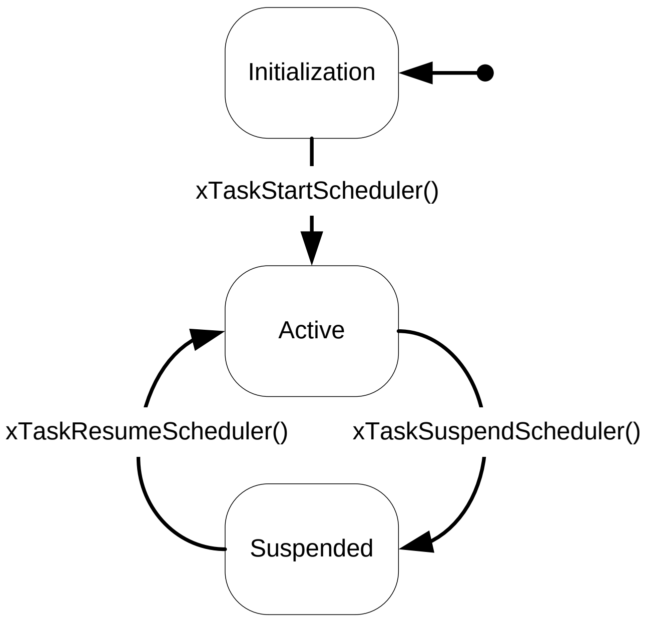

Figure 2 Valid scheduler state transitions

The scheduler enters the Suspended state following a call to xTaskSuspendScheduler(), and returns to the Active state following a call to xTaskResumeScheduler().

A code section that must be executed atomically (without interruption from other tasks or interrupts) to guarantee data integrity is called a critical region. The traditional method of implementing a critical region of code is to disable then re-enable interrupts as the critical region is entered then exited respectively. The macros taskENTER_CRITICAL() and taskEXIT_CRITICAL() are provided for this purpose.

Implementing a critical section through the use of taskENTER_CRITICAL() and taskEXIT_CRITICAL() has the disadvantage of the application being unresponsive to interrupts for the duration of the critical region. The scheduler suspension mechanism provides an alternative approach that permits interrupts to remain enabled during the critical region itself.

When the scheduler is in the Suspended state, by calling xTaskSuspendScheduler(), a switch to another task will never occur. The task executing the critical region is guaranteed to remain as the task in the Running state until xTaskResumeScheduler() is called.

⚠ Interrupts remain enabled while the scheduler is in the Suspended state. Critical regions implemented using the scheduler suspension mechanism therefore protect the critical data from access by other tasks, but not by interrupts. It is safe for an interrupt to access a queue while the scheduler is in the Suspended state.

⚠ A switch to a higher priority task that enters the Ready state while the scheduler is in the Suspended state will be held pending until xTaskResumeScheduler() is called. It is therefore still desirable for the scheduler not to be held in the Suspended state for an extended period. Doing so will reduce the responsiveness of high priority tasks.

2.3.2.6 Intertask Communication

SafeRTOS provides a queue implementation that permits data to be transferred safely between tasks. The queue mechanism removes the need for data that is shared between tasks to be declared globally, or for the application writer to concern themselves with mutual exclusion primitives when accessing the data.

The queue implementation is flexible and can be used to achieve a number of objectives, including simple data transfer, synchronization and semaphore type behavior.

2.3.2.7 Queue Characteristics

The following bullet points summarize the queue implementation:

- At any time a queue can contain zero or more ‘items’.
- The size of each item and the maximum number of items that the queue can hold are configured when the queue is created.
- Items are sent to a queue using the xQueueSend() and xQueueSendFromISR() API functions.
- Items are read from a queue using the xQueueReceive() and xQueueReceiveFromISR() API functions.
- Queues are FIFO buffers - that is, the first item sent to a queue using xQueueSend() (or xQueueSendFromISR()) is the first item retrieved from the queue when using xQueueReceive() (or xQueueReceiveFromISR()).
- Data transferred through a queue is done so by copy - the data is copied byte for byte into the queue when the data is sent, and then copied byte for byte out of the queue when the data is subsequently received.

2.3.2.8 Queue Events

Data being sent to or received from a queue is called a queue ‘event’.

When calling xQueueSend() a task can specify a period during which it should be held in the Blocked state to wait for space to become available on the queue if it finds the queue to already be full. The task is blocking on a queue event and will leave the Blocked state automatically when another task or interrupt removes an item from the queue.

When calling xQueueReceive() a task can specify a period during which it should be held in the Blocked state to wait for data to become available from the queue if it finds the queue to already be empty. Again the task is blocking on a queue event and will leave the Blocked state automatically when another task or interrupt writes data to the queue.

If more than one task is blocked waiting for the same event then the task unblocked upon the occurrence of the event is the task that has the highest priority. Where more than one task of the same priority are blocked waiting for the same event then the task unblocked upon the occurrence of the event will be the task that has been in the Blocked state for the longest time.

2.3.2.9 Data Formatting

The queue sender and receiver must agree on the meaning of the data placed in the queue. This could be a simple data type, such as a char or long, or a compound data type, such as a structure containing a number of complex data items. For example, a structure may be used to hold both a data value and the identity of the task sending the data.

Should the amount of data requiring transfer in each item be large then it may be preferable to queue a pointer to the data rather than the data itself. This will be more efficient as only the pointer value need be copied (typically 4 bytes) rather than each byte of the data itself.

⚠ When data is sent to a queue by copy then the queue implementation ensures access is consistent and mutual exclusion primitives are not required when accessing the data. When data is queued by reference (that is, a pointer to the data is queued rather than the data itself) then each task with access to the referenced data must agree how consistent and exclusive access is to be achieved.

2.3.2.10 Using Queues as Binary Semaphores

Semaphores are a means for a task to signal that it wishes to have exclusive access to data or other resources. While the task 'has' the semaphore other tasks know they are excluded from accessing the protected resource.

To be permitted access to the resource the task must first 'take' the semaphore, and when it has finished with the resource 'give' the semaphore back. If it cannot 'take' the semaphore it knows the resource is already in use by another task and it must wait for the semaphore to become available. If a task chooses to enter the Blocked state to wait for a semaphore it will automatically be moved back to the Ready state as soon as the semaphore is available.

FreeRTOS.org provided a binary semaphore implementation as a set of macros that simply called queue functions. A binary semaphore can be considered to be a queue that can contain, as a maximum, one item. For efficiency the item size can be zero, thus preventing any data actually being copied into and out of the queue. The important information is whether or not the queue is empty or full (the only two states as it can only contain one item), not the value of the data it contains.

When the resource is available the queue (representing the semaphore) is full. To 'take' the semaphore the task simply receives from the queue - resulting in the queue being empty. To 'give' the semaphore the task simply sends to the queue, resulting in the queue again being full. If, when attempting to receive from the queue, it finds the queue is already empty a task knows it cannot access the resource and can choose whether or not it wishes to enter the Blocked state to wait for the resource to become available again.

The Listing 'Using queues to implement binary semaphores' provides example semaphore 'Create', 'Take' and 'Give' macros that use the SafeRTOS queue implementation. Refer to CHAPTER 4 for reference information on the API functions used (xQueueCreate(), xQueueReceive() and xQueueSend()).

```
#define vSemaphoreCreateBinary( pcSemaphoreBuffer, xSemaphore )                          \
{                                                                                        \
       if( xQueueCreate( pcSemaphoreBuffer, portQUEUE_OVERHEAD_BYTES,                    \
              semBINARY_SEMAPHORE_QUEUE_LENGTH, semSEMAPHORE_QUEUE_ITEM_LENGTH,          \
              &xSemaphore ) == pdPASS )                                                  \
       {                                                                                 \
              xSemaphoreGive( xSemaphore );                                              \
       }                                                                                 \
       else                                                                              \
       {                                                                                 \
              xSemaphore = NULL;                                                         \
       }                                                                                 \
}

#define xSemaphoreTake( xSemaphore, xBlockTime )                                         \
       xQueueReceive( ( xQueueHandle ) xSemaphore, NULL, xBlockTime )

#define xSemaphoreGive( xSemaphore )                                                     \
       xQueueSend( ( xQueueHandle ) xSemaphore, NULL, semGIVE_BLOCK_TIME )

#define xSemaphoreGiveFromISR( xSemaphore, pxHigherPriorityTaskWoken )                   \
       xQueueSendFromISR( ( xQueueHandle ) xSemaphore, NULL, pxHigherPriorityTaskWoken )
```

Listing 4 Using queues to implement binary semaphores

Counting semaphores can be implemented in a similar fashion.

Where semaphores are used to control access to a resource, consideration should be given to whether or not including a 'gatekeeper' task would provide a neater application solution. A 'gatekeeper' task is a task that has exclusive access to the kept resource. As an example, consider an application where more than one task wishes to write messages to stdout. stdout can be controlled by a gatekeeper task. When a task wants to display a message, instead of writing to the display directly the message is instead sent to the stdout gatekeeper through a queue. The gatekeeper spends most of its time Blocked on a queue, but is woken by arriving messages at which point it removes the message from the queue and writes it to the display before re-entering the Blocked state. This is demonstrated in the Listing 'Using a gatekeeper task to control access to a resource'.

```
xQueueHandle xPrintQueue;

int main( void )
{
	/* Create the gatekeeper queue.  Its length is 5 and itemsize equal to sizeof( char * ). */
	xQueueCreate( pcQueueMemory, uxBufferLengthBytes, 5, sizeof( portCHAR * ), &xPrintQUeue );

	/* Create the gatekeeper task. */
	xTaskCreate(  vGateKeeperTask,     /* The function to execute.                   */
	              "stdout keeper",     /* The name of the task.                      */
	              pcStackBuffer1,      /* The memory to be used to create the task.  */
	              400,                 /* The stack size.                            */
	              NULL,                /* We are not passing in any parameters.      */
	              2,                   /* The priority.                              */
	              NULL );              /* We are not storing the task handle.        */

	/* Create the task that uses stdout. */
	xTaskCreate(  vAnotherTask,        /* The function to execute.                   */
	              "Another task",      /* The name of the task.                      */
	              pcStackBuffer2,      /* The memory to be used to create the task.  */
	              400,                 /* The stack size.                            */
	              NULL,                /* We are not passing in any parameters.      */
	              1,                   /* The priority.                              */
	              NULL );              /* We are not storing the task handle.        */

	/* Start the scheduler to run the created tasks. */
	xTaskStartScheduler( pdFALSE );

	/* Will not reach here as the scheduler is now running the tasks. */
	return 1;
}

/* The gate keeper task implementation. ------------------------------------------------ */
void vGateKeeperTask( void *pvParameters )
{
portCHAR *pcMessage;

	for( ;; )
	{
		/* Wait for a message to arrive. */
		xQueueReceive( xPrintQueue, &pcMessage, portMAX_DELAY );

		/* Write the message to stdout. */
		printf( "%s", pcMessage );
	}
}

/* A task that wants to write to stdout. ----------------------------------------------- */
void vAnotherTask( void *pvParameters )
{
const portCHAR *pcMessage1 = "Message to display 1\r\n";

	for( ;; )
	{
		/* Task code goes here....
		At some point the task wants to write to stdout so generates
		the string to send (in this case its just a constant) and
		sends it to the gatekeeper task. */

		xQueueSend( xPrintQueue, &pcMessage1, 0 );

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

Listing 5 Using a gatekeeper task to control access to a resource

2.3.3 Communication Between Tasks and Interrupts

⚠ Interrupt handlers must not under any circumstances call an API function that could cause a task to block. For this reason xQueueSend() and xQueueReceive() must not be called from within an ISR and xQueueSendFromISR() and xQueueReceiveFromISR() must be used in their place.

xQueueSendFromISR() and xQueueReceiveFromISR() (interrupt safe versions of xQueueSend() and xQueueReceive()) are often used to unblock a task upon the occurrence on an interrupt (see the Section 'Interrupts' regarding interrupt management). However for efficiency reasons it is not advised to make multiple calls within a single ISR in order to send or receive lots of small data items. Instead multiple data items should be packed into a single queue-able object. Alternatively a simple buffering scheme could be used, followed by a single call to an API function to unblock the task required to process the buffered data.

2.3.4 Interrupts

In the interest of stack usage predictability and to facilitate system behavioral analysis it is preferred that interrupt handlers do nothing but collect event data and clear the interrupt source - and therefore exit very promptly by deferring the processing of the event data to the task level. Task level processing can be performed with interrupts enabled. This scenario is demonstrated by the Listing 'Deferring interrupt processing to the task level'.

```
void vISRFunction( void )
{
char cData;
portBASE_TYPE xTaskWoken = pdFALSE;

       /* Read the data input from the peripheral that triggered the interrupt. */
       cData = ReceivedValue;

       /* Send the data to the peripheral handler task. */
       xQueueSendFromISR( xPrintQueue, &cData, &xTaskWoken );

       /* If the peripheral handler task has a priority higher than the interrupted
          task request a switch to the handler task. */
       taskYIELD_FROM_ISR( xTaskWoken );

       /* Clear interrupt here.  If taskYIELD_FROM_ISR() was called then the interrupt
          will return directly to the handler task where cData will be processed contiguous
          in time with the ISR exiting. */
}

void vPeripheralHandlerTask( void *pvParameters )
{
portCHAR *pcMessage;

       for( ;; )
       {
              /* Wait for a message to arrive. */
              xQueueReceive( xPrintQueue, &pcMessage, portMAX_DELAY );

              /* Write the message to stdout. */
              printf( "%s", pcMessage );
       }
}
```

Listing 6 Deferring interrupt processing to the task level

This scheme has the added advantage of flexible event processing prioritization. Task priorities are used instead of the prioritization being dependent on the priority assigned to each interrupt source by the target processor. The prioritization of peripheral handler tasks would normally be chosen to be higher than ordinary tasks within the same application - thereby allowing the interrupt handler to return directly into the peripheral handler task for immediate processing.

Refer to the documentation specific to your port for further information on writing interrupt service routines - in particular whether or not the port you are using permits interrupts to become nested.

⚠ Interrupt service routines that call API functions must not be permitted to execute prior to the scheduler being started. The easiest method of ensuring this is for interrupts to remain disabled until after the scheduler is started. Interrupts will automatically be enabled when the first task starts executing.

⚠ Refer to your port specific documentation for information on whether or not interrupts are permitted to nest, and the interrupt priorities from which SAFERTOS API functions can be called.

⚠ Calling API function while the scheduler is in the Initializing state will result in interrupts becoming disabled.

⚠ API functions that do not end in “FromISR” or macros that do not end in “FROM_ISR” must not be used within an interrupt service routine.

WITTENSTEIN

CHAPTER 3 INSTALLATION AND CONFIGURATION

3.1 INSTALLATION

3.1.1 Source Code and Libraries

SafeRTOS is supplied as either a set of source files, a library and set of header files, or, depending on the processor, pre-programmed in to the processor ROM. If you are using source files or library and header files, these files must be built as part of your application. Instructions on including the files in your application are contained in the port specific documentation.

3.1.2 Hook Functions

The host application (the application that uses SafeRTOS) is required to provide three hook (or callback) functions.

3.1.2.1 vApplicationErrorHook()

vApplicationErrorHook() is called upon the detection of a fatal error - either a corruption within the scheduler data structures or a potential stack overflow while performing a context switch. It has the prototype demonstrated in the Figure 'vApplicationErrorHook() Function Prototype'.

```
void vApplicationErrorHook( xTaskHandle xCurrentTask, signed portCHAR *pcErrorString, portBASE_TYPE xErrorCode );
```

Figure 3 vApplicationErrorHook() Function Prototype

vApplicationErrorHook() enables the host application to perform application specific error handling to ensure the system is placed into a safe state.

⚠ vApplicationErrorHook() must not return.

⚠ vApplicationErrorHook() is called with interrupts disabled.

3.1.2.1.1 vApplicationErrorHook() Parameters

xCurrentTask	The handle to the task that was in the Running state when the error occurred.
pcErrorString	A text string related to the error. This may be an error message or the name of the task that was in the Running state when the error occurred.
xErrorCode	Can take the following values: errINVALID_TICK_VALUE errINVALID_TASK_SELECTED errTASK_STACK_OVERFLOW

3.1.2.2 vApplicationTaskDeleteHook()

vApplicationTaskDeleteHook() is called when a task is deleted. Its purpose is to inform the host application that the memory allocated by the application for use by the task is once again free for use for other purposes. It has the prototype demonstrated by the Figure 'vApplicationTaskDeleteHook() function prototype'.

```
void vApplicationTaskDeleteHook( xTaskHandle xDeletedTask );
```

Figure 4 vApplicationTaskDeleteHook() function prototype

3.1.2.2.1 vApplicationTaskDeleteHook() Parameters

xDeletedTask	The handle of the task that was deleted.

3.1.2.3 vApplicationIdleHook()

vApplicationIdleHook() is called repeatedly by the scheduler idle task to allow application specific functionality to be executed within the idle task context. It is common to use the idle task hook to perform low priority application specific background tasks, or simply put the processor into a low power sleep mode.

vApplicationIdleHook() has the prototype demonstrated by the Figure 'vApplicationIdleHook() function prototype'.

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

Figure 5 vApplicationIdleHook() function prototype

⚠ Code contained within vApplicationIdleHook() must never call an API function that could result in the idle task entering the blocked state.

⚠ Should vApplicationIdleHook() be used to place the processor into a low power mode then the mode chosen must not prevent tick interrupts from being serviced.

3.1.3 Configuration Constants

The host application is required to supply a header file called SafeRTOSConfig.h in which the constants described within the Table 'Application Configuration Definitions' must be defined.

Table 3-1 Application Configuration Definitions

Definition	Type	Description
configCPU_CLOCK_HZ	unsigned long	The frequency at which the timer peripheral used to generate the tick interrupt is running.
configTICK_RATE_HZ	portTickType	The frequency at which the tick interrupt should occur. ⚠ Refer to the port specific documentation for information on the frequency and accuracy limitations of your particular hardware.
configMAX_PRIORITIES	unsigned portBASE_TYPE	The maximum number of unique priorities. The maximum priority that can be assigned to a task is (configMAX_PRIORITIES - 1)
configMAX_TASK_NAME_LEN	Port dependent	The maximum number of characters (including the NULL terminator) the name assigned to a task can take.

Further configuration is performed at run time by calling the API function vTaskInitializeScheduler().

⚠ vTaskInitializeScheduler() must be the first SAFERTOS API function to be called, and must only be called once.

WITTENSTEIN

CHAPTER 4 API REFERENCE

4.1 TASK FUNCTIONS

4.1.1 vTaskInitializeScheduler()

```
task.h
void vTaskInitializeScheduler( signed portCHAR *pcInIdleTaskStackBuffer,
                               unsigned portLONG ulInIdleTaskStackSizeBytes,
                               unsigned portLONG ulAdditionalStackCheckMarginBytes,
                               const xPORT_INIT_PARAMETERS * const pxPortInitParameters
                             );
```

4.1.1.1 Summary

Initializes all scheduler private data and passes application specific configuration data to the scheduler and portable layer. This removes any reliance on the C startup code to perform this task.

4.1.1.2 Parameters

pcInIdleTaskStackBuffer	Pointer to the start of (lowest address) the buffer that should be used to hold the stack of the idle task.
ulInIdleTaskStackSizeBytes	The size in bytes of the buffer pointed to by the pcInIdleTaskStackBuffer parameter. This is effectively the size in bytes of the idle task stack.
ulAdditionalStackCheckMarginBytes	When moving a task out of the Running state the task context is saved onto the task stack. If following the save there would remain fewer than ulAdditionalStackCheckMarginBytes free bytes on the task stack the application error hook will be called. Therefore the higher the ulAdditionalStackCheckMarginBytes value the more sensitive the stack overflow checking becomes - zero is a valid value and will result in the least sensitive stack overflow checking. Note that when a potential stack overflow is detected the error hook is called without having actually saved the task context.
pxPortInitParameters	Initialization data to be passed to the portable layer. xPORT_INIT_PARAMTERS is defined differently for each port. Refer to the port specific documentation for further information.

4.1.1.3 Return Values

None.

4.1.1.4 Notes

⚠ vTaskInitializeScheduler() must be the first SAFERTOS API function to be called, and must only be called once.

4.1.1.5 Example

```
/* Allocate a buffer for use by the idle task as its stack.  The size required
   will depend on the port and application. */
static signed portCHAR cIdleTaskStack[ mainIDLE_TASK_STACK_DEPTH_BYTES ];

int main( void )
{
/* Setup a xPORT_INIT_PARAMETERS structure to configure the portable layer. */
xPORT_INIT_PARAMETERS xPortParameters =
{
       /* The number of parameters and the meaning of each parameter is dependent
          on the port in use.  Refer to the port specific documentation for more
          information. */
};

       /* Setup the hardware. */
       prvSetupHardware();

       /* Initialize the scheduler before calling any other API functions. */
       vTaskInitializeScheduler( cIdleTaskStack, mainIDLE_TASK_STACK_DEPTH_BYTES, 20, &xPortParameters );

       /* Other SafeRTOS API functions can be called from this point on. */
       ....
}
```

Listing 7 Example use of the vTaskInitializeScheduler() API function

4.1.2 xTaskCreate()

```
task.h
portBASE_TYPE xTaskCreate( pdTASK_CODE pvTaskCode,
                           const signed portCHAR * const pcName,
                           signed portCHAR * const pcStackBuffer,
                           unsigned portLONG ulStackDepthBytes,
                           void *pvParameters,
                           unsigned portBASE_TYPE uxPriority,
                           xTaskHandle *pxCreatedTask
                         );
```

4.1.2.1 Summary

Creates a new task. The created task is placed into the Ready state.

4.1.2.2 Parameters

pvTaskCode	Pointer to the function that implements the task.
pcName	A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by configMAX_TASK_NAME_LEN.
pcStackBuffer	Pointer to the start of the memory to be used as the task stack.
ulStackDepthBytes	The size in bytes of the memory pointed to by the pcStackBuffer pointer. The minimum allowable size for the stack buffer is port-dependent and documented within the port-specific documentation.

pvParameters	Task functions take a void * parameter - the value of which is set by pvParameters when the task is created.
uxPriority	The priority of the task. Can take any value between 0 and (configMAX_PRIORITIES - 1). The lower the numeric value of the assigned priority the lower the relative priority of the task.
pxCreatedTask	Used to pass back a handle by which the created task can be referenced, for example when changing the priority of the task or subsequently deleting the task.

4.1.2.3 Return Values

pdPASS	The task was created successfully.
errINVALID_TASK_CODE_POINTER	The pvTaskCode parameter was found to be NULL.
errINVALID_PRIORITY	The uxPriority parameter was greater than or equal to configMAX_PRIORITIES.
errSUPPLIED_BUFFER_TOO_SMALL	ulStackDepthBytes was less than the minimum specified in the port specific documentation.
errINVALID_BYTE_ALIGNMENT	The alignment of the pcStackBuffer value was not correct for the target hardware.
errNULL_PARAMETER_SUPPLIED	The value of pcStackBuffer was found to be NULL.

The handle to the created task is returned in the pxCreatedTask parameter.

4.1.2.4 Notes

A task can be created while the scheduler is in the Initialization state, or from another task while the scheduler is in the Running or Suspended state.

Creating a task while the scheduler is in the Active state can cause the task being created to enter the Running state prior to xTaskCreate() returning. This will occur if the task being created has a priority higher than the task calling xTaskCreate().

⚠ Calling xTaskCreate() while interrupts are disabled will not prevent the task being created entering the Running state should it have a priority higher than the task calling xTaskCreate(). The task being created will commence execution with interrupts enabled. Interrupts will once again be disabled when the task calling xTaskCreate() once again enters the Running state.

⚠ Calling xTaskCreate() while the scheduler was in the Suspended state would defer any necessary context switch until such time that the scheduler re-entered the Active state.

⚠ xTaskCreate() must not be called from an interrupt service routine.

4.1.2.5 Example

```
/* Define the priority at which the task is to be created. */
#define TASK_PRIORITY 1

/* Define the buffer to be used by the tasks stack. */
#define STACK_SIZE 400
const portCHAR cTaskStack[ STACK_SIZE ];

/* Define a structure 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 of the structure just defined.  A reference to this
 * variable is passed in as the task parameter. */
xStruct xParameter = { 1, 2 };

/* The task being created. */
void vTaskCode( void * pvParameters )
{
xStruct *pxParameters;

       /* Cast the parameter to the expected type. */
       pxParameters = ( xStruct * ) pvParameters;

       /* The parameter can now be accessed. */
       if( pxParameters->cStructMember1 != 1 )
       {
              /* Etc. */
       }

       /* Enter an infinite loop to perform the task processing. */
       for( ;; )
       {
              /* Task code goes here. */
       }
}

/* Function that creates a task.  This could be called while the scheduler was in the
 * Initialization state or from another task while the scheduler was in the Running or
 * Suspended state. */

void vAnotherFunction( void )
{
xTaskHandle xHandle;

       /* Create the task defined by the vTaskCode function, storing the handle. */
       if( xTaskCreate( vTaskCode,
                      "Demo task",
                      cTaskStack,
                      STACK_SIZE,
                      &xParameter,     /* Pass in the structure as the task parameter. */
                      TASK_PRIORITY,
                      &xHandle
                      ) != pdPASS )
       {
              /* The task was not successfully created.  The return value could have been
                 checked to find out why. */
       }
       else
       {
              /* The task was created successfully.  If this function is called from a task,
               * the scheduler is in the Active state, and the task just created has a priority
               * higher than the calling task then vTaskCode will have executed before this task
               * reaches this point. */
       }

       /* The handle can now be used in other API functions, for example to change the
        * priroity of the task. */
       if( xTaskPrioritySet( xHandle, 1 ) != pdPASS )
       {
              /* The priority was not changed. */
       }
```

```
        else
        {
                /* The priority was changed. */
        }
}
```

Listing 8 Example usage of the xTaskCreate() API function

4.1.3 xTaskDelete()

```
task.h
portBASE_TYPE xTaskDelete( xTaskHandle pxTaskToDelete );
```

4.1.3.1 Summary

Deletes the task referenced by the pxTaskToDelete parameter.

4.1.3.2 Parameters

pxTaskToDelete	The handle of the task to be deleted. The handle to a task is obtained via the pxCreatedTask parameter to the xTaskCreate() API function when the task is created. A task may delete itself by passing NULL as the pxTaskToDelete parameter.

4.1.3.3 Return Values

pdPASS	The task was successfully deleted.
errINVALID_TASK_HANDLE	The pxTaskToDelete parameter was not found to reference a valid task.

4.1.3.4 Notes

Deleting a task will cause the task delete hook function to be called (see the Section 'vApplicationTaskDeleteHook()'). This lets the host application know that the memory that was used by the task is now free for reuse.

The handle of the deleted task will be invalidated and cannot therefore be used in further API function calls. Attempting to do so will result in the API function returning an error.

⚠ xTaskDelete() must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

⚠ xTaskDelete() must not be called to delete the calling task while the scheduler is in the Suspended state as while the scheduler is suspended a switch away from the task being deleted cannot be performed.

⚠ xTaskDelete() must not be called from an interrupt service routine.

⚠ xTaskDelete() must not be used to delete the idle task unless at least one other task has been created that is guaranteed never to enter the Blocked or Suspended state.

⚠ Once a task has been deleted the memory allocated for use as the task stack can be reused. If the same memory buffer is passed into another call to xTaskCreate() (to create a new task) then the handle of the deleted task and the handle of the newly created task will be identical.

4.1.3.5 Example

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

       /* Create a task, storing the handle. */
       if( xTaskCreate( vTaskCode,
                     "Demo task",
                     cTaskStack,
                     STACK_SIZE,
                     NULL,
                     TASK_PRIORITY,
                     &xHandle
                    ) != pdPASS )
       {
              /* The task was not created successfully.  The return value could have
               * been checked to find out why. */
       }
       else
       {
              /* Use the handle obtained when the task was created to delete the task. */
              if( xTaskDelete( xHandle ) != pdPASS )
              {
                     /* The task could not be deleted.  The return value could have been
                      * checked to find out why. */
              }
              else
              {
                     /* The task was deleted and execution will never reach here. */
              }
       }

       /* Delete ourselves. */
       xTaskDelete( NULL );
}
```

Listing 9 Example use of the xTaskDelete() API function

4.1.4 xTaskDelay()

```
task.h
portBASE_TYPE xTaskDelay( portTickType xTicksToDelay );
```

4.1.4.1 Summary

Places the calling task into the Blocked state for a fixed number of tick periods. The task therefore delays for the requested number of ticks before being transitioned back into the Ready state.

4.1.4.2 Parameters

xTicksToDelay	The number of ticks for which the calling task should be held in the Blocked state.

4.1.4.3 Return Values

pdPASS	The calling task was held in the Blocked state for the specified number of ticks.
errSCHEDULER_IS_SUSPENDED	The scheduler was in the Suspended state when xTaskDelay() was called. The scheduler cannot select a different task to enter the Running state when it is suspended and therefore is unable to transition the calling task into the Blocked state.

4.1.4.4 Notes

The actual time between a task calling xTaskDelay() to enter the Blocked state, and then subsequently being moved back to the Ready state, can only be specified to the available time resolution. If xTaskDelay() is called a fraction of a tick period prior to the next tick increment then this fraction will count as one of the tick periods for which the task is to be held in the Blocked state.

Specifying a delay period of 0 ticks will not cause the task to enter the Blocked state but will cause the task to yield. It has the same effect as calling taskYIELD().

⚠ xTaskDelay() 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).

⚠ xTaskDelay() must not be called from within an interrupt service routine.

⚠ Calling xTaskDelay() while interrupts are disabled will not prevent the task from entering the Blocked state and a different task being selected as the task to enter the Running state. Each task maintains its own interrupt state and therefore the task entering the Running state could have interrupts enabled. Interrupts would once again be disabled when the task calling xTaskDelay() re-entered the Running state.

4.1.4.5 Example

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

		/* Delay for a fixed period. */
		if( xTaskDelay( 20 ) == pdPASS )
		{
			/* The scheduler was not suspended. */
		}

		/* 20 ticks will have passed since calling xTaskDelay() prior to reaching here. */
	}
}
```

Listing 10 Example of using the xTaskDelay() API function.

4.1.5 xTaskDelayUntil()

```
task.h
portBASE_TYPE xTaskDelayUntil( portTickType *pxPreviousWakeTime, portTickType xTimeIncrement );
```

4.1.5.1 Summary

Places the calling task into the Blocked state until an absolute time is reached.

4.1.5.1.1 Differences Between xTaskDelay() and xTaskDelayUntil()

xTaskDelay() will cause the calling task to enter the Blocked state for the specified number of ticks from the time xTaskDelay() was called. Therefore xTaskDelay() specifies a delay period relative to the time at which the function is called. xTaskDelayUntil() instead specifies the absolute (exact) time at which it wishes to re-enter the Ready state.

xTaskDelayUntil() can be used by cyclical tasks to ensure a constant execution frequency. It is difficult to use xTaskDelay() for this purpose as the time taken between cycles of the task will not be fixed (the task may take a different path though the code between calls, or may get interrupted or pre-empted a different number of times each time it executes) making it impossible to specify a relative delay period with any accuracy.

4.1.5.2 Parameters

pxPreviousWakeTime	Pointer to a variable that holds the time at which the task was last unblocked. The variable must be initialized with the current time prior to its first use (see the example below). Following this the variable is automatically updated within xTaskDelayUntil().
xTimeIncrement	The cycle time period. The task will be unblocked at time (*pxPreviousWakeTime + xTimeIncrement).

4.1.5.3 Return Values

pdPASS	The calling task was held in the Blocked state until the specified time.
errSCHEDULER_IS_SUSPENDED	The scheduler was in the Suspended state when xTaskDelayUntil() was called. The scheduler cannot select a different task to enter the Running state when it is suspended and therefore is unable to transition the calling task into the Blocked state.
errDID_NOT_YIELD	The parameters passed into the function were valid, but the time at which the task specified that it should re-enter the Ready state has already passed. The task did not enter the Blocked state and a yield was not performed.

4.1.5.4 Notes

⚠ xTaskDelayUntil() 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).

⚠ xTaskDelayUntil() must not be called from within an interrupt service routine.

⚠ Calling xTaskDelayUntil() while interrupts are disabled will not prevent the task from entering the Blocked state and a different task being selected as the task to enter the Running state. Each task maintains its own interrupt state and therefore the task entering the Running state could have interrupts enabled. Interrupts would once again be disabled when the task calling xTaskDelayUntil() re-entered the Running state.

4.1.5.5 Example

```
/* A function that performs an action every 50 ticks. */
void vCyclicTaskFunction( void * pvParameters )
{
portTickType xLastWakeTime;
const portTickType xFrequency = 50;

       /* Initialize the xLastWakeTime variable with the current time. */
       xLastWakeTime = xTaskGetTickCount();

       /* Enter the loop that defines the task behavior. */
       for( ;; )
       {
              /* Wait for the next cycle. */
              if( xTaskDelayUntil( &xLastWakeTime, xFrequency ) != pdPASS )
              {
                     /* The scheduler is not suspended so it must have taken longer than 50
                      * ticks to perform a cycle of this task. */
              }

              /* Perform task action here.  This code will be executed every 50 ticks.
               *
               * xLastWakeTime is automatically updated by the xTaskDelayUntil() function
               * so need not be modified once it has been initialized. */
       }
}
```

Listing 11 Example of using the xTaskDelayUntil() API function

4.1.6 xTaskPriorityGet()

```
task.h
portBASE_TYPE xTaskPriorityGet( xTaskHandle pxTask, unsigned portBASE_TYPE *puxPriority );
```

4.1.6.1 Summary

Queries the priority of a task.

4.1.6.2 Parameters

pxTask	The handle of the task being queried. The handle to a task is obtained via the pxCreatedTask parameter to the xTaskCreate() API function when the task is created. A task may query its own priority by passing NULL as the pxTask parameter.
puxPriority	Pointer to the variable that will be set to the priority of the task being queried.

4.1.6.3 Return Values

pdPASS	*puxPriority was set to the priority of the task being queried.
errNULL_PARAMETER_SUPPLIED	puxPriority was found to be NULL.
errINVALID_TASK_HANDLE	pxTask was found not to be a valid task handle.

4.1.6.4 Notes

⚠ xTaskPriorityGet() must not be called from within an interrupt service routine.

4.1.6.5 Example

```
void vAFunction( void )
{
xTaskHandle xHandle;
unsigned portBASE_TYPE uxCreatedPriority, uxOurPriority;

       /* Create a task, storing the handle. */
       if( xTaskCreate( vTaskCode,
                      "Demo task",
                      cTaskStack,
                      STACK_SIZE,
                      NULL,
                      TASK_PRIORITY,
                      &xHandle
                     ) != pdPASS )
       {
              /* The task was not created successfully.  The return value
               * could have been checked to find out why. */
       }
       else
       {
              /* Use the handle to query the priority of the created task. */
              if( xTaskPriorityGet( xHandle, &uxCreatedPriority ) != pdPASS )
              {
                     /* Could not obtain the task priority.  The return value could have been
                      * checked to find out why. */
              }

              /* Query our own priority. */
              if( xTaskPriorityGet( NULL, &uxOurPriority ) != pdPASS )
              {
                     /* Could not obtain our own priority - should never get here when using NULL. */
              }

              /* Is our priority higher than the priority of the task just created? */
              if( uxOurPriority > uxCreatedPriority )
              {
                     /* Yes. */
              }
       }
}
```

Listing 12 Example of using the xTaskPriorityGet() API function

4.1.7 xTaskPrioritySet()

```
task.h
portBASE_TYPE xTaskPrioritySet( xTaskHandle pxTask, unsigned portBASE_TYPE uxNewPriority );
```

4.1.7.1 Summary

Changes the priority of a task.

4.1.7.2 Parameters

pxTask	The handle of the task being modified. The handle to a task is obtained via the pxCreatedTask parameter to the xTaskCreate() API function when the task is created. A task may change its own priority by passing NULL as the pxTask parameter.
uxNewPriority	The priority to which the task identified by the pxTask parameter should be set.

4.1.7.3 Return Values

pdPASS	The priority of the task was changed.
errINVALID_TASK_HANDLE	pxTask was found not to be a valid task handle.
errINVALID_PRIORITY	The value of uxNewPriority was greater than the highest available task priority (configMAX_PRIORITIES - 1).

4.1.7.4 Notes

⚠ xTaskPrioritySet() must not be called from within an interrupt service routine.

⚠ xTaskPrioritySet() 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 not recommended that xTaskPrioritySet() be used to modify the priority of the idle task. The idle task never enters the Blocked or Suspended state so will completely starve lower priority tasks of execution time should its priority not be the lowest (or be equal to the lowest) priority in the application.

⚠ It is possible for more than one task to be in the Blocked state while waiting for an event to occur on the same queue. When this is the case the set of tasks that are waiting for the same event are referenced in priority order. When the queue event occurs it is the task that is referenced first that is moved out of the Blocked state and into the Ready state - thus ensuring (due to the priority ordering) that it is the highest priority task that is unblocked. Using xTaskPrioritySet() to change the priority of a task that is one of a set of tasks blocked to wait for an event does not force the series in which the tasks are referenced to be reordered. This could lead to a queue event transitioning a task into the Ready state when there is a task of higher priority waiting for the same event.

⚠ Calling xTaskPrioritySet() can result in a context switch being performed. Each task maintains its own interrupt state, therefore calling xTaskPrioritySet() while interrupts are disabled could cause a context switch to a task that has interrupts enabled. Interrupts would once again be disabled when the task calling xTaskPrioritySet() next entered the Running state.

⚠ Calling xTaskPrioritySet() while the scheduler was in the Suspended state would defer any necessary context switch until such time that the scheduler re-entered the Active state.

4.1.7.5 Example

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

       /* Create a task, storing the handle. */
       if( xTaskCreate( vTaskCode,
                      "Demo task",
                      cTaskStack,
                      STACK_SIZE,
                      NULL,
                      TASK_PRIORITY,
                      &xHandle
                     ) != pdPASS )
       {
              /* The task was not created successfully.  The return value could
               * have been checked to find out why. */
       }
       else
       {
              /* Use the handle to raise the priority of the created task. */
              vTaskPrioritySet( xHandle, TASK_PRIORITY + 1 );

              /* Use a NULL handle to modify our own priority. */
              vTaskPrioritySet( NULL, 1 );
       }
}
```

Listing 13 Example of using the xTaskPrioritySet() API function

4.1.8 xTaskSuspend()

```
task.h
portBASE_TYPE xTaskSuspend( xTaskHandle pxTaskToSuspend );
```

4.1.8.1 Summary

Places a task into the Suspended state.

4.1.8.2 Parameters

pxTaskToSuspend	The handle of the task being suspended. The handle to a task is obtained via the pxCreatedTask parameter to the xTaskCreate() API function when the task is created. A task may suspend itself by passing NULL as the pxTaskToSuspend parameter.

4.1.8.3 Return Values

pdPASS	The task was successfully suspended.

errSCHEDULER_IS_SUSPENDED	The scheduler was in the Suspended state when xTaskSuspend() was called. The scheduler cannot select a different task to enter the Running state when it is suspended and therefore would be unable to select a new task to run if a task suspended itself.
errINVALID_TASK_HANDLE	pxTaskToSuspend was found not to be a valid task handle.
errTASK_ALREADY_SUSPENDED	The task referenced by the pxTaskToSuspend handle was already in the Suspended state.

4.1.8.4 Notes

⚠ xTaskSuspend() must not be called from within an interrupt service routine.

⚠ xTaskSuspend() 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).

⚠ xTaskSuspend() must not be used to suspend the idle task unless at least one other task has been created that is guaranteed never to enter the Blocked or Suspended state.

⚠ Calling xTaskSuspend() can result in a context switch being performed. Each task maintains its own interrupt state, therefore calling xTaskSuspend() while interrupts are disabled could cause a context switch to a task that has interrupts enabled. Interrupts would once again be disabled when the task calling xTaskSuspend() next entered the Running state.

4.1.8.5 Example

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

       /* Create a task, storing the handle. */
       if( xTaskCreate( vTaskCode,
                      "Demo task",
                      cTaskStack,
                      STACK_SIZE,
                      NULL,
                      TASK_PRIORITY,
                      &xHandle
                     ) != pdPASS )
       {
              /* The task was not created successfully.  The return value could have been
               * checked to find out why. */
       }
       else
       {
              /* Use the handle to suspend the created task. */
              if( xTaskSuspend( xHandle ) != pdPASS )
              {
                     /* Could not suspend the task.  The return value could have been checked to
                      * find out why. */
              }

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

              /* Suspend ourselves. */
              xTaskSuspend( NULL );

              /* We cannot reach here unless another task calls xTaskResume() with the handle
               * to the task from which this function was called as the parameter. */
       }
}
```

Listing 14 Example of using the xTaskSuspend() API function

4.1.9 xTaskResume()

```
task.h
portBASE_TYPE xTaskResume( xTaskHandle pxTaskToResume );
```

4.1.9.1 Summary

Transition a task from the Suspended state to the Ready state. The task must have previously been suspended using a call to xTaskSuspend().

4.1.9.2 Parameters

pxTaskToResume	The handle of the task being resumed - transitioned out of the Suspended state. The handle to a task is obtained via the pxCreatedTask parameter to the xTaskCreate() API function when the task is created.

4.1.9.3 Return Values

pdPASS	The task was successfully resumed - transitioned out of the Suspended state.
errNULL_PARAMETER_SUPPLIED	pxTaskToResume was found to be NULL.
errINVALID_TASK_HANDLE	pxTaskToResume was found not to be a valid task handle (and not NULL).
errTASK_WAS_NOT_SUSPENDED	The task referenced by the pxTaskToResume handle was not in the Suspended state.

4.1.9.4 Notes

A task can block 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 xTaskSuspend(), then out of the Suspended state and into the Ready state using a call to xTaskResume(). 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 block to wait for a temporal event using the xTaskDelay() or xTaskDelayUntil() API functions. It is legitimate to move such a Blocked task into the Suspended state using a call to xTaskSuspend(), then out of the Suspended state and into the Ready state using a call to xTaskResume(). Following this scenario, the next time the task enters the Running state it shall exit the xTaskDelay() or xTaskDelayUntil() function as if the specified delay period had expired, even if this is not actually the case.

⚠ xTaskResume() must not be called from within an interrupt service routine.

⚠ xTaskResume() 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).

⚠ Calling xTaskResume() can result in a context switch being performed. Each task maintains its own interrupt state, therefore calling xTaskResume() while interrupts are disabled could cause a context switch to a task that has interrupts enabled. Interrupts would once again be disabled when the task calling xTaskResume() next entered the Running state.

⚠ Calling xTaskResume() while the scheduler was in the Suspended state would defer any necessary context switch until such time that the scheduler re-entered the Active state.

4.1.9.5 Example

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

       /* Create a task, storing the handle. */
       if( xTaskCreate( vTaskCode,
                      "Demo task",
                      cTaskStack,
                      STACK_SIZE,
                      NULL,
                      TASK_PRIORITY,
                      &xHandle
                     ) != pdPASS )
       {
              /* The task was not created successfully.  The return value could have been checked
               * to find out why. */
       }
       else
       {
              /* Use the handle to suspend the created task.  The return value should be checked to
               * ensure the task is successfully suspended. */
              xTaskSuspend( xHandle );

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

              /* Resume the suspended task again. */
              if( xTaskResume( xHandle ) != pdPASS )
              {
                     /* Could not resume the task.  The return value could have been checked to find
                      * out why. */
              }

              /* The created task is again available to the scheduler. */
       }
}
```

Listing 15 Example of using the xTaskResume() API function

4.2 SCHEDULER CONTROL FUNCTIONS

4.2.1 xTaskStartScheduler()

```
task.h
portBASE_TYPE xTaskStartScheduler( portBASE_TYPE xUseKernelConfigurationCheck );
```

4.2.1.1 Summary

Starts the scheduler by transitioning the scheduler from the Initialization state into the Active state.

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

4.2.1.2 Parameters

xUseKernelConfigurationCheck	A Boolean which indicates whether the kernel configuration parameters should be checked or not.

4.2.1.3 Return Values

errNO_TASKS_CREATED	A task was not created prior to calling xTaskStartScheduler().
errSCHEDULER_ALREADY_RUNNING	The scheduler is already in the Active state.
errCOULD_NOT_START_IDLE_TASK	The scheduler could not be started as an error was encountered while creating the idle task.

xTaskStartScheduler() will not return if the scheduler is started successfully.

4.2.1.4 Notes

⚠ xTaskStartScheduler() must not be called from within an interrupt service routine.

⚠ Consult the port specific documentation for details of the architecture specific requirements that must be fulfilled prior to calling xTaskStartScheduler() - for example the processor mode from which the function can be called.

4.2.1.5 Example

See the Listing 'Using a gatekeeper task to control access to a resource'.

4.2.2 vTaskSuspendScheduler()

```
task.h
void vTaskSuspendScheduler( void );
```

4.2.2.1 Summary

Transitions the scheduler from the Active state to the Suspended state.

A context switch will not occur while the scheduler is in the Suspended state but instead be held pending until the scheduler re-enters the Active state.

4.2.2.2 Parameters

None.

4.2.2.3 Return Values

None.

4.2.2.4 Notes

Suspending the scheduler allows a task to execute without the risk of interference from other tasks.

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

⚠ vTaskSuspendScheduler() must not be called from an interrupt service routine.

⚠ Interrupts remain enabled while the scheduler is suspended.

⚠ The tick count value will not increase while scheduler is in the Suspended state (although tick interrupts are not missed).

⚠ vTaskSuspendScheduler() 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).

⚠ The count of nested calls to vTaskSuspendScheduler() will eventually overflow - with the maximum value that can be held in the type defined as portBASE_TYPE being the maximum nesting count that can be maintained.

4.2.2.5 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. */
       vTaskSuspendScheduler();

       /* Perform an action here. */

       /* As calls to vTaskSuspendScheduler() are nested resuming the scheduler
        * does not cause the scheduler to re-enter the active state at this time. */
       xTaskResumeScheduler();
}

void vTask1( void * pvParameters )
{
       for( ;; )
       {
              /* Perform some actions here. */

              /* At some point the task wants to perform a long 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. */
              vTaskSuspendScheduler();

              /* 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 vTaskSuspendScheduler can be nested so it is safe to
               * call a function which also calls vTaskSuspendScheduler. */
              vDemoFunction();

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

              if( xTaskResumeScheduler() == pdTRUE )
              {
                     /* A context switch occurred as we resumed the scheduler. */
              }
              else
              {
                     /* A context switch did not occur as we resumed the scheduler.
                      * Maybe we want to perform one here? */
                     taskYIELD();
              }
       }
}
```

Listing 16 Example of using the vTaskSuspendScheduler() and xTaskResumeScheduler() API functions

4.2.3 xTaskResumeScheduler()

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

4.2.3.1 Summary

Transitions the scheduler out of the Suspended state into the Active state.

4.2.3.2 Parameters

None.

4.2.3.3 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 vTaskSuspendScheduler().
errSCHEDULER_WAS_NOT_SUSPENDED	The scheduler was not in the Suspended state.

4.2.3.4 Notes

Calls to xTaskResumeScheduler() transition the scheduler out of the Suspended state following a previous call to vTaskSuspendScheduler(). Calls to vTaskSuspendScheduler() can be nested. The same number of calls must be made to xTaskResumeScheduler() as have previously been made to vTaskSuspendScheduler() before the scheduler will leave the Suspended state and re-enter the Active state.

⚠ xTaskResumeScheduler() must not be called from within an interrupt service routine.

⚠ xTaskResumeScheduler() 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).

⚠ Calling xTaskResumeScheduler() can result in a context switch being performed. Each task maintains its own interrupt state, therefore calling xTaskResumeScheduler() while interrupts are disabled could cause a context switch to a task that has interrupts enabled. Interrupts would once again be disabled when the task calling xTaskResumeScheduler() next entered the Running state.

4.2.3.5 Example

See the Listing 'Example of using the vTaskSuspendScheduler() and xTaskResumeScheduler() API functions'.

4.2.4 xTaskGetTickCount()

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

4.2.4.1 Summary

Returns the current tick value.

4.2.4.2 Parameters

None.

4.2.4.3 Return Values

xTaskGetTickCount() always returns the current tick count value.

4.2.4.4 Notes

Time is measured in ticks. xTaskGetTickCount() effectively returns the time since the scheduler was started.

⚠ xTaskGetTickCount() must not be called from an interrupt service routine.

⚠ The tick value will eventually overflow, returning to zero. The frequency at which this occurs is dependent both on the type chosen to hold the tick value (See the Section 'Project Definitions' for information about portTickType) and the frequency of the tick interrupt.

⚠ xTaskGetTickCount() will always return zero prior to a successful call to xTaskStartScheduler().

4.2.4.5 Example

```
void vAFunction( void )
{
portTickType xTime1, xTime2, xExecutionTime

       /* Get the time when 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 17 Example of using the xTaskGetTickCount() API function

4.2.5 taskYIELD()

```
task.h
Macro: taskYIELD()
```

4.2.5.1 Summary

Yield, as described in the Section 'Yielding'.

Yielding is where a task volunteers to leave the Running state by re-entering the ready state before using all of its time slice.

4.2.5.2 Parameters

None.

4.2.5.3 Return Values

None.

4.2.5.4 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).

⚠ Calling taskYIELD() while the scheduler is suspended will not result in a yield being performed until such a time that the scheduler re-enters the Active state. The yield is held pending.

⚠ taskYIELD() must not be called from an interrupt service routine.

⚠ Each task maintains its own interrupt status. Yielding when interrupts are disabled could cause a context switch to a task that has interrupts enabled. Interrupts would once again be disabled when the task calling taskYIELD() next enters the Running state.

4.2.5.5 Example

```
void vATask( void * pvParameters)
{
       for( ;; )
       {
              /* Perform some actions. */

              /* We are not desperate for processing time now.  If there are any tasks of
               * equal priority to this task that are in the Ready state then let them execute
               * now even though we have not used all of our time slice. */
              taskYIELD();

              /* If there were any tasks of equal priority to this task in the Ready state
               * then they will have executed before we reach here.  If there were no other
               * tasks of equal priority in the Ready state we would have just continued.
               *
               * There will not be any tasks of higher priority that are in the Ready state as
               * if there were this task would not be in the Running state in the first place. */
       }
}
```

Listing 18 Example of using the taskYIELD() API function

4.2.6 taskYIELD_FROM_ISR()

```
task.h
Macro: taskYIELD_FROM_ISR( xSwitchRequired )
```

4.2.6.1 Summary

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

4.2.6.2 Parameters

xSwitchRequired	Set to zero if a context switch is not required, or a non zero value if a context switch is required.

4.2.6.3 Return Values

None.

4.2.6.4 Notes

Calling either xQueueSendFromISR() or xQueueReceiveFromISR() within an interrupt service routine can potentially cause a task to leave the Blocked state - necessitating a context switch should the unblocked task have a priority higher than the interrupted task.

A context switch is performed transparently (within the API functions) when either xQueueSend() or xQueueReceive() cause a task of higher priority than the calling task to exit the Blocked state. This behavior is desirable from a task, but not from an interrupt service routine. Therefore xQueueSendFromISR() and xQueueReceiveFromISR(), rather than performing the context switch themselves, instead return a value indicative of whether a context switch is required. If a context switch is required the application writer can use taskYIELD_FROM_ISR() to perform the context switch at the most appropriate time - normally at the end of the interrupt handler.

See the Sections 'xQueueSendFromISR()' and 'xQueueReceiveFromISR()' which describe the xQueueSendFromISR() and xQueueReceiveFromISR() functions respectively for more information.

⚠ taskYIELD_FROM_ISR() must only be called from within an interrupt service routine that conforms to the requirements for such routines described in the port specific documentation.

⚠ Interrupt service routines that call taskYIELD_FROM_ISR() must not be permitted to execute prior to the scheduler being started.

4.2.6.5 Example

See the Listings 'Deferring interrupt processing to the task level', 'Example of using the xQueueSendFromISR() API function' and 'Example of using the xQueueReceiveFromISR() API function'.

4.2.7 taskENTER_CRITICAL()

```
task.h
Macro: taskENTER_CRITICAL()
```

4.2.7.1 Summary

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

Preemptive context switches can only occur from within an interrupt, so as long as interrupts remain disabled the task that called taskENTER_CRITICAL() is guaranteed to remain in the Running state until the critical section is exited.

It is safe for critical sections to become nested because the kernel keeps a count of the nesting depth. The critical section will only be exited when the nesting depth returns to zero - which is 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().

SafeRTOSRTOS API functions should not be called from within a critical section.

For more information on interrupts see the Interrupt section in the Product Variant User Manual [Reference 3].

4.2.7.2 Parameters

None.

4.2.7.3 Return Values

None.

4.2.7.4 Notes

Calls to taskENTER_CRITICAL() can be nested. The same number of calls must be made to taskEXIT_CRITICAL() as have previously been made to taskENTER_CRITICAL() before the critical region is exited and interrupts are enabled.

The longer a critical region takes to execute the less responsive the application will be to interrupts. Therefore all calls to taskENTER_CRITICAL() should be closely followed by a matching call to taskEXIT_CRITICAL().

Each call to taskENTER_CRTICAL() must have a corresponding call to taskEXIT_CRITICAL().

⚠ taskENTER_CRITICAL() must not be called from an interrupt service routine.

⚠ Critical sections implemented using the taskENTER_CRITICAL() and taskEXIT_CRITICAL() macros must be kept short in order that the system responsiveness to interrupts is maintained. The actual acceptable length is application dependent.

⚠ Calling taskENTER_CRITICAL() and taskEXIT_CRITICAL() should be the only method used to disable and enable interrupts respectively.

⚠ Consult the documentation specific to the port being used for further information on interrupt handling.

⚠ Calling API functions from within a critical section implemented using the taskENTER_CRITICAL() macro will not prevent the API function causing a context switch, and as each task maintains its own interrupt status the context switch could be to a task that has interrupts enabled. Refer to the Section 'vTaskSuspendScheduler()' for an alternative method of implementing critical regions.

⚠ The count of nested calls to taskENTER_CRITICAL() will eventually overflow - with the maximum value that can be held in the type defined as portBASE_TYPE being the maximum nesting count that can be maintained.

4.2.7.5 Example

```
/* A function that also uses a critical region. */
void vDemoFunction( void )
{
       /* This function uses taskENTER_CRITICAL() to implement a critical region.
        * It is itself called from within a critical region within vTask1, so this
        * call creates a nesting depth of 2. */
       taskENTER_CRITICAL();

       /* Perform an action here. */

       /* As calls to taskENTER_CRITICAL() are nested this call does not result in
        * interrupts being enabled. */
       taskEXIT_CRITICAL();
}

void vTask1( void * pvParameters )
{
       for( ;; )
       {
              /* Perform some actions here. */

              /* At some point the task wants to perform an operation within a critical
               * region so calls taskENTER_CRITICAL() to disable interrupts. */
              taskENTER_CRITICAL();

              /* Perform the operation here.  This part of the code must be kept
               * short as interrupts cannot execute. */

              /* Calls to taskENTER_CRITICAL() can be nested so it is safe to
               * call a function which also calls taskENTER_CRITICAL. */
              vDemoFunction();

              /* The operation is complete.  Exit the critical region. */
              taskEXIT_CRITICAL();
       }
}
```

Listing 19 Example of using the taskENTER_CRITICAL() and taskEXIT_CRITICAL() macros

4.2.8 taskEXIT_CRITICAL()

```
task.h
Macro: taskEXIT_CRITICAL()
```

4.2.8.1 Summary

Critical sections are exited by calling taskEXIT_CRITICAL().

Preemptive context switches can only occur from within an interrupt, so as long as interrupts remain within a critical section the task is guaranteed to remain in the Running state until taskEXIT_CRITICAL() is called.

It is safe for critical sections to become nested because the kernel keeps a count of the nesting depth. The critical section will only be exited when the nesting depth returns to zero - which is 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().

SafeRTOSRTOS API functions should not be called from within a critical section.

For more information on interrupts see the Interrupt section in the Product Variant User Manual [Reference 3].

4.2.8.2 Parameters

None.

4.2.8.3 Return Values

None.

4.2.8.4 Notes

Calls to taskENTER_CRITICAL() can be nested. The same number of calls must be made to taskEXIT_CRITICAL() as have previously been made to taskENTER_CRITICAL() before the critical region is exited and interrupts are enabled.

The longer a critical region takes to execute the less responsive the application will be to interrupts. Therefore all calls to taskENTER_CRITICAL() should be closely followed by a matching call to taskEXIT_CRITICAL().

Each call to taskENTER_CRTICAL() must have a corresponding call to taskEXIT_CRITICAL().

⚠ taskEXIT_CRITICAL() must not be called from an interrupt service routine.

⚠ Critical sections implemented using the taskENTER_CRITICAL() and taskEXIT_CRITICAL() macros must be kept short in order that the system responsiveness to interrupts is maintained. The actual acceptable length is application dependent.

⚠ Calling taskENTER_CRITICAL() and taskEXIT_CRITICAL() should be the only method used to disable and enable interrupts respectively.

⚠ Consult the documentation specific to the port being used for further information on interrupt handling.

⚠ Calling API functions from within a critical section implemented using the taskENTER_CRITICAL() macro will not prevent the API function causing a context switch, and as each task maintains its own interrupt status the context switch could be to a task that has interrupts enabled. Refer to the Section 'vTaskSuspendScheduler()' for an alternative method of implementing critical regions.

⚠ Calling taskEXIT_CRITICAL() prior to the scheduler starting will not necessarily cause interrupts to be enabled.

4.2.8.5 Example

See the Listing 'Example of using the taskENTER_CRITICAL() and taskEXIT_CRITICAL() macros'.

4.2.9 taskSET_INTERRUPT_MASK_FROM_ISR()

```
task.h
Macro: taskSET_INTERRUPT_MASK_FROM_ISR()
```

4.2.9.1 Summary

Sets the interrupt mask to mask all interrupts at the Kernel System Call interrupt priority or lower. This prevents any interrupts that calls a system API function from interrupting the current interrupt handler on ports where interrupt nesting is enabled.

4.2.9.2 Parameters

None.

4.2.9.3 Return Values

The previous interrupt mask read from the priority interrupt mask register.

4.2.9.4 Notes

⚠ taskSET_INTERRUPT_MASK_FROM_ISR() is only usable from within an interrupt service routine.

⚠ taskSET_INTERRUPT_MASK_FROM_ISR() should not be called unless closely paired with a call to taskCLEAR_INTERRUPT_MASK_FROM_ISR(), with taskSET_INTERRUPT_MASK_FROM_ISR() being called first and the return value of taskSET_INTERRUPT_MASK_FROM_ISR() being used as the parameter in the call to taskCLEAR_INTERRUPT_MASK_FROM_ISR().

⚠ On ports where interrupt nesting is not used taskSET_INTERRUPT_MASK_FROM_ISR() has no effect.

4.2.9.5 Example

```
void vAnExampleISR( void )
{
unsigned portBASE_TYPE uxOriginalPriority;

       /* This interrupt handler wants to call API functions but must do so as if it were in a
        * critical section. At the end of the critical section, restore the original interrupt mask.*/
       uxOriginalPriority = taskSET_INTERRUPT_MASK_FROM_ISR();
       {
              /* Perform uninterrupted API calls. */
       }
       taskCLEAR_INTERRUPT_MASK_FROM_ISR( uxOriginalPriority );

       /* Can now be interrupted by higher priority interrupts which still call API functions. */
}
```

Listing 20 Example of using the taskSET_INTERRUPT_MASK_FROM_ISR() and taskCLEAR_INTERRUPT_MASK_FROM_ISR() API macros.

4.2.10 taskCLEAR_INTERRUPT_MASK_FROM_ISR()

```
task.h
Macro: taskCLEAR_INTERRUPT_MASK_FROM_ISR( uxOriginalPriority )
```

4.2.10.1 Summary

Sets the interrupt mask to the value passed in as a parameter. This function is intended to take the value that was previously returned from the taskSET_INTERRUPT_ MASK_FROM_ISR().

4.2.10.2 Parameters

uxOriginalPriority	The interrupt priority mask used to set the interrupt priority mask register.

4.2.10.3 Return Values

None.

4.2.10.4 Notes

⚠ taskCLEAR_INTERRUPT_MASK_FROM_ISR() is only usable from within an interrupt service routine.

⚠ On ports where interrupt nesting is not used taskCLEAR_INTERRUPT_MASK_FROM_ISR() has no effect.

⚠ taskCLEAR_INTERRUPT_MASK_FROM_ISR() should not be called unless closely paired with a call to taskSET_INTERRUPT_MASK_FROM_ISR(), with taskSET_INTERRUPT_MASK_FROM_ISR() being called first and the return value of taskSET_INTERRUPT_MASK_FROM_ISR() being used as the parameter in the call to taskCLEAR_INTERRUPT_MASK_FROM_ISR().

4.2.10.5 Example

See the Listing 'Example of using the taskSET_INTERRUPT_MASK_FROM_ISR() and taskCLEAR_INTERRUPT_MASK_FROM_ISR() API macros'.

4.3 QUEUE FUNCTIONS

4.3.1 xQueueCreate()

```
queue.h
portBASE_TYPE xQueueCreate( signed portCHAR *pcQueueMemory,
                            unsigned portBASE_TYPE uxBufferLength,
                            unsigned portBASE_TYPE uxQueueLength,
                            unsigned portBASE_TYPE uxItemSize,
                            xQueueHandle *pxQueue
                          );
```

4.3.1.1 Summary

Creates a queue.

4.3.1.2 Parameters

pcQueueMemory	Pointer to the start of the memory to be used to hold the queue.
uxBufferLength	The length of the memory pointed to by the pcQueueMemory parameter. This must be equal to: (uxQueueLength * uxItemSize) + portQUEUE_OVERHEAD_BYTES where uxQueueLength and uxItemSize are the values passed into the respective parameters of the xQueueCreate() function and portQUEUE_OVERHEAD_BYTES is a constant available through the inclusion of SafeRTOS.h.
uxQueueLength	The maximum number of items the queue can hold at any time.
uxItemSize	The size in bytes of each item the queue will hold.
pxQueue	Used to pass back a handle by which the created queue can be referenced, for example when sending data to or reading data from the queue.

4.3.1.3 Return Values

pdPASS	The queue was created successfully.
errINVALID_BYTE_ALIGNMENT	The alignment of the pcQueueMemory value was not correct for the target hardware.
errINVALID_QUEUE_LENGTH	uxQueueLength was found to equal zero.
errINVALID_BUFFER_SIZE	uxBufferLengthBytes was found to not equal (uxQueueLength * uxItemSize) + portQUEUE_OVERHEAD_BYTES

errNULL_PARAMETER_SUPPLIED   Either pcQueueMemory or pxQueue was NULL.

4.3.1.4 Notes

Queues can be created prior to the scheduler being started and from within a task after the scheduler has been started.

4.3.1.5 Example

```
/* Define the data type that will be queued. */
typedef struct A_Message
{
       portCHAR ucMessageID;
       portCHAR ucData[ 20 ];
} AMessage;

/* Define the queue parameters. */
#define QUEUE_LENGTH 5
#define QUEUE_ITEM_SIZE sizeof( AMessage )

/* Define the buffer to be used by the queue. */
#define REQUIRED_BUFFER_SIZE    ( ( QUEUE_LENGTH * QUEUE_ITEM_SIZE ) + portQUEUE_OVERHEAD_BYTES )
portCHAR cQueueBuffer[ REQUIRED_BUFFER_SIZE ];

int main( void )
{
xQueueHandle xQueue;

       if( xQueueCreate(
              cQueueBuffer,
              REQUIRED_BUFFER_LENGTH,
              QUEUE_LENGTH,
              QUEUE_ITEM_SIZE,
              &xHandle
                     ) != pdPASS )
       {
              /* The queue could not be created.  The return value could have been checked to find out why. */
       }

       return 1;
}
```

Listing 21 Example of using the xQueueCreate() API function

4.3.2 xQueueSend()

```
queue.h
portBASE_TYPE xQueueSend( xQueueHandle pxQueue, const void * const pvItemToQueue, portTickType xTicksToWait );
```

4.3.2.1 Summary

Sends an item to a queue.

4.3.2.2 Parameters

pxQueue	The handle of the queue to which the data is to be sent. The handle of a queue is obtained from the pxQueue parameter of the call to xQueueCreate() that created the queue.
pvItemToQueue	A pointer to the data to be sent to the queue.
xTicksToWait	The number of ticks for which the calling task should be held in the Blocked state to wait for space to become available on the queue should the queue already be full. A value of zero will prevent the calling task from entering the Blocked state.

4.3.2.3 Return Values

pdPASS	Data was successfully sent to the queue. The calling task may have been temporarily blocked to wait for space to become available on the queue.
errSCHEDULER_IS_SUSPENDED	The scheduler was in the Suspended state when xQueueSend() was called. As xQueueSend() can potentially cause the calling task to enter the Blocked state it cannot be called when the scheduler is suspended.
errINVALID_QUEUE_HANDLE	The pxQueue parameter was either NULL or did not reference a valid queue.
errNULL_PARAMETER_SUPPLIED	pvItemToQueue was found to be NULL. pvItemToQueue is only permitted to be NULL when the queue item size (set when the queue was created) is zero.
errQUEUE_FULL	The queue is already full and the send cannot therefore complete. The calling task may have been temporarily blocked to wait for space to become available.

4.3.2.4 Notes

⚠ xQueueSend() 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).

⚠ xQueueSend() can potentially be a lengthy operation (partly dependent on the size of the data being sent to the queue). It is therefore recommended that xQueueSend() is not called from within a critical region.

⚠ If xQueueSend() were called from within a critical section then the critical section would not prevent the calling task from blocking. Each task maintains its own interrupt status and therefore the calling task blocking could cause a switch to a task that has interrupts enabled.

4.3.2.5 Example

This example sends an item to the queue created in the Listing 'Example of using the xQueueCreate() API function'. It assumes the queue handle is passed into the task using the tasks parameter.

```
void vATask( void *pvParameters )
{
xQueueHandle xQueue;
AMessage xMessage;

	/* The queue handle is passed into this task as the task parameter. */
	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 become available
		 * should the queue already be full. */
		if( xQueueSend( xQueue, &xMessage, 10 ) != pdPASS )
		{
			/* We could not send to the queue.  The return value could have been checked to find out why.
*/
		}
	}
}
```

Listing 22 Example of using the xQueueSend() API function

4.3.3 xQueueReceive()

```
queue.h
portBASE_TYPE xQueueReceive( xQueueHandle pxQueue, void *const pvBuffer, portTickType xTicksToWait );
```

4.3.3.1 Summary

Retrieves an item from a queue.

4.3.3.2 Parameters

pxQueue	The handle of the queue from which the data is to be received. The handle of a queue is obtained from the pxQueue parameter of the call to xQueueCreate() that created the queue.
pvBuffer	A pointer to the memory into which the data received from the queue should be copied. ⚠ The length of the buffer must be at least equal to the queue item size (set when the queue was created).

xTicksToWait	The number of ticks for which the calling task should be held in the Blocked state to wait for data to become available from the queue should the queue already be empty. A value of zero will prevent the calling task from entering the Blocked state.

4.3.3.3 Return Values

pdPASS	Data was successfully received from the queue. The calling task may have been temporarily blocked to wait for data to become available.
errSCHEDULER_IS_SUSPENDED	The scheduler was in the Suspended state when xQueueReceive() was called. As xQueueReceive() can potentially cause the calling task to enter the Blocked state it cannot be called when the scheduler is suspended.
errINVALID_QUEUE_HANDLE	The pxQueue parameter was either NULL or did not reference a valid queue.
errNULL_PARAMETER_SUPPLIED	pvBuffer was found to be NULL. pvBuffer is only permitted to be NULL when the queue item size (set when the queue was created) is zero.
errQUEUE_EMPTY	The queue is already empty so the receive cannot complete. The calling task may have been temporarily blocked to wait for data to become available on the queue.

4.3.3.4 Notes

⚠ xQueueReceive() 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).

⚠ xQueueReceive() can potentially be a lengthy operation (partly dependent on the size of the data being retrieved from the queue). It is therefore recommended that xQueueReceive() is not called from within a critical region.

⚠ If xQueueReceive() were called from within a critical section then the critical section would not prevent the calling task from blocking. Each task maintains its own interrupt status and therefore the calling task blocking could cause a switch to a task that has interrupts enabled.

4.3.3.5 Example

This example receives an item from the queue created in the Listing 'Example of using the xQueueCreate() API function'. It assumes the queue handle is passed into the task using the tasks parameter.

```
void vAnotherTask( void *pvParameters )
{
xQueueHandle xQueue;
AMessage xMessage;

       /* The queue handle is passed into this task as the task parameter. */
       xQueue = ( xQueueHandle ) pvParameters;

       for( ;; )
       {
              /* Wait for the maximum period for data to become available on the queue. */
              if( xQueueReceive( xQueue, &xMessage, portMAX_DELAY) != pdPASS )
              {
                     /* We could not receive from the queue.  The return value could have been
                      * checked to find out why. */
              }
              else
              {
                     /* xMessage now contains the received data. */
              }
       }
}
```

Listing 23 Example of using the xQueueReceive() API function

4.3.4 xQueueMessagesWaiting()

```
queue.h
portBASE_TYPE xQueueMessagesWaiting( const xQueueHandle pxQueue, unsigned portBASE_TYPE *puxMessagesWaiting );
```

4.3.4.1 Summary

Queries the number of items that are currently within a queue.

4.3.4.2 Parameters

pxQueue	The handle of the queue being queried. The handle of a queue is obtained from the pxQueue parameter of the call to xQueueCreate() that created the queue.
puxMessagesWaiting	Address of the variable into which the number of items in the queue will be written.

4.3.4.3 Return Values

pdPASS	The number of items in the queue was successfully written to the variable at address puxMessagesWaiting.
errNULL_PARAMETER_SUPPLIED	Either pxQueue or puxMessagesWaiting was NULL.
errINVALID_QUEUE_HANDLE	pxQueue did not reference a valid queue.

4.3.4.4 Notes

⚠ xQueueMessagesWaiting() must not be called from within an interrupt service routine.

4.3.4.5 Example

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

       /* How many items are currently in the queue? */
       if( xQueueMessagesWaiting( xQueue, &uxNumberOfItems ) != pdPASS )
       {
              /* Could not query the queue.  The return value could have been checked to find out why. */
       }
       else
       {
              /* uxNumberOfItems is now set to the number of items currently within xQueue. */
       }
}
```

Listing 24 Example of using the xQueueMessagesWaiting() API function

4.3.5 xQueueSendFromISR()

```
queue.h
portBASE_TYPE xQueueSendFromISR( xQueueHandle pxQueue,
                                  const void *const pvItemToQueue,
                                  portBASE_TYPE *pxHigherPriorityTaskWoken
                                );
```

4.3.5.1 Summary

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

4.3.5.2 Parameters

pxQueue	The handle of the queue to which the data is to be sent. The handle of a queue is obtained from the pxQueue parameter of the call to xQueueCreate() that created the queue.
pvItemToQueue	A pointer to the data to be sent to the queue.
pxHigherPriorityTaskWoken	*pxHigherPriorityTaskWoken will be set to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the current Running state task, otherwise *pxHigherPriorityTaskWoken will remain unchanged. The value of *pxHigherPriorityTaskWoken can be used to determine whether or not a context switch should be performed prior to the interrupt exiting, as demonstrated in the Listing 'Example of using the xQueueSendFromISR() API function'.

4.3.5.3 Return Values

pdTRUE	Data was successfully written to the queue.
errINVALID_QUEUE_HANDLE	pxQueue was either NULL or did not reference a valid queue.
errNULL_PARAMETER_SUPPLIED	pvItemToQueue or pxHigherPriorityTaskWoken was found to be NULL. It is only valid for pvItemToQueue to be NULL if the queue item size (set when the queue was created) is zero.
errQUEUE_FULL	The queue is already full and the send cannot therefore complete.

4.3.5.4 Notes

Calling xQueueSendFromISR() within an interrupt service routine can potentially cause a task to leave the Blocked state - necessitating a context switch if the unblocked task has a priority higher than that of the interrupted task. The context switch will ensure that the interrupt returns directly to the highest priority Ready state task. However, unlike the xQueueSend() API function, xQueueSendFromISR() will not itself cause a context switch to occur.

A context switch is performed transparently (within the API function itself) when xQueueSend() causes a task of higher priority than the calling task to exit the Blocked state. While this behavior is desirable during the execution of a task it might be undesirable during the execution on an interrupt if the interrupt service routine had not yet completed its processing. Therefore xQueueSendFromISR(), rather than performing the context switch itself, instead returns a value in the pxHigherPriorityTaskWoken parameter to indicate whether a context switch is required. This is demonstrated in the Listing 'Example of using the xQueueSendFromISR() API function'.

⚠ xQueueSendFromISR() should only be called from within an interrupt service routine.

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

4.3.5.5 Example

```
void vAnExampleISR( void )
{
portCHAR cIn;
portBASE_TYPE xHigherPriorityTaskWoken;

       /* We have not yet woken a task. */
       xHigherPriorityTaskWoken = pdFALSE;

       /* By way of example, assume this interrupt empties a FIFO, sending
          each character it obtains onto a queue.  Sending each character individually
          in this manner would in reality be inefficient and should normally be avoided. */
       while( prvCharactersInFIFO() == pdTRUE )
       {
              cIn = prvGetNextCharacterFromFIFO();

              /* Send the character onto the queue.  xHigherPriorityTaskWoken will get
                 set to pdTRUE if the send operation causes a task to unblock, and the
                 unblocked task has a priority higher than the current Running state task.
                 It does not matter how many times this is called.  For simplicity the return
                 value is ignored.  It is assumed that the queue xQueue has already been
                 created and is expecting to receive single bytes. */
              xQueueSendFromISR( xQueue, &cIn, &xHigherPriorityTaskWoken );
       }
       /* Ensure the interrupt is cleared before leaving the function. */

       /* Now the buffer is empty and we have cleared the interrupt we pass
          xHigherPriorityTaskWoken to taskYIELD_FROM_ISR() - which will cause a context
          switch only if xHigherPriorityTaskWoken was set to pdTRUE by one of the calls to
          xQueueSendFromISR(). */
       taskYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Listing 25 Example of using the xQueueSendFromISR() API function

4.3.6 xQueueReceiveFromISR()

```
queue.h
portBASE_TYPE    xQueueReceiveFromISR(    xQueueHandle    pxQueue,    void    *const    pvBuffer,    portBASE_TYPE
*pxHigherPriorityTaskWoken);
```

4.3.6.1 Summary

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

4.3.6.2 Parameters

pxQueue	The handle of the queue from which data is to be received. The handle of a queue is obtained from the pxQueue parameter of the call to xQueueCreate() that created the queue.
pvBuffer	A pointer to the buffer into which the data received from the queue will be copied. ⚠ The length of the buffer must be at least equal to the queue item size (set when the queue was created).

pxHigherPriorityTaskWoken	*pxHigherPriorityTaskWoken will be set to pdTRUE if receiving from the queue caused a task to unblock, and the unblocked task has a priority higher than the current Running state task, otherwise *pxHigherPriorityTaskWoken will remain unchanged. The value of *pxHigherPriorityTaskWoken can be used to determine whether or not a context switch should be performed prior to the interrupt exiting, as demonstrated in the Listing 'Example of using the xQueueReceiveFromISR() API function'.

4.3.6.3 Return Values

pdPASS	Data was successfully received from the queue.
errNULL_PARAMETER_SUPPLIED	pxHigherPriorityTaskWoken or pvBuffer was found to be NULL. It is only valid for pvBuffer to be NULL if the queue item size (set when the queue was created) is zero.
errINVALID_QUEUE_HANDLE	pxQueue was either NULL or did not reference a valid queue.
errQUEUE_EMPTY	The queue is already empty so the receive cannot complete.

4.3.6.4 Notes

Calling xQueueReceiveFromISR() within an interrupt service routine can potentially cause a task to leave the Blocked state - necessitating a context switch if the unblocked task has a priority higher than that of the interrupted task. The context switch will ensure that the interrupt returns directly to the highest priority Ready state task. However, unlike the xQueueReceive() API function, xQueueReceiveFromISR() will not itself cause a context switch to occur.

A context switch is performed transparently (within the API function itself) when xQueueReceive() causes a task of higher priority than the calling task to exit the Blocked state. While this behavior is desirable during the execution of a task it might be undesirable during the execution on an interrupt if the interrupt service routine had not yet completed its processing. Therefore xQueueReceiveFromISR(), rather than performing the context switch itself, instead sets the variable pointed to by pxHigherPriorityTaskWoken to a value to indicate whether a context switch is required. This is demonstrated in the Listing 'Example of using the xQueueReceiveFromISR() API function'.

⚠ xQueueReceiveFromISR() should only be called from within an interrupt service routine.

⚠ 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.

4.3.6.5 Example

```
/* 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 )
{
portCHAR cByte;
portBASE_TYPE xHigherPriorityTaskWoken;

       /* No tasks have yet been woken. */
       xHigherPriorityTaskWoken = pdFALSE;

       /* Loop until the queue is empty. */
       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 pass
          xHigherPriorityTaskWoken to taskYIELD_FROM_ISR() - which will cause a context
          switch only if xHigherPriorityTaskWoken was set to pdTRUE by one of the calls to
          xQueueReceiveFromISR(). */
       taskYIELD_FROM_ISR( xYieldRequired );
}
```

Listing 26 Example of using the xQueueReceiveFromISR() API function

WITTENSTEIN

CONTACT INFORMATION

User feedback is essential to the continued maintenance and development of SafeRTOS. Please provide all software and documentation comments and suggestions to the most convenient contact point listed below.

Address:	WITTENSTEIN high integrity systems Brown's Court, Long Ashton Business Park Yanley Lane, Long Ashton Bristol, BS41 9LB England
Phone:	+44 (0)1275 395 600
Fax:	+44 (0)1275 393 630
Email:	support@HighIntegritySystems.com
Website	www.HighIntegritySystems.com

All Trademarks acknowledged.