SAFE**RTOS**™

User's Manual

TEXAS INSTRUMENTS

Copyright

Texas Instruments
108 Wild Basin, Suite 350
Austin, TX 78746
Main: +1-512-279-8800
Fax: +1-512-279-8879
http://www.luminarymicro.com

SAFERTOS™ is a robust, specialized Real-time Operating System which has been independently certified by the TÜV as having been developed in compliance with IEC61508 up to Safety Integrity Level (SIL) 3.

WITTENSTEIN high integrity systems is a trading name of WITTENSTEIN Aerospace and Simulation Ltd.

Table of Contents

List of Code Examples

List of Figures

List of Tables

P R E F A C E

About this Manual

Identification

This is the user's manual for SAFE**RTOS**™ - a low over head, mini, pre-emptive real time scheduler. SAFE**RTOS** is pre-programmed into the processor ROM, providing a unique way to develop high integrity applications quickly and safely.

Incorporating SAFE**RTOS** 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 task. Chapter 1, "System Overview", elaborates on the states in which a task can exist.

This SAFE**RTOS** User's Manual contains detailed reference information related to using SAFE**RTOS** from ROM.

SAFE**RTOS** is based on the FREE**RTOS**™ and OPEN**RTOS**™ code base and can be used either as a general purpose real-time operating system or in a mission critical environment.

Use in Safety-Related Systems

SAFE**RTOS** was developed using a formal and rigorous process. The process was certified by TÜV SÜD to confirm that it was in compliance with that mandated by IEC 61508 [Reference 3] parts 1 and 3 for Safety Integrity Level (SIL) 3 projects. The same processes have been used throughout the SAFE**RTOS** development.

Simply using SAFERTOS in an application does not mean developers can make a claim related to the conformance of SAFE**RTOS** to any requirements or process specification (including IEC 61508 [Reference 3]) without first following a recognized system wide conformance verification process. 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 SAFE**RTOS** to be used in any safety or otherwise commercially critical application.

In order to facilitate low risk certification, WITTENSTEIN have developed a Design Assurance Pack which contains full conformance evidence for SAFE**RTOS**. The Design Assurance Pack facilitates certification and speeds and de-risks the use of SAFE**RTOS** in industrial, medical and other similar critical applications.

In order to obtain the Design Assurance Packs for either IEC61508 (SIL3) or FDA510(k) certification, please contact your local WITTENSTEIN sales representative. Information can be found at http://www.HighIntegritySystems.com/ or by sending an email to info@highintegritysystems.com.

Document Overview

Scope

Engineers holding a position of responsibility within a safety or commercially critical development team must be adequately trained or have adequate prior experience to fulfill their responsibilities competently. It is therefore assumed that readers are already familiar with the concepts and development of multitasking embedded systems and these fundamental concepts are omitted from this manual. The eBook "Using the FreeRTOS Real Time Kernel – A Practical Guide" provides a more introductory text that can be referenced if required.

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

Contents

The SAFE**RTOS** *User's Manual* is organized into the following chapters:

- Chapter 1, "System Overview," provides an overview of SAFE**RTOS** and the description of the SAFE**RTOS** task, queue, semaphore and scheduling mechanisms.

- Chapter 2, "Installation," describes the installation and setup required to use SAFE**RTOS** in your application.

- Chapter 3, "API Reference," provides the SAFE**RTOS** API reference.

- Chapter 4, "Stellaris® ARM® Cortex™-M3 Processor Core Port-Specific Information," provides information on using Stellaris® ARM® Cortex™-M3 Processor Core product variants.

⚠ **ESSENTIAL COMPLIANCE INFORMATION**

SAFE**RTOS** users must not call functions within the SAFE**RTOS** code base that are not documented in Chapter 3, "API Reference."

C H A P T E R   1

System Overview

This chapter provides an overview of SAFE**RTOS**.

Summary of the SAFERTOS Scheduler

The SAFE**RTOS** pre-emptive real time scheduler has the following characteristics:

- Any number of tasks can be created – the availability of RAM being the only limiting factor.

- Each task is assigned a priority between zero and ten, zero being the lowest priority. Source code versions of SAFE**RTOS** (as opposed to ROMed versions) do not impose restrictions on the number of priorities available.

- Any number of tasks can share the same priority – allowing for maximum application design flexibility.

- The highest priority task that is able to execute (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 Scheduling Policy" on page 18).

- Tasks can block for a fixed period.

- Tasks can block to wait for an absolutespecified time.

- Tasks can block with a specified time-out period to wait for queue events (either data being written to or read from the queue).

- Queues can be used to send data between tasks, and to send data between tasks and interrupt service routines (ISR).

- Semaphores can be used to synchronize tasks with other tasks and to synchronize tasks with interrupt service routines.

- Semaphores can be used to ensure mutually exclusive access to shared resources.

Differences between SAFERTOS and OPENRTOS

While SAFE**RTOS** and OPEN**RTOS** share many attributes, the development process has necessitated some notable differences. In partular SAFE**RTOS** does not perform any dynamic memory allocation, and SAFE**RTOS** performs numerious parameter and internal data validity checks.

SAFE**RTOS** is a statically declared subset of OPEN**RTOS**. OPEN**RTOS** to SAFE**RTOS** conversion instructions are provided in a separate technical note.

Design Goals

The design goal of SAFE**RTOS** is to achieve its stated functionality using a small, simple, and (most importantly) robust implementation.

Coding Conventions

This section defines the coding conventions used for the SAFE**RTOS** API.

Project Definitions

Each C file that utilizes the SAFE**RTOS** API must include the SAFERTOS.h header. The SAFERTOS.h header file itslef includes the ProjDefs.h header file which contains the definitions shown in Table 1-1 and Table 1-2.

Table 1-1.   Project Definitions

Definition	Value
pdTRUE[a]	1
pdFALSE	0
pdPASS	1
pdFAIL	0

a. The 'pd' prefix denotes that the constant is defined within the ProjDefs.h header file. The ProjDefs.h header file also contains error code definitions that begin with the 'err' prefix.

Table 1-2.   Port-Dependent Definitions

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

a. Port-dependent values are described in Chapter 4, "Stellaris® ARM® Cortex™-M3 Processor Core Port-Specific Information" on page 73.

Naming Conventions

The following conventions are used throughout the code:

■ Parameter names are prefixed with their type as follows:

Table 1-3.   Naming Conventions

Parameter Name	Type	Prefix
Variables	portCHAR	c
	portSHORT	s
	portLONG	l
	portBASE_TYPE	x
	structures, and so on	x
	void	v[a]
Pointers	—	p[b]
Unsigned variables	—	u[c]

a. For example, pointers to void and void functions.
b. For example, a pointer to a short will have the prefix ps, a
    pointer to void will have the prefix pv, and so on.
c. For example, an unsigned short will have the prefix us.

■ Historically function names were also prefixed with their return type using the same convention. The additional validity checking performed by SAFE**RTOS** has resulted in nearly all API functions returning a value, and for reasons of portability this value is always of type portBASE_TYPE (prefix 'x'). It is simpler therefore to consider any function that is prefixed 'x' as returning either a status code or a value, and any function that is prefixed 'v' (void) as returning no value.

■ API functions are also prefixed with the feature to which they relate, either Task or Queue. For example, the prototype for the API function `xTaskGetTickCount()`, or `xQueueSend()`.

■ Macro names are written in all uppercase other than a lowercase prefix that indicates in which header file the macro is defined. The exception to this rule are the error codes which are prefixed with 'err' but contained in the ProjDefs.h header file.

System Components

Tasks

Including SAFERTOS in your application allows the application to be structured as a set of autonomous tasks. Each task executes within its own context with no coincidental dependency on other tasks within the system or the scheduler itself.

Task Functions

Functions that implement a task must be of pdTASK_CODE type, where pdTASK_CODE is defined as shown in Code Example 1-1 with an example of such a function shown in Code Example 1-2.

A task will typically execute indefinitely and as such be written as an infinite loop, also shown in Code Example 1-2.

Code Example 1-1 pdTASK_CODE definition

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

Code Example 1-2 Typical structure of a task

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

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

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

⚠ **ESSENTIAL COMPLIANCE INFORMATION**

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 shown in Code Example 1-3.

Code Example 1-3 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 );
    }
```

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 on page 32 for further information.
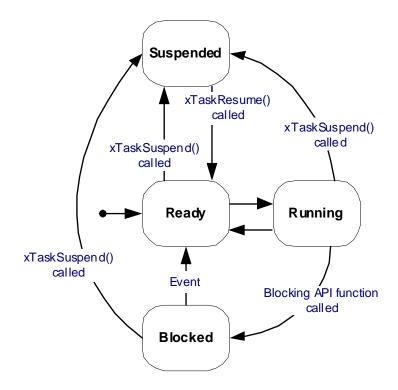
Task States

Only one task can be executing at a 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 Table 1-4, with valid transitions between states shown in Figure 1-1 on page 17.

Table 1-4.   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. The task cannot continue until the event occurs and until that time, it cannot be selected by the scheduler as the task to enter the Running state. Tasks in the Blocked state always have a time-out period, after which the task becomes unblocked.
Suspended	A task enters the Suspended state when it is the subject of a call to the `xTaskSuspend()` API function, and remains in the Suspended state until unsuspended by a call to the `xTaskResume()` API function. A time-out period cannot be specified. A Suspended state task 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 1-1.   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. "Intertask Communication" on page 21 provides more information on using queues.

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 10 (this restriction applies only when executing SafeRTOS out of ROM).

The Scheduler

The scheduler has responsibility for:

- Deciding which task to select to enter the Running state
- Performing the applicable context switching
- Measuring the passage of time
- Transitioning tasks from the Blocked state into the Ready state upon the expiration of a time-out period

Measuring Time

A periodic (tick) timer interrupt is used to measure time. The time between two consecutive timer interrupts is defined as one tick. All times are measured and specified in tick units.

The number of milliseconds between each tick is set using the ulTickRateHz member of the structure passed to the vTaskInitializeScheduler() API function.

The core SysTick timer is used to generate the tick interrupt.

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 executes for a maximum of one tick period before the scheduler selects another task of equal priority to enter the Running state.

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

Starting the Scheduler

The scheduler is started using the `xTaskStartScheduler()` API function. See Code Example 1-5 on page 23 for an example usage scenario.

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

Calling the `xTaskStartScheduler()` function 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 used to execute application-specific code within the Idle task.

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 will 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.

Scheduler States

The scheduler can exist in one of the states Table 1-5, with valid transitions between states shown in Figure 1-2 on page 20.

Table 1-5. 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 at any given time..
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.
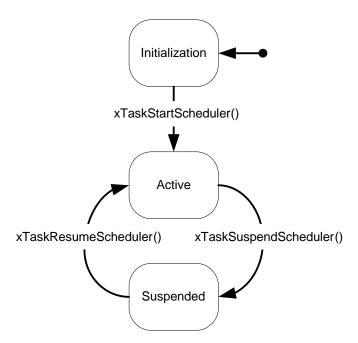
Figure 1-2.   Valid Scheduler State Transitions

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

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 and then re-enable interrupts as the critical region is entered and then exited respectively. The taskENTER_CRITICAL() and taskEXIT_CRITICAL() macros are provided for this purpose. Critical sections will only disable interrupts that have a priority up to and including interrupt priority 5 (a basepri value of 191). The execution of interrupts with a higher priority (those with priority 4 to 0) will not be effected by critical sections.

Implementing a critical section through the use of the `taskENTER_CRITICAL()` and `taskEXIT_CRITICAL()` macros has the disadvantage of the application being unresponsive to interrupts of priority 5 and below 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.

The xTaskSuspendScheduler() API function places the scheduler into the Suspended state. While in the Suspended state 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 the xTaskResumeScheduler() function is called.

⚠ ESSENTIAL COMPLIANCE INFORMATION

- 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 or semaphore while the scheduler is in the Suspended state.

- The xTaskSuspendScheduler() API function places the scheduler into the Suspended state. While in the Suspended state 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 the xTaskResumeScheduler() function is called. It is still desirable for the scheduler not to be held in the Suspended state for an extended period as doing so will reduce the responsiveness of high-priority tasks.

Intertask Communication

SAFE**RTOS** 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.

Queue Characteristics

The queue is implemented as follows:

- 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 received from a queue using the `xQueueReceive()` and `xQueueReceiveFromISR()` API functions.

- Queues are FIFO buffers – that is, the first item sent to a queue using the `xQueueSend()` (or `xQueueSendFromISR()`) function is the first item retrieved from the queue when using the `xQueueReceive()` (or `xQueueReceiveFromISR()`) function.

- 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.

Queue Events

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

When calling the `xQueueSend()` function, 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 the queue is already full. The task is blocking on a queue event and leaves the Blocked state automatically when another task or interrupt removes an item from the queue.

When calling the `xQueueReceive()` function, 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 the queue is already empty. Again, the task is blocking on a queue event and leaves 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.

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 can be used to hold both a data value and the identity of the task sending the data.

If the amount of data requiring transfer in each item is large, then it may be preferable to queue a pointer to the data rather than the data itself. This is more efficient as only the pointer value needs to be copied rather than each byte of the data itself.

⚠ ESSENTIAL COMPLIANCE INFORMATION

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.

Using Queues as Binary Semaphores

Semaphores can be used for task to task synchronization, interrupt to task synchronization, and as a means for a task to signal that it wants to have exclusive access to data or other resources. In the latter case, 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 resourc, it must 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.

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 from 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 available 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 which results in the queue being empty. To give the semaphore, the task simply sends to the queue which results 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.

Code Example 1-4 provides an example semaphore function that creates, takes, and gives that uses the SAFE**RTOS** queue implementation. See Chapter 3, "API Reference" on page 29 for more information on the API functions used (`xQueueCreate()`, `xQueueReceive()`, and `xQueueSend()`). Macros are also provided to hide the underlying mechanism.

Code Example 1-4 Using queues to implement binary semaphores

```
portBASE_TYPE xSemaphoreCreateBinary( xSemaphoreHandle *xCreatedHandle )
{
    /* The first two parameters define the memory buffer to be used to hold
    the created semaphore.  1 is the length of the queue being created (1 as
    this is a binary semaphore), 0 is the queue item size.

    pdPASS will be returned if the semaphore is created successfully. */

    return xQueueCreate( pcBuffer, uxBufferLengthBytes, 1, 0, xCreatedHandle );
}

portBASE_TYPE xSemaphoreTake( xSemaphoreHandle xSemaphore, portTickTYpe xBlockTime )
{
    /* The queue item size is zero so we do not need to specify the buffer
    into which the received data will be placed, therefore NULL is passed.

    pdPASS will be returned if the semaphore is successfully 'taken'. */

    return xQueueReceive( xSemaphore, NULL, xBlockTime );
}

portBASE_TYPE xSemaphoreGive( xSemaphoreHandle xSemaphore )
{
    /* The queue item size is zero so we do not need to specify the buffer
    from which the sent data will be retrieved, therefore NULL is passed. */

    return xQueueSend( xSemaphore, NULL, 0 );
}
```

Counting semaphores can be implemented in a similar way.

Where semaphores are used to control access to a resource, consideration must be given to whether including a gatekeeper task would provide a neater application solution. A gatekeeper task is a task that has exclusive access to the kept resource. For example, consider an application where more than one task wants 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 in the Blocked state on a queue, but is awakened 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 shown in Code Example 1-5.

Code Example 1-5 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 * ), &xPrint-
QUeue );

    /* 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. */
        }
    }
```

Communication between Tasks and Interrupts

⚠️ **ESSENTIAL COMPLIANCE INFORMATION**

Interrupt handlers must not under any circumstances call an API function that could cause a task to block. For this reason the `xQueueSend()` and `xQueueReceive()` functions must not be called from within an ISR, instead, use the `xQueueSendFromISR()` and `xQueueReceiveFromISR()` functions.

The `xQueueSendFromISR()` and `xQueueReceiveFromISR()` (interrupt-safe versions of the `xQueueSend()` and `xQueueReceive()`) functions are often used to unblock a task upon the occurrence on an interrupt (see "Interrupts" on page 25 regarding interrupt management). However, for better efficiency, do not 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 object that can be queued. 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.

Interrupts

In the interest of stack usage predictability and to facilitate system behavioral analysis, interrupt handlers should only collect event data and clear the interrupt source – and therefore exit 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 shown in Code Example 1-6.

Code Example 1-6 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 );
```

```
            }
        }
```

This scheme has the added advantage of flexible event processing prioritization as any task priority can be used. 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.

⚠️ **ESSENTIAL COMPLIANCE INFORMATION**

- Do not allow interrupt service routines that call API functions to execute prior to the scheduler being started. The easiest method to ensure this is for interrupts to remain disabled until after the scheduler is started. Interrupts are automatically enabled when the first task starts executing.
- Do not call API functions from interrupts that have a priority greater than 5 (interrupts with priority 4 to 0).
- Calling an 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.

CHAPTER 2

Installation

This chapter describes how to integrate a host application (the application that uses SAFE**RTOS**) with the SAFE**RTOS** ROM code.

Source Code and Libraries

SAFE**RTOS** is pre-programmed in to the processor ROM. The SAFE**RTOS** API is made available to a host application by including the SAFERTOS.h header files from within the host application source files.

Hook Functions

The host application is required to provide three hook (or callback) functions.

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. Figure 2-1 shows the prototype for the `vApplicationErrorHook()` function.

Code Example 2-1 vApplicationErrorHook() Function Prototype

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

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

⚠ **ESSENTIAL COMPLIANCE INFORMATION**
- `vApplicationErrorHook()` must not return.
- `vApplicationErrorHook()` is called with interrupts disabled.

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

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. Figure 2-1 shows the prototype for the
`vApplicationTaskDeleteHook()` function.

Code Example 2-2vApplicationTaskDeleteHook() function prototype

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

vApplicationTaskDeleteHook() Parameters

xDeletedTask   The handle of the task that was deleted.

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 shown in Listing 5.

Code Example 2-3vApplicationIdleHook() function prototype

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

⚠️ **ESSENTIAL COMPLIANCE INFORMATION**

- Code contained within `vApplicationIdleHook()` must never call an API function that could
  result in the idle task entering the blocked state.
- If the `vApplicationIdleHook()` function is used to place the processor into a low-power
  mode, then the mode chosen must not prevent tick interrupts from being serviced.

Configuration

Configuration is performed at run time by calling the `vTaskInitializeScheduler()` API
function.

⚠️ **ESSENTIAL COMPLIANCE INFORMATION**

`vTaskInitializeScheduler()` must be the first SAFE**RTOS** API function to be called, and
must only be called once.

API Reference

This chapter provides the SAFE**RTOS** API reference and is divided into the following sections:

- Task Functions on page 29

- Scheduler Control Functions on page 49

- Queue Functions on page 61

All API functions reside in ROM and are made available to the host application through the inclusion of the SAFE**RTOS**.h header file within the host application C source files. Additional functionality is provided by macros that are contained within the semphr.h header file.

Task Functions

The following task functions are provided in the SAFE**RTOS** API:

- vTaskInitializeScheduler() on page 30

- xTaskCreate() on page 32

- xTaskDelete() on page 35

- xTaskDelay() on page 37

- xTaskDelayUntil() on page 39

- xTaskPriorityGet() on page 41

- xTaskPrioritySet() on page 43

- xTaskSuspend() on page 45

- xTaskResume() on page 47

vTaskInitializeScheduler()

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

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.

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 remains fewer than ulAdditionalStackCheckMarginBytes free bytes on the task stack, the application error hook is called. Therefore, the higher the ulAdditionalStackCheckMarginBytes value, the more sensitive the stack overflow checking becomes—zero is a valid value and results in the least sensitive stack overflow checking.

When a potential stack overflow is detected, the error hook is called without having actually saved the task context.

pxPortInitParameters   Pointer to a structure that contains initialization data. See section TBD in chapter TBD for details of the xPORT_INIT_PARAMTERS structure.

Return Values

None.

Notes

⚠ **ESSENTIAL COMPLIANCE INFORMATION**

`vTaskInitializeScheduler()` must be the first SAFE**RTOS** API function to be called, and must only be called once.

Example

Code Example 3-1   Using the vTaskInitializeScheduler() API function

```
/* 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. */
const xPORT_INIT_PARAMETERS xPortInit =
    {
        50000000UL,                               /* ulCPUClockHz */
        1000UL,                                   /* ulTickRateHz */
        prvTaskDeleteHook,                        /* pxTaskDeleteHook */
        prvErrorHook,                             /* pxErrorHook */
        prvIdleHook,                              /* pxIdleHook */
        ( void * ) *( ( unsigned portLONG * ) 0 ),  /* pulSystemStackLocation */
        200,                                      /* ulSystemStackSizeBytes */
        ( unsigned portLONG * ) 0,                /* pulVectorTableBase */
    };
        /* Setup the hardware. */
        prvSetupHardware();

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


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

        ....

    }
```

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
                         );
```

Summary

Creates a new task and places it into the Ready state.

Parameters

pvTaskCode	Pointer to the function that implements the task.
pcName	A descriptive name for the task. This is mainly used to facilitate debugging. Maximum length is defined by the configMAX_TASK_NAME_LEN parameter.
pcStackBuffer	Pointer to the start of the memory to be used as the task stack. The stack should be aligned on an 8 byte boundary.
ulStackDepthBytes	The size in bytes of the memory pointed to by the pcStackBuffer pointer. The minimum allowable size for the stack buffer is 136 bytes.
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.

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 stated minimum.
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.

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 the `xTaskCreate()` function returning. This occurs if the task being created has a priority higher than the task calling the `xTaskCreate()` function.

⚠ **ESSENTIAL COMPLIANCE INFORMATION**

- Calling `xTaskCreate()` while interrupts are disabled does not prevent the task being created from entering the Running state if it has a higher priority than the task calling `xTaskCreate()`. The task being created commences execution with interrupts enabled. Interrupts are again disabled when the task calling `xTaskCreate()` once again enters the Running state.
- Calling `xTaskCreate()` while the scheduler is in the Suspended state defers any necessary context switch until such time that the scheduler re-enters the Active state.
- `xTaskCreate()` must not be called from an interrupt service routine.

Example

Code Example 3-2   Using the xTaskCreate() API function

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

xTaskDelete()

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

Summary

Deletes the task referenced by the pxTaskToDelete parameter.

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 can delete itself by passing NULL as the pxTaskToDelete parameter.

Return Values

pdPASS   The task was successfully deleted.

errINVALID_TASK_HANDLE

The pxTaskToDelete parameter was not found to reference a valid task.

Notes

Deleting a task causes the task delete hook function to be called (see "vApplicationTaskDeleteHook()" on page 27). 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 is invalidated and cannot therefore, be used in further API function calls. Attempting to do so results in the API function returning an error.

⚠ ESSENTIAL COMPLIANCE INFORMATION

- `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. 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 are identical.

Example

Code Example 3-3   Using the xTaskDelete() API function

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

```c
    /* 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 );
}
```

xTaskDelay()

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

Summary

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

Parameters

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

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.

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 counts as one of the tick periods for which the task is held in the Blocked state.

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

⚠ **ESSENTIAL COMPLIANCE INFORMATION**

- `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 does 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-enters the Running state.

Example

Code Example 3-4   Using the xTaskDelay() API function

```
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. */
    }
}
```

xTaskDelayUntil()

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

Summary

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

Differences between xTaskDelay() and xTaskDelayUntil()

xTaskDelay() causes the calling task to enter the Blocked state for the specified number of ticks from the time xTaskDelay() is 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 wants 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 are 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.

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 is unblocked at time (*pxPreviousWakeTime + xTimeIncrement).

Return Values

pdPASS	The calling task was held in the Blocked state until the specified time.
errSCHEDULER_IS_SUSPENDED	The scheduler is in the Suspended state when xTaskDelayUntil() is 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 are 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.

Notes

⚠ **ESSENTIAL COMPLIANCE INFORMATION**

- `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 does 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 are once again be disabled when the task calling `xTaskDelayUntil()` re-enters the Running state.

Example

Code Example 3-5   Using the xTaskDelayUntil() API function

```
/* 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 ) == errDID_NOT_YIELD )
        {
            /* 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. */
    }
}
```

---

xTaskPriorityGet()

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

Summary

Queries the priority of a task.

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 sets the priority of the task being queried.

Return Values

pdPASS   *puxPriority is set to the priority of the task being queried.

errNULL_PARAMETER_SUPPLIED
                        puxPriority is NULL.

errINVALID_TASK_HANDLE
                        pxTask is not a valid task handle.

Notes

> ⚠ **ESSENTIAL COMPLIANCE INFORMATION**
>
> `xTaskPriorityGet()` must not be called from within an interrupt service routine.

Example

Code Example 3-6   Using the xTaskPriorityGet() API function

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

xTaskPrioritySet()

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

Summary

Changes the priority of a task.

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.

Return Values

pdPASS   The priority of the task was changed.

errINVALID_TASK_HANDLE

pxTask is not a valid task handle.

errINVALID_PRIORITY   The value of uxNewPriority is greater than the highest available task priority (configMAX_PRIORITIES – 1).

Notes

ESSENTIAL COMPLIANCE INFORMATION

- `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).
- Do not use the `xTaskPrioritySet()` API function to modify the priority of the idle task. The idle task never enters the Blocked or Suspended state and completely starves lower priority tasks of execution time if its priority is not the lowest (or 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.

ESSENTIAL COMPLIANCE INFORMATION (CONTINUED)

- Calling `xTaskPrioritySet()` while the scheduler is in the Suspended state defers any necessary context switch until such time that the scheduler re-enters the Active state.

Example

Code Example 3-7   Using the xTaskPrioritySet() API function

```
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 );
        }
    }
```

---

xTaskSuspend()

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

Summary

Places a task into the Suspended state.

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 can suspend itself by passing NULL as the pxTaskToSuspend parameter.

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, is unable to select a new task to run if a task suspends itself.

errINVALID_TASK_HANDLE

pxTaskToSuspend is not a valid task handle.

errTASK_ALREADY_SUSPENDED

The task referenced by the pxTaskToSuspend handle was already in the Suspended state.

Notes

⚠ **ESSENTIAL COMPLIANCE INFORMATION**

- `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 are once again disabled when the task calling `xTaskSuspend()` next enters the Running state.

Example

Code Example 3-8   Using the xTaskSuspend() API function

```
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. */
    }
}
```

xTaskResume()

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

Summary

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

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.

Return Values

pdPASS   The task was successfully resumed (transitioned out of the Suspended state).

errNULL_PARAMETER_SUPPLIED
                        pxTaskToResume is NULL.

errINVALID_TASK_HANDLE
                        pxTaskToResume is not a valid task handle (and not NULL).

errTASK_WAS_NOT_SUSPENDED
                        The task referenced by the pxTaskToResume handle was not in the Suspended state.

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 checks whether its timeout period has expired in the meantime. If the timeout period has not expired, the task once again enters 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 exits the `xTaskDelay()` or `xTaskDelayUntil()` function as if the specified delay period had expired, even if this is not actually the case.

⚠ **ESSENTIAL COMPLIANCE INFORMATION**

- `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 are once again disabled when the task calling `xTaskResume()` next enters the Running state.
- Calling `xTaskResume()` while the scheduler is in the Suspended state defers any necessary context switch until such time that the scheduler re-enters the Active state.

Example

Code Example 3-9   Using the xTaskResume() API function

```
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. */
        }
    }
```

Scheduler Control Functions

The following Schedule Control functions are included in the SAFE**RTOS** API:

- xTaskStartScheduler() on page 50
- vTaskSuspendScheduler() on page 51
- xTaskResumeScheduler() on page 53
- xTaskGetTickCount() on page 54
- taskYIELD() on page 55
- taskYIELD_FROM_ISR() on page 56
- taskENTER_CRITICAL() on page 57
- taskEXIT_CRITICAL() on page 59

xTaskStartScheduler()

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

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.

Parameters

xUseKernelConfigurationCheck

A Boolean which indicates whether the kernel configuration parameters should be checked.

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.

The `xTaskStartScheduler()` API function does not return if the scheduler starts successfully.

Notes

⚠ **ESSENTIAL COMPLIANCE INFORMATION**

- `xTaskStartScheduler()` must not be called from within an interrupt service routine.
- See Chapter 4, "Stellaris® ARM® Cortex™-M3 Processor Core Port-Specific Information," for details of the architecture-specific requirements that must be fulfilled before calling `xTaskStartScheduler()` (for example, the processor mode from which the function can be called).

Example

See Code Example 1-5, "Using a gatekeeper task to control access to a resource" on page 23.

vTaskSuspendScheduler()

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

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.

Parameters

None.

Return Values

None.

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 were previously made to `vTaskSuspendScheduler()` before the scheduler leaves the Suspended state and re-enters the Active state.

⚠ **ESSENTIAL COMPLIANCE INFORMATION**

- `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 overflow if it reaches the 0xffffffff (the maximum unsigned 32bit value).

Example

Code Example 3-10 Using the vTaskSuspendScheduler() and xTaskResumeScheduler() API functions

```
/* 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();
        }
    }
}
```

xTaskResumeScheduler()

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

Summary

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

Parameters

None.

Return Values

pdTRUE   The scheduler was transitioned into the Active state. The transition caused a pending context switch to occur.

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

errSCHEDULER_WAS_NOT_SUSPENDED
                     The scheduler was not in the Suspended state.

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 were previously made to `vTaskSuspendScheduler()` before the scheduler will leave the Suspended state and re-enter the Active state.

⚠ **ESSENTIAL COMPLIANCE INFORMATION**

- `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 are once again disabled when the task calling `xTaskResumeScheduler()` next enters the Running state.

Example

See Code Example 3-10, "Using the vTaskSuspendScheduler() and xTaskResumeScheduler() API functions" on page 51

xTaskGetTickCount()

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

Summary

Returns the current tick value.

Parameters

None.

Return Values

`xTaskGetTickCount()` always returns the current tick count value.

Notes

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

⚠ **ESSENTIAL COMPLIANCE INFORMATION**

- `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 Table 1-4 on page 17 for information about portTickType) and the frequency of the tick interrupt.
- `xTaskGetTickCount()` will always return zero prior to a successful call to xTaskStartScheduler().

Example

Code Example 3-11 Using the xTaskGetTickCount() API function

```
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;
    }
```

taskYIELD()

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

Summary

Yielding is where a task volunteers to leave the Running state by re-entering the Ready state before using all of its time slice. For more information, see "Yielding" on page 19.

Parameters

None.

Return Values

None.

Notes

> ⚠ **ESSENTIAL COMPLIANCE INFORMATION**
>
> - `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.

Example

Code Example 3-12 Using the taskYIELD() API function

```
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. */
        }
    }
```

taskYIELD_FROM_ISR()

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

Summary

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

Parameters

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

Return Values

None.

Notes

Calling either `xQueueSendFromISR()` or `xQueueReceiveFromISR()` within an interrupt service routine can potentially cause a task to leave the Blocked state which then necessitates a context switch if the unblocked task has a higher priority 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 "xQueueSendFromISR()" on page 69 and "xQueueReceiveFromISR()" on page 71 which describe the `xQueueSendFromISR()` and `xQueueReceiveFromISR()` functions respectively for more information.

⚠️ **ESSENTIAL COMPLIANCE INFORMATION**

- `taskYIELD_FROM_ISR()` must only be called from within an interrupt service routine.
- Interrupt service routines that call `taskYIELD_FROM_ISR()` must not be permitted to execute prior to the scheduler being started.

Example

See Code Example 1-6, "Deferring interrupt processing to the task level" on page 25, Code Example 3-18, "Using the xQueueSendFromISR() API function" on page 70, and Code Example 3-19, "Using the xQueueReceiveFromISR() API function" on page 72.

taskENTER_CRITICAL()

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

Summary

Critical sections are entered by calling `taskENTER_CRITICAL()` and exited by calling `taskEXIT_CRITICAL()`. Entering a critical section disables interrupts that have been assigned a priority of 5 or below (that is interrupts with priorities 5, 6 and 7). Exiting a critical section will enable all interrupt priority levels (assuming the nesting count is zero).

Preemptive context switches can only occur from within an interrupt or priority 7, so as long as tasks remain disabled at priority 5 and below, 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 is only 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 short, otherwise, they will adversely affect interrupt response times. Every call to `taskENTER_CRITICAL()` must be closely paired with a call to `taskEXIT_CRITICAL()`.

SAFE**RTOS** API functions should not be called from within a critical section.

For more information on interrupts see "Interrupts" on page 75 in Chapter 4, "Stellaris® ARM® Cortex™-M3 Processor Core Port-Specific Information."

Parameters

None.

Return Values

None.

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()`.

⚠ **ESSENTIAL COMPLIANCE INFORMATION**

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

Example

Code Example 3-13 Using the taskENTER_CRITICAL() and taskEXIT_CRITICAL() macros

---

```
/* 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();
        }
    }
```

---

taskEXIT_CRITICAL()

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

Summary

Critical sections are exited by calling `taskEXIT_CRITICAL()`.

Preemptive context switches can only occur from within an interrupt of priority 7, so as long as tasks 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 is exited only 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()`.

SAFE**RTOS** API functions should not be called from within a critical section.

For more information on interrupts see "Interrupts" on page 75 in Chapter 4, "Stellaris® ARM® Cortex™-M3 Processor Core Port-Specific Information."

Parameters

None.

Return Values

None.

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 is 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()`.

⚠ ESSENTIAL COMPLIANCE INFORMATION

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

Example

See Code Example 3-13, "Using the taskENTER_CRITICAL() and taskEXIT_CRITICAL() macros" on page 58.

Queue Functions

The following Queue functions are available in the SAFE**RTOS** API:

- xQueueCreate() on page 62
- xQueueSend() on page 64
- xQueueReceive() on page 66
- xQueueMessagesWaiting() on page 68
- xQueueSendFromISR() on page 69
- xQueueReceiveFromISR() on page 71

xQueueCreate()

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

Summary

Creates a queue.

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 the SAFERTOS.h header file.
uxQueueLength	The maximum number of items the queue can hold at any time.
uxItemSize	The size in bytes of each item the queue can 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.

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.

Notes

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

Example

Code Example 3-14 Using the xQueueCreate() API function

```
/* 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;
    }
```

xQueueSend()

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

Summary

Sends an item to a queue.

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 is held in the Blocked state to wait for space to become available on the queue if the queue is already full. A value of zero prevents the calling task from entering the Blocked state.

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 therefore, the send cannot complete. The calling task may have been temporarily blocked to wait for space to become available.

Notes

⚠ **ESSENTIAL COMPLIANCE INFORMATION**

- `xQueueSend()` must only be called from an executing task. Do not call while the scheduler is in the Initialization state (prior to the scheduler being started).
- If `xQueueSend()` is 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.

Example

This example sends an item to the queue created in Code Example 3-14. It assumes the queue handle is passed into the task using the tasks parameter.

Code Example 3-15 Using the xQueueSend() API function

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

xQueueReceive()

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

Summary

Retrieves an item from a queue.

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.

> ⚠ ESSENTIAL COMPLIANCE INFORMATION
> The length of the buffer for the pvBuffer parameter 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 is held in the Blocked
                        state to wait for data to become available from the queue if the queue
                        is already empty. A value of zero prevents the calling task from
                        entering the Blocked state.

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.

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).
- 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.

Example

This example receives an item from the queue created in [TBD this should be a reference to example 3-14] Code Example 3-15. It assumes the queue handle is passed into the task using the tasks parameter.

Code Example 3-16 Using the xQueueReceive() API function

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

xQueueMessagesWaiting()

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

Summary

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

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.

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.

Notes

⚠ **ESSENTIAL COMPLIANCE INFORMATION**

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

Example

Code Example 3-17 Using the xQueueMessagesWaiting() API function

```
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. */
        }
    }
```

xQueueSendFromISR()

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

Summary

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

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 a context switch should be performed prior to the interrupt exiting, as shown in Code Example 3-18.

Return Values

pdPASS   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 therefore, the send cannot complete.

Notes

Calling `xQueueSendFromISR()` within an interrupt service routine can potentially cause a task to leave the Blocked state, which necessitates a context switch if the unblocked task has a higher priority than that of the interrupted task. The context switch ensures that the interrupt returns directly to the highest priority Ready state task. However, unlike the `xQueueSend()` API function, `xQueueSendFromISR()` does 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 shown in Code Example 3-18.

⚠ **ESSENTIAL COMPLIANCE INFORMATION**

- `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.

Example

Code Example 3-18 Using the xQueueSendFromISR() API function

```
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 );
    }
```

xQueueReceiveFromISR()

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

Summary

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

Parameters

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.

⚠ ESSENTIAL COMPLIANCE INFORMATION

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 a context switch should be performed prior to the interrupt exiting, as shown in Code Example 3-19.

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.

Notes

Calling `xQueueReceiveFromISR()` within an interrupt service routine can potentially cause a task to leave the Blocked state, which necessitates a context switch if the unblocked task has a priority higher than that of the interrupted task. The context switch ensures that the interrupt returns directly to the highest priority Ready state task. However, unlike the `xQueueReceive()` API function, `xQueueReceiveFromISR()` does 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 shown in Code Example 3-19.

⚠ **ESSENTIAL COMPLIANCE INFORMATION**

- `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.

Example

Code Example 3-19 Using the xQueueReceiveFromISR() API function

```
/* 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 );
   }
```

Stellaris® ARM® Cortex™-M3 Processor Core Port-Specific Information

This chapter describes the SAFE**RTOS** port-specific documentation for the Stellaris® ARM® Cortex™-M3 Processor Core.

Installation

C Startup Code

A startup code example is provided; use this as a reference to obtain the correct settings. Function outlines are provided for the NMI_ISR and FaultISR handlers which must be configured by the user.

For correct operation, the C startup code at a minimum must reserve adequate space on the system stack for main() and interrupts to execute, and must use the system stack when main() is called. While SAFE**RTOS** tasks make use of the Process stack (as opposed to the Main stack), the stack space required is statically allocated within the C files and does not need to be allocated in the C startup file.

Vector Table

- `vSafeRTOS_SVC_Handler_Address` must be installed as the SVCall handler.

- `vSafeRTOS_PendSV_Handler_Address` must be installed as the PendSV handler.

- `vSafeRTOS_SysTick_Handler_Address` must be installed as the System Tick Timer, SysTick, handler.

Definitions for the three interrupt handler addresses are contained in the SAFERTOS.h header file.

Execution Context

The Stellaris® Cortex™-M3 port makes use of the xPORT_INIT_PARAMETERS structure which is defined as shown in Code Example 4-1.

Code Example 4-1 Definition of the xPORT_INIT_PARAMETERS Structure

```
typedef struct PORT_INIT_PARAMETERS
{
    unsigned portLONG ulCPUClockHz;           /* The frequency at which the
                                                 MCU is running. */
    unsigned portLONG ulTickRateHz;           /* The frequency at which the
                                                 tick interrupt should execute. */
    portTASK_DELETE_HOOK pxTaskDeleteHook;    /* Pointer to the delete hook
                                                 function implementation. */
    portERROR_HOOK pxErrorHook;               /* Pointer to the error hook
                                                 function implementation. */
    portIDLE_HOOK pxIdleHook;                 /* Pointer to the idle hook
```

```
                                                     function implementation. */
    unsigned portLONG *pulSystemStackLocation;    /* The address that holds the
                                                     pointer to the start of the
                                                     system stack - normally 0. */
    unsigned portLONG ulSystemStackSizeBytes;     /* The size of the system stack
                                                     - set within the C start up code. */
    unsigned portLONG *pulVectorTableBase;        /* Pointer to the start of the
                                                     interrupt vector table. */
} xPORT_INIT_PARAMETERS;
```

It is required that a variable of type xPORT_INIT_PARAMETERS is passed into the
vTaskInitializeScheduler() function and that the structure's contents are initialized to
ensure the correct running of SAFE**RTOS**.

Table 4-1 shows the initialization values for the xPORT_INIT_PARAMETERS structure, used by
the Demo Project supplied with the GCC Stellaris® Cortex™-M3 Product Variant of SAFE**RTOS**.

Table 4-1.   Example xPORT_INIT_PARAMETERS Initialization Values

Field	Assigned Value
ulCPUClockHz	50000000UL
ulTickRateHz	1000UL
pxTaskDeleteHook	prvTaskDeleteHook (Function pointer to a user created function which will handle task deletions)
pxErrorHook	prvErrorHook (Function pointer to a user created function which will handle errors)
pxIdleHook	prvIdleHook (A Function pointer to a user created function which will handle the idle task)
pulSystemStackLocation	(void *) * ((unsigned portLONG *) 0). By default on a Cortex M3 address 0 holds the address of the system stack.
ulSystemStackSizeBytes	200
pulVectorTableBase	(unsigned portLONG *) 0. By default on a Cortex M3 address 0 holds the start of the vector table - the first value in which is actually the address of the start of the system stack.

Interrupts

This section describes interrupt usage for the GCC Stellaris® Cortex™-M3 Product Variant.

Interrupt Entry and Exit

An application-defined interrupt handler that wants to request a context switch need only call `taskYIELD_FROM_ISR()` as described in "taskYIELD_FROM_ISR()" on page 56 and shown in Code Example 4-2.

Code Example 4-2 The ISR

```
void vXYZ_ISR( void )
{
portBASE_TYPE xYieldRequired = pdFALSE;

/* Clear the interrupt. */

/* Perform ISR work here. */

/* A yield is required. */
xYieldRequired = pdTRUE;

/* Perform the yield. */
taskYIELD_FROM_ISR( xYieldRequired );
}
```

Interrupt Priorities and Nesting

The tick interrupt has a priority of 7. Interrupts that call interrupt safe API functions (those that end in "FromISR") can be safely assigned priorities of 7, 6 and 5. Interrupts that do not call API functions can execute at priorities higher than 5 and will never have their execution delayed by kernel activity (within the limits of the hardware itself).

Important: Be aware that the Cortex™-M3 core uses numerically low priority numbers to represent HIGH priority interrupts. If you wish to assign an interrupt a low priority, do NOT assign it a priority of 0 (or other low numeric value) as this can result in the interrupt actually having the highest priority; and potentially make your system crash if this priority is above portSYSCALL_INTERRUPT_PRIORITY.

See Chapter 3, "API Reference" on page 29, for details of which API functions can be safely called from within interrupt service routines.

Interrupt Vectors

See the "Vector Table" on page 73.

System Tick Timer (SysTick)

SAFE**RTOS** uses the ARM® Cortex™-M3 system tick timer (SysTick) and must have exclusive access to SysTick. In addition, SAFE**RTOS** does not perform any other processor configuration, such as clock frequencies and memory interfaces. These items are the responsibility of the host application. An example configuration is provided within the supplied demonstration application.

The timer is configured with a timer compare value during the initialization of the port layer that is calculated using the following equation:

```
( ulCPU_CLOCK_HZ / ulTICK_RATE_HZ ) - 1UL;
```

The range and accuracy of the tick interrupt is dependent on the ulCPUClockHz and ulTickRateHz fields of the xPORT_INIT_PARAMETERS structure which is passed in to the `vTaskInitializeScheduler()` API function.

RAM Usage

SAFE**RTOS** requires 0x20C bytes of RAM. RAM in the range 0x20000000 to 0x2000020C must be resurved for use by the kernel.