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

OPEN**RTOS** Quick Reference Guide

CHAPTER 1 INTRODUCTION

1.1 ABOUT THIS GUIDE

1.1.1 Introduction

This is the quick reference guide for OPEN**RTOS**™ a pre-emptive real time scheduler. OPEN**RTOS**™ is supplied as C source code.

Using OPEN**RTOS**™ in an embedded software application allows the 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 events and the state and relative priority of each task.

CHAPTER 2 SYSTEM DESCRIPTION

2.1 SUMMARY OF THE OPENRTOS™ SCHEDULER

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

- Any number of tasks can be created – system RAM constraints being the limiting factor.
- Each task is assigned a priority (any number of priorities can be used) – system RAM constraints being the limiting factor.
- Any number of tasks can share the same priority – allowing for maximum application design flexibility.
- The highest priority task that is able to execute (i.e. that is not blocked or suspended) will be the task selected by the scheduler to execute. Numerically high priority numbers denote high priorities. The lowest priority is priority zero.
- Tasks of equal priority will each get an equal share of the processing time available to tasks of that priority.
- Queues can be used to send data between tasks between Interrupt Service Routines (ISR).
- Binary semaphores, counting semaphores, recursive semaphores and mutexes make use of the queue primitive – ensuring code size is kept to a minimum.
- Tasks can be blocked for a fixed period.
- Tasks can be blocked to wait for a specified time.
- Tasks can be blocked with a specified timeout period to wait for queue (and semaphore) events (either data being written to or read from the queue).

2.2 SYSTEM TASKS

Using OPEN**RTOS**™ in your application allows the application to be structured as a set of autonomous tasks, the resultant system functionality being the sum of the functionality of the multiple tasks that make up the application. The scheduler rapidly switches execution from one task to another in accordance with events, the state and priority of each task.

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

2.2.1 Task States

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

A task can exist in one of the states described by Table 2-1, with valid transitions between states depicted by Figure 2-1.

Table 2-1 Task States

State	Description
Running	When a task is actually executing it is said to be in the Running state. It is the task selected by the scheduler to execute and is currently utilizing the processor. Only one task can be in the Running state at any given time.
Blocked	A task is in the Blocked state if it is waiting for an event. It cannot continue until the event occurs – and until that time cannot be selected by the scheduler as the task to enter the Running state. A task can be blocked on: 1. A queue/semaphore/mutex event. 2. A temporal event (a timeout). 3. Both a queue/semaphore/mutex event and a temporal event simultaneously. In this case the task will unblock (enter the Ready state) should either event occur. For example, a task can be blocked with a timeout to wait for data to become available on a queue.
Suspended	A task will enter the Suspended state following a call to the vTaskSuspend() API function, and remain in the Suspended state until resumed by another task calling to the vTaskResume() API function. A timeout period cannot be specified. Suspended state tasks cannot be selected by the scheduler as the task to enter the Running state and cannot therefore resume themselves.
Ready	A task that 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. Ready is the initial state when a task is created.

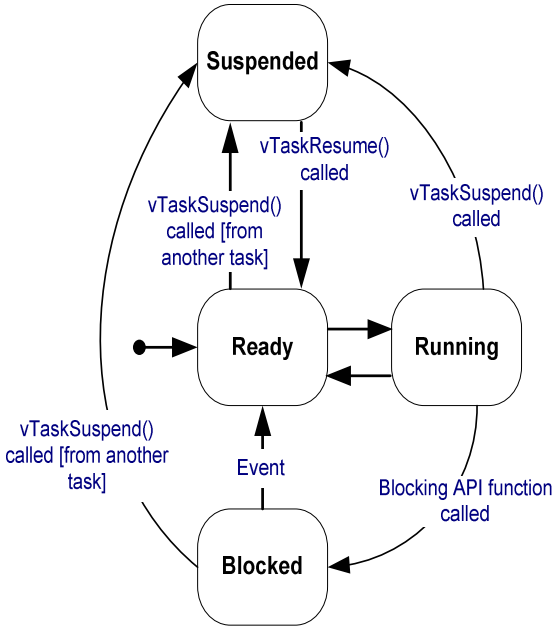

Figure 2-1 Valid task state transitions

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 vTaskSuspend() API function can cause a task in the Running state, Blocked state or Ready state to enter the Suspended state. In the first case the calling task would be suspending itself. In the latter two cases the calling task would be suspending a task other than itself.

Calls to the vTaskDelay() and vTaskDelayUntil() 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 (and their derivatives) 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.

2.2.2 Scheduler States

The scheduler can exist in one of the states described by Table 2-2, with valid transitions between states depicted by Figure 2-2.

Table 2-2 Scheduler States

State	Description
Initialization	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 to be in the Running State.
Suspended	The Scheduler does not perform a context switching while in the Suspended state. The task that was in the Running state when the scheduler entered the Suspended state shall remain in the Running state until the scheduler returns to the Active state.

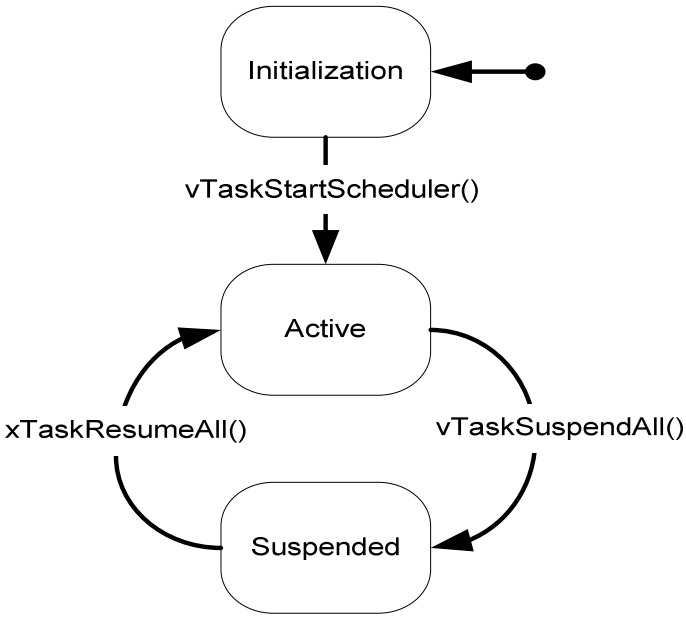

Figure 2-2 Valid Scheduler State Transitions

CHAPTER 3 API QUICK REFERENCE

This sections should be treated as an abridged version of the full documentation which is available as a part of the licensed version of OPEN**RTOS**™.

3.1 API LOOKUP

3.2 PORT DEPENDENT TYPES

The prototype descriptions within this chapter make use of a number of port variant types which are defined within Table 3-1.

Table 3-1 Port Dependent Definitions

Definition	Value
portBASE_TYPE	Port dependent – 32 bit type on 32 bit architectures, 16 bit type on 16 bit architectures.
portMAX_DELAY	0xFFFF if configUSE_16_BIT_TICKS is set to 1, or 0xFFFFFFFF if configUSE_16_BIT_TICKS is set to 0.
portTickType	unsigned 16 bit type if configUSE_16_BIT_TICKS is set to 1, or unsigned 32 bit type if configUSE_16_BIT_TICKS is set to 0.

3.3 TASK FUNCTIONS

3.3.1 xTaskCreate()

```
task.h
portBASE_TYPE xTaskCreate( pdTASK_CODE pvTaskCode,
                           const signed char * const pcName,
                           unsigned short usStackDepth,
                           void *pvParameters,
                           unsigned portBASE_TYPE uxPriority,
                           xTaskHandle *pxCreatedTask )
```

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

3.3.2 vTaskDelete()

```
task.h
void vTaskDelete( xTaskHandle pxTask )
```

Deletes the task referenced by the pxTask parameter.

3.3.3 vTaskDelay()

```
task.h
void vTaskDelay( portTickType xTicksToDelay )
```

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

3.3.4 vTaskDelayUntil()

```
task.h
void vTaskDelayUntil( portTickType *pxPreviousWakeTime, portTickType xTimeIncrement )
```

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

3.3.5 uxTaskPriorityGet()

```
task.h
unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle pxTask )
```

Queries the priority of a task.

3.3.6 vTaskPrioritySet()

```
task.h
void vTaskPrioritySet( xTaskHandle pxTask, unsigned portBASE_TYPE uxNewPriority )
```

Changes the priority of a task.

3.3.7 vTaskSuspend()

```
task.h
void vTaskSuspend( xTaskHandle pxTaskToSuspend )
```

Places a task into the Suspended state.

3.3.8 vTaskResume()

```
task.h
void vTaskResume( xTaskHandle pxTaskToResume )
```

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

3.3.9 xTaskResumeFromISR()

```
task.h
portBASE_TYPE xTaskResumeFromISR( xTaskHandle pxTaskToResume )
```

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

3.3.10 taskYIELD_FROM_ISR()

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

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

3.3.11 uxTaskGetHighWaterMark(xTaskHandle xTask)

```
task.h
unsigned portBASE_TYPE uxTaskGetStackHighWaterMark( xTaskHandle xTask )
```

The stack used by a task will grow and shrink as the task executes and interrupts are processed. uxTaskGetHighWaterMark() returns the minimum amount of remaining stack space that was available to the task since the task started executing – that is the amount of stack that remained unused when the task stack was at its greatest (deepest) value. This is what is referred to as the stack ‘high water mark’.

3.3.12 uxTaskGetNumberOfTasks()

```
task.h
unsigned portBASE_TYPE uxTaskGetNumberOfTasks( void )
```

Simply returns the number of tasks running at the time of calling.

3.4 SCHEDULER CONTROL FUNCTIONS

3.4.1 vTaskStartScheduler()

```
task.h
void vTaskStartScheduler( void )
```

Starts the scheduler by transitioning the scheduler from the Initialisation 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.

3.4.2 vTaskSuspendAll()

```
task.h
void vTaskSuspendAll( void )
```

Suspends all scheduler activity by transitioning the scheduler from the Active state into 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.

3.4.3 xTaskResumeAll()

```
task.h
portBASE_TYPE xTaskResumeAll( void )
```

Resumes scheduler activity by transitioning the scheduler out of the Suspended state into the Active state.

3.4.4 xTaskGetTickCount()

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

Returns the current tick value.

3.4.5 taskYIELD()

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

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

3.4.6 taskYIELD_FROM_ISR()

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

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

3.4.7 taskENTER_CRITICAL()

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

Creates a critical region by simply disabling interrupts. If configKERNEL_INTERRUPT_PRIORITY is defined for your port then interrupts will only be disabled up to that level. If configMAX_SYSCALL_INTERRUPT_PRIORITY is defined then interrupts will only be disabled up to that priority.

3.4.8 taskEXIT_CRITICAL()

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

Exits a critical region by enabling interrupts.

3.5 QUEUE FUNCTIONS

3.5.1 xQueueCreate()

```
queue.h
xQueueHandle xQueueCreate( unsigned portBASE_TYPE uxQueueLength,
                           unsigned portBASE_TYPE uxItemSize )
```

Creates a queue.

3.5.2 xQueueSend(), xQueueSendToFront(), xQueueSendToBack()

```
queue.h

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

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

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

Sends an item to the front or the back of a queue.

xQueueSend() and xQueueSendToBack() are analogous. xQueueSend() is deprecated and only included for backward compatibility.

WITTENSTEIN

3.5.3 xQueueReceive()

```
queue.h
portBASE_TYPE xQueueReceive( xQueueHandle xQueue, void *pvBuffer, portTickType xTicksToWait )
```

Retrieves an item from a queue.

3.5.4 xQueuePeek()

```
queue.h

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

Reads an item from a queue, but does not remove the item. Therefore the same item would be returned the next time xQueueReceive() or xQueuePeek() was called on the same queue.

3.5.5 uxQueueMessagesWaiting()

```
queue.h
unsigned portBASE_TYPE uxQueueMessagesWaiting( const xQueueHandle xQueue )
```

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

3.5.6 uxQueueMessagesWaitingFromISR()

```
queue.h
unsigned portBASE_TYPE uxQueueMessagesWaitingFromISR( const xQueueHandle xQueue )
```

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

3.5.7 xQueueSendFromISR(), xQueueSendToBackFromISR(), xQueueSendToFrontFromISR()

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

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

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

Versions of xQueueSend(), xQueueSendToFront() and xQueueSendToBack() API functions that can be called from an ISR. Unlike xQueueSend(), xQueueSendToFront() and xQueueSendToBack(), these functions do not permit a block time to be specified.

xQueueSendFromISR() and xQueueSendToBackFromISR() are analogous. xQueueSendFromISR() is deprecated and only included for backward compatibility.

WITTENSTEIN

3.5.8 xQueueReceiveFromISR()

```
queue.h
portBASE_TYPE xQueueReceiveFromISR( xQueueHandle xQueue,
                                    void *pvBuffer,
                                    portBASE_TYPE *pxHigherPriorityTaskWoken )
```

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

3.5.9 xQueueIsQueueFullFromISR()

```
queue.h
portBASE_TYPE xQueueIsQueueFullFromISR( const xQueueHandle xQueue )
```

A utility function that can be called from inside an interrupt service routine to determine whether or not a queue is full. xQueueIsQueueFullFromISR() can be used in combination with xQueueSendToFront/BackFromISR() to check the queue status prior to attempting to send any data.

3.5.10 xQueueIsQueueEmptyFromISR()

```
queue.h
portBASE_TYPE xQueueIsQueueEmptyFromISR( const xQueueHandle xQueue )
```

A utility function that can be called from inside an interrupt service routine to determine whether or not a queue is empty. xQueueIsQueueEmptyFromISR() can be used in combination with xQueueRecieveFromISR() to check the queue status prior to attempting to receive any data.

3.5.11 vQueueAddToRegistry()

```
queue.h
void vQueueAddToRegistry( xQueueHandle xQueue, signed char *pcQueueName )
```

Assigns a name to a queue and adds the queue to the registry.

The queue registry has two purposes, both of which are associated with kernel aware debugging:

1. It allows a textual name to be associated with a queue for easy queue identification within a debugging GUI.

2. It contains the information required by a debugger to locate each registered queue and semaphore.

The queue registry has no purpose unless you are using a kernel aware debugger.

configQUEUE_REGISTRY_SIZE defines the maximum number of queues and semaphores that can be registered. Only the queues and semaphores that you want to view using a kernel aware debugger need to be registered.

3.5.12 vQueueUnregisterQueue()

```
queue.h
void vQueueUnregisterQueue( xQueueHandle xQueue )
```

Removes a queue from the queue registry.

The queue registry has two purposes, both of which are associated with kernel aware debugging:

3. It allows a textual name to be associated with a queue for easy queue identification within a debugging GUI.

4. It contains the information required by a debugger to locate each registered queue and semaphore.

The queue registry has no purpose unless you are using a kernel aware debugger.

configQUEUE_REGISTRY_SIZE defines the maximum number of queues and semaphores that can be registered. Only the queues and semaphores that you want to view using a kernel aware debugger need to be registered.

3.6 SEMAPHORE FUNCTIONS AND MACROS

3.6.1 vSemaphoreCreateBinary()

```
semphr.h
vSemaphoreCreateBinary( xSemaphoreHandle xSemaphore )
```

Macro that creates a binary semaphore.

3.6.2 xSemaphoreCreateMutex()

```
semphr.h
xSemaphoreHandle xSemaphoreCreateMutex( void )
```

Macro that creates a mutex type semaphore.

3.6.3 xSemaphoreCreateCounting()

```
semphr.h
xSemaphoreHandle xSemaphoreCreateCounting( unsigned portBASE_TYPE uxMaxCount,
                                           unsigned portBASE_TYPE uxInitialCount )
```

Macro that creates a counting semaphore.

3.6.4 xSemaphoreCreateRecursiveMutex()

```
semphr.h
xSemaphoreHandle xSemaphoreCreateRecursiveMutex( void )
```

Macro that creates a recursive mutex type semaphore.

3.6.5 xSemaphoreTake()

```
semphr.h
xSemaphoreTake( xSemaphoreHandle xSemaphore, portTickType xBlockTime )
```

Macro to ‘take’ (or obtain) a semaphore that has previously been created using vSemaphoreCreateBinary(), xSemaphoreCreateCounting() or xSemaphoreCreateMutex().

3.6.6 xSemaphoreTakeRecursive()

WITTENSTEIN

```
semphr.h
portBASE_TYPE xSemaphoreTakeRecursive( xSemaphoreHandle xMutex, portTickType xBlockTime )
```

Macro to recursively ‘take’ (or obtain) a mutex type semaphore that has previously been created using the xSemaphoreCreateRecursiveMutex() API function.

3.6.7 xSemaphoreGive()

```
semphr.h
xSemaphoreGive( xSemaphoreHandle xSemaphore, portTickType xBlockTime )
```

Macro to ‘give’ (or release) a semaphore. The semaphore must have first been created using vSemaphoreCreateBinary(), and obtained using xSemaphoreTake().

3.6.8 xSemaphoreGiveRecursive()

```
semphr.h
portBASE_TYPE xSemaphoreGiveRecursive( xSemaphoreHandle xMutex )
```

Macro to recursively ‘give’ (or release) a mutex type semaphore that has previously been created using the xSemaphoreCreateRecursiveMutex() API function.

3.6.9 xSemaphoreGiveFromISR()

```
semphr.h
xSemaphoreGiveFromISR( xSemaphoreHandle xSemaphore, portBASE_TYPE xHigherPriorityTaskWoken )
```

Macro to ‘give’ (or release) a semaphore. The semaphore must have first been created using vSemaphoreCreateBinary() or xSemaphoreCreateCounting().

This macro can be used from an ISR.

CONTACT INFORMATION

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

WIITENSTEIN high integrity systems is a division of WITTENSTEIN aerospace & simulation limited