Carleton University
Department of Systems and Computer Engineering

SYSC5701: Operating System Methods for
Real-Time Applications

An Analysis and Description of the Inner Workings of the FreeRTOS Kernel

1 April 07

Rich Goyette

Abstract

This document is an analysis and functional decomposition of FreeRTOS version 4.1.3. FreeRTOS is a real-time, preemptive operating system targeting embedded devices. The FreeRTOS scheduling algorithm is dynamic and priority based. Interprocess communication is achieved via message queues and basic binary semaphores. Deadlocks are avoided by forcing all blocking processes to timeout with the result that application developers are required to set and tune timeouts and deal with resource allocation failures. Basic memory allocation schemes are provided but more complex schemes can be directly coded and incorporated. FreeRTOS provides some unique capabilities. Cooperative instead of preemptive scheduling can be used and the scheduler can be suspended by any task for any duration of time. No mechanisms to counter priority inversion are implemented. Overall, FreeRTOS was determined to be slightly too feature-rich for limited resource embedded devices. A simplified version may be beneficial to certain communities.

Table of contents

Introduction

Objectives

The primary objective of this document was to support and reinforce the understanding of RTOS concepts and mechanisms as they apply to embedded systems. To achieve this objective, an open source RTOS for embedded targets was selected, decomposed, and characterized in terms of traditional RTOS concepts and functionality.

Scope

This document is an analysis of FreeRTOS version 4.1.3. In certain cases, the analysis requires description in the context of a hardware target. In those cases, execution on a Freescale HC9S12C32 microcontroller unit (MCU) is assumed with compilation being performed by m6811-elf-gcc version 3.3.6.

FreeRTOS implements co-routines. These are not considered in this document as they are duplicative of the existing functionality.

Typographic Conventions

The following typographic conventions are used throughout this document:

- *Directory or folder name*;
- `Variable or configuration setting`;
- ***Name of function***;
- **Name of code file;**
- {ListName}

FreeRTOS Overview

General Features

A free, embedded RTOS has been made available by Richard Barry [FRTOS07]. This RTOS claims to be a portable, open source, mini real-time kernel that can be operated in pre-emptive or cooperative. Some of the main features of FreeRTOS are listed below:

a.) Real-time: FreeRTOS could, in fact, be a hard real-time operating system. The assignment of the label "hard real time" depends on the application in which FreeRTOS would function and on strong validation within that context.

b.) Preemptive or cooperative operation: The scheduler can be preemptive or cooperative (the mode is decided in a configuration switch). Cooperative scheduling does not implement a timer based scheduler decision point – processes pass control to one another by yielding. The scheduler interrupts at regular frequency simply to increment the tick count.

c.) Dynamic scheduling: Scheduler decision points occur at regular clock frequency. Asynchronous events (other than the scheduler) also invoke scheduler decisions points.

d.) Scheduling Algorithm: The scheduler algorithm is highest priority first. Where more than one task exists at the highest priority, tasks are executed in round robin fashion.

e.) Inter-Process Communication: Tasks within FreeRTOS can communicate with each other through the use of queuing and synchronization mechanisms:

    i.) Queuing: Inter-process communication is achieved via the creation of queues. Most information exchanged via queues is passed by value not by reference which should be a consideration for memory constrained applications. Queue reads or writes from within interrupt service routines (ISRs) are non-blocking. Queue reads or writes with zero timeout are non-blocking. All other queue reads or writes block with configurable timeouts.

    ii.) Synchronization: FreeRTOS allows the creation and use of *binary* semaphores. The semaphores themselves are specialized instances of message queues with queue length of one and data size of zero. Because of this, taking and giving semaphores are atomic operations since interrupts are disabled and the scheduler is suspended in order to obtain a lock on the queue.

f.) Blocking and Deadlock avoidance: In FreeRTOS, tasks are either non-blocking or will block with a fixed period of time. Tasks that wake up at the end of timeout and still cannot get access to a resource must have made provisions for the fact that the API call to the resource may return an access failure notification. Timeouts on each block reduce the likelihood of resource deadlocks.

g.) Critical Section Processing: Critical section processing is handled by the disabling of interrupts. Critical sections within a task can be nested and each task tracks its own nesting count. However, it is possible to yield from within a critical section (in support of the cooperative scheduling) because software interrupts (SWI) are non-maskable and yield uses SWI to switch context. The state of interrupts are restored on each task context switch by the restoration of the I bit in the condition code register (CCR).

h.) Scheduler Suspension: When exclusive access to the MCU is required without jeopardizing the operation of ISRs, the scheduler can be *suspended*. Suspending the scheduler guarantees that the current process will not be pre-empted by a scheduling event while at the same time continuing to service interrupts.

i.) Memory Allocation: FreeRTOS provides three heap models as part of the distribution. The simplest model provides for fixed memory allocation on the creation of each task but no de-allocation or memory reuse (therefore tasks cannot be deleted). A more complex heap model allows the allocation and de-allocation of memory and uses a best-fit algorithm to locate space in the heap. However, the algorithm does not combine adjacent free segments. The most complex heap algorithm provides wrappers for ***malloc()*** and ***calloc()***. A custom heap algorithm can be created to suit application requirements.

j.) Priority Inversion: FreeRTOS does not implement any advanced techniques (such as priority inheritance or priority ceilings [SYSC07]) to deal with priority inversion.

Source Code Distribution

FreeRTOS is distributed in a code tree as shown in Figure 1. FreeRTOS includes target independent source code in the `Source` directory. Most of the functionality of FreeRTOS is provided within the **tasks.c**, **queue.c**, **list.c**, and **coroutines.c** files (and associated header files).

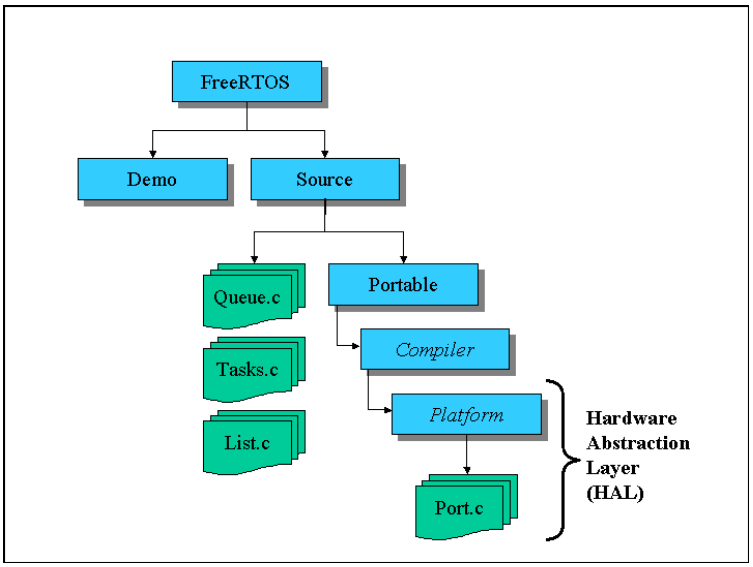


Figure 1: FreeRTOS Source Distribution

FreeRTOS provides a Hardware Abstract Layer (HAL) for various combinations of compiler and hardware target. Target-specific functionality for each compiler/hardware target is provided in the **port.c** and **portmacro.h** files within the HAL. All functions referenced in this document that are start with *port* belong to the HAL and are implemented in one of the portable files.

The `Demo` directory provides sample code for a several demonstration applications. This directory is organized in the same fashion as the `Portable` directory because the demonstrations are written and compiled to operate on certain hardware targets using various compilers.

FreeRTOS Kernel Description

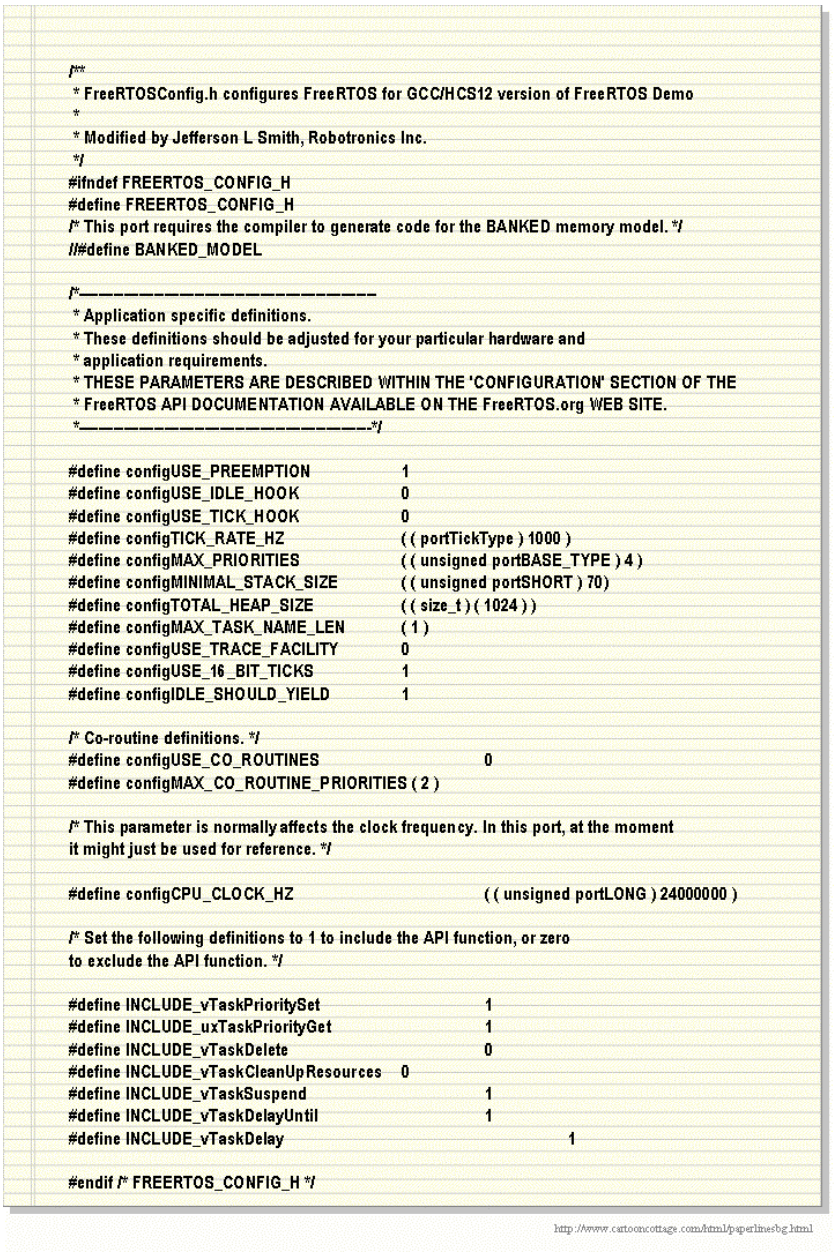
Introduction

This section provides a detailed description of the FreeRTOS kernel. Where necessary, kernel elaboration involving hardware dependent functions (i.e. elements of the HAL) will assume the target to be a Freescale HC9S12C32 microcontroller on a NanoCore12 DXC32 MCU Module [TA] with code compiled by GNU m6811-gnu-gcc version 3.3.6 [GCC0]. Annex A describes the effort to implement FreeRTOS on the HC9S12C32 with GCC. Although not in time for this document, the intent of that effort was to implement FreeRTOS on the chosen target in order to extract and report on certain RTOS related performance parameters (e.g., interrupt latency, context switch latency, etc). While those results are not yet available, significant experience was obtained with respect to the HAL and that experience is reflected in the analysis to follow.

The analysis and description begins with a review of significant pre-execution configuration items. This is followed by an overview of task management and list management. These overviews are in preparation for a detailed analysis of the scheduler and queuing mechanisms that follow.

FreeRTOS Configuration

The operation of FreeRTOS is governed significantly by the contents of the **FreeRTOSConfig.h** file. The contents of this file is shown in Figure 2. It was a combination of the **FreeRTOSConfig.h** files for the GCC and Code Warrior based C32 demos used in Attachment 1.



```
/*
 * FreeRTOSConfig.h configures FreeRTOS for GCC/HCS12 version of FreeRTOS Demo
 *
 * Modified by Jefferson L Smith, Robotronics Inc.
 */
#ifndef FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H
/* This port requires the compiler to generate code for the BANKED memory model. */
//#define BANKED_MODEL

/*-----------------------------
 * Application specific definitions.
 * These definitions should be adjusted for your particular hardware and
 * application requirements.
 * THESE PARAMETERS ARE DESCRIBED WITHIN THE 'CONFIGURATION' SECTION OF THE
 * FreeRTOS API DOCUMENTATION AVAILABLE ON THE FreeRTOS.org WEB SITE.
 *-----------------------------*/

#define configUSE_PREEMPTION            1
#define configUSE_IDLE_HOOK             0
#define configUSE_TICK_HOOK             0
#define configTICK_RATE_HZ              ( ( portTickType ) 1000 )
#define configMAX_PRIORITIES            ( ( unsigned portBASE_TYPE ) 4 )
#define configMINIMAL_STACK_SIZE        ( ( unsigned portSHORT ) 70)
#define configTOTAL_HEAP_SIZE           ( ( size_t ) ( 1024 ) )
#define configMAX_TASK_NAME_LEN         ( 1 )
#define configUSE_TRACE_FACILITY        0
#define configUSE_16_BIT_TICKS          1
#define configIDLE_SHOULD_YIELD         1

/* Co-routine definitions. */
#define configUSE_CO_ROUTINES                   0
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )

/* This parameter is normally affects the clock frequency. In this port, at the moment
it might just be used for reference. */

#define configCPU_CLOCK_HZ                      ( ( unsigned portLONG ) 24000000 )

/* Set the following definitions to 1 to include the API function, or zero
to exclude the API function. */

#define INCLUDE_vTaskPrioritySet                1
#define INCLUDE_uxTaskPriorityGet               1
#define INCLUDE_vTaskDelete                     0
#define INCLUDE_vTaskCleanUpResources   0
#define INCLUDE_vTaskSuspend                    1
#define INCLUDE_vTaskDelayUntil                 1
#define INCLUDE_vTaskDelay                              1

#endif /* FREERTOS_CONFIG_H */
```


Figure 2: FreeRTOSConfig.h File for Annex A

Each of the important configuration settings is described briefly below (paraphrased from the customization section of [FRTOS]). The uses of many of the configurable parameters will be described later in this document.

- `configUSE_PREEMPTION`: This is set to 1 if the preemptive kernel is desired. The cooperative kernel was not of interest for this study.
- `configUSE_IDLE_HOOK`: An idle task hook will execute a function during each cycle of the idle task. This is set to 1 if idle hooks are desired. The function will operate at the priority of the idle task. For "probing purposes" an idle hook

would be ideal. However, for basic implementation purposes, this value is set to zero (no idle hooks).

- `configUSE_TICK_HOOK`: A tick hook function will execute on *each* RTOS tick interrupt if this value is set to 1. Again, this will be useful for "probing" the system.
- `configCPU_CLOCK_HZ`: This is the internal MCU clock. The C32 core clock signal, without enabling the PLL, will run at 8 MHz. However, one CPU cycle is equivalent to one bus cycle and the bus runs at half the core frequency. Therefore, the CPU clock frequency is 4 MHz.
- `configTICK_RATE_HZ`: This is the frequency at which the RTOS tick will operate. As the tick frequency goes up, the kernel will become less efficient since it must service the tick interrupt service request (ISR) more often. However, a higher frequency gives a greater resolution in time. A trade study is required within the context of the application to determine an optimal value. Initially, for lack of direction, this will be set to about 1000 Hz (as it is with all the demos).
- `configMAX_PRIORITIES`: The total number of priority levels that can be assigned when prioritizing a task. Each new priority level creates a new list so memory sensitive targets should be stripped to the minimum number of levels possible.
- `configMAX_TASK_NAME_LEN`: The maximum number of characters that can be used to name a task. The character based task names are used mostly for debugging and visualization of the system.
- `configUSE_16_BIT_TICKS`: This configuration item controls whether or not to use a 16-bit or 32-bit counter for recording elapsed time. A 16-bit value will perform better on the HS12C32 because the native counter size is 16-bits. However, this bit size combined with the value set by `configTICK_RATE_HZ` may place an unrealistic upper bound on the total time that can be recorded.
- `configIDLE_SHOULD_YIELD`: This configuration item controls how processes that are running with idle priority react to a preemption request from a higher priority process. If it is set to 0, a process with idle priority is not preempted until the end of its allocated time slice. If it is set to 1, a process with idle priority will yield immediately to the higher priority process. However, the higher priority process will only be given whatever time was left within the time slice originally assigned to the idle task (i.e., it will not have a whole time slice to compute within).
- `configUSE_CO_ROUTINES`: This configuration item controls whether or not co-routines are used. It is set to 0 since this work does not deal with co-routines.
- `configMAX_CO_ROUTINE_PRIORITIES`: Set to 0 (N/A).
- `configUSE_TRACE_FACILITY`: The FreeRTOS core has trace functionality built in. This item is set to 1 if a kernel activity trace is desired. Note that a trace log is created in RAM (so a buffer needs to be identified an more RAM is required
- `configMINIMAL_STACK_SIZE`: This is the stack size used by the idle task. The FreeRTOS authors suggest that this value not be changed from that provided within each demo. However, an analysis of the optimal value should be possible.

- `configTOTAL_HEAP_SIZE`: This configuration item determines how much RAM is used by FreeRTOS for stacks, task control blocks, lists, and queues. This is the RAM available to the heap allocation methods.

The second half of Figure 2 is used to include certain API functionality. Much of this functionality will become evident as the analysis and description proceeds.

Task Management

Overview

This section will describe task management structures and mechanisms used by the scheduler.

Task Control Block

The FreeRTOS kernel manages tasks via the Task Control Block (TCB). A TCB exists for each task in FreeRTOS and contains all information necessary to completely describe the state of a task. The fields in the TCB for FreeRTOS are shown in Figure 3 (derived from **tasks.c**).

Top of Stack	Pointer to last item placed on the stack for this task
Task State	List item that puts the TCB in the ready or blocked queues
Event List	List item used to place the TCB in event lists.
Priority	Task priority (0 = lowest)
Stack Start	Pointer to the start of the process stack
TCB Number	A debugging and tracing field.
Task Name	A task name
Stack Depth	Total depth of the stack in variables (not bytes)

Figure 3: Task Control Block for FreeRTOS

Task State Diagram

A task in FreeRTOS can exist in one of five states. These are *Deleted*, *Suspended*, *Ready*, *Blocked* and *Running*. A state diagram for FreeRTOS tasks is shown in Figure 4.

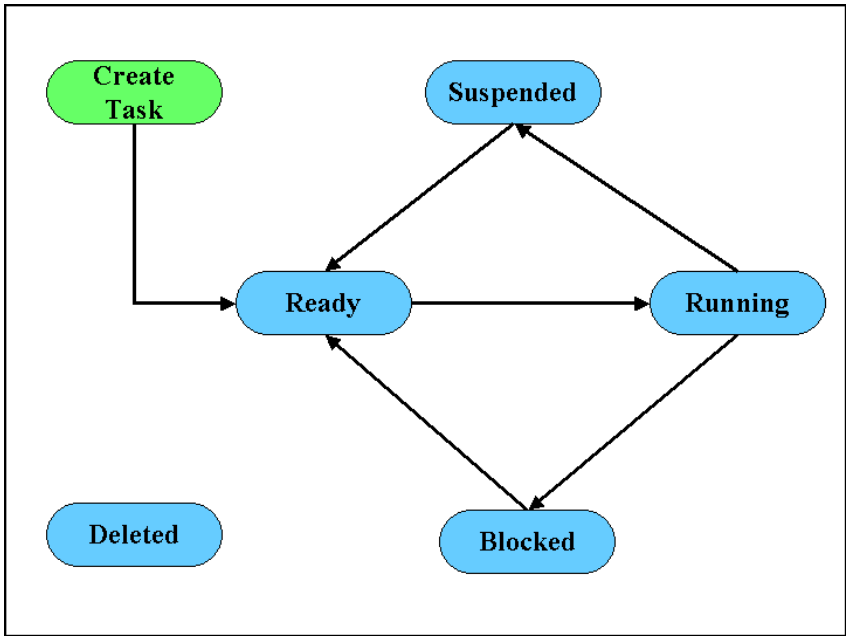


Figure 4: Basic Process State Diagram for FreeRTOS

The FreeRTOS kernel creates a task by instantiating and populating a TCB. New tasks are placed immediately in the Ready state by adding them to the Ready list.

The Ready list is arranged in order of priority with tasks of equal priority being serviced on a round-robin basis. The implementation of FreeRTOS actually uses multiple Ready lists – one at each priority level. When choosing the next task to execute, the scheduler starts with the highest priority list and works its way progressively downward.

The FreeRTOS kernel does not have an explicit "Running" list or state. Rather, the kernel maintains the variable `pxCurrentTCB` to identify the process in the Ready list that is currently running. `pxCurrentTCB` is therefore defined as a pointer to a TCB structure.

Tasks in FreeRTOS can be blocked when access to a resource is not currently available. The scheduler blocks tasks only when they attempt to read from or write to a queue that is either empty or full respectively. This includes attempts to obtain semaphores since these are special cases of queues.

As indicated earlier, access attempts against queues can be blocking or non-blocking. The distinction is made via the `xTicksToWait` variable which is passed into the queue access request as an argument. If `xTicksToWait` is 0, and the queue is empty/full, the task does not block. Otherwise, the task will block for a period of `xTicksToWait` scheduler ticks or until an event on the queue frees up the resource.

Tasks can also be blocked voluntarily for periods of time via the API. The scheduler maintains a "delayed" task list for this purpose. The scheduler visits this task list at every

scheduler decision point to determine if any of the tasks have timed-out. Those that have are placed on the Ready list. The FreeRTOS API provides ***vTaskDelay*** and ***vTaskDelayUntil*** functions that can be used to put a task on the delayed task list.

Any task or, in fact, all tasks except the one currently running (and those servicing ISRs) can be placed in the Suspended state indefinitely. Tasks that are placed in this state are not waiting on events and do not consume any resource or kernel attention until they are moved out of the Suspended state. When un-suspended, they are returned to the Ready state.

Tasks end their lifecycle by being deleted (or deleting themselves). The Deleted state is required since deletion does not necessarily result in the immediate release of resources held by a task. By putting the task in the Deleted state, the scheduler in the FreeRTOS kernel is directed to ignore the task. The IDLE task has the responsibility to clean up after tasks have been deleted and, since the IDLE task has the lowest priority, this may take time.

List Management

Overview

This section provides an overview of list creation and management in FreeRTOS. This information is useful for understanding the functionality of various FreeRTOS modules described in later sections.

Ready and Blocked Lists

Figure 5 shows all of the lists that are created and used by the scheduler and their dependencies on configuration values in **FreeRTOSConfig.h**.

FreeRTOSConfig.h	Lists Created
configMAX_PRIORITIES	ReadyTasksLists[0] : ReadyTasksLists[configMAX_PRIORITIES]
INCLUDE_vTaskDelete == 1	TasksWaitingTermination
INCLUDE_vTaskSuspend == 1	SuspendedTaskList
N/A	PendingReadyList
N/A	DelayedTaskList
N/A	OverflowDelayedTaskList

Figure 5: Lists Created by the Scheduler

Note that the {Ready} list is not a single list but actually *n* lists where

n= `configMAX_PRIORITIES`

Each of the lists in Figure 5 is created as type `xList`. which is a structure defined as shown in Figure 6.

NumberOfItems	The number of items in the list.
(xListItem) * pxIndex	Pointer used to walk through the list. It points to successive list items in the list
(xMiniListItem) xListEnd	A list item that marks the end of the list. It contains the maximum value in xItemValue and therefore always appears at the end of the list.

Figure 6: Type xList

Each list has an entry identifying the number of items in the list. The list has an index pointer `pxIndex` that points to one of the items in the list (which is used to iterate through a list). The `pxIndex` points to type `xListItem` which is the only type that a list can hold. The only exception is `xListEnd` which is of type `xMiniListItem`. The structures `xListItem` and `xMiniListItem` are shown in Figure 7.

xListItem

xItemValue	The value being listed – normally a time (value is defined as portTickType). Used to order the list.
*** pxNext**	Pointer to the next List Item in the list
*** pxPrevious**	Pointer to the previous List Item in the list.
*** pvOwner**	Pointer to the object that contains the list item. This is normally a TCB
*** pvContainer**	Pointer to the list in which this list item is placed.

xMiniListItem

xItemValue	The value being listed – normally a time (value is defined as portTickType). Used to order the list.
*** pxNext**	Pointer to the next List Item in the list
*** pxPrevious**	Pointer to the previous List Item in the list.

Figure 7: xList Types

List Initialization

Figure 8 shows an example of the initialization of the list {DelayedTaskList}. The number of items is initially set to zero. The `pxIndex` pointer and `pxNext` and `pxPrevious` pointers are all set to the address of the `xListEnd` structure.

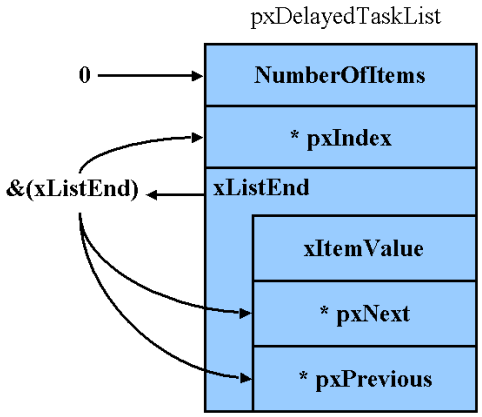


Figure 8: List Initialization

The `xItemValue` in the `xListEnd` structure must hold the maximum possible value. Because {DelayedTaskList} is used to list tasks based on the amount of time that they can block, this value is set to `portMAX_DELAY`.

Inserting a Task Into a List

To insert a task into a list (for example, the {DelayedTaskList}), FreeRTOS uses ***vListInsert***. Arguments to this function include the pointer to the list to be modified and a pointer to the Generic List Item portion of the TCB about to be listed as shown in Figure 9.

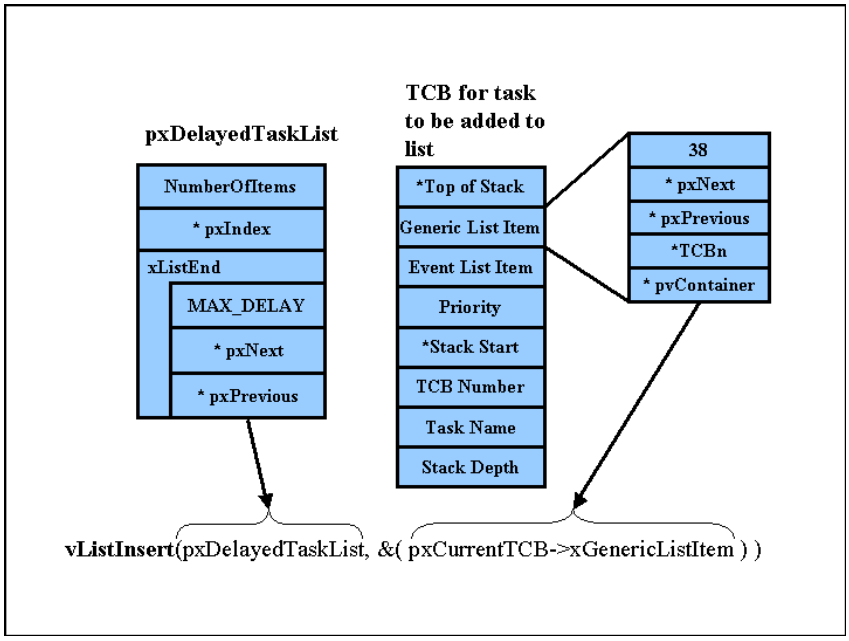


Figure 9: ***vListInsert*** With Arguments

In the figure, the `xItemValue` within the Generic List Item field has already been set to 38 (an arbitrary number). In this case, that would represent the absolute clock tick upon which the task associated with this TCB should be woken up and re-inserted into the {Ready} list. Also note that the `*pvOwner` pointer has been set to point to the TCB containing the Generic List Item. This allows fast identification of the TCB.

Figure 10 shows and example of what a {DelayedTaskList} might look like with two listed tasks. The `*pxNext` pointer in the `xListEnd` structure of the list is **not** NULL – it points to the **first** entry in the list as shown in the figure.

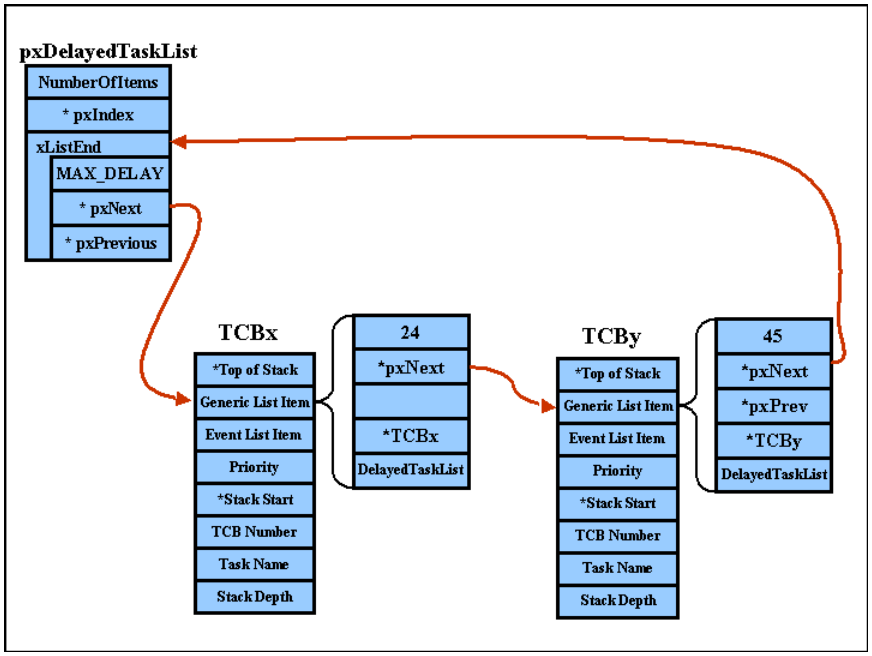


Figure 10: Hypothetical DelayedTaskList

To insert a new task into the {DelayedTaskList}, `vListInsert` proceeds as follows. The `xItemValue` within the new `Generic List Item` is compared with the `xItemValue` from the first TCB in the list. In the {DelayedTaskList} case, this will be the absolute clock tick on which the task should be woken. If the existing value is lower (in this case, 24<38), the `*pxNext` pointer is used to move on to the next TCB in the list. When the comparison fails, then the current TCB must be "moved to the right" while the new task TCB is inserted. This is done by modifying the `*pxNext` and `*pxPrev` pointers of the adjacent list items and both the `*pxNext` and `*pxPrev` pointers within the new TCB itself. Finally, the `*pxContainer` pointer in the newly listed TCB is modified to point to the {DelayedTaskList}. This pointer is apparently used for quick removal at a later time. Once the new TCB is entered, the `NumberOfItems` value in the {DelayedTaskList} structure is updated.

The code that implements this insertion is shown in Figure 11. Normally, code segments will not be presented within this document. However, in this case, the code is exceptionally concise and therefore worthy of presentation.

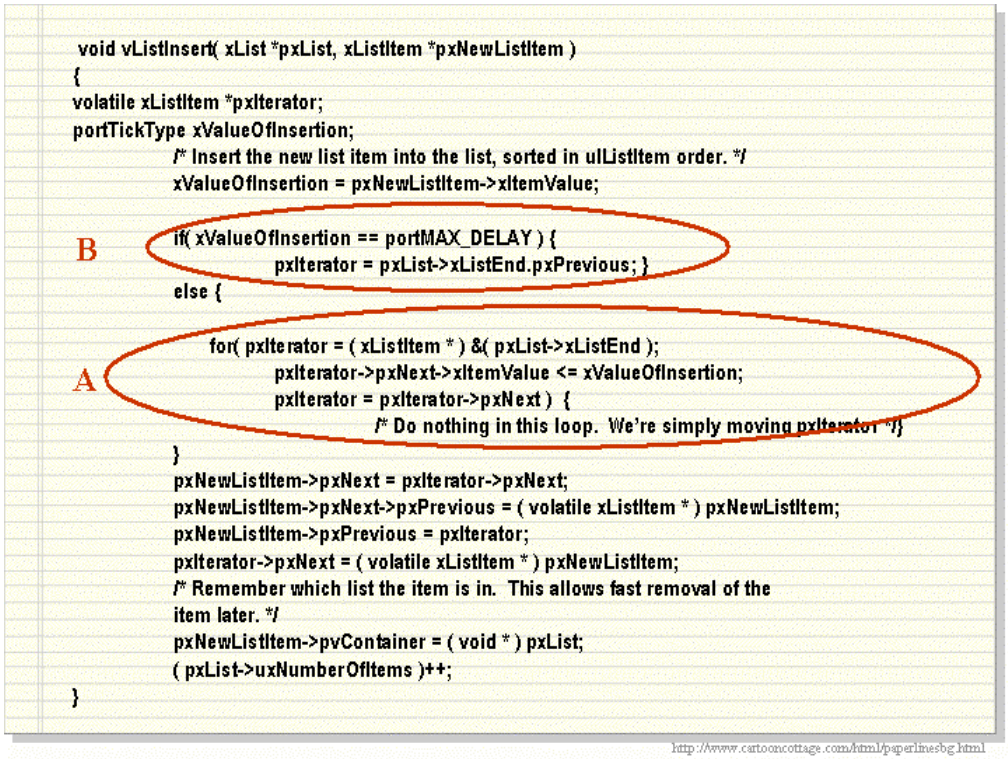


Figure 11: Code Extract from **Lists.c** in FreeRTOS

The *for* loop at point A initializes `pxIterator` (having type `ListItem`) to the last item in the list which is, by default, `ListEnd`. As mentioned, the `*pxNext` pointer of `ListEnd` points to the first item in the list. The comparison operation in the *for* loop checks the `xItemValue` in the structure pointed to by the current `*pxNext` and, if true, `pxIterator` takes on the value of the next list item.

It should be noted that a boundary condition occurs when the new `xItemValue` is equal to `portMAX_DELAY` as defined in **FreeRTOSConfig.h**. FreeRTOS handles this exception at point B in Figure 11 by assigning the task the second last place in the list.

Timer Counter Size and {DelayedTaskList}

Tasks that are placed on the {DelayedTaskList} are placed there by the scheduler or by API calls such as ***vTaskDelay*** or ***vTaskDelayUntil***. In all cases, an absolute time is calculated for the task to be woken. For example, if the task is to delay for 10 ticks, then 10 is added to the current tick count and that becomes the `xItemValue` to be stored in the Generic List Item structure.

However, the embedded controllers being targeted by FreeRTOS have counters that can be as small as 8-bits – resulting in a counter rollover after only 255 ticks. To deal with this, FreeRTOS defines and uses two delay lists – {DelayedTaskList} and {OverflowDelayedTaskList}.

As shown in Figure 12, the time to sleep is added to the current time at point A. At point B, if the sum turns out to be less than the current timer value, then the time to wake should be inserted into the {OverflowDelayedTaskList}.

```
/* Calculate the time to wake - this may overflow but this is not a problem. */
xTimeToWake = xTickCount + xTicksToDelay;                                    A

/* We must remove ourselves from the ready list before adding ourselves to the
blocked list as the same list item is used for both lists. */

vListRemove( ( xListItem * ) &( pxCurrentTCB->xGenericListItem ) );

/* The list item will be inserted in wake time order. */

listSET_LIST_ITEM_VALUE( &( pxCurrentTCB->xGenericListItem ), xTimeToWake );

if( xTimeToWake < xTickCount )                                               B
{
        /* Wake time has overflowed.  Place this item in the
        overflow list. */
        vListInsert((xList*)pxOverflowDelayedTaskList,(xListItem*)&(pxCurrentTCB->xGenericListItem));
}
else
{
        /* The wake time has not overflowed, so we can use the current
        block list. */
        vListInsert((xList*)pxDelayedTaskList,(xListItem*)&(pxCurrentTCB->xGenericListItem ) );
}
```


Figure 12: Deciding Which Delayed List To Insert (from **Task.c**)

Note that, for this to work, the maximum number of ticks that a task can be blocked must be less than the size of the counter (i.e., **FF** in the case of an 8-bit counter). This maximum value is set in the **FreeRTOSConfig.h** variable `portMAX_DELAY`.

Each time the tick count is increased (in the function ***vTaskIncrementTick***), a check is performed to determine if the counter has rolled over. If it has, then the pointers to {DelayedTaskList} and {OverflowDelayTaskList} are swapped as shown in the code segment in Figure 13.

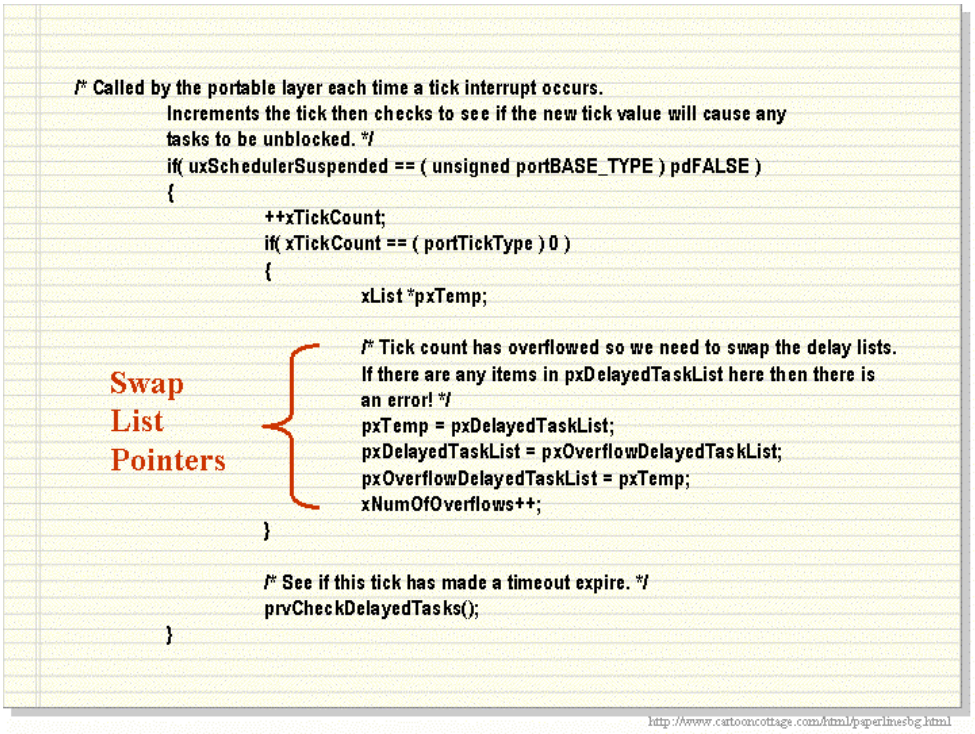


Figure 13: Exchanging List Pointers When Timer Overflows

The FreeRTOS Scheduler

Overview

This section provides a detailed overview of the scheduler mechanism in FreeRTOS. Because of the configuration options that allow cooperative operation and scheduler suspension, the scheduler mechanism has considerable complexity.

Figure 14 provides an overview of the scheduler algorithm. The scheduler operates as a timer interrupt service routine (***vPortTickInterrupt***) that is activated once every tick period. The tick period is defined by configuring the **FreeRTOSConfig.**h parameter `configTICK_RATE_HZ`.

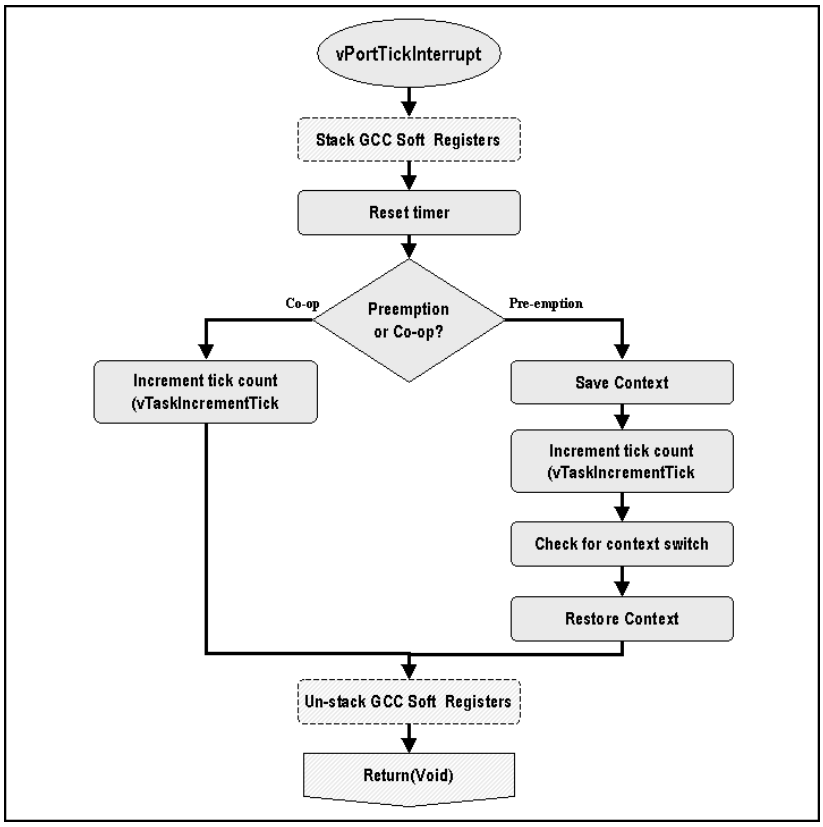


Figure 14: Scheduler Algorithm

Because the scheduler operates as an interrupt, it is part of the HAL and contains implementation specific code. In Figure 14, the HAL implementation for the 68HC12 includes stacking (and un-stacking) a set of "soft registers" that are used by GCC (shown in the sections with dashed lines). Details of the nature and use of soft registers can be found in [GCC1].

The first operation performed by the scheduler is to reset the counter timer (a hardware specific instruction) in order to start the next tick period. FreeRTOS can be configured to be co-operative or preemptive. In the scheduler, after the clock is reset, the **FreeRTOSConfig.h** variable `configUSE_PREEMPTION` is referenced to determine which mode is being used.

In the co-operative case, the only operation performed before returning from the timer interrupt is to increment the tick count. There is a significant amount of logic behind this operation that is required in order to deal with special cases and timer size limitations. We will visit that logic shortly.

If the scheduler is preemptive, then the first step is to stack the context of the current task in the event that a context switch is required. The scheduler increments the tick count and then checks to see if this action caused a blocked task to unblock. If a task did unblock and that task has a higher priority than the current task, then a context switch is executed. Finally, context is restored, soft registers are un-stacked, and the scheduler returns from the interrupt.

Task Context Frame

The following several paragraphs describe the construction of the FreeRTOS "context frame" and the mechanism by which a context switch is executed. A task's context is constructed from data that is provided automatically as part of interrupt servicing as well as additional context information provided from several macros. It is important to know what is expected within a context frame and how to populate it when both starting a task or when performing a context switch between tasks.

When an ISR occurs, the HCS12 (like most other embedded MCUs) will immediately stack the *MCU context* using the current stack pointer. The MCU context for the HCS12 consists of the program counter (the return address), the Y and X registers, the A and B registers, and the condition code register (CCR) [S12CPUV2]. All of these registers are stacked in the order just indicated prior to the MCU jumping to the interrupt service routine. Figure 15 shows a task, Task 1, with its associated TCB and stack space both prior to an ISR and immediately before the ISR takes control of the MCU.

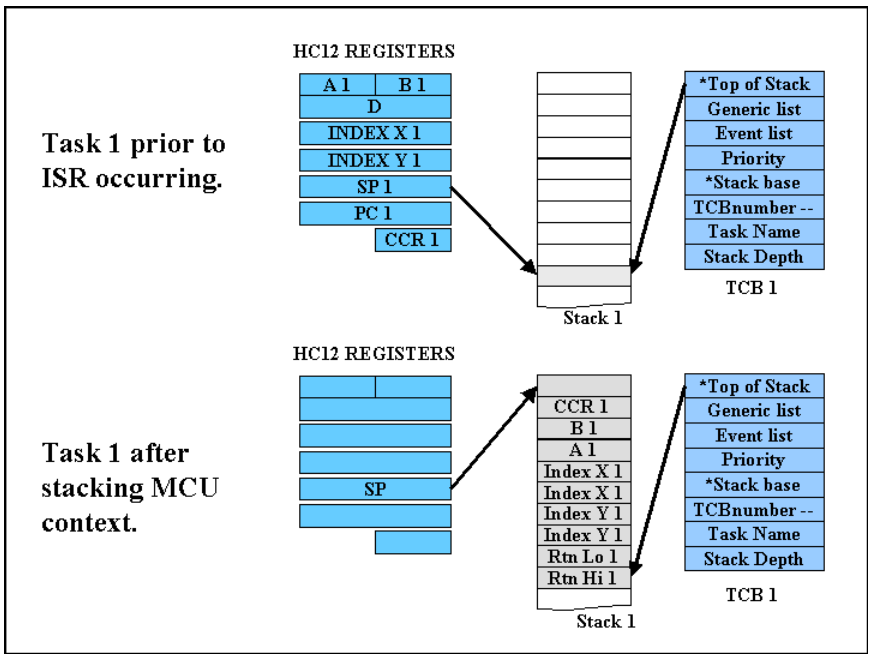


Figure 15: Stacking of MCU Context

In the GCC implementation of FreeRTOS on the HC11 or HC12 MCU, up to 12 bytes of "soft registers" are stacked on top of the MCU state provided by the ISR mechanism. These registers are stacked explicitly by executing the ***portISR_HEAD*** macro within the HAL. They are un-stacked using ***portISR_TAIL***.

The final context information is provided by executing the ***portSAVE_CONTEXT*** macro within the HAL. This macro first stacks a variable that tracks the critical nesting depth for the task (discussed later). If the target had been using the banked memory model for Freescale devices, then the PPAGE register would also be stacked. The macro then stores the current value of the stack pointer register into the head entry of the TCB for Task 1. The context frame, as built by the ISR mechanism, portISR_HEAD, and portSAVE_CONTEXT is shown in Figure 16

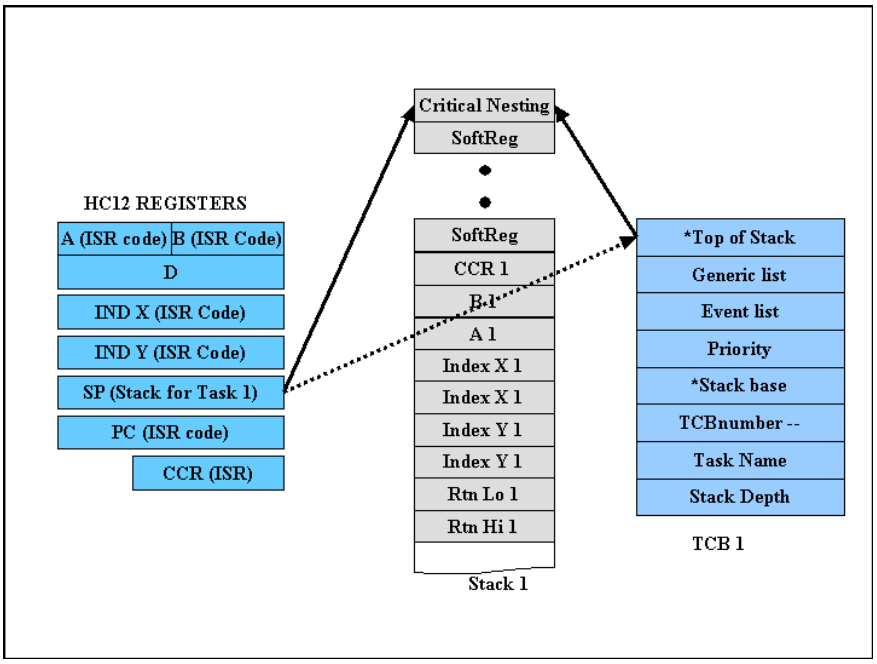


Figure 16: Context Frame on Stack 1

To exit from the ISR following its work, ***portRESTORE_CONTEXT***, ***portISR_TAIL***, and an RTI must be executed in that sequence in order to properly clear the stack of the context frame.

Context Switch By Stack Pointer Manipulation

In Figure 14, one of the tasks of the scheduler is to determine if a context switch is required. If that is the case, then a stack pointer manipulation is performed to execute the switch. The scheduler copies the head entry of Task 2 into the stack pointer register (recall that the head entry is a pointer to the stack space of Task 2). If the context of Task 2 had been saved according to the context frame definition, then executing ***portRESTORE_CONTEXT***, ***portISR_TAIL***, and an RTI will restore the context of Task 2 to the MCU.

Starting and Stopping Tasks

Although the act of starting or stopping a task is not a direct scheduler function, a brief description will be provided here since the scheduler manipulates the data structures and stacks created when a task is created. Therefore, an understanding of task creation and deletion will assist in describing the remaining scheduler functions.

Tasks are created by invoking ***xTaskCreate()*** from within **main.c** or within a task itself. Parameters required to create a task include:

- A pointer to the function that implements the task. For obvious reasons, the code that implements the task function must invoke an infinite loop.

- A name for the task. This is used mainly for code debugging and monitoring in FreeRTOS.
- The depth of the task's stack.
- The task's priority.
- A pointer to any parameters needed by the task function.

An overview of the process of creating a task is shown in Figure 17.

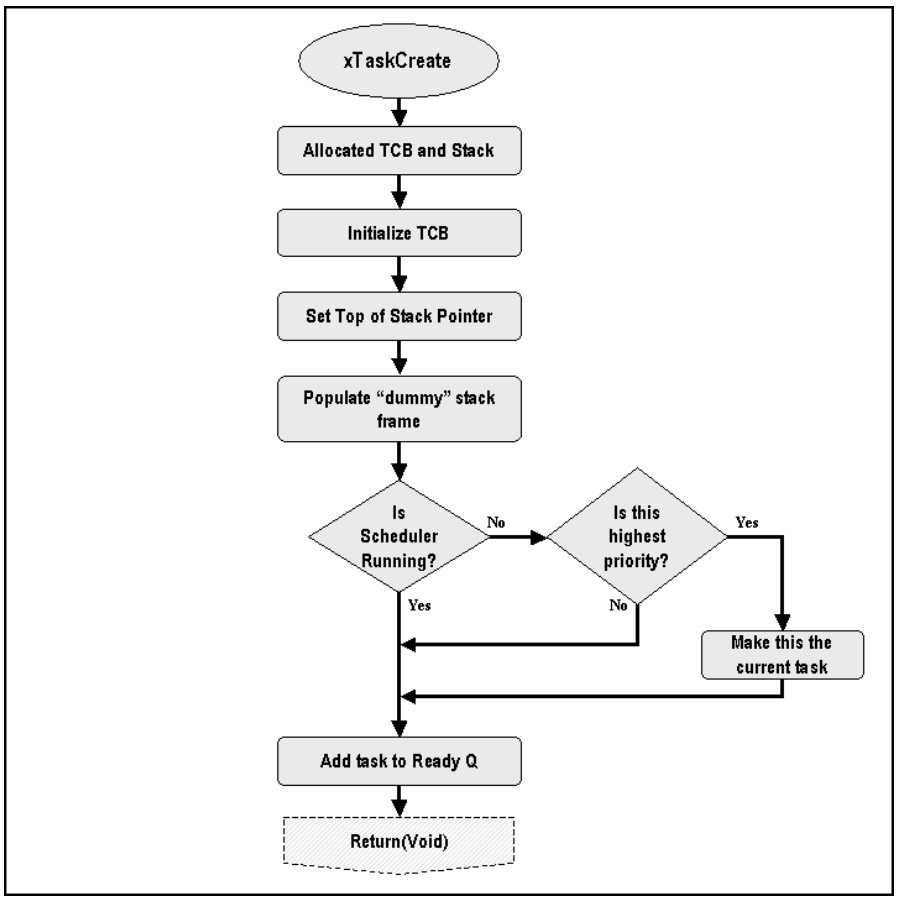


Figure 17: Overview of Task Creation

xTaskCreate must first allocate memory for the task's TCB and stack. This is accomplished by calling ***AllocateTCBandStack*** as shown in Figure 18. This function invokes ***portMalloc*** to obtain a block of memory for the TCB that is the size of the TCB structure and a block of memory for the stack that is the size of the stack data type (e.g., 8, 16 bits) multiplied by the size of the stack requested. These two memory blocks are obtained from the heap whose maximum size is specified in the FreeRTOSConfig parameter `configTOTAL_HEAP_SIZE`. As a final exercise, ***AllocateTCBandStack*** sets a pointer to the base address of the stack inside the TCB.

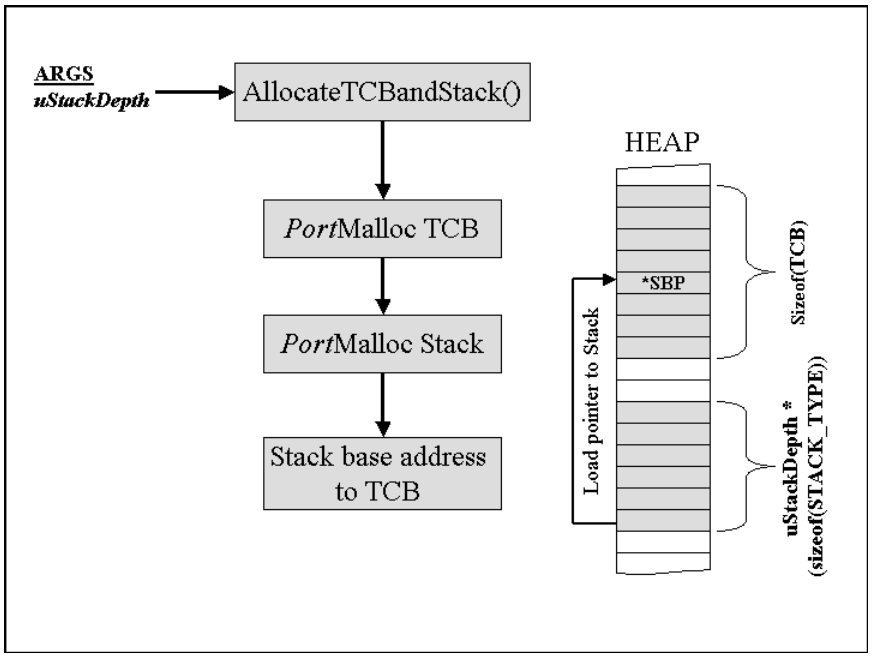


Figure 18: Allocate Stack and TCB Memory

portMalloc is implemented within the HAL. Specifically, by choosing to compile one of **heap1.c**, **heap2.c**, or **heap3.c** with the project, a range of memory allocation strategies (and the corresponding ***portMalloc*** implementations) can be achieved. For example, heap1.c implements a policy of allocating heap memory to a task once and does not allow deallocation of that memory. This policy is good for applications with a known set of tasks that will not vary with time. The policy invoked in **heap2.c** allows for allocation and deallocation of heap memory using best-fit to located the request block but it does not perform cleanup on fragmented but adjacent blocks. The allocation policy in **heap3.c** simply provides wrappers for traditional ***malloc()*** and ***calloc()*** allocation.

Referring back to Figure 17, the second task performed by ***xTaskCreate*** (assuming memory was successfully obtained) is to initialize the TCB with known values. This includes initializing the task name, task priority, and stack depth fields of the TCB from function call parameters.

The third and fourth steps in ***xTaskCreate*** prepare the task for its first context switch. A pointer to the top-of-stack is initialized to the base stack address found in the task TCB (an adjustment is necessary depending on the stack growth mechanism on the target – some targets "grow" the stack towards lower memory while others do the opposite). The stack for the task is then populated with a **dummy frame** that perfectly matches what is required when a context switch is performed by a combination of ***portRESTORE_CONTEXT*** and ***portISRTAIL*** macros as discussed earlier. The content of the dummy frame is shown in Figure 19. The important element of the dummy frame is the return address which will point to the start address of the task code.

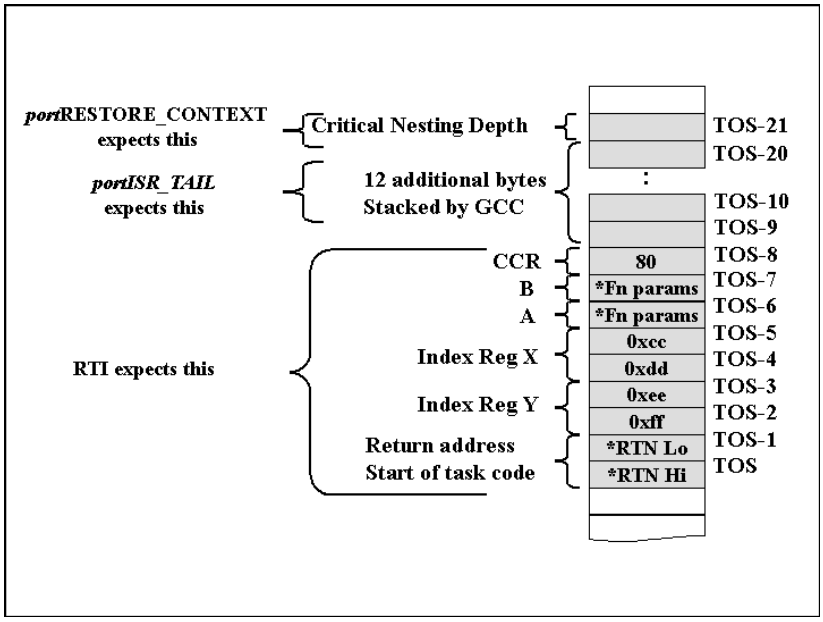


Figure 19: Dummy Stack Frame

After populating the dummy stack for the task, the top-of-stack pointer (which now points to TOS-21 in Figure 19) is updated and written back to the TCB. This stack pointer is the head value in the TCB is extracted directly to perform a context switch as discussed earlier.

Each time a task is created successfully, ***xTaskCreate*** must determine if the scheduler is running. If it is running, then the new task can simply be added to the Ready list and the scheduler will, on its first (or next) interrupt, determine which task has the highest priority. If the scheduler is not running, then xTaskCreate must determine if the task just created is the highest priority task and then ensure that this is tracked (using the `pxCurrentTCB` global variable).

The final step in creating the task is to add it to the Ready list. This is performed by a call to the function ***prvAddTaskToReadyQueue***. This function determines what priority level the task is and adds it to the back of the appropriate Ready list. If a list does not exist at that priority (which would only happen if the task were created dynamically after startup), then the appropriate list is first created. `pxCurrentTCB` is adjusted by ***prvAddTaskToReadyQueue*** to track the TCB to be context-switched in next.

Yeilding Between Ticks

The scheduler responds to the timer ISR. A second ISR is required for yielding a task in the event of being blocked or completing early. This is implemented by a software interrupt (SWI). Any call to ***portYIELD*** causes the assembly instruction "swi" to execute which, in turn, invokes the ISR code attached to that interrupt (defined in **port.c** as ***vportYIELD***). The SWI builds a context frame as described previously – it executes

portISR_HEAD and ***portSAVE_CONTEXT***, determines if any context switch is required (and loads the new task's TCB head into the stack pointer if necessary), and then un-stacks the context frame as appropriate. Note that an SWI is non-maskable whereas the timer responsible for the scheduler can be masked.

Starting the Scheduler

Figure 20 shows the process that occurs when the FreeRTOS scheduler is started. A call to the FreeRTOS function ***vTaskStartScheduler()*** should be the last function call made in **main.c** after all of the other required tasks have been created using the function ***xTaskCreate()***.

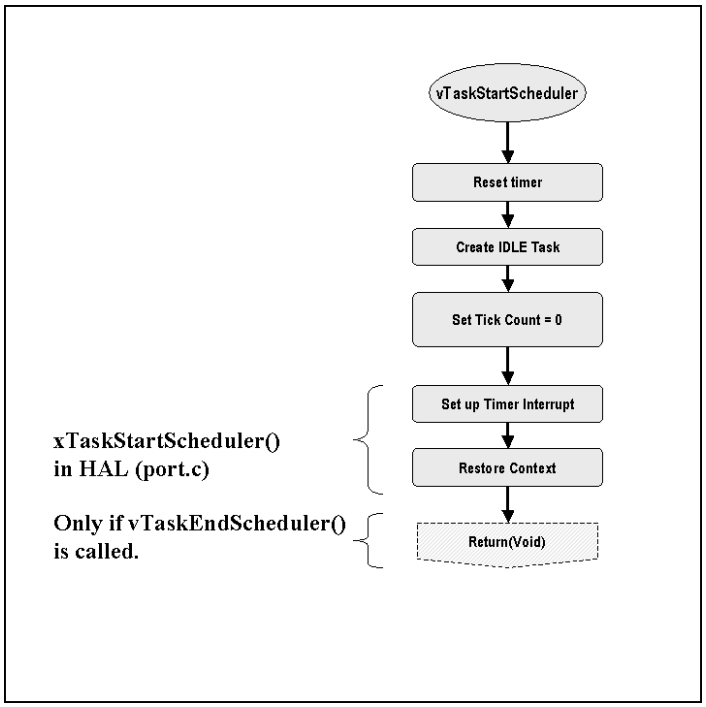


Figure 20: FreeRTOS Task Scheduler Startup

The ***vTaskStartScheduler()*** function first creates the IDLE task with the lowest priority and then sets the global timer `xTickCount` to zero. The global `xSchedulerRunning` is set to TRUE. This variable is used to in several areas to determine if the scheduler is available to make scheduler decisions or if those decisions need to be made locally. For example, tasks can be created before or after the scheduler is started. When tasks are created before, the creation mechanism includes a method to determine whether the task just created is the new priority task and switches `pxTCBCurrent` to reflect that status (without performing a context switch). Otherwise, the scheduler is used.

vTaskStartScheduler() passes control to ***xTaskStartScheduler()*** in the HAL. The HAL is needed at this point because the first order of business for ***xTaskStartScheduler()*** is to set up a timer

interrupt to invoke the scheduler. Since the timer is hardware dependent, configuring it must occur in the HAL.

The last thing that ***xTaskStartScheduler()*** does is to restore the context of the currently selected task which is pointed to by `pxTCBCurrent` and which, by virtue of the previous operations, is the highest priority task. The context is switched by calling ***portRESTORE_CONTEXT*** and ***portISR_TAIL***. This might seem to be a logic error since no task was previously running. However, as described earlier, each task is provided with a dummy stack frame when it is first created. This frame provides the start address of the task and the head entry of the TCB for the task is a pointer to the top of the task stack. This is all the information required to initiate the task.

Suspending the Scheduler

FreeRTOS provides a task the ability to monopolize the MCU from all other tasks for an unlimited amount of time by suspending the scheduler. Indeed, this capability is used by FreeRTOS itself. A task might conceivably suspend the scheduler in the event that it would like to process for a long period but not miss any interrupts. Using a critical section blocks all interrupts – including the timer interrupts. Extending the critical section for longer than necessary breaks the basic tenet of keeping critical sections short both in time and space.

Regardless, normal scheduler operation can be suspended through the use of ***vTaskSuspendAll*** and ***vTaskResumeAll***. ***vTaskSuspendAll*** guarantees that the current process will not be pre-empted while at the same time continues to service interrupts (including the timer interrupt). Normal scheduler operation is resumed by ***vTaskResumeAll***.

Scheduler suspensions are nested. The nesting depth is tracked by the global variable `uxSchedulerSuspended` in **tasks.c**. Figure 20 shows the algorithms implemented for ***vTaskSuspendAll*** and ***vTaskResumeAll***.

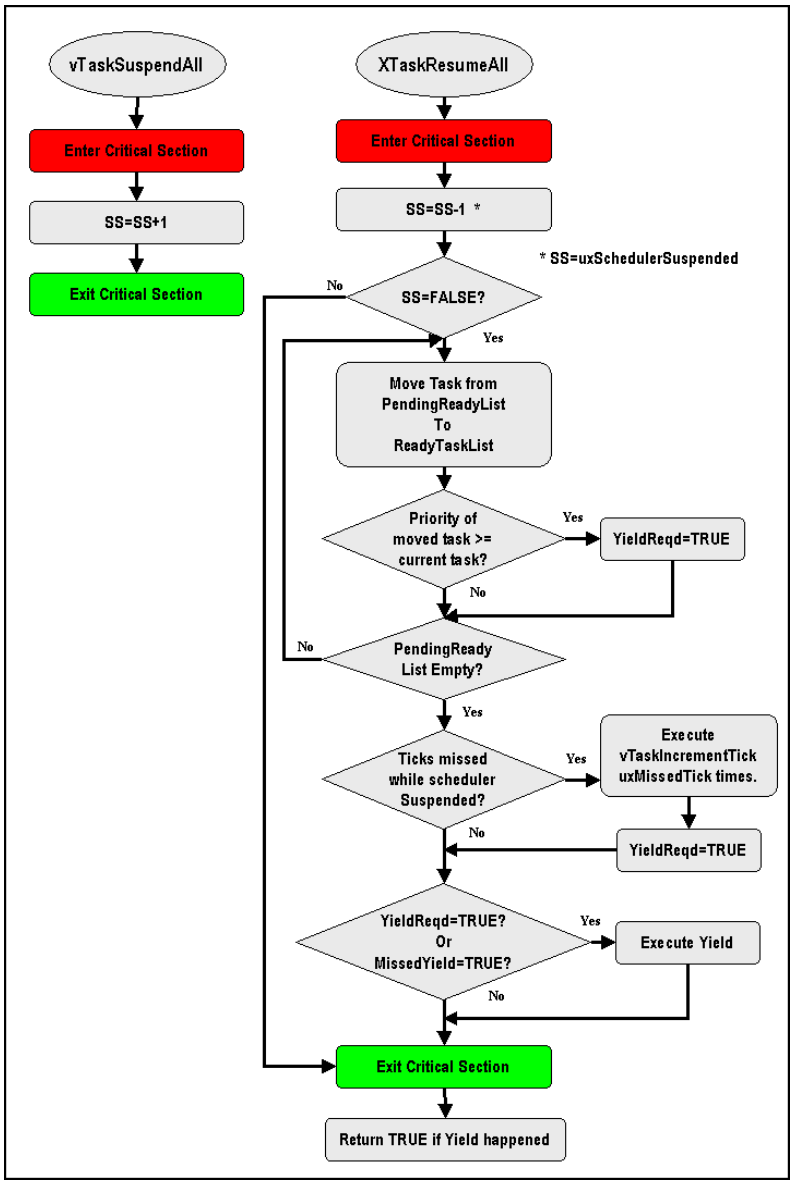


Figure 20: Algorithms for ***vTaskSuspend*** and ***xTaskResumeAll***

Each time ***vTaskSuspend*** is executed, `uxSchedulerSuspended` is incremented. Each time ***xTaskResumeAll*** is executed, ***uxSchedulerSuspended*** is decremented. If `uxSchedulerSuspended` is not made zero (FALSE) when ***xTaskResumeAll*** is executed, then nothing in that function is performed.

However, if `uxSchedulerSuspended` is made FALSE in ***xTaskResumeAll***, then all tasks that were placed on the {PendingReadyList} are moved to the {ReadyTasksList}.

A small digression is required to understand the general concept of the {PendingReadyList} (it will be explained in greater detail shortly). While the scheduler is suspended, tasks on the delayed lists or event lists are not being checked on each timer

tick to see if they should be woken up. However, suspending the scheduler does not stop ISRs from executing and these may cause events that will unblock tasks. However, while the scheduler is suspended, the ISR cannot modify the ready list. Therefore, tasks that are made ready as a result of an ISR are placed on the {PendingReadyList} and are serviced by the scheduler when it is no longer suspended.

In ***xTaskResumeAll***, as each task on the {PendingReadyList} is reassigned to the {ReadyTasksList}, the priority of that task is compared to the priority of the currently executing task. If it is greater, then a yield is required as soon as practicable in order to get the higher priority task in control. Note that there may be more than one task with a higher priority than the current one – the yield will determine which is the highest and context switch to that one. A yield is required because it is essential to move with haste to the higher priority task – otherwise, the current task will execute until the next tick.

If timer ticks were missed while the scheduler was suspended, these will show up in the global variable `uxMissedTicks`. ***xTaskResumeAll*** will attempt to catch up on these ticks by executing ***vTaskIncrementTick*** in bulk (once for each `uxMissedTicks`). If missed ticks existed and were processed, they may have made one or more tasks Ready with higher priority than the currently executing task. Therefore, a yield is once again required as soon as practicable.

The algorithm for ***vTaskIncrementTick*** is shown in Figure 21. *vTaskIncrementTick* is called once each clock tick by the HAL (whenever the timer ISR occurs). The right hand branch of the algorithm deals with normal scheduler operation while the left hand branch executes when the scheduler is suspended. As discussed earlier, the right hand branch simply increments the tick count and then checks to see if the clock has overflowed. If that's the case, then the {DelayedTask} and {OverflowDelayedTask} list pointers are swapped and a global counter tracking the number of overflows is incremented. An increase in the tick count may have caused a delayed task to wake up so a check is again performed.

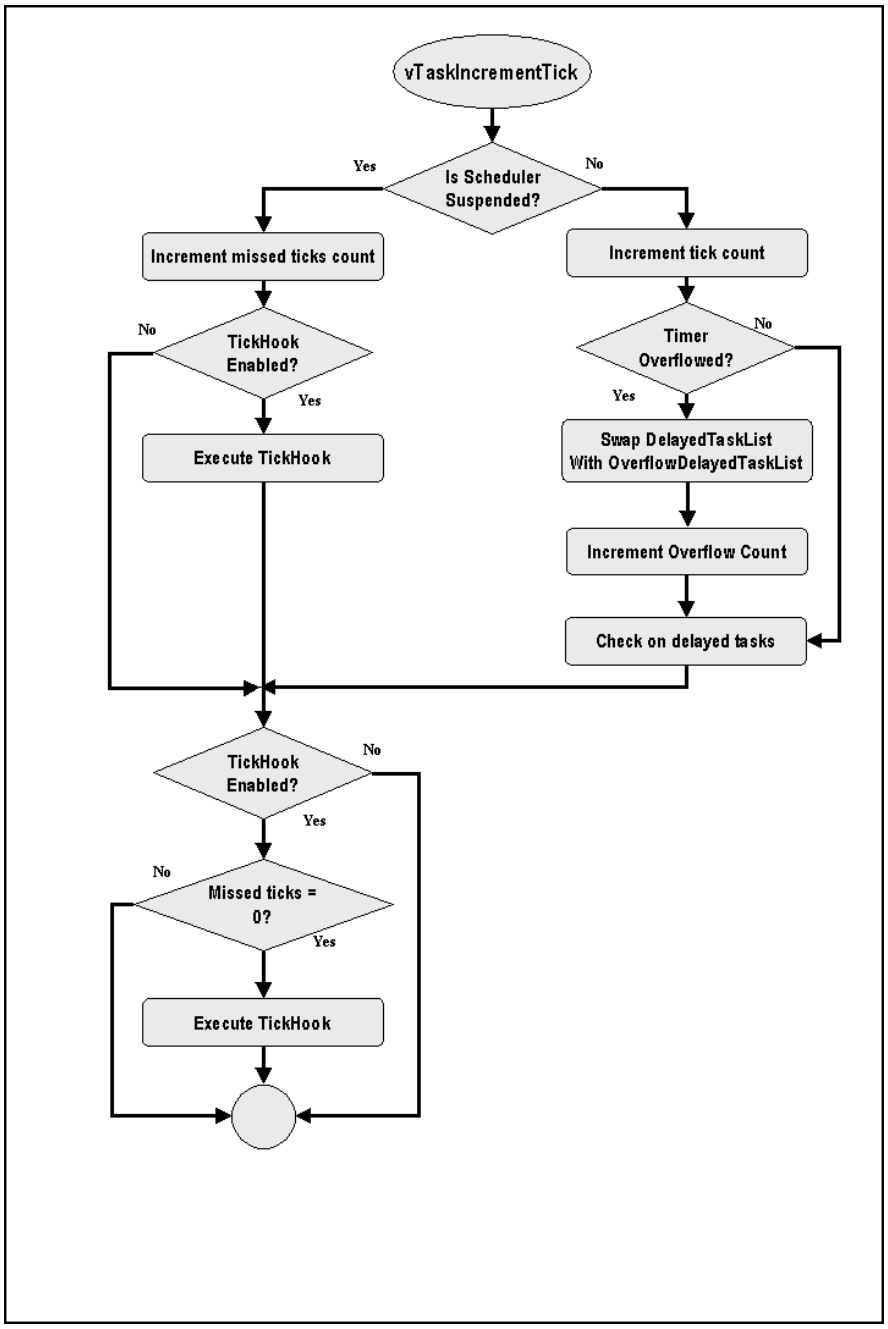


Figure 21: Algorithm for ***vTaskIncrementTick***

If the scheduler *is* suspended, then the global missed tick counter is incremented. If tick hooks were enabled in **FreeRTOSConfig.h**, then any tick hook function is executed – note that tick hooks operate regardless of whether the scheduler is suspended or not.

The final section of the algorithm again checks to see if tick hooks are enabled. Further, a check is performed to see that there are no missed ticks. If both conditions are true, then the tick hook function is executed. This final section will ensure that tick hooks are executed each time that ***vTaskIncrementTick*** is executed and the scheduler is not suspended –

except when tick counts are being processed in bulk to reduce the missed tick count to zero (for example, as discussed in Figure 17 for ***xTaskResumeAll***). In that case, tick hooks will only be executed once for the entire set of missed ticks.

Checking the Delayed Task List

The scheduler checks {DelayedTaskList} once each tick and locates any tasks whose absolute time is less than the current time. These tasks are moved into the appropriate Ready list. Delayed tasks are stored in {DelayedTaskList} in the order of their absolute wake time. Therefore, checking completes when the first delayed task with an un-expired time is found.

Critical Section Processing

FreeRTOS implements critical sections by disabling interrupts. Critical sections are invoked through the *taskENTER_CRITICAL()* macro definition (which maps to *portENTER_CRITICAL()* since entering a critical section will invoke operations in the HAL). An equivalent *taskEXIT_CRITICAL()* exists.

Critical sections in FreeRTOS can be nested. Nesting will occur when a function enters a critical section to perform some processing and, while in that section, calls another function that also calls a critical section (the two functions may be designed to operate independently). If nesting is not performed, the second function will execute an exit from the critical section (turning interrupts back on) when it is complete and then return to the first function which is expecting interrupts to be disabled. By using a nesting counter, each function increments the count on entry and decrements on exit. If the count is decremented and equals zero, interrupts can safely be enabled.

In FreeRTOS, the nesting depth is held in the `uxCriticalNesting` variable which is stacked as part of task context. This means that each task keeps track of its *own* critical nesting count because it is possible for a task to yield from within a critical section (need to find an example of this).

Queue Management

Overview

This section provides an overview of queue creation and management. The mechanisms used to implement blocking and non-blocking accesses to a queue (queue read) are described in depth. Queue writes are very similar to reads and will only be peripherally described.

Figure 22 shows the elements of a queue structure. Two structures of type `xList` hold the {TasksWaitingToSend} and {TasksWaitingToReceive} event lists. The items in these event lists are sorted and stored in order of task priority so that taking an item from the list head is equivalent to obtaining the highest priority item without searching.

***pcHead**	**Pointer to the byte at the start of Q in memory**
***pcTail**	**Pointer to the byte at the end of the Q in memory (one more than necessary)**
***pcWriteTo**	**Pointer to the next free byte in the Q**
***pcReadFrom**	**Pointer to the last byte that was read from the Q**
(xList) TasksWaitingToSend	**List of tasks (in priority order) waiting to send on this Q**
(xList) TasksWaitingToReceive	**List of tasks (in priority order) waiting to receive on this Q**
uxMessagesWaiting	**Number of items currently in the Q**
uxLength	**The length of the Q in Q-able items (not bytes).**
uxItemSize	**The size of each Q-able item in bytes.**
xRxLock	**The number of items received (removed) from the Q while the queue was locked.**
xTxLock	**Stores the number of items transmitted (added) to the Q while the queue was locked**

Figure 22: Queue Structure Elements

The physical queue size is determined by the number of queueable items (`uxLength`) multiplied by the size (in bytes) of each item (`uxItemSize`). This is an important factor to keep in mind when calculating space requirements for memory-constrained applications.

Figure 23 shows a logical overview of a queue and the entities that affect it or are affected by it. It also shows the initial positions of the `*pcHead`, `*pcTail`, `*pcWriteTo`, and `*pcReadFrom` pointers (assuming that the head of the queue is the left-most position).

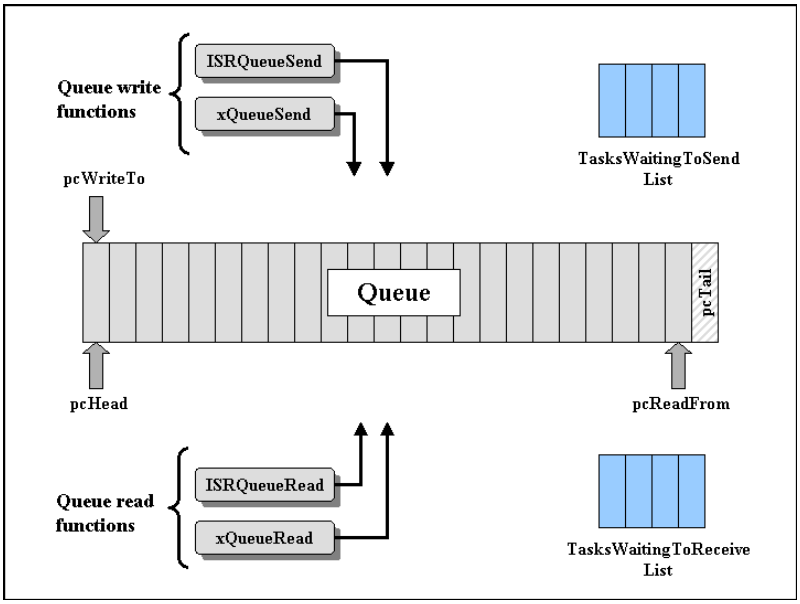


Figure 23: Queue Elements

When a blocking task fails to read or write to a queue, it is placed in one of the waiting lists shown in the figure. The difference between a blocking task and a non-blocking task in FreeRTOS is the number of ticks that a task should wait when blocked. If the number of ticks is set to zero, the task does not block. Otherwise, it blocks for the period specified. As a result, *every task that ends up on either the TaskWaitingToReceive or TaskWaitingToSend event lists will also end up in the DelayedTasks list*. A task that is on either list will be made Ready when its delay time expires or when an event occurs that frees it from the waiting list.

Queues can be written via an API call or from within an ISR. Since ISRs are atypical, their behaviour when writing to the queue is different from that of a normally schedulable task. Therefore, there are two implementations for writing to a queue. The situation is similar for reading from a queue.

Posting to a Queue from an ISR

The most significant difference between an ISR-based queue post and one that originates from within a schedulable task is that ISR-based posts are *non-blocking*. If the queue is not ready to receive data, the send attempt fails without signaling an error.

The algorithm followed by ***ISRQueueSend*** is shown in Figure 24. If the Queue is not full, data from the ISR is copied into it. At this point, the algorithm must check to see if the act of posting data into the queue is an event that would un-block a process that is waiting to read from the queue.

The first step is to determine if the queue is locked. If it is, then it is forbidden for the ISR to modify the event list. However, the fact that the queue was written must be recorded so *TxLock* is incremented for later action.

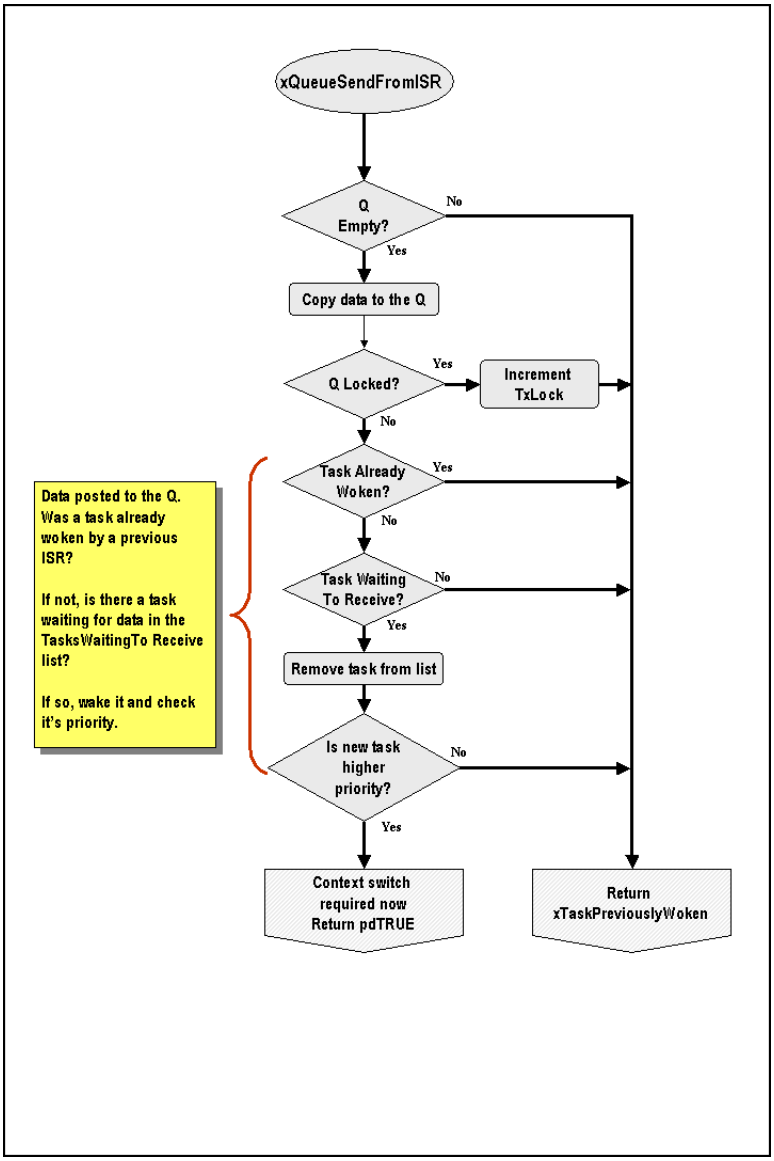


Figure 24: Algorithm for Sending to a Queue from an ISR

If the queue is not locked, the algorithm checks to see if a task has already been un-blocked (or woken) by a previous write to the queue. In order to appreciate this logic, it is important to understand that a single ISR can write many times to the same queue by invoking ***xQueueSendFromISR*** multiple times (for example, placing one queue object at a time as they are received). Therefore, to prevent each subsequent post from pulling another task off of the event list, history is maintained via the `xTaskPreviouslyWoken` variable.

On the initial call to ***xQueueSendFromISR***, `xTaskPreviouslyWoken` is passed as an argument that is initially defined as FALSE. If a task is unblocked during that first call, ***xQueueSendFromISR*** returns TRUE – otherwise, it returns the value that was passed in (as shown at the bottom of Figure 24). Therefore, subsequent calls to ***xQueueSendFromISR*** from within the same ISR must pass in the return value from the previous call to ***xQueueSendFromISR***. This ensures that multiple posts to a queue from a single ISR will only unblock a single task (if one exists).

If no task has been previously woken (unblocked), the algorithm then checks to see if a task is actually waiting to receive data. If so, the task is to be pulled from the event list.

Figure 25 shows the ***xTaskRemoveFromEventList*** function which implements the steps required to remove a task from one of the event lists (either {TasksWaitingToRead} or {TasksWaitingToSend}). This function will only ever be invoked if there are no locks on the queue.

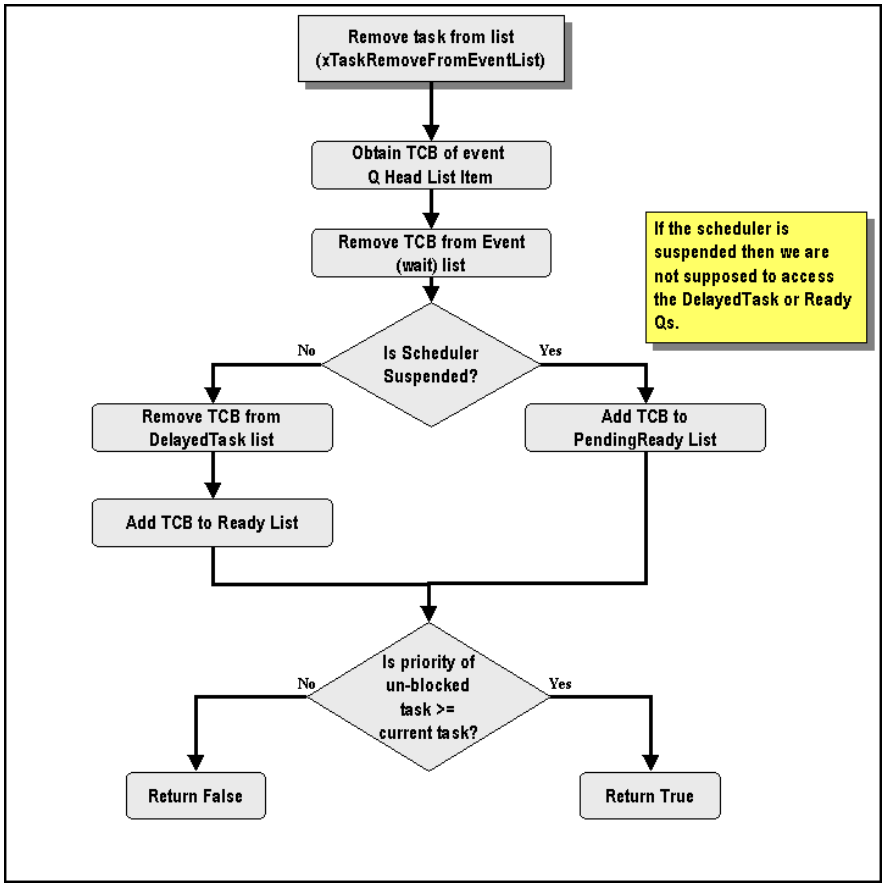


Figure 25: Remove Task From Event List

The ***xTaskRemoveFromEventList*** function removes the first available task from the head of the event list (they are listed in order of descending priority). At this point, it is important to recall that every blocked task will appear on the {DelayedTaskList} whether it was

placed there by a specific delay API call or if it was blocked. As previously described, this ensures that every blocked task has a timeout to prevent deadlock. TCBs are linked into Event and {DelayedTaskList} via the Generic List Item and Event structures in the TCB as shown in Figure 26.

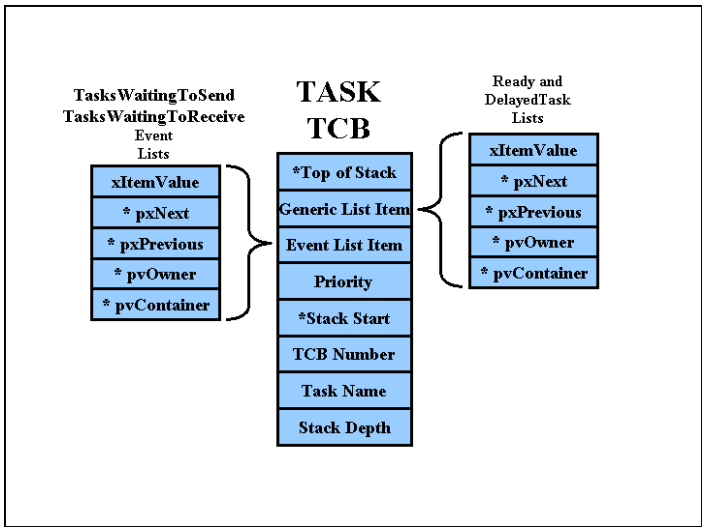


Figure 26: Generic and Event Lists in TCB

If the scheduler is not suspended after removing the TCB from the {EventList}, then the function removes the task from the {DelayedTaskList} and inserts it into the Ready list. If the scheduler *has* been suspended, then there is probably an operation being performed on the Ready or {DelayedTaskList} (or both) so the task is placed in a temporary list called {PendingReadyList}. When the scheduler is reinstated, tasks in this list will be examined and added to the Ready list in batch.

Regardless of the list to which the task is added, ***xTaskRemoveFromEventList*** determines if the task just unblocked has a priority that is equal to or greater than the currently executing task. It provides this information to the calling function as either a TRUE return (priority equal or higher) or a FALSE return (priority not higher). The calling function uses this information to determine if a context switch is needed immediately.

Posting to a Queue from a Schedulable Task

The act of making a post to a queue from within a schedulable task (as distinct from within an ISR) is one of the most interesting aspects of FreeRTOS. Figures 27 and 28 present the algorithm used to perform this operation.

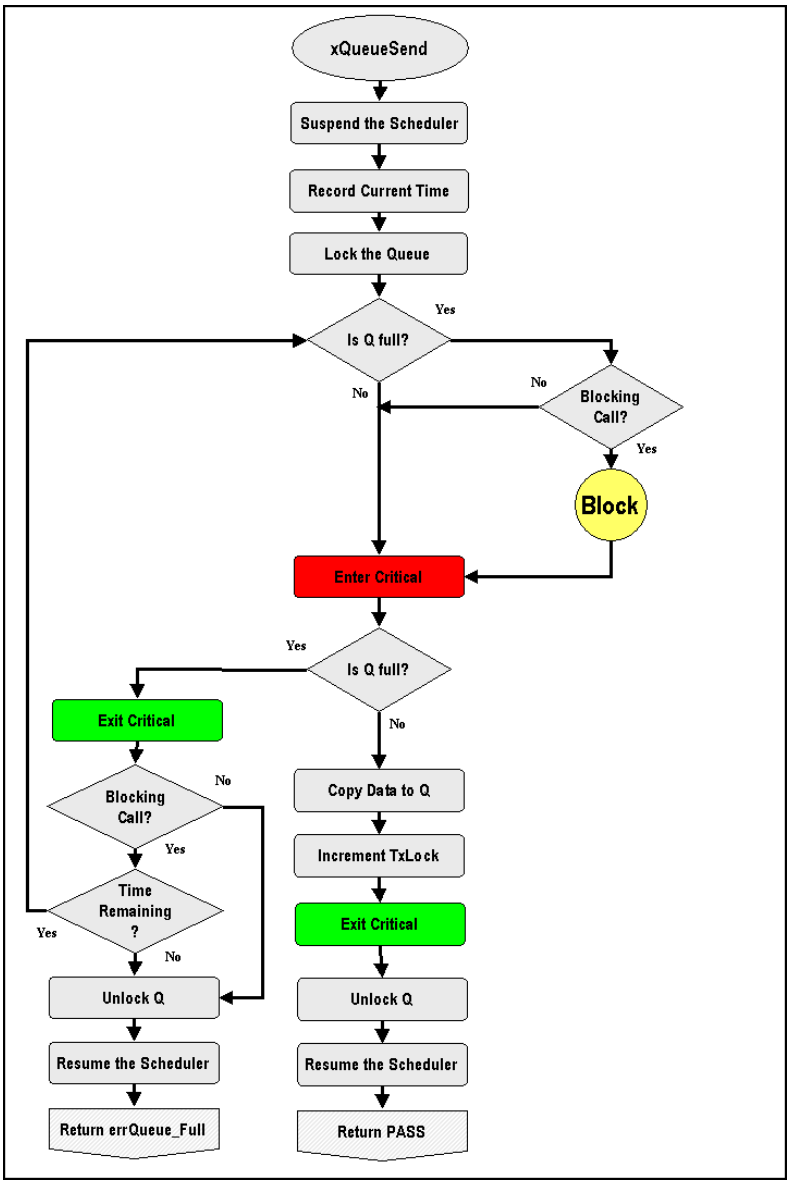


Figure 27: Posting to a Queue From a Task

The function ***xQueueSend*** suspends the scheduler, records the current time, and locks the queue when it is invoked. Recall that locking the queue prevents ISRs from modifying the event list but does not prevent them from posting to the queue. ***xQueueSend*** senses the queue to see if it is full. If it is, and the call was blocking (a non-zero tick time was provided as part of the call), then ***xQueueSend*** blocks. Figure 28 provides greater detail to the blocking process. We will digress slightly to describe that process.

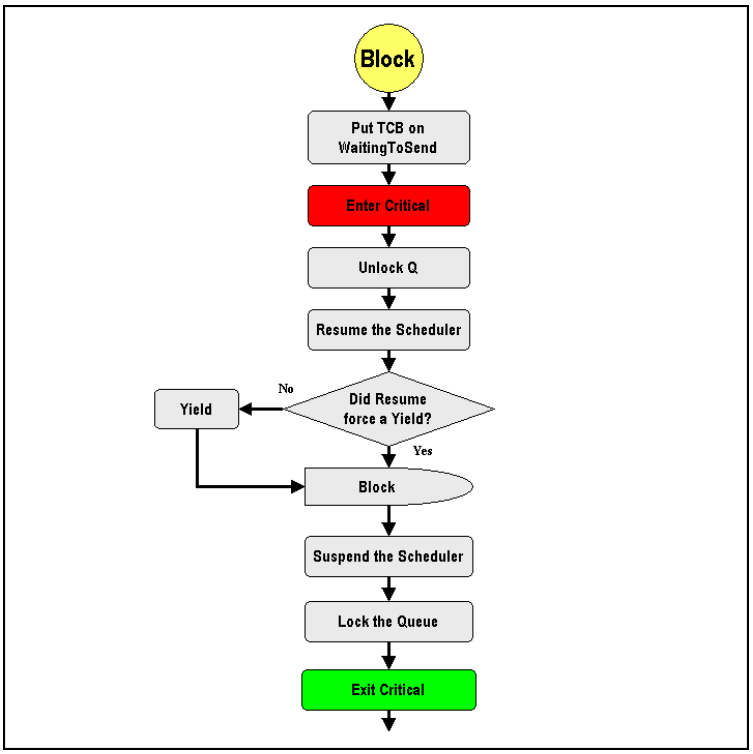


Figure 28: Blocking Call on Queue Blocks

xQueueSend puts the TCB for the calling task onto the {WaitingToSend} list. As detailed in the source code, this operation does not require a mutex on the list because nothing else can modify it while the scheduler is suspended and the queue is locked. Since the intent is to block, the queue must be unlocked and the scheduler resumed so a critical section is entered to prevent anything else from interrupting these operations.

When the code to resume the scheduler is executed, it is possible that the no yield was performed. As described earlier, scheduler suspensions can be nested. If they are, then no yield is performed when a call is made to resume. If that occurs, the algorithm of Figure 28 will force a manual yield. Once the yield is completed, the task that made this attempt to post to the queue is effectively blocked.

Note that a yield from within a critical section does not affect interrupts in other tasks. Unlike the nesting of the scheduler, each task keeps its own nesting depth variable. Interrupts are enabled or disabled on each context switch based on the status of the I bit in the condition code register so no global variable is required to share the nesting status between tasks.

When the task becomes unblocked, the scheduler is suspended, the queue is locked and the critical section is exited whereupon it is immediately re-entered as indicated in Figure 27.

If the queue is not full when the task is resumed, then the requested data is posted and the variable `TxLock` is incremented. This variable tracks whether items were posted or removed from a queue while the queue was locked. It is necessary because event and ready lists cannot be modified while the queue is locked.

A successful post is followed by an exit from the critical section (the scheduler is still suspended) which is then followed by unlocking the queue. When the queue is unlocked, it is necessary to check to see if there are any tasks waiting to receive. Figure 29 shows the algorithm for unlocking the queue.

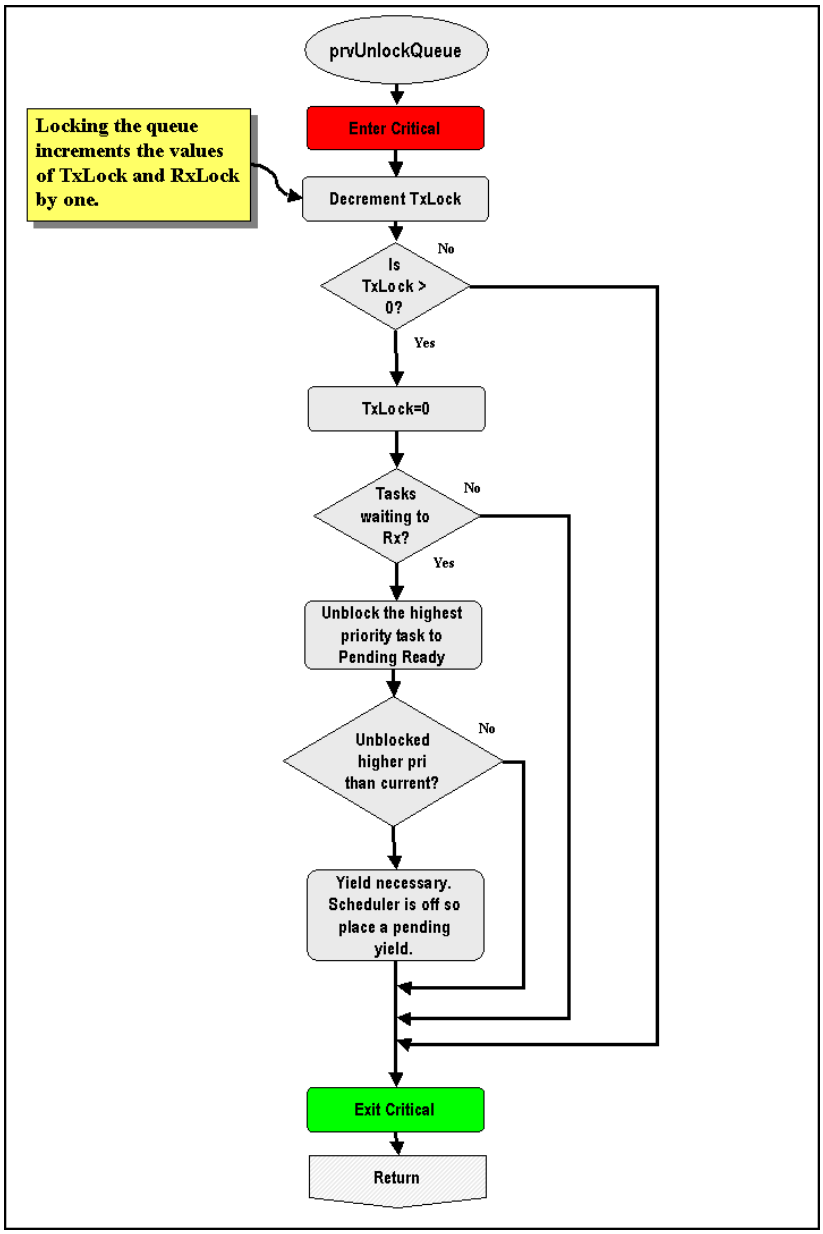


Figure 29: Checking for Blocked Tasks On Queue Unlock

To unlock the queue, a critical section is invoked. TxLock is decremented and checked to see if it is zero. When the queue is locked, TxLock is incremented by one – therefore, other operations on the queue would only have happened if TxLock is greater than one.

If the decremented TxLock is still greater than zero (i.e. something modified the queue), then the waiting lists should be checked to see if a blocked task can be unblocked. TxLock is set to zero. If tasks are waiting, then the highest priority task is taken off the list (the TCB for this task would be the head since tasks are inserted into the list by priority). If the task taken off is higher priority, then it is necessary to yield to that task – however, the scheduler is not running so a pending yield is signaled.

The algorithm shown in Figure 29 has a similar section for RxLock.

Once the queue is unlocked, the scheduler is resumed and QueueSend returns PASS to the calling task.

If the queue was full when the task unblocked (refer to Figure 27), the critical section is exited immediately. If the post request was a blocking post and if the time on the block has not expired and if the queue is full, then the task that made the call is blocked again. The expiry time of the task is determined by adding the block time value to the tick time captured when the function was first invoked. If the current time is less than that value, then the task can block. Otherwise the operation requested has timed out. The queue is unlocked, the scheduler is resumed and the function returns an error condition to the parent task.

Receiving from a Queue – Schedulable Task and ISR

The descriptions provided for posting to a queue from both a schedulable task and from within an ISR have equivalent analogues for receiving from a queue. These operations won't be covered.

Summary and Conclusions

FreeRTOS is a small, nominally real-time operating system for embedded devices. It includes traditional preemptive operating system concepts such as dynamic priority based scheduling and inter-process communication via message queues and synchronization mechanisms.

FreeRTOS provides other features that are intended to allow the operating system to be more flexible to embedded operations. These include cooperative operation (instead of preemptive), co-routines, and the ability to suspend the scheduler. This last feature appears to invoke significant overhead for marginal utility. A scaled version of the FreeRTOS with these features removed might prove to be more attractive to certain communities.

The insistence on timeouts for each blocking task appears to provide a solution to deadlocks that is commensurate with the level of complexity of the operating system. Unfortunately, it pushes the problem upwards since the developer must now pay attention to the problems of assigning and tuning timeouts and dealing with failed access to resources.

Overall, FreeRTOS is a reasonable – if slightly too complex – attempt at a real-time operating system for small embedded targets. The second portion of Attachment 1 to this document should provide the performance analyses in the near future to complete an evaluation of utility.

References

[FRTOS07] Barry, Richard, FreeRTOS, accessed 10 January 07 at: http://www.freertos.org/

[GCC1] GNU Compiler Collection Description of Port (Soft Registers) accessed March 2007 at: http://m68hc11.serveftp.org/m68hc11_gcc.php.

[GNU2.2] GNU Development Chain for M6811/M6812 on Windows, Version 2.2. Accessed at: http://www.gnu.org/software/m68hc11/m68hc11_pkg_zip.html

[S12CPUV2] HC12 Microcontrollers, S12CPUV2 Reference Manual, Rev. 0, 2003.

[SYSC07] SYSC5701 Course Notes, 2007.

[TA] Technological Arts NanoCore12DX.