Understanding FreeRTOS: A Requirement Analysis

Ming-Yuan Zhu

CoreTek Systems, Inc.
1109 CEC Building
6 South Zhongguancun Street
Beijing 100086
People's Republic of China
E-Mail: zhumy@coretek.com.cn

Contents

List of Figures

List of Tables

Chapter 1

Introduction

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.

The target of this report is to provide a survey of **FreeRTOS** including its specifications and implementation according to all of its source code downloaded and the documentation from the reports [Bar07] and [Goy07].

1.1 Synopsis

1.1.1 Features

The following standard features are provided.

- Choice of RTOS scheduling policy
    1. Preemptive: Always runs the highest available task. Tasks of identical priority share CPU time (fully preemptive with round robin time slicing).
    2. Cooperative: Context switches only occur if a task blocks, or explicitly calls `taskYIELD()`.
- Coroutines (light weight tasks that utilize very little RAM).

- Message queues
- Semaphores [via macros]
- Trace visualization ability (requires more RAM)
- Majority of source code common to all supported development tools
- Wide range of ports and examples

Additional features can quickly and easily be added.

1.1.2 Design Philosophy

FreeRTOS is designed to be:

- Simple
- Portable
- Concise

Nearly all the code is written in **C**, with only a few assembler functions where completely unavoidable. This does not result in tightly optimized code, but does mean the code is readable, maintainable and easy to port. If performance were an issue it could easily be improved at the cost of portability. This will not be necessary for most applications.

The RTOS kernel uses multiple priority lists. This provides maximum application design flexibility. Unlike bitmap kernels any number of tasks can share the same priority.

1.2 RTOS Fundamentals

1.2.1 Multitasking

The kernel is the core component within an operating system. Operating systems such as Linux employ kernels that allow users access to the computer seemingly simultaneously. Multiple users can execute multiple programs apparently concurrently.

Each executing program is a task under control of the operating system. If an operating system can execute multiple tasks in this manner it is said to be multitasking.

The use of a multitasking operating system can simplify the design of what would otherwise be a complex software application:

- The multitasking and inter-task communications features of the operating system allow the complex application to be partitioned into a set of smaller and more manageable tasks.

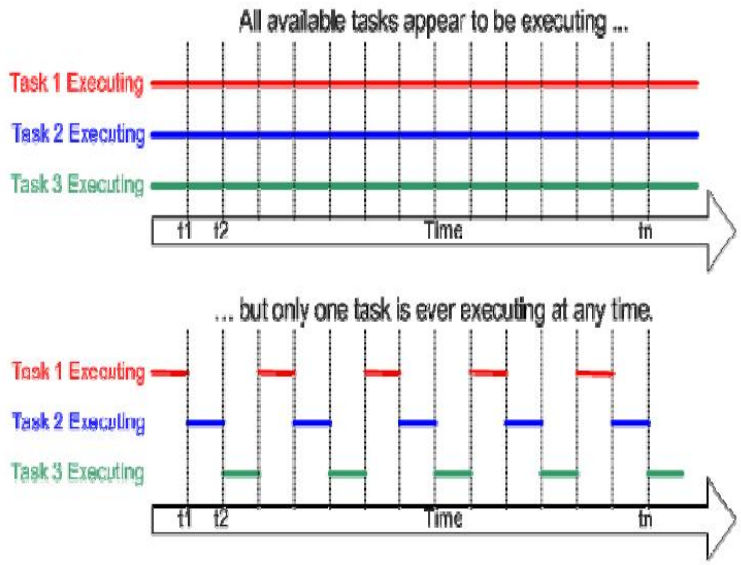

Figure 1.1: Contexts of Task in **FreeRTOS**

- The partitioning can result in easier software testing, work breakdown within teams, and code reuse.
- Complex timing and sequencing details can be removed from the application code and become the responsibility of the operating system.

1.2.2 Multitasking Vs Concurrency

A conventional processor can only execute a single task at a time - but by rapidly switching between tasks a multitasking operating system can make it appear as if each task is executing concurrently. This is depicted by Figure 1.1 which shows the execution pattern of three tasks with respect to time. The task names are color coded and written down the left hand. Time moves from left to right, with the colored lines showing which task is executing at any particular time. The upper diagram demonstrates the perceived concurrent execution pattern, and the lower the actual multitasking execution pattern.

1.2.3 Scheduling

The scheduler is the part of the kernel responsible for deciding which task should be executing at any particular time. The kernel can suspend and later resume a task many times during the task lifetime.

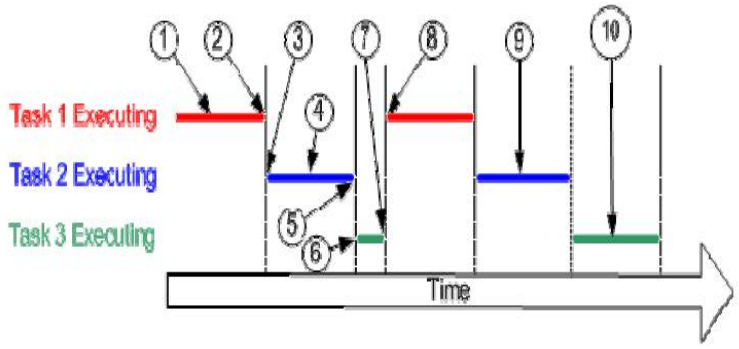

Figure 1.2: The Execution of Task in **FreeRTOS**

The scheduling policy is the algorithm used by the scheduler to decide which task to execute at any point in time. The policy of a (non real time) multi user system will most likely allow each task a "fair" proportion of processor time. The policy used in real time embedded systems is described later.

In addition to being suspended involuntarily by the RTOS kernel a task can choose to suspend itself. It will do this if it either wants to delay (sleep) for a fixed period, or wait (block) for a resource to become available (e.g. a serial port) or an event to occur (eg a key press). A blocked or sleeping task is not able to execute, and will not be allocated any processing time.

Referring to the numbers in Figure 1.2:

- At (1) task 1 is executing.
- At (2) the kernel suspends task 1 ...
- ... and at (3) resumes task 2.
- While task 2 is executing (4), it locks a processor peripheral for it's own exclusive access.
- At (5) the kernel suspends task 2 ...
- ... and at (6) resumes task 3.
- Task 3 tries to access the same processor peripheral, finding it locked task 3 cannot continue so suspends itself at (7).
- At (8) the kernel resumes task 1.
- Etc.
- The next time task 2 is executing (9) it finishes with the processor peripheral and unlocks it.

- The next time task 3 is executing (10) it finds it can now access the processor peripheral and this time executes until suspended by the kernel.

1.2.4 Context Switching

As a task executes it utilizes the processor / microcontroller registers and accesses RAM and ROM just as any other program. These resources together (the processor registers, stack, etc.) comprise the task execution context.

A task is a sequential piece of code - it does not know when it is going to get suspended or resumed by the kernel and does not even know when this has happened. Consider the example of a task being suspended immediately before executing an instruction that sums the values contained within two processor registers.

While the task is suspended other tasks will execute and may modify the processor register values. Upon resumption the task will not know that the processor registers have been altered - if it used the modified values the summation would result in an incorrect value.

To prevent this type of error it is essential that upon resumption a task has a context identical to that immediately prior to its suspension. The operating system kernel is responsible for ensuring this is the case - and does so by saving the context of a task as it is suspended. When the task is resumed its saved context is restored by the operating system kernel prior to its execution. The process of saving the context of a task being suspended and restoring the context of a task being resumed is called context switching as shown in Figure 1.3.

1.2.5 Real Time Applications

Real time operating systems (RTOS's) achieve multitasking using these same principals - but their objectives are very different to those of non real time systems. The different objective is reflected in the scheduling policy. Real time embedded systems are designed to provide a timely response to real world events. Events occurring in the real world can have deadlines before which the real time embedded system must respond and the RTOS scheduling policy must ensure these deadlines are met.

To achieve this objective the software engineer must first assign a priority to each task. The scheduling policy of the RTOS is then to simply ensure that the highest priority task that is able to execute is the task given processing time. This may require sharing processing time "fairly" between tasks of equal priority if they are ready to run simultaneously.

Example:

The most basic example of this is a real time system that incorporates a keypad and LCD. A user must get visual feedback of each key press within a reasonable period - if the user cannot see that the key press has been accepted within this period the software product will at best be awkward to use. If the longest acceptable period was 100ms - any response between 0 and 100ms would be

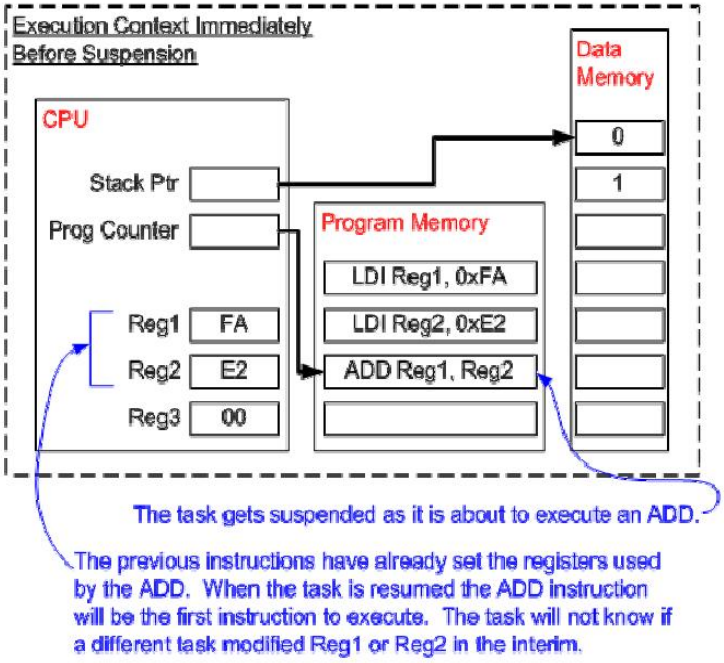

Figure 1.3: The Context Switching of Tasks in **FreeRTOS**

acceptable. This functionality could be implemented as an autonomous task with the following structure:

```
void vKeyHandlerTask(void *pvParameters)
{
  // Key handling is a continuous process and as such the task
  // is implemented using an infinite loop (as most real time
  // tasks are).
  for (;;)
  {
    [Suspend waiting for a key press] [Process the key press]
  }
}
```

Now assume the real time system is also performing a control function that relies on a digitally filtered input. The input must be sampled, filtered and the control cycle executed every 2ms. For correct operation of the filter the temporal regularity of the sample must be accurate to 0.5ms. This functionality could be implemented as an autonomous task with the following structure:

```
void vControlTask(void *pvParameters)
{
  for (;;)
  {
    [Suspend waiting for 2ms since the start of the previous cycle]
    [Sample the input]
    [Filter the sampled input]
    [Perform control algorithm]
    [Output result]
  }
}
```

The software engineer must assign the control task the highest priority as:

1. The deadline for the control task is stricter than that of the key handling task.

2. The consequence of a missed deadline is greater for the control task than for the key handler task.

The next page demonstrates how these tasks would be scheduled by a real time operating system.

1.2.6 Real-Time Scheduling

Figure 1.4 demonstrates how the tasks defined on the previous page would be scheduled by a real time operating system. The RTOS has itself created a task - the idle task - which will execute only when there are no other tasks able to do so. The RTOS idle task is always in a state where it is able to execute.

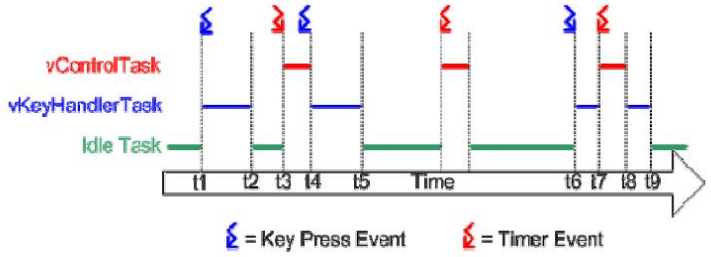

Figure 1.4: Real-Time Scheduling in **FreeRTOS**

Referring to Figure 1.4 above:

- At the start neither of our two tasks are able to run - `vControlTask` is waiting for the correct time to start a new control cycle and `vKeyHandlerTask` is waiting for a key to be pressed. Processor time is given to the RTOS idle task.
- At time `t1`, a key press occurs. `vKeyHandlerTask` is now able to execute - it has a higher priority than the RTOS idle task so is given processor time.
- At time `t2` `vKeyHandlerTask` has completed processing the key and updating the LCD. It cannot continue until another key has been pressed so suspends itself and the RTOS idle task is again resumed.
- At time `t3` a timer event indicates that it is time to perform the next control cycle. `vControlTask` can now execute and as the highest priority task is scheduled processor time immediately.
- Between time `t3` and `t4`, while `vControlTask` is still executing, a key press occurs. `vKeyHandlerTask` is now able to execute, but as it has a lower priority than `vControlTask` it is not scheduled any processor time.
- At `t4` `vControlTask` completes processing the control cycle and cannot restart until the next timer event - it suspends itself. `vKeyHandlerTask` is now the task with the highest priority that is able to run so is scheduled processor time in order to process the previous key press.
- At `t5` the key press has been processed, and `vKeyHandlerTask` suspends itself to wait for the next key event. Again neither of our tasks are able to execute and the RTOS idle task is scheduled processor time.
- Between `t5` and `t6` a timer event is processed, but no further key presses occur.

- The next key press occurs at time `t6`, but before `vKeyHandlerTask` has completed processing the key a timer event occurs. Now both tasks are able to execute. As `vControlTask` has the higher priority `vKeyHandlerTask` is suspended before it has completed processing the key, and `vControlTask` is scheduled processor time.
- At `t8` `vControlTask` completes processing the control cycle and suspends itself to wait for the next. `vKeyHandlerTask` is again the highest priority task that is able to run so is scheduled processor time so the key press processing can be completed.

Chapter 2

Tasks and Coroutines in FreeRTOS

Coroutine functionality is new to **FreeRTOS V4.0.0** and will continue to be developed through subsequent releases. **FreeRTOS V4.0.0** is basically backward compatible with earlier releases so the use of coroutines within an application is completely optional.

The tasks and coroutine documentation pages provide information to allow judgment as to when coroutine use may and may not be appropriate. Below is a brief summary. Note that an application can be designed using just tasks, just coroutines, or a mixture of both - however tasks and coroutines use different API functions and therefore a queue (or semaphore) cannot be used to pass data from a task to a coroutine or visa versa.

2.1 Concepts of Tasks and Coroutines

2.1.1 Characteristics of a 'Task'

`FreeRTOS` versions prior to **V4.0.0** allow a real time application to be structured as a set of autonomous 'tasks' only. This is the traditional model used by an RTOS scheduler.

In brief: A real time application that uses an RTOS can be structured as a set of independent tasks. Each task executes within its own context with no coincidental dependency on other tasks within the system or the scheduler itself. Only one task within the application can be executing at any point in time and the real time scheduler is responsible for deciding which task this should be. The scheduler may therefore repeatedly start and stop each task (swap each task in and out) as the application executes. As a task has no knowledge of the scheduler activity it is the responsibility of the real time scheduler to ensure that the processor context (register values, stack contents, etc) when a task is

swapped in is exactly that as when the same task was swapped out. To achieve this each task is provided with its own stack. When the task is swapped out the execution context is saved to the stack of that task so it can also be exactly restored when the same task is later swapped back in. See the **How FreeRTOS Works** section for more information.

2.1.2 Task Summary

- Simple.
- No restrictions on use.
- Supports full preemption.
- Fully prioritized.
- Each task maintains its own stack resulting in higher RAM usage.
- Re-entrancy must be carefully considered if using preemption.

2.1.3 Characteristics of a 'Coroutine'

FreeRTOS version **V4.0.0** onwards allows a real time application to optionally include coroutines as well as, or instead of, tasks. Coroutines are conceptually similar to tasks but have the following fundamental differences (elaborated further on the coroutine documentation page):

1. `Stack usage`. All the coroutines within an application share a single stack. This greatly reduces the amount of RAM required compared to a similar application written using tasks.
2. `Scheduling and priorities`. Coroutines use prioritized cooperative scheduling with respect to other coroutines, but can be included in an application that uses preemptive tasks.
3. `Macro implementation`. The coroutine implementation is provided through a set of macros.
4. `Restrictions on use`. The reduction in RAM usage comes at the cost of some stringent restrictions in how coroutines can be structured.

2.1.4 Coroutine Summary

- Sharing a stack between coroutines results in much lower RAM usage.
- Cooperative operation makes re-entrancy less of an issue.
- Very portable across architectures.
- Fully prioritized relative to other coroutines, but can always be preempted by tasks if the two are mixed.

- Lack of stack requires special consideration.
- Restrictions on where API calls can be made.
- Co-operative operation only amongst coroutines themselves

2.2 Tasks

2.2.1 Task States

A task can exist in one of the following states:

Running When a task is actually executing it is said to be in the Running state. It is currently utilizing the processor.

Ready Ready tasks are those that are able to execute (they are not blocked or suspended) but are not currently executing because a different task of equal or higher priority is already in the Running state.

Blocked A task is said to be in the Blocked state if it is currently waiting for either a temporal or external event. For example, if a task calls `vTaskDelay()` it will block (be placed into the Blocked state) until the delay period has expired - a temporal event. Tasks can also block waiting for queue and semaphore events. Tasks in the Blocked state always have a 'timeout' period, after which the task will be unblocked. Blocked tasks are not available for scheduling.

Suspended Tasks in the Suspended state are also not available for scheduling. Tasks will only enter or exit the suspended state when explicitly commanded to do so through the `vTaskSuspend()` and `xTaskResume()` API calls respectively. A 'timeout' period cannot be specified.

2.2.2 Task Priorities

Each task is assigned a priority from 0 to (`configMAX_PRIORITIES` - 1). `configMAX_PRIORITIES` is defined within `FreeRTOSConfig.h` and can be set on an application by application basis. The higher the value given to `configMAX_PRIORITIES` the more RAM the **FreeRTOS** kernel will consume.

Low priority numbers denote low priority tasks, with the default idle priority defined by `tskIDLE_PRIORITY` as being zero.

The scheduler will ensure that a task in the ready or running state will always be given processor time in preference to tasks of a lower priority that are also in the ready state. In other words, the task given processing time will always be the highest priority task that is able to run.

2.2.3 Implementing a Task

A task should have the following structure:

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

The type `pdTASK_CODE` is defined as a function that returns void and takes a void pointer as it's only parameter. All functions that implement a task should be of this type. The parameter can be used to pass information of any type into the task - this is demonstrated by several of the standard demo application tasks.

Task functions should never return so are typically implemented as a continuous loop. Again, see the RTOS demo application for numerous examples.

Tasks are created by calling `xTaskCreate()` and deleted by calling `vTaskDelete()`.

2.2.4 Task Creation Macros

Task functions can optionally be defined using the `portTASK_FUNCTION` and `portTASK_FUNCTION_PROTO` macros. These macro are provided to allow compiler specific syntax to be added to the function definition and prototype respectively. Their use is not required unless specifically stated in documentation for the port being used (currently only the PIC18 fedC port). The prototype for the function shown above can be written as:

```
void vATaskFunction(void *pvParameters);
```

Or,

```
portTASK_FUNCTION_PROTO(vATaskFunction, pvParameters);
```

Likewise the function above could equally be written as:

```
portTASK_FUNCTION(vATaskFunction, pvParameters)
{
  for (;;)
  {
    -- Task application code here. --
  }
}
```

2.2.5 The Idle Task

The idle task is created automatically when the scheduler is started.

The idle task is responsible for freeing memory allocated by the RTOS to tasks that have since been deleted. It is therefore important in applications that make use of the `vTaskDelete()` function to ensure the idle task is not starved of processing time. The activity visualization utility can be used to check the micro-controller time allocated to the idle task.

The idle task has no other active functions so can legitimately be starved of micro-controller time under all other conditions. It is possible for application tasks to share the idle task priority. (`tskIDLE_PRIORITY`).

2.2.6 The Idle Task Hook

An idle task hook is a function that is called during each cycle of the idle task. If you want application functionality to run at the idle priority then there are two options:

1. Implement the functionality in an idle task hook. There must always be at least one task that is ready to run. It is therefore imperative that the hook function does not call any API functions that might cause the task to block (`vTaskDelay()` for example. It is permissible for coroutines to block within the hook function).

2. Create an idle priority task to implement the functionality. This is a more flexible solution but has a higher RAM usage overhead.

See the Embedded software application design section for more information on using an idle hook.

To create an idle hook:

1. Set `configUSE_IDLE_HOOK` to 1 within `FreeRTOSConfig.h`.

2. Define a function that has the following prototype:

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

A common use for an idle hook is to simply put the processor into a power saving mode.

1. **To flash an LED** The following code defines a very simple coroutine that does nothing but periodically flashes an LED.

```
void vFlashCoRoutine(xCoRoutineHandle
                     xHandle,
                     unsigned portBASE_TYPE uxIndex)
{
```

```
  // Coroutines must start with a call to crSTART().
  crSTART(xHandle);
  for (;;)
  {
    // Delay for a fixed period.
    crDELAY(xHandle, 10);
    // Flash an LED.
    vParTestToggleLED(0);
  }
  // Coroutines must end with a call to crEND().
  crEND();
}
```

Thats it!

2. **Scheduling the coroutine** Coroutines are scheduled by repeated calls to `vCoRoutineSchedule()`. The best place to do this is from within the idle task by writing an idle task hook. First ensure that `configUSE_IDLE_HOOK` is set to 1 within `FreeRTOSConfig.h`. Then write the idle task hook as:

```
void vApplicationIdleHook(void)
{
  vCoRoutineSchedule(void);
}
```

Alternatively, if the idle task is not performing any other function it would be more efficient to call `vCoRoutineSchedule()` from within a loop as:

```
void vApplicationIdleHook(void)
{
  for (;;)
  {
    vCoRoutineSchedule(void);
  }
}
```

3. **Create the coroutine and start the scheduler** The coroutine can be created within `main()`.

```
#include "task.h" #include "croutine.h"
#define PRIORITY_0 0
void main(void)
{
  // In this case the index is not used and is passed
```

```
  // in as 0.
  xCoRoutineCreate(vFlashCoRoutine, PRIORITY_0, 0);
  // NOTE: Tasks can also be created here!
  // Start the scheduler.
  vTaskStartScheduler();
}
```

4. Extending the example: Using the index parameter Now assume that we want to create 8 such coroutines from the same function. Each coroutine will flash a different LED at a different rate. The index parameter can be used to distinguish between the coroutines from within the coroutine function itself.

   This time we are going to create 8 coroutines and pass in a different index to each.

```
#include "task.h"
#include "croutine.h"
#define PRIORITY_0 0
#define NUM_COROUTINES 8
void main(void)
{
  int i;
  for (i = 0; i < NUM_COROUTINES; i++)
  {
    // This time i is passed in as the index.
    xCoRoutineCreate(vFlashCoRoutine, PRIORITY_0, i);
  }
  // NOTE: Tasks can also be created here!
  // Start the scheduler.
  vTaskStartScheduler();
}
```

   The coroutine function is also extended so each uses a different LED and flash rate.

```
const int
iFlashRates[NUM_COROUTINES] = {10, 20, 30, 40, 50, 60, 70, 80};
const int iLEDToFlash[NUM_COROUTINES] = {0, 1, 2, 3, 4, 5, 6, 7}
void vFlashCoRoutine(xCoRoutineHandle xHandle,
                     unsigned portBASE_TYPE uxIndex)
{
  // Coroutines must start with a call to crSTART().
  crSTART(xHandle);
  for (;;)
  {
```

```
    // Delay for a fixed period. uxIndex is used to index into
    // the iFlashRates. As each coroutine was created with
    // a different index value each will delay for a different
    // period.
    crDELAY(xHandle, iFlashRate[uxIndex]);
    // Flash an LED. Again uxIndex is used as an array index,
    // this time to locate the LED that should be toggled.
    vParTestToggleLED(iLEDToFlash[uxIndex]);
  }
  // Coroutines must end with a call to crEND().
  crEND();
}
```

2.2.7 Demo Application Examples

Two files are included in the download that demonstrate using coroutines with queues:

1. `crflash.c` This is functionally equivalent to the standard demo file `flash.c` but uses coroutines instead of tasks. In addition, and just for demonstration purposes, instead of directly toggling an LED from within a coroutine (as per the quick example above) the number of the LED that should be toggled is passed on a queue to a higher priority coroutine.

2. `crhook.c` Demonstrates passing data from a interrupt to a coroutine. A tick hook function is used as the data source.

The PC and ARM Cortex-M3 demo applications are already pre-configured to use these sample coroutine files and can be used as a reference. All the other demo applications are configured to use tasks only, but can be easily converted to demonstrate coroutines by following the procedure below. This replaces the functionality implemented within `flash.c` with that implemented with `crflash.c`:

1. In `FreeRTOSConfig.h` set `configUSE_CO_ROUTINES` and `configUSE_IDLE_HOOK` to 1.

2. In the IDE project or project `makefile` (depending on the demo project being used):

   (a) Replace the reference to file `FreeRTOS/Demo/Common/Minimal/flash.c` with `FreeRTOS/Demo/Common/Minimal/crflash.c`.

   (b) Add the file `FreeRTOS/Source/croutine.c` to the build.

3. In `main.c`:

   (a) Include the header file `croutine.h` which contains the coroutine macros and function prototypes.

(b) Replace the inclusion of `flash.h` with `crflash.h`.

(c) Remove the call to the function that creates the flash tasks `vStartLEDFlashTasks()` ....

(d) ... and replace it with the function that creates the flash coroutines `vStartFlashCoRoutines(n)`, where n is the number of coroutines that should be created. Each coroutine flashes a different LED at a different rate.

(e) Add an idle hook function that schedules the coroutines as:

```
void vApplicationIdleHook(void)
{
  vCoRoutineSchedule(void);
}
```

If `main()` already contains an idle hook then simply add a call to `vCoRoutineSchedule()` to the existing hook function.

4. Replacing the flash tasks with the flash coroutines means there are at least two less stacks that need allocating and less heap space can therefore be set aside for use by the scheduler. If your project has insufficient RAM to include `croutine.c` in the build then simply reduce the definition of `portTOTAL_HEAP_SPACE` by `(2 * portMINIMAL_STACK_SIZE)` within `FreeRTOSConfig.h`.

2.3 RTOS Kernel Utilities

2.3.1 Queue Implementation

Items are placed in a queue by copy - not by reference. It is therefore preferable, when queuing large items, to only queue a pointer to the item. RTOS demo application files `blockq.c` and `pollq.c` demonstrate queue usage. The queue implementation used by the RTOS is also available for application code. Tasks and coroutines can block on a queue to wait for either data to become available on the queue, or space to become available to write to the queue.

2.3.2 Semaphore Implementation

Binary semaphore functionality is provided by a set of macros. The macros use the queue implementation as this provides everything necessary with no extra code or testing overhead. The macros can easily be extended to provide counting semaphores if required. The RTOS demo application file `semtest.c` demonstrates semaphore usage. Also see the RTOS API documentation.

2.3.3 Tick Hook Function

A tick hook function is a function that executes during each RTOS tick interrupt. It can be used to optionally execute application code during each tick ISR. To use a tick hook function `configUSE_TICK_HOOK` must be set to 1 within `FreeRTOSConfig.h`. The prototype for the tick hook function is:

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

`vApplicationTickHook()` executes from within an ISR so must be short and never make a blocking API call. See the demo application file `crhook.c` for an example of how to use a tick hook. It is also possible to include an idle task hook.

2.4 Trace Utility

The trace visualization utility allows the RTOS activity to be examined.

It records the sequence in which tasks are given micro-controller processing time.

To use the utility the macro `configUSE_TRACE_FACILITY` must be defined as 1 within `FreeRTOSConfig.h` when the application is compiled. See the configuration section in the RTOS API documentation for more information.

The trace is started by calling `vTaskStartTrace()` and ended by calling `ulTaskEndTrace()`. It will end automatically if it's buffer becomes full.

The completed trace buffer can be stored to disk for offline examination. The `DOS/Windows` utility `tracecon.exe` converts the stored buffer to a tab delimited text file. This can then be opened and examined in a spread sheet application.

Below is a 10 millisecond example output collected from the AMD 186 demo application. The x axis shows the passing of time, and the y axis the number of the task that is running.

Each task is automatically allocated a number when it is created. The idle task is always number 0. `vTaskList()` can be used to obtain the number allocated to each task, along with some other useful information. The information returned by `vTaskList()` during the demo application is shown below, where:

- Name - is the name given to the task when it was created. Note that the demo application creates more than one instance of some tasks.
- State - shows the state of a task. This can be either 'B'locked, 'R'eady, 'S'uspended or 'D'eleted.
- Priority - is the priority given to the task when it was created.
- Stack - shows the high water mark of the task stack. This is the minimum amount of free stack that has been available during the lifetime of the task.

- Num - is the number automatically allocated to the task.

Note: In it's current implementation, the time resolution of the trace is equal to the tick rate. Context switches can occur more frequently than the system tick (if a task blocks for example). When this occurs the trace will show that a context switch has occurred and will accurately shows the context switch sequencing.

However, the timing of context switches that occur between system ticks cannot accurately be recorded. The ports could easily be modified to provide a higher resolution time stamp by making use of a free running timer.

Chapter 3

Application Programming Interfaces of FreeRTOS

3.1 General Information

- Only those API functions specifically designated for use from within an ISR should be used from within an ISR.
- Tasks and coroutines use different API functions to access queues. A queue cannot be used to communicate between a task and a coroutine or visa versa.
- Intertask communication can be achieved using both the full featured API functions and the light weight API functions (those with "`FromISR`" in their name). Use of the light weight functions requires special consideration, as described under the heading "Performance tips and tricks - using the light weight API".

3.1.1 Task API

A task may call any API function listed in the menu frame on the left other than those under the coroutine specific section.

3.1.2 Coroutine API

In addition to the API functions listed under the coroutine specific section, a coroutine may use the following API calls.

- `taskYIELD()` - will yield the task in which the coroutines are running.
- `taskENTER_CRITICAL()`.
- `taskEXIT_CRITICAL()`.

- `vTaskStartScheduler()` - this is still used to start the scheduler even if the application only includes coroutines and no tasks.
- `vTaskSuspendAll()` - can still be used to lock the scheduler.
- `xTaskResumeAll()`
- `xTaskGetTickCount()`
- `uxTaskGetNumberOfTasks()`

3.2 Configuration Customization

A number of configurable parameters exist that allow the **FreeRTOS** kernel to be tailored to your particular application. These items are located in a file called `FreeRTOSConfig.h`.

Each demo application included in the **FreeRTOS** source code download has its own `FreeRTOSConfig.h` file. Here is a typical example, followed by an explanation of each parameter:

```
#ifndef FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H

/*
   Here is a good place to include header files that are required
   across your application.
*/

#include "something.h"
#define configUSE_PREEMPTION 1
#define configUSE_IDLE_HOOK 0
#define configUSE_TICK_HOOK 0
#define configCPU_CLOCK_HZ 58982400
#define configTICK_RATE_HZ 250
#define configMAX_PRIORITIES 5
#define configMINIMAL_STACK_SIZE 128
#define configTOTAL_HEAP_SIZE 10240
#define configMAX_TASK_NAME_LEN 16
#define configUSE_TRACE_FACILITY 0
#define configUSE_16_BIT_TICKS 0
#define configIDLE_SHOULD_YIELD 1
#define configUSE_MUTEXES 0

#define INCLUDE_vTaskPrioritySet 1
#define INCLUDE_uxTaskPriorityGet 1
#define INCLUDE_vTaskDelete 1
#define INCLUDE_vTaskCleanUpResources 0
#define INCLUDE_vTaskSuspend 1
#define INCLUDE_vResumeFromISR 1
```

```
#define INCLUDE_vTaskDelayUntil 1
#define INCLUDE_vTaskDelay 1
#define INCLUDE_xTaskGetSchedulerState 1
#define INCLUDE_xTaskGetCurrentTaskHandle 1

#define configUSE_CO_ROUTINES 0
#define configMAX_CO_ROUTINE_PRIORITIES 1

#define configKERNEL_INTERRUPT_PRIORITY [dependent of processor]

#endif /* FREERTOS_CONFIG_H */
```

3.2.1 'config' Parameters

3.2.1.1 configUSE_PREEMPTION

Set to 1 to use the preemptive kernel, or 0 to use the cooperative kernel.

3.2.1.2 configUSE_IDLE_HOOK

Set to 1 if you wish to use an idle hook, or 0 to omit an idle hook.

3.2.1.3 configUSE_TICK_HOOK

Set to 1 if you wish to use an tick hook, or 0 to omit an tick hook.

3.2.1.4 configCPU_CLOCK_HZ

Enter the frequency in Hz at which the internal processor core will be executing. This value is required in order to correctly configure timer peripherals.

3.2.1.5 configTICK_RATE_HZ

The frequency of the RTOS tick interrupt.

The tick interrupt is used to measure time. Therefore a higher tick frequency means time can be measured to a higher resolution. However, a high tick frequency also means that the kernel will use more CPU time so be less efficient. The RTOS demo applications all use a tick rate of 1000Hz. This is used to test the kernel and is higher than would normally be required.

More than one task can share the same priority. The kernel will share processor time between tasks of the same priority by switching between the tasks during each RTOS tick. A high tick rate frequency will therefore also have the effect of reducing the 'time slice' given to each task.

3.2.1.6 `configMAX_PRIORITIES`

The number of priorities available to the application tasks. Any number of tasks can share the same priority. Co-routines are prioritised separately - see `configMAX_CO_ROUTINE_PRIORITIES`.

Each available priority consumes RAM within the kernel so this value should not be set any higher than actually required by your application.

3.2.1.7 `configMINIMAL_STACK_SIZE`

The size of the stack used by the idle task. Generally this should not be reduced from the value set in the `FreeRTOSConfig.h` file provided with the demo application for the port you are using.

3.2.1.8 `configTOTAL_HEAP_SIZE`

The total amount of RAM available to the kernel. This value will only be used if your application makes use of one of the sample memory allocation schemes provided in the **FreeRTOS** source code download. See the memory configuration section for further details.

3.2.1.9 `configMAX_TASK_NAME_LEN`

The maximum permissible length of the descriptive name given to a task when the task is created. The length is specified in the number of characters including the NULL termination byte.

3.2.1.10 `configUSE_TRACE_FACILITY`

Set to 1 if you wish the trace visualisation functionality to be available, or 0 if the trace functionality is not going to be used. If you use the trace functionality a trace buffer must also be provided.

3.2.1.11 `configUSE_16_BIT_TICKS`

Time is measured in 'ticks' - which is the number of times the tick interrupt has executed since the kernel was started. The tick count is held in a variable of type `portTickType`.

Defining `configUSE_16_BIT_TICKS` as 1 causes `portTickType` to be defined (`typedef`'ed) as an unsigned 16 bit type. Defining `configUSE_16_BIT_TICKS` as 0 causes `portT`.

Using a 16 bit type will greatly improve performance on 8 and 16 bit architectures, but limits the maximum specifiable time period to 65535 'ticks'. Therefore, assuming a tick frequency of 250Hz, the maximum time a task can delay or block when a 16bit counter is used is 262 seconds, compared to 17179869 seconds when using a 32bit counter.

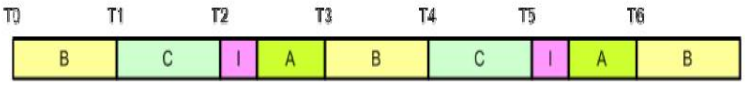

Figure 3.1: The Execution Pattern of Four Tasks at the Idle Priority

3.2.1.12 `configIDLE_SHOULD_YIELD`

This parameter controls the behaviour of tasks at the idle priority. It only has an effect if:

1. The preemptive scheduler is being used.
2. The users application creates tasks that run at the idle priority.

Tasks that share the same priority will time slice. Assuming none of the tasks get preempted, it might be assumed that each task of at a given priority will be allocated an equal amount of processing time - and if the shared priority is above the idle priority then this is indeed the case.

When tasks share the idle priority the behavior can be slightly different. When `configIDLE_SHOULD_YIELD` is set to 1 the idle task will yield immediately should any other task at the idle priority be ready to run. This ensures the minimum amount of time is spent in the idle task when application tasks are available for scheduling. This behavior can however have undesirable effects (depending on the needs of your application) as depicted below:

Figure 3.1 shows the execution pattern of four tasks at the idle priority. Tasks A, B and C are application tasks. Task I is the idle task. A context switch occurs with regular period at times $\mathtt{T}_0, \mathtt{T}_1, \ldots, \mathtt{T}_6$. When the idle task yields task A starts to execute - but the idle task has already taken up some of the current time slice. This results in task I and task A effectively sharing a time slice. The application tasks B and C therefore get more processing time than the application task A.

This situation can be avoided by:

- If appropriate, using an idle hook in place of separate tasks at the idle priority.
- Creating all application tasks at a priority greater than the idle priority.
- Setting `configIDLE_SHOULD_YIELD` to 0.

Setting `configIDLE_SHOULD_YIELD` prevents the idle task from yielding processing time until the end of its time slice. This ensure all tasks at the idle priority are allocated an equal amount of processing time - but at the cost of a greater proportion of the total processing time being allocated to the idle task.

3.2.1.13 `configUSE_USE_MUTEXES`

Set to 1 to include mutex functionality in the build, or 0 to omit mutex functionality from the build. Readers should familiarise themselves with the differences between mutexes and binary semaphores in relation to the `FreeRTOS.org` functionality.

3.2.1.14 `configUSE_CO_ROUTINES`

Set to 1 to include coroutine functionality in the build, or 0 to omit coroutine functionality from the build. To include coroutines `croutine.c` must be included in the project.

3.2.1.15 `configMAX_CO_ROUTINE_PRIORITIES`

The number of priorities available to the application coroutines. Any number of coroutines can share the same priority. Tasks are prioritised separately - see `configMAX_PRIORITIES`.

3.2.1.16 `configKERNEL_INTERRUPT_PRIORITY`

Currently only in Cortex-M3/IAR, PIC24 and dsPIC ports. Other ports will get upgraded shortly.

This sets the interrupt priority used by the kernel. The kernel should use a low interrupt priority, allowing higher priority interrupts to be unaffected by the kernel entering critical sections. Instead of critical sections globally disabling interrupts, they only disable interrupts that are below the kernel interrupt priority.

This permits very flexible interrupt handling:

- At the kernel priority level interrupt handling 'tasks' can be written and prioritised as per any other task in the system. These are tasks that are woken by an interrupt. The interrupt service routine (ISR) itself should be written to be as short as it possibly can be - it just grabs the data then wakes the high priority handler task. The ISR then returns directly into the woken handler task - so interrupt processing is contiguous in time just as if it were all done in the ISR itself. The benefit of this is that all interrupts remain enabled while the handler task executes.

- ISR's running above the kernel priority are never masked out by the kernel itself, so their responsiveness is not effected by the kernel functionality. However, such ISR's cannot use the **FreeRTOS** API functions.

To utilize this scheme your application design must adhere to the following rule:

Any interrupt that uses the `FreeRTOS.org` API must be set to the same priority as the kernel (as configured by the `configKERNEL_INTERRUPT_PRIORITY` macro).

3.2.2 `INCLUDE` Parameters

The macros starting '`INCLUDE`' allow those components of the real time kernel not utilized by your application to be excluded from your build. This ensures the RTOS does not use any more ROM or RAM than necessary for your particular embedded application.

Each macro takes the form ...

```
INCLUDE_FunctionName ...
```

where `FunctionName` indicates the API function (or set of functions) that can optionally be excluded. To include the API function set the macro to 1, to exclude the function set the macro to 0. For example, to include the `vTaskDelete()` API function use:

```
#define INCLUDE_vTaskDelete 1
```

To exclude `vTaskDelete()` from your build use:

```
#define INCLUDE_vTaskDelete 0
```

3.3 Memory Management

The RTOS kernel has to allocate RAM each time a task, queue or semaphore is created. The `malloc()` and `free()` functions can sometimes be used for this purpose, but ...

1. they are not always available on embedded systems,
2. take up valuable code space,
3. are not thread safe, and
4. are not deterministic (the amount of time taken to execute the function will differ from call to call)

... so more often than not an alternative scheme is required.

One embedded/real time system can have very different RAM and timing requirements to another - so a single RAM allocation algorithm will only ever be appropriate for a subset of applications.

To get around this problem the memory allocation API is included in the RTOS portable layer - where an application specific implementation appropriate for the real time system being developed can be provided. When the real time kernel requires RAM, instead of calling `malloc()` it makes a call to `pvPortMalloc()`. When RAM is being freed, instead of calling `free()` the real time kernel makes a call to `vPortFree()`.

3.3.1 Schemes Included in the Source Code Download

Three sample RAM allocation schemes are included in the **FreeRTOS** source code download (**V2.5.0** onwards). These are used by the various demo applications as appropriate. The following subsections describe the available schemes, when they should be used, and highlight the demo applications that demonstrate their use.

Each scheme is contained in a separate source file (`heap_1.c`, `heap_2.c` and `heap_3.c` respectively) which can be located in the `Source/Portable/MemMang` directory. Other schemes can be added if required.

3.3.1.1 Scheme 1 - `heap_1.c`

This is the simplest scheme of all. It does not permit memory to be freed once it has been allocated, but despite this is suitable for a surprisingly large number of applications.

The algorithm simply subdivides a single array into smaller blocks as requests for RAM are made. The total size of the array is set by the definition `configTOTAL_HEAP_SIZE` - which is defined in `FreeRTOSConfig.h`.

This scheme:

- Can be used if your application never deletes a task or queue (no calls to `vTaskDelete()` or `vQueueDelete()` are ever made).
- Is always deterministic (always takes the same amount of time to return a block).
- Is used by the PIC, AVR and 8051 demo applications - as these do not dynamically create or delete tasks after `vTaskStartScheduler()` has been called.

`heap_1.c` is suitable for a lot of small real time systems provided that all tasks and queues are created before the kernel is started.

3.3.1.2 Scheme 2 - `heap_2.c`

This scheme uses a best fit algorithm and, unlike scheme 1, allows previously allocated blocks to be freed. It does not however combine adjacent free blocks into a single large block.

Again the total amount of available RAM is set by the definition `configTOTAL_HEAP_SIZE` - which is defined in `FreeRTOSConfig.h`.

This scheme:

- Can be used even when the application repeatedly calls `vTaskCreate()`/`vTaskDelete()` or `vQueueCreate()`/`vQueueDelete()` (causing multiple calls to `pvPortMalloc()` and `vPortFree()`).
- Should not be used if the memory being allocated and freed is of a random size - this would only be the case if tasks being deleted each had a different stack depth, or queues being deleted were of different lengths.

- Could possible result in memory fragmentation problems should your application create blocks of queues and tasks in an unpredictable order. This would be unlikely for nearly all applications but should be kept in mind.
- Is not deterministic - but is also not particularly inefficient.
- Is used by the ARM7, and `Flashlite` demo applications - as these dynamically create and delete tasks.

`heap_2.c` is suitable for most small real time systems that have to dynamically create tasks.

3.3.1.3 Scheme 3 - `heap_3.c`

This is just a wrapper for the standard `malloc()` and `free()` functions. It makes them thread safe.

This scheme:

- Requires the linker to setup a heap, and the compiler library to provide `malloc()` and `free()` implementations.
- Is not deterministic.
- Will probably considerably increase the kernel code size.
- Is used by the PC (x86 single board computer) demo application.

3.4 Task Management

3.4.1 Task Management

3.4.1.1 `xTaskHandle`

Type by which tasks are referenced. For example, a call to `xTaskCreate` returns (via a pointer parameter) an `xTaskHandle` variable that can then be used as a parameter to `vTaskDelete` to delete the task.

3.4.1.2 `xTaskCreate`

Prototype

```
portBASE_TYPE xTaskCreate(pdTASK_CODE pvTaskCode,
                          const portCHAR * const pcName,
                          unsigned portSHORT usStackDepth,
                          void *pvParameters,
                          unsigned portBASE_TYPE uxPriority,
                          xTaskHandle *pvCreatedTask);
```

Semantics

Create a new task and add it to the list of tasks that are ready to run.

Parameters

`pvTaskCode` Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).

`pcName` A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by `configMAX_TASK_NAME_LEN`.

`usStackDepth` The size of the task stack specified as the number of variables the stack can hold - not the number of bytes. For example, if the stack is 16 bits wide and `usStackDepth` is defined as 100, 200 bytes will be allocated for stack storage. The stack depth multiplied by the stack width must not exceed the maximum value that can be contained in a variable of type size_t.

`pvParameters` Pointer that will be used as the parameter for the task being created.

`uxPriority` The priority at which the task should run.

`pvCreatedTask` Used to pass back a handle by which the created task can be referenced.

Returns

`pdPASS` if the task was successfully created and added to a ready list, otherwise an error code defined in the file `projdefs.h`

Example Usage

```
// Task to be created.
void vTaskCode(void * pvParameters)
{
  for (;;)
  {
    // Task code goes here.
  }
}

// Function that creates a task.
void vOtherFunction(void)
{
  unsigned char ucParameterToPass;
  xTaskHandle xHandle;
  // Create the task, storing the handle.
  xTaskCreate(vTaskCode,
```

```
              "NAME",
              STACK_SIZE,
              &ucParameterToPass,
              tskIDLE_PRIORITY,
              &xHandle);

  // Use the handle to delete the task.
  vTaskDelete(xHandle);
}
```

3.4.1.3 vTaskDelete

Prototype

```
void vTaskDelete(xTaskHandle pxTask);
```

INCLUDE_vTaskDelete must be defined as 1 for this function to be available. See the configuration section for more information.

Semantics

Remove a task from the RTOS real time kernels management. The task being deleted will be removed from all ready, blocked, suspended and event lists.

NOTE: The idle task is responsible for freeing the kernel allocated memory from tasks that have been deleted. It is therefore important that the idle task is not starved of microcontroller processing time if your application makes any calls to vTaskDelete(). Memory allocated by the task code is not automatically freed, and should be freed before the task is deleted.

See the demo application file death.c for sample code that utilises vTaskDelete().

Parameters

pxTask The handle of the task to be deleted. Passing NULL will cause the calling task to be deleted.

Example Usage

```
void vOtherFunction(void)
{
  xTaskHandle xHandle;
  // Create the task, storing the handle.
  xTaskCreate(vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle);
  // Use the handle to delete the task.
  vTaskDelete(xHandle);
}
```

3.4.2 Task Control

3.4.2.1 `vTaskDelay`

Prototype

```
void vTaskDelay(portTickType xTicksToDelay);
```

`INCLUDE_vTaskDelay` must be defined as 1 for this function to be available. See the configuration section for more information.

Semantics

Delay a task for a given number of ticks. The actual time that the task remains blocked depends on the tick rate. The constant portTICK_RATE_MS can be used to calculate real time from the tick rate - with the resolution of one tick period.

Parameters

`xTicksToDelay` The amount of time, in tick periods, that the calling task should block.

Example Usage

```
// Perform an action every 10 ticks.
// NOTE:
// This is for demonstration only and would be better achieved
// using vTaskDelayUntil().
void vTaskFunction(void * pvParameters)
{
  portTickType xDelay,
  xNextTime;
  // Calc the time at which we want to perform the action
  // next.
  xNextTime = xTaskGetTickCount() + (portTickType) 10;
  for (;;)
  {
    xDelay = xNextTime - xTaskGetTickCount();
    xNextTime += (portTickType) 10;
    // Guard against overflow
    if (xDelay <= (portTickType) 10)
    {
      vTaskDelay(xDelay);
    }
    // Perform action here.
```

```
  }
}
```

3.4.2.2 `vTaskDelayUntil`

Prototype

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

`INCLUDE_vTaskDelayUntil` must be defined as 1 for this function to be available. See the configuration section for more information.

Semantics

Delay a task until a specified time. This function can be used by cyclical tasks to ensure a constant execution frequency.

This function differs from `vTaskDelay()` in one important aspect: `vTaskDelay()` specifies a time at which the task wishes to unblock relative to the time at which `vTaskDelay()` is called, whereas `vTaskDelayUntil()` specifies an absolute time at which the task wishes to unblock.

`vTaskDelay()` will cause a task to block for the specified number of ticks from the time `vTaskDelay()` is called. It is therefore difficult to use `vTaskDelay()` by itself to generate a fixed execution frequency as the time between a task unblocking following a call to `vTaskDelay()` and that task next calling `vTaskDelay()` may not be fixed [the task may take a different path though the code between calls, or may get interrupted or preempted a different number of times each time it executes].

Whereas `vTaskDelay()` specifies a wake time relative to the time at which the function is called, `vTaskDelayUntil()` specifies the absolute (exact) time at which it wishes to unblock.

It should be noted that `vTaskDelayUntil()` will return immediately (without blocking) if it is used to specify a wake time that is already in the past. Therefore a task using `vTaskDelayUntil()` to execute periodically will have to re-calculate its required wake time if the periodic execution is halted for any reason (for example, the task is temporarily placed into the Suspended state) causing the task to miss one or more periodic executions. This can be detected by checking the variable passed by reference as the `pxPreviousWakeTime` parameter against the current tick count. This is however not necessary under most usage scenarios.

The constant `configTICK_RATE_MS` can be used to calculate real time from the tick rate - with the resolution of one tick period.

Parameters

`pxPreviousWakeTime` Pointer to a variable that holds the time at which the task was last unblocked. The variable must be initialised with the current time prior to its first use (see the example below). Following this the variable is automatically updated within `vTaskDelayUntil()`.

`xTimeIncrement` The cycle time period. The task will be unblocked at time (`*pxPreviousWakeTime + xTimeIncrement`). Calling `vTaskDelayUntil` with the same `xTimeIncrement` parameter value will cause the task to execute with a fixed interval period.

Example Usage

```
// Perform an action every 10 ticks.
void vTaskFunction(void * pvParameters)
{
  portTickType xLastWakeTime;
  const portTickType xFrequency = 10;
  // Initialise the xLastWakeTime variable with the current time.
  xLastWakeTime = xTaskGetTickCount();
  for (;;)
  {
    // Wait for the next cycle.
    vTaskDelayUntil(&xLastWakeTime, xFrequency);
    // Perform action here.
  }
}
```

3.4.2.3 `uxTaskPriorityGet`

Prototype

```
unsigned portBASE_TYPE uxTaskPriorityGet(xTaskHandle pxTask);
```

`INCLUDE_vTaskPriorityGet` must be defined as 1 for this function to be available. See the configuration section for more information.

Semantics

Obtain the priority of any task.

Parameters

`pxTask` Handle of the task to be queried. Passing a NULL handle results in the priority of the calling task being returned.

Returns

The priority of `pxTask`.

Example Usage

```
void vAFunction(void)
{
  xTaskHandle xHandle;
  // Create a task, storing the handle.
  xTaskCreate(vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle);
  // ...
  // Use the handle to obtain the priority of the created task.
  // It was created with tskIDLE_PRIORITY, but may have changed
  // it itself.
  if (uxTaskPriorityGet(xHandle) != tskIDLE_PRIORITY)
  {
    // The task has changed it's priority.
  }
  // ...
  // Is our priority higher than the created task?
  if (uxTaskPriorityGet(xHandle) < uxTaskPriorityGet(NULL))
  {
    // Our priority (obtained using NULL handle) is higher.
  }
}
```

3.4.2.4 `vTaskPrioritySet`

Prototype

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

`INCLUDE_vTaskPrioritySet` must be defined as 1 for this function to be available. See the configuration section for more information.

Semantics

Set the priority of any task. A context switch will occur before the function returns if the priority being set is higher than the currently executing task.

Parameters

`pxTask` Handle to the task for which the priority is being set. Passing a NULL handle results in the priority of the calling task being set.

`uxNewPriority` The priority to which the task will be set.

Example Usage

```
void vAFunction(void)
{
  xTaskHandle xHandle;
  // Create a task, storing the handle.
  xTaskCreate(vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle);
  // ...
  // Use the handle to raise the priority of the created task.
  vTaskPrioritySet(xHandle, tskIDLE_PRIORITY + 1);
  // ...
  // Use a NULL handle to raise our priority to the same value.
  vTaskPrioritySet(NULL, tskIDLE_PRIORITY + 1);
}
```

3.4.2.5 `vTaskSuspend`

Prototype

```
void vTaskSuspend(xTaskHandle pxTaskToSuspend);
```

`INCLUDE_vTaskSuspend` must be defined as 1 for this function to be available. See the configuration section for more information.

Semantics

Suspend any task. When suspended a task will never get any microcontroller processing time, no matter what its priority. Calls to `vTaskSuspend` are not accumulative - i.e. calling `vTaskSuspend()` twice on the same task still only requires one call to `vTaskResume()` to ready the suspended task.

Parameters

`pxTaskToSuspend` Handle to the task being suspended. Passing a NULL handle will cause the calling task to be suspended.

Example Usage

```
void vAFunction(void)
{
  xTaskHandle xHandle;
  // Create a task, storing the handle.
  xTaskCreate(vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle);
  // ...
  // Use the handle to suspend the created task.
  vTaskSuspend(xHandle);
  // ...
```

```
  // The created task will not run during this period, unless
  // another task calls vTaskResume(xHandle).
  //...
  // Suspend ourselves.
  vTaskSuspend(NULL);
  // We cannot get here unless another task calls vTaskResume
  // with our handle as the parameter.
}
```

3.4.2.6 vTaskResume

Prototype

```
void vTaskResume(xTaskHandle pxTaskToResume);
```

`INCLUDE_vTaskSuspend` must be defined as 1 for this function to be available. See the configuration section for more information.

Semantics

Resumes a suspended task. A task that has been suspended by one of more calls to `vTaskSuspend()` will be made available for running again by a single call to `vTaskResume()`.

Parameters

`pxTaskToResume` Handle to the task being readied.

Example Usage

```
void vAFunction(void)
{
  xTaskHandle xHandle;
  // Create a task, storing the handle.
  xTaskCreate(vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle);
  // ...
  // Use the handle to suspend the created task.
  vTaskSuspend(xHandle);
  // ...
  // The created task will not run during this period, unless
  // another task calls vTaskResume(xHandle).
  //...
  // Resume the suspended task ourselves.
  vTaskResume(xHandle);
  // The created task will once again get microcontroller processing
  // time in accordance with it priority within the system.
}
```

3.4.2.7 `vTaskResumeFromISR`

Prototype

```
portBASE_TYPE vTaskResumeFromISR(xTaskHandle pxTaskToResume);
```

`INCLUDE_vTaskSuspend` and `INCLUDE_xTaskResumeFromISR` must be defined as 1 for this function to be available. See the configuration section for more information.

Semantics

A function to resume a suspended task that can be called from within an ISR.

A task that has been suspended by one of more calls to `vTaskSuspend()` will be made available for running again by a single call to `xTaskResumeFromISR()`.

`vTaskResumeFromISR()` should not be used to synchronize a task with an interrupt if there is a chance that the interrupt could arrive prior to the task being suspended - as this can lead to interrupts being missed. Use of a semaphore as a synchronization mechanism would avoid this eventuality.

Parameters

`pxTaskToResume` Handle to the task being readied.

Returns

`pdTRUE` if resuming the task should result in a context switch, otherwise `pdFALSE`. This is used by the ISR to determine if a context switch may be required following the ISR.

Example Usage

```
xTaskHandle xHandle; void vAFunction(void)
{
  // Create a task, storing the handle.
  xTaskCreate(vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle);
  // ... Rest of code.
}

void vTaskCode(void *pvParameters)
{
  // The task being suspended and resumed.
  for (;;)
  {
    // ... Perform some function here.
    // The task suspends itself.
    vTaskSuspend(NULL);
    // The task is now suspended, so will not reach here
```

```
    // until the ISR resumes it.
  }
}

void vAnExampleISR(void)
{
  portBASE_TYPE

  xYieldRequired;
  // Resume the suspended task.
  xYieldRequired = xTaskResumeFromISR(xHandle);
  if (xYieldRequired == pdTRUE)
  {
    // We should switch context so the ISR returns to a different task.
    // NOTE: How this is done depends on the port you are using. Check
    // the documentation and examples for your port.
    portYIELD_FROM_ISR();
  }
}
```

3.4.3 Kernel Control

3.4.3.1 `taskYIELD`

Macro for forcing a context switch.

3.4.3.2 `taskENTER_CRITICAL`

Macro to mark the start of a critical code region. Preemptive context switches cannot occur when in a critical region.

NOTE: This may alter the stack (depending on the portable implementation) so must be used with care!

3.4.3.3 `taskEXIT_CRITICAL`

Macro to mark the end of a critical code region. Preemptive context switches cannot occur when in a critical region.

NOTE: This may alter the stack (depending on the portable implementation) so must be used with care!

3.4.3.4 `taskDISABLE_INTERRUPTS`

Macro to disable all maskable interrupts.

3.4.3.5 `taskENABLE_INTERRUPTS`

Macro to enable microcontroller interrupts.

3.4.3.6 `vTaskStartScheduler`

Prototype

```
void vTaskStartScheduler(void);
```

Semantics

Starts the real time kernel tick processing. After calling the kernel has control over which tasks are executed and when.

The idle task is created automatically when `vTaskStartScheduler()` is called.

If `vTaskStartScheduler()` is successful the function will not return until an executing task calls `vTaskEndScheduler()`. The function might fail and return immediately if there is insufficient RAM available for the idle task to be created.

See the demo application file `main.c` for an example of creating tasks and starting the kernel.

Example Usage

```
void vAFunction(void)
{
  // Create at least one task before starting the kernel.
  xTaskCreate(vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, NULL);
  // Start the real time kernel with preemption.
  vTaskStartScheduler();
  // Will not get here unless a task calls vTaskEndScheduler()
}
```

3.4.3.7 `vTaskEndScheduler`

Prototype

```
void vTaskEndScheduler(void);
```

Semantics

Stops the real time kernel tick. All created tasks will be automatically deleted and multitasking (either preemptive or cooperative) will stop. Execution then resumes from the point where `vTaskStartScheduler()` was called, as if `vTaskStartScheduler()` had just returned.

See the demo application file `main.c` in the demo/PC directory for an example that uses `vTaskEndScheduler()`.

`vTaskEndScheduler()` requires an exit function to be defined within the portable layer (see `vPortEndScheduler()` in `port.c` for the PC port). This performs hardware specific operations such as stopping the kernel tick.

`vTaskEndScheduler()` will cause all of the resources allocated by the kernel to be freed - but will not free resources allocated by application tasks.

Example Usage

```
void vTaskCode(void * pvParameters)
{
  for (;;)
  {
    // Task code goes here.
    // At some point we want to end the real time kernel processing
    // so call ...
    vTaskEndScheduler();
  }
}

void vAFunction(void)
{
  // Create at least one task before starting the kernel.
  xTaskCreate(vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, NULL);
  // Start the real time kernel with preemption.
  vTaskStartScheduler();
  // Will only get here when the vTaskCode() task has called
  // vTaskEndScheduler(). When we get here we are back to single task
  // execution.
}
```

3.4.3.8 `vTaskSuspendAll`

Prototype

```
void vTaskSuspendAll(void);
```

Semantics

Suspends all real time kernel activity while keeping interrupts (including the kernel tick) enabled.

After calling `vTaskSuspendAll()` the calling task will continue to execute without risk of being swapped out until a call to `xTaskResumeAll()` has been made.

Example Usage

```
void vTask1(void * pvParameters)
{
  for (;;)
  {
    // Task code goes here.
    // ...
    // At some point the task wants to perform a long operation during
    // which it does not want to get swapped out. It cannot use
    // taskENTER_CRITICAL()/taskEXIT_CRITICAL() as the length of the
    // operation may cause interrupts to be missed - including the
    // ticks.
    // Prevent the real time kernel swapping out the task.
    vTaskSuspendAll();
    // Perform the operation here. There is no need to use critical
    // sections as we have all the microcontroller processing time.
    // During this time interrupts will still operate and the kernel
    // tick count will be maintained.
    // ...
    // The operation is complete. Restart the kernel.
    xTaskResumeAll();
  }
}
```

3.4.3.9 `xTaskResumeAll`

Prototype

```
portBASE_TYPE xTaskResumeAll(void);
```

Semantics

Resumes real time kernel activity following a call to `vTaskSuspendAll()`.

After a call to `xTaskSuspendAll()` the kernel will take control of which task is executing at any time.

Returns

If resuming the scheduler caused a context switch then `pdTRUE` is returned, otherwise `pdFALSE` is returned.

Example Usage

```
void vTask1(void * pvParameters)
```

```
{
  for (;;)
  {
    // Task code goes here.
    // ...
    // At some point the task wants to perform a long operation during
    // which it does not want to get swapped out. It cannot use
    // taskENTER_CRITICAL()/taskEXIT_CRITICAL() as the length of the
    // operation may cause interrupts to be missed - including the
    // ticks.
    // Prevent the real time kernel swapping out the task.
    xTaskSuspendAll();
    // Perform the operation here. There is no need to use critical
    // sections as we have all the microcontroller processing time.
    // During this time interrupts will still operate and the real
    // time kernel tick count will be maintained.
    // ...
    // The operation is complete. Restart the kernel. We want to force
    // a context switch - but there is no point if resuming
    // the scheduler caused a context switch already.
    if (!xTaskResumeAll()) {taskYIELD();}
  }
}
```

3.4.4 Task Utilities

3.4.4.1 `xTaskGetCurrentTaskHandle`

Prototype

```
xTaskHandle xTaskGetCurrentTaskHandle(void);
```

`INCLUDE_xTaskGetCurrentTaskHandle` must be set to 1 for this function to be available.

Returns

The handle of the currently running (calling) task.

3.4.4.2 `xTaskGetTickCount`

Prototype

```
volatile portTickType xTaskGetTickCount(void);
```

`INCLUDE_xTaskGetSchedulerState` must be set to 1 for this function to be available.

Returns

The count of ticks since `vTaskStartScheduler` was called.

3.4.4.3 `xTaskGetSchedulerState`

Prototype

```
portBASE_TYPE xTaskGetSchedulerState(void);
```

Returns

One of the following constants (defined within `task.h`): `taskSCHEDULER_NOT_STARTED`, `taskSCHEDULER_RUNNING`, `taskSCHEDULER_SUSPENDED`.

3.4.4.4 `uxTaskGetNumberOfTasks`

Prototype

```
unsigned portBASE_TYPE uxTaskGetNumberOfTasks(void);
```

Returns

The number of tasks that the real time kernel is currently managing. This includes all ready, blocked and suspended tasks. A task that has been deleted but not yet freed by the idle task will also be included in the count.

3.4.4.5 `vTaskList`

Prototype

```
void vTaskList(portCHAR *pcWriteBuffer);
```

`configUSE_TRACE_FACILITY`, `INCLUDE_vTaskDelete` and `INCLUDE_vTaskSuspend` must all be defined as 1 for this function to be available. See the configuration section for more information.

NOTE: This function will disable interrupts for its duration. It is not intended for normal application runtime use but as a debug aid. Lists all the current tasks, along with their current state and stack usage high water mark. Tasks are reported as blocked ('B'), ready ('R'), deleted ('D') or suspended ('S').

Parameters

`pcWriteBuffer` A buffer into which the above mentioned details will be written, in ascii form. This buffer is assumed to be large enough to contain the generated report. Approximately 40 bytes per task should be sufficient.

3.4.4.6 `vTaskStartTrace`

Prototype

```
void vTaskStartTrace(portCHAR * pcBuffer,
                     unsigned portLONG ulBufferSize);
```

Semantics

Starts a real time kernel activity trace. The trace logs the identity of which task is running when. The trace file is stored in binary format. A separate DOS utility called `convtrce.exe` is used to convert this into a tab delimited text file which can be viewed and plotted in a spread sheet.

Parameters

`pcBuffer` The buffer into which the trace will be written.

`ulBufferSize` The size of `pcBuffer` in bytes.

The trace will continue until either the buffer in full, or `ulTaskEndTrace()` is called.

3.4.4.7 `ulTaskEndTrace`

Prototype

```
unsigned portLONG ulTaskEndTrace(void);
```

Semantics

Stops a kernel activity trace. See `vTaskStartTrace()`.

Returns

The number of bytes that have been written into the trace buffer.

3.4.5 Queue Management

3.4.5.1 `uxQueueMessagesWaiting`

Prototype

```
unsigned portBASE_TYPEvuxQueueMessagesWaiting(xQueueHandle xQueue);
```

Semantics

Return the number of messages stored in a queue.

Parameters

`xQueue` A handle to the queue being queried.

Returns

The number of messages available in the queue.

3.4.5.2 `vQueueDelete`

Prototype

```
void vQueueDelete(xQueueHandle xQueue);
```

Semantics

Delete a queue - freeing all the memory allocated for storing of items placed on the queue.

Parameters

`xQueue` A handle to the queue to be deleted.

3.4.5.3 `xQueueCreate`

Prototype

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

Semantics

Creates a new queue instance. This allocates the storage required by the new queue and returns a handle for the queue.

Parameters

`uxQueueLength` The maximum number of items that the queue can contain.

`uxItemSize` The number of bytes each item in the queue will require. Items are queued by copy, not by reference, so this is the number of bytes that will be copied for each posted item. Each item on the queue must be the same size.

Returns

If the queue is successfully create then a handle to the newly created queue is returned. If the queue cannot be created then 0 is returned.

Example Usage

```
struct AMessage
{
  portCHAR ucMessageID;
  portCHAR ucData[20];
};

void vATask(void *pvParameters)
{
  xQueueHandle xQueue1, xQueue2;
  // Create a queue capable of containing 10 unsigned long values.
  xQueue1 = xQueueCreate(10, sizeof(unsigned portLONG));
  if (xQueue1 == 0)
  {
    // Queue was not created and must not be used.
  }
  // Create a queue capable of containing 10 pointers
  // to AMessage structures.
  // These should be passed by pointer as they contain a lot of data.
  xQueue2 = xQueueCreate(10, sizeof(struct AMessage *));
  if (xQueue2 == 0)
  {
    // Queue was not created and must not be used.
  }
  // ... Rest of task code.
}
```

3.4.5.4 `xQueueSend`

Prototype

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

This is a macro that calls `xQueueGenericSend()`. It is included for backward compatibility with versions of `FreeRTOS.org` that did not include the `xQueueSendToFront()` and `xQueueSendToBack()` macros. It is equivalent to `xQueueSendToBack()`.

Semantics

Post an item on a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See `xQueueSendFromISR()` for an alternative which may be used in an ISR.

This function is part of the fully featured intertask communications API. See Design concepts and performance optimisation for advanced options and other information.

Parameters

`xQueue` The handle to the queue on which the item is to be posted.

`pvItemToQueue` A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from `pvItemToQueue` into the queue storage area.

`xTicksToWait` The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0. The time is defined in tick periods so the constant `portTICK_RATE_MS` should be used to convert to real time if this is required.

If `INCLUDE_vTaskSuspend` is set to '1' then specifying the block time as `portMAX_DELAY` will cause the task to block indefinitely (without a timeout).

Returns

`pdTRUE` if the item was successfully posted, otherwise `errQUEUE_FULL`.

Example Usage

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

unsigned portLONG ulVar = 10UL;

void vATask(void *pvParameters)
{
  xQueueHandle xQueue1, xQueue2;
  struct AMessage *pxMessage;
  // Create a queue capable of containing 10 unsigned long values.
  xQueue1 = xQueueCreate(10, sizeof(unsigned portLONG));
  // Create a queue capable of containing 10 pointers
  // to AMessage structures.
  // These should be passed by pointer as they contain a lot of data.
  xQueue2 = xQueueCreate(10, sizeof(struct AMessage *));
  // ...
  if (xQueue1 != 0)
  {
    // Send an unsigned long. Wait for 10 ticks for space to become
    // available if necessary.
    if (xQueueSend(xQueue1,
                   (void *) &ulVar,
                   (portTickType) 10) != pdPASS)
    {
      // Failed to post the message, even after 10 ticks.
    }
  }
  if (xQueue2 != 0)
  {
    // Send a pointer to a struct AMessage object. Don't block if the
    // queue is already full.
    pxMessage = & xMessage;
    xQueueSend(xQueue2, (void *) &pxMessage, (portTickType) 0);
  }
  // ... Rest of task code.
}
```

3.4.5.5 xQueueSendToBack

Only available from **FreeRTOS V4.5.0** onwards.

Prototype

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

This is a macro that calls `xQueueGenericSend()`. It is equivalent to `xQueueSend()`.

Semantics

Post an item to the back of a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine.

See `xQueueSendToBackFromISR()` for an alternative which may be used in an ISR. This function is part of the fully featured intertask communications API. See Design concepts and performance optimisation for advanced options and other information.

Parameters

`xQueue` The handle to the queue on which the item is to be posted.

`pvItemToQueue` A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from `pvItemToQueue` into the queue storage area.

`xTicksToWait` The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0. The time is defined in tick periods so the constant `portTICK_RATE_MS` should be used to convert to real time if this is required.

If `INCLUDE_vTaskSuspend` is set to '1' then specifying the block time as `portMAX_DELAY` will cause the task to block indefinitely (without a timeout).

Returns

`pdTRUE` if the item was successfully posted, otherwise `errQUEUE_FULL`.

Example Usage

```
struct AMessage
{
  portCHAR ucMessageID;
  portCHAR ucData[20];
```

```
} xMessage;

unsigned portLONG ulVar = 10UL;
void vATask(void *pvParameters)
{
  xQueueHandle xQueue1, xQueue2;
  struct AMessage *pxMessage;
  // Create a queue capable of containing 10 unsigned long values.
  xQueue1 = xQueueCreate(10, sizeof(unsigned portLONG));
  // Create a queue capable of containing 10 pointers
  // to AMessage structures.
  // These should be passed by pointer as they contain a lot of data.
  xQueue2 = xQueueCreate(10, sizeof(struct AMessage *));
  // ...
  if (xQueue1 != 0)
  {
    // Send an unsigned long. Wait for 10 ticks for space to become
    // available if necessary.
    if (xQueueSendToBack(xQueue1,
                         (void *) &ulVar,
                         (portTickType) 10) != pdPASS)
    {
      // Failed to post the message, even after 10 ticks.
    }
  }
  if (xQueue2 != 0)
  {
    // Send a pointer to a struct AMessage object. Don't block if the
    // queue is already full.
    pxMessage = & xMessage;
    xQueueSendToBack(xQueue2, (void *) &pxMessage, (portTickType) 0);
  }
  // ... Rest of task code.
}
```

3.4.5.6 `xQueueSendToFront`

Only available from **FreeRTOS V4.5.0** onwards.

Prototype

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

This is a macro that calls `xQueueGenericSend()`.

Semantics

Post an item to the front of a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine.

See `xQueueSendToFrontFromISR()` for an alternative which may be used in an ISR. This function is part of the fully featured intertask communications API. See Design concepts and performance optimisation for advanced options and other information.

Parameters

`xQueue` The handle to the queue on which the item is to be posted.

`pvItemToQueue` A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from `pvItemToQueue` into the queue storage area.

`xTicksToWait` The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0. The time is defined in tick periods so the constant `portTICK_RATE_MS` should be used to convert to real time if this is required.

If `INCLUDE_vTaskSuspend` is set to '1' then specifying the block time as `portMAX_DELAY` will cause the task to block indefinitely (without a time-out).

Returns

`pdTRUE` if the item was successfully posted, otherwise `errQUEUE_FULL`.

Example Usage

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

unsigned portLONG ulVar = 10UL;

void vATask(void *pvParameters)
{
  xQueueHandle xQueue1, xQueue2;
  struct AMessage *pxMessage;
  // Create a queue capable of containing 10 unsigned long values.
  xQueue1 = xQueueCreate(10, sizeof(unsigned portLONG));
  // Create a queue capable of containing 10 pointers
```

```
  // to AMessage structures.
  // These should be passed by pointer as they contain a lot of data.
  xQueue2 = xQueueCreate(10, sizeof(struct AMessage *));
  // ...
  if (xQueue1 != 0)
  {
    // Send an unsigned long. Wait for 10 ticks for space to become
    // available if necessary.
    if (xQueueSendToFront(xQueue1,
                          (void *) &ulVar,
                          (portTickType) 10) != pdPASS)
    {
      // Failed to post the message, even after 10 ticks.
    }
  }
  if (xQueue2 != 0)
  {
    // Send a pointer to a struct AMessage object. Don't block if the
    // queue is already full.
    pxMessage = & xMessage;
    xQueueSendToFront(xQueue2, (void *) &pxMessage, (portTickType) 0);
  }
  // ... Rest of task code.
}
```

3.4.5.7 `xQueueReceive`

Prototype

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

This is a macro that calls the `xQueueGenericReceive()` function.

Semantics

Receive an item from a queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

This function must not be used in an interrupt service routine. See `xQueueReceiveFromISR` for an alternative that can. This function is part of the fully featured intertask communications API. See Design concepts and performance optimisation for advanced options and other information.

Parameters

`pxQueue` The handle to the queue from which the item is to be received.

`pvBuffer` Pointer to the buffer into which the received item will be copied.

`xTicksToWait` The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call. The time is defined in tick periods so the constant `portTICK_RATE_MS` should be used to convert to real time if this is required.

If `INCLUDE_vTaskSuspend` is set to '1' then specifying the block time as `portMAX_DELAY` will cause the task to block indefinitely (without a time-out).

Returns

`pdTRUE` if an item was successfully received from the queue, otherwise `pdFALSE`.

Example Usage

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

xQueueHandle xQueue;

// Task to create a queue and post a value.
void vATask(void *pvParameters)
{
  struct AMessage *pxMessage;
  // Create a queue capable of containing 10 pointers
  // to AMessage structures.
  // These should be passed by pointer as they contain a lot of data.
  xQueue = xQueueCreate(10, sizeof(struct AMessage *));
  if (xQueue == 0)
  {
    // Failed to create the queue.
  }
  // ...
  // Send a pointer to a struct AMessage object. Don't block if the
  // queue is already full.
  pxMessage = & xMessage;
  xQueueSend(xQueue, (void *) &pxMessage, (portTickType) 0);
  // ... Rest of task code.
}

// Task to receive from the queue.
```

```
void vADifferentTask(void *pvParameters)
{
  struct AMessage *pxRxedMessage;
  if (xQueue != 0)
  {
    // Receive a message on the created queue.
    // Block for 10 ticks if a
    // message is not immediately available.
    if (xQueueReceive(xQueue, &(pxRxedMessage), (portTickType) 10))
    {
      // pcRxedMessage now points to the struct
      // AMessage variable posted
      // by vATask.
    }
  }
  // ... Rest of task code.
}
```

3.4.5.8 `xQueuePeek`

Only available from **FreeRTOS V4.5.0** onwards.

Prototype

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

This is a macro that calls the `xQueueGenericReceive()` function.

Semantics

Receive an item from a queue without removing the item from the queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

Successfully received items remain on the queue so will be returned again by the next call, or a call to `xQueueReceive()`.

This macro must not be used in an interrupt service routine.

Parameters

`xQueue` The handle to the queue from which the item is to be received.

`pvBuffer` Pointer to the buffer into which the received item will be copied. This must be at least large enough to hold the size of the queue item defined when the queue was created.

`xTicksToWait` The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call. The time is defined in tick periods so the constant `portTICK_RATE_MS` should be used to convert to real time if this is required. If `INCLUDE_vTaskSuspend` is set to '1' then specifying the block time as `portMAX_DELAY` will cause the task to block indefinitely (without a timeout).

Returns

`pdTRUE` if an item was successfully received (peeked) from the queue, otherwise `pdFALSE`.

Example Usage

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

xQueueHandle xQueue;

// Task to create a queue and post a value.
void vATask(void *pvParameters)
{
  struct AMessage *pxMessage;
  // Create a queue capable of containing 10 pointers
  // to AMessage structures.
  // These should be passed by pointer as they contain a lot of data.
  xQueue = xQueueCreate(10, sizeof(struct AMessage *));

  if (xQueue == 0)
  {
    // Failed to create the queue.
  }
  // ...
  // Send a pointer to a struct AMessage object. Don't block if the
  // queue is already full.
  pxMessage = & xMessage;
  xQueueSend(xQueue, (void *) &pxMessage, (portTickType) 0);
  // ... Rest of task code.
}

// Task to peek the data from the queue.
void vADifferentTask(void *pvParameters)
{ struct AMessage *pxRxedMessage;

  if (xQueue != 0)
```

```
  {
    // Peek a message on the created queue. Block for 10 ticks if a
    // message is not immediately available.
    if (xQueuePeek(xQueue, &(pxRxedMessage), (portTickType) 10))
    {
      // pcRxedMessage now points to the struct AMessage variable
      // posted by vATask, but the item still remains on the queue.
    }
  }
  // ... Rest of task code.
}
```

3.4.5.9 `xQueueSendFromISR`

Prototype

```
portBASE_TYPE xQueueSendFromISR(xQueueHandle pxQueue,
                                const void *pvItemToQueue,
                                portBASE_TYPE xTaskPreviouslyWoken);
```

This is a macro that calls `xQueueGenericSendFromISR()`. It is included for backward compatibility with versions of `FreeRTOS.org` that did not include the `xQueueSendToBackFromISR()` and `xQueueSendToFrontFromISR()` macros.

Semantics

Post an item on a queue. It is safe to use this function from within an interrupt service routine.

Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR. In most cases it would be preferable to store a pointer to the item being queued.

Parameters

`xQueue` The handle to the queue on which the item is to be posted.

`pvItemToQueue` A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from `pvItemToQueue` into the queue storage area.

`xTaskPreviouslyWoken` This is included so an ISR can post onto the same queue multiple times from a single interrupt. The first call should always pass in`pdFALSE`. Subsequent calls should pass in the value returned from the previous call. See the file `serial.c` in the PC port for a good example of this mechanism.

Returns

`pdTRUE` if a task was woken by posting onto the queue. This is used by the ISR to determine if a context switch may be required following the ISR.

Example Usage - For Buffered IO

Where the ISR can obtain more than one value per call:

```
void vBufferISR(void)
{
  portCHAR cIn;
  portBASE_TYPE xTaskWokenByPost;
  // We have not woken a task at the start of the ISR.
  xTaskWokenByPost = pdFALSE;
  // Loop until the buffer is empty.
  do
  {
    // Obtain a byte from the buffer.
    cIn = portINPUT_BYTE(RX_REGISTER_ADDRESS);
    // Post the byte. The first time round the loop xTaskWokenByPost
    // will be pdFALSE. If the queue send causes a task to wake we do
    // not want the task to run until we have finished the ISR, so
    // xQueueSendFromISR does not cause a context switch. Also we
    // don't want subsequent posts to wake any other tasks, so we store
    // the return value back into xTaskWokenByPost so xQueueSendFromISR
    // knows not to wake any task the next iteration of the loop.
    xTaskWokenByPost = xQueueSendFromISR(xRxQueue,
                                         &cIn,
                                         xTaskWokenByPost);
  } while(portINPUT_BYTE(BUFFER_COUNT));
  // Now the buffer is empty we can switch context if necessary.
  if (xTaskWokenByPost)
  {
    // We should switch context so the ISR returns to a different task.
    // NOTE: How this is done depends on the port you are using. Check
    // the documentation and examples for your port.
    portYIELD_FROM_ISR();
  }
}
```

3.4.5.10 `xQueueSendToBackFromISR`

Only available from **FreeRTOS V4.5.0** onwards.

Prototype

```
portBASE_TYPE xQueueSendToBackFromISR(xQueueHandle pxQueue,
```

```
                                    const void *pvItemToQueue,
                                    portBASE_TYPE xTaskPreviouslyWoken);
```

This is a macro that calls `xQueueGenericSendFromISR()`.

Semantics

Post an item to the back of a queue. It is safe to use this function from within an interrupt service routine. Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR. In most cases it would be preferable to store a pointer to the item being queued.

Parameters

`xQueue` The handle to the queue on which the item is to be posted.

`pvItemToQueue` A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from `pvItemToQueue` into the queue storage area.

`xTaskPreviouslyWoken` This is included so an ISR can post onto the same queue multiple times from a single interrupt. The first call should always pass in `pdFALSE`. Subsequent calls should pass in the value returned from the previous call. See the file `serial.c` in the PC port for a good example of this mechanism.

Returns

`pdTRUE` if a task was woken by posting onto the queue. This is used by the ISR to determine if a context switch may be required following the ISR.

Example Usage for Buffered IO

Where the ISR can obtain more than one value per call:

```
void vBufferISR(void)
{
  portCHAR cIn;
  portBASE_TYPE xTaskWokenByPost;
  // We have not woken a task at the start of the ISR.
  xTaskWokenByPost = pdFALSE;
  // Loop until the buffer is empty.
  do
  {
    // Obtain a byte from the buffer.
    cIn = portINPUT_BYTE(RX_REGISTER_ADDRESS);
    // Post the byte. The first time round the loop xTaskWokenByPost
    // will be pdFALSE. If the queue send causes a task to wake we do
    // not want the task to run until we have finished the ISR, so
```

```
    // xQueueSendToBackFromISR does not cause a context switch. Also we
    // don't want subsequent posts to wake any other tasks, so we store
    // the return value back into xTaskWokenByPost so xQueueSendToBackFromISR
    // knows not to wake any task the next iteration of the loop.
    xTaskWokenByPost = xQueueSendToBackFromISR(xRxQueue,
                                               &cIn,
                                               xTaskWokenByPost);
  } while (portINPUT_BYTE(BUFFER_COUNT));
  // Now the buffer is empty we can switch context if necessary.
  if (xTaskWokenByPost)
  {
    // We should switch context so the ISR returns to a different task.
    // NOTE: How this is done depends on the port you are using. Check
    // the documentation and examples for your port.
    portYIELD_FROM_ISR();
  }
}
```

3.4.5.11 `xQueueSendToFrontFromISR`

Only available from **FreeRTOS V4.5.0** onwards.

Prototype

```
portBASE_TYPE xQueueSendToFrontFromISR(xQueueHandle pxQueue,
                                       const void *pvItemToQueue,
                                       portBASE_TYPE xTaskPreviouslyWoken);
```

This is a macro that calls `xQueueGenericSendFromISR()`.

Semantics

Post an item to the front of a queue. It is safe to use this function from within an interrupt service routine. Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR. In most cases it would be preferable to store a pointer to the item being queued.

Parameters

`xQueue` The handle to the queue on which the item is to be posted.

`pvItemToQueue` A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from `pvItemToQueue` into the queue storage area.

`xTaskPreviouslyWoken` This is included so an ISR can post onto the same queue multiple times from a single interrupt. The first call should always pass in `pdFALSE`. Subsequent calls should pass in the value returned from

the previous call. See the file `serial.c` in the PC port for a good example of this mechanism.

Returns

`pdTRUE` if a task was woken by posting onto the queue. This is used by the ISR to determine if a context switch may be required following the ISR.

Example Usage for Buffered IO

Where the ISR can obtain more than one value per call:

```
void vBufferISR(void)
{
  portCHAR cIn;
  portBASE_TYPE xTaskWokenByPost;
  // We have not woken a task at the start of the ISR.
  xTaskWokenByPost = pdFALSE;
  // Loop until the buffer is empty.
  do
  {
    // Obtain a byte from the buffer.
    cIn = portINPUT_BYTE(RX_REGISTER_ADDRESS);
    // Post the byte. The first time round the loop xTaskWokenByPost
    // will be pdFALSE. If the queue send causes a task to wake we do
    // not want the task to run until we have finished the ISR, so
    // xQueueSendToFrontFromISR does not cause a context switch. Also we
    // don't want subsequent posts to wake any other tasks, so we store
    // the return value back into xTaskWokenByPost so xQueueSendToFrontFromISR
    // knows not to wake any task the next iteration of the loop.
    xTaskWokenByPost = xQueueSendToFrontFromISR(xRxQueue,
                                                &cIn,
                                                xTaskWokenByPost);
  } while (portINPUT_BYTE(BUFFER_COUNT));
  // Now the buffer is empty we can switch context if necessary.
  if (xTaskWokenByPost)
  {
    // We should switch context so the ISR returns to a different task.
    // NOTE: How this is done depends on the port you are using. Check
    // the documentation and examples for your port.
    portYIELD_FROM_ISR();
  }
}
```

3.4.5.12 `xQueueReceiveFromISR`

Prototype

```
portBASE_TYPE xQueueReceiveFromISR(xQueueHandle pxQueue,
```

```
                                        void *pvBuffer,
                                        portBASE_TYPE *pxTaskWoken);
```

Semantics

Receive an item from a queue. It is safe to use this function from within an interrupt service routine.

Parameters

`pxQueue` The handle to the queue from which the item is to be received.

`pvBuffer` Pointer to the buffer into which the received item will be copied.

`pxTaskWoken` A task may be blocked waiting for space to become available on the queue. If `xQueueReceiveFromISR` causes such a task to unblock `*pxTaskWoken` will get set to `pdTRUE`, otherwise `*pxTaskWoken` will remain unchanged.

Returns

`pdTRUE` if an item was successfully received from the queue, otherwise `pdFALSE`.

Example Usage

```
xQueueHandle xQueue;

// Function to create a queue and post some values.
void vAFunction(void *pvParameters)
{
  portCHAR cValueToPost;
  const portTickType xBlockTime = (portTickType) 0xff;
  // Create a queue capable of containing 10 characters.
  xQueue = xQueueCreate(10, sizeof(portCHAR));
  if (xQueue == 0)
  {
    // Failed to create the queue.
  }
  // ...
  // Post some characters that will be used within an ISR. If the queue
  // is full then this task will block for xBlockTime ticks.
  cValueToPost = 'a'; xQueueSend(xQueue, (void *) &cValueToPost, xBlockTime);
  cValueToPost = 'b'; xQueueSend(xQueue, (void *) &cValueToPost, xBlockTime);
  // ... keep posting characters ... this task may block when the queue
  // becomes full.
  cValueToPost = 'c'; xQueueSend(xQueue, (void *) &cValueToPost, xBlockTime);
}

// ISR that outputs all the characters received on the queue.
void vISR_Routine(void)
```

```
{
  portBASE_TYPE xTaskWokenByReceive = pdFALSE;
  portCHAR cRxedChar;
  while (xQueueReceiveFromISR(xQueue,
                              (void *) &cRxedChar,
                              &xTaskWokenByReceive))
  {
    // A character was received. Output the character now.
    vOutputCharacter(cRxedChar);
    // If removing the character from the queue woke the task that was
    // posting onto the queue xTaskWokenByReceive will have been set to
    // pdTRUE. No matter how many times this loop iterates only one
    // task will be woken.
  }
  if (xTaskWokenByPost != pdFALSE)
  {
    // We should switch context so the ISR returns to a different task.
    // NOTE: How this is done depends on the port you are using. Check
    // the documentation and examples for your port.
    taskYIELD();
  }
}
```

3.4.6 Semaphore Management

3.4.6.1 `SemaphoreCreateBinary`

Prototype

```
vSemaphoreCreateBinary(xSemaphoreHandle xSemaphore)
```

Semantics

Macro that creates a semaphore by using the existing queue mechanism. The queue length is 1 as this is a binary semaphore. The data size is 0 as we don't want to actually store any data - we just want to know if the queue is empty or full.

Binary semaphores and mutexes are very similar but have some subtle differences: Mutexes include a priority inheritance mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronization (between tasks or between tasks and an interrupt), and mutexes the better choice for implementing simple mutual exclusion.

A binary semaphore need not be given back once obtained, so task synchronization can be implemented by one task/interrupt continuously 'giving' the semaphore while another continuously 'takes' the semaphore. This is demonstrated by the sample code on the `xSemaphoreGiveFromISR()` documentation page.

The priority of a task that 'takes' a mutex can potentially be raised if another task of higher priority attempts to obtain the same mutex. The task that owns the mutex 'inherits' the priority of the task attempting to 'take' the same mutex. This means the mutex must always be 'given' back - otherwise the higher priority task will never be able to obtain the mutex, and the lower priority task will never 'disinherit' the priority. An example of a mutex being used to implement mutual exclusion is provided on the `xSemaphoreTake()` documentation page.

Both mutex and binary semaphores are assigned to variables of type `xSemaphoreHandle` and can be used in any API function that takes a parameter of this type.

Parameters

`xSemaphore` Handle to the created semaphore. Should be of type `xSemaphoreHandle`.

Example Usage

xSemaphoreHandle xSemaphore;

void vATask(void * pvParameters) // Semaphore cannot be used before a call to vSemaphoreCreateBinary(). // This is a macro so pass the variable in directly. vSemaphoreCreateBinary(xSemaphore); if (xSemaphore != NULL) // The semaphore was created successfully. // The semaphore can now be used.

3.4.6.2 `xSemaphoreCreateMutex`

This only available from **FreeRTOS V4.5.0** onwards.

Prototype

```
xSemaphoreHandle xQueueCreateMutex(void)
```

Semantics

Macro that creates a mutex semaphore by using the existing queue mechanism.

Mutexes and binary semaphores are very similar but have some subtle differences: Mutexes include a priority inheritance mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronisation (between tasks or between tasks and an interrupt), and mutexes the better choice for implementing simple mutual exclusion.

The priority of a task that 'takes' a mutex can potentially be raised if another task of higher priority attempts to obtain the same mutex. The task that owns the mutex 'inherits' the priority of the task attempting to 'take' the same mutex. This means the mutex must always be 'given' back - otherwise the higher priority task will never be able to obtain the mutex, and the lower priority task will never

'disinherit' the priority. An example of a mutex being used to implement mutual exclusion is provided on the `xSemaphoreTake()` documentation page.

A binary semaphore need not be given back once obtained, so task synchronisation can be implemented by one task/interrupt continuously 'giving' the semaphore while another continuously 'takes' the semaphore. This is demonstrated by the sample code on the `xSemaphoreGiveFromISR()` documentation page.

Both mutex and binary semaphores are assigned to variables of type `xSemaphoreHandle` and can be used in any API function that takes a parameter of this type.

Parameters

`xSemaphore` Handle to the created semaphore. Should be of type `xSemaphoreHandle`.

Example Usage

```
xSemaphoreHandle xSemaphore;

void vATask(void * pvParameters)
{
  // Mutex semaphores cannot be used before a call to
  // vSemaphoreCreateMutex(). The created mutex is returned.
  xSemaphore = vSemaphoreCreateBinary();
  if (xSemaphore != NULL)
  {
    // The semaphore was created successfully.
    // The semaphore can now be used.
  }
}
```

3.4.6.3 `xSemaphoreTake`

Prototype

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

Semantics

Macro to obtain a semaphore.

The semaphore must of been created using `vSemaphoreCreateBinary()`. This macro must not be called from an ISR. `xQueueReceiveFromISR()` can be used to take a semaphore from within an interrupt if required, although this

would not be a normal operation. Semaphores use queues as their underlying mechanism, so functions are to some extent interoperable. This function is part of the fully featured intertask communications API. See Design concepts and performance optimization for advanced options and other information.

Parameters

`xSemaphore` A handle to the semaphore being obtained. This is the handle returned by `vSemaphoreCreateBinary()`;

`xBlockTime` The time in ticks to wait for the semaphore to become available. The macro `portTICK_RATE_MS` can be used to convert this to a real time. A block time of zero can be used to poll the semaphore.

If `INCLUDE_vTaskSuspend` is set to '1' then specifying the block time as `portMAX_DELAY` will cause the task to block indefinitely (without a time-out).

Returns

`pdTRUE` if the semaphore was obtained. `pdFALSE` if `xBlockTime` expired without the semaphore becoming available.

Example Usage

```
xSemaphoreHandle xSemaphore = NULL;

// A task that creates a semaphore.
void vATask(void * pvParameters)
{
  // Create the semaphore to guard a shared resource. As we are using
  // the semaphore for mutual exclusion we create a mutex semaphore
  // rather than a binary semaphore.
  xSemaphore = xSemaphoreCreateMutex();
}

// A task that uses the semaphore.
void vAnotherTask(void * pvParameters)
{
  // ... Do other things.
  if (xSemaphore != NULL)
  {
    // See if we can obtain the semaphore. If the semaphore is not available
    // wait 10 ticks to see if it becomes free.
    if (xSemaphoreTake(xSemaphore, (portTickType) 10) == pdTRUE)
    {
      // We were able to obtain the semaphore and can now access the
      // shared resource.
      // ...
      // We have finished accessing the shared resource. Release the
      // semaphore.
```

```
      xSemaphoreGive(xSemaphore);
    }
    else
    {
      // We could not obtain the semaphore and can therefore not access
      // the shared resource safely.
    }
  }
}
```

3.4.6.4 `xSemaphoreGive`

Prototype

```
xSemaphoreGive(xSemaphoreHandle xSemaphore)
```

Semantics

Macro to release a semaphore.

The semaphore must of been created using `vSemaphoreCreateBinary()`, and obtained using `xSemaphoreTake()`. This must not be used from an ISR. See `xSemaphoreGiveFromISR()` for an alternative which can be used from an ISR.

This function is part of the fully featured intertask communications API. See Design concepts and performance optimisation for advanced options and other information.

Parameters

`xSemaphore` A handle to the semaphore being released. This is the handle returned by `vSemaphoreCreateBinary()`;

Returns

`pdTRUE` if the semaphore was released. `pdFALSE` if an error occurred.

Semaphores are implemented using queues. An error can occur if there is no space on the queue to post a message - indicating that the semaphore was not first obtained correctly.

Example Usage

```
xSemaphoreHandle xSemaphore = NULL;
void vATask(void * pvParameters)
{
  // Create the semaphore to guard a shared resource. As we are using
  // the semaphore for mutual exclusion we create a mutex semaphore
```

```
  // rather than a binary semaphore.
  xSemaphore = xSemaphoreCreateMutex();
  if (xSemaphore != NULL)
  {
    if (xSemaphoreGive(xSemaphore) != pdTRUE)
    {
      // We would expect this call to fail because we cannot give
      // a semaphore without first "taking" it!
    }
    // Obtain the semaphore - don't block if the semaphore is not
    // immediately available.
    if (xSemaphoreTake(xSemaphore, (portTickType) 0))
    {
      // We now have the semaphore and can access the shared resource.
      // ...
      // We have finished accessing the shared resource so can free the
      // semaphore.
      if (xSemaphoreGive(xSemaphore) != pdTRUE)
      {
        // We would not expect this call to fail because we must have
        // obtained the semaphore to get here.
      }
    }
  }
}
```

3.4.6.5 `xSemaphoreGiveFromISR`

Prototype

```
xSemaphoreGiveFromISR(xSemaphoreHandle xSemaphore,
                      portBASE_TYPE xTaskPreviouslyWoken)
```

Semantics

Macro to release a semaphore.

The semaphore must of been created using `vSemaphoreCreateBinary()`, and obtained using `xSemaphoreTake()`.

This macro can be used from an ISR.

Parameters

`xSemaphore` A handle to the semaphore being released. This is the handle returned by `vSemaphoreCreateBinary()`;

`xTaskPreviouslyWoken` This is included so an ISR can make multiple calls to `xSemaphoreGiveFromISR()` from a single interrupt. The first call should always pass in `pdFALSE`. Subsequent calls should pass in the value returned

from the previous call. See the file `serial.c` in the PC port for a good example of using `xSemaphoreGiveFromISR()`.

Returns

`pdTRUE` if a task was woken by releasing the semaphore. This is used by the ISR to determine if a context switch may be required following the ISR.

Example Usage

```
#define LONG_TIME 0xffff
#define TICKS_TO_WAIT 10

xSemaphoreHandle xSemaphore = NULL;

// Repetitive task.
void vATask(void * pvParameters)
{
  // We are using the semaphore for synchronisation to we create a binary
  // semaphore rather than a mutex. We must make sure that the interrupt
  // does not attempt to use the semaphore before it is created!
  xSemaphoreCreateBinary(xSemaphore);
  for (;;)
  {
    // We want this task to run every 10 ticks or a timer. The semaphore
    // was created before this task was started
    // Block waiting for the semaphore to become available.
    if (xSemaphoreTake(xSemaphore, LONG_TIME) == pdTRUE)
    {
      // It is time to execute.
      // ...
      // We have finished our task. Return to the top of the loop where
      // we will block on the semaphore until it is time to execute
      // again.
    }
  }
}

// Timer ISR
void vTimerISR(void * pvParameters)
{
  static unsigned portCHAR ucLocalTickCount = 0;
  static portBASE_TYPE xTaskWoken = pdFALSE;

  // A timer tick has occurred.
  // ... Do other time functions.
  // Is it time for vATask() to run?
  ucLocalTickCount++;
  if (ucLocalTickCount >= TICKS_TO_WAIT)
  {
    // Unblock the task by releasing the semaphore.
    xTaskWoken = xSemaphoreGiveFromISR(xSemaphore, xTaskWoken);
```

```
    // Reset the count so we release the semaphore again in 10 ticks time.
    ucLocalTickCount = 0;
  }
  // If xTaskWoken was set to true you may want to yield (force a switch)
  // here.
}
```

3.4.7 Coroutine Management

3.4.7.1 xCoRoutineCreate

Prototype

```
portBASE_TYPE xCoRoutineCreate(crCOROUTINE_CODE pxCoRoutineCode,
                               unsigned portBASE_TYPE uxPriority,
                               unsigned portBASE_TYPE uxIndex);
```

Semantics

Create a new coroutine and add it to the list of coroutines that are ready to run.

Parameters

`pxCoRoutineCode` Pointer to the coroutine function. Co-routine functions require special syntax - see the coroutine section of the WEB documentation for more information.

`uxPriority` The priority with respect to other coroutines at which the coroutine will run.

`uxIndex` Used to distinguish between different coroutines that execute the same function. See the example below and the coroutine section of the WEB documentation for further information.

Returns

`pdPASS` if the coroutine was successfully created and added to a ready list, otherwise an error code defined with `ProjDefs.h`.

Example Usage

```
// Co-routine to be created.
void vFlashCoRoutine(xCoRoutineHandle xHandle,
                     unsigned portBASE_TYPE uxIndex)
{
```

```
  // Variables in coroutines must be declared static
  // if they must maintain value across a blocking call.
  // This may not be necessary for const variables.
  static const char cLedToFlash[2] = {5, 6};
  static const portTickType xTimeToDelay[2] = {200, 400};
  // Must start every coroutine with a call to crSTART();
  crSTART(xHandle);
  for (;;)
  {
    // This coroutine just delays for a fixed period, then toggles
    // an LED. Two coroutines are created using this function, so
    // the uxIndex parameter is used to tell the coroutine which
    // LED to flash and how long to delay. This assumes xQueue has
    // already been created.
    vParTestToggleLED(cLedToFlash[uxIndex]);
    crDELAY(xHandle, uxFlashRates[uxIndex]);
  }
  // Must end every coroutine with a call to crEND();
  crEND();
}

// Function that creates two coroutines.
void vOtherFunction(void)
{
  unsigned char ucParameterToPass;
  xTaskHandle xHandle;
  // Create two coroutines at priority 0. The first is given index 0
  // so (from the code above) toggles LED 5 every 200 ticks. The second
  // is given index 1 so toggles LED 6 every 400 ticks.
  for (uxIndex = 0; uxIndex < 2; uxIndex++)
  {
    xCoRoutineCreate(vFlashCoRoutine, 0, uxIndex);
  }
}
```

`xCoRoutineHandle`

Type by which coroutines are referenced. The coroutine handle is automatically passed into each coroutine function.

3.4.7.2 `xCoRoutineCreate`

Prototype

```
portBASE_TYPE xCoRoutineCreate(crCOROUTINE_CODE pxCoRoutineCode,
                               unsigned portBASE_TYPE uxPriority,
                               unsigned portBASE_TYPE uxIndex);
```

Semantics

Create a new coroutine and add it to the list of coroutines that are ready to run.

Parameters

`pxCoRoutineCode` Pointer to the coroutine function. Co-routine functions require special syntax - see the coroutine section of the WEB documentation for more information.

`uxPriority` The priority with respect to other coroutines at which the coroutine will run.

`uxIndex` Used to distinguish between different coroutines that execute the same function. See the example below and the coroutine section of the WEB documentation for further information.

Returns

`pdPASS` if the coroutine was successfully created and added to a ready list, otherwise an error code defined with `ProjDefs.h`.

Example Usage

```
// Co-routine to be created.
void vFlashCoRoutine(xCoRoutineHandle xHandle,
                     unsigned portBASE_TYPE uxIndex)
{
  // Variables in coroutines must be declared static
  // if they must maintain value across a blocking call.
  // This may not be necessary for const variables.
  static const char cLedToFlash[2] = {5, 6};
  static const portTickType xTimeToDelay[2] = {200, 400};
  // Must start every coroutine with a call to crSTART();
  crSTART(xHandle);
  for (;;)
  {
    // This coroutine just delays for a fixed period, then toggles
    // an LED. Two coroutines are created using this function, so
    // the uxIndex parameter is used to tell the coroutine which
    // LED to flash and how long to delay. This assumes xQueue has
    // already been created.
    vParTestToggleLED(cLedToFlash[uxIndex]);
    crDELAY(xHandle, uxFlashRates[uxIndex]);
  }
  // Must end every coroutine with a call to crEND();
  crEND();
}

// Function that creates two coroutines.
```

```
void vOtherFunction(void)
{
  unsigned char ucParameterToPass;
  xTaskHandle xHandle;
  // Create two coroutines at priority 0. The first is given index 0
  // so (from the code above) toggles LED 5 every 200 ticks. The second
  // is given index 1 so toggles LED 6 every 400 ticks.
  for (uxIndex = 0; uxIndex < 2; uxIndex++)
  {
    xCoRoutineCreate(vFlashCoRoutine, 0, uxIndex);
  }
}
```

`xCoRoutineHandle`

Type by which coroutines are referenced. The coroutine handle is automatically passed into each coroutine function.

3.4.7.3 `crDELAY`

Prototype

```
void crDELAY(xCoRoutineHandle xHandle,
             portTickType xTicksToDelay)
```

Semantics

`crDELAY` is a macro.

The data types in the prototype above are shown for reference only. Delay a coroutine for a fixed period of time.

`crDELAY` can only be called from the coroutine function itself - not from within a function called by the coroutine function. This is because coroutines do not maintain their own stack.

Parameters

`xHandle` The handle of the coroutine to delay. This is the `xHandle` parameter of the coroutine function.

`xTickToDelay` The number of ticks that the coroutine should delay for. The actual amount of time this equates to is defined by `configTICK_RATE_HZ` (set in `FreeRTOSConfig.h`). The constant `portTICK_RATE_MS` can be used to convert ticks to milliseconds.

Example Usage

```
// Co-routine to be created.
void vACoRoutine(xCoRoutineHandle xHandle,
                 unsigned portBASE_TYPE uxIndex)
{
  // Variables in coroutines must be declared static
  // if they must maintain value across a blocking call.
  // This may not be necessary for const variables.
  // We are to delay for 200ms.
  static const xTickType xDelayTime = 200 / portTICK_RATE_MS;
  // Must start every coroutine with a call to crSTART();
  crSTART(xHandle);
  for (;;)
  {
    // Delay for 200ms.
    crDELAY(xHandle, xDelayTime);
    // Do something here.
  }
  // Must end every coroutine with a call to crEND();
  crEND();
}
```

3.4.7.4 `crQUEUE_SEND`

Prototype

```
crQUEUE_SEND(xCoRoutineHandle xHandle,
             xQueueHandle pxQueue,
             void *pvItemToQueue,
             portTickType xTicksToWait,
             portBASE_TYPE *pxResult)
```

Semantics

`crQUEUE_SEND` is a macro. The data types are shown in the prototype above for reference only.

The macro's `crQUEUE_SEND()` and `crQUEUE_RECEIVE()` are the coroutine equivalent to the `xQueueSend()` and `xQueueReceive()` functions used by tasks.

`crQUEUE_SEND` and `crQUEUE_RECEIVE` can only be used from a coroutine whereas `xQueueSend()` and `xQueueReceive()` can only be used from tasks. Note that coroutines can only send data to other coroutines. A coroutine cannot use a queue to send data to a task or visa versa.

crQUEUE_SEND can only be called from the coroutine function itself - not from within a function called by the coroutine function. This is because coroutines do not maintain their own stack.

See the coroutine section of the WEB documentation for information on passing data between tasks and coroutines and between ISR's and coroutines.

Parameters

xHandle The handle of the calling coroutine. This is the xHandle parameter of the coroutine function.

pxQueue The handle of the queue on which the data will be posted. The handle is obtained as the return value when the queue is created using the xQueueCreate() API function.

pvItemToQueue A pointer to the data being posted onto the queue. The number of bytes of each queued item is specified when the queue is created. This number of bytes is copied from pvItemToQueue into the queue itself.

xTickToWait The number of ticks that the coroutine should block to wait for space to become available on the queue, should space not be available immediately. The actual amount of time this equates to is defined by configTICK_RATE_HZ (set in FreeRTOSConfig.h). The constant portTICK_RATE_MS can be used to convert ticks to milliseconds (see example below).

pxResult The variable pointed to by pxResult will be set to pdPASS if data was successfully posted onto the queue, otherwise it will be set to an error defined within ProjDefs.h.

Example Usage

```
// Co-routine function that blocks for a fixed period then posts a number onto
// a queue.
static void prvCoRoutineFlashTask(xCoRoutineHandle xHandle,
                                  unsigned portBASE_TYPE uxIndex)
{
  // Variables in coroutines must be declared static if they must
  // maintain value across a blocking call. static portBASE_TYPE
  xNumberToPost = 0;
  static portBASE_TYPE xResult;

  // Co-routines must begin with a call to crSTART().
  crSTART(xHandle);
  for (;;)
  {
    // This assumes the queue has already been created.
    crQUEUE_SEND(xHandle, xCoRoutineQueue, &xNumberToPost, NO_DELAY, &xResult);
    if (xResult != pdPASS)
    {
```

```
      // The message was not posted!
    }
    // Increment the number to be posted onto the queue.
    xNumberToPost++;
    // Delay for 100 ticks.
    crDELAY(xHandle, 100);
  }
  // Co-routines must end with a call to crEND().
  crEND();
}
```

3.4.7.5 crQUEUE_RECEIVE

Prototype

```
void crQUEUE_RECEIVE(xCoRoutineHandle xHandle,
                     xQueueHandle pxQueue,
                     void *pvBuffer,
                     portTickType xTicksToWait,
                     portBASE_TYPE *pxResult)
```

Semantics

`crQUEUE_RECEIVE` is a macro. The data types are shown in the prototype above for reference only.

The macro's `crQUEUE_SEND()` and `crQUEUE_RECEIVE()` are the coroutine equivalent to the `xQueueSend()` and `xQueueReceive()` functions used by tasks.

`crQUEUE_SEND` and `crQUEUE_RECEIVE` can only be used from a coroutine whereas `xQueueSend()` and `xQueueReceive()` can only be used from tasks. Note that coroutines can only send data to other coroutines.

A coroutine cannot use a queue to send data to a task or visa versa. `crQUEUE_RECEIVE` can only be called from the coroutine function itself - not from within a function called by the coroutine function. This is because coroutines do not maintain their own stack.

See the coroutine section of the WEB documentation for information on passing data between tasks and coroutines and between ISR's and coroutines.

Parameters

`xHandle` The handle of the calling coroutine. This is the `xHandle` parameter of the coroutine function.

`pxQueue` The handle of the queue from which the data will be received. The handle is obtained as the return value when the queue is created using the `xQueueCreate()` API function.

`pvBuffer` The buffer into which the received item is to be copied. The number of bytes of each queued item is specified when the queue is created. This number of bytes is copied into pvBuffer.

`xTickToWait` The number of ticks that the coroutine should block to wait for data to become available from the queue, should data not be available immediately. The actual amount of time this equates to is defined by `configTICK_RATE_HZ` (set in `FreeRTOSConfig.h`). The constant `portTICK_RATE_MS` can be used to convert ticks to milliseconds (see the `crQUEUE_SEND` example).

`pxResult` The variable pointed to by `pxResult` will be set to `pdPASS` if data was successfully retrieved from the queue, otherwise it will be set to an error code as defined within `ProjDefs.h`.

Example Usage

```
// A coroutine receives the number of an LED to flash from a queue. It
// blocks on the queue until the number is received.
static void prvCoRoutineFlashWorkTask(xCoRoutineHandle xHandle,
                                      unsigned portBASE_TYPE uxIndex)
{
  // Variables in coroutines must be declared static if they must
  // maintain value across a blocking call. static portBASE_TYPE xResult;
  static unsigned portBASE_TYPE uxLEDToFlash;

  // All coroutines must start with a call to crSTART().
  crSTART(xHandle);
  for (;;)
  {
    // Wait for data to become available on the queue.
    crQUEUE_RECEIVE(xHandle,
                    xCoRoutineQueue,
                    &uxLEDToFlash,
                    portMAX_DELAY,
                    &xResult);
    if (xResult == pdPASS)
    {
      // We received the LED to flash - flash it!
      vParTestToggleLED(uxLEDToFlash);
    }
  }
  crEND();
}
```

3.4.7.6 crQUEUE_SEND_FROM_ISR

Prototype

```
portBASE_TYPE crQUEUE_SEND_FROM_ISR(xQueueHandle pxQueue,
                                    void *pvItemToQueue,
                                    portBASE_TYPE xCoRoutinePreviouslyWoken)
```

Semantics

`crQUEUE_SEND_FROM_ISR()` is a macro. The data types are shown in the prototype above for reference only.

The macro's `crQUEUE_SEND_FROM_ISR()` and `crQUEUE_RECEIVE_FROM_ISR()` are the coroutine equivalent to the `xQueueSendFromISR()` and `xQueueReceiveFromISR()` functions used by tasks.

`crQUEUE_SEND_FROM_ISR()` and `crQUEUE_RECEIVE_FROM_ISR()` can only be used to pass data between a coroutine and and ISR, whereas `xQueueSendFromISR()` and `xQueueReceiveFromISR()` can only be used to pass data between a task and and ISR.

`crQUEUE_SEND_FROM_ISR` can only be called from an ISR to send data to a queue that is being used from within a coroutine.

See the coroutine section of the WEB documentation for information on passing data between tasks and coroutines and between ISR's and coroutines.

Parameters

`xQueue` The handle to the queue on which the item is to be posted. pvItemToQueue A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from `pvItemToQueue` into the queue storage area.

`xCoRoutinePreviouslyWoken` This is included so an ISR can post onto the same queue multiple times from a single interrupt. The first call should always pass in `pdFALSE`. Subsequent calls should pass in the value returned from the previous call.

Returns

`pdTRUE` if a coroutine was woken by posting onto the queue. This is used by the ISR to determine if a context switch may be required following the ISR.

Example Usage

```
// A coroutine that blocks on a queue waiting for characters to be
// received.
```

```
static void vReceivingCoRoutine(xCoRoutineHandle xHandle,
                                unsigned portBASE_TYPE uxIndex)
{
  portCHAR cRxedChar;
  portBASE_TYPE xResult;
  // All coroutines must start with a call to crSTART().
  crSTART(xHandle);
  for (;;)
  {
    // Wait for data to become available on the queue. This assumes the
    // queue xCommsRxQueue has already been created!
    crQUEUE_RECEIVE(xHandle,
                    xCommsRxQueue,
                    &uxLEDToFlash,
                    portMAX_DELAY,
                    &xResult);
    // Was a character received?
    if (xResult == pdPASS)
    {
      // Process the character here.
    }
  }
  // All coroutines must end with a call to crEND().
  crEND();
}

// An ISR that uses a queue to send characters received on a serial port to
// a coroutine.
void vUART_ISR(void)
{
  portCHAR cRxedChar;
  portBASE_TYPE
  xCRWokenByPost = pdFALSE;

  // We loop around reading characters until there are none left
  // in the UART.
  while (UART_RX_REG_NOT_EMPTY())
  {
    // Obtain the character from the UART.
    cRxedChar = UART_RX_REG;
    // Post the character onto a queue. xCRWokenByPost will be pdFALSE
    // the first time around the loop. If the post causes a coroutine
    // to be woken (unblocked) then xCRWokenByPost will be set to pdTRUE.
    // In this manner we can ensure that if more than one coroutine is
    // blocked on the queue only one is woken by this ISR no matter how
    // many characters are posted to the queue.
    xCRWokenByPost = crQUEUE_SEND_FROM_ISR(xCommsRxQueue,
                                           &cRxedChar,
                                           xCRWokenByPost);
  }
}
```

3.4.7.7 crQUEUE_RECEIVE_FROM_ISR

Prototype

```
portBASE_TYPE crQUEUE_SEND_FROM_ISR(xQueueHandle pxQueue,
                                    void *pvBuffer,
                                    portBASE_TYPE * pxCoRoutineWoken)
```

Semantics

The macro's `crQUEUE_SEND_FROM_ISR()` and `crQUEUE_RECEIVE_FROM_ISR()` are the coroutine equivalent to the `xQueueSendFromISR()` and `xQueueReceiveFromISR()` functions used by tasks.

`crQUEUE_SEND_FROM_ISR()` and `crQUEUE_RECEIVE_FROM_ISR()` can only be used to pass data between a coroutine and and ISR, whereas `xQueueSendFromISR()` and `xQueueReceiveFromISR()` can only be used to pass data between a task and and ISR.

`crQUEUE_RECEIVE_FROM_ISR` can only be called from an ISR to receive data from a queue that is being used from within a coroutine (a coroutine posted to the queue).

See the coroutine section of the WEB documentation for information on passing data between tasks and coroutines and between ISR's and coroutines.

Parameters

`xQueue` The handle to the queue on which the item is to be posted.

`pvBuffer` A pointer to a buffer into which the received item will be placed. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from the queue into `pvBuffer`.

`pxCoRoutineWoken` A coroutine may be blocked waiting for space to become available on the queue. If `crQUEUE_RECEIVE_FROM_ISR` causes such a coroutine to unblock `*pxCoRoutineWoken` will get set to `pdTRUE`, otherwise `*pxCoRoutineWoken` will remain unchanged

Returns

`pdTRUE` an item was successfully received from the queue, otherwise `pdFALSE`.

Example Usage

```
// A coroutine that posts a character to a queue then blocks for a fixed
// period. The character is incremented each time.
static void vSendingCoRoutine(xCoRoutineHandle xHandle,
                              unsigned portBASE_TYPE uxIndex)
{
```

```
  // cChar holds its value while this coroutine is blocked and must therefore
  // be declared static.
  static portCHAR cCharToTx = 'a';
  portBASE_TYPE xResult;
  // All coroutines must start with a call to crSTART().
  crSTART(xHandle);
  for (;;)
  {
    // Send the next character to the queue.
    crQUEUE_SEND(xHandle, xCoRoutineQueue, &cCharToTx, NO_DELAY, &xResult);
    if (xResult == pdPASS)
    {
      // The character was successfully posted to the queue.
    }
    else
    {
      // Could not post the character to the queue.
    }
    // Enable the UART Tx interrupt to cause an interrupt in this
    // hypothetical UART. The interrupt will obtain the character
    // from the queue and send it.
    ENABLE_RX_INTERRUPT();
    // Increment to the next character then block for a fixed period.
    // cCharToTx will maintain its value across the delay as it is
    // declared static.
    cCharToTx++;
    if (cCharToTx > 'x')
    {
      cCharToTx = 'a';
    }
    crDELAY(100);
  }
  // All coroutines must end with a call to crEND().
  crEND();
}

// An ISR that uses a queue to receive characters to send on a UART.
void vUART_ISR(void)
{
  portCHAR cCharToTx;
  portBASE_TYPE
  xCRWokenByPost = pdFALSE;
  while (UART_TX_REG_EMPTY())
  {
    // Are there any characters in the queue waiting to be sent?
    // xCRWokenByPost will automatically be set to pdTRUE if a coroutine
    // is woken by the post - ensuring that only a single coroutine is
    // woken no matter how many times we go around this loop.
    if (crQUEUE_RECEIVE_FROM_ISR(pxQueue, &cCharToTx, &xCRWokenByPost))
    {
      SEND_CHARACTER(cCharToTx);
    }
  }
}
```

3.4.7.8 `vCoRoutineSchedule`

Prototype

```
void vCoRoutineSchedule(void);
```

Semantics

Run a coroutine.

`vCoRoutineSchedule()` executes the highest priority coroutine that is able to run. The coroutine will execute until it either blocks, yields or is preempted by a task. Co-routines execute cooperatively so one coroutine cannot be preempted by another, but can be preempted by a task.

If an application comprises of both tasks and coroutines then `vCoRoutine-Schedule` should be called from the idle task (in an idle task hook).

Example Usage

```
void vApplicationIdleHook(void)
{
  vCoRoutineSchedule(void);
}
```

Alternatively, if the idle task is not performing any other function it would be more efficient to call `vCoRoutineSchedule()` from within a loop as:

```
void vApplicationIdleHook(void)
{
  for (;;)
  {
    vCoRoutineSchedule(void);
  }
}
```

Chapter 4

FreeRTOS Implementation and Source Code Analysis

4.1 General Features

FreeRTOS is a free, embedded RTOS has been made available by [Bar07]. This RTOS claims to be a portable, open source, mini real-time kernel that can be operated in preemptive or cooperative. Some of the main features of FreeRTOS are listed below:

- 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.
- 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 C processes pass control to one another by yielding. The scheduler interrupts at regular frequency simply to increment the tick count.
- Dynamic scheduling: Scheduler decision points occur at regular clock frequency. Asynchronous events (other than the scheduler) also invoke scheduler decisions points.
- 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.
- Inter-Process Communication: Tasks within **FreeRTOS** can communicate with each other through the use of queuing and synchronization mechanisms:

  - 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.
  - 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.

- 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.

- 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 bit in the condition code register (CCR).

- 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 preempted by a scheduling event while at the same time continuing to service interrupts.

- 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.

- Priority Inversion: **FreeRTOS** does not implement any advanced techniques (such as priority inheritance or priority ceilings) to deal with priority inversion.

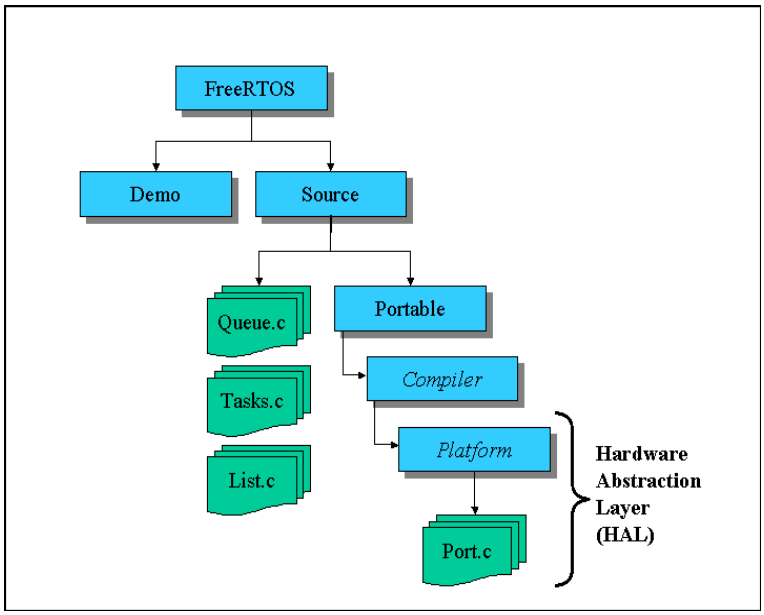

Figure 4.1: **FreeRTOS** Source Distribution

4.2 Source Code Distribution and Organization

FreeRTOS is distributed in a code tree as shown in Figure 4.1. **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).

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.

4.2.1 Basic Directory Structure

The **FreeRTOS** download includes source code for every processor port, and every demo application. Placing all the ports in a single download greatly simplifies distribution, but the number of files may seem daunting. The directory structure is however very simple, and the **FreeRTOS** real time kernel is con-

tained in just 3 files (4 if coroutines are used).

From the top, the download is split into two sub directories:

```
FreeRTOS
    |
    +- Demo Contains the demo application.
    |
    +- Source Contains the real time kernel source code.
```

The majority of the real time kernel code is contained in three files that are common to every processor architecture (four if coroutines are used). These files, `tasks.c`, `queue.c` and `list.c`, are in the source directory.

Each processor architecture requires a small amount of kernel code specific to that architecture. The processor specific code is contained in a directory called `Portable`, under the `Source` directory.

The download also contains a demo application for every processor architecture and compiler port. The majority of the demo application code is common to all ports and is contained in a directory called Common, under the Demo directory. The remaining sub directories under `Demo` contain build files for building the demo for that particular port.

```
FreeRTOS
    |
    +- Demo
    |    |
    |    +- Common The demo application files that are used by all the ports.
    |    +- Dir x The demo application build files for port x
    |    +- Dir y The demo application build files for port y
    |
    +- Source
         |
         +- Portable Processor specific code.
```

The following subsections provide more details of the `Demo` and `Source` directories.

4.2.2 RTOS Source Code Directory List

[the `Source` directory]

To use **FreeRTOS** you need to include the real time kernel source files in your `makefile`. It is not necessary to modify them or understand their implementation.

The real time kernel source code consists of three files that are common to all micro-controller ports (four if coroutines are used), and a single file that is specific to the port you are using. The common files can be found in the `FreeRTOS/Source` directory. The port specific files can be found in subdirectories contained in the `FreeRTOS/Source/Portable` directory.

For example:

- If using the MSP430 port with the GCC compiler: The MSP430 specific file (`port.c`) can be found in the `FreeRTOS/Source/Portable/GCC/MSP-430F449` directory, and all the other sub directories in the `FreeRTOS/Source/Portable` directory relate to other microcontroller ports and can be ignored.
- If using the PIC18 port with the MPLAB compiler: The PIC18 specific file (`port.c`) can be found in the `FreeRTOS/Source/Portable/MPLAB/PIC18` directory, and all the other sub directories in the `FreeRTOS/Source/Portable` directory relate to other micro-controller ports and can be ignored.
- And so on for all the ports ... `FreeRTOS/Portable/MemMang` contains the sample memory allocators as described in the memory management section.

4.2.3 Demo Application Source Code Directory List

[the `Demo` directory]

The Demo directory tree contains a demo application for each port. Most of the code for the demo application is common to every port. The code that is common to every port is located in the `FreeRTOS/Demo/Common` directory. See the demo application section for more details. Port specific code, including the demo application project files, can be found in subdirectories contained in the `FreeRTOS/Demo` directory.

For example:

- If building the MSP430 GCC demo application: The MSP430 demo application `makefile` can be found in the `FreeRTOS/Demo/MSP430` directory. All the other sub directories contained in the `FreeRTOS/Demo` directory (other than the `Common` directory) relate to demo application's targeted at other micro-controllers and can be ignored.
- If building the PIC18 MPLAB demo application: The PIC18 demo application project file can be found in the `FreeRTOS/Demo/PIC18_MPLAB` directory. All the other sub directories contained in the `FreeRTOS/Demo` directory (other than the Common directory) relate to demo application's targeted at other micro-controllers and can be ignored.
- And so on for all the ports ...

4.2.4 Creating Your Own Application

When writing your own application it is preferable to use the demo application `makefile` (or project file) as a starting point. You can leave all the files included from the Source directory included in the `makefile`, and replace the files included from the Demo directory with those for your own application. This will ensure both the RTOS source files included in the `makefile` and the compiler switches used in the `makefile` are both correct.

4.2.5 Naming Conventions

The RTOS kernel and demo application source code use the following conventions:

- Variables
  - Variables of type char are prefixed c
  - Variables of type short are prefixed s
  - Variables of type long are prefixed l
  - Variables of type float are prefixed f
  - Variables of type double are prefixed d
  - Enumerated variables are prefixed e
  - Other types (e.g. structs) are prefixed x
  - Pointers have an additional prefixed p, for example a pointer to a short will have prefix ps
  - Unsigned variables have an additional prefixed u, for example an unsigned short will have prefix us
- Functions
  - File private functions are prefixed with `prv`
  - API functions are prefixed with their return type, as per the convention defined for variables
  - Function names start with the file in which they are defined. For example `vTaskDelete` is defined in `Task.c`

4.2.6 Data Types

Data types are not directly referenced within the RTOS kernel source code. Instead each port has it's own set of definitions. For example, the char type is `#defined` to portCHAR, the short data type is `#defined` to portSHORT, etc. The demo application source code also uses this notation - but this is not necessary and your application can use whatever notation you prefer. In addition there are two other types that are defined for each port. These are:

- `portTickType`: This is a configurable as either an unsigned 16bit type or an unsigned 32bit type. See the customisation section of the API documentation for full information.
- `portBASE_TYPE`: This is defined for each port to be the most efficient type for that particular architecture.

If `portBASE_TYPE` is define to char then particular care must be taken to ensure signed chars are used for function return values that can be negative to indicate an error.

The control algorithm is reliant on accurate timing, it is therefore paramount that these timing requirements are met.

4.2.7 Local Operator Interface [Keypad and LCD]

The keypad and LCD can be used by the operator to select, view and modify system data. The operator interface shall function while the plant is being controlled.

To ensure no key presses are missed the keypad shall be scanned at least every 15ms. The LCD shall update within 50ms of a key being pressed.

4.2.8 LED

The LED shall be used to indicate the system status. A flashing green LED shall indicate that the system is running as expected. A flashing red LED shall indicate a fault condition.

The correct LED shall flash on and off once ever second. This flash rate shall be maintained to within 50ms.

4.2.9 RS232 PDA Interface

The PDA RS232 interface shall be capable of viewing and accessing the same data as the local operator interface, and the same timing constraints apply - discounting any data transmission times.

4.2.10 TCP/IP Interface

The embedded WEB server shall service HTTP requests within one second.

4.2.11 Application Components

The timing requirements of the hypothetical system can be split into three categories:

1. Strict timing - the plant control. The control function has a very strict timing requirement as it must execute every 10ms.

2. Flexible timing - the LED. While the LED outputs have both maximum and minimum time constraints, there is a large timing band within which they can function.

3. Deadline only timing - the human interfaces. This includes the keypad, LCD, RS232 and TCP/IP Ethernet communications.

   The human interface functions have a different type of timing requirement as only a maximum limit is specified. For example, the keypad must be scanned at least every 10ms, but any rate up to 10ms is acceptable.

4.2.12 More Info

The best way to learn about the real time kernel is to use the demo application and read the API documentation.

A demo application is provided for each microcontroller. If you don't have any hardware available then the PC port will execute under Windows, and the Keil ARM7 port will run entirely in the Keil simulator.

4.2.13 RTOS Demo Introduction

The RTOS source code download includes a demonstration project for each port. The sample projects are preconfigured to execute on the single board computer or prototyping board used during the port development. Each should build directly as downloaded without any warnings or errors.

The demonstration projects are provided as:

1. An aid to learning how to use **FreeRTOS** - each source file demonstrates a component of the RTOS.

2. A preconfigured starting point for new applications - to ensure the correct development tool setup (compiler switches, debugger format, etc) it is recommended that new applications are created by modifying the existing demo projects.

The table below lists the files that make up the demo projects along with a brief indication of the RTOS features demonstrated. The following page describes each task and coroutine within the demo project in more detail.

Two implementations are provided for the majority files listed below. The files contained in the `Demo/Common/Minimal` directory are for more RAM challenged systems such as the AVR. These files do not contain console IO. The files contained in the `Demo/Common/Full` directory are predominantly for the x86 demo projects and contain console IO. Other than that the functionality of the two implementations are basically the same. See the Source Code Organization section for more information on the demo project directory structure.

A few of points to note:

- Not all the Demo/Common files are used in every demonstration project. How many files are used depends on processor resources.

- The demo projects often use all the available RAM on the target processor. This means that you cannot add more tasks to the project without first removing some! This is especially true for the projects configured to run on the low end 8bit processors.

- In addition to the standard demo projects, two embedded WEB server projects are included in the download. These provide a more application orientated example.

- Each demo project also contains a file called `main.c` which contains the `main()` function. This function is responsible for creating all the demo application tasks and then starting the real time kernel.
- The standard demo project files are provided for the purpose of demonstrating the use of the real time kernel and are not intended to provide an optimized solution. This is particularly true of `comtest.c`.

The demo application does not free all it's resources when it exits, although the kernel does. This has been done purely to minimize lines of code.

4.2.13.1 Demo Project Files

This page describes the functionality of the files within the standard RTOS demo projects. The descriptions relate to the files in the Demo/Common/Full directory. Their equivalents in the Demo/Common/Minimal directory will have similar functionality but use less RAM and not contain any console IO.

4.2.13.2 `blockQ.c`

Creates six tasks that operate on three queues as follows:

The first two tasks send and receive an incrementing number to/from a queue. One task acts as a producer and the other as the consumer. The consumer is a higher priority than the producer and is set to block on queue reads. The queue only has space for one item - as soon as the producer posts a message on the queue the consumer will unblock, pre-empt the producer, and remove the item.

The second two tasks work the other way around. Again the queue used only has enough space for one item. This time the consumer has a lower priority than the producer. The producer will try to post on the queue blocking when the queue is full. When the consumer wakes it will remove the item from the queue, causing the producer to unblock, pre-empt the consumer, and immediately re-fill the queue.

The last two tasks use the same queue producer and consumer functions. This time the queue has enough space for lots of items and the tasks operate at the same priority. The producer will execute, placing items into the queue. The consumer will start executing when either the queue becomes full (causing the producer to block) or a context switch occurs (tasks of the same priority will time slice).

4.2.13.3 `comtest.c`

Creates two tasks that operate on an interrupt driven serial port. A loopback connector should be used so that everything that is transmitted is also received. The serial port does not use any flow control. On a standard 9 way 'D' connector pins two and three should be connected together. The first task repeatedly sends a string to a queue, character at a time. The serial port interrupt will empty the

queue and transmit the characters. The task blocks for a pseudo random period before resending the string. The second task blocks on a queue waiting for a character to be received. Characters received by the serial port interrupt routine are posted onto the queue - unblocking the task making it ready to execute. If this is then the highest priority task ready to run it will run immediately - with a context switch occurring at the end of the interrupt service routine. The task receiving characters is spawned with a higher priority than the task transmitting the characters. With the loop back connector in place, one task will transmit a string and the other will immediately receive it. The receiving task knows the string it expects to receive so can detect an error. This also creates a third task. This is used to test semaphore usage from an ISR and does nothing interesting.

4.2.13.4 `crflash.c`

This demo application file demonstrates the use of queues to pass data between coroutines and the use of the coroutine index parameter. N 'fixed delay' coroutines are created that just block for a fixed period then post the number of an LED onto a queue. Each such coroutine uses its index parameter as an index into array in order to obtain the block period and LED that is flashed. A single 'flash' coroutine is also created that blocks on the same queue, waiting for the number of the next LED it should flash. Upon receiving a number it simply toggle the instructed LED then blocks on the queue once more. In this manner each LED from LED 0 to LED N-1 is caused to flash at a different rate. The 'fixed delay' coroutines are created with coroutine priority 0. The flash coroutine is created with coroutine priority 1. This means that the queue should never contain more than a single item. This is because posting to the queue will unblock the higher priority 'flash' coroutine which will only block again when the queue is empty. An error is indicated if an attempt to post data to the queue fails - indicating that the queue is already full.

4.2.13.5 `crhook.c`

This demo file demonstrates how to send data between an ISR and a coroutine. A tick hook function is used to periodically pass data between the RTOS tick and a set of 'hook' coroutines. `hookNUM_HOOK_CO_ROUTINES` coroutines are created. Each coroutine blocks to wait for a character to be received on a queue from the tick ISR, checks to ensure the character received was that expected, then sends the number back to the tick ISR on a different queue. The tick ISR checks the numbers received back from the 'hook' coroutines matches the number previously sent. If at any time a queue function returns unexpectedly, or an incorrect value is received either by the tick hook or a coroutine then an error is latched. This demo relies on each 'hook' coroutine to execute between each `hookTICK_CALLS_BEFORE_POST` tick interrupts. This and the heavy use of queues from within an interrupt may result in an error being detected on slower targets simply due to timing.

4.2.13.6 `death.c`

Create a single persistent task which periodically dynamically creates another four tasks. The original task is called the creator task, the four tasks it creates are called suicidal tasks. Two of the created suicidal tasks kill one other suicidal task before killing themselves - leaving just the original task remaining. The creator task must be spawned after all of the other demo application tasks as it keeps a check on the number of tasks under the scheduler control. The number of tasks it expects to see running should never be greater than the number of tasks that were in existence when the creator task was spawned, plus one set of four suicidal tasks. If this number is exceeded an error is flagged.

4.2.13.7 `dynamic.c`

The first test creates three tasks - two counter tasks (one continuous count and one limited count) and one controller. A "count" variable is shared between all three tasks. The two counter tasks should never be in a "ready" state at the same time. The controller task runs at the same priority as the continuous count task, and at a lower priority than the limited count task. One counter task loops indefinitely, incrementing the shared count variable on each iteration. To ensure it has exclusive access to the variable it raises it's priority above that of the controller task before each increment, lowering it again to it's original priority before starting the next iteration. The other counter task increments the shared count variable on each iteration of it's loop until the count has reached a limit of 0xff - at which point it suspends itself. It will not start a new loop until the controller task has made it "ready" again by calling `vTaskResume()`. This second counter task operates at a higher priority than controller task so does not need to worry about mutual exclusion of the counter variable. The controller task is in two sections. The first section controls and monitors the continuous count task. When this section is operational the limited count task is suspended. Likewise, the second section controls and monitors the limited count task. When this section is operational the continuous count task is suspended. In the first section the controller task first takes a copy of the shared count variable. To ensure mutual exclusion on the count variable it suspends the continuous count task, resuming it again when the copy has been taken. The controller task then sleeps for a fixed period - during which the continuous count task will execute and increment the shared variable. When the controller task wakes it checks that the continuous count task has executed by comparing the copy of the shared variable with its current value. This time, to ensure mutual exclusion, the scheduler itself is suspended with a call to `vTaskSuspendAll()`. This is for demonstration purposes only and is not a recommended technique due to its inefficiency. After a fixed number of iterations the controller task suspends the continuous count task, and moves on to its second section. At the start of the second section the shared variable is cleared to zero. The limited count task is then woken from it's suspension by a call to `vTaskResume()`. As this counter task operates at a higher priority than the controller task the

controller task should not run again until the shared variable has been counted up to the limited value causing the counter task to suspend itself. The next line after `vTaskResume()` is therefore a check on the shared variable to ensure everything is as expected. The second test consists of a couple of very simple tasks that post onto a queue while the scheduler is suspended. This test was added to test parts of the scheduler not exercised by the first test.

4.2.13.8 `flash.c`

Creates eight tasks, each of which flash an LED at a different rate. The first LED flashes every 125ms, the second every 250ms, the third every 375ms, etc. The LED flash tasks provide instant visual feedback. They show that the scheduler is still operational. The PC port uses the standard parallel port for outputs, the Flashlite 186 port uses IO port F.

4.2.13.9 `flop.c`

Creates eight tasks, each of which loops continuously performing an (emulated) floating point calculation. All the tasks run at the idle priority and never block or yield. This causes all eight tasks to time slice with the idle task. Running at the idle priority means that these tasks will get preempted any time another task is ready to run or a time slice occurs. More often than not the pre-emption will occur mid calculation, creating a good test of the schedulers context switch mechanism - a calculation producing an unexpected result could be a symptom of a corruption in the context of a task.

4.2.13.10 `integer.c`

This does the same as `flop.c`, but uses variables of type long instead of type double. As with `flop.c`, the tasks created in this file are a good test of the scheduler context switch mechanism. The processor has to access 32bit variables in two or four chunks (depending on the processor). The low priority of these tasks means there is a high probability that a context switch will occur mid calculation. See the `flop.c` documentation for more information.

4.2.13.11 `pollQ.c`

This is a very simple queue test. See the `BlockQ.c` documentation for a more comprehensive version. Creates two tasks that communicate over a single queue. One task acts as a producer, the other a consumer. The producer loops for three iteration, posting an incrementing number onto the queue each cycle. It then delays for a fixed period before doing exactly the same again. The consumer loops emptying the queue. Each item removed from the queue is checked to ensure it contains the expected value. When the queue is empty it blocks for a fixed period, then does the same again. All queue access is performed without blocking. The consumer completely empties the queue each time it runs so the

producer should never find the queue full. An error is flagged if the consumer obtains an unexpected value or the producer find the queue is full.

4.2.13.12 `print.c`

Manages a queue of strings that are waiting to be displayed. This is used to ensure mutual exclusion of console output. A task wishing to display a message will call `vPrintDisplayMessage()`, with a pointer to the string as the parameter. The pointer is posted onto the `xPrintQueue` queue. The task spawned in `main.c` blocks on `xPrintQueue`. When a message becomes available it calls `pcPrintGetNextMessage()` to obtain a pointer to the next string, then uses the functions defined in the portable layer `FileIO.c` to display the message.

NOTE: Using console IO can disrupt real time performance - depending on the port. Standard C IO routines are not designed for real time applications. While standard IO is useful for demonstration and debugging an alternative method should be used if you actually require console IO as part of your application.

4.2.13.13 `semtest.c`

Creates two sets of two tasks. The tasks within a set share a variable, access to which is guarded by a semaphore. Each task starts by attempting to obtain the semaphore. On obtaining a semaphore a task checks to ensure that the guarded variable has an expected value. It then clears the variable to zero before counting it back up to the expected value in increments of 1. After each increment the variable is checked to ensure it contains the value to which it was just set. When the starting value is again reached the task releases the semaphore giving the other task in the set a chance to do exactly the same thing. The starting value is high enough to ensure that a tick is likely to occur during the incrementing loop. An error is flagged if at any time during the process a shared variable is found to have a value other than that expected. Such an occurrence would suggest an error in the mutual exclusion mechanism by which access to the variable is restricted. The first set of two tasks poll their semaphore. The second set use blocking calls.

4.3 Task Management

4.3.1 Overview

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

4.3.2 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

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 the 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)

Table 4.1: Task Control Block for **FreeRTOS**

to completely describe the state of a task. The fields in the TCB for **FreeRTOS** are shown in Table 4.1 (derived from `tasks.c`).

4.3.3 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 4.2.

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 C 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

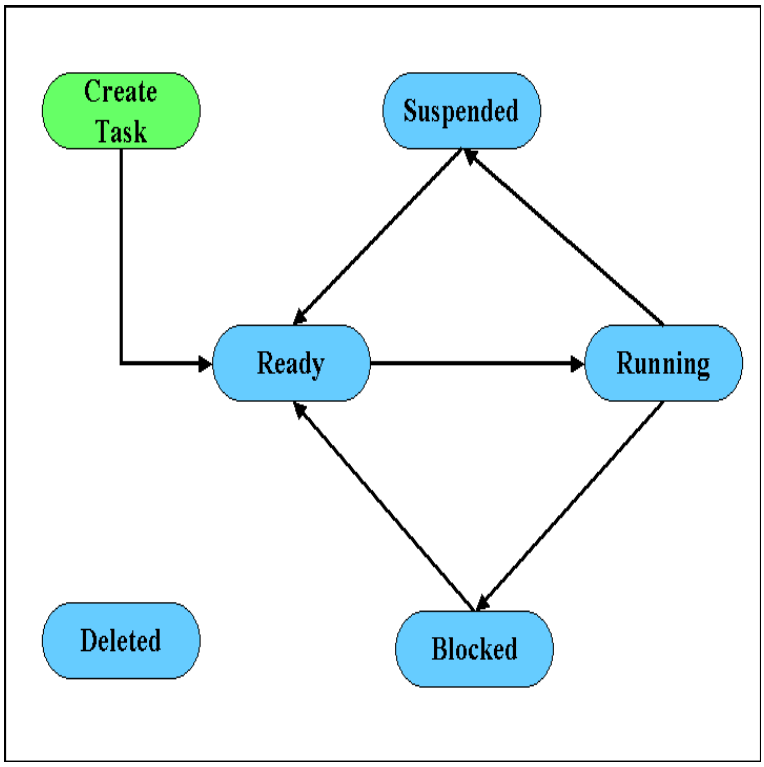

Figure 4.2: Basic Task State Diagram for **FreeRTOS**

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

Table 4.2: Lists Created by the Scheduler

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 life-cycle 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.

4.4 List Management

4.4.1 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.

4.4.2 Ready and Blocked Lists

Table 4.2 shows all of the lists that are created and used by the scheduler and their dependencies on configuration values in `FreeRTOSConfig.h`. Note that the {Ready} list is not a single list but actually `n` lists where `n` = `configMAX_PRIORITIES`.

`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 `xListValue` and therefore always appears at the end of the list

Table 4.3: Type `xList`

`ItemValue`	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 a normally TCB
`* pvContainer`	Pointer to the list in which this list item placed

Table 4.4: Type `xListItem`

Each of the lists in Table 4.2 is created as type `xList` which is a structure defined as shown in Table 4.3. 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 Table4.4 and 4.5.

4.4.3 List Initialization

Figure 4.3 shows an example is 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.

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`.

`ItemValue`	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.

Table 4.5: Type `xMiniListItem`

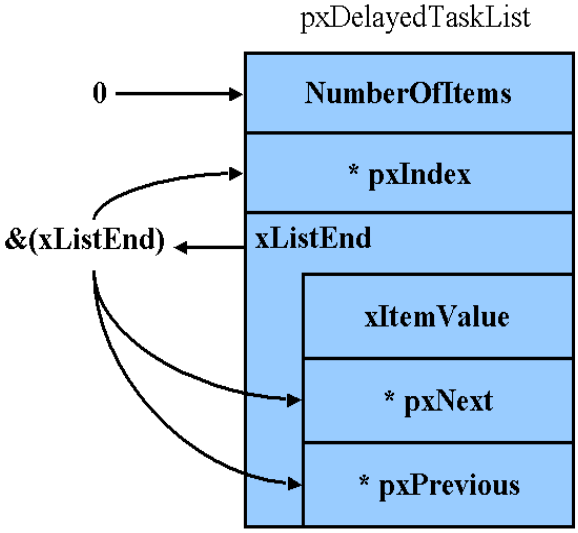

Figure 4.3: List Initialization

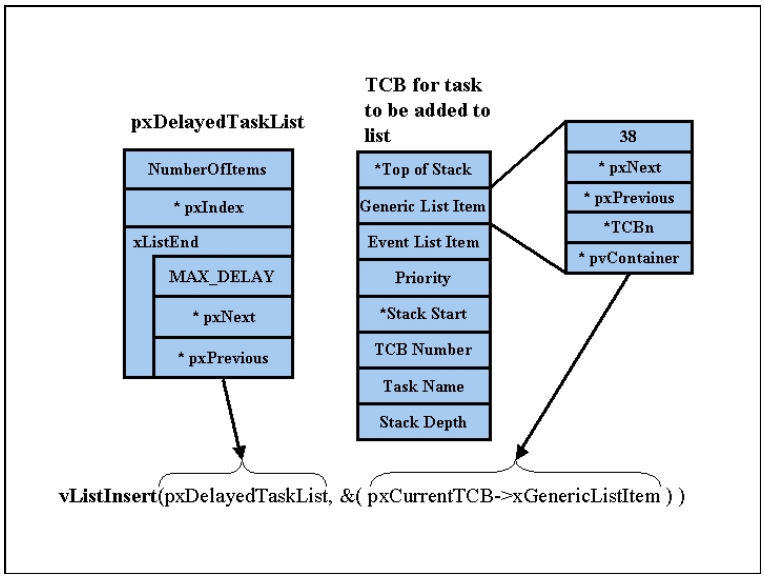

Figure 4.4: `vListInsert` with Arguments

4.4.4 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 4.4.

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 4.5 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` C it points to the first entry in the list as shown in the figure.

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.

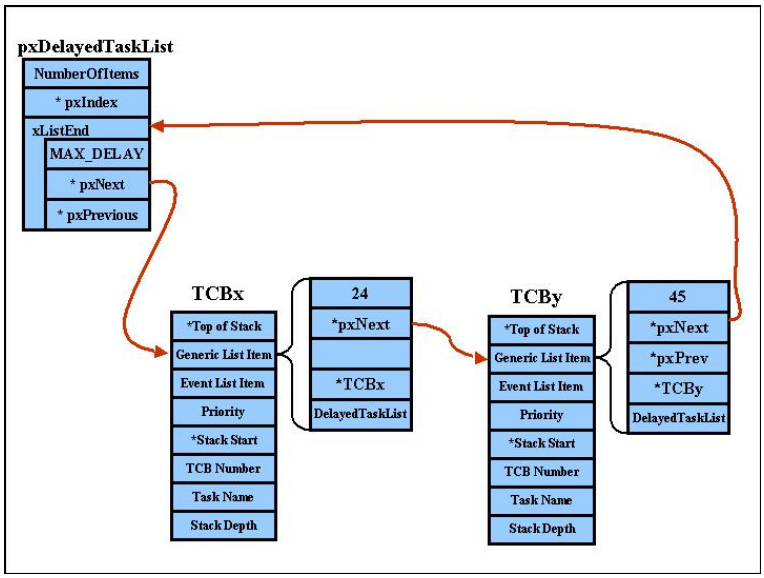

Figure 4.5: Hypothetical `DelayedTaskList`

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 4.6. Normally, code segments will not be presented within this document. However, in this case, the code is exceptionally concise and therefore worthy of presentation.

The for loop at point A initializes `pxIterator` (having type `xListItem`) to the last item in the list which is, by default, `xListEnd`. As mentioned, the `*pxNext` pointer of `xListEnd` 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 by assigning the task the second last place in the list.

4.4.5 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

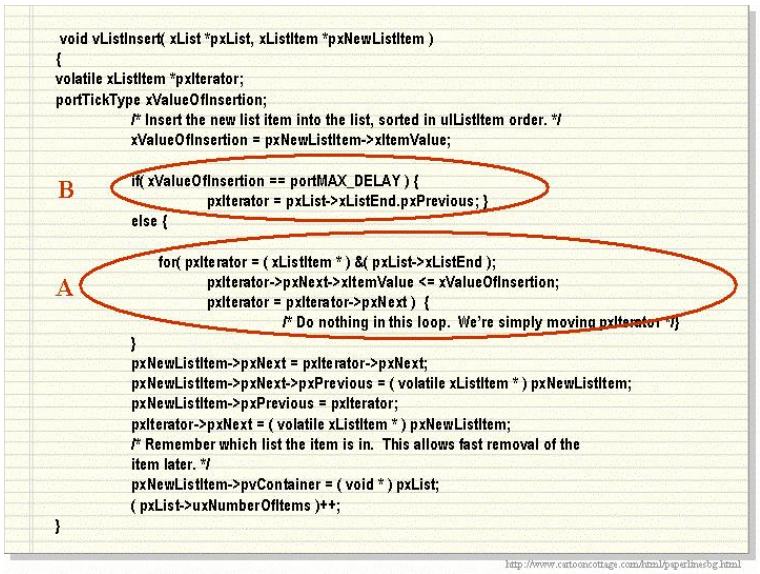

Figure 4.6: Code Extract from `Lists.c` in **FreeRTOS**

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 C resulting in a counter rollover after only 255 ticks. To deal with this, **FreeRTOS** defines and uses two delay lists C {`DelayedTaskList`} and {`OverflowDelayedTaskList`}.

As shown in Figure 4.7, 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`}.

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 4.8.

4.5 Context Switch

This section describes the RTOS context switch source code from the bottom up. The **FreeRTOS** Atmel AVR microcontroller port is used as an example. The section ends with a detailed step by step look at one complete context switch.

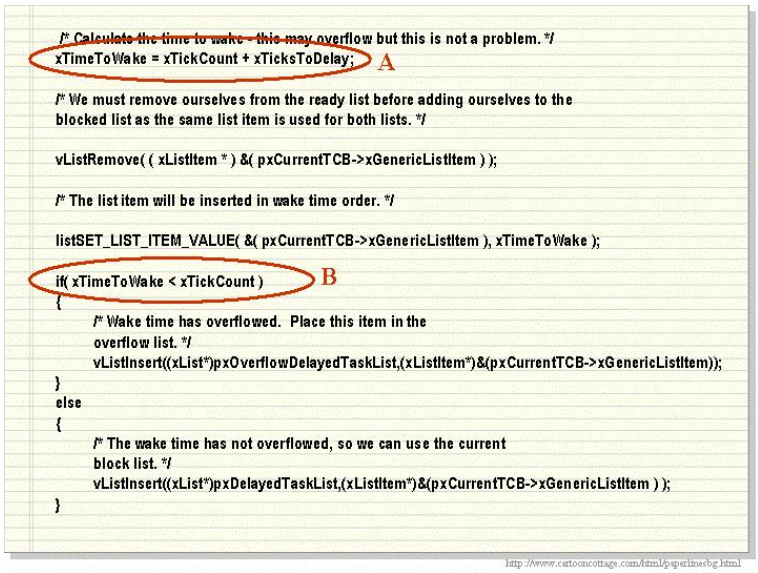

Figure 4.7: Deciding Which Delayed List To Insert (from `Task.c`)

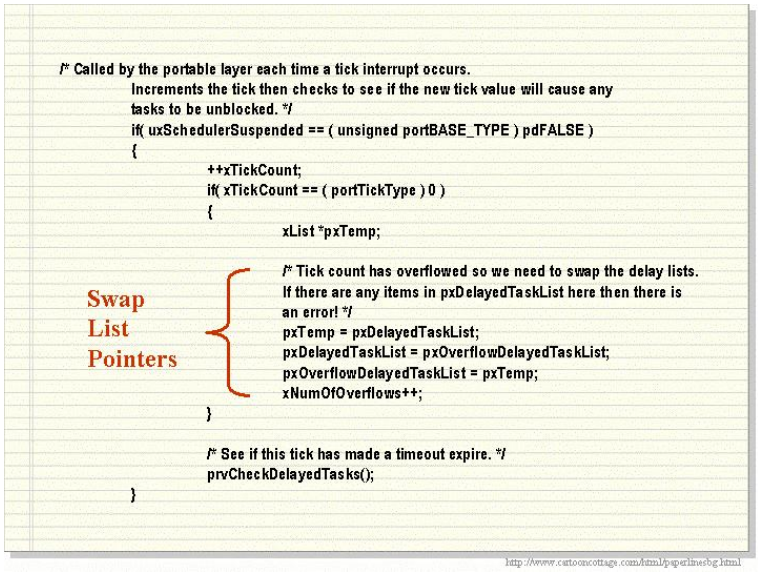

Figure 4.8: Exchanging List Pointers When Timer Overflows

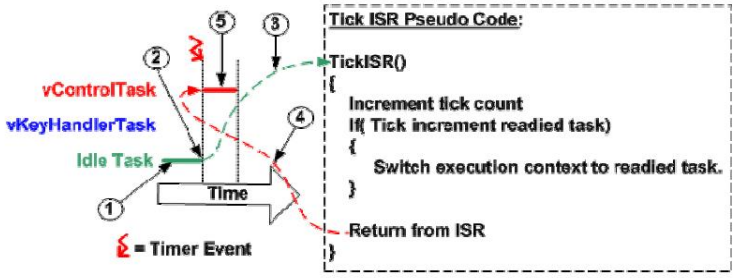

Figure 4.9: ISR for TICK

4.5.1 C Development Tools

A goal of **FreeRTOS** is that it is simple and easy to understand. To this end the majority of the RTOS source code is written in C, not assembler. The example presented here uses the WinAVR development tools. WinAVR is a free Windows to AVR cross compiler based on GCC.

4.5.2 The RTOS Tick

When sleeping, a task will specify a time after which it requires 'waking'. When blocking, a task can specify a maximum time it wishes to wait.

The **FreeRTOS** real time kernel measures time using a tick count variable. A timer interrupt (the RTOS tick interrupt) increments the tick count with strict temporal accuracy - allowing the real time kernel to measure time to a resolution of the chosen timer interrupt frequency.

Each time the tick count is incremented the real time kernel must check to see if it is now time to unblock or wake a task. It is possible that a task woken or unblocked during the tick ISR will have a priority higher than that of the interrupted task. If this is the case the tick ISR should return to the newly woken/unblocked task - effectively interrupting one task but returning to another. This is depicted below:

Referring to the numbers in Figure 4.9 above:

- At (1) the RTOS idle task is executing.
- At (2) the RTOS tick occurs, and control transfers to the tick ISR (3).
- The RTOS tick ISR makes vControlTask ready to run, and as vControlTask has a higher priority than the RTOS idle task, switches the context to that of vControlTask.
- As the execution context is now that of vControlTask, exiting the ISR (4) returns control to vControlTask, which starts executing (5).

A context switch occurring in this way is said to be Preemptive, as the interrupted task is preempted without suspending itself voluntarily.

The AVR port of **FreeRTOS** uses a compare match event on timer 1 to generate the RTOS tick. The following pages describe how the RTOS tick ISR is implemented using the **WinAVR** development tools.

4.5.3 GCC Signal Attribute

The GCC development tools allow interrupts to be written in **C**. A compare match event on the AVR timer 1 peripheral can be written using the following syntax.

```
void SIG_OUTPUT_COMPARE1A(void)__attribute__((signal));

void SIG_OUTPUT_COMPARE1A(void)
{
  /* ISR C code for RTOS tick. */
  vPortYieldFromTick();
}
```

The '`__attribute__((signal))`' directive on the function prototype informs the compiler that the function is an ISR and results in two important changes in the compiler output.

1. The 'signal' attribute ensures that every processor register that gets modified during the ISR is restored to its original value when the ISR exits. This is required as the compiler cannot make any assumptions as to when the interrupt will execute, and therefore cannot optimize which processor registers require saving and which don't.

2. The 'signal' attribute also forces a 'return from interrupt' instruction (`RETI`) to be used in place of the 'return' instruction (`RET`) that would otherwise be used. The AVR microcontroller disables interrupts upon entering an ISR and the `RETI` instruction is required to re-enable them on exiting.

Code output by the compiler:

```
;void SIG_OUTPUT_COMPARE1A(void)
;{
  ;--------------------------------------
  ; CODE GENERATED BY THE COMPILER TO SAVE
  ; THE REGISTERS THAT GET ALTERED BY THE
  ; APPLICATION CODE DURING THE ISR.
  PUSH R1
  PUSH R0
  IN R0,0x3F
  PUSH R0
  CLR R1
```

```
  PUSH R18
  PUSH R19
  PUSH R20
  PUSH R21
  PUSH R22
  PUSH R23
  PUSH R24
  PUSH R25
  PUSH R26
  PUSH R27
  PUSH R30
  PUSH R31
  ;--------------------------------------
  ; CODE GENERATED BY THE COMPILER FROM THE
  ; APPLICATION C CODE.
  ; vPortYieldFromTick()
  ; {
  ;  CALL 0x0000029B
  ;  Call subroutine
  ; }
  ;--------------------------------------
  ; CODE GENERATED BY THE COMPILER TO
  ; RESTORE THE REGISTERS PREVIOUSLY
  ; SAVED.
  POP R31
  POP R30
  POP R27
  POP R26
  POP R25
  POP R24
  POP R23
  POP R22
  POP R21
  POP R20
  POP R19
  POP R18
  POP R0
  OUT 0x3F, R0
  POP R0
  POP R1
  RETI
  ;--------------------------------------
;}
```

4.5.4 GCC Naked Attribute

The previous section showed how the 'signal' attribute can be used to write an ISR in C and how this results in part of the execution context being auto-

matically saved (only the processor registers modified by the ISR get saved). Performing a context switch however requires the entire context to be saved.

The application code could explicitly save all the processor registers on entering the ISR, but doing so would result in some processor registers being saved twice - once by the compiler generated code and then again by the application code. This is undesirable and can be avoided by using the 'naked' attribute in addition to the 'signal' attribute.

```
void SIG_OUTPUT_COMPARE1A(void) __attribute__ ((signal, naked));
void SIG_OUTPUT_COMPARE1A(void)
{
  /* ISR C code for RTOS tick. */
  vPortYieldFromTick();
}
```

The 'naked' attribute prevents the compiler generating any function entry or exit code. Now compiling the code results in much simpler output:

```
;void SIG_OUTPUT_COMPARE1A(void)
;{
  ;---------------------------------------
  ; NO COMPILER GENERATED CODE HERE TO SAVE
  ; THE REGISTERS THAT GET ALTERED BY THE
  ; ISR.
  ;---------------------------------------
  ; CODE GENERATED BY THE COMPILER FROM THE
  ; APPLICATION C CODE.
  ; vTaskIncrementTick();
  CALL 0x0000029B
  ; Call subroutine
  ;---------------------------------------
  ; NO COMPILER GENERATED CODE HERE TO RESTORE
  ; THE REGISTERS OR RETURN FROM THE ISR.
  ;---------------------------------------
;}
```

When the 'naked' attribute is used the compiler does not generate any function entry or exit code so this must now be added explicitly. The macros `portSAVE_CONTEXT()` and `portRESTORE_CONTEXT()` respectively save and restore the entire execution context.:

```
void SIG_OUTPUT_COMPARE1A(void) __attribute__ ((signal, naked));

void SIG_OUTPUT_COMPARE1A(void)
{
  /* Macro that explicitly saves
     the execution context.
  */
  portSAVE_CONTEXT();
```

```
  /* ISR C code for RTOS tick. */
  vPortYieldFromTick();
  /* Macro that explicitly restores the execution context. */
  portRESTORE_CONTEXT();
  /*
     The return from interrupt call must also be explicitly added.
  */
  asm volatile ("reti");
}
```

The 'naked' attribute gives the application code complete control over when and how the AVR context is saved. If the application code saves the entire context on entering the ISR there is no need to save it again before performing a context switch so none of the processor registers get saved twice.

4.5.5 FreeRTOS Tick Code

The actual source code used by the **FreeRTOS** AVR port is slightly different to the examples shown on the previous pages. `vPortYieldFromTick()` is itself implemented as a 'naked' function, and the context is saved and restored within `vPortYieldFromTick()`. It is done this way due to the implementation of nonpreemptive context switches (where a task blocks itself) - which are not described here.

The **FreeRTOS** implementation of the RTOS tick is therefore (see the comments in the source code snippets for further details):

```
void SIG_OUTPUT_COMPARE1A(void) __attribute__ ((signal, naked));
void vPortYieldFromTick(void) __attribute__ ((naked));

/* Interrupt service routine for the RTOS tick. */
void SIG_OUTPUT_COMPARE1A(void)
{
  /* Call the tick function. */
  vPortYieldFromTick();
  /*
     Return from the interrupt. If a context
     switch has occurred this will return to a different task.
  */
  asm volatile ("reti");
}

void vPortYieldFromTick(void)
{
  /* This is a naked function so the is saved. */
  portSAVE_CONTEXT();
  /*
     Increment the tick count and check to see if the new tick
     value has caused a delay period to expire. This function
```

```
    call can cause a task to become ready to run.
  */
  vTaskIncrementTick();
  /*
    See if a context switch is
    required. Switch to the context of a task made ready to run by
    vTaskIncrementTick() if it has a priority higher than the
    interrupted task.
  */
  vTaskSwitchContext();
  /*
    Restore the context.
    If a context switch has occurred this will restore the context of
    the task being resumed.
  */
  portRESTORE_CONTEXT();
  /*
    Return from
    this naked function.
  */
  asm volatile ("ret");
}
```

4.5.6 The AVR Context

A context switch requires the entire execution context to be saved. As shown in Figure 4.10, on the AVR microcontroller the context consists of:

- 32 general purpose processor registers. The GCC development tools assume register R1 is set to zero.
- Status register. The value of the status register affects instruction execution, and must be be preserved across context switches.
- Program counter. Upon resumption, a task must continue execution from the instruction that was about to be executed immediately prior to its suspension.
- The two stack pointer registers.

Each real time task has it's own stack memory area so the context can be saved by simply pushing processor registers onto the task stack. Saving the AVR context is one place where assembly code is unavoidable.

`portSAVE_CONTEXT()` is implemented as a macro, the source code for which is given below:

```
#define portSAVE_CONTEXT() \
asm volatile ( \
  "push r0 \n\t" \ (1)
```

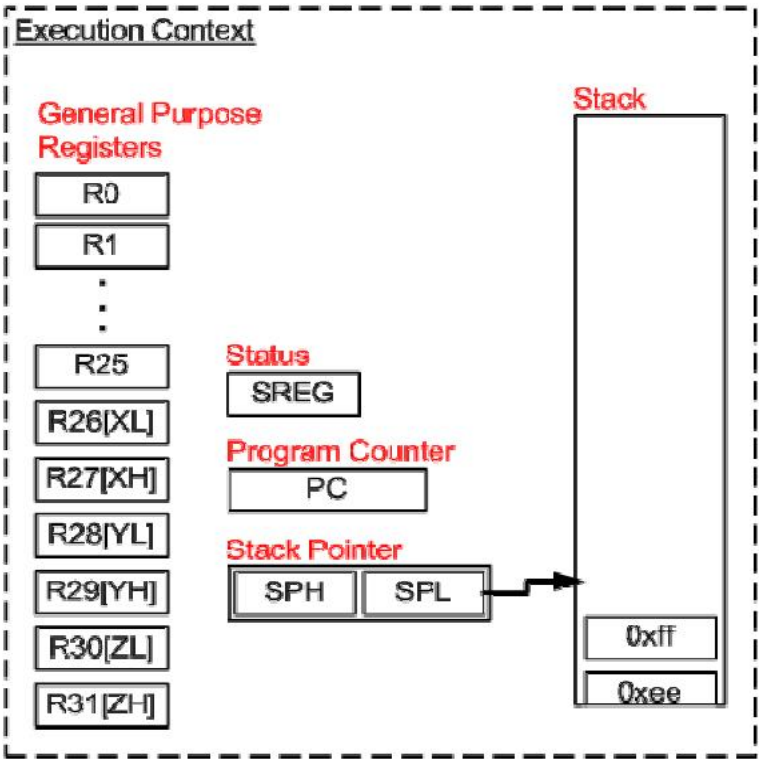

Figure 4.10: The Context of AVR

```
  "in r0, __SREG__ \n\t" \ (2)
  "cli \n\t" \ (3)
  "push r0 \n\t" \ (4)
  "push r1 \n\t" \ (5)
  "clr r1 \n\t" \ (6)
  "push r2 \n\t" \ (7)
  "push r3 \n\t" \
  "push r4 \n\t" \
  "push r5 \n\t" \
     :
     :
     :
  "push r30 \n\t" \
  "push r31 \n\t" \
  "lds r26, pxCurrentTCB \n\t" \ (8)
  "lds r27, pxCurrentTCB + 1 \n\t" \ (9)
  "in r0, __SP_L__ \n\t" \ (10)
  "st x+, r0 \n\t" \ (11)
  "in r0, __SP_H__ \n\t" \ (12)
  "st x+, r0 \n\t" \ (13)
);
```

Referring to the source code above:

- Processor register R0 is saved first as it is used when the status register is saved, and must be saved with its original value.
- The status register is moved into R0 (2) so it can be saved onto the stack (4).
- Processor interrupts are disabled (3). If `portSAVE_CONTEXT()` was only called from within an ISR there would be no need to explicitly disable interrupts as the AVR will have already done so. As the `portSAVE_CONTEXT()` macro is also used outside of interrupt service routines (when a task suspends itself) interrupts must be explicitly cleared as early as possible.
- The code generated by the compiler from the ISR C source code assumes R1 is set to zero. The original value of R1 is saved (5) before R1 is cleared (6).
- Between (7) and (8) all remaining processor registers are saved in numerical order.
- The stack of the task being suspended now contains a copy of the tasks execution context. The kernel stores the tasks stack pointer so the context can be retrieved and restored when the task is resumed. The X processor register is loaded with the address to which the stack pointer is to be saved (8 and 9).

- The stack pointer is saved, first the low byte (10 and 11), then the high nibble (12 and 13).

4.5.7 Restoring the Context

`portRESTORE_CONTEXT()` is the reverse of `portSAVE_CONTEXT()`. The context of the task being resumed was previously stored in the tasks stack. The real time kernel retrieves the stack pointer for the task then `POP`'s the context back into the correct processor registers.

```
#define portRESTORE_CONTEXT() \
asm volatile ( \
  "lds r26, pxCurrentTCB \n\t" \ (1)
  "lds r27, pxCurrentTCB + 1 \n\t" \ (2)
  "ld r28, x+ \n\t" \
  "out __SP_L__, r28 \n\t" \ (3)
  "ld r29, x+ \n\t" \
  "out __SP_H__, r29 \n\t" \ (4)
  "pop r31 \n\t" \
  "pop r30 \n\t" \
    :
    :
    :
  "pop r1 \n\t" \
  "pop r0 \n\t" \ (5)
  "out __SREG__, r0 \n\t" \ (6)
  "pop r0 \n\t" \ (7)
);
```

Referring to the code above:

- `pxCurrentTCB` holds the address from where the tasks stack pointer can be retrieved. This is loaded into the X register (1 and 2).
- The stack pointer for the task being resumed is loaded into the AVR stack pointer, first the low byte (3), then the high nibble (4).
- The processor registers are then popped from the stack in reverse numerical order, down to R1.
- The status register stored on the stack between registers R1 and R0, so is restored (6) before R0 (7).

4.5.8 Putting It All Together

The final part of section 2 shows how these building blocks and source code modules are used to achieve an RTOS context switch on the AVR microcontroller. The example demonstrates in seven steps the process of switching from a lower priority task, called `TaskA`, to a higher priority task, called `TaskB`. The source code is compatible with the WinAVR C development tools.

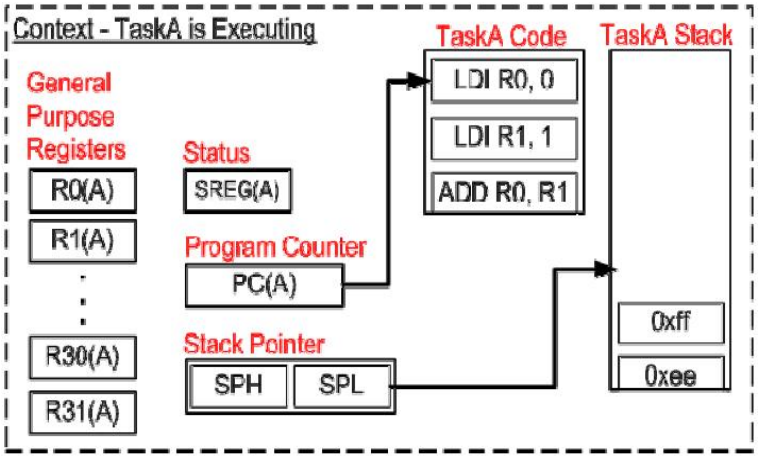

Figure 4.11: The Context Switch - Step 1

4.5.8.1 Context Switch - Step 1

Prior to the RTOS Tick Interrupt

This example starts with `TaskA` executing. `TaskB` has previously been suspended so its context has already been stored on the `TaskB` stack. `TaskA` has the context demonstrated by Figure 4.11.

The (A) label within each register shows that the register contains the correct value for the context of `TaskA`.

4.5.8.2 Context Switch - Step 2

The RTOS Tick Interrupt Occurs

The RTOS tick occurs just as `TaskA` is about to execute an `LDI` instruction. When the interrupt occurs the AVR microcontroller automatically places the current program counter (PC) onto the stack before jumping to the start of the RTOS tick ISR as shown in Figure 4.11.

4.5.8.3 Context Switch - Step 3

The RTOS Tick Interrupt Executes

The ISR source code is given below. The comments have been removed to ease reading, but can be viewed on a previous page.

```
/* Interrupt service routine for the RTOS tick. */
void SIG_OUTPUT_COMPARE1A(void)
{
  vPortYieldFromTick();
```

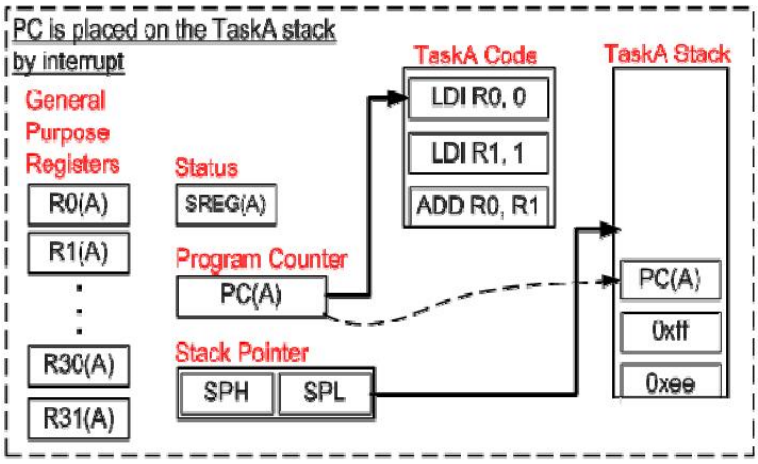

Figure 4.12: The Context Switch - Step 2

```
  asm volatile ("reti");
}

void vPortYieldFromTick(void)
{
  portSAVE_CONTEXT();
  vTaskIncrementTick();
  vTaskSwitchContext();
  portRESTORE_CONTEXT();
  asm volatile ("ret");
}
```

`SIG_OUTPUT_COMPARE1A()` is a naked function, so the first instruction is a call to `vPortYieldFromTick()`. `vPortYieldFromTick()` is also a naked function so the AVR execution context is saved explicitly by a call to `portSAVE_CONTEXT()`.

`portSAVE_CONTEXT()` pushes the entire AVR execution context onto the stack of `TaskA`, resulting in the stack illustrated in Figure 4.13. The stack pointer for `TaskA` now points to the top of it's own context. `portSAVE_CONTEXT()` completes by storing a copy of the stack pointer. The real time kernel already has copy of the `TaskB` stack pointer - taken the last time `TaskB` was suspended.

4.5.8.4 Context Switch - Step 4

Incrementing the Tick Count

`vTaskIncrementTick()` executes after the `TaskA` context has been saved. For the purposes of this example assume that incrementing the tick count has caused `TaskB` to become ready to run. `TaskB` has a higher priority than `TaskA` so

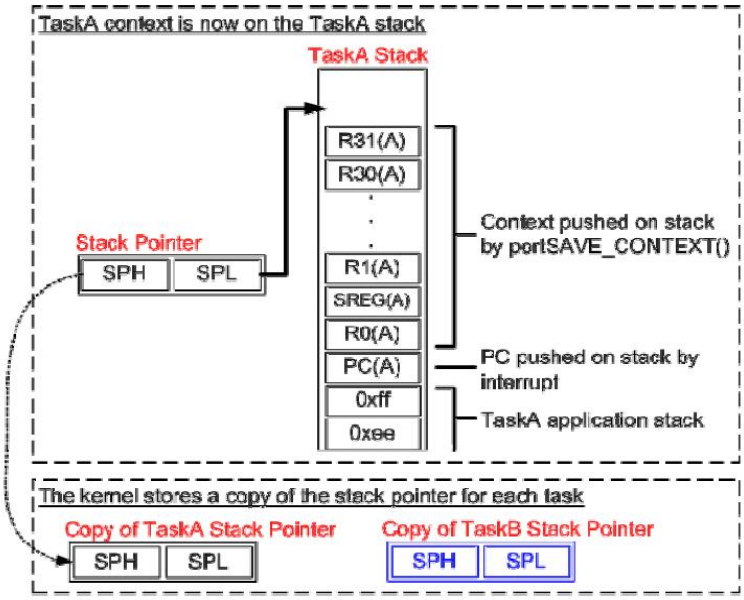

Figure 4.13: The Context Switch - Step 3

vTaskSwitchContext() selects `TaskB` as the task to be given processing time when the ISR completes.

4.5.8.5 Context Switch - Step 5

The `TaskB` Stack Pointer is Retrieved

The `TaskB` context must be restored. The first thing `portRESTORE_CONTEXT` does is retrieve the `TaskB` stack pointer from the copy taken when `TaskB` was suspended. The `TaskB` stack pointer is loaded into the processor stack pointer, so now the AVR stack points to the top of the `TaskB` context as shown in Figure 4.14.

4.5.8.6 Context Switch - Step 6

Restore the `TaskB` context `portRESTORE_CONTEXT()` completes by restoring the `TaskB` context from its stack into the appropriate processor registers. Only the program counter remains on the stack as shown in Figure 4.15.

4.5.8.7 Context Switch - Step 7

The RTOS tick exits `vPortYieldFromTick()` returns to `SIG_OUTPUT_COMPARE-1A()` where the final instruction is a return from interrupt (RETI). A RETI

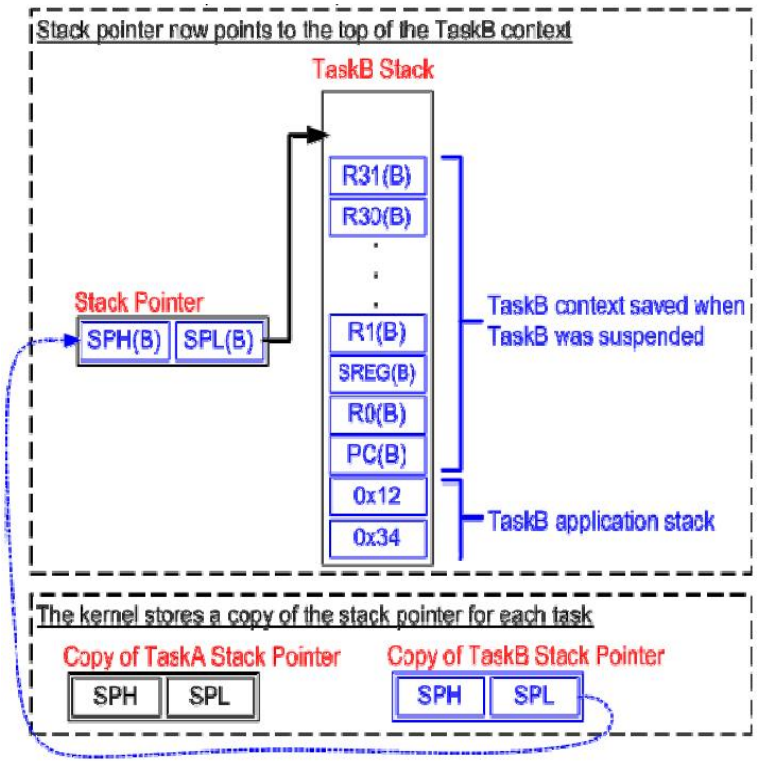

Figure 4.14: The Context Switch - Step 5

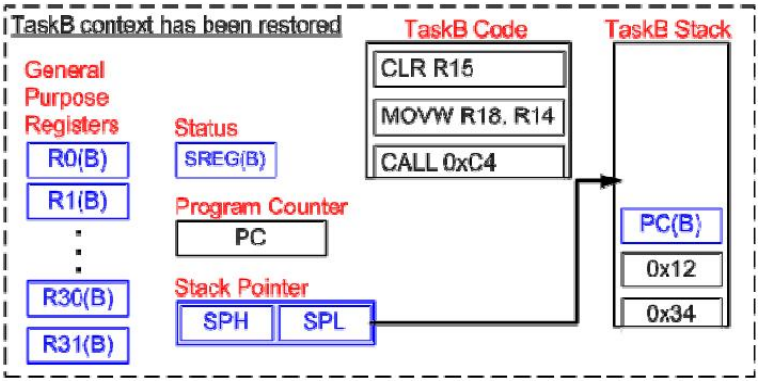

Figure 4.15: The Context Switch - Step 6

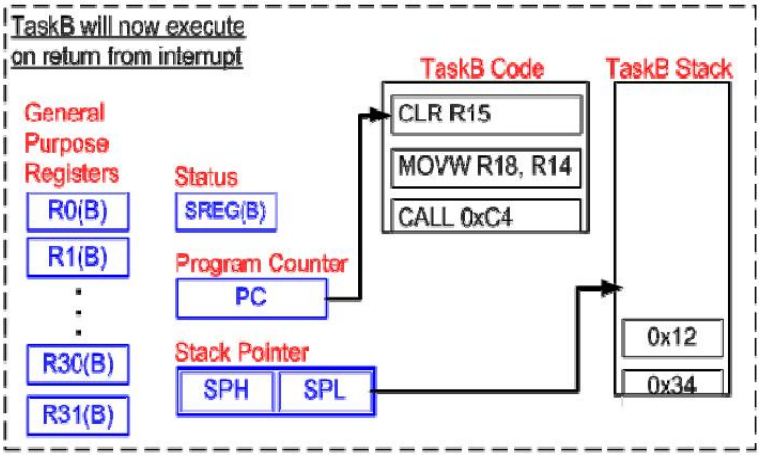

Figure 4.16: The Context Switch - Step 7

instruction assumes the next value on the stack is a return address placed onto the stack when the interrupt occurred. When the RTOS tick interrupt started the AVR automatically placed the `TaskA` return address onto the stack - the address of the next instruction to execute in `TaskA`. The ISR altered the stack pointer so it now points to the `TaskB` stack. Therefore the return address POP'ed from the stack by the `RETI` instruction is actually the address of the instruction `TaskB` was going to execute immediately before it was suspended. The RTOS tick interrupt interrupted `TaskA`, but is returning to `TaskB` - the context switch is complete! If you would like more information, take a look at the **FreeRTOS** ColdFire Implementation Report. This was written by the Motorola ColdFire port authors, and details both the ColdFire source code and the development process undertaken in producing the port as shown in Figure 4.16.

4.6 The FreeRTOS Scheduler

4.6.1 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 4.17 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`.

Because the scheduler operates as an interrupt, it is part of the HAL and contains implementation specific code. In Figure 4.17, the HAL implementation

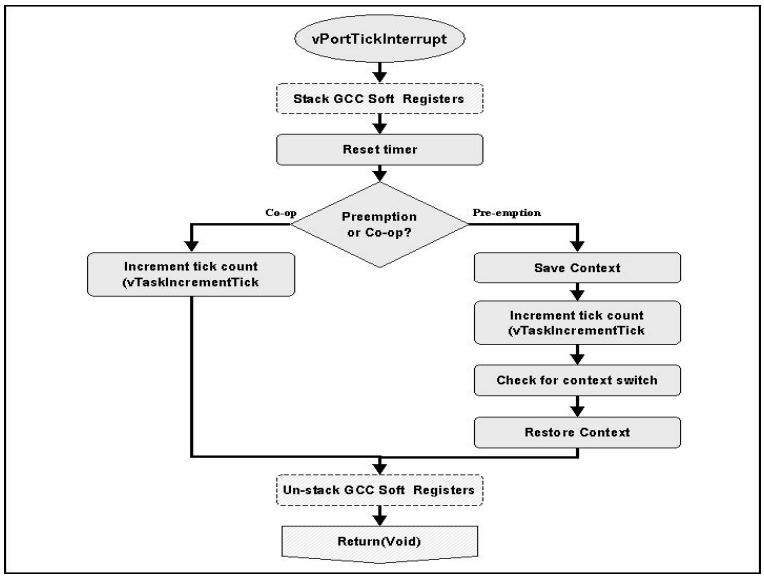

Figure 4.17: Scheduler Algorithm

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.

4.6.2 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

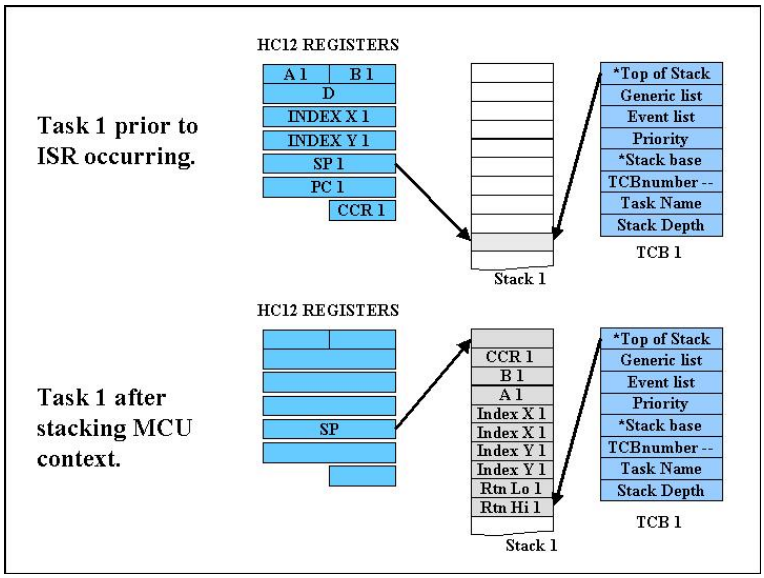

Figure 4.18: Stacking of MCU Context

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 4.18 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.

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

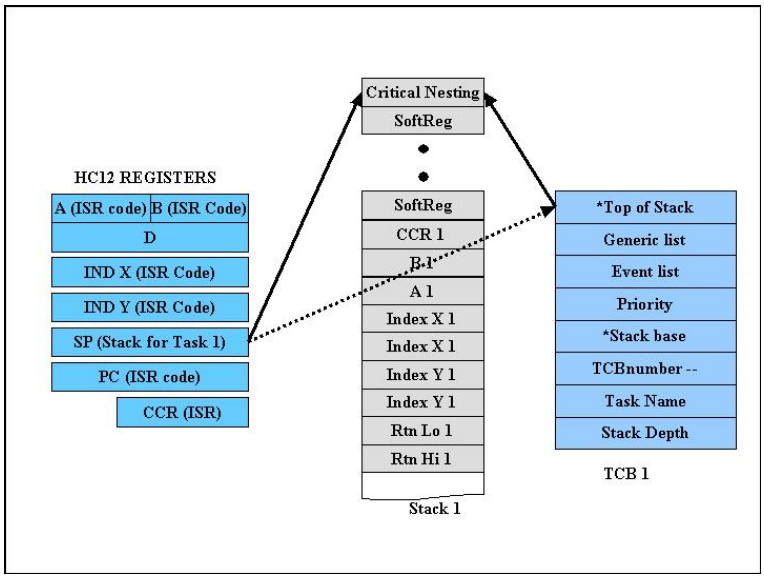

Figure 4.19: Context Frame on Stack 1

built by the ISR mechanism, `portISR_HEAD`, and `portSAVE_CONTEXT` is shown in Figure 4.19.

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.

4.6.3 Context Switch By Stack Pointer Manipulation

In Figure 4.18, 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.

4.6.4 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 4.20.

`xTaskCreate` must first allocate memory for the task's TCB and stack. This is accomplished by calling `AllocateTCBandStack` as shown in Figure 4.21. 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.

`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 4.20, 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 C 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 `port_ISRTAIL` macros as discussed earlier. The content of the dummy frame is

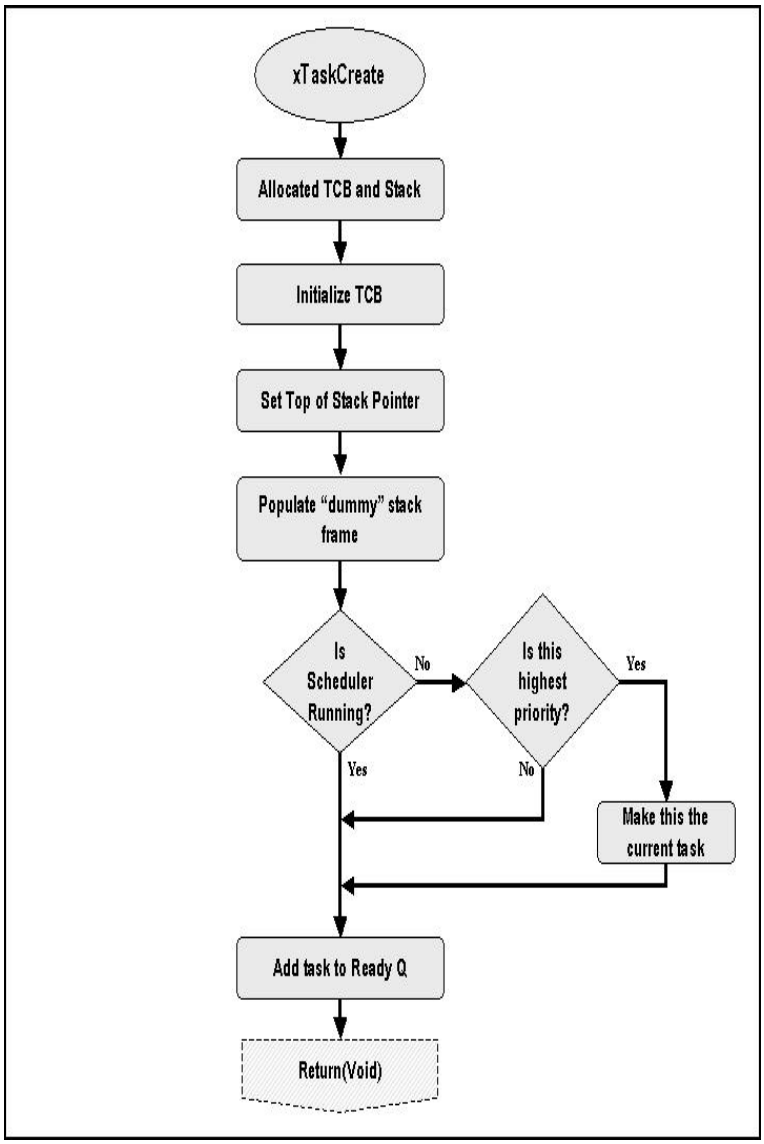

Figure 4.20: Overview of Task Creation

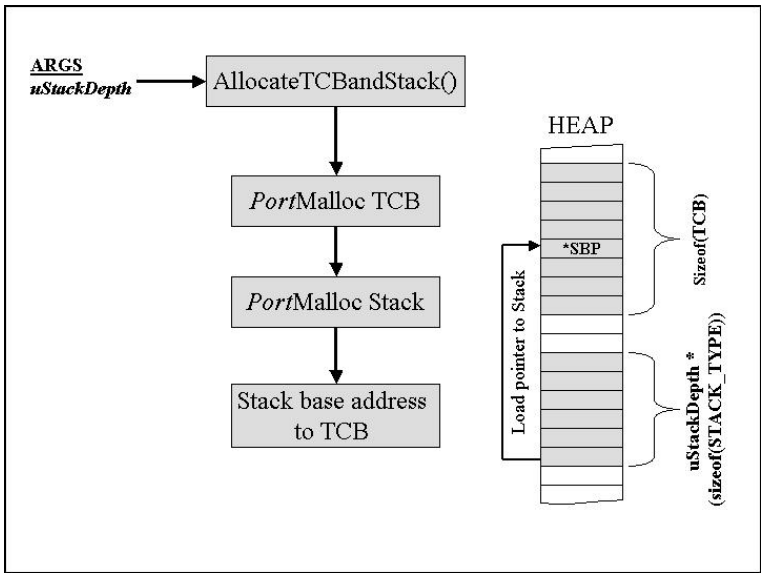

Figure 4.21: Allocate Stack and TCB Memory

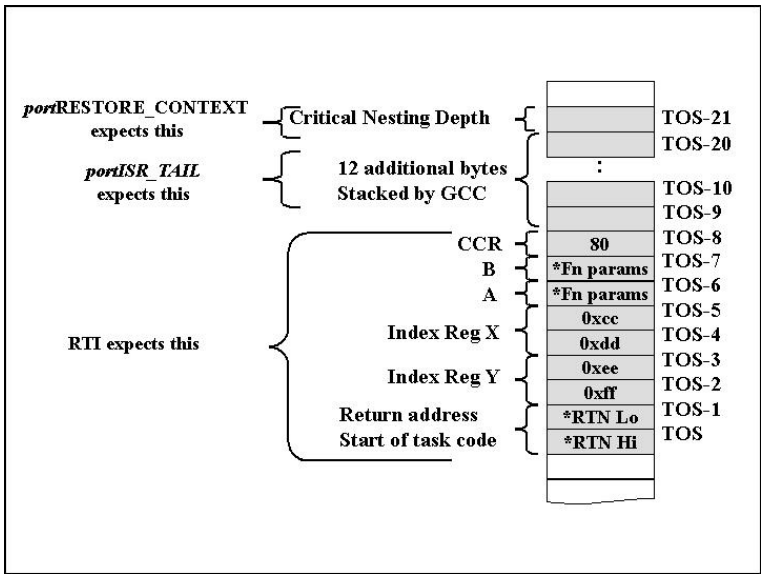

Figure 4.22: Dummy Stack Frame

shown in Figure 4.22. The important element of the dummy frame is the return address which will point to the start address of the task code.

After populating the dummy stack for the task, the top-of-stack pointer (which now points to TOS-21 in Figure 4.22) 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.

4.6.5 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 C 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.

4.6.6 Starting the Scheduler

Figure 4.23 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()`.

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

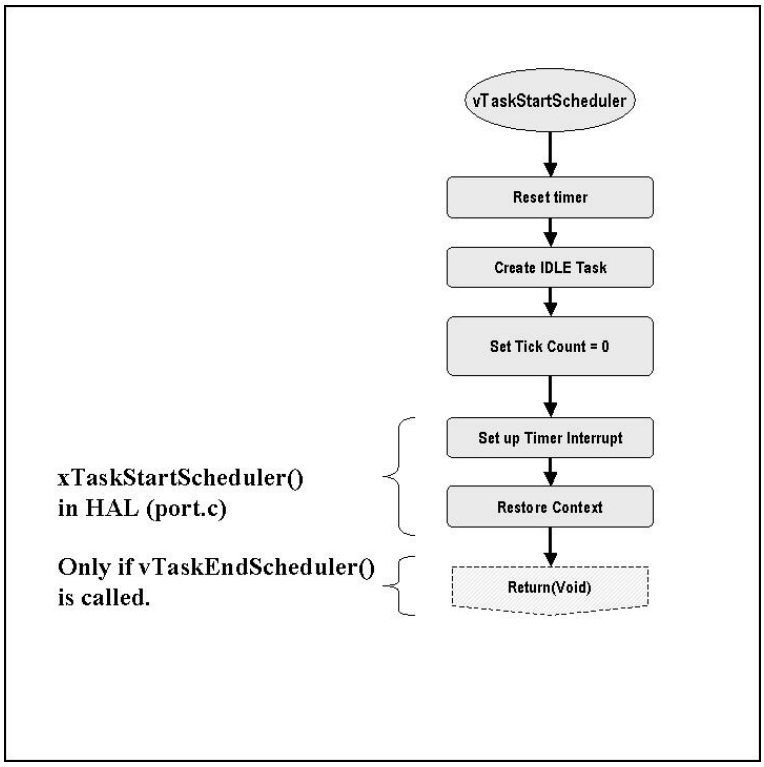

Figure 4.23: **FreeRTOS** Task Scheduler Startup

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.

4.6.7 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 C 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 preempted 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 4.24 shows the algorithms implemented for `vTaskSuspendAll` and `vTaskResumeAll`.

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

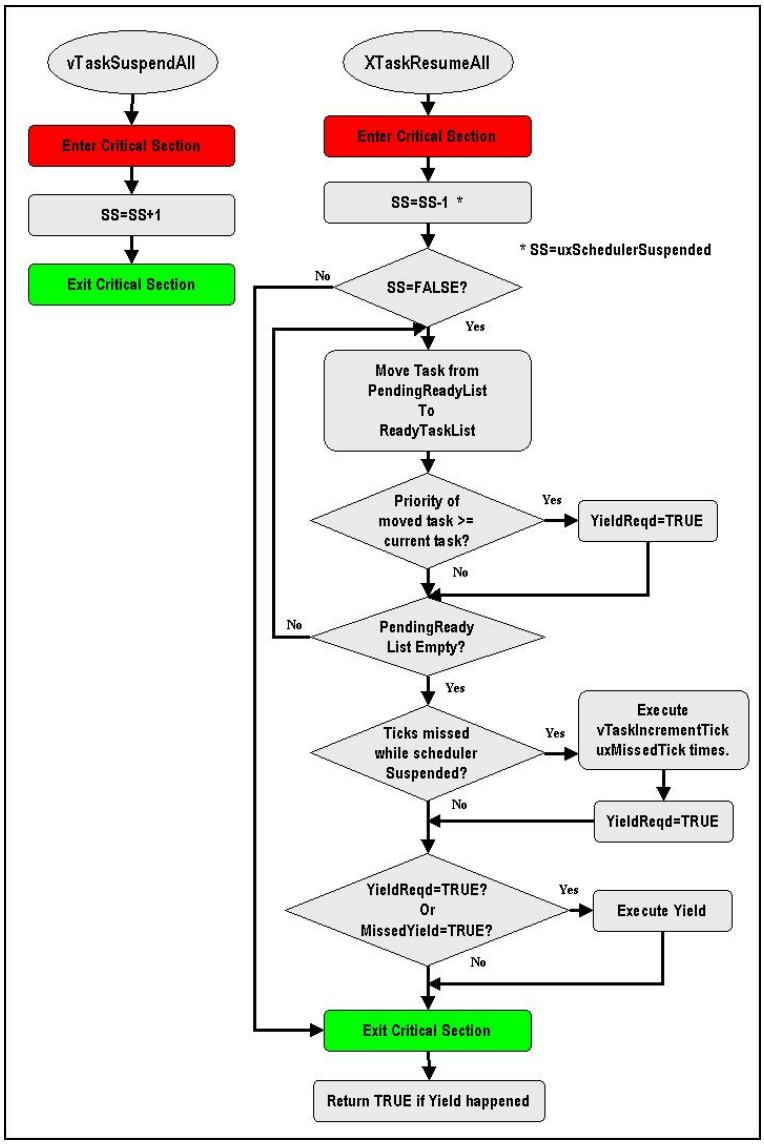

Figure 4.24: Algorithms for `vTaskSuspend` and `xTaskResumeAll`

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 C 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 C 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 4.25. `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.

4.6.8 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 unexpired time is found.

4.7 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

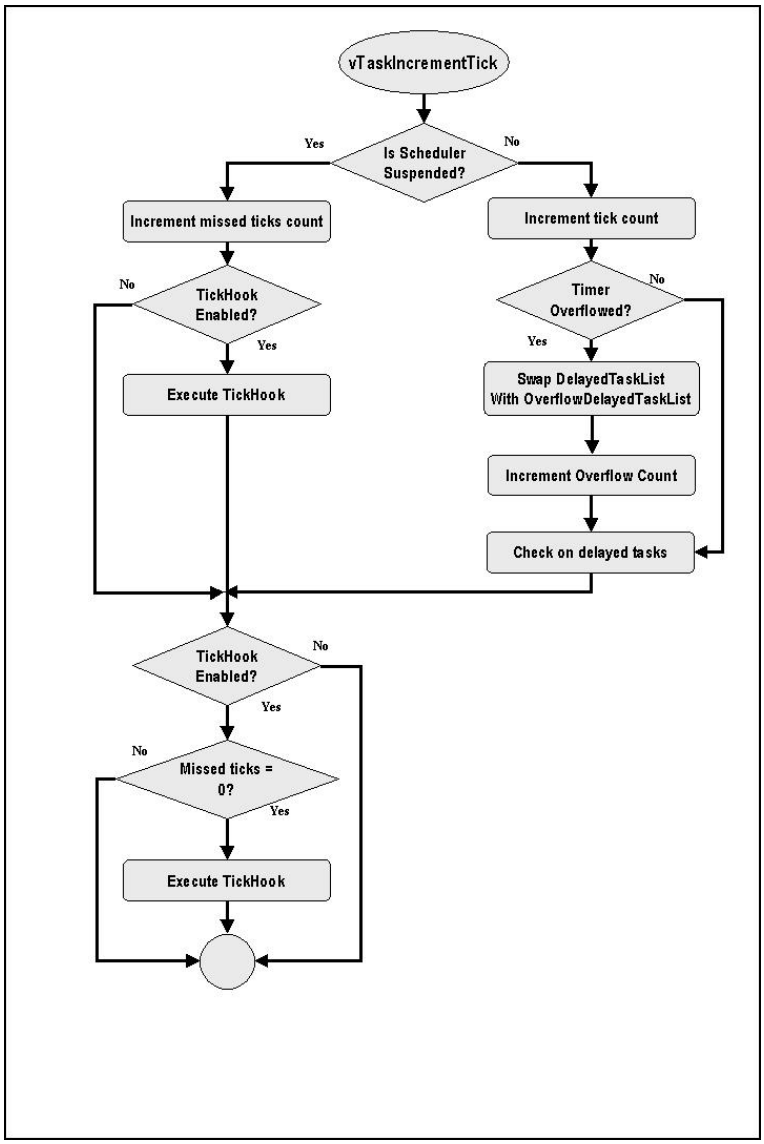

Figure 4.25: Algorithm for `vTaskIncrementTick`

`* pcHead`	Pointer to the byte at the start of Q in memory
`* pcTail`	Pointer to the byte at the end of Q in memory (one more than neccesary)
`* 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 byte)
`uxItemSize`	The size of each Q-able items in bytes
`xRxLocks`	The number of items received (removed) from the Q while the queue was locked
`xTxLocks`	Store the number of items transmitted (added) to the Q while the queue was locked

Table 4.6: Queue Structure Elements

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).

4.8 Queue Management

4.8.1 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. Table 4.6 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.

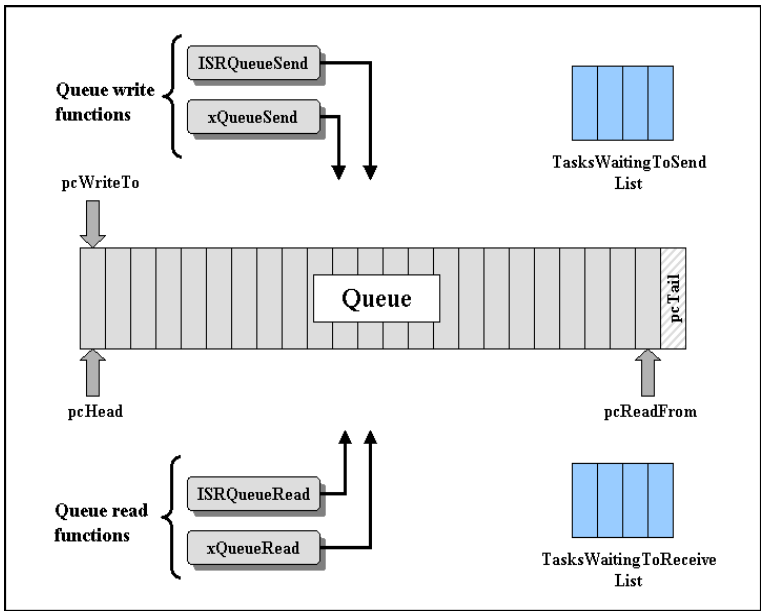

Figure 4.26: Queue Elements

The physical queue size is determined by the number of queue-able 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 4.26 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).

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.

4.8.2 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 4.27. 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 task 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.

If the queue is not locked, the algorithm checks to see if a task has already been unblocked (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` C otherwise, it returns the value that was passed in (as shown at the bottom of Figure 4.27). 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 4.28 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.

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 4.29.

If the scheduler is not suspended after removing the TCB from the {`Event-List`}, then the function removes the task from the {`DelayedTaskList`} and inserts it into the Ready list. If the scheduler has been suspended, then there

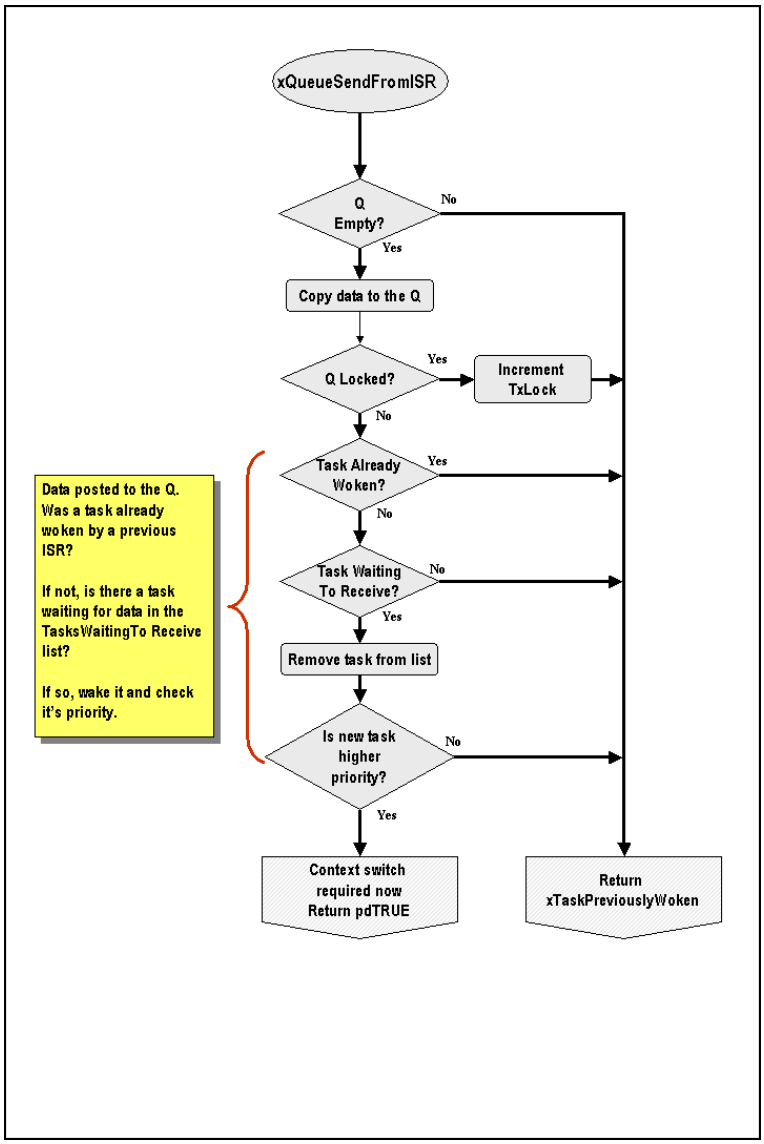

Figure 4.27: Algorithm for Sending to a Queue from an ISR

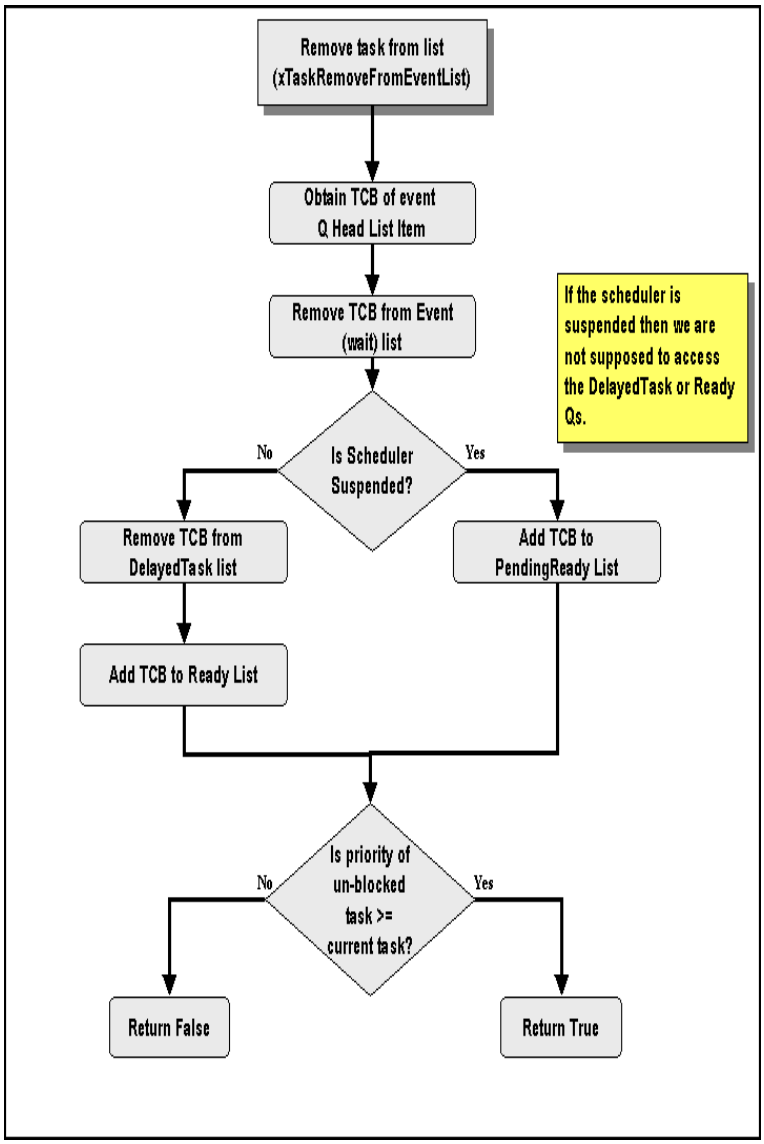

Figure 4.28: Remove Task From Event List

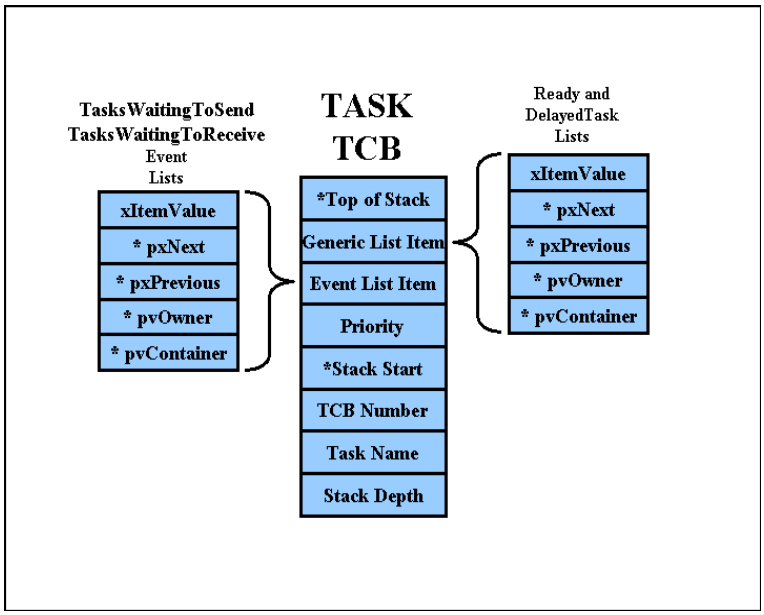

Figure 4.29: Generic and Event Lists in TCB

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.

4.8.3 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**. Figure 4.30 and Figure 4.31 present the algorithm used to perform this operation.

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 4.31 provides greater detail to the blocking process.

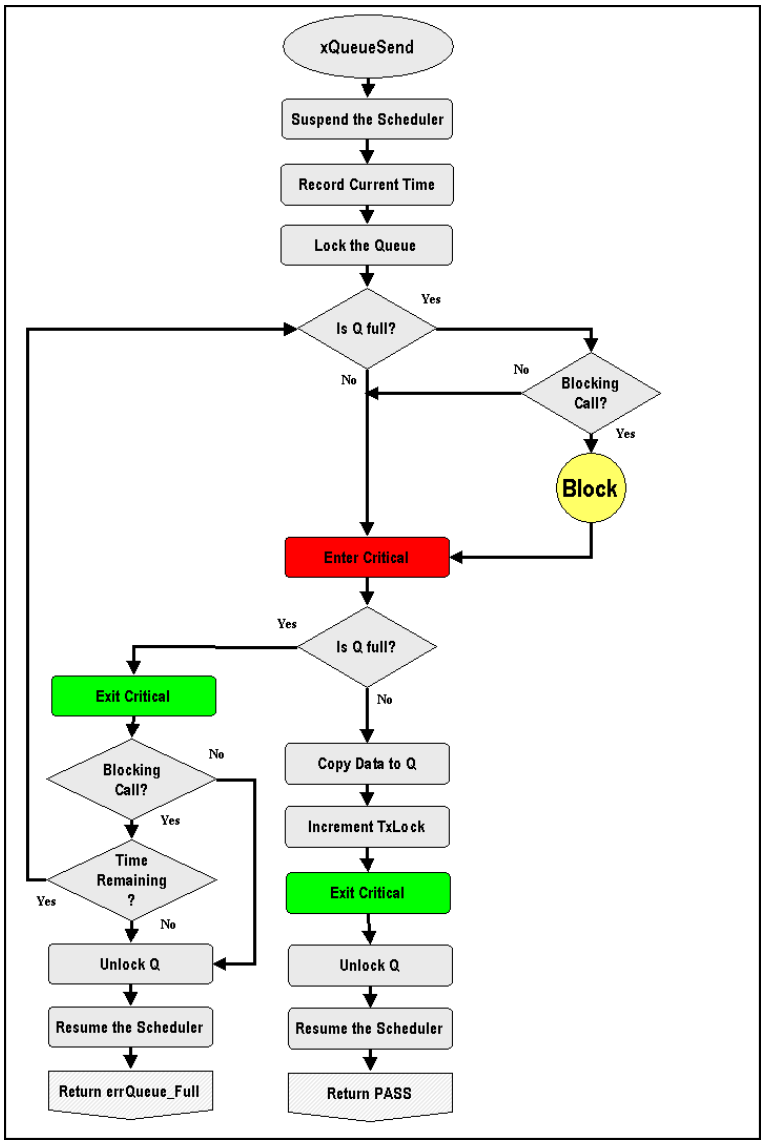

Figure 4.30: Posting to a Queue From a Task

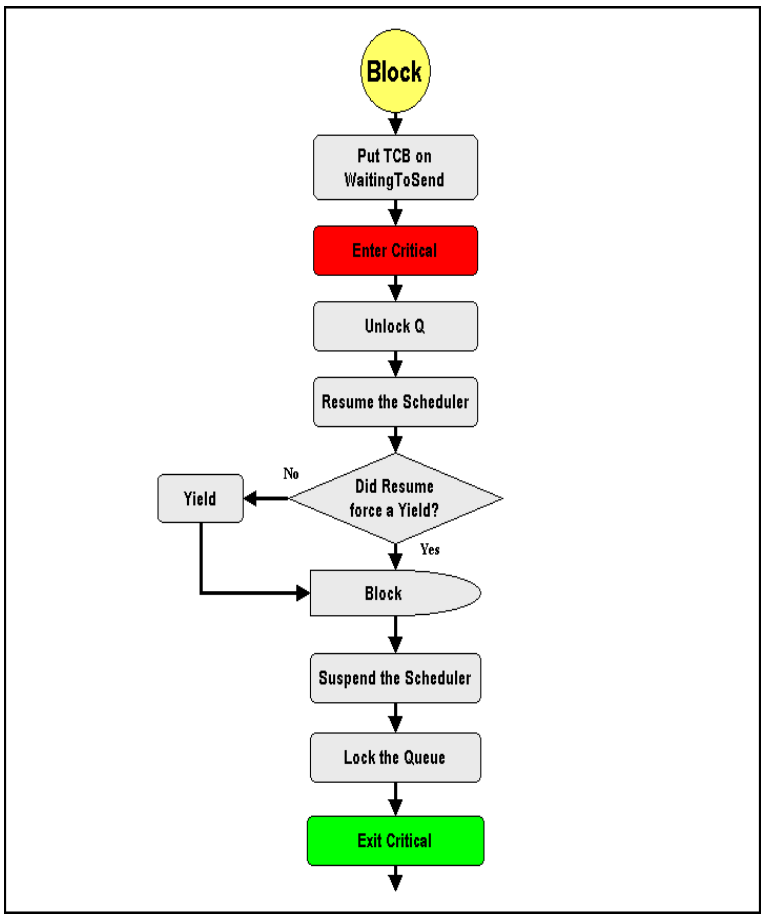

Figure 4.31: Posting to a Queue From a Task

We will digress slightly to describe that process.

`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 4.31 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 4.30.

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 4.32 shows the algorithm for unlocking the queue.

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 C 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 C however, the scheduler is not running so a pending yield is signaled.

The algorithm shown in Figure 4.32 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 4.30), 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.

4.8.4 Receiving from a Queue C 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.

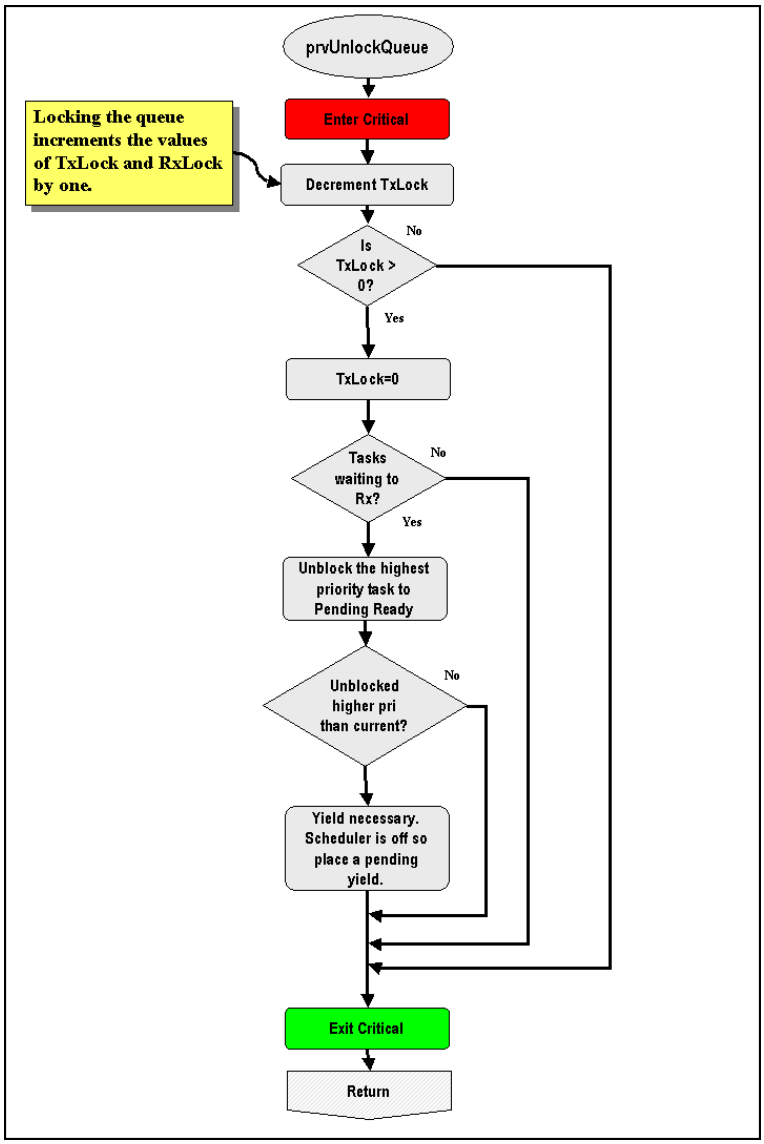

Figure 4.32: Checking for Blocked Tasks On Queue Unlock

These operations won't be covered.

Chapter 5

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.

We will deal with the formalization issue of **FreeRTOS** in [Zhu11a, Zhu11b, Zhu11c].

Bibliography

[Bar07] Richard Barry. **FreeRTOS**, January 2007.

[Goy07] Rich Goyette. An Analysis and Description of the Inner Workings of the **FreeRTOS** Kernel. Technical Report SYSC5701: Operating System Methods for Real-Time Applications, Department of Systems and Computer Engineering, Carleton University, April 2007.

[Zhu11a] Ming-Yuan Zhu. A Formal Model of **FreeRTOS**, Volume 1: The Requirement Specifications. Technical report, CoreTek Systems, Inc., 2011.

[Zhu11b] Ming-Yuan Zhu. A Formal Model of **FreeRTOS**, Volume 2: The Functional Specifications. Technical report, CoreTek Systems, Inc., 2011.

[Zhu11c] Ming-Yuan Zhu. A Formal Model of **FreeRTOS**, Volume 3: The Design Specifications. Technical report, CoreTek Systems, Inc., 2011.