Chapter 16 – FreeRTOS

Table of Contents

Section 1: Synopsis

Multi tasking

See Section 1 of "How FreeRTOS.org works" for an introduction to multi tasking and RTOS concepts.

Features

The following standard features are provided.

- Choice of RTOS scheduling policy
  1. Pre-emptive:
     Always runs the highest available task. Tasks of identical priority share CPU time (fully pre-emptive with round robin time slicing).
  2. Cooperative:
     Context switches only occur if a task blocks, or explicitly calls taskYIELD().
- Co-routines (light weight tasks that utilise very little RAM).
- Message queues
- Semaphores [via macros]
- Trace visualisation 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.

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.

Section 2: RTOS Fundamentals

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.

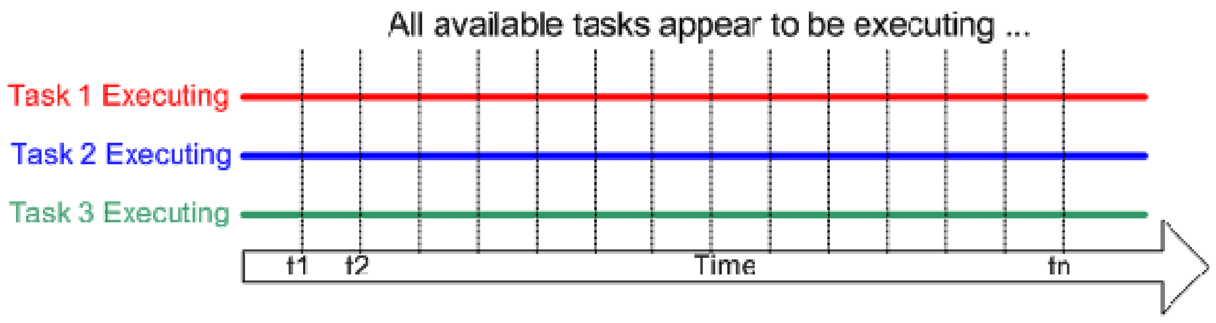
Each executing program is a **task** under control of the operating system. If an operating system can execute multiple tasks in this manner it is said to be **multitasking**.

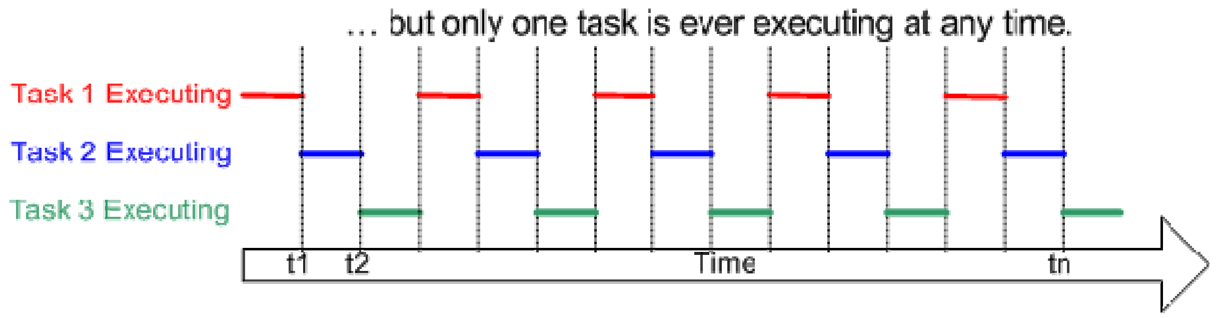
The use of a multitasking operating system can simplify the design of what would otherwise be a complex software application:

- 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.
- 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.

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 the diagram below 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.



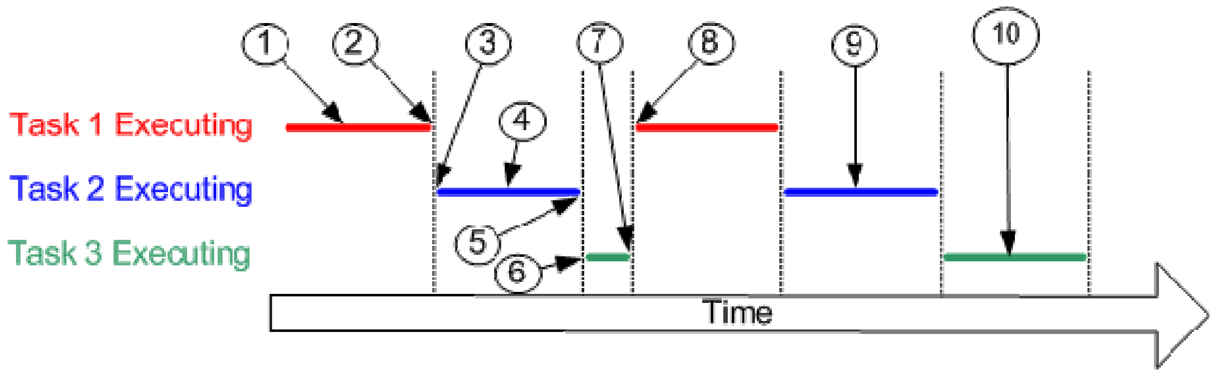


Scheduling

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.

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 (eg 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.



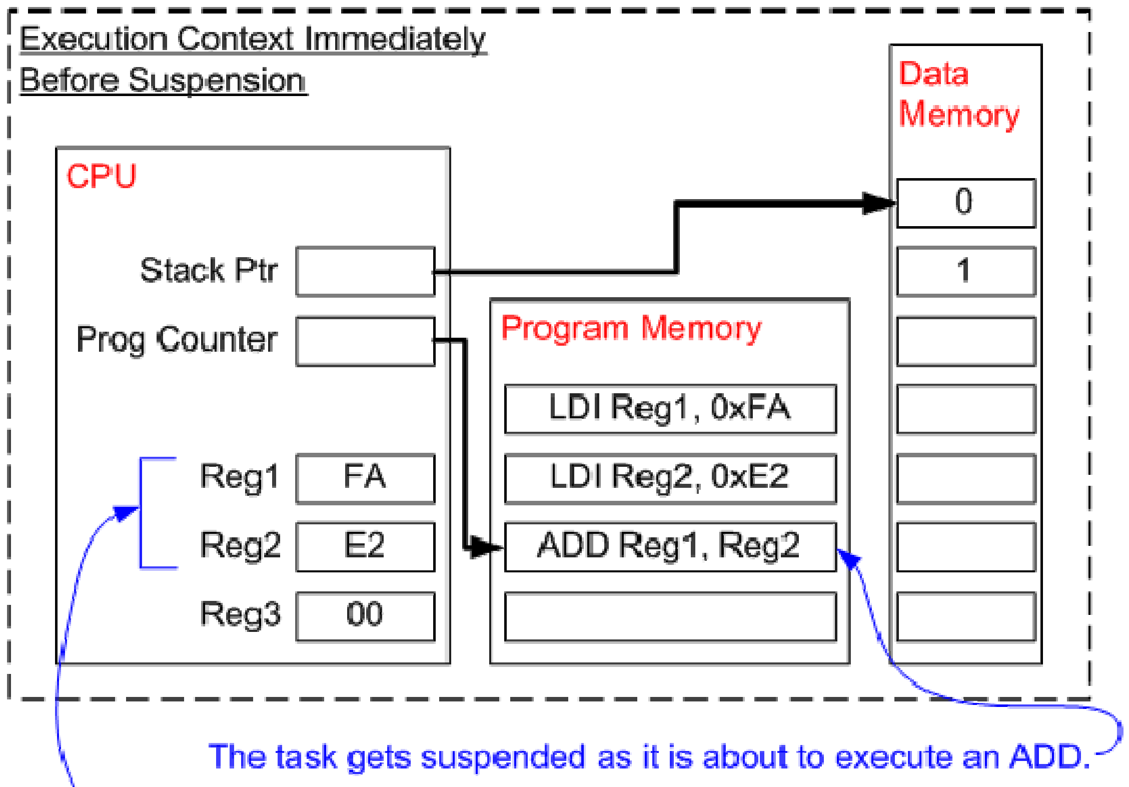
Referring to the numbers in the diagram above:

- 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.

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.



The previous instructions have already set the registers used by the ADD. When the task is resumed the ADD instruction will be the first instruction to execute. The task will not know if a different task modified Reg1 or Reg2 in the interim.

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**.

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 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]
    }
}
```

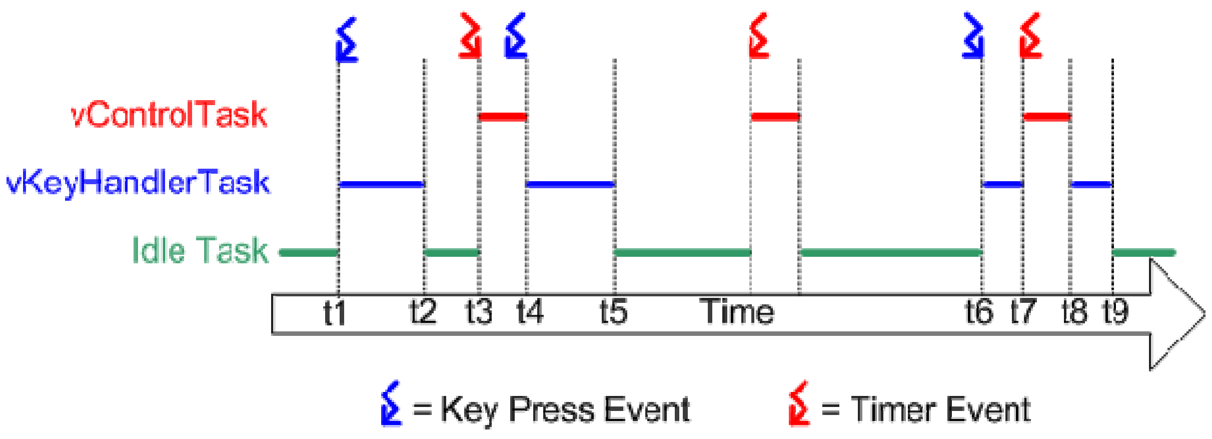
The software engineer must assign the control task the highest priority as:

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.

Real Time Scheduling

The diagram below 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.



Referring to the diagram 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.

Section 3: RTOS Implementation

Detailed Description

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.

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.

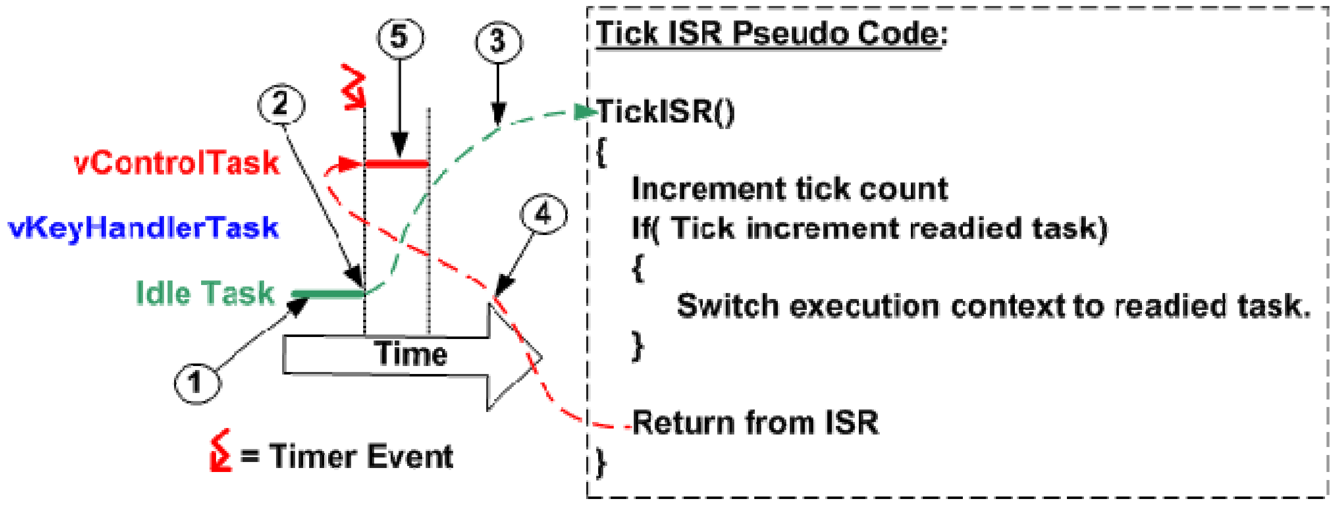
The example presented here uses the WinAVR development tools. WinAVR is a free Windows to AVR cross compiler based on GCC.

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 the diagram 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

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
    ; -----------------------------------------
```

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 automatically 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.

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 non-preemptive context switches (where a task blocks itself) - which are not described here.

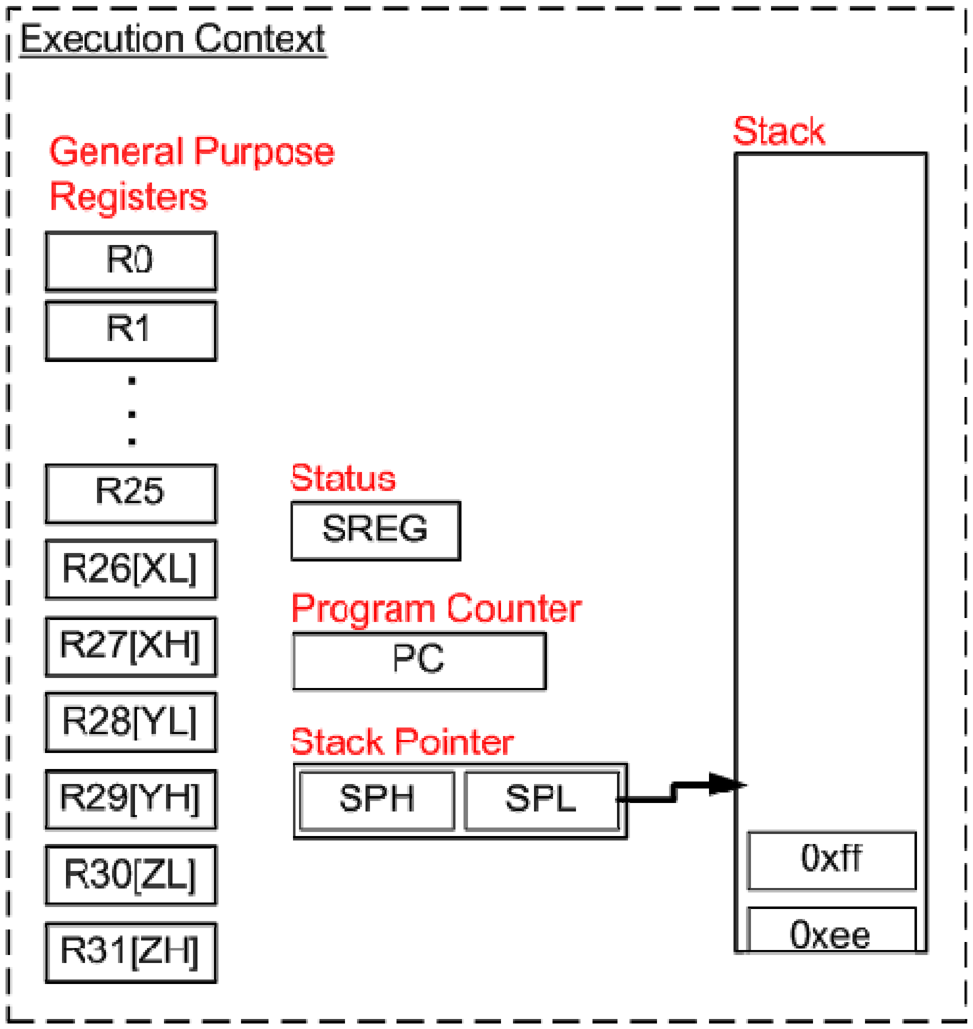
The FreeRTOS implementation of the RTOS tick is therefore *(see the comments in the source code snippets for further details)*:

```
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 context
    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" );
}
/*--------------------------------------------------*/
```

The AVR Context

A context switch requires the entire execution context to be saved. 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)
  "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).

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).

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.

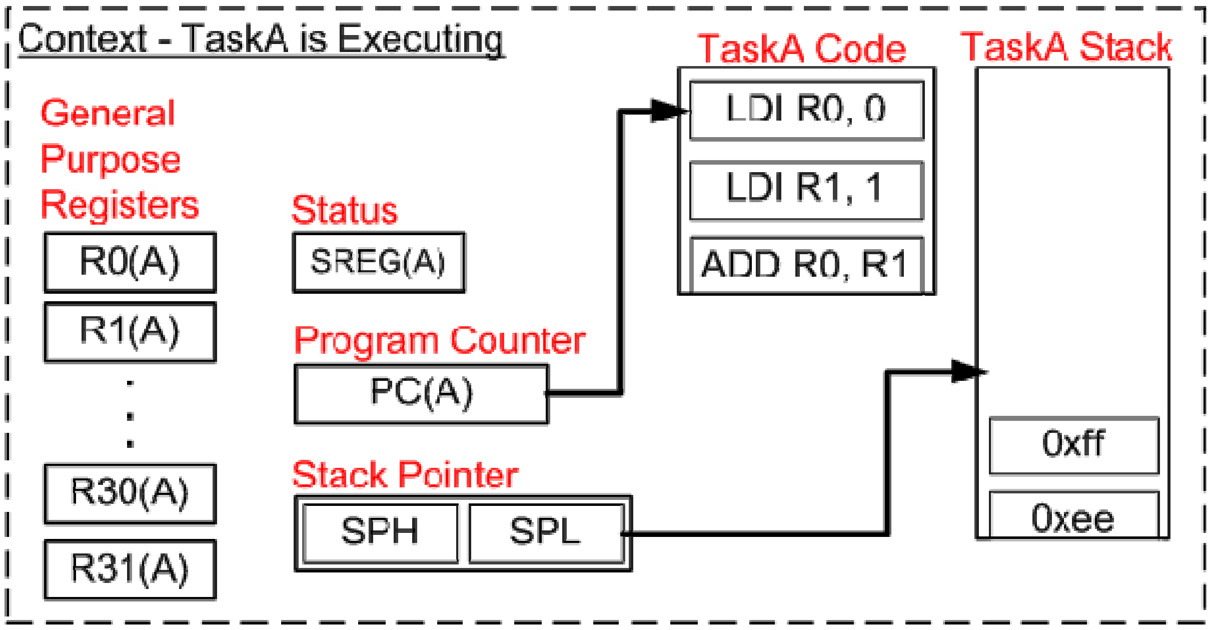
The source code is compatible with the WinAVR C development tools.

RTOS 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 the diagram below.

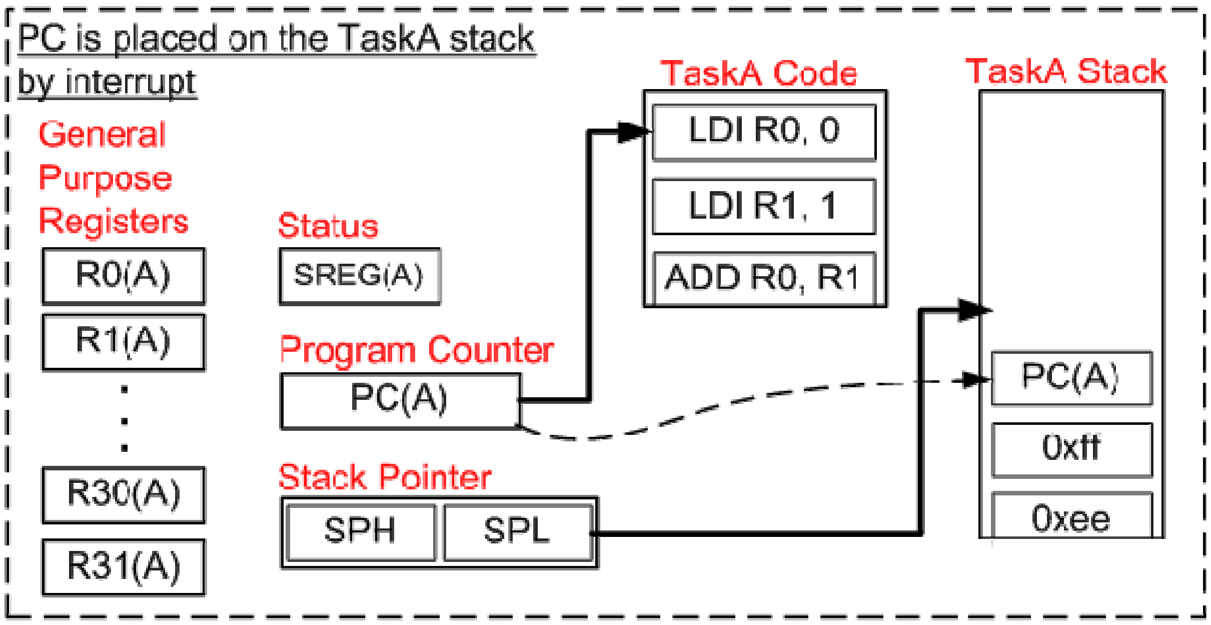


The (A) label within each register shows that the register contains the correct value for the context of task A.

RTOS 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.



RTOS Context Switch - Step 3

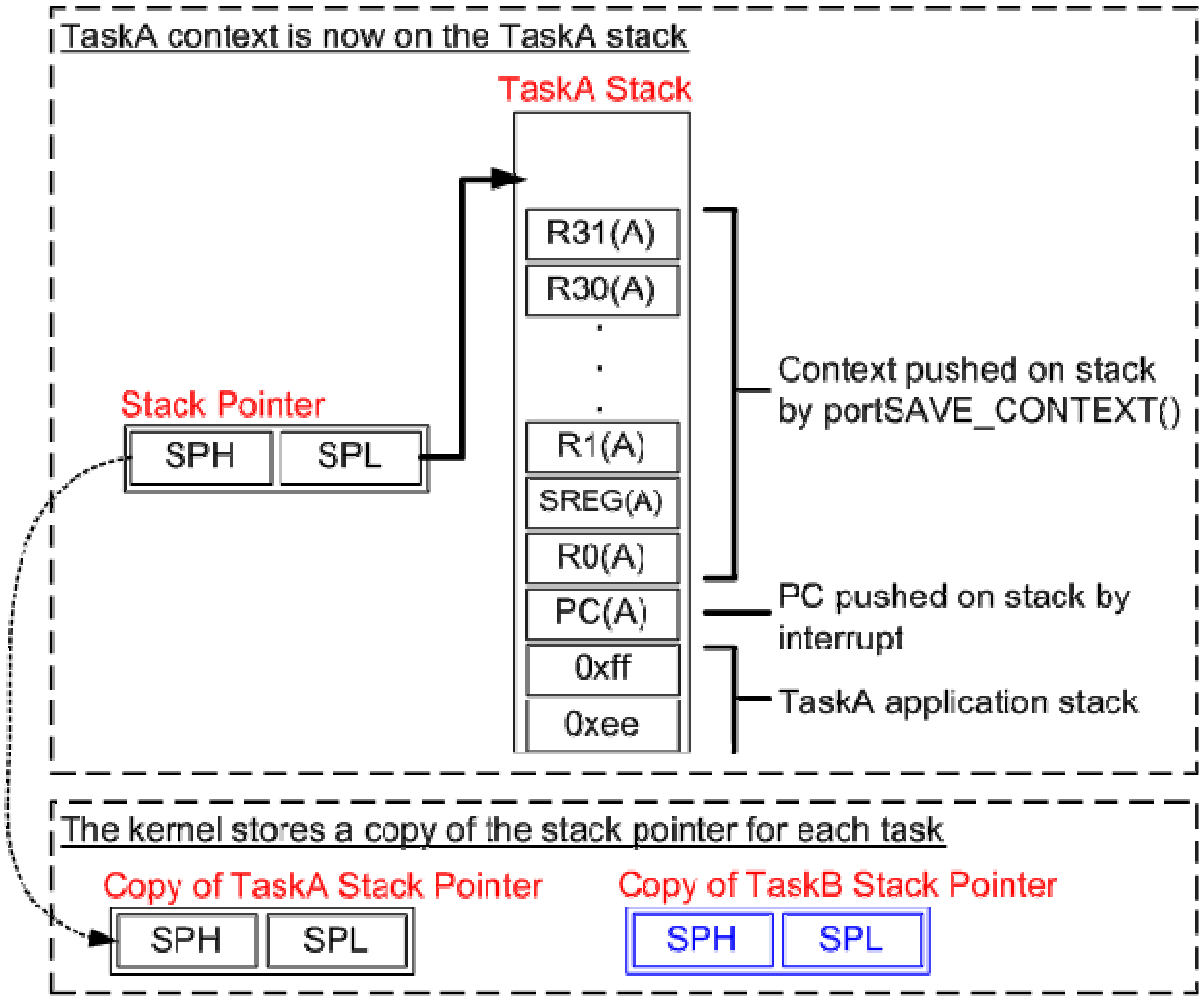
The RTOS tick interrupt executes

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();
    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 below. 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.


TaskA context is now on the TaskA stack
TaskA Stack
R31(A)
R30(A)
R1(A)
SREG(A)
R0(A)
PC(A)
0xff
0xee
Context pushed on stack by portSAVE_CONTEXT()
PC pushed on stack by interrupt
TaskA application stack
Stack Pointer
SPH
SPL
The kernel stores a copy of the stack pointer for each task
Copy of TaskA Stack Pointer
SPH
SPL
Copy of TaskB Stack Pointer
SPH
SPL

RTOS Context Switch - Step 4

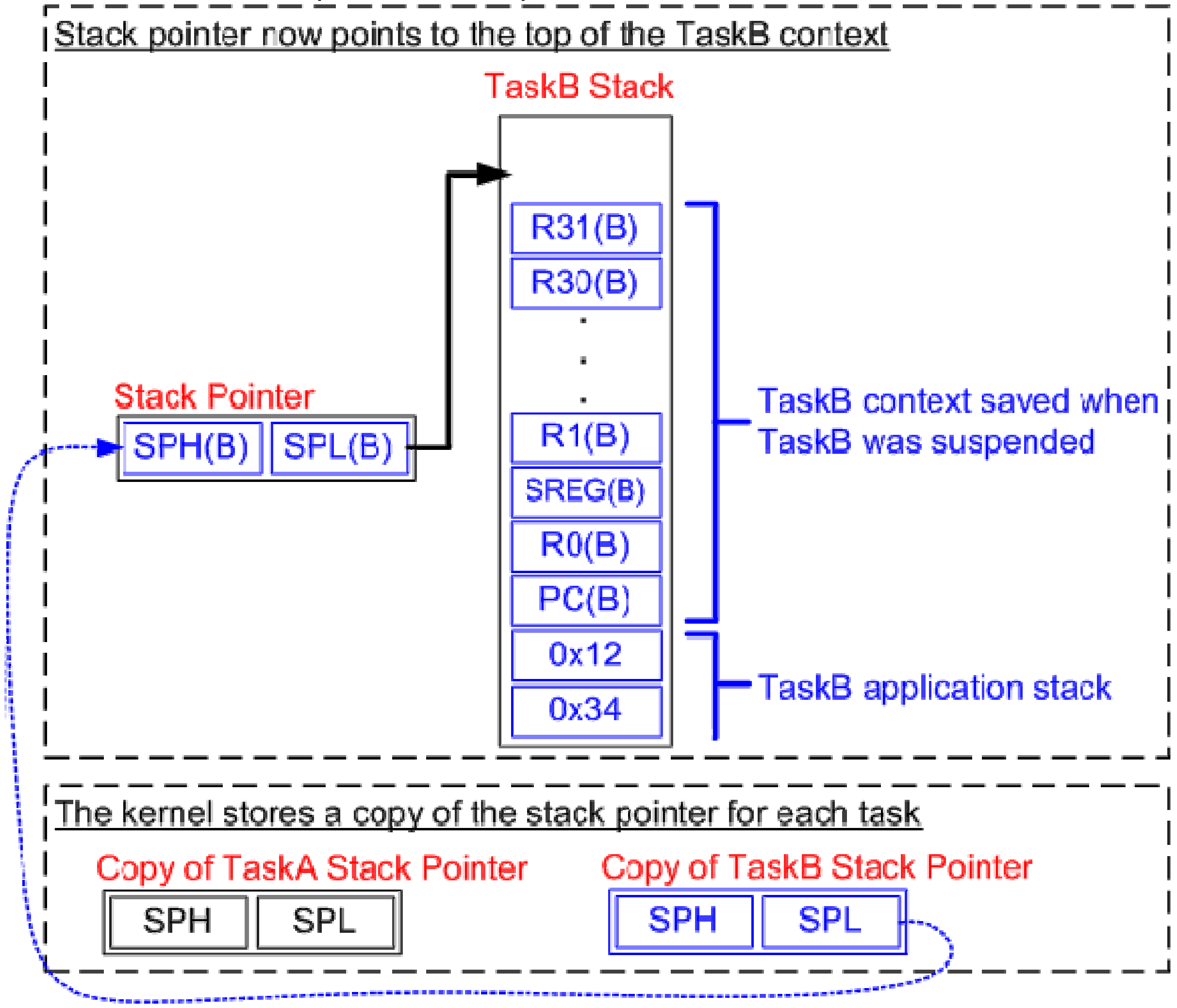
Incrementing the Tick Count

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

RTOS Context Switch - Step 5

The TaskB stack pointer is retrieved

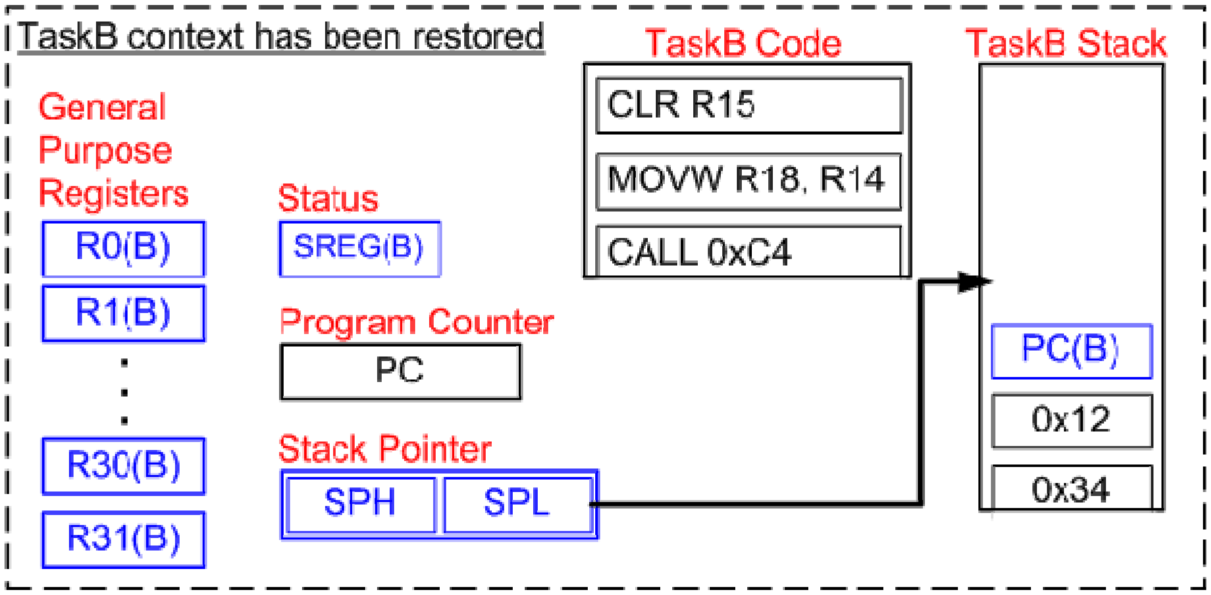
The TaskB context must be restored. The first thing portRESTORE_CONTEXT does is retrieve the TaskB stack pointer from the copy taken when TaskB was suspended. The TaskB stack pointer is loaded into the processor stack pointer, so now the AVR stack points to the top of the TaskB context.


Stack pointer now points to the top of the TaskB context
TaskB Stack
R31(B)
R30(B)
R1(B)
SREG(B)
R0(B)
PC(B)
0x12
0x34
Stack Pointer
SPH(B)
SPL(B)
TaskB context saved when TaskB was suspended
TaskB application stack
The kernel stores a copy of the stack pointer for each task
Copy of TaskA Stack Pointer
SPH
SPL
Copy of TaskB Stack Pointer
SPH
SPL

RTOS Context Switch - Step 6

Restore the TaskB context

portRESTORE_CONTEXT() completes by restoring the TaskB context from its stack into the appropriate processor registers.



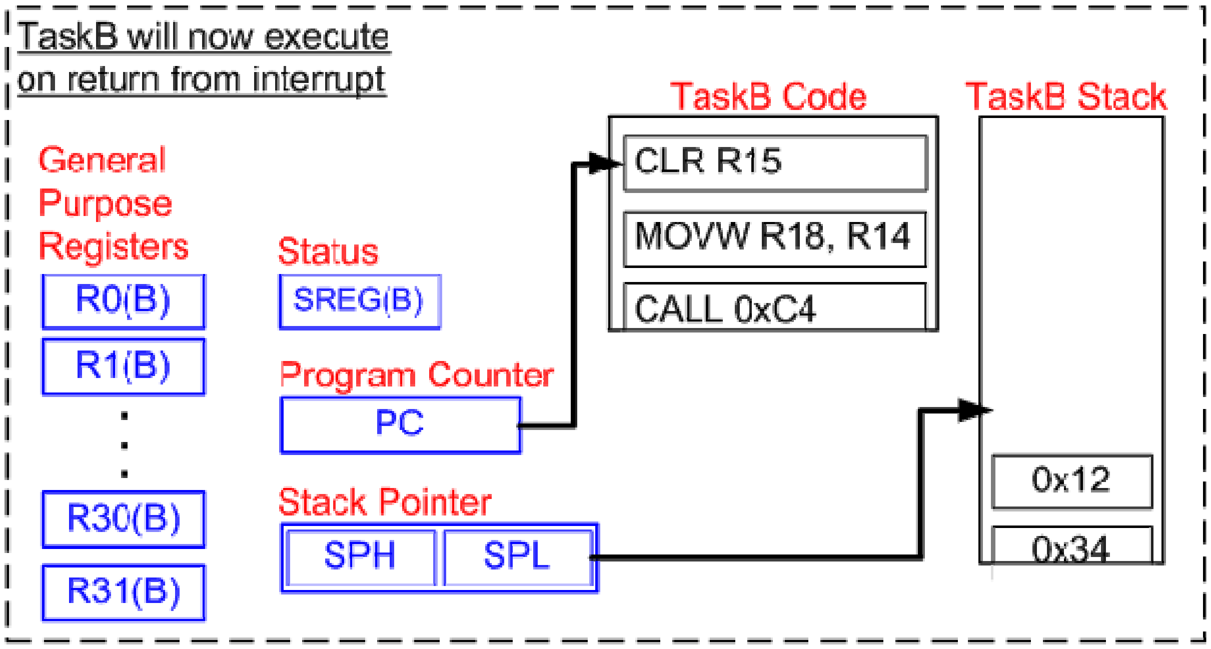
Only the program counter remains on the stack.

RTOS Context Switch - Step 7

The RTOS tick exits

vPortYieldFromTick() returns to SIG_OUTPUT_COMPARE1A() where the final instruction is a return from interrupt (RETI). A RETI 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.

Tasks and Co-routines

Co-routine 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 co-routines within an application is completely optional.

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

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.

Task Summary

🙂 Simple.

🙂 No restrictions on use.

🙂 Supports full preemption.

🙂 Fully prioritised.

☹️ Each task maintains its own stack resulting in higher RAM usage.

☹️ Re-entrancy must be carefully considered if using preemption.

Characteristics of a 'Co-routine'

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

1. **Stack usage**

   All the co-routines 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**

   Co-routines use prioritised cooperative scheduling with respect to other co-routines, but can be included in an application that uses preemptive tasks.

3. **Macro implementation**

   The co-routine 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 co-routines can be structured.

Co-Routine Summary

🙂 Sharing a stack between co-routines results in much lower RAM usage.

🙂 Cooperative operation makes re-entrancy less of an issue.

🙂 Very portable across architectures.

😐 Fully prioritised relative to other co-routines, 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 co-routines themselves

Tasks

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 utilising the processor.

- **Ready**

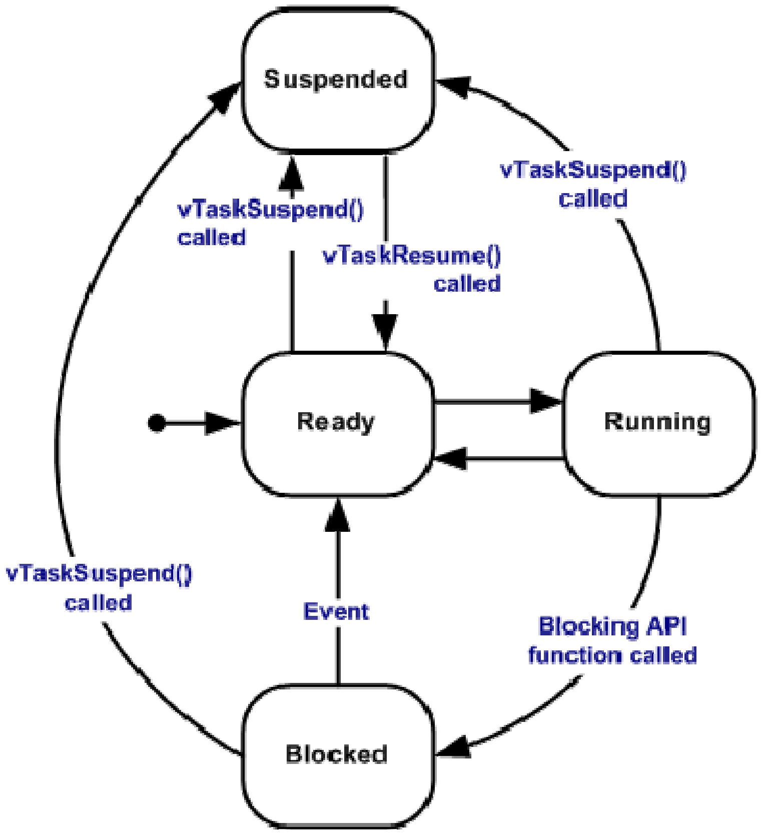
  Ready tasks are those that are able to execute (they are not blocked or suspended) but are not currently executing because a different task of equal or higher priority is already in the Running state.

- **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.



Valid task state transitions

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.

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().

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. --
        }
    }
```

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 visualisation utility can be used to check the microcontroller time allocated to the idle task.

The idle task has no other active functions so can legitimately be starved of microcontroller time under all other conditions.

It is possible for application tasks to share the idle task priority. (tskIDLE_PRIORITY).

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 co-routines 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 co-routine that does nothing but periodically flashes an LED.

```
    void vFlashCoRoutine( xCoRoutineHandle xHandle, unsigned
portBASE_TYPE uxIndex )
    {
        // Co-routines must start with a call to crSTART().
        crSTART( xHandle );

        for( ;; )
        {
            // Delay for a fixed period.
            crDELAY( xHandle, 10 );

            // Flash an LED.
            vParTestToggleLED( 0 );
        }

        // Co-routines must end with a call to crEND().
        crEND();
    }
```

   Thats it!

2. **Scheduling the co-routine**

   Co-routines 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 co-routine and start the scheduler**

   The co-routine 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 co-routines from the same function. Each co-routine will flash a different LED at a different rate. The index parameter can be used to distinguish between the co-routines from within the co-routine function itself.

   This time we are going to create 8 co-routines 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 co-routine 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 )
    {
        // Co-routines 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 co-routine 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 ] );
        }

        // Co-routines must end with a call to crEND().
        crEND();
    }
```

Demo Application Examples

Two files are included in the download that demonstrate using co-routines with queues:

1. **crflash.c**

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

2. **`crhook.c`**

   Demonstrates passing data from a interrupt to a co-routine. 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 co-routine 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 co-routines 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):
   i. Replace the reference to file `FreeRTOS/Demo/Common/Minimal/flash.c` with `FreeRTOS/Demo/Common/Minimal/crflash.c`.
   ii. Add the file `FreeRTOS/Source/croutine.c` to the build.
3. In `main.c`:
   i. Include the header file `croutine.h` which contains the co-routine macros and function prototypes.
   ii. Replace the inclusion of `flash.h` with `crflash.h`.
   iii. Remove the call to the function that creates the flash tasks `vStartLEDFlashTasks() ....`
   iv. ... and replace it with the function that creates the flash co-routines `vStartFlashCoRoutines( n )`, where n is the number of co-routines that should be created. Each co-routine flashes a different LED at a different rate.
   v. Add an idle hook function that schedules the co-routines as:

```
  vi.         void vApplicationIdleHook( void )
 vii.         {
viii.             vCoRoutineSchedule( void );
  ix.         }
```

   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 co-routines 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`.

RTOS Kernel Utilities

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 co-routines 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.

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.

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.

Trace Utility

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

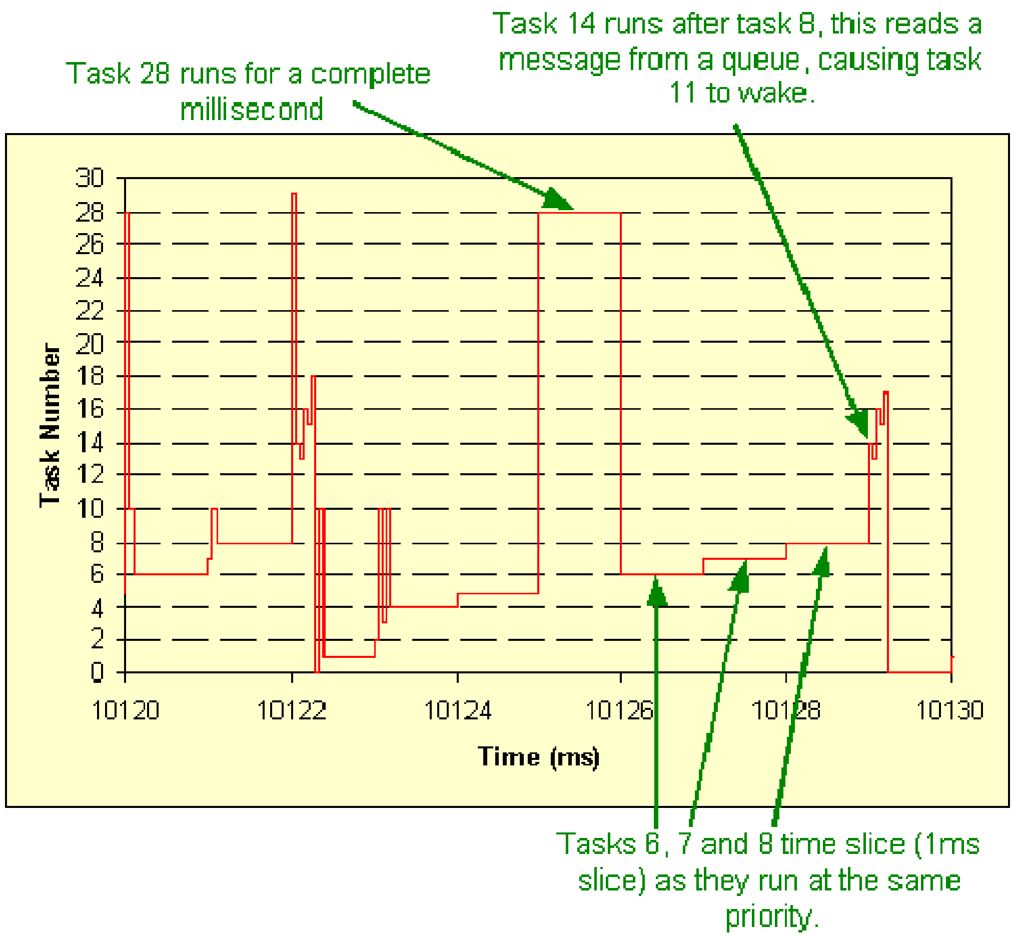
It records the sequence in which tasks are given microcontroller processing time.

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.

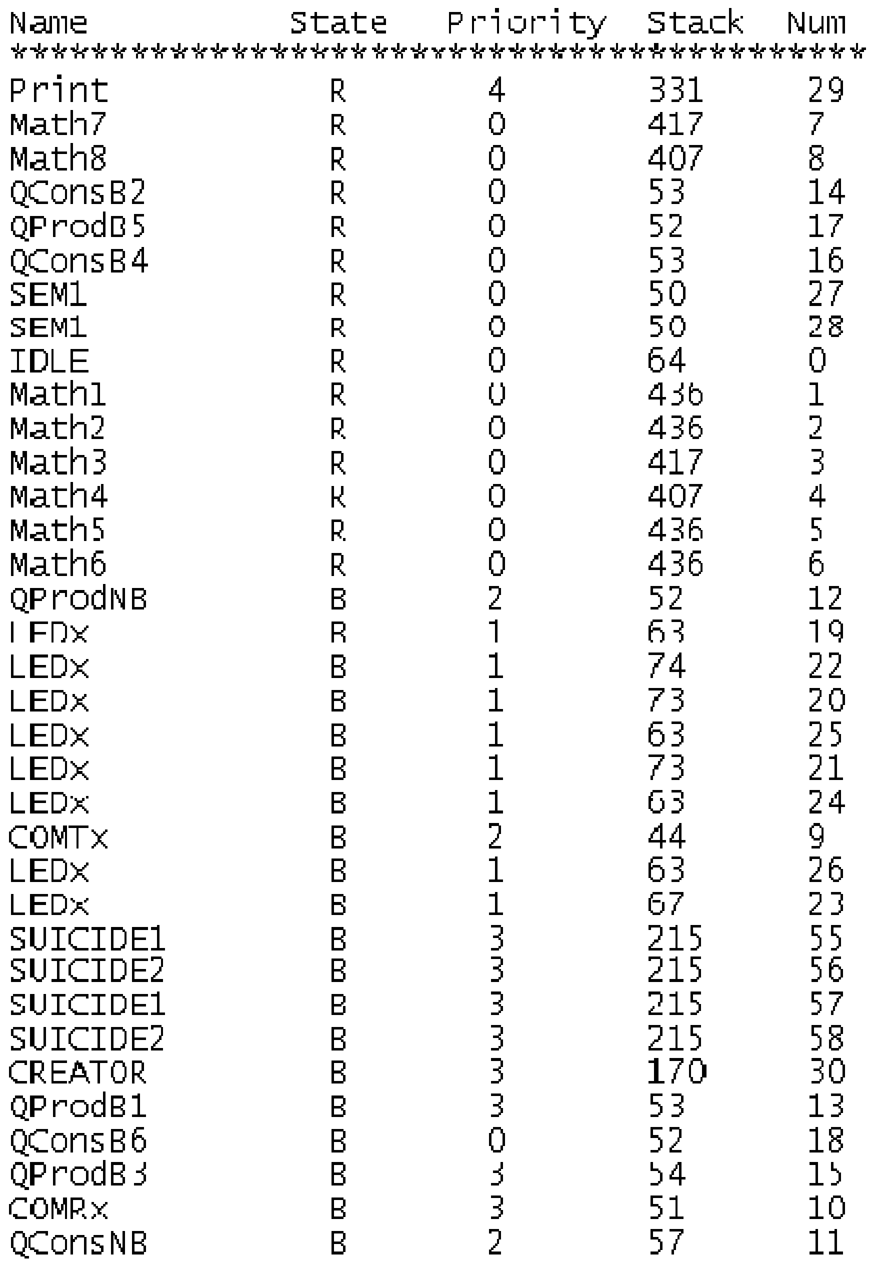
The completed trace buffer can be stored to disk for offline examination. The DOS/Windows utility tracecon.exe converts the stored buffer to a tab delimited text file. This can then be opened and examined in a spread sheet application.

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.



```
Name            State   Priority  Stack   Num
*********************************************
Print             R        4       331     29
Math7             R        0       417     7
Math8             R        0       407     8
QConsB2           R        0       53      14
QProdB5           R        0       52      17
QConsB4           R        0       53      16
SEM1              R        0       50      27
SEM1              R        0       50      28
IDLE              R        0       64      0
Math1             R        0       436     1
Math2             R        0       436     2
Math3             R        0       417     3
Math4             R        0       407     4
Math5             R        0       436     5
Math6             R        0       436     6
QProdNB           B        2       52      12
LEDx              R        1       63      19
LEDx              B        1       74      22
LEDx              B        1       73      20
LEDx              B        1       63      25
LEDx              B        1       73      21
LEDx              B        1       63      24
COMTx             B        2       44      9
LEDx              B        1       63      26
LEDx              B        1       67      23
SUICIDE1          B        3       215     55
SUICIDE2          B        3       215     56
SUICIDE1          B        3       215     57
SUICIDE2          B        3       215     58
CREATOR           B        3       170     30
QProdB1           B        3       53      13
QConsB6           B        0       52      18
QProdB3           B        3       54      15
COMRx             B        3       51      10
QConsNB           B        2       57      11
```

In this example, it can be seen that tasks 6, 7, 8 and 14 are all running at priority 0. They therefore time slice between themselves and the other priority 0 tasks (including the idle task). Task 14 reads a message from a queue (see BlockQ.c in the demo application). This frees a space on the queue. Task 13 was blocked waiting for a space to be available, so now wakes, posts a message then blocks again.

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.

Source Organization

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 contained ***in just 3 files*** (4 if co-routines 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 co-routines 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.

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 microcontroller ports (four if co-routines 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/MSP430F449 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 microcontroller 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.

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 sub directories 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 microcontrollers 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 microcontrollers and can be ignored.*

- And so on for all the ports ...

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.

The complete directory tree

The entire directory tree is shown below - including a brief description of each node:

```
FreeRTOS
 ¦
 +-Demo             Contains all directories associated with the demo
application, one sub directory per port.
 ¦   ¦
 ¦   +-Common                   Demo application files common to all
ports
 ¦   ¦   +-Minimal                 Minimal version of common demo
application files
 ¦   ¦   +-Full                    Full version of common demo
application files
 ¦   ¦   +-include                 Demo application header files
 ¦   ¦
 ¦   +-ARM7_AtmelSAM7S64_IAR    Demo application source code for the
AT91SAM7S64 port using the IAR compiler
 ¦   ¦   +-ParTest
 ¦   ¦   +-serial
 ¦   ¦
 ¦   +-ARM7_AT91SAM7X256_Eclipse Demo application source code for the
AT91SAM7X port using GCC and Eclipse
 ¦   ¦   +-Webserver
 ¦   ¦   +-USB
 ¦   ¦
 ¦   +-ARM7_LPC2106_GCC         Demo application source code for the
LPC2106 port
 ¦   ¦   +-ParTest
 ¦   ¦   +-serial
 ¦   ¦
 ¦   +-ARM7_LPC2129_Keil        Demo application source code for the
LPC2109 port using the Keil compiler
 ¦   ¦   +-ParTest
 ¦   ¦   +-serial
 ¦   ¦
 ¦   +-ARM7_AT91FR40008_GCC     Demo application source code for the
AT91 port using the GCC compiler
 ¦   ¦   +-ParTest
 ¦   ¦   +-serial
 ¦   ¦
 ¦   +-ARM7_STR75x_IAR          Demo application source code for the
STR75x port using the IAR compiler
 ¦   ¦   +-ParTest
 ¦   ¦   +-serial
 ¦   ¦
 ¦   +-ARM7_STR71x_IAR          Demo application source code for the
STR71x port using the IAR compiler
 ¦   ¦   +-ParTest
 ¦   ¦   +-serial
 ¦   ¦
 ¦   +-ARM9_STR91X_IAR          Demo application source code for the
STR912 port using the IAR compiler
 ¦   ¦   +-ParTest
 ¦   ¦   +-serial
 ¦   ¦   +-webserver
 ¦   ¦
 ¦   +-ARM7_LPC2138_Rowley      Demo application project for the
LPC2138 ARM7 port using the CrossStudio tools
 ¦   ¦
 ¦   +-ARM7_LPC2138_Eclipse     Demo application project for the
LPC2138 ARM7 port using GCC and Eclipse.
 ¦   ¦
 ¦   +-HCS12_CodeWarrior_small  Demo application source code for the
HCS12 port small memory model
 ¦   ¦   +-ParTest
 ¦   ¦   +-serial
 ¦   ¦
 ¦   +-HCS12_CodeWarrior_banked Demo application source code for the
HCS12 port banked memory model
 ¦   ¦   +-ParTest
 ¦   ¦   +-serial
 ¦   ¦
 ¦   +-H8S                      Demo application source code for the
H8/S port
 ¦   ¦   +-ParTest
 ¦   ¦   +-serial
 ¦   ¦
 ¦   +-MSP430                   Demo application source code for the
MSP430F449 port
 ¦   ¦   +-ParTest
 ¦   ¦   +-serial
 ¦   ¦
 ¦   +-AVR32_UC3                Demo application source code for the
AVR32 port using the GCC and IAR compiler
 ¦   ¦   +-AT32UC3A
 ¦   ¦       +-GCC              Make file to build the standard AVR32
demo using the GCC compiler
 ¦   ¦       +-IAR              Make file to build the standard AVR32
demo using the IAR compiler
 ¦   ¦
 ¦   +-AVR_ATMega323_WinAVR     Demo application source code for the
AVR port using the GCC compiler
 ¦   ¦   +-ParTest
 ¦   ¦   +-serial
 ¦   ¦
 ¦   +-AVR_ATMega323_IAR        Demo application source code for the
AVR port using the IAR compiler
 ¦   ¦   +-ParTest
 ¦   ¦   +-serial
 ¦   ¦
 ¦   +-PIC32_MPLAB              Demo application source code for PIC32
(MIPS M4K) port using MPLAB and the C32 compiler
 ¦   ¦   +-ParTest
 ¦   ¦   +-serial
 ¦   ¦
 ¦   +-PIC18_MPLAB              Demo application source code for PIC18
port using the MPLAB compiler
 ¦   ¦   +-ParTest
 ¦   ¦   +-serial
 ¦   ¦
 ¦   +-PIC18_WIZC               Demo application source code for PIC18
port using the WizC compiler
```

```
 ¦   ¦   +-ParTest
 ¦   ¦   +-serial
 ¦   ¦
 ¦   +-PIC24_MPLAB              Demo application source code for PIC24
port using the MPLAB compiler
 ¦   ¦   +-ParTest
 ¦   ¦   +-serial
 ¦   ¦
 ¦   +-dsPIC_MPLAB              Demo application source code for dsPIC
port using the MPLAB compiler
 ¦   ¦   +-ParTest
 ¦   ¦   +-serial
 ¦   ¦
 ¦   +-Cygnal                   Demo application source code for Cygnal
(Silicon labs) 8051 port
 ¦   ¦   +-ParTest
 ¦   ¦   +-serial
 ¦   ¦
 ¦   +-Flshlite                 Demo application source for Flashlite
186 port
 ¦   ¦   +-FileIO
 ¦   ¦   +-serial
 ¦   ¦   +-ParTest
 ¦   ¦
 ¦   +-Microblaze               Demo application source for Xilinx
Microblaze port
 ¦   ¦   +-serial
 ¦   ¦   +-ParTest
 ¦   ¦
 ¦   +-CORTEX_STM32F103_IAR     Demo application source for the
STM32F103 Cortex-M3 port using the IAR tools
 ¦   ¦
 ¦   +-CORTEX_STM32F103_Primer_GCC Demo application source for the
STM32F103 Cortex-M3 port using the GCC and RIDE tools
 ¦   ¦
 ¦   +-CORTEX_LM3S102_GCC       Demo application source for the LM3S102
Cortex-M3 port using the GCC tools
 ¦   ¦
 ¦   +-CORTEX_LM3S102_Keil      Demo application source for the LM3S102
Cortex-M3 port using the Keil/RVDS tools
 ¦   ¦
 ¦   +-CORTEX_LM3S102_Rowley    Demo application source for the LM3S102
Cortex-M3 port using Rowley CrossWorks
 ¦   ¦
 ¦   +-CORTEX_LM3S316_IAR       Demo application source for the LM3S314
Cortex-M3 port using IAR
 ¦   ¦
 ¦   +-CORTEX_LM3S811_GCC       Demo application source for the LM3S811
Cortex-M3 port using the GCC tools
 ¦   ¦
 ¦   +-CORTEX_LM3S811_IAR       Demo application source for the LM3S811
Cortex-M3 port using IAR
 ¦   ¦
 ¦   +-CORTEX_LM3S811_KEIL      Demo application source for the LM3S811
Cortex-M3 port using Keil
 ¦   ¦
 ¦   +-CORTEX_LM3Sxxxx_IAR_Keil Demo application source for the 
LM3S6965, LM3S2965, LM3S1962 and LM3S8962 Cortex-M3 port using IAR and 
Keil
 ¦   ¦
 ¦   +-CORTEX_LM3Sxxxx_Eclipse  Demo application source for the 
LM3S6965, LM3S2965, LM3S1962 and LM3S8962 Cortex-M3 port using Eclipse 
with GCC
 ¦   ¦
 ¦   +-uIP_Demo_IAR_ARM7        Embedded WEB server demo using uIP and 
the IAR development tools
 ¦   ¦
 ¦   +-lwIP_Demo_Rowley_ARM7    Embedded WEB server demo using lwIP and 
the Rowley development tools
 ¦   ¦
 ¦   +-lwIP_AVR32_UC3           Embedded WEB and TFTP server demo using 
lwIP and GCC on the AT32UC3A0512 AVR32
 ¦   ¦
 ¦   +-uIP_Demo_Rowley_ARM7     Embedded WEB server demo using uIP and 
the Rowley development tools
 ¦   ¦
 ¦   +-WizNET_DEMO_GCC_ARM7     Embedded WEB server demo application 
using the WizNET coprocessor with I2C interface
 ¦   ¦
 ¦   +-WizNET_DEMO_TERN_186     Embedded WEB server demo application 
using the WizNET coprocessor with mapped interface
 ¦   ¦
 ¦   +-PC                       Demo application source for PC port
 ¦       +-FileIO
 ¦       +-ParTest
 ¦       +-serial
 ¦
 +-Source           Contains all directories associated with the 
scheduler source code
    ¦
    ¦               3 core scheduler files common to all ports (4 is 
using co-routines)
    ¦
     +-include                  Scheduler header files
     ¦
     +-portable                 Scheduler port layer for all ports
         ¦
         +-MemMang                  Sample memory allocators can be 
used for all ports
         ¦
         ¦
         +-GCC                      Scheduler port layer for ports 
using GCC compiler
         ¦   +-ATmega32                 Scheduler port files for AVR 
using GCC compiler
         ¦   +-MSP430F449               Scheduler port files for MSP430 
using GCC compiler
         ¦   +-ARM7_LPC2000             Scheduler port files for 
LPC2106 using GCC compiler
         ¦   +-ARM7_AT91FR40008         Scheduler port files for AT91 
using GCC compiler
         ¦   +-H8S2329                  Scheduler port files for H8/S 
using GCC compiler
```

```
         ¦   +-Microblaze               Scheduler port files for
Microblaze using GCC compiler
         ¦   +-ARM_MC3                  Scheduler port files for ARM
Cortex-M3 using GCC compiler
         ¦   +-AVR32_UC3                Scheduler port files for AVR32
AT32UC3A using GCC compiler
         ¦
         ¦
         +-RVDS                     Scheduler port layer for ports
using RVDS/Keil compiler
         ¦   +-ATmega32                 Scheduler port files for ARM
Cortex-M3 port
         ¦
         ¦
         +-IAR                      Scheduler port layer for ports
using IAR compiler
         ¦   +-ATmega32                 Scheduler port files for AVR
using IAR compiler
         ¦   +-AtmelSAM7S64             Scheduler port files for SAM7
using IAR compiler
         ¦   +-STR71x                   Scheduler port files for STR71x
using IAR compiler
         ¦   +-STR91x                   Scheduler port files for STR91x
using IAR compiler
         ¦   +-ARM_CM3                  Scheduler port files for
Cortex-M3 using IAR compiler
         ¦   +-LPC2000                  Scheduler port files for
Philips LPC2000 using IAR compiler
         ¦   +-AVR32_UC3                Scheduler port files for AVR32
AT32UC3A using the IAR compiler
         ¦
         ¦
         +-Keil                     Scheduler port layer for ports
using Keil compiler
         ¦   +-ARM7                     Scheduler port files for ARM7
using Keil compiler
         ¦
         ¦
         +-CodeWarrior              Scheduler port layer for ports
using CodeWarrior compiler
         ¦   +-HCS12                    Scheduler port files for HCS12
using CodeWarrior compiler
         ¦
         ¦
         +-MPLAB                    Scheduler port layer for ports
using Microchip C18 compiler
         ¦   +-PIC18                    Scheduler port files for PIC18
using Microchip C18 compiler
         |   +-PIC24_dsPIC              Scheduler port files for PIC24,
dsPIC
         |   +-PIC32                    Scheduler port files for PIC32
         ¦
         ¦
         +-SDCC                     Scheduler port layer for ports
using SDCC compiler
         ¦   +-Cygnal                   Scheduler port files for Cygnal
8051 using SDCC compiler
```

```
         ¦
         ¦
         +-oWatcom                 Scheduler port layer for ports
using Open Watcom compiler
         ¦   +-Flsh186                 Scheduler port files for
Flashlite 186 port
         ¦   +-common                  Scheduler port files common to
all OW DOS based ports
         ¦   +-PC                      Scheduler port files for PC
port
         ¦
         ¦
         +-Paradigm                Scheduler port layer for ports
using Paradigm compiler
         ¦   +-Tern_EE                 Scheduler port files for Tern
186 port
         ¦      +-small                   Small memory model port for
Tern 186
         ¦      +-large_untested          Large memory model port for
Tern 186 - untested!
         ¦
         ¦
         +-BCC                     Scheduler port layer for ports
using Borland compiler
             +-Flsh186                 Scheduler port files for
Flashlite 186 port
             +-common                  Scheduler port files common to
all BCC DOS based ports
             +-PC                      Scheduler port files for PC
port
```

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

---

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.

request every 10ms exactly, and the resultant command shall be transmitted within 5ms of this request. The control algorithm is reliant on accurate timing, it is therefore paramount that these timing requirements are met.

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.

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.

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.

TCP/IP Interface

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

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.

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.

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 co-routine 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.

File	**Features Demonstrated**
main.c	• Starting/Stopping the kernel • Using the trace visualisation utility • Allocation of priorities
dynamic.c	• Passing parameters into a task • Dynamically changing priorities • Suspending tasks • Suspending the scheduler
BlockQ.c	• Inter-task communications • Blocking on queue reads • Blocking on queue writes • Passing parameters into a task • Pre-emption • Creating tasks
ComTest.c	• Serial communications • Using queues from an ISR • Using semaphores from an ISR • Context switching from an ISR • Creating tasks

CRFlash.c	• Creating co-routines • Using the index of a co-routine • Blocking on a queue from a co-routine • Communication between co-routines
CRHook.c	• Creating co-routines • Passing data from an ISR to a co-routine • Tick hook function • Co-routines blocking on queues
Death.c	• Dynamic creation of tasks (at run time) • Deleting tasks • Passing parameters to tasks
Flash.c	• Delaying • Passing parameters to tasks • Creating tasks
Flop.c	• Floating point math • Time slicing • Creating tasks
Integer.c	• Time slicing • Creating tasks
PollQ.c	• Inter-task communications • Manually yielding processor time • Polling a queue for space to write

	• Polling a queue for space to read • Pre-emption • Creating tasks
Print.c	• Queue usage
Semtest.c	• Binary semaphores • Mutual exclusion • Creating tasks

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.

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.

---

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).

---

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 9way '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.

---

crflash.c

This demo application file demonstrates the use of queues to pass data between co-routines and the use of the co-routine index parameter.

N 'fixed delay' co-routines are created that just block for a fixed period then post the number of an LED onto a queue. Each such co-routine uses its index parameter as an index into array in order to obtain the block period and LED that is flashed. A single 'flash' co-routine 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' co-routines are created with co-routine priority 0. The flash co-routine is created with co-routine 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' co-routine 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.

---

crhook.c

This demo file demonstrates how to send data between an ISR and a co-routine. A tick hook function is used to periodically pass data between the RTOS tick and a set of 'hook' co-routines.

hookNUM_HOOK_CO_ROUTINES co-routines are created. Each co-routine 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' co-routines 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 co-routine then an error is latched.

This demo relies on each 'hook' co-routine 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.

---

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.

---

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.

---

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.

---

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 pre-empted 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.

---

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.

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.

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.

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.

Embedded TCP/IP Examples

There are currently eight embedded Ethernet TCP/IP example programs. Each demo creates an embedded WEB server within a fully preemptive multitasking project. Three of the examples include a live clickable demonstration.

1. Open source uIP TCP/IP stack on an LM3S6965 and LM3S8962 Cortex-M3:

   Permits commands to be sent to the target from a WEB browser. Also permits the display of dynamically generated run time data. A version for use with Eclipse is also available

2. AVR32 AT32UC3A lwIP WEB and TFTP server:

   This example uses lwIP to create both a simple WEB and TFTP server on the AVR32 flash microcontroller.

3. Open source uIP TCP/IP stack on an AT91SAM7X:

   Includes a simple interrupt driven driver for the SAM7X integrated EMAC peripheral.

4. The open source uIP TCP/IP stack on an AT91SAM7X again - this time using Eclipse:

A simple mouse driver is provided along with the WEB server demo.

5. Open source lwIP TCP/IP stack on an AT91SAM7X:

   Includes a more comprehensive interrupt driven driver for the SAM7X integrated EMAC peripheral.

6. Open source uIP TCP/IP stack on an LPC2368:

   Demonstrates control over the target hardware IO from the served WEB pages.

7. LPC2368 project again - this time using Eclipse:

   The LPC2368 embedded WEB server example created using completely open source development tools.

8. Open source uIP TCP/IP stack on an LPC2124:

   Includes a polled mode Crystal LAN CS8900 driver (thanks to Paul Curtis).

9. WizNET hardware TCP/IP stack - I^2C interface:

   This example uses a TCP/IP coprocessor to produce an embedded WEB server through the I^2C port!

10. Open source uIP TCP/IP stack and lwIP TCP/IP stack on an STR912 (ARM9):

    This demo includes options to use either the uIP or the lwIP stack, this time targeted at an ARM9 processor.

11. WizNET hardware TCP/IP stack - memory mapped interface:

    This example uses the same TCP/IP coprocessor, but with a memory mapped interface on a Tern E-Engine controller.

API

Upgrading FreeRTOS to V4.x.x

FreeRTOS V4.0.0 introduces co-routine functionality. As the use of co-routines is optional FreeRTOS V4.0.0 is essentially backward compatable with V3.x.x releases.

To upgrade a project created with FreeRTOS V3.x.x for use with V4.0.0 simply ensure the following definitions are added to the projects FreeRTOSConfig.h file:

- `#define configUSE_TICK_HOOK 0`
- `#define configUSE_CO_ROUTINES 0`
- `#define configMAX_CO_ROUTINE_PRIORITIES 1`

See the Customisation pages for information on using these new definitions, and the co-routine documentation pages for information on using co-routines.

Upgrading V4.0.5 to V4.1.0

- The definition `errQUEUE_EMPTY` has been redefined to 0.
- Prior to V4.1.0, under certain documented circumstances, it was possible for xQueueSend() and xQueueReceive() to return without having completed and without their block time expiring. The block time effectively stated a maximum block time, and the return value of the function needed to be checked to determine the reason for returning. This is no longer the case as the functions will only return once the block time has expired or they are able to complete their operation. It is therefore no longer necessary to wrap calls within loops.

Upgrading V4.2.1 to V4.3.0

V4.3.0 introduces the new configKERNEL_INTERRUPT_PRIORITY parameter to the Cortex-M3/IAR, dsPIC and PIC24 ports. This will be rolled out to other ports in time.

Users of any of the above mentioned ports must ensure they set interrupt priorities in accordance with the instruction provided on the Customisation page.

Upgrading from V4.4.0 to V4.5.0

V4.5.0 introduces the xQueueSendToFront(), xQueueSendToBack() xQueuePeek() and mutex (as opposed to binary semaphore) functionality.

The xQueueSend() syntax is maintained to ensure backward compatibility, but new applications should use the new and equivalent xQueueSendToBack() in its place.

A new configuration item configUSE_MUTEXES has been introduced. This must be set to 1 in order for the new mutex functionality to be available.

API Usage

General Information

- Only those API functions specifically designated for use from within an ISR should be used from within an ISR.
- Tasks and co-routines use different API functions to access queues. A queue cannot be used to communicate between a task and a co-routine 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".

Task API

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

Co-Routine API

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

- taskYIELD() - will yield the task in which the co-routines are running.
- taskENTER_CRITICAL().
- taskEXIT_CRITICAL().
- vTaskStartScheduler() - this is still used to start the scheduler even if the application only includes co-routines and no tasks.
- vTaskSuspendAll() - can still be used to lock the scheduler.
- xTaskResumeAll()
- xTaskGetTickCount()
- uxTaskGetNumberOfTasks()

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 */
```

'config' Parameters

configUSE_PREEMPTION

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

configUSE_IDLE_HOOK

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

configUSE_TICK_HOOK

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

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.

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.

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.

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.

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.

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.

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.

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 16bit type. Defining configUSE_16_BIT_TICKS as 0 causes portTickType to be defined (typedef'ed) as an unsigned 32bit type.

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.

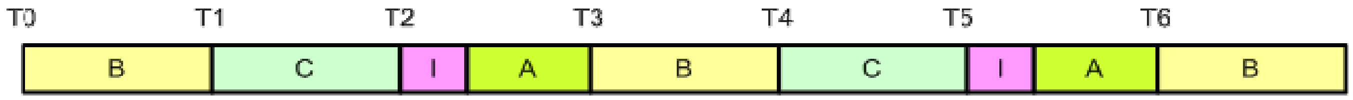
configIDLE_SHOULD_YIELD

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 behaviour 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 behaviour can however have undesirable effects (depending on the needs of your application) as depicted below:



This diagram 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 T0, T1, ..., T6. 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.

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.

configUSE_CO_ROUTINES

Set to 1 to include co-routine functionality in the build, or 0 to omit co-routine functionality from the build. To include co-routines croutine.c must be included in the project.

configMAX_CO_ROUTINE_PRIORITIES

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

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.org 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).

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
```

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().

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.

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.

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.

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.

Task Creation

Detailed Description

xTaskHandle

task. h

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.

xTaskCreate

task. h

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

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 );
 }
```

vTaskDelete

task. h

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

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

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 );
 }
```

Task Control

vTaskDelay

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

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

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.
     }
 }
```

vTaskDelayUntil

task. h

```
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.

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.
    }
}
```

uxTaskPriorityGet

task. h

```
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.

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.
     }
 }
```

vTaskPrioritySet

task. h

```
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.

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 );
 }
```

vTaskSuspend

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

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

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.
 }
```

vTaskResume

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

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

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.
 }
```

vTaskResumeFromISR

task. h

```
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.

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 synchronise 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 synchronisation 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();
     }
 }
```

Kernel Control

Detailed Description

taskYIELD

task. h

Macro for forcing a context switch.

taskENTER_CRITICAL

task. h

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!

taskEXIT_CRITICAL

task. h

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!

taskDISABLE_INTERRUPTS

task. h

Macro to disable all maskable interrupts.

taskENABLE_INTERRUPTS

task. h

Macro to enable microcontroller interrupts.

vTaskStartScheduler

task. h

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

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 ()
 }
```

vTaskEndScheduler

task. h

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

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.
 }
```

vTaskSuspendAll

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

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 ();
     }
 }
```

xTaskResumeAll

task. h
`portBASE_TYPE xTaskResumeAll( void );`

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 ();
         }
     }
 }
```

Task Utilities

xTaskGetCurrentTaskHandle

task.h
```
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.

---

xTaskGetTickCount

task.h
```
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.

---

xTaskGetSchedulerState

task.h

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

Returns:

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

---

uxTaskGetNumberOfTasks

task.h

```
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.

---

vTaskList

task.h

```
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.

---

vTaskStartTrace

task.h

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

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.

---

ulTaskEndTrace

task.h
```
unsigned portLONG ulTaskEndTrace( void );
```

Stops a kernel activity trace. See vTaskStartTrace().

Returns:

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

Queue Management

Detailed Description

---

uxQueueMessagesWaiting

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

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.

---

vQueueDelete

queue.h
```
void vQueueDelete( xQueueHandle xQueue );
```

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.

xQueueCreate

queue. h

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

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.
 }
```

xQueueSend

queue.h

```
 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().

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.
 }
```

xQueueSendToBack

Only available from FreeRTOS.org V4.5.0 onwards.

queue.h

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

```
                              );
```

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

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.
 }
```

xQueueSendToFront

Only available from FreeRTOS.org V4.5.0 onwards.

queue.h

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

This is a macro that calls xQueueGenericSend().

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 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( 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.
 }
```

xQueueReceive

queue. h

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

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

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 timeout).

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.
 }
```

xQueuePeek

Only available from FreeRTOS.org V4.5.0 onwards.

queue.h

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

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

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.
 }
```

xQueueSendFromISR

queue.h

```
 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.

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();
    }
 }
```

xQueueSendToBackFromISR

Only available from FreeRTOS.org V4.5.0 onwards.

queue.h

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

This is a macro that calls xQueueGenericSendFromISR().

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();
    }
 }
```

xQueueSendToFrontFromISR

Only available from FreeRTOS.org V4.5.0 onwards.

queue.h

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

This is a macro that calls xQueueGenericSendFromISR().

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();
    }
 }
```

xQueueReceiveFromISR

queue. h

```
 portBASE_TYPE xQueueReceiveFromISR(
                                       xQueueHandle pxQueue,
                                       void *pvBuffer,
                                       portBASE_TYPE *pxTaskWoken
                                   );
```

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 ();
    }
 }
```

Semaphores

SemaphoreCreateBinary

semphr. h
`vSemaphoreCreateBinary( xSemaphoreHandle xSemaphore )`

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 synchronisation (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 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.

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.
    }
 }
```

xSemaphoreCreateMutex

Only available from FreeRTOS.org V4.5.0 onwards.

semphr. h

```
xSemaphoreHandle xQueueCreateMutex( void )
```

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.
   }
}
```

xSemaphoreTake

semphr. h

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

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 optimisation 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 timeout).

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.
        }
    }
 }
```

xSemaphoreGive

semphr. h

```
xSemaphoreGive( xSemaphoreHandle xSemaphore )
```

Macro to release a semaphore. The semaphore must of been created using vSemaphoreCreateBinary (), and obtained using sSemaphoreTake ().

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.
            }
        }
    }
 }
```

xSemaphoreGiveFromISR

semphr. h

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

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.
 }
```

Co-routine specific

xCoRoutineCreate

croutine.h

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

Create a new co-routine and add it to the list of co-routines that are ready to run.

Parameters:

pxCoRoutineCode	Pointer to the co-routine function. Co-routine functions require special syntax - see the co-routine section of the WEB documentation for more information.
uxPriority	The priority with respect to other co-routines at which the co-routine will run..
uxIndex	Used to distinguish between different co-routines that execute the same function. See the example below and the co-routine section of the WEB documentation for further information.

Returns:

pdPASS if the co-routine 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 co-routines 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 co-routine with a call to crSTART();
     crSTART( xHandle );

     for( ;; )
     {
         // This co-routine just delays for a fixed period, then
toggles
         // an LED.  Two co-routines are created using this function,
so
         // the uxIndex parameter is used to tell the co-routine 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 co-routine with a call to crEND();
     crEND();
 }
 // Function that creates two co-routines.
 void vOtherFunction( void )
 {
 unsigned char ucParameterToPass;
 xTaskHandle xHandle;

     // Create two co-routines 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 co-routines are referenced. The co-routine handle is automatically passed into each co-routine function.

xCoRoutineCreate

croutine.h

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

Create a new co-routine and add it to the list of co-routines that are ready to run.

Parameters:

pxCoRoutineCode	Pointer to the co-routine function. Co-routine functions require special syntax - see the co-routine section of the WEB documentation for more information.
uxPriority	The priority with respect to other co-routines at which the co-routine will run..
uxIndex	Used to distinguish between different co-routines that execute the same function. See the example below and the co-routine section of the WEB documentation for further information.

Returns:

pdPASS if the co-routine 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 co-routines 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 co-routine with a call to crSTART();
     crSTART( xHandle );

     for( ;; )
     {
         // This co-routine just delays for a fixed period, then
toggles
         // an LED.  Two co-routines are created using this function,
so
         // the uxIndex parameter is used to tell the co-routine 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 co-routine with a call to crEND();
     crEND();
 }

 // Function that creates two co-routines.
 void vOtherFunction( void )
 {
 unsigned char ucParameterToPass;
 xTaskHandle xHandle;

     // Create two co-routines 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 co-routines are referenced. The co-routine handle is automatically passed into each co-routine function.

crDELAY

croutine.h

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

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

Delay a co-routine for a fixed period of time.

crDELAY can only be called from the co-routine function itself - not from within a function called by the co-routine function. This is because co-routines do not maintain their own stack.

Parameters:

xHandle	The handle of the co-routine to delay. This is the xHandle parameter of the co-routine function.
xTickToDelay	The number of ticks that the co-routine 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 co-routines 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 co-routine with a call to crSTART();
     crSTART( xHandle );

     for( ;; )
     {
        // Delay for 200ms.
        crDELAY( xHandle, xDelayTime );

        // Do something here.
     }
```

```
    // Must end every co-routine with a call to crEND();
    crEND();
}
```

crQUEUE_SEND

croutine.h

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

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 co-routine equivalent to the xQueueSend() and xQueueReceive() functions used by tasks.

crQUEUE_SEND and crQUEUE_RECEIVE can only be used from a co-routine whereas xQueueSend() and xQueueReceive() can only be used from tasks. **Note** that co-routines can only send data to other co-routines. A co-routine cannot use a queue to send data to a task or visa versa.

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

See the co-routine section of the WEB documentation for information on passing data between tasks and co-routines and between ISR's and co-routines.

Parameters:

xHandle	The handle of the calling co-routine. This is the xHandle parameter of the co-routine 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.
xTickToDelay	The number of ticks that the co-routine 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 co-routines 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();
 }
```

crQUEUE_RECEIVE

croutine.h

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

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 co-routine equivalent to the xQueueSend() and xQueueReceive() functions used by tasks.

crQUEUE_SEND and crQUEUE_RECEIVE can only be used from a co-routine whereas xQueueSend() and xQueueReceive() can only be used from tasks. **Note** that co-routines can only send data to other co-routines. A co-routine cannot use a queue to send data to a task or visa versa.

crQUEUE_RECEIVE can only be called from the co-routine function itself - not from within a function called by the co-routine function. This is because co-routines do not maintain their own stack.

See the co-routine section of the WEB documentation for information on passing data between tasks and co-routines and between ISR's and co-routines.

Parameters:

xHandle	The handle of the calling co-routine. This is the xHandle parameter of the co-routine 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.
xTickToDelay	The number of ticks that the co-routine 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 co-routine 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 co-routines must be declared static if they must
maintain value across a blocking call.
 static portBASE_TYPE xResult;
 static unsigned portBASE_TYPE uxLEDToFlash;

    // All co-routines 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();
 }
```

crQUEUE_SEND_FROM_ISR

croutine.h

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

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 co-routine 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 co-routine 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 co-routine.

See the co-routine section of the WEB documentation for information on passing data between tasks and co-routines and between ISR's and co-routines.

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 co-routine 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 co-routine 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 co-routines 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 co-routines must end with a call to crEND().
     crEND();
 }

 // An ISR that uses a queue to send characters received on a serial
port to
 // a co-routine.
 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 co-
routine
         // to be woken (unblocked) then xCRWokenByPost will be set to
pdTRUE.
         // In this manner we can ensure that if more than one co-
routine 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 );
     }
 }
```

crQUEUE_RECEIVE_FROM_ISR

croutine.h

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

The macro's crQUEUE_SEND_FROM_ISR() and crQUEUE_RECEIVE_FROM_ISR() are the co-routine 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 co-routine 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 co-routine (a co-routine posted to the queue).

See the co-routine section of the WEB documentation for information on passing data between tasks and co-routines and between ISR's and co-routines.

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 co-routine may be blocked waiting for space to become available on the queue. If crQUEUE_RECEIVE_FROM_ISR causes such a co-routine 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 co-routine 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 co-routine is blocked and must
therefore
 // be declared static.
 static portCHAR cCharToTx = 'a';
 portBASE_TYPE xResult;

     // All co-routines 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 co-routines 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 co-
routine
         // is woken by the post - ensuring that only a single co-
routine is
         // woken no matter how many times we go around this loop.
         if( crQUEUE_RECEIVE_FROM_ISR( pxQueue, &cCharToTx,
&xCRWokenByPost ) )
         {
             SEND_CHARACTER( cCharToTx );
         }
     }
 }
```

vCoRoutineSchedule

croutine.h

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

Run a co-routine.

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

If an application comprises of both tasks and co-routines then vCoRoutineSchedule 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 );
    }
}
```

License Details

Options

FreeRTOS is licensed under a modified GPL and can be used in commercial applications under this license. An alternative commercial license option is also available in cases that:

- You cannot fulfil the requirements stated in the "Modified GPL license" column of the table below.
- You wish to receive direct technical support.
- You wish to have assistance with your development.

License feature comparison

	FreeRTOS Modified GPL license	OpenRTOS Commercial license
Is it free?	Yes	No
Can I use it in a commercial application?	Yes	Yes

Is it royalty free?	Yes	Yes
Do I have to open source my application code?	No	No
Do I have to open source my changes to the kernel?	Yes	No
Do I have to document that my product uses FreeRTOS.org?	Yes a WEB link to the FreeRTOS.org site is sufficient	No
Do I have to offer to provide the FreeRTOS.org code to users of my application?	Yes	No
Can I receive support on a commercial basis?	No	Yes

Commercial Licensing

OpenRTOS™ is a commercially licensed version of FreeRTOS.org. The OpenRTOS license does not contain any references to the GPL.

SAFERTOS™ is a derivative version of FreeRTOS.org that has been analyzed, documented and tested to meet the stringent requirements of the IEC 61508 safety standard. Complete safety lifecycle documentation artifacts have been created and independently audited to verify IEC 61508 SIL 3 conformance.

The SafeRTOS FAQ provides information on the differences between FreeRTOS.org, OpenRTOS and SafeRTOS.

Modified GPL (Open Source) Licensing

The FreeRTOS source code is licensed by the GNU General Public License (GPL) with an exception. The full text of the GPL is available here. The text of the exception is available at the bottom of this file.

The exceptions permits the source code of applications that use FreeRTOS solev through the API published on this WEB site to remain closed source, thus permitting the use of FreeRTOS in commercial applications without necessitating that the whole application be open sourced. The exception should only be used if you wish to combine FreeRTOS with a proprietary product and you comply with the terms stated in the exception itself.

The FreeRTOS download also includes demo application source code, some of which is provided by third parties *AND IS LICENSED SEPARATELY FROM FREERTOS*.

For the avoidance of any doubt refer to the comment included at the top of each source and header file for license and copyright information.

This is a list of files for which Richard Barry is not the copyright owner and are NOT COVERED BY THE GPL.

1. Various header files and linker scripts provided by silicon manufacturers and tool vendors that define processor specific memory addresses and utility macros and functions. Permission has been granted by the various copyright holders for these files to be included in the FreeRTOS download. Users must ensure license conditions are adhered to for any use other than compilation of the FreeRTOS demo application.
2. The uIP TCP/IP stack the copyright of which is held by Adam Dunkels. Users must ensure the open source license conditions stated at the top of each uIP source file is understood and adhered to.
3. The lwIP TCP/IP stack the copyright of which is held by the Swedish Institute of Computer Science. Users must ensure the open source license conditions stated at the top of each lwIP source file is understood and adhered to.
4. All files contained within the FreeRTOS\Demo\CORTEX_LM3S102_GCC\hw_include and FreeRTOS\Demo\CORTEX_LM3S316_IAR\hw_include directories. The copyright of these files is owned by Luminary Micro. Permission has been granted by Luminary Micro for these files to be included in the FreeRTOS download. Users must ensure the license conditions stated in the EULA.txt file located in the same directories is understood and adhered at all times for all files in those directories.
5. The files contained within FreeRTOS\Demo\WizNET_DEMO_TERN_186\tern_code, which are slightly modified versions of code provided by and copyright to Tern Inc.

Errors and omissions should be reported to Richard Barry, contact details for whom can be obtained from the Contact page.

GPL Exception

If you opt to use this exception you are encouraged to make a donation to the FreeRTOS project. The link in the frame on the left can be used for this purpose. Any such donation is entirely voluntary and does not result in any enhanced support or any warranty rights.

EXCEPTION TEXT:

Clause 1

Linking FreeRTOS statically or dynamically with other modules is making a combined work based on FreeRTOS. Thus, the terms and conditions of the GNU General Public License cover the whole combination.

As a special exception, the copyright holder of FreeRTOS gives you permission to link FreeRTOS with independent modules that communicate with FreeRTOS solely through the FreeRTOS API interface, regardless of the license terms of these independent modules, and to copy and distribute the resulting combined work under terms of your choice, provided that

1. **Every copy of the combined work is accompanied by a written statement that details to the recipient the version of FreeRTOS used and an offer by yourself to provide the FreeRTOS source code should the recipient request it.**
2. The combined work is not itself an RTOS, scheduler, kernel or related product.
3. The combined work is not itself a library intended for linking into other software applications.

Any FreeRTOS source code, whether modified or in it's original release form, or whether in whole or in part, can only be distributed by you under the terms of the GNU General Public License plus this exception. An independent module is a module which is not derived from or based on FreeRTOS.

Note that people who make modified versions of FreeRTOS are not obligated to grant this special exception for their modified versions; it is their choice whether to do so. The GNU General Public License gives permission to release a modified version without this exception; this exception also makes it possible to release a modified version which carries forward this exception.

Clause 2

FreeRTOS.org may not be used for any competitive or comparative purpose, including the publication of any form of run time or compile time metric, without the express permission of Richard Barry (this is the norm within the industry and is intended to ensure information accuracy).