FreeRTOS Implementation Modules

Real Time Engineers Ltd.
neatened by dormousebhu@smth

2010 年 4 月 16 日

目录

1 About This Article

This article will be helpful if you:

- wish to modify the FreeRTOS source code.
- port the real time kernel to another microcontroller or prototyping board.
- are new to using an RTOS and wish to get more information on their operation and implementation.

The FreeRTOS real time kernel has been ported to a number of different microcontroller architectures. The Atmel AVR port was chosen for this example due to:

- the simplicity of the AVR architecture.
- the free availability of the utilized WinAVR (GCC) development tools.
- the low cost of the STK500 prototyping board

2 RTOS Fundamentals

This section provides a very brief introduction to real time and multitasking concepts. These must be understood before reading section “RTOS Implementation”.

2.1 Multitasking

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

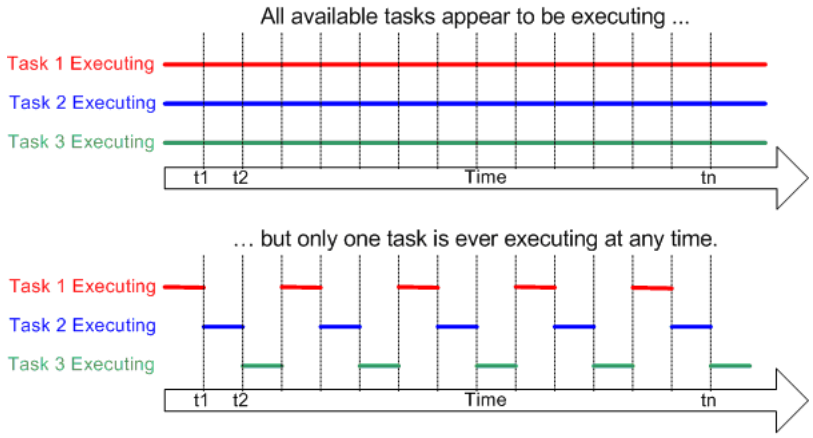
Each executing program is a task under control of the operating system. If an operating system can execute multiple tasks in this manner it is said to be multitasking.

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

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

2.1.1 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.



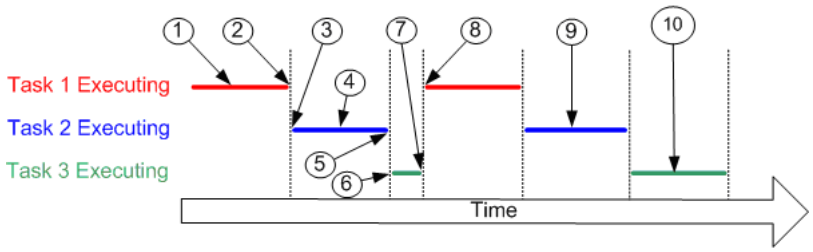
2.2 Scheduling

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:

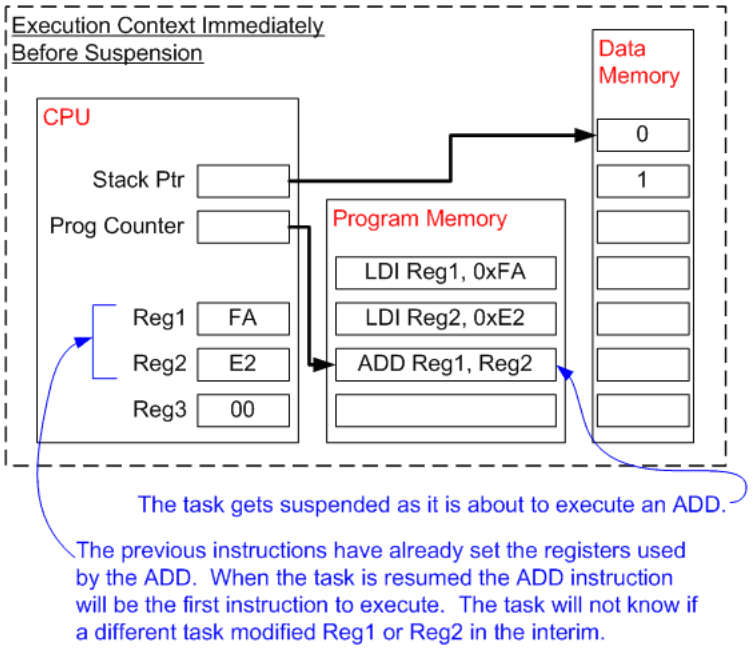
- At (1) task 1 is executing.
- At (2) the kernel suspends task 1 ...
- ... and at (3) resumes task 2.
- While task 2 is executing (4), it locks a processor peripheral for it's own exclusive access.
- At (5) the kernel suspends task 2 ...
- ... and at (6) resumes task 3.
- Task 3 tries to access the 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.

2.3 Context Switching

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

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



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

would result in an incorrect value.

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

2.4 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]
    }
}
```

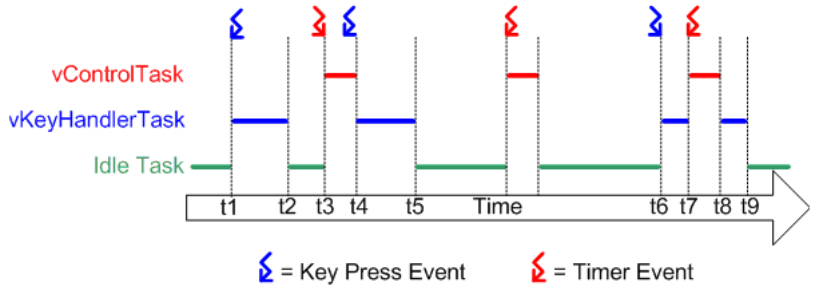
The software engineer must assign the control task the highest priority as:

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

2.5 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.

3 RTOS Implementation

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.

3.1 Building Blocks

3.1.1 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.

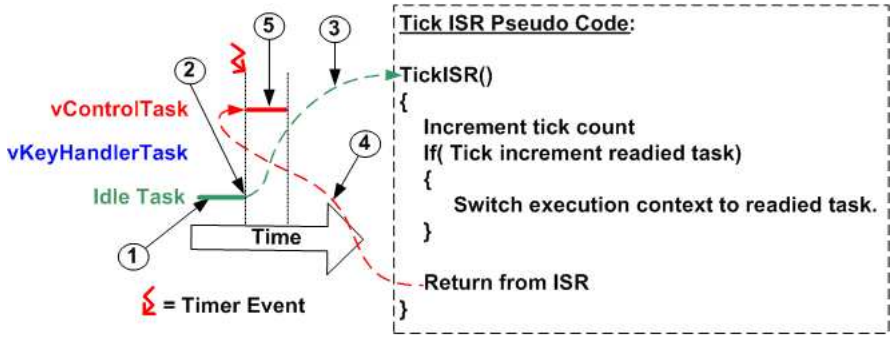
The example presented here uses the WinAVR development tools. WinAVR is a free Windows to AVR cross compiler based on GCC.

3.1.2 The RTOS Tick

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

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

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



Referring to the numbers in 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.

3.1.3 WinAVR 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.

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.

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

3.1.4 WinAVR 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.

3.1.5 FreeRTOS Tick Code

The actual source code used by the FreeRTOS AVR port is slightly different to the examples shown on the previous pages. **vPortYieldFromTick()** is itself implemented as a 'naked' function, and the context is saved and restored within **vPortYieldFromTick()** . It is done this way due to the implementation of 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. */
```

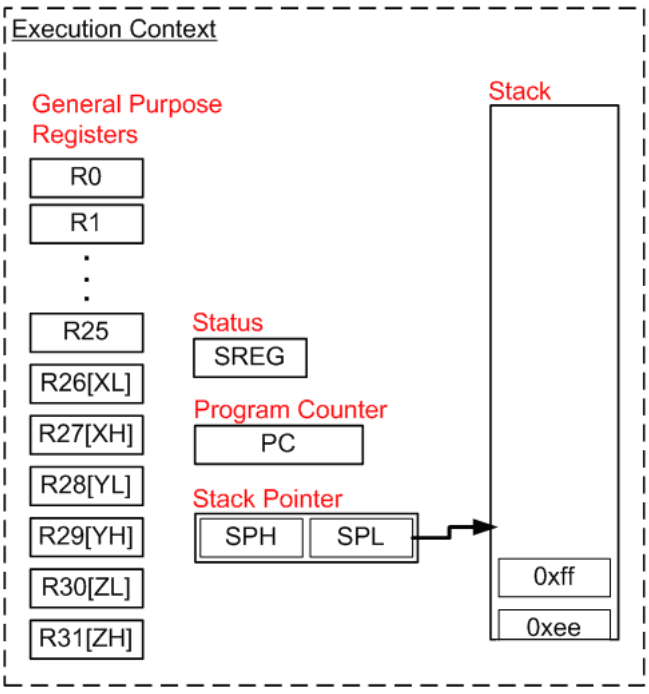
```
    vPortYieldFromTick();

    /* 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" );
}
```

```
/*-----------------------------------------------*/
```

3.1.6 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.



3.1.7 Saving the Context

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).

3.1.8 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).

3.2 Detailed Example

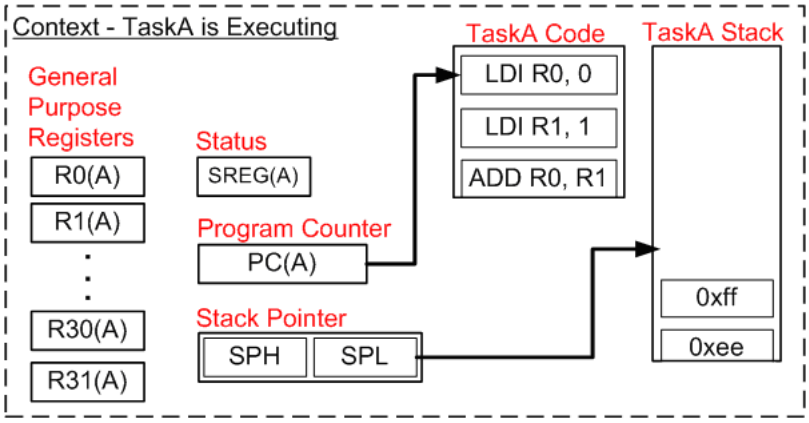
The final part of section 2 shows how these building blocks and source code modules are used to achieve a context switch on the AVR microcontroller. The example demonstrates in seven steps the process of switching from a lower priority task, called TaskA, to a higher priority task, called TaskB.

The source code is compatible with the WinAVR development tools.

3.2.1 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.

3.2.2 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.

3.2.3 Step 3 The RTOS tick interrupt executes

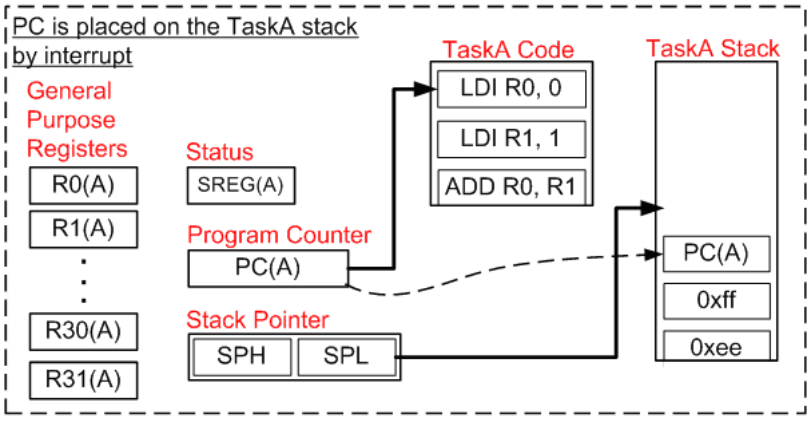
The ISR source code is given below. The comments have been removed to ease reading, but can be viewed on a previous page.

```
/* 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.



3.2.4 Step 4 Incrementing the Tick Count

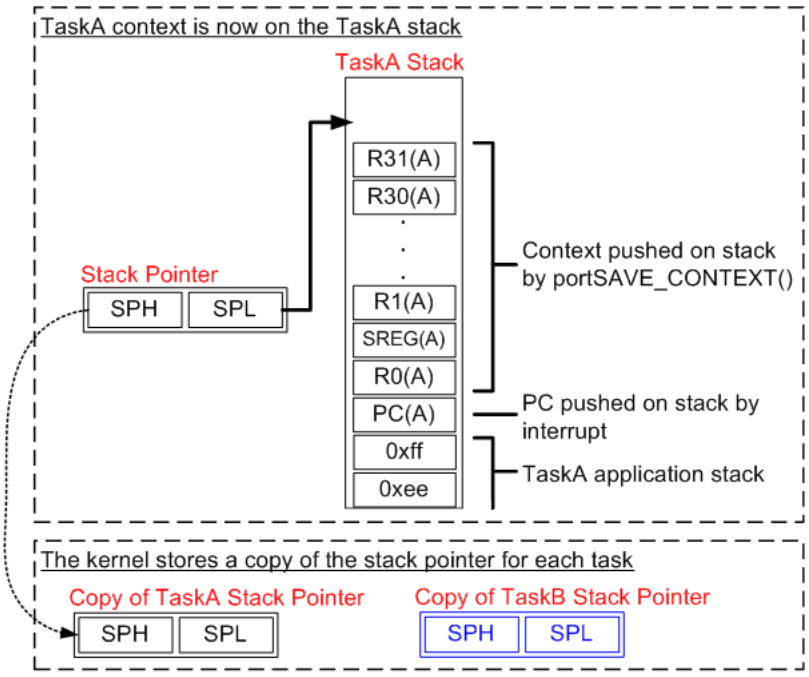
vTaskIncrementTick() executes after the TaskA context has been saved. For the purposes of this example assume that incrementing the tick count has caused TaskB to become ready to run. TaskB has a higher priority than TaskA so vTaskSwitchContext() selects TaskB as the task to be given processing time when the ISR completes.

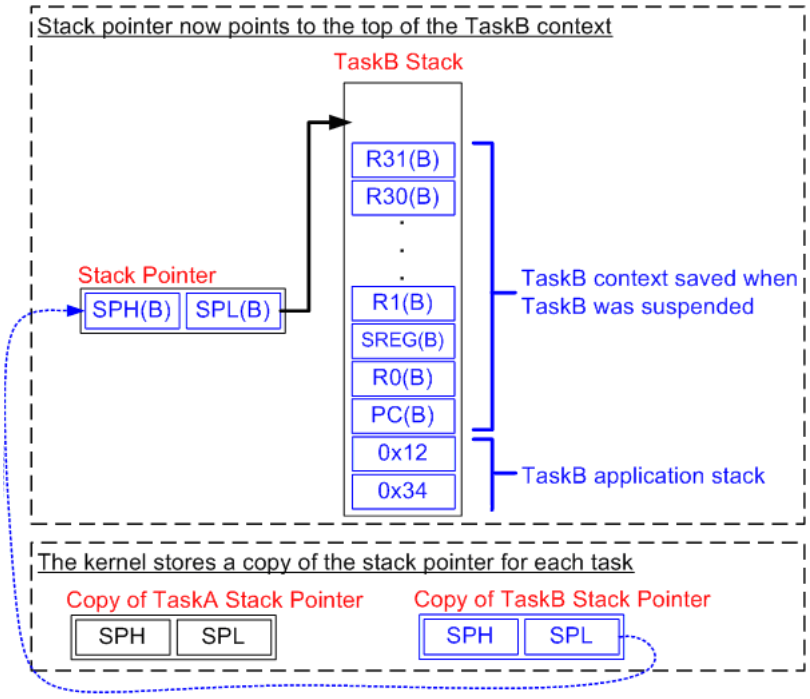
3.2.5 Step 5 The TaskB stack pointer is retrieved

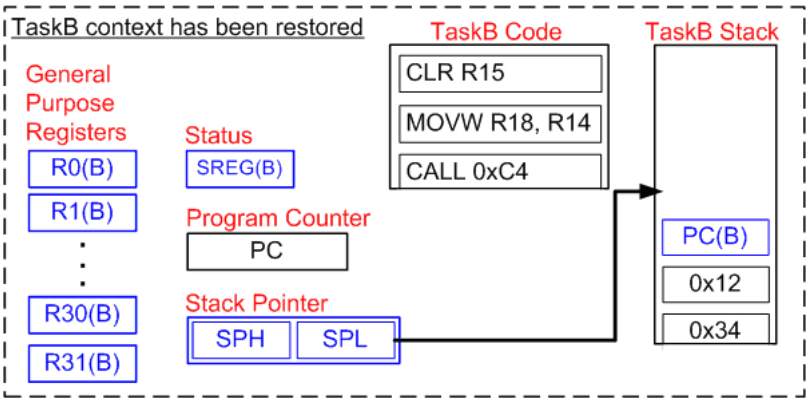
The **TaskB** context must be restored. The first thing **portRESTORE_CONTEXT** does is retrieve the **TaskB** stack pointer from the copy taken when **TaskB** was suspended. The TaskB stack pointer is loaded into the processor stack pointer, so now the AVR stack points to the top of the **TaskB** context.

3.2.6 Step 6 Restore the TaskB context

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




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

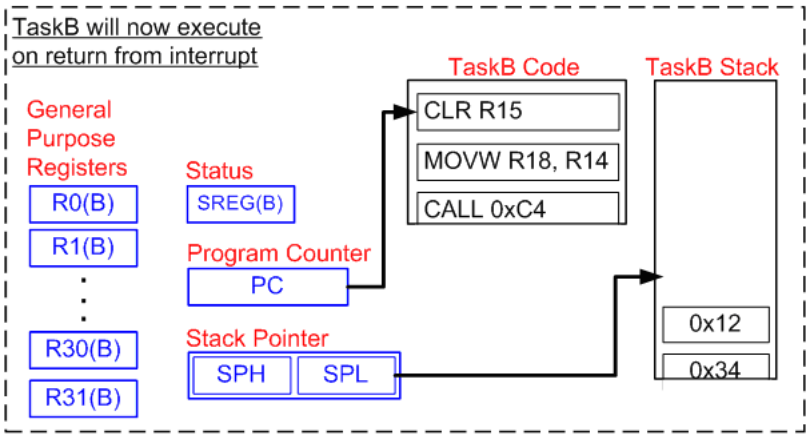

TaskB context has been restored
General Purpose Registers
R0(B)
R1(B)
R30(B)
R31(B)
Status
SREG(B)
Program Counter
PC
Stack Pointer
SPH
SPL
TaskB Code
CLR R15
MOVW R18, R14
CALL 0xC4
TaskB Stack
PC(B)
0x12
0x34

Only the program counter remains on the stack.

3.2.7 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.