MULTITASKING ON AN AVR

EXAMPLE C IMPLEMENTATION OF A MULTITASKING KERNEL FOR THE AVR

RICHARD BARRY

MARCH 2004

TABLE OF CONTENTS

Introduction

A real time operating system has to switch execution from one task to another to ensure each task is given processing time in accordance with the tasks priority. How this switch is performed is dependent on the microcontroller architecture. This article uses source code from FreeRTOS (an open source real time scheduler) and the free GCC development tools to demonstrate how a task switch can be implemented on an AVR.

The source code is explained from the bottom up. Topics covered include the setup of a periodic tick interrupt, using GCC to write interrupt service routines in C, special GCC features used by FreeRTOS and the AVR execution context.

The last pages demonstrates the source code operation with a detailed step by step guide to one complete task switch.

Following the source code can be a good way of learning both the compiler and hardware - even if task switching is not directly relevant to your application. I hope the topics will be of interest to those wishing to learn how to write interrupts using AVR builds of GCC, people new to AVR microcontrollers, or those who are just interested in RTOS implementation.

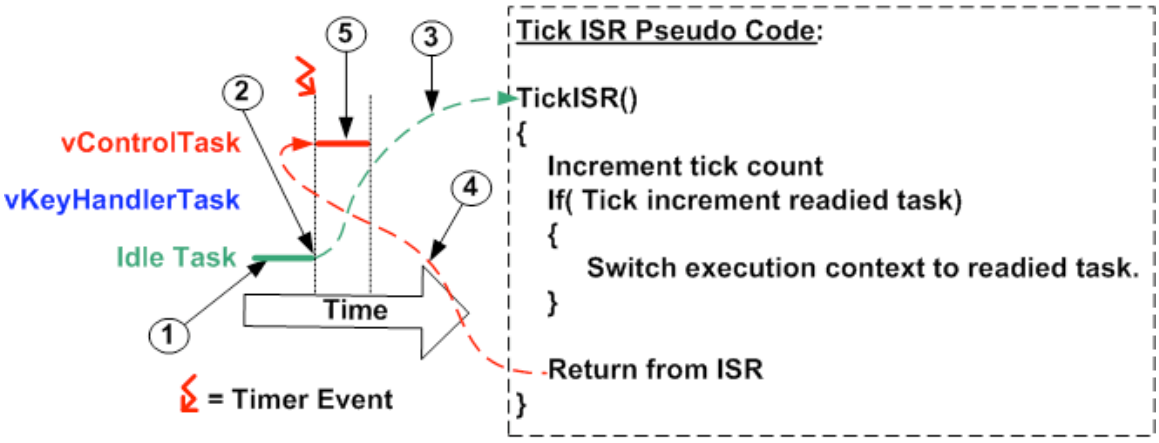
Readers should be familiar with the basic concepts of a real time operating system - such as tasks, multitasking and context switching. A brief introduction to these topics along with the complete FreeRTOS source code can be obtained from www.FreeRTOS.org.

The RTOS Tick

Applications that use a real time operating system (RTOS) are structured as a set of autonomous tasks, with the operating system deciding which task should execute at any given time. The RTOS kernel will suspend and resume tasks as necessary to ensure the task with the highest priority that is ready to run is the task given processing time. In addition to being suspended by the RTOS kernel a task can choose to suspend itself. It will do this if it either wants to sleep for a fixed period or wait (with a timeout) for a resource to become available. See the FreeRTOS WEB site for a more detailed explanation if you are not familiar with these concepts.

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

When a task suspends itself it specifies a delay (or "sleep") period. Each time the tick count is incremented the RTOS kernel must check to see if the new tick value has caused a delay period to expire. Any task found by the RTOS kernel to have an expired delay period is made ready to run. A context switch will be required within the RTOS tick if a task made ready to run by the tick interrupt service routine (ISR) has a priority higher than the task interrupted by the tick ISR. When this occurs the RTOS tick will interrupt one task, but return to another. This is depicted below:



In this type of diagram time moves from left to right. The coloured lines show which task is executing at any particular time. Referring to the numbers in the diagram above:

- At (1) the idle task is executing.
- At (2) the RTOS tick occurs, and control transfers to the tick ISR (3).
- The tick ISR makes vControlTask ready to run, and as vControlTask has a higher priority than the 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).

Generating the Tick Interrupt

A compare match interrupt on the AVR timer 1 peripheral is used to generate the RTOS tick.

Timer 1 is set to increment at a known frequency which is the system clock input frequency divided by a prescaler. The prescaler is required to ensure the timer count does not overflow too quickly. The compare match value is calculated to be the value to which timer 1 will have incremented from 0 in the required tick period. When the timer 1 value reaches the compare match value the compare match interrupt will execute and the AVR will automatically reset the timer 1 count back to 0 - so the following tick interrupt will occur after exactly the same interval.

```
/* Hardware constants for timer 1 on ATMega323. */
#define portCLEAR_COUNTER_ON_MATCH            ( 0x08 )
#define portPRESCALE_256                      ( 0x04 )
#define portCLOCK_PRESCALER                   ( 256 )
#define portCOMPARE_MATCH_A_INTERRUPT_ENABLE  ( 0x10 )

/*
 Setup timer 1 compare match A to generate a tick interrupt.
 */
static void prvSetupTimerInterrupt( void )
{
unsigned portLONG ulCompareMatch;
unsigned portCHAR ucHighByte, ucLowByte;

    /* Generate the compare match value for our required tick
    frequency. */
    ulCompareMatch = portCPU_CLOCK_HZ / portTICK_RATE_HZ;
```

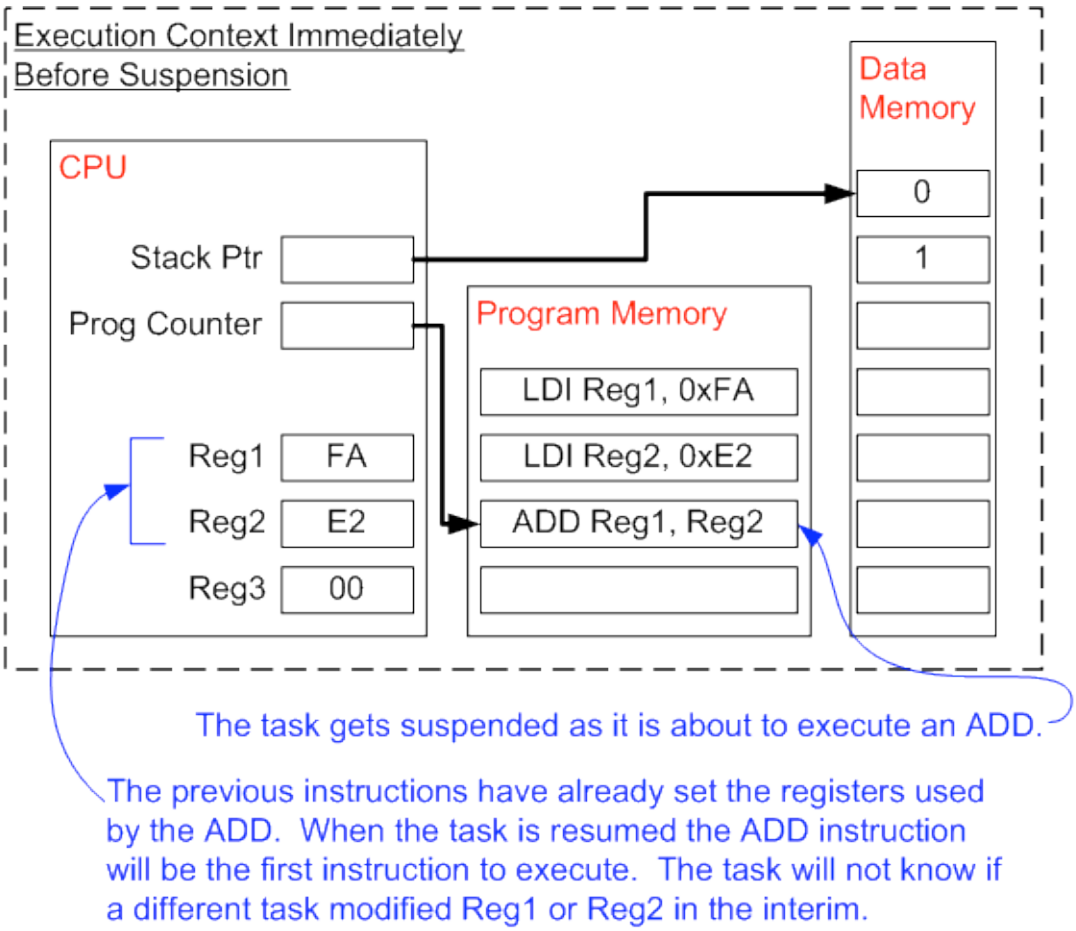
```
    /* We only have 16 bits so have to scale to get our
    required tick rate. */
    ulCompareMatch /= portCLOCK_PRESCALER;

    /* Setup compare match value for compare match A.
    Interrupts are disabled before calling this function so
    we need not bother here. [casting has been removed for
    each of reading] */
    ucLowByte = ulCompareMatch & 0xff;
    ulCompareMatch >>= 8;
    ucHighByte = ulCompareMatch & 0xff;
    outb( OCR1AH, ucHighByte );
    outb( OCR1AL, ucLowByte );

    /* Setup clock source and compare match behaviour. */
    ucLowByte = portCLEAR_COUNTER_ON_MATCH | portPRESCALE_256;
    outb( TCCR1B, ucLowByte );

    /* Enable the interrupt - this is okay as interrupt
    are currently globally disabled. */
    ucLowByte = inb( TIMSK );
    ucLowByte |= portCOMPARE_MATCH_A_INTERRUPT_ENABLE;
    outb( TIMSK, ucLowByte );
}
```

Source code to setup the tick interrupt

'Execution Context' - a Definition

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

A task is a sequential piece of code that does not know when it is going to get suspended (stopped from executing) or resumed (given more processing time) by the RTOS 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 registers.



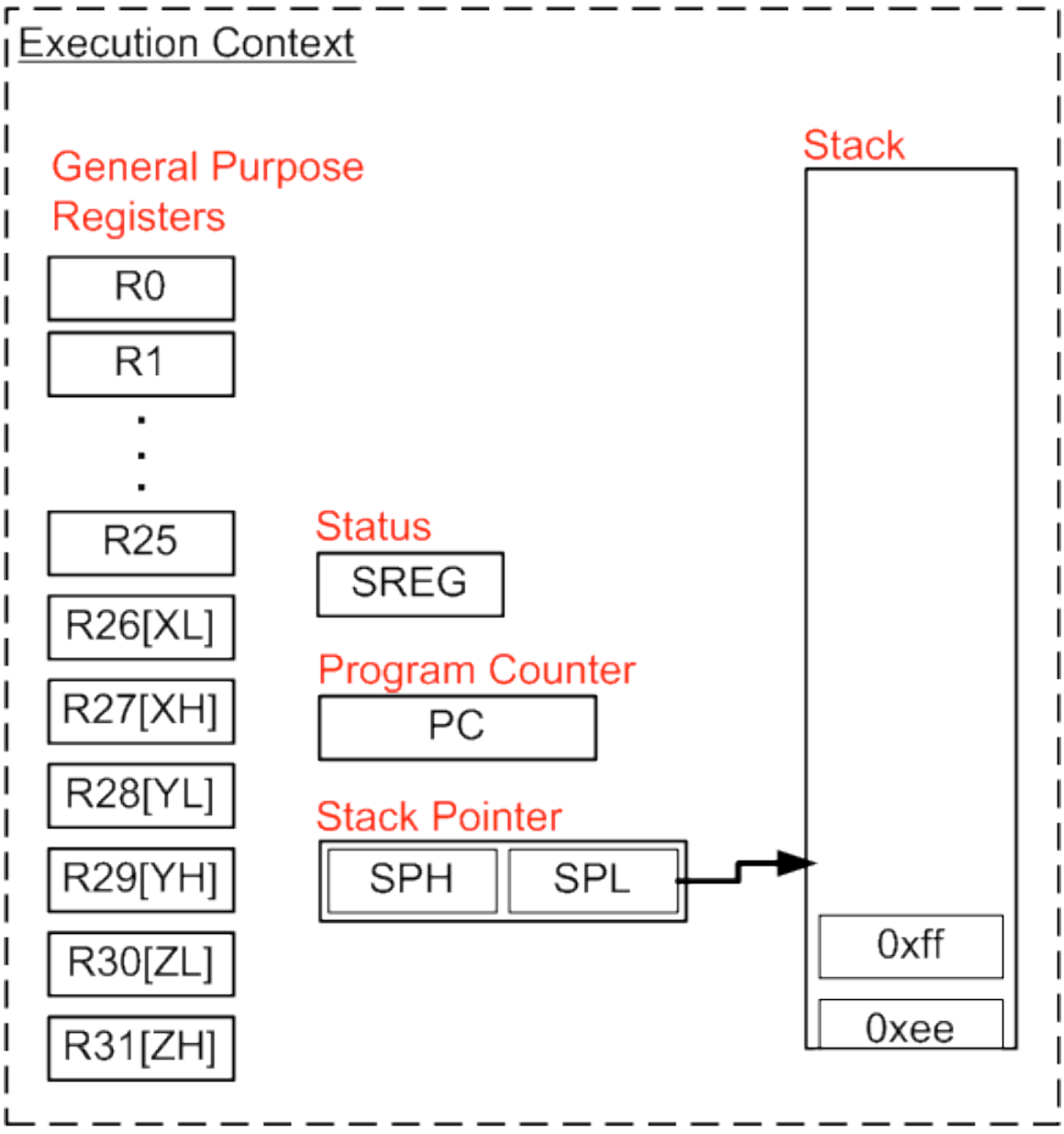
While the task is suspended other tasks will execute and may modify the register values. Upon resumption the task will not know that the registers have been altered - if it used the modified values the summation would result in an incorrect value.

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 RTOS 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 RTOS 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.

The AVR Context

On the AVR microcontroller the context consists of:

- 32 general purpose registers. The gcc compiler assumes register R1 is set to zero.
- Status register. The value of the status register affects instruction execution, and must 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.



Writing the ISR - The GCC 'signal' Attribute

FreeRTOS generates the tick interrupt from a compare match event on the AVR timer 1 peripheral. Using GCC the tick ISR function can be written in C by using the following syntax.

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

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

C code for compare match ISR

The *'__attribute__ ((signal))'* directive informs the compiler that the function is an ISR and results in two important changes to the code output by the compiler:

1. The 'signal' attribute ensures that every AVR 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 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 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.

    ;vTaskIncrementTick();
    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
       ; --------------------------------------
```

Organizing the Context - The GCC 'naked' Attribute

The previous page shows 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 microcontroller registers modified by the ISR get saved). Performing a context switch however requires the *entire* context to be saved. If the application code saved the entire context some AVR registers would get saved twice - once by the compiler generated code and then again by the application code. This can be avoided by also using the 'naked' attribute.

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

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

Naked C code for compare match ISR

The 'naked' attribute prevents the compiler generating any function entry or exit code.

Code output by the compiler when both the signal and naked attributes are used:

```
;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 be written explicitly as follows:

```
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" );
}
```

Naked ISR with explicit entry and exit code

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 microcontroller registers get saved twice.

Saving and Restoring the Context

Registers are saved by simply pushing them onto the stack. Each task has it's own stack, into which it's context is saved when the task gets suspended.

Saving the AVR context is one place where assembly code is unavoidable. portSAVE_CONTEXT() is implemented as a macro, the source 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 code above:

- Register R0 is saved first (1) 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).
- 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 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 microcontroller 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 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 kernel retrieves the stack pointer for the task then POP's the context back into the correct microcontroller 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 microcontroller 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).

The Complete FreeRTOS ISR

The actual source code used by the FreeRTOS AVR port is slightly different to the examples shown on the previous pages. The context is saved from within vPortYieldFromTick() which is itself implemented as a 'naked' function. It is done this way due to the implementation of non-preemptive context switches (not described here).

The FreeRTOS implementation of the RTOS tick is therefore ***(see the comments in the 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" );
}
/*-----------------------------------------------------*/
```

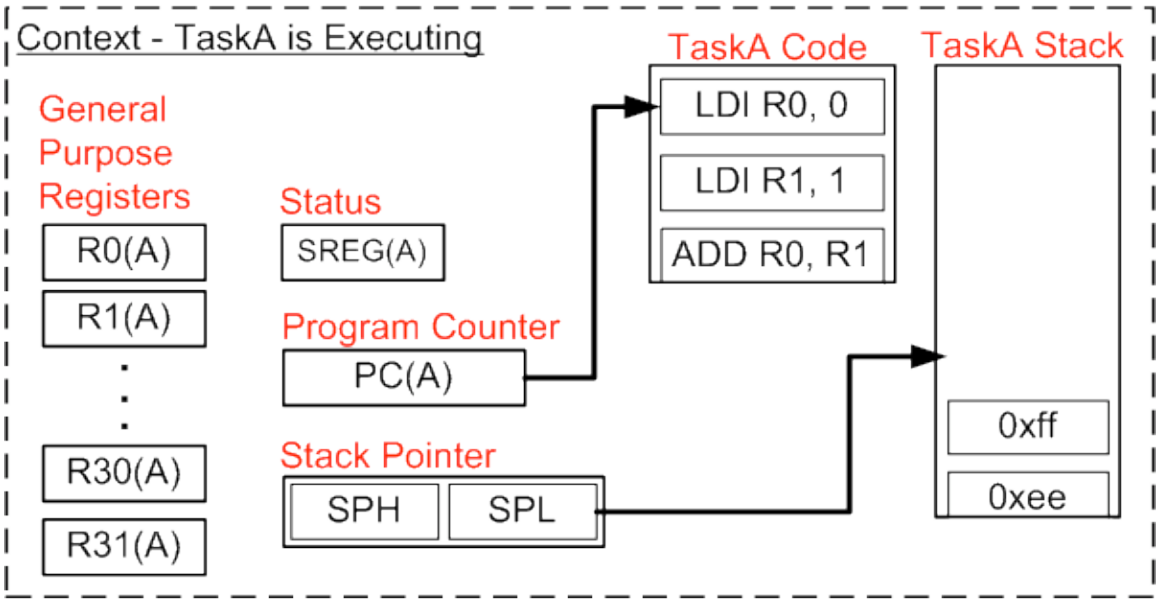
Putting it All Together - A Step By Step Example

This page presents a detailed demonstration of the source code operation in achieving 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.

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.

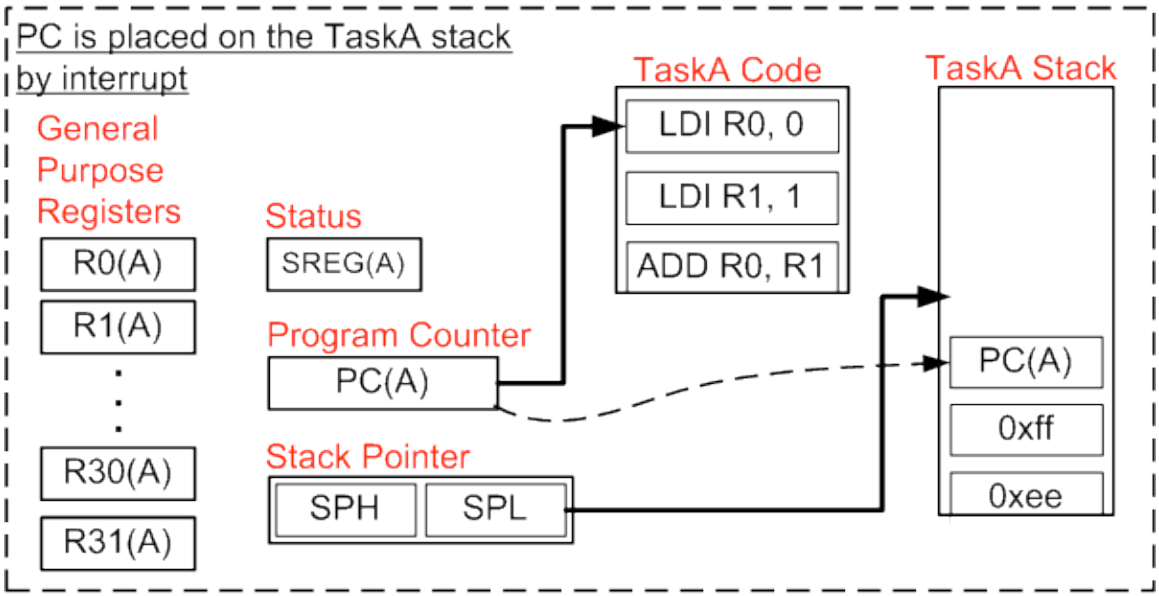
TaskA has the context demonstrated by the diagram below.



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

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 automatically places the current program counter (PC) onto the stack before jumping to the start of the RTOS tick ISR.



Step 3: The RTOS tick interrupt executes

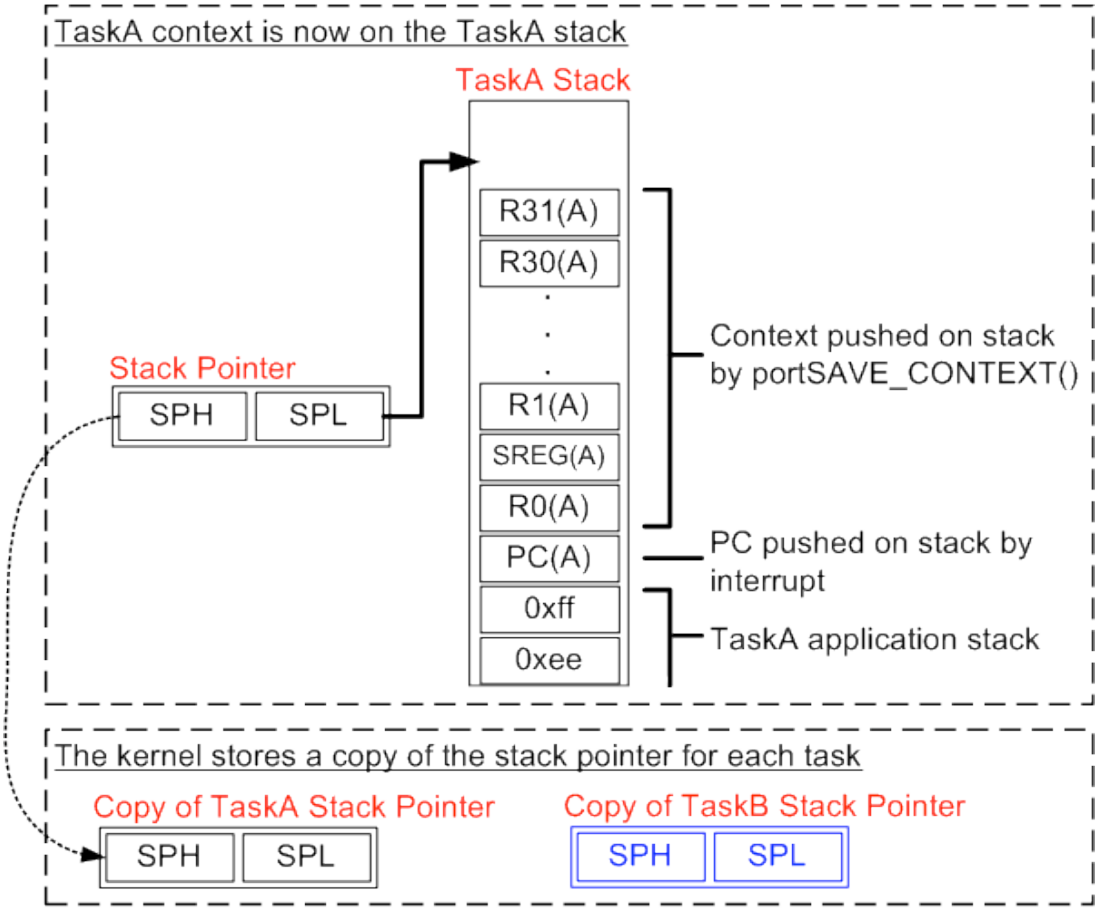
The ISR application code is given below. The comments have been removed to ease reading, but can be viewed on a previous page.

```
/* 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 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 kernel already has copy of the TaskB stack pointer - taken the last time TaskB was suspended.

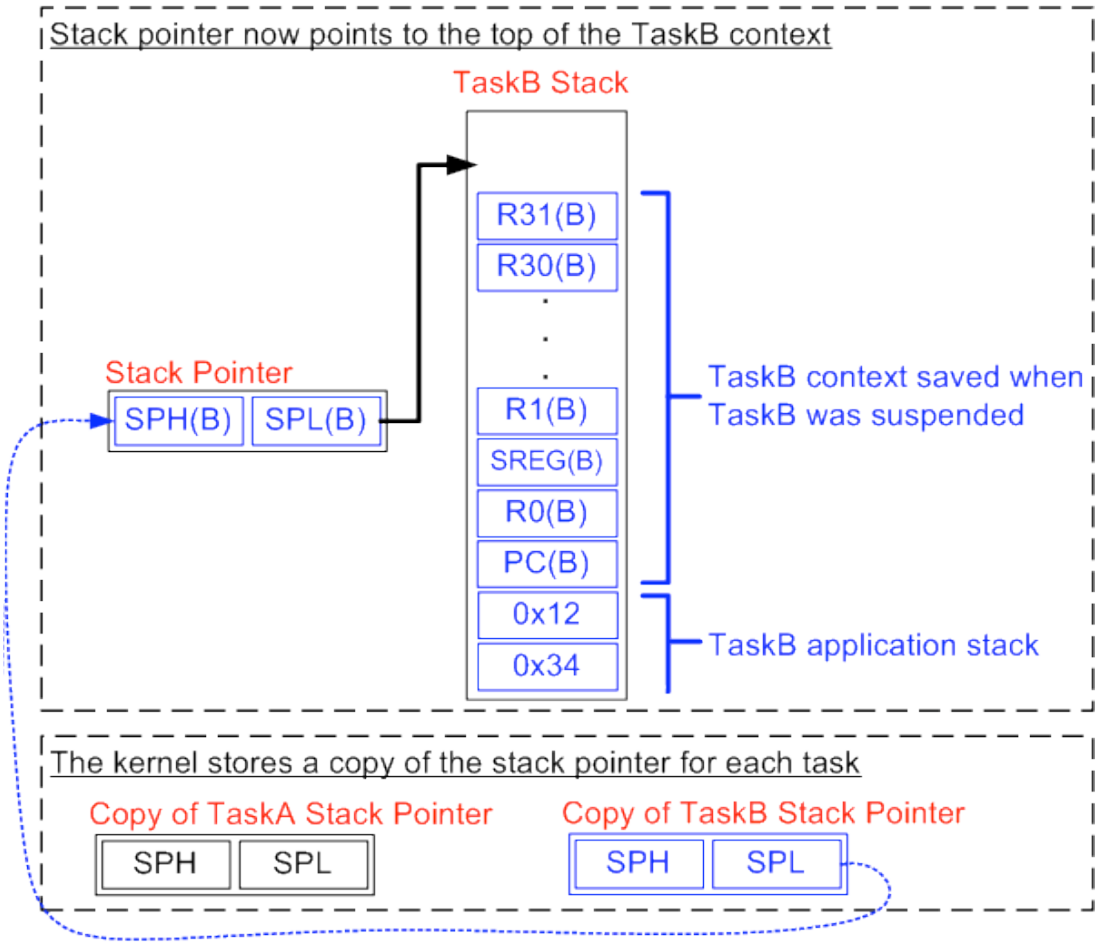


Step 4: Incrementing the Tick Count

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.

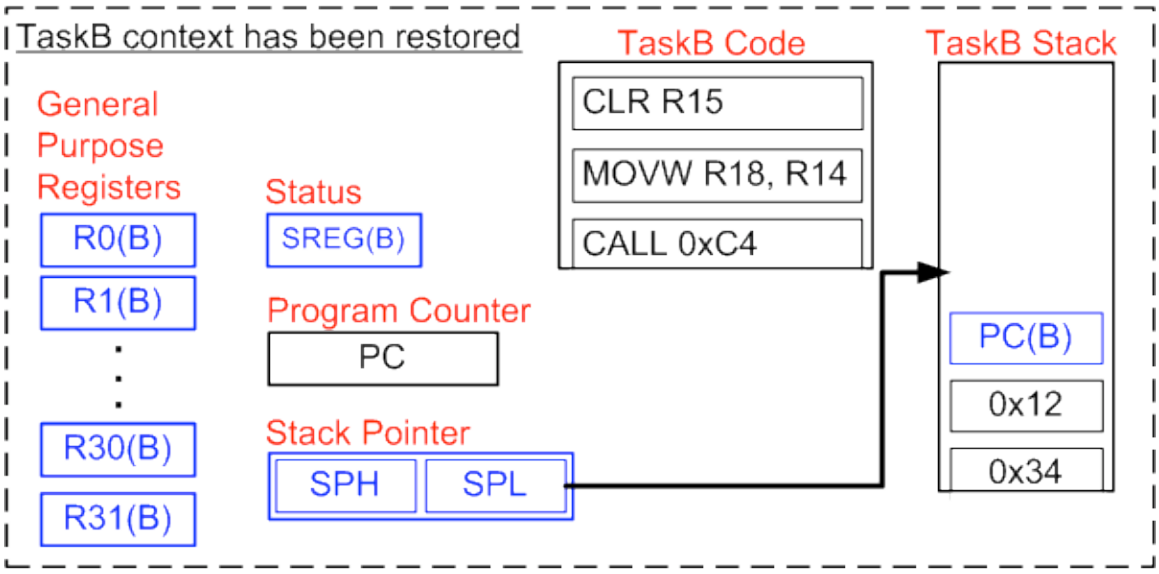
Step 5: The TaskB stack pointer is retrieved

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 AVR stack pointer, so now the AVR stack points to the top of the TaskB context.



Step 6: Restore the TaskB context

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

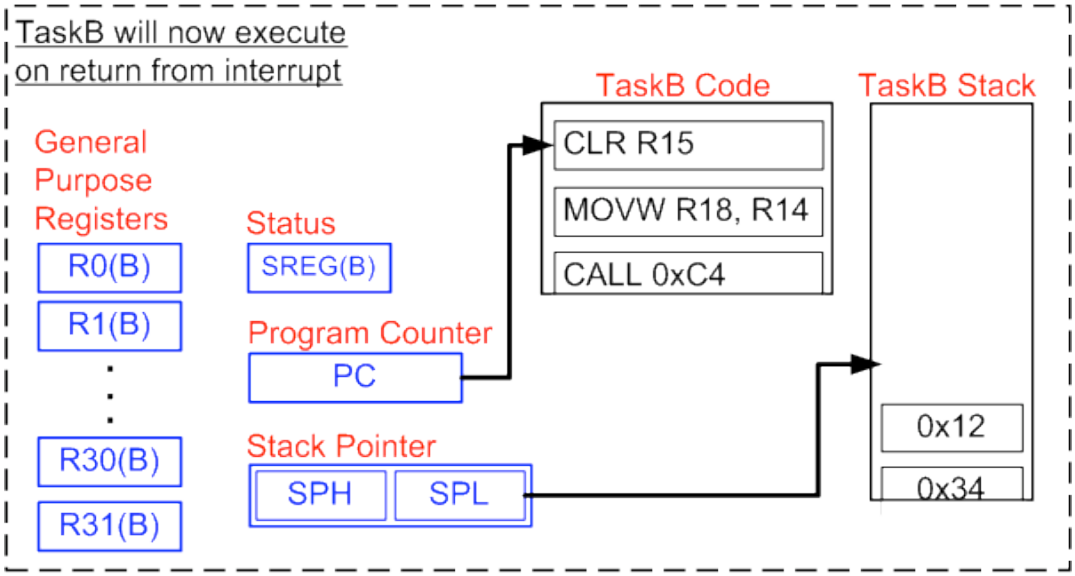


Only the program counter remains on the stack.

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 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!