从基本原理到实现

本文描述 FreeRTOS(飞拓)是如何实现的。

如果你

1. 希望修改 FreeRTOS 源代码
2. 移植实时内核到另一个微控制器或者原型板(prototyping board)
3. 第一次接触 FreeRTOS，希望得到关于它们在操作和实现上的更多信息

这些文档会有用。

本文档分为两个章节:

1. 基本原理和 RTOS 概念

包括多任务的背景信息和基本实时概念，这是为初学者准备的(is intended for beginners)

2. **从底向上(from the bottom up)解释实时内核源代码**

FreeRTOS 实时内核已经移植到许多不同的微控制器架构下。这份文档是以 Atmel AVR 为范例，因为:

1. AVR 架构简单
2. 有免费可用的开发工具 **WinAVR (GCC) development tools**.
3. 非常便宜的原型板 **STK500 prototyping board**

在本文的最后，还一步一步地详细描述了一个**完整的上下文切换**(context switch)。

RTOS 基本原理

多任务

调度

上下文切换

实时应用

实时调度

这一节提供一个关于实时和多任务概念的简介。读下一节之前必须理解这些概念。

多任务(Multitasking)

在一个操作系统内部，内核[**kernel]**是最核心的部件。像 Linux 那样的操作系统使用的内核，从表面上看(seemingly)，允许用户并发（simultaneously）访问 计算机。多个用户似乎(apparently)可以并行(concurrently)执行多个程序。

在操作系统的控制下，每个正在执行的程序就是一个任务[task]。如果一个操作系统能够以这种方法执行多个任务，这就叫做多任务[multitasking].

多任务操作系统的使用可以简化应用程序的设计:

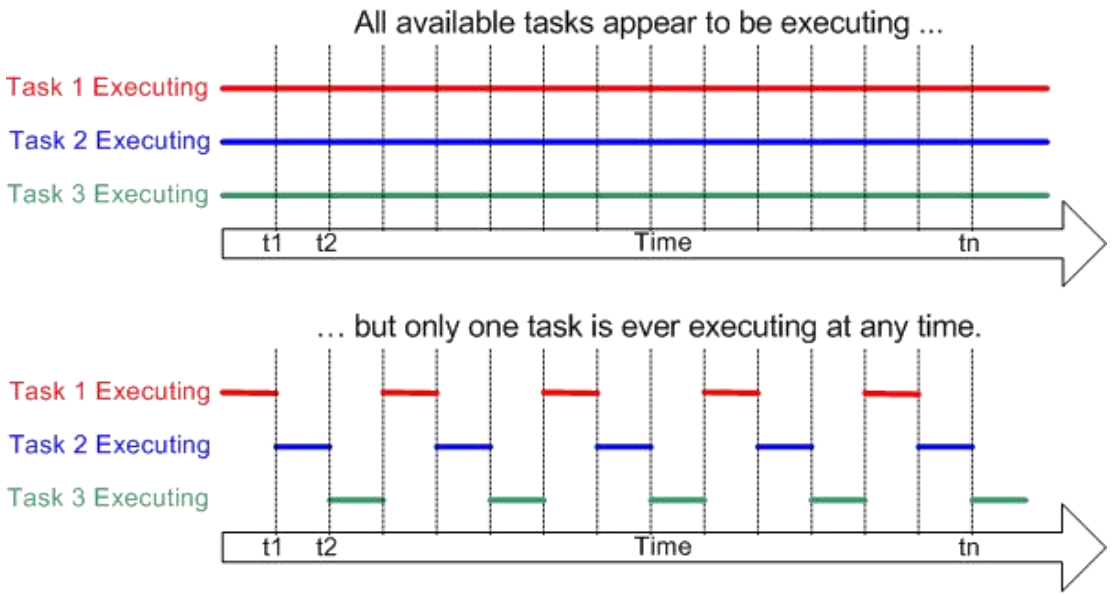
1．操作系统的多任务和任务间通信的机制允许复杂的应用程序被分成一系列更小的和更多的可以管理的任务。

2．(程序的)划分(partitioning)让软件测试更容易，团队工作分解(work breakdown within teams),也有利于代码复用。

3． 复杂的定时和先后顺序的细节 可以从应用程序代码中 删除。（因为）这成为操作系统的职责。

多任务 Vs 并发

传统的(conventional)的处理器同时只能执行一个任务。但通过快速的任务切换，一个多任务操作系统可以使它看起来(appear)好像每个任务并行执行一样。这可以下面的示意图来描述（depicted）。它显示了有关(with respect to)时间的 3 个任务的执行模式。任务名用颜色标注出来，写在左手边。时间从左到右增加，相应的颜色的线条 显示该任务在某个特殊时间正在执行。上面的图 演示的是用户所觉察到的并行执行模式，下面的图是实际的多任务执行模式。



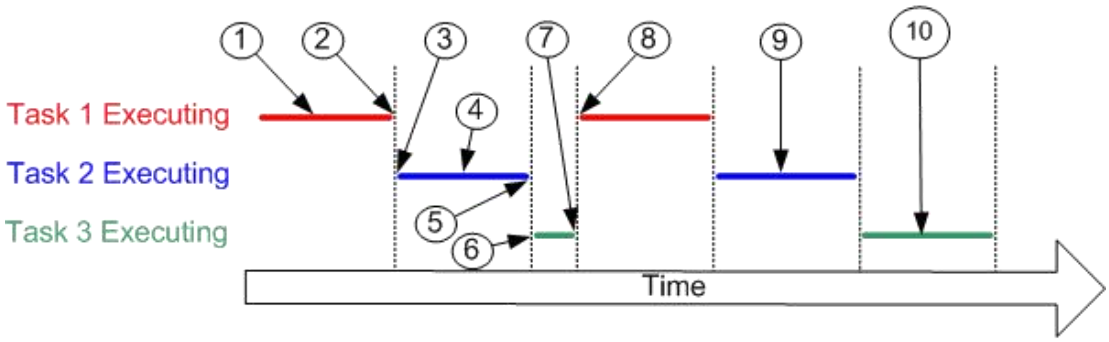
----所有可用的任务都好像在执行，但实际上在任何一个时刻都只有一个任务在执行

调度

调度器（scheduler）是内核中负责 决定在某个特殊时间 哪个任务应该执行的部分。内核可以在任务的生命期(lifetime) 挂起(suspend) / 恢复（resume）一个任务许多次。

调度策略（scheduling policy）是调度器用来决定哪个任务在哪个时间点执行的算法。一个（非实时）多用户系统的策略很可能分配(allow)给每个任务一个"公平"(fair)的处理器时间片(proportion of processor time)。用在实时系统/嵌入式系统的策略稍后再描述。

除了被 RTOS 内核无意的挂起外，一个任务还可以自己挂起自己。如果一个任务想延迟一段固定的时间(也就是 sleep),或者等待(也就是 block)某个资源可用（比如一个串口），或者等待一个事件出现(比如一个键按下)。一个阻塞或者睡眠的任务是不能执行的，不会为它分配任何处理时间。



上图中提到的编号:

1) Task1 正在运行

2) 内核挂起 Task1

3) 恢复任务 Task2

4) Task2 正在执行，为独占访问（exclusive access），它锁定一个处理器外设

5) 内核 挂起 Task2

6) 恢复 Task3

7) Task3 试图访问同样的处理器外设，发现它被锁定，Task3 不能继续，所以自己挂起自己。

8) 内核恢复 Task1

……

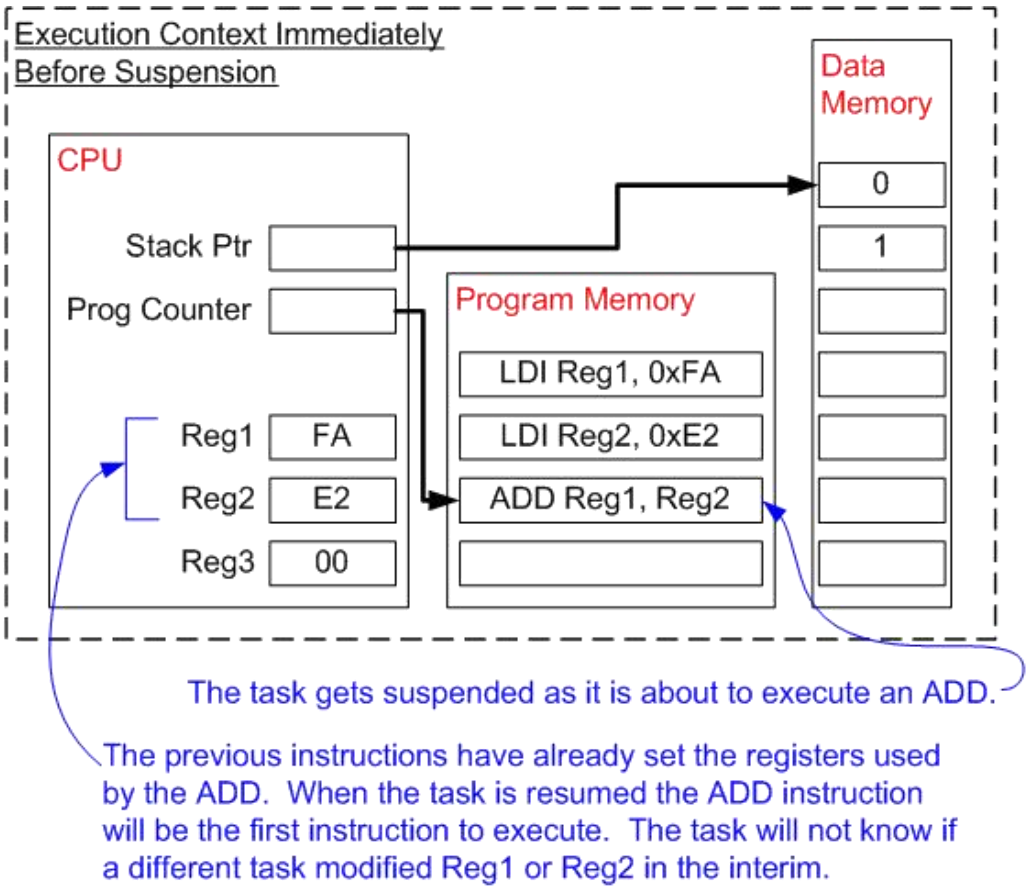
9) 接下来(the next time)，Task2 在 9 处执行。它完成了对处理器外设的访问，所以解锁它

10) 再下来，Task3 在 10 处执行。它发现 现在可以访问处理器外设了，于是开始执行，直到被内核挂起。

上下文切换

跟任何其他程序一样，一个任务执行时，它使用 处理器/微控制器 的寄存器，访问 RAM ROM。这些资源(处理器的寄存器，stack 等)一起组成任务的执行上下文(the task execution context).

一个任务是一个连续有序的代码片断。它并不知道它将何时被内核挂起或者恢复，甚至不知道这些事情（挂起或者恢复）在什么时候已经发生了。下面考查(Consider)的这个例子是用来求两个处理器中的寄存器值之和，该任务在执行 1 条指令后就立即被挂起。



-->任务将要执行 ADD 指令时，被挂起

-->先前的指令已经把数取到寄存器(Reg1,Reg2)中了,而这些寄存器(Reg1,Reg2)将要被 ADD 指令用到。当这个任务被恢复后，ADD 就是要执行的第 1 条指令。这个任务不知道是否有另一个的任务会在中间时期 修改 Reg1 或者 Reg2)

当这个任务挂起时，其他任务继续执行，可能会修改处理器寄存器的值。在恢复之后，这个任务也不知道处理器的寄存器被修改过(altered).如果它使用这个修改过的值，就会导致计算的和的结果不正确。

为了避免这类错误，必须保证，在恢复一个任务之后，其上下文环境跟 即将挂起前是一样的。操作系统内核有责任 通过在任务挂起前保存其上下文 来确保这种状况。当任务恢复时，保存的上下文就被 操作系统内核恢复到先前的执行情况。保存一个被挂起的任务的上下文 并在 任务恢复时 恢复其上下文的这个处理过程就叫做上下文切换（context switching）

实时应用

实时操作系统（**RTOS**'s）通过同样的原理达到多任务的目的。但他们的目标与那些非实时系统相比是很不一样的。不同的目标影响到不同的调度策略。实时/嵌入式系统设计成提供一个对 真实世界的事件的及时响应(timely response)。出现在真实世界中的事件可能有一个时间限制(deadline)，在此期限之前，实时/嵌入式系统必须给出响应，RTOS 调度策略必须确保时间限制是恰当的(met).

为了达到这个目的，软件工程师必须首先为每个任务设置一个优先级(priority)。RTOS 的调度策略只是简单地确认 能被执行的最高优先级别的任务 是分配了处理时间的任务（the task given processing time）。这可能要求在相同优先级的 任务之间公平的共享 处理时间，如果他们准备并发运行的话。

代码示例:

最基本的例子是 一个由键盘和 LCD 组成的实时系统。用户必须 在合理的时段 为 每个键盘按下取得视觉反馈(visual feedback).。如果用户不能 在这时段 看到 键盘按下 已经被接受，软件产品将会很难使用（be awkward to use）。如果最长的接受期是 100ms,那么在 0 到 100ms 的响应 可被接受。这个功能可以用一个 下面这样的结构的独立(autonomous)任务 实现:

```
void vKeyHandlerTask( void *pvParameters )
{
    //键盘处理是一个连续的过程。就像大多实时任务那样，这个任务
    //也是用一个无限循环实现的。
    for( ;; )
    {
```

```
        [Suspend waiting for a key press]
        [Process the key press]
    }
}
```

现在假设 实时系统也执行一个依赖数字滤波输入的控制功能。这个输入必须被取样（sampled），滤波(filtered)，并且 每 2ms 执行一次控制循环。为了让滤波器正常操作，取样的时间规律（the temporal regularity）必须精确到 0.5ms。 这个功能可以 用下面这个结构的 独立任务 实现:

```
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]
    }
}
```

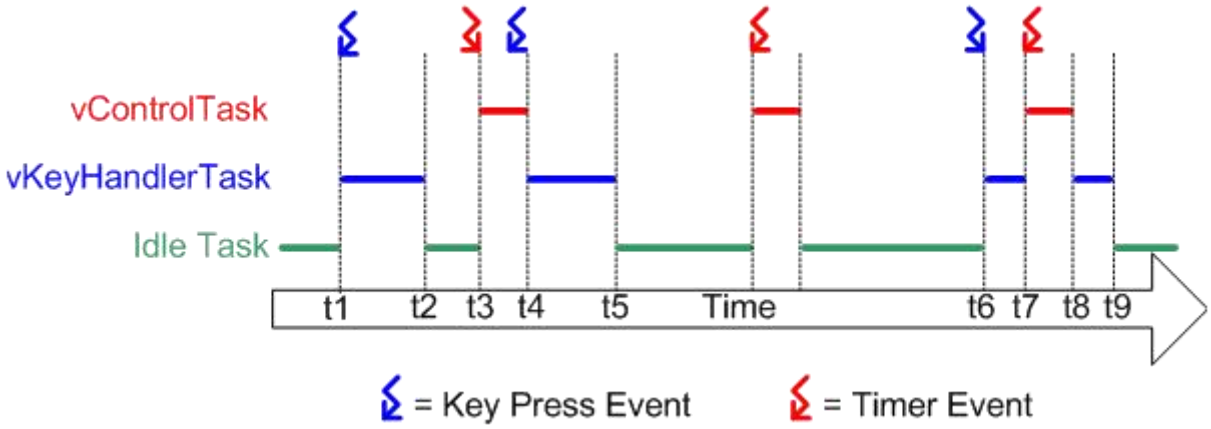
软件工程师必须设置 控制任务为 最高的优先级，因为：

1. 控制任务的时间限制(deadline) 比 键盘处理任务的 要严格；
2. 控制任务错过最后期限(deadline)的后果 比 键盘处理任务 要严重。

下面将演示 这些任务是如何被实时操作系统调度的.

实时调度

下面的图 演示 前面定义的那些任务是如何被时实操作系统调度的。RTOS 自己已经建立了一个任务----idle task---它只在没有其他任务执行的时候才被执行。RTOS idle task 总是处于可以执行的状态（注:也就是它不可能会因为等待什么外设资源而被阻塞,而是处于一种随时待命的状态）.



上图中:

1. 在最开始，我们的两个任务都不能被执行-vControlTask 等待合适（correct）的时间开始新的控制循环，vKeyHandlerTask 等待键盘按下。处理器时间分配给 RTOS 的 idle task.

2. 在 t1 时刻，一个键盘按下(事件)出现. VKeyHandlerTask 任务现在可以执行，它比 RTOS 的 idle task 有更高的优先级，所以处理器时间给它。

3. 在 t2 时刻，vKeyHandlerTask 已经完成了对按键的处理，并更新了 LCD。它不能继续，直到另一个键被按下，所以必须挂起它自己。RTOS idle task 又被恢复执行。

4. 在 t3 时刻，一个定时器事件预示（indicates），可以执行下一个控制循环了。VControlTask 现在可以执行，作为最高优先级的任务被立刻分配(scheduled)到处理器时间。

5. 在 t3 和 t4 之间，当 vControlTask 任务还在执行的时候，一个键按下。VKeyHandlerTask 不

能被执行，因为它没有 vControlTask 的优先级高。不能分配（scheduled）到任何处理器时间。

6. 在 t4 时刻，vControlTask 完成了控制循环的处理，不能够重新开始，直到下一个时间事件出现，所以它自己挂起自己。而 vKeyHandlerTask 现在是最高优先级的任务，可以运行了，所以，为了处理先前的键盘事件，分配(scheduled)到了处理器时间.

7. 在 t5 时刻，键盘已经被处理。VkeyHandlerTask 为了等待下一个键盘事件，自己挂起自己。现在，我们的两个任务再度不能执行了。RTOS idle task 分配到处理器时间。

8. 在 t5 和 t6 之间，一个定时器事件被处理，但是没有更多的键盘事件出现。

9. 下一个键盘按下出现在 t6 时刻，但在 vKeyHandlerTask 完成处理键之前，一个定时器事件出现了。现在两个任务都能被执行，而 vControlTask 比 vKeyHandlerTask 有更多的优先级，所以 vKeyHandlerTask 在它完成处理键盘之前就被挂起了。VControlTask 分配到处理器时间。

10. 在 t8 时刻，vControlTask 完成处理控制循环，挂起自己以等待下一个事件。

VKeyHandlerTask 再次成为最高优先级的任务，能够运行，所以分配到处理器时间，从而键盘按下事件 处理能够完成。

RTOS 实现

模块(Building Block)

详细实例(Detailed Example)

这一节从底向上描述了 RTOS 上下文切换的源代码。使用 FreeRTOS Atmel AVR 微控制器移植的代码作为例子。本节的最后还一步一步地浏览(step by step look)了一个完整的上下文切换。

C 开发工具

FreeRTOS 的目标是简单且易于理解。为了达到这个目标(To this end),RTOS 的源代码的大部分都是用 C 写的，而不是汇编。

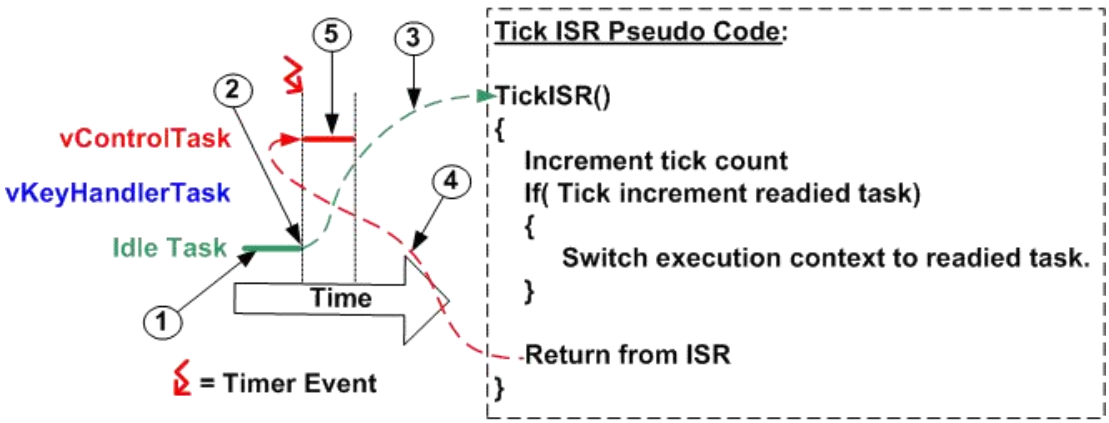
这里演示的例子使用了 **WinAVR development tools**。 WinAVR 是一个自由/免费的在 windows 下的 AVR 交叉编译器，它是基于 GCC 的。

RTOS Tick

睡眠时，一个任务将指定多长时间后它会醒来。阻塞时，一个任务将指定一个 希望最多等多久的时间。

FreeRTOS 实时内核用 tick count 变量 来度量时间的。定时器中断(RTOS tick interrupt) 用严格的时间精度（temporal accuracy ）来增加 tick count------- 允许实时内核 用一个指定的定时器中断频率的精度（resolution）来测量 时间。

每次 tick count 增加后，实时内核 必须检查，看现在是否 解除阻塞 或者 唤醒 一个任务。一个比被中断的任务有更高的优先级的 任务 在 tick ISR 期间 被 唤醒或者解除阻塞 是可能的。如果是这种情况，tick ISR 应该返回到新的唤醒/解锁的任务---实际(effectively)中断一个任务，却返回到另一个任务。如下所述：



上图提到的几个点：

1） RTOS idle task 正在运行

2） RTOS tick 出现，控制转移到 tick ISR(3)

3） RTOS tick ISR 使得 vControlTask 准备运行，当 vControlTask 比 RTOS idle task 有更高的优先级，切换上下文到 vControlTask.。

4） 现在的执行上下文是 vControlTask 的。从 ISR(4)退出 返回到 vControlTask，vControlTask 从（5）开始执行。

以这种方式出现的上下文切换，称为 Preemptive。 因为被中断的任务 没有自愿（voluntarily）地挂起它自己就被抢占了（preempted）。

FreeRTOS 的 AVR 移植版本 用一个在定时器 1(timer1)的比较匹配（compare match）事件来产生 RTOS tick. 后续将描述 RTOS tick ISR 是如何用 WinAVR 开发工具实现的。

GCC 信号属性（Signal Attribute）

GCC development tools 允许用 C 来写中断程序。一个在 AVR 定时器 1 外设的比较匹配事件 可以用下面的 语法(syntax)实现:

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

在函数原型前的 '__attribute__ ((signal))' 指示符 告知 编译器，这个函数是一个 ISR,会引起编译器输出的两个重要改变：

1. signal 属性保证，每个在 ISR 期间 被修改的 处理器的寄存器，在从 ISR 中退出时恢复到它原来的值。这就要求，当中断将要执行时，编译器不能做任何假定。所以，不能优化哪个处理器寄存器要求保护或者不保护。

2. 'signal'也强制使用 一个 从中断返回（'return from interrupt'）指令（RETI），而不是返回(return)指令 RET. AVR 微控制器 在进入 ISR 前禁止中断，RETI 指令要求在 退出时重新打开中断。

下面是由编译器输出的代码：

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

GCC Naked 属性

前一节讲述了如何在 C 中用 signal 属性来写一个 ISR.,以及它是如何使 执行上下文自动保存的（只有那些被 ISR 修改过的处理器寄存器才会得到保存）。然而，执行一个上下文切换需要保存完整的上下文。

应用程序代码能够 在进入 ISR 时，明确(explicitly)地 保存所有寄存器，但是这样会使 某些处理器寄存器 保存两次---一次是由编译器生成的代码，另一次是由应用程序自己。这不是我们所需要的，可以在'signal'属性后 添加 'naked'属性来避免：

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

'naked'属性阻止编译器生成任何函数入口或退出代码。现在变异这段代码，会得到更少的编译器输出：

```
;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.
    ; ------------------------------------
;}
```

看看，入口 和 出口代码都没有了吧

使用 naked 属性，编译器不会生成任何入口和出口代码，所以必须明确（explicitly）加入。portSAVE_CONTEXT()和 portRESTORE_CONTEXT()这两个宏 是用来保存和恢复完整的执行上下文的：

```
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 属性给了应用程序完整的控制权，在何时 ，怎么样保存 AVR 的上下文。如果应用程序代码在进入 ISR 前保存了完整的上下文，在执行上下文切换时不必再保存，所以不会有处理器的寄存器被保存两次。

FreeRTOS Tick Code

FreeRTOS 的 AVR 移植版本的实际源代码 与 前一节的例子有些轻微的不同。VPortYieldFromTick（）是作为一个 naked 函数的 它自己的实现，上下文在 vPortYieldFromTick().里被保存和恢复。这样做的目的是为了实现一个 non-preemptive 的上下文切换（这里，一个任务自己阻塞自己）. 这里暂时不讲这种 non-preemptive 切换。

RTOS tick 是这样在 FreeRTOS 中实现的(看代码中注释片断获取更多细节)：

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


/*-----------------------------------------------------*/
/* RTOS tick 中断服务程序. */
void SIG_OUTPUT_COMPARE1A( void )
{
   /*调用 tick 函数. */
vPortYieldFromTick();

/*从中断返回. 如果出现上下文切换，将返回到一个不同的任务中 */
   asm volatile ( "reti" );
}
```

```
/*-----------------------------------------------*/
void vPortYieldFromTick( void )
{
   /* 这是一个 naked 函数，所以需要保存上下文 */
portSAVE_CONTEXT();

/* 增加 tick count，检查新的 tick count 值是否引起一个延迟周期过期，这个函数调用可导致一个任务变成准备运行. */
vTaskIncrementTick();

/*检查是否要求上限文切换。如果 由 vTaskIncrementTick()准备好的任务比已经中断的任务有更高优先级，就切换过去 */
vTaskSwitchContext();

/*恢复上下文.如果发生了上下文切换，这将恢复要继续运行的任务的上下文 */
portRESTORE_CONTEXT();

/*从这 naked 函数返回. */
   asm volatile ( "ret" );
}
/*-----------------------------------------------*/
```

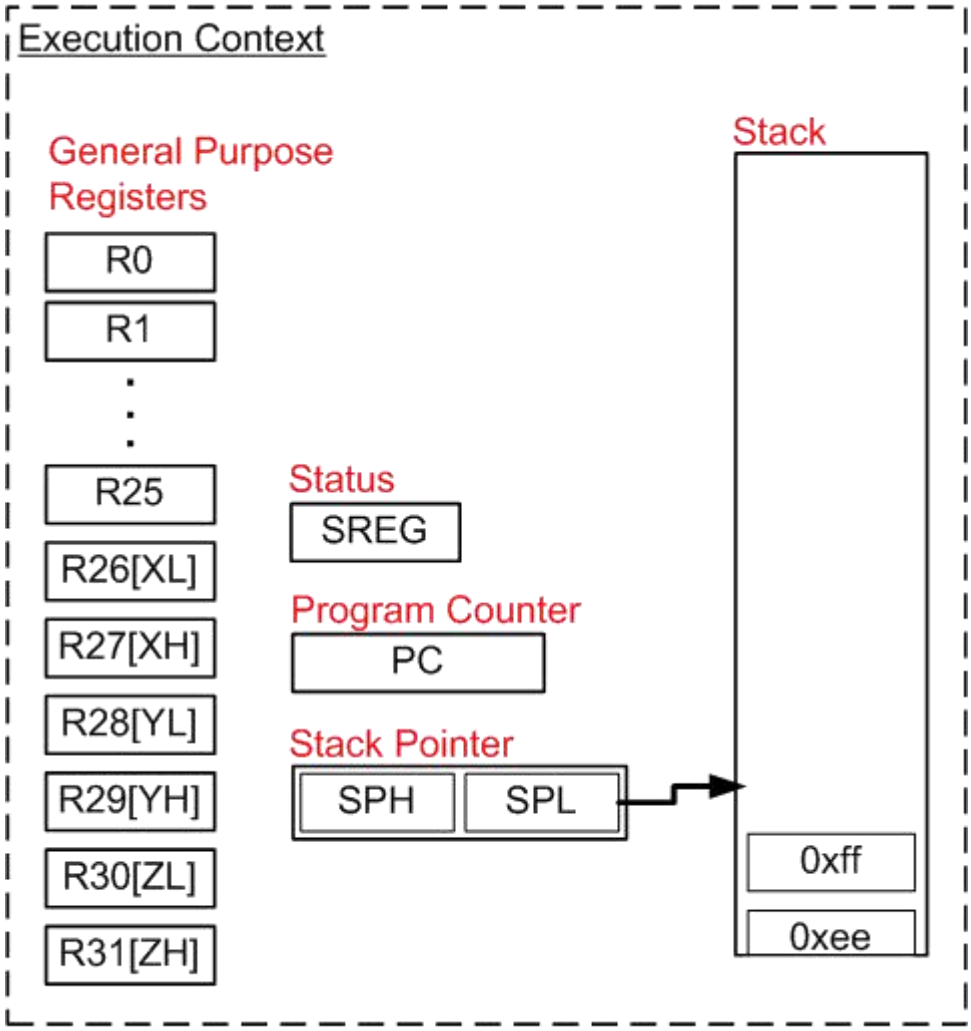
The AVR Context

上下文切换要求保存完整的上下文。在 AVR MCU 中，上下文包括

1. 32 位通用寄存器。Gcc 开发工具假定寄存器 R1 设定为 0
2. 状态寄存器。状态寄存器的值影响指令的执行，必须通过上下文切换保存(preserved)
3. 程序计数器(PC).恢复执行后，一个任务必须从上次被挂起的地方继续执行。（a task must continue execution from the instruction that was about to be executed immediately

prior to its suspension.）

4. 两个 stack 指针寄存器



Saving the Context

每个时实任务都有它自己的 stack 内存区域，所以上下文可 简单的通过将寄存器压入到任务栈 来保存上下文。保存 AVR 的上下文是一个不可避免的要使用到汇编语言的地方。

portSAVE_CONTEXT() 是作为一个宏来实现的，源代码在下面给出

```
#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)
);
```

在上面的源代码中：

1. 寄存器 R0 首先被保存，因为当状态寄存器被保存时，它会被用到。所以必须先把它的原始值保

存下来。

2. 状态寄存器被搬移到 R0 (2),所以它能保存到 stack (4)

3. 禁止处理器中断。如果 portSAVE_CONTEXT(),只是从 ISR 中调用，就不需要明确的禁止中断，因为 AVR 已经这么做了。portSAVE_CONTEXT()宏用在中断服务程序的外部（一个任务自己挂起自己的时候）,中断应该尽可能早的明确清除(也就是禁止中断 CLI)

4. 从 ISR 的 C 源代码中，由编译器生成的代码假定 R1 被设置为 0。R1 的原始值 在 R1 被清除(6)前保存(5)

5. 在(7)和(8),所有寄存器都被按顺序保存

6. 现在被挂起的任务的栈中还有一个任务执行上下文的拷贝。内核保存任务的栈指针，所以 当任务恢复执行时，上下文可以取得并恢复。The X processor register is loaded with the address to which the stack pointer is to be saved (8 and 9). 载入栈指针

7. 栈指针保存，先低位(10,11)后高位(12 13)

Restoring the Context

portRESTORE_CONTEXT()是 portSAVE_CONTEXT().的逆过程。要恢复执行的任务的上下文被预先存储到任务栈中。实时内核为该任务取得栈指针，然后弹出 POP 上下文到 正确的寄存器中。

```
#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)
);
```

上面的代码中：

1. pxCurrentTCB 含有任务栈指针被取得的地址。这被载入到 X 寄存器(1 和 2)

2. 被恢复的任务的栈指针 载入到 AVR 的 stack pointer,先低字节(3)，后高半字节 nibble（4）

3.寄存器以相反的顺序从栈中，一直到 R1

4.状态寄存其保存在 stack 中的位置是在 R1 和 R0 之间，所以，它在 R0(7)前重新恢复(6)

下面介绍一个完整的上下文切换的例子

连起来(Putting It All Together)

在第 2 节的最后一部分 展示了这些 积木模块(building blocks)和源代码模块(source code modules)使如何来达到 一个在 AVR 微控制器上进行 RTOS 上下文切换的目的的。这个例子分几步演示了从一个低的优先级的任务 Task A,切换到高优先级的任务 Task B 的。

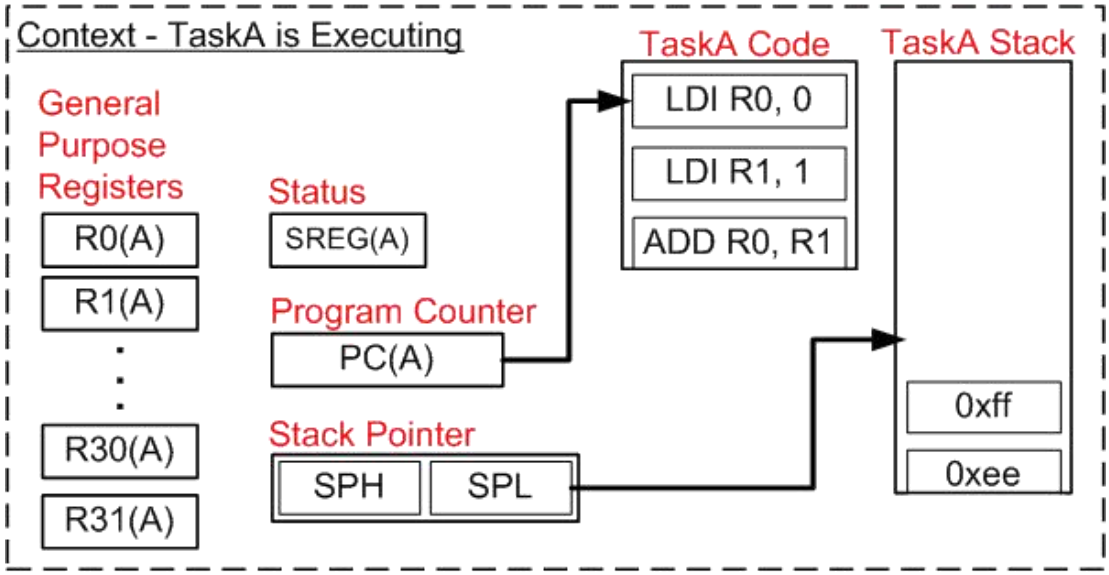
源代码与 WinAVR C 开发工具兼容。

上下文切换-第 1 步

在 RTOS tick 中断发生之前

这个例子，以 TaskA 的执行开始。TaskB 先前已经被挂起，所以它的上下文已经被保存到 TaskB 的 stack 里面。

TaskA 的上下文如下图所示：

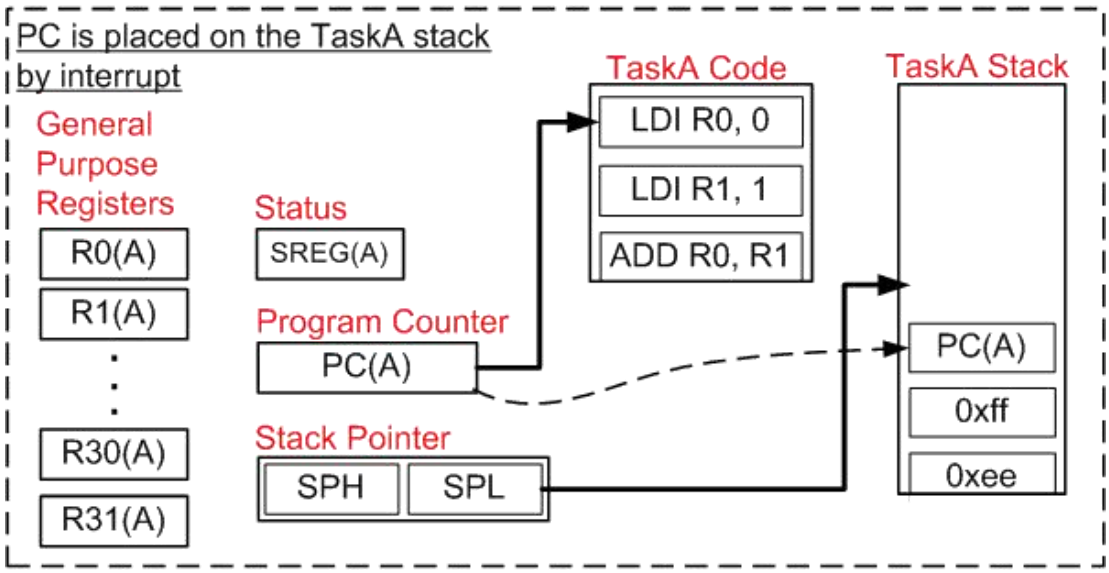


在每个寄存器里标上(A)，表示，这寄存器含有 任务 A 的上下文的正确值。

上下文切换—第 2 步

RTOS tick 中断 发生了

当 TaskA 将要执行到一个 LDI 指令的时候，RTOS tick 发生。当中断发生时，AVR 微控制器会在跳入到 RTOS tickISR 之前，自动放置当前的 PC 到 stack 里面。



上下文切换-第 3 步

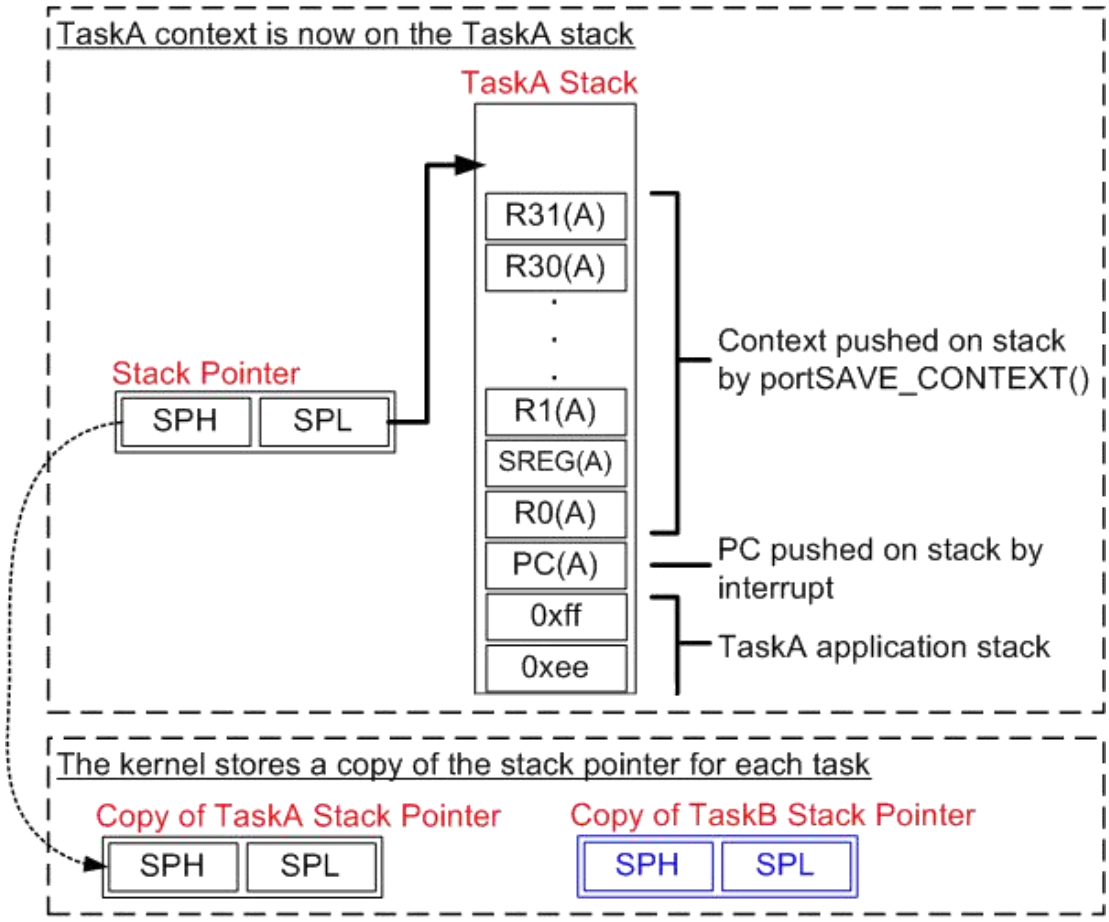
RTOS tick 中断(服务)执行

ISR 源代码如下。为便于阅读，移除注释。

```
/* 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()是一个 naked 函数。所以第一条指令是调用 vPortYieldFromTick()..vPortYieldFromTick()也是一个 naked 函数，所以 AVR 执行上下文被 portSAVE_CONTEXT().明确保存。

portSAVE_CONTEXT()将整个 AVR 执行上下文全部压入 TaskA 的 Stack..示意图如下。TaskA 的栈顶指针（Stack Pointer）现在指向它自己的上下文的顶部。portSAVE_CONTEXT()通过保存栈顶指针的一份拷贝来完成。在上次 TaskB 挂起时，实时内核已经复制了 TaskB 的栈顶指针。



上下文切换-第 4 步

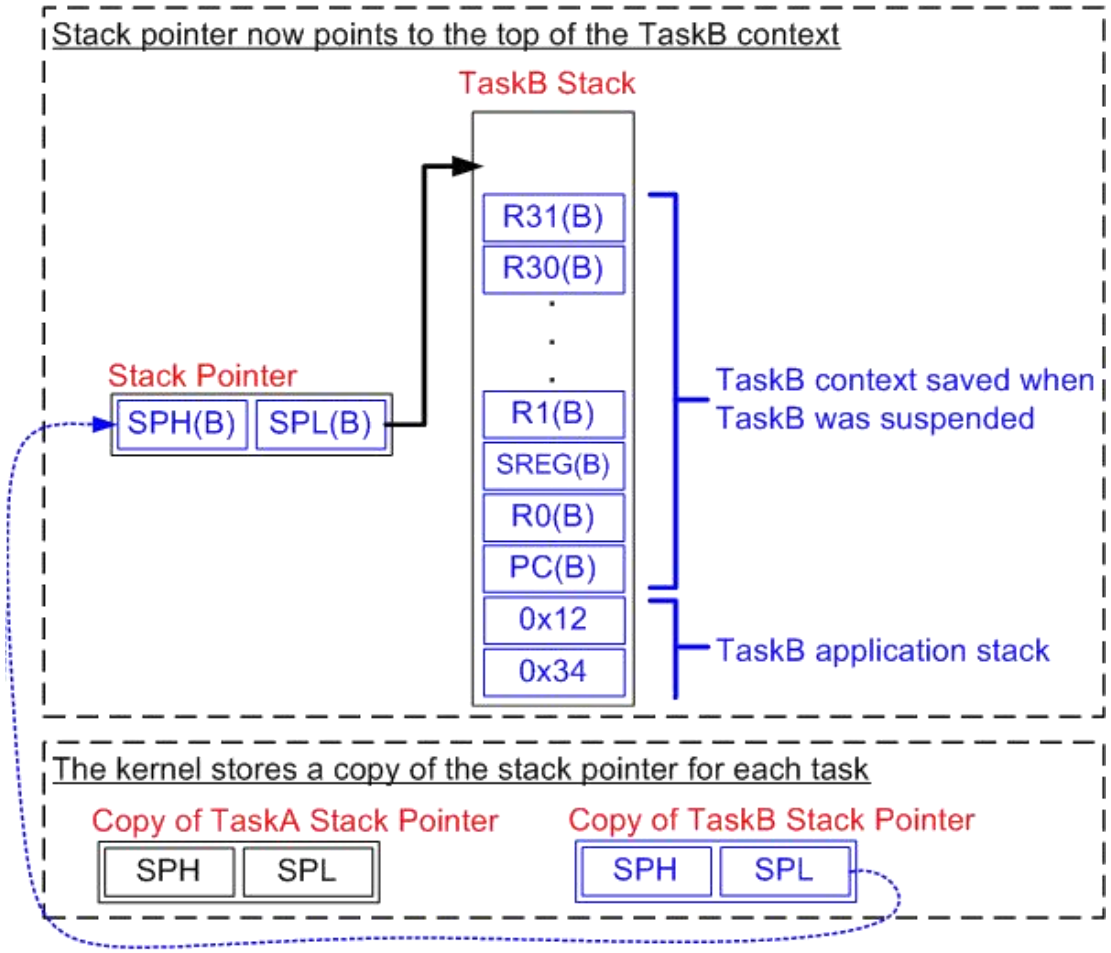
增加 tick count

在 TaskA 上下文被保存之后 再执行 vTaskIncrementTick()。这个例子，假定 tick count 的增加会引起 TaskB 准备运行。TaskB 比 TaskA 有更高的优先级，所以 vTaskSwitchContext()在 ISR 完成后将处理器交给 TaskB.

上下文切换-第 5 步

TaskB 的栈顶指针被取得

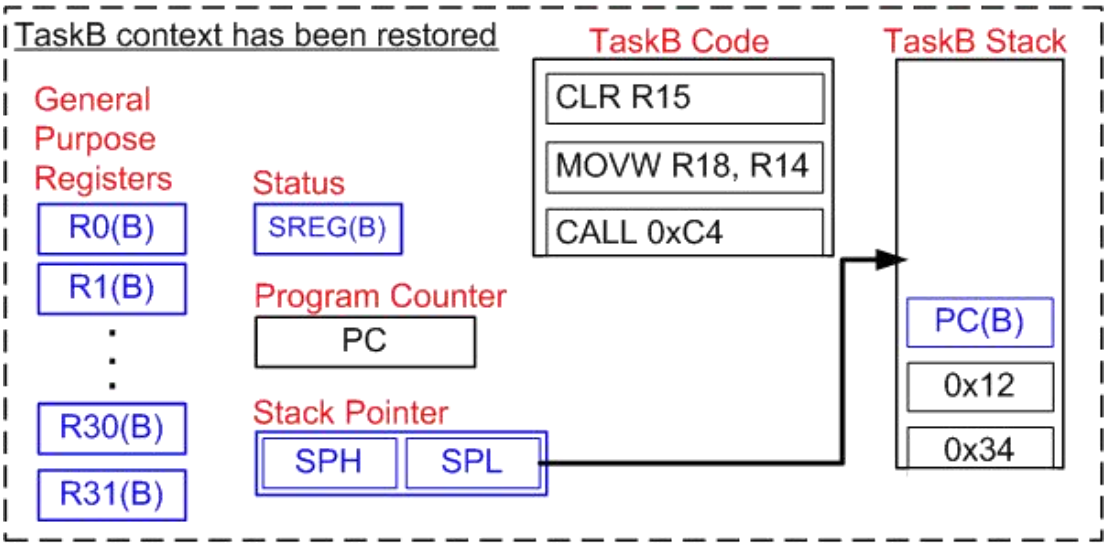
TaskB 的任务必须被恢复。portRESTORE_CONTEXT 要做的第 1 件事情就是从 TaskB 被挂起时保存的拷贝里 取得 TaskB 的栈顶指针。TaskB 的栈顶指针 被载入到处理器的 stack pointer.所以现在 AVR 的栈顶指针就是指向 TaskB 的上下文。



上下文切换-第 6 步

恢复 TaskB 的上下文

portRESTORE_CONTEXT()从 TaskB 的 stack 中恢复它的上下文到相应的处理器寄存器。



只有 PC(program counter)还留在 stack 中。

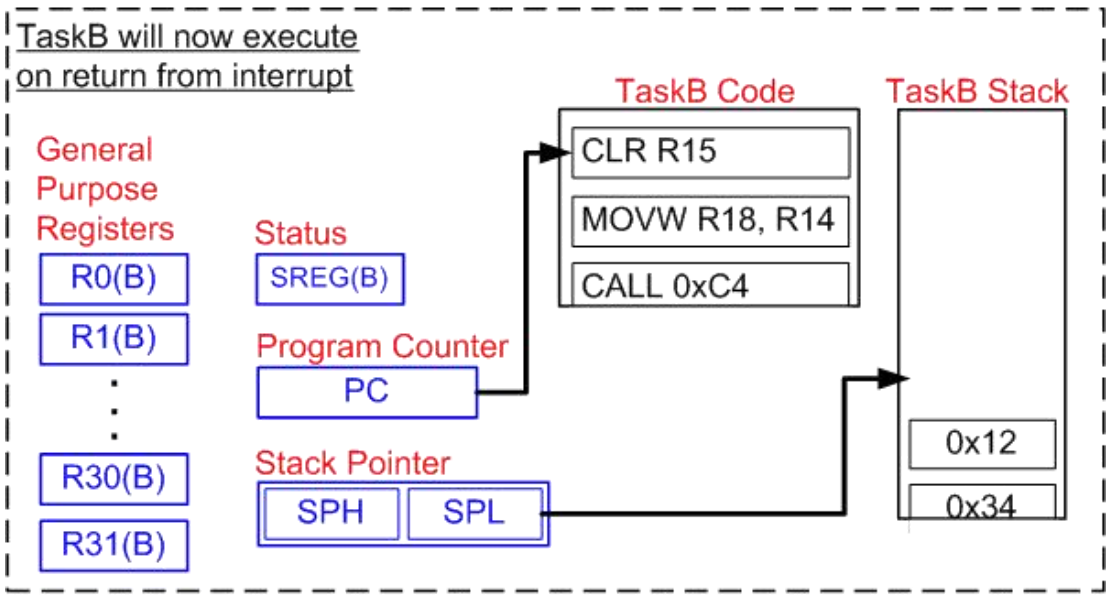
上下文切换-第 7 步

RTOS tick 退出

vPortYieldFromTick()返回到 SIG_OUTPUT_COMPARE1A()，在那里，最后一条指令是从中断返回(RETI). RETI 指令假定 stack 中的下个值就是中断发生时返回地址。

当 RTOS tick 中断发生时，AVR 自动放置 TaskA 的返回地址（在 TaskA 中的下条指令的地址）到 stack.。 ISR 修改 stack pointer 它指向了 TaskB 的 stack. 从而，RETI 指令从 stack 弹出的返

回地址是 TaskB 在挂起前 将要立即执行的指令的地址.



RTOS tick，它中断的是 TaskA,然而却返回到了 TaskB,这就完成了上下文切换。