uCOS-II 在 ATmega128 上的移植 Step by Step

－ICC-AVR 编译器

（ICC7.14C）

ba_wang_mao

2009 年 5 月

本文详细介绍了把 μC/OS- II 移植到 ATMEL 公司的 8 位微控制器 ATmega128 上的全过程（**编译器为 ICC-AVR**）。所谓移植，就是使一个实时内核能某个微处理器或微控制器上运行。在移植之前，希望读者能熟悉所用微处理器和 C 编译器的特点。

1. ATmega128 内核特点

之所以要先介绍 ATmega128MCU 内核特点，是因为在 μC/OS- II 的移植过程中，仍需要用户用 C 语言和汇编语言编写一些与微处理器相关的代码。这里主要介绍 ATmega128 与 μC/OS- II 移植相关的内核特点。如果读者已经对 ATmega128 比较了解了，那就不必阅读这一部分了。

1.1. 微控制器（MCU）

ATmega128 的 MCU 包括一个算术逻辑单元（ALU），一个状态寄存器（SREG），一个通用工作寄存器组和一个堆栈指针。状态寄存器（SREG）的最高位 I 是全局中断允许位。如果全局中断允许位为零，则所有中断都被禁止。当系统响应一个中断后，I 位将由硬件自动清“ 0 ”；当执行中断返回（RETI）指令时，I 位由硬件自动置“ 1 ”，从而允许系统再次响应下一个中断请求。

通用工作寄存器组是由 32 个 8 位的通用工作寄存器组成。其中 R26～R31 这 6 个寄存器还可以两两合并为 3 个 16 位的间接地址寄存器。这些寄存器可以用来对数据存储空间进行间接寻址。这 3 个间接地址寄存器的名称为：X 寄存器、Y 寄存器、Z 寄存器。其中 Z 寄存器还能用作对程序存储空间进行间接寻址的寄存器。有些 AVRC 语言编译器还把 Y 寄存器作为软件堆栈的堆栈指针，比如 ICC-AVR。

堆栈指针（SP）是一个指示堆栈顶部地址的 16 位寄存器。在 ICCAVR 中，它被用作指向硬件堆栈的堆栈指针。AVR 单片机上电复位后，SP 指针的初始值为 0x0000，由于 AVR 单片机的堆栈是向下生长的（从高地址向低地址生长），所以系统程序一开始必须对堆栈指针 SP 进行初始化，即将 SP 的值设为数据存储空间的最高地址。ICCAVR 编译器在链接 C 程序文件的时候，会自动在程序头链入 startup 文件。startup 文件里面的程序将会去做初始化 SP 指针的工作。链入 tartup 文件是 ICCAVR 这个编译器的特点，在用其它编译器的时候，希望读者确认所使用的编译器是否带有自动初始化 SP 的功能，若没有，应在用户程序中初始化 SP。

1.2. 数据存储空间（仅内部）

AVR 单片机的数据存储器是线形的，从低地址到高地址依次是 CPU 寄存器区（32 个通用寄存器），I/O 寄存器区，数据存储区 ICCAVR 编译器又将数据存储区划分为全局变量和字符串区，软件堆栈区和硬件堆栈区三个空间。

CPU 寄存器区	低地址
I/O 寄存器区	
全局变量和字符串区	
软件堆栈区	
硬件堆栈区	高地址

ICCAVR 编译器将堆栈分成了两个功能不同的堆栈来处理（这一点与 8051 系列的单片机编译器处理方式不同）。

硬件堆栈：用于储存子程序和中断服务子程序调用时的函数返回地址。这块数据区域由堆栈指针 SP 进行寻址，数据的进栈和出栈有专门的汇编指令（POP，PUSH 等）支持，所以叫做硬件堆栈区。

软件堆栈：用于传递参数，储存临时变量和局部变量。这块数据区域是用软件模拟堆栈储存数据的方式进行数据存储，用 Y 寄存器进行寻址，所以叫做软件堆栈区。

AVR 单片机的硬件堆栈的生长方向是向下的（从高地址向低地址生长），所以软件堆栈在定义的时候，也采取相同的生长方向。

这里没有用 ATmega128 而采用 AVR 单片机的提法是因为 ATmega128 属于 AVR 系列单片机中的一种，而所有的 AVR 单片机的数据存储器组织方式都是一致的。在创建 μC/OS- II 的任务栈时，需要了解所用微处理器数据存储空间尤其是堆栈空间的组形式及相关的操作。读者应参阅所用微处理器的资料和编译器的帮助文档，了解该部分知识。

1.3. Tmega128 的中断响应机制

ATmega128 有 34 个不同的中断源，每个中断源和系统复位在程序存储空间都有一个独立的中断向量（中断入口地址）。每个中断源都有各自独立的中断允许控制位，当某个中断源的中断允许控制位为“ 1 ”且全局中断允许位 I 也为“ 1 ”时，系统才响应该中断。

当系统响应一个中断请求后，会自动将全局中断允许位 I 清零，此时，后续中断响应被屏蔽。当系统执行中断返回指令 RETI 时，会将全局中断允许位 I 置“ 1 ”，以允许响应下一个中断。若用户想实现中断嵌套，必须在中断服务子程序中将全局中断允许位 I 置“ 1 ”。（这一点与 8051 系列的单片机不同）中断向量表中，处于低地址的中断具有高的优先级。优先级高只是表明在多个中断同时发生的时候，系统先响应优先级高的中断，并不含有高优先级的中断能打断低优先级的中断处理工程的意思。这与 8051 系列单片机的中断优先级概念不同。

由于 μC/OS- II 的任务切换实际上是模拟一次中断，因此需要知道 CPU 的中断响应机制。中断发生

时,ATmega128 按以下步骤顺序执行：

1、全局中断允许位 I 清零。

2、将指向下一条指令的 PC 值压入硬件堆栈，同时堆栈指针 SP 减 2。

3、选择最高优先级的中断向量装入 PC，程序从此地址继续执行中断处理。

4、当执行中断处理时，中断源的中断允许控制位清零。

中断结束后，执行 RETI 指令，此时

1、全局中断允许位 I 置“ 1 ”。

2、PC 从堆栈推出，程序从被中断的地方继续执行。

特别要注意的是：AVR 单片机在响应中断及从中断返回时，并不会对状态寄存器 SREG 和通用寄存器自动进行保存和恢复操作，因此，对状态寄存器 SREG 和通用寄存器的中断保护工作必须由用户来完成（只不过如果用户使用 C 语言编程，编译器已经帮你完成了上述寄存器入栈工作）。

1.4. Tmega128 的定时器中断

ATmega128 有三个定时器：T0,T1,T2，它们三者都有计数溢出中断功能，而且 T1 和 T2 还有匹配比较中断，即定时器计数到设定的值时，产生中断并自动清零。若系统采用这种中断方式，其好处是在中断服务程序 ISR 中不需要重新装载定时器的值。但本文出于通用性的考虑，仍采用定时器计数溢出中断方式。

2. μC/OS-Ⅱ的移植

2.1. 移植条件

要实现 μC/OS-Ⅱ的移植，所用的处理器和编译器必须满足一定的条件：

(1)、所用的 C 编译器能产生可重入代码。

可重入代码是指可以被一个以上的任务调用，而不必担心其数据会被破坏的代码。可重入代码任何时候都可以被中断，一段时间以后又可以重新运行，而相应的数据不会丢失，不可重入代码则不行。本文所使用 ImageCraft 公司的 ICCAVRV6.29 编译器能产生可重入代码。

(2)、用 C 语言就可以打开和关闭中断。

本文所使用的 ICCAVRV6.29 编译器支持在 C 语言中内嵌汇编语句且提供专门开关中断的宏：CLI()和 SEI()。这样，使得在 C 语言中开关中断非常方便。

(3)、处理器支持中断，并且能产生定时中断（通常在 10 至 100Hz 之间）

本文使用的 ATmega128，有 3 个定时器，能产生 μC/OS-Ⅱ所需的定时中断。

(4)、处理器支持能够容纳一定数量数据的硬件堆栈。

本文使用的 ATmega128 有 4KRAM，硬件堆栈可以开辟在这 4KRAM 中。

(5)、处理器有将堆栈指针和其它 CPU 寄存器从内存中读出和存储到堆栈或内存中的指令。

一般的单片机都满足这个要求(如 PUSH、POP 指令)，且 ATmega128 还具有直接访问 I/O 寄存器的指令（IN、OUT 等），它比 8051 系列的单片机更容易实现上述要求。

2.2. 移植的实现

μC/OS-Ⅱ的移植工作包括以下几个内容：

用 typedef 声明与编译器相关的 10 个数据类型（OS_CPU.H）

用#define 设置一个常量的值（OS_CPU.H）

#define 声明三个宏（OS_CPU.H）

用 C 语言编写六个简单的函数（OS_CPU_C.C）

编写四个汇编语言函数（OS_CPU_A.S）

根据这几项内容，本文逐步来完成。

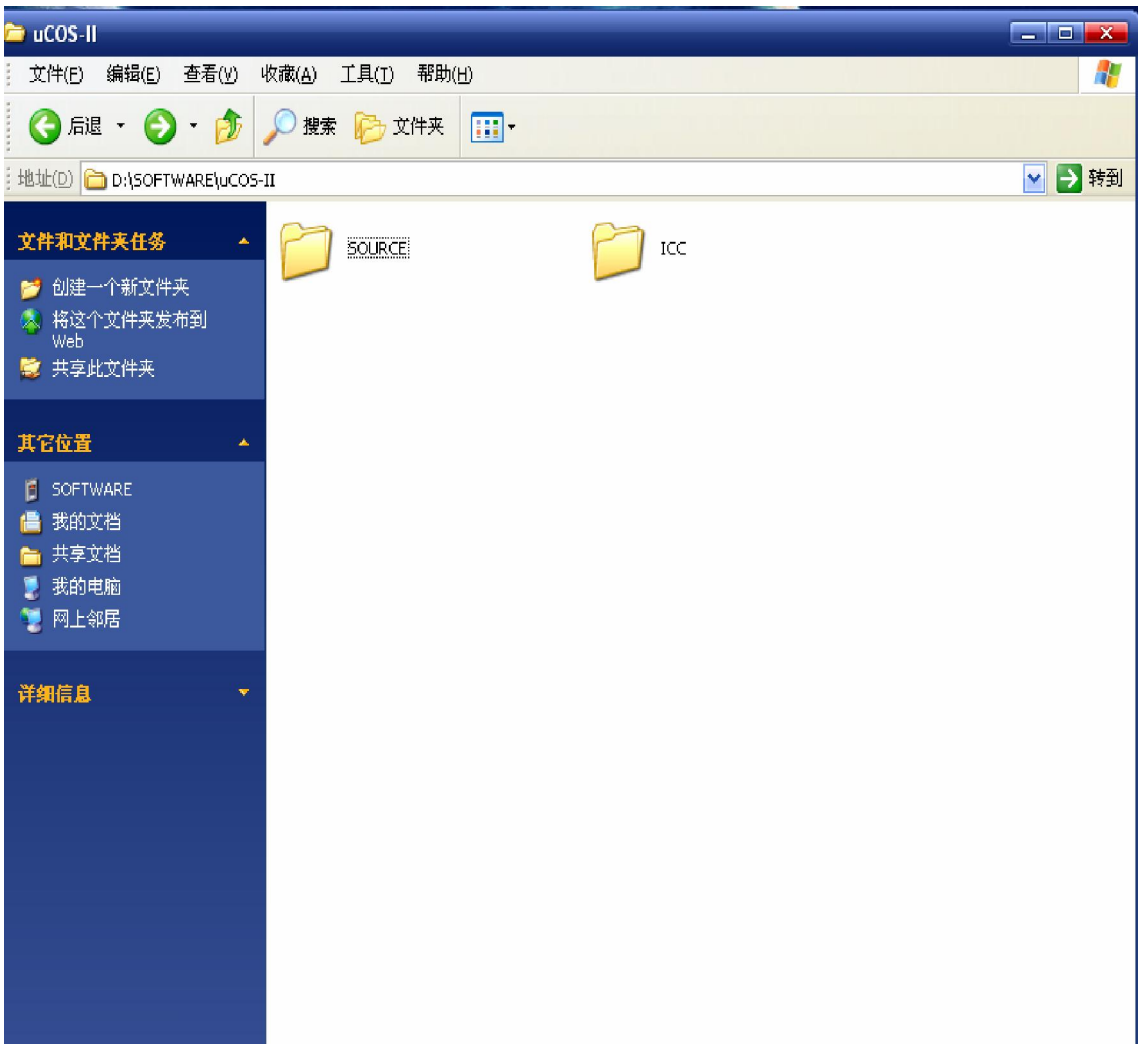
INCLUDES.H 文件

是主头文件，在所有后缀名为.C 的文件的开始都包含 INCLUDES.H 文件。使用 INCLUDES.H 的好处是所有的.C 文件都只包含一个头文件，简洁，可读性强。缺点是.C 文件可能会包含一些它并不需要的头文件，增加编译时间。我们是以增加编译时间为代价来换取程序的可移植性的。用户可以改写 INCLUDES.H 文件，增加自己的头文件，但必须加在文件末尾。

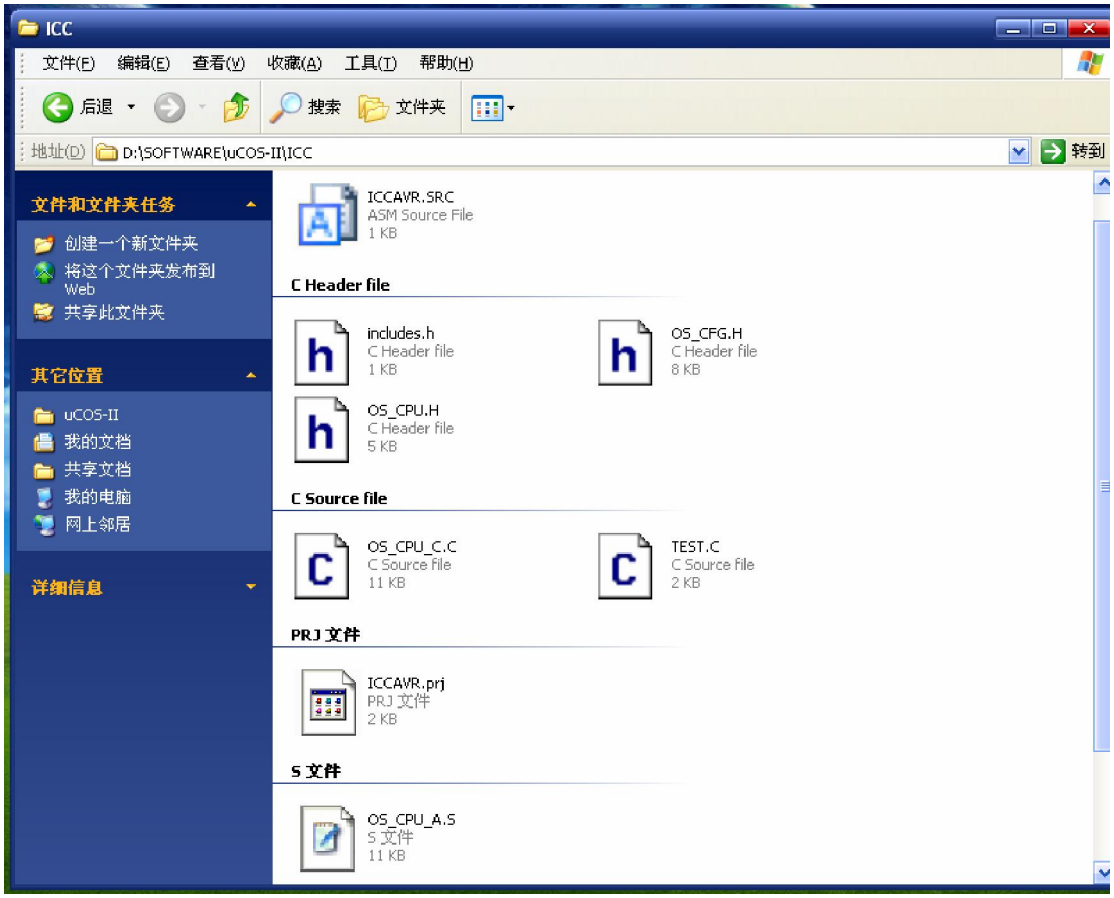
程序清单 INCLUDES.H.

```
#include <iom128v.h>
#include <macros.h>
#include      "os_cpu.h"
#include      "os_cfg.h"
#include      "..\source\ucos_ii.h"
```

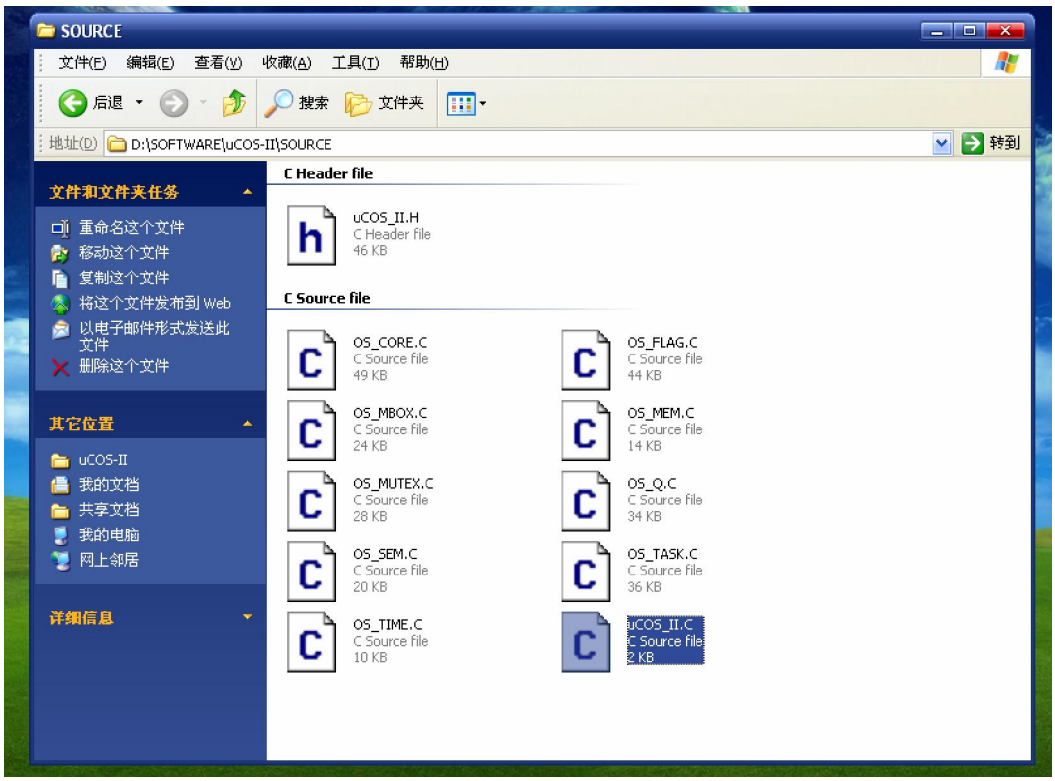
本移植路径如下图：



(2)、ICC 目录：存放工程文件



(2)、SOURCE 目录：存放 uC/OS-II 源程序



uCOS_II.C 文件

必须修改 D:\SOFTWARE\UCOS-II\SOURCE\UCOS-II.C 文件，否则编译不通过。下面是编译通过后的 UCOS-II.C 文件。

```
#define    OS_GLOBALS             /* Declare GLOBAL variables                                    */
#include "..\ICC\includes.h"


#define    OS_MASTER_FILE     /* Prevent the following files from including includes.h */
#include "..\source\os_core.c"
#include "..\source\os_flag.c"
#include "..\source\os_mbox.c"
#include "..\source\os_mem.c"
#include "..\source\os_mutex.c"
#include "..\source\os_q.c"
#include "..\source\os_sem.c"
#include "..\source\os_task.c"
#include "..\source\os_time.c"
```

OS_CPU.H 文件

OS_CPU.H 包括了用#define 定义的与处理器相关的常量、宏和类型定义。其中需要注意以下三点：

一、是堆栈的生长方向

正如前面所述，ATmega128 的堆栈生长方向是向下生长，即从高地址到低地址，因此，OS_STK_GROWTH 要被定义为 1。

二、是进入临界代码段(critical code section)的方法。μC/OS-II 提供了三种进入临界代码段的方法。

（1）、第一种方法是直接对中断允许位置 1 或清零，即进入临界代码段时，把中断允许位清零，退出临界代码段时，把中断允许位置 1；

（2）、第二种方法是进入临界代码段时，先将中断状态保存到堆栈中，然后关闭中断。与之对应的是，退出临界代码段时，从堆栈中恢复前面保存的中断状态。

（3）、第三种方法是，由于某些编译提供了扩展功能，用户可以得到当前处理器状态字的值，并将其保存在 C 函的局部变量之中。这个变量可用于恢复状态寄存器 SREG 的值。

由于 ICCAVR 不提供此项扩展功能，所以本文暂不考虑用第三种方法进入临界代码段。第一种方法存在着一个小小的问题：如果在关闭中断后调用 μC/OS-II 的功能函数，当函数返回后，中断可能会被打开。我们希望如果在调用 μC/OS-II 的功能函数前，中断是关着的，那么在函数返回后，中断仍然是关着的。方法 1 显然不满足要求。本文使用 μC/OS-II 的第二种方法——先将中断状态保存到堆栈中，然后关闭中断。

三、是任务切换函数 OS_TASK_SW()是个宏，具体的实现是在 OSCtxSw()（OS_CPU_A.S）中。

程序清单 OS_CPU.H.

```
#ifdef   OS_CPU_GLOBALS
#define   OS_CPU_EXT
#else
#define   OS_CPU_EXTextern
#endif
/*********************************************************************
*数据类型*(与编译器相关的内容)
*********************************************************************/
typedef   unsigned   char   BOOLEAN;
typedef      unsigned   char      INT8U;              //    无符号 8 位数
```

```
typedef     signed   char     INT8S;              //   带符号 8 位数
typedef     unsigned  int     INT16U;      //   无符号 16 位数
typedef     signed   int      INT16S;             //   带符号 16 位数
typedef     unsigned  long    INT32U;      //   无符号 32 位数
typedef     signed   long   INT32S;        //   带符号 32 位数
typedef     float   FP32;                  //   单精度浮点数
typedef     unsigned  char    OS_STK;      //   堆栈入口宽度为 8 位
typedef     unsigned  char    OS_CPU_SR; //   定义状态寄存器为 8 位
```

```
/*********************************************************************************
*方法 1：用简单指令开关中断。
*          注意，用方法 1 关闭中断，从调用函数返回后中断会重新打开。
*方法 2：关中断前保存中断被关闭的状态
*********************************************************************************/
```

```
#define   OS_CRITICAL_METHOD   2
#if   OS_CRITICAL_METHOD==1
#define   OS_ENTER_CRITICAL()          CLI()        //   关闭中断
#define   OS_EXIT_CRITICAL()           SEI()        //   打开中断
#endif

#if   OS_CRITICAL_METHOD==2
#define   OS_ENTER_CRITICAL()
          asm("ST   -Y,R16   \n              //   Y ß Y－1      ；（Y）ß R16
          IN   R16,0x3F   \n                 //   R16 ß SREG
          CLI   \n                           //   CLI
          PUSH   R16   \n                    //   PUSH   R16（R16压入硬件堆栈）
          LD   R16,Y+");                     //   R16 ß Y      ；Y ß Y＋1
#define   OS_EXIT_CRITICAL()
          asm("ST   -Y,R16   \n              //   Y ß Y－1      ；（Y）ß R16
          POP   R16   \n                     //   POP   R16
```

```
        OUT   0x3F,R16   \n              //   SREG ß R16
        LD   R16,Y+");                   //   R16 ß Y      ；Y ß Y＋1
#endif


#define   OS_STK_GROWTH      1          //   堆栈向下生长
#define   OS_TASK_SW()      OSCtxSw()   //   任务级任务切换（宏定义）
```

OS_CPU_C.C 文件

μC/OS-II 的移植需要用户编写 OS_CPU_C.C 中的十个函数：

OSTaskStkInit()；

OSInitHookBegin ()

OSInitHookEnd ()

OSTaskCreateHook()

OSTaskDelHook()

OSTaskSwHook()

OSTaskStatHook()

OSTimeTickHook()

OSTCBInitHook ()

OSTaskIdleHook ()

实际需要修改的只有OSTaskStkInit()函数，其它九个函数都是由用户定义的。如果用户需要使用这九个函数，可将文件OS_CFG.H中的#define constant OS_CPU_HOOKS_EN设为1，设为0表示不使用这些函数。本文自定义的任务堆栈结构下图所示。

函数OSTaskStkInit()是由OSTaskCreate()或OSTaskCreateExt()调用，用来初始化任务堆栈的。经初始化后的任务堆栈应该跟发生过一次中断后任务的堆栈结构一样。由前叙述可知，ATmega128在发生中断后，自动保存了程序计数器PC。为了保存全部现场，还需要保存状态寄存器SREG，R0～R31这32个通用寄存器及SP的值。

需要注意的是：μC/OS- II 规定，在建立任务时，只能传递一个参数给任务，而且这个参数是一个指针；ICCAVR 编译器规定，传递给函数的第一个参数是放在 R16、R17 中的，所以在 R16、R17 的位置中放置的是向任务传递的参数。R28、R29 的值不需要入栈，是因为 R28、R29 所组成的 Y 指针被用作软件堆栈的指针返回给调用函数。

根据上述自定义任务堆栈的结构，编写 OSTaskStkInit()。

程序清单 OS_CPU_C.C

```
#defineOS_CPU_GLOBALS
#include "INCLUDES.h"
/*********************************************************************************
OS_STK    *OSTaskStkInit(void(*task)(void*pd),void*pdata,OS_STK*ptos,INT16Uopt)
{
```

```
INT8U    soft_stk;                  //    软件堆栈而建立的临时指针
INT8U    hard_stk;                  //    硬件堆栈而建立的临时指针
INT16U    tmp;


opt=opt;                            //    'opt未使用，防止编译器编译时产生错误警告
soft_stk=(INT8U*)ptos;              //    软堆栈指针
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

OS_TASK_SOFT_STK_SIZE是指用户任务栈栈空间的大小，这个值在Os_cfg.h中设定。决定任务栈栈空间的大小是一件很困难的事情，因为不仅要计算任务本身的需求（局部变量、函数调用等），还需要计算最多中断嵌套层数（保存寄存器，中断服务子程序的局部变量等）。在本文的移植过程中，所建任务不是特别复杂，所以将任务栈栈空间的大小设定为80。

OS_TASK_HARD_STK_SIZE 是指默认的系统硬件堆栈空间的大小，这个值也是在Os_cfg.h中设定。本文设定为20，因为在程序嵌套程度不是很深的情况下，20个字节的硬件堆栈空间是足够了。当OS_TASK_HARD_STK_SIZE设定为20时，应该在ICCAVR编译器中，把Complier Option ->Target下的Return Stack Size也设置为20。这样设置后，编译器才会将硬件堆栈空间按照我们所希望的大小分配。需要注意的是：任务栈栈空间要根据具体的微处理器来定义，比如8051系列的微处理器没有软件堆栈和硬件堆栈的概念，就不需要在任务栈中划分硬件堆栈空间了。

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
hard_stk=(INT8U*)ptos-OS_TASK_SOFT_STK_SIZE–32)        //任务栈栈空间的大小（硬件堆栈）


tmp=*(INT16Uconst*)task;            //    任务的起始地址
*hard_stk--=(INT8U)tmp;
*hard_stk--=(INT8U)(tmp>>8);        //    任务的起始地址放入硬件堆栈
////////////////////////////////////////////////////////////////////////////////////////
//模仿PUSHA，保护现场（保护现象到软件堆栈＝Y指针访问）
////////////////////////////////////////////////////////////////////////////////////////
*soft_stk--=(INT8U)0x00;            //    R0=0x00
*soft_stk--=(INT8U)0x01;            //    R1=0x01
*soft_stk--=(INT8U)0x02;            //    R2=0x02
*soft_stk--=(INT8U)0x03;            //    R3=0x03
```

```
*soft_stk--=(INT8U)0x04;            //    R4=0x04
*soft_stk--=(INT8U)0x05;            //    R5=0x05
*soft_stk--=(INT8U)0x06;            //    R6=0x06
*soft_stk--=(INT8U)0x07;            //    R7=0x07
*soft_stk--=(INT8U)0x08;            //    R8=0x08
*soft_stk--=(INT8U)0x09;            //    R9=0x09
*soft_stk--=(INT8U)0x10;            //    R10=0x10
*soft_stk--=(INT8U)0x11;            //    R11=0x11
*soft_stk--=(INT8U)0x12;            //    R12=0x12
*soft_stk--=(INT8U)0x13;            //    R13=0x13
*soft_stk--=(INT8U)0x14;            //    R14=0x14
*soft_stk--=(INT8U)0x15;            //    R15=0x15
////////////////////////////////////////////////////////////////////////////////
//模仿任务的调用（R16、R17中放置向任务传递的参数）
////////////////////////////////////////////////////////////////////////////////
tmp=(INT16U)pdata;
*soft_stk--=(INT8U)tmp;
*soft_stk--=(INT8U)(tmp >> 8);
////////////////////////////////////////////////////////////////////////////////
//模仿PUSHA，保护现场（软件堆栈＝Y指针访问）
////////////////////////////////////////////////////////////////////////////////
*soft_stk--=(INT8U)0x18;            //    R18=0x18
*soft_stk--=(INT8U)0x19;            //    R19=0x19
*soft_stk--=(INT8U)0x20;            //    R20=0x20
*soft_stk--=(INT8U)0x21;            //    R21=0x21
*soft_stk--=(INT8U)0x22;            //    R22=0x22
*soft_stk--=(INT8U)0x23;            //    R23=0x23
*soft_stk--=(INT8U)0x24;            //    R24=0x24
*soft_stk--=(INT8U)0x25;            //    R25=0x25
*soft_stk--=(INT8U)0x26;            //    R26=0x26
```

```
*soft_stk--=(INT8U)0x27;                //     R27=0x27
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//R28、R29用作软件堆栈的指针Y，因此不需要保护
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
*soft_stk--=(INT8U)0x30;                //     R30=0x30
*soft_stk--=(INT8U)0x31;                //     R31=0x31

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//模仿PUSH    PSW
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
*soft_stk--=(INT8U)0x80;                //     SREG=0x80，开全局中断

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//将硬件堆栈指针SP，压入当前任务的软件堆栈中
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
tmp=(INT16U)hard_stk;
*soft_stk--=(INT8U)(tmp>>8);             //     SPH
*soft_stk=(INT8U)tmp;                   //     SPL

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//返回软件堆栈栈顶指针Y
//说明：创建任务时，OSTaskCreate（）会首先调用OSTaskStkInit（）得到当前任务软件
//堆栈的栈顶指针Y，然后调用OS_TCBInit（）将软件堆栈的栈顶指针Y保存到任务控制
//块的OSTCBStkPtr成员中。
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
return((void*)soft_stk);
}
```

OS_CPU_A.S 文件

μC/OS-II 的移植需要用户编写OS_CPU_A.S中的四个函数：OSStartHighRdy()；OSTaskSwitch()；OSIntCtxSw()；OSTickISR()；以及入栈和出栈宏定义栈操作，因此需要用汇编语言编写。注意，在ICCAVR中，汇编语言文件是以.S而不是.asm作为文件后缀名的。

```
macro PUSHA                    //   通用寄存器入栈宏定义
    ST      -Y,R0              //   Y ß Y－1    ；（Y）  ß R0
    ST      -Y,R1              //   Y ß Y－1    ；（Y）  ß R1
    ST      -Y,R2              //   Y ß Y－1    ；（Y）  ß R2
    ST      -Y,R3              //   Y ß Y－1    ；（Y）  ß R3
    ST      -Y,R4              //   Y ß Y－1    ；（Y）  ß R4
    ST      -Y,R5              //   Y ß Y－1    ；（Y）  ß R5
    ST      -Y,R6              //   Y ß Y－1    ；（Y）  ß R6
    ST      -Y,R7              //   Y ß Y－1    ；（Y）  ß R7
    ST      -Y,R8              //   Y ß Y－1    ；（Y）  ß R8
    ST      -Y,R9              //   Y ß Y－1    ；（Y）  ß R9
    ST      -Y,R10             //   Y ß Y－1    ；（Y）  ß R10
    ST      -Y,R11             //   Y ß Y－1    ；（Y）  ß R11
    ST      -Y,R12             //   Y ß Y－1    ；（Y）  ß R12
    ST      -Y,R13             //   Y ß Y－1    ；（Y）  ß R13
    ST      -Y,R14             //   Y ß Y－1    ；（Y）  ß R14
    ST      -Y,R15             //   Y ß Y－1    ；（Y）  ß R15
    ST      -Y,R16             //   Y ß Y－1    ；（Y）  ß R16
    ST      -Y,R17             //   Y ß Y－1    ；（Y）  ß R17
    ST      -Y,R18             //   Y ß Y－1    ；（Y）  ß R18
    ST      -Y,R19             //   Y ß Y－1    ；（Y）  ß R19
    ST      -Y,R20             //   Y ß Y－1    ；（Y）  ß R20
    ST      -Y,R21             //   Y ß Y－1    ；（Y）  ß R21
    ST      -Y,R22             //   Y ß Y－1    ；（Y）  ß R22
    ST      -Y,R23             //   Y ß Y－1    ；（Y）  ß R23
```

```
    ST      -Y,R24          //  Y ß Y－1    ；（Y）  ß R24
    ST      -Y,R25          //  Y ß Y－1    ；（Y）  ß R25
    ST      -Y,R26          //  Y ß Y－1    ；（Y）  ß R26
    ST      -Y,R27          //  Y ß Y－1    ；（Y）  ß R27
    ST      -Y,R30          //  Y ß Y－1    ；（Y）  ß R30
    ST      -Y,R31          //  Y ß Y－1    ；（Y）  ß R31
endmacro

macro POPA                  //  通用寄存器出栈宏定义
    LD      R31,Y+          //  R31 ß （Y）     ；Y ß Y＋1
    LD      R30,Y+          //  R30 ß （Y）     ；Y ß Y＋1
    LD      R27,Y+          //  R27 ß （Y）     ；Y ß Y＋1
    LD      R26,Y+          //  R26 ß （Y）     ；Y ß Y＋1
    LD      R25,Y+          //  R25 ß （Y）     ；Y ß Y＋1
    LD      R24,Y+          //  R24 ß （Y）     ；Y ß Y＋1
    LD      R23,Y+          //  R23 ß （Y）     ；Y ß Y＋1
    LD      R22,Y+          //  R22 ß （Y）     ；Y ß Y＋1
    LD      R21,Y+          //  R21 ß （Y）     ；Y ß Y＋1
    LD      R20,Y+          //  R20 ß （Y）     ；Y ß Y＋1
    LD      R19,Y+          //  R19 ß （Y）     ；Y ß Y＋1
    LD      R18,Y+          //  R18 ß （Y）     ；Y ß Y＋1
    LD      R17,Y+          //  R17 ß （Y）     ；Y ß Y＋1
    LD      R16,Y+          //  R16 ß （Y）     ；Y ß Y＋1
    LD      R15,Y+          //  R15 ß （Y）     ；Y ß Y＋1
    LD      R14,Y+          //  R14 ß （Y）     ；Y ß Y＋1
    LD      R13,Y+          //  R13 ß （Y）     ；Y ß Y＋1
    LD      R12,Y+          //  R12 ß （Y）     ；Y ß Y＋1
    LD      R11,Y+          //  R11 ß （Y）     ；Y ß Y＋1
    LD      R10,Y+          //  R10 ß （Y）     ；Y ß Y＋1
```

```
    LD      R9,Y+                   //  R9 ß  (Y)       ;Y ß Y+1
    LD      R8,Y+                   //  R8 ß  (Y)       ;Y ß Y+1
    LD      R7,Y+                   //  R7 ß  (Y)       ;Y ß Y+1
    LD      R6,Y+                   //  R6 ß  (Y)       ;Y ß Y+1
    LD      R5,Y+                   //  R5 ß  (Y)       ;Y ß Y+1
    LD      R4,Y+                   //  R4 ß  (Y)       ;Y ß Y+1
    LD      R3,Y+                   //  R3 ß  (Y)       ;Y ß Y+1
    LD      R2,Y+                   //  R2 ß  (Y)       ;Y ß Y+1
    LD      R1,Y+                   //  R1 ß  (Y)       ;Y ß Y+1
    LD      R0,Y+                   //  R0 ß  (Y)       ;Y ß Y+1
endmacro

macro PUSH_SP                       //  SP寄存器入栈宏定义
    IN      R16,SPH                 //  R16 ß SPH
    ST      -Y,R16                  //  Y ß Y－1     ;（Y） ß R16
    IN      R16,SPL                 //  R16 ß SPL
    ST      -Y,R16                  //  Y ß Y－1     ;（Y） ß R16
endmacro

macro POP_SP                        //  SP寄存器出栈宏定义
    LD      R16,Y+                  //  R16 ß  (Y)      ;Y ß Y+1
    OUT     SPL,R16                 //  SPL ß R16
    LD      R16,Y+                  //  R16 ß  (Y)      ;Y ß Y+1
    OUT     SPH,R16                 //  SPH ß R16
endmacro

macro PUSH_PSW                      //  SREG入栈宏定义
    IN      R16,SREG                //  R16 ß SREG
    ST      -Y,R16                  //  Y ß Y－1     ;（Y） ß R16
endmacro
```

```
macro POP_PSW                   //  SREG出栈宏定义
    LD      R16,Y+              //  R16 ß  （Y）    ；Y ß Y＋1
    OUT     SREG,R16            //  SREG ß R16
endmacro
```

（1）、**OSStartHighRdy()**

```
;*********************************************************************************
;                                  START HIGHEST PRIORITY TASK READY-TO-RUN
```

（多任务刚刚开始运行，首先执行OSStartHighRdy（）调度刚刚创建任务优先级最高的一个任务）

```
; Description : This function is called by OSStart() to start the highest
; priority task that was created by your application before calling OSStart().

;     1) The (data)stack frame is assumed to look as follows:（软件堆栈区域示意图如下）
;OSTCBHighRdy->OSTCBStkPtr -->  LSB of (return) stack pointer                (Low memory)
;                               MSB of (return) stack pointer   ＝SP指针
;                               Flags to load in status register  ＝PSW
;                               R31                               ＝R31
;                               R30
;                               R27
;                               R26
;                               R25
;                               R24
;                               R23
;                               R22
;                               R21
;                               R20
;                               R19
;                               R18
;                            .  R17（）
;                               R16（）
;                               R15
;                               R14
;                               R13
;                               R12
;                               R11
;                               R10
;                               R9
;                               R8
;                               R7
;                               R6
;                               R5
;                               R4
;                               R3
;                               R2
;                            .  R1
```

```
;                                               R0                                              (High memory)
;;          where the stack pointer points to the task start address.
;
;     2) OSStartHighRdy()     MUST:
;               （a）Call OSTaskSwHook() then,
;               （b）Set OSRunning to TRUE,
;               （c）Switch to the highest priority task.
;*********************************************************************************************************************
_OSStartHighRdy::
    CALL     _OSTaskSwHook              ；调用用户定义的任务切换接口函数
    LDS      R16,_OSRunning
    INC      R16
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
ICCAVR规定：在汇编程序中，向编译器声明一个全局变量是在该变量的标识符前加上一条下划线作为前缀。_OSRunning表示OSRunning是一个全局变量；向编译器声明一个外部函数则是在该函数的标识符前加上一条下划线并在其后加上两个冒号。_OSStartHighRdy::表示OSStartHighRdy是个外部函数，其它C言语文件可以以OSStartHighRdy（）的形式对它进行调用。
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    STS      _OSRunning,R16             ;设置OSRunning=TRUE，表明多任务调度开始

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//让Z指针指向高优先级任务的任务控制块OSTCBHighRdy
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    LDS      R30,_OSTCBHighRdy          ; R30 ß OSTCBHighRdy
    LDS      R31,_OSTCBHighRdy+1        ; R31 ß OSTCBHighRdy

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//高优先级任务的栈顶指针 à Y指针
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    LD       R28,Z+                     ; Load Y （R29：R28） pointer
    LD       R29,Z+             ;
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//我们知道，在创建任务时，首先调用了任务堆栈初始化子程序OSTaskStkInit（）依次将
//（1）、任务的起始地址压入硬件堆栈（2）、R0～R31压入软件堆栈（R28、R29除外）
//（3）、SREG压入软件堆栈（4）、硬件堆栈指针SP压入软件堆栈
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    POP_SP                          ; 装入当前任务的SP（从软件堆栈中恢复硬件堆栈指针SP）
    POP_PSW                         ; 装入当前任务的SREG（从软件堆栈中恢复全局中断）
    POPA                            ; 装入当前任务的寄存器（从软件堆栈中恢复现场）
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//RET＝子程序返回。
//操作： SP ß SP＋2       ；  PC ß STACK
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        RET                             ; 开始执行当前任务（从硬件堆栈中弹出任务的起始地址，
                                        ; 执行高优先级任务
```

（2）、**OSCtxSw()**

OSCtxSw()是一个任务级的任务切换函数，它主要完成以下几件事：保存当前任务现场；保存当前任务的任务栈指针到当前任务的任务控制块；切换最高优先级任务为当前任务；使SP指向最高优先级任务的任务栈的栈顶；恢复新任务的运行环境。由于ATmega128没有软中断指令，只能通过汇编子程序来模拟中断。

```
;*********************************************************************************
                                TASK LEVEL CONTEXT SWITCH
                                （任务级任务切换）
; Description : This function is called when a task makes a higher priority task ready-to-run.
;
; Note(s):
;     1) Upon entry,（入口）
;          OSTCBCur             points to the OS_TCB of the task to suspend
;                               OSTCBCur＝指向将要被挂起任务的任务控制块指针
;          OSTCBHighRdy         points to the OS_TCB of the task to resume
;                               OSTCBHighRdy＝指向将要获取CPU控制权任务的任务控制块指针
;
;     2) The stack frame of the task to suspend looks as follows:（硬件堆栈区域示意图如下）
;                                                                    (Low memory)
;          SP  +0 -->           LSB of task code address  ＝任务起始地址Lo
;              +1               MSB of task code address ＝任务起始地址Hi
;                                                                    (High memory)
;     3) The saved context of the task to resume looks as follows:（软件堆栈区域示意图如下）
;                                                                    (Low memory)
;OSTCBHighRdy->OSTCBStkPtr -->  LSB of (return) stack pointer
;                               MSB of (return) stack pointer      ＝SP指针
;                               Flags to load in status register   ＝PSW
;                               R31                                ＝R31
;                               R30                                ＝R30
;                               R27
;                               R26
;                               R25
```

```
;                                       R24
;                                       R23
;                                       R22
;                                       R21
;                                       R20
;                                       R19
;                                       R18
;                                    .  R17（）
;                                       R16（）
;                                       R15
;                                       R14
;                                       R13
;                                       R12
;                                       R11
;                                       R10
;                                       R9
;                                       R8
;                                       R7
;                                       R6
;                                       R5
;                                       R4
;                                       R3
;                                       R2
;                                    .  R1
;                                       R0
;                                                                    (High memory)
;*******************************************************************************
_OSCtxSw::
        PUSHA                   ; 保存将要被挂起任务的现场（R0～R31）到软件堆栈区域
        PUSH_PSW                ; 保存将要被挂起任务的全局中断SREG 到软件堆栈区域
        PUSH_SP                 ; 保存将要被挂起任务的硬件堆栈指针SP到软件堆栈区域


//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
保存当前任务软件堆栈的栈顶指针到当前任务的任务控制块TCB
（1）、Z指针指向当前将要被挂起任务的任务控制块TCB  OSTCBStkPtr成员
（2）、保存当前任务软件堆栈的栈顶指针Y到Z指针指向的OSTCBStkPtr成员
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        LDS     R30,_OSTCBCur     ; R30 ß OSTCBCur                          （1）
        LDS     R31,_OSTCBCur+1   ; R30 ß OSTCBCur＋1
```

```
        ST      Z+,R28              ; Z ß Z＋1     ；（Z）  ß R28              （2）
        ST      Z+,R29              ; Save Y（R28、R29）pointer
        CALL    _OSTaskSwHook       ; 调用用户接口程序

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//把最高优先级的任务切换为当前任务（高优先级任务的ID号就成为当前任务的ID号）
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        LDS     R16,_OSPrioHighRdy
        STS     _OSPrioCur,R16      ; OSPrioCur = OSPrioHighRdy

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
把最高优先级任务的任务控制块TCB切换为当前任务的任务控制块TCB
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        LDS     R30,_OSTCBHighRdy   ; Z指针＝OSTCBHighRdy->OSTCBStkPtr
        LDS     R31,_OSTCBHighRdy+1
        STS     _OSTCBCur,R30       ; OSTCBCur = OSTCBHighRdy
        STS     _OSTCBCur+1,R31     ;

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//得到最高优先级任务的软件堆栈栈顶指针 è Y
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        LD      R28,Z+              ; Restore Y （R28、R29）pointer
        LD      R29,Z+              ;

        POP_SP                      ; 从高优先级任务的软件堆栈中恢复硬件堆栈指针SP
        POP_PSW                     ; 从高优先级任务的软件堆栈中恢复全局中断 SREG
        POPA                        ; 从高优先级任务的软件堆栈中恢复现场（R0－R31）
        RET                         ; 从硬件堆栈中弹出任务的起始地址，执行高优先级任务
```

虽然新任务有可能还是原来正在运行的任务，如果原来任务的优先级仍然是最高的话。这里的新任务指经过任务切换后，即将运行的任务。

（3）、**OSIntCtxSw()**

由于中断可能会使更高优先级的任务进入就绪态。为了让更高优先级的任务能立即运行，所以需要在中断中进行任务切换。在中断服务子程序的最后，OSIntExit（）函数会调用OSIntCtxSw()做任务切换。OSIntCtxSw()是一个中断级的任务切换函数。由于在此之前，中断服务程序已经保存了被中断任务的现场，因此不需要再保存现场了。

```
;*********************************************************************************
                                INTERRUPT LEVEL CONTEXT SWITCH
                                （中断级任务切换）
; Description : This function is called by OSIntExit() to perform a context
; switch to a task that has been made ready-to-run by an ISR.
;
; Note(s):
;     1) Upon entry,（入口）
;         OSTCBCur      points to the OS_TCB of the task to suspend
; ;                          OSTCBCur＝指向将要被挂起任务的任务控制块指针
          OSTCBHighRdy points to the OS_TCB of the task to resume
;                          OSTCBHighRdy＝指向将要获取CPU控制权任务的任务控制块指针

;     2) The stack frame of the task to suspend looks as follows:（硬件堆栈区域示意如下图）
;                                                                    (Low memory)
;         SP  +0 -->   LSB of return address of OSIntCtxSw()  ＝OSIntCtxSw()返回地址Lo
;             +1       MSB of return address of OSIntCtxSw()  ＝OSIntCtxSw()返回地址Hi
;             +2       LSB of return address of OSIntExit()   ＝OSIntExit()返回地址Lo
;             +3       MSB of return address of OSIntExit()   ＝OSIntExit()返回地址Hi
;             +4       LSB of task code address               ＝任务起始地址Lo
;             +5       MSB of task code address               ＝任务起始地址Hi
                                                                    (High memory)
;     3) The saved context of the task to resume looks as follows:（软件堆栈区域示意图如下）
;                                                                    (Low memory)
;;         OSTCBHighRdy->OSTCBStkPtr -->      LSB of (return) stack pointer
```

```
;                                               MSB of (return) stack pointer   ＝SP指针
;                                               Flags to load in status register  ＝PSW
;                                               R31                             ＝R31
;                                               R30                             ＝R30
;                                               R27
;                                               R26
;                                               R25
;                                               R24
;                                               R23
;                                               R22
;                                               R21
;                                               R20
;                                               R19
;                                               R18
;                                           .   R17（）
;                                               R16（）
;                                               R15
;                                               R14
;                                               R13
;                                               R12
;                                               R11
;                                               R10
;                                               R9
;                                               R8
;                                               R7
;                                               R6
;                                               R5
;                                               R4
;                                               R3
;                                               R2
;                                           .   R1
;                                               R0                              (High memory)
;*------------------------------------------------------------------------------
_OSIntCtxSw::
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//Z指针＝SP，即Z指针指向硬件堆栈栈顶
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        IN      R30,SPL                 ; R30 ß SPL
        IN      R31,SPH                 ; R31 ß SPH
```

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//调整Z指针，使得Z指针指向硬件堆栈的“ 任务起始地址 ”。也就是丢弃硬件堆栈中
//通过执行CALL    OSIntCtxSw()  和  执行CALL    OSIntExit()  压入硬件堆栈中的上
//述两个函数的返回地址
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    ;    ADIW     R30,4 ;
         ADIW     R30,5                    ;
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//Save SP（保存硬件堆栈栈顶指针SP到Y指针）
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
         ST       -Y,R31                   ;    Y ß Y－1     ；（Y）  ß R31
         ST       -Y,R30                   ;    Y ß Y－1     ；（Y）  ß R30
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
Z = OSTCBCur->OSTCBStkPtr（Z指针指向待挂起任务的任务控制块的软件堆栈栈顶指针）
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
         LDS      R30,_OSTCBCur            ;
         LDS      R31,_OSTCBCur+1
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//保存待挂起任务的任务控制块的软件堆栈栈的顶指针到Y指针
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
         ST       Z+,R28                   ; Save Y （R28、R29）pointer
         ST       Z+,R29                ;
         CALL     _OSTaskSwHook            ; Call user defined task switch hook
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//OSPrioCur = OSPrioHighRdy
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
         LDS      R16,_OSPrioHighRdy       ; OSPrioCur = OSPrioHighRdy
         STS      _OSPrioCur,R16 ;
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// OSTCBCur = OSTCBHighRdy（当前任务的任务控制块指针就指向高优先级任务的任务控制块）
```

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        LDS     R30,_OSTCBHighRdy     ; Z = OSTCBHighRdy->OSTCBStkPtr
        LDS     R31,_OSTCBHighRdy+1   ;
        STS     _OSTCBCur,R30         ; OSTCBCur = OSTCBHighRdy
        STS     _OSTCBCur+1,R31       ;

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//从高优先级任务的OSTCBStkPtr成员中得到高优先级任务的软件堆栈栈顶指针 è Y
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        LD      R28,Z+                ; Restore Y pointer
        LD      R29,Z+

        POP_SP                        ; 从高优先级任务的软件堆栈中恢复硬件堆栈指针SP
        POP_PSW                       ; 从高优先级任务的软件堆栈中恢复全局中断 SREG
        POPA                          ; 从高优先级任务的软件堆栈中恢复现场（R0－R31）
        RET                           ; 从硬件堆栈中弹出任务的起始地址，执行高优先级任务
```

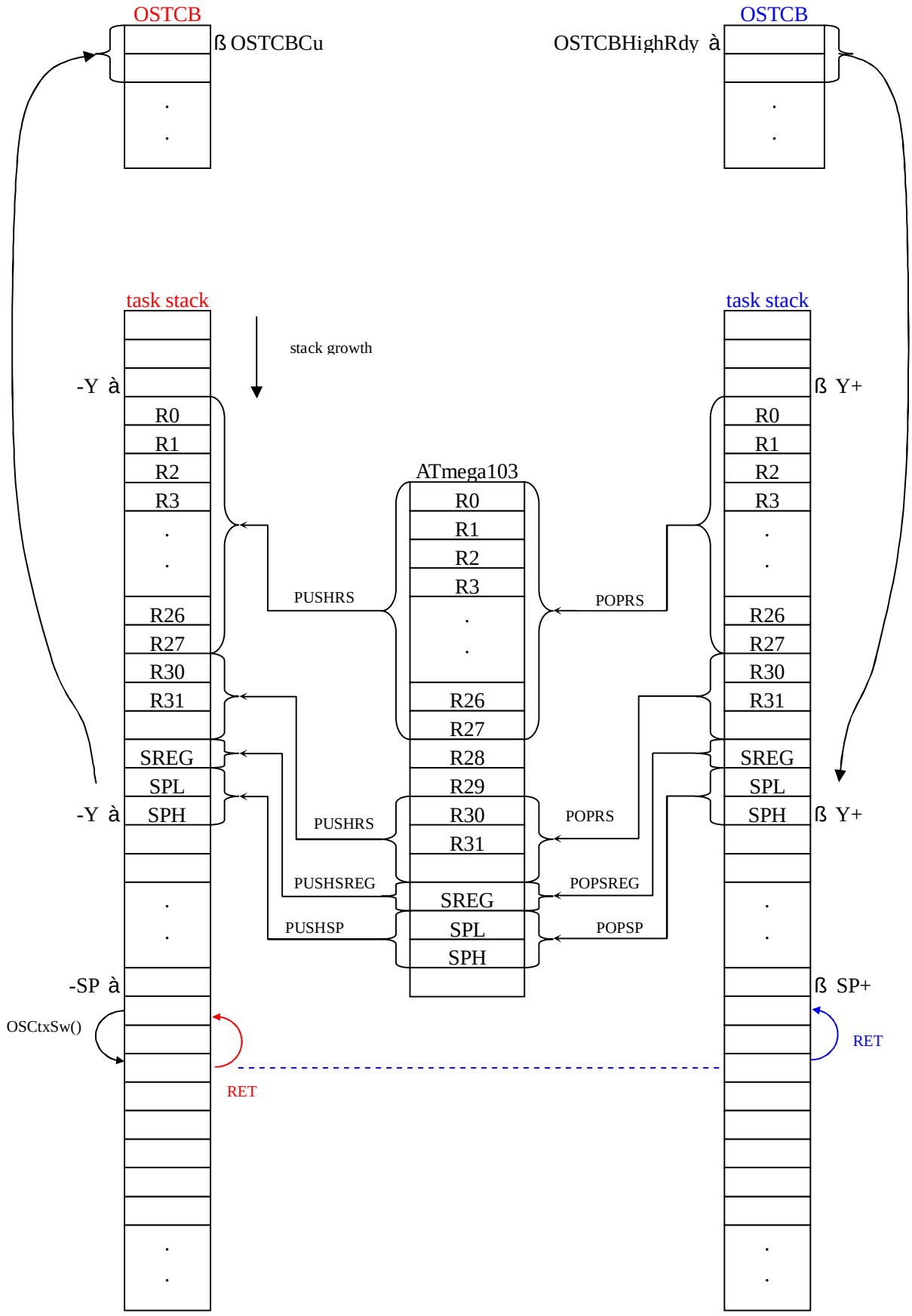
（4）、OSTickISR()

```
;*********************************************************************************
                                        ; SYSTEM TICK ISR
                                        系统时钟节拍
; Description: This function is the ISR used to notify uC/OS-II that a system
; tick has occurred.
;;*********************************************************************************

_OSTickISR::
        PUSHA                   ; 保存现场

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

如前所述，当中断发生后，硬件会自动将SREG寄存器的全局中断允许位I清零，执行中断返回指令RETI以后，硬件会自动将全局中断允许位置“1”。那么，我们为什么还要在保存SREG寄存器前使它的保存值重新带上允许中断的信息呢，我们为什么不直接用中断返回指令RETI而要用函数返回指令RET呢？原因是这样的，假设任务A正在执行，这时系统时钟节拍到了，μC/OS-II开始进行时钟节拍服务，它先保存A任务的现场，然后执行中断服务程序，在退出中断以前，μC/OS-II调用OSIntExit（）进行任务切换，如果这时有一个优先级比A高的任务B就绪，这样，恢复新任务的运行环境就是恢复B任务的环境，A任务的运行环境将被保存在A任务的任务栈中。试想，如果在保存A任务的SREG寄存器的值的时候，没有使它带上允许中断的信息，那在下次恢复A任务的运行环境时，中断将会被永远关掉。

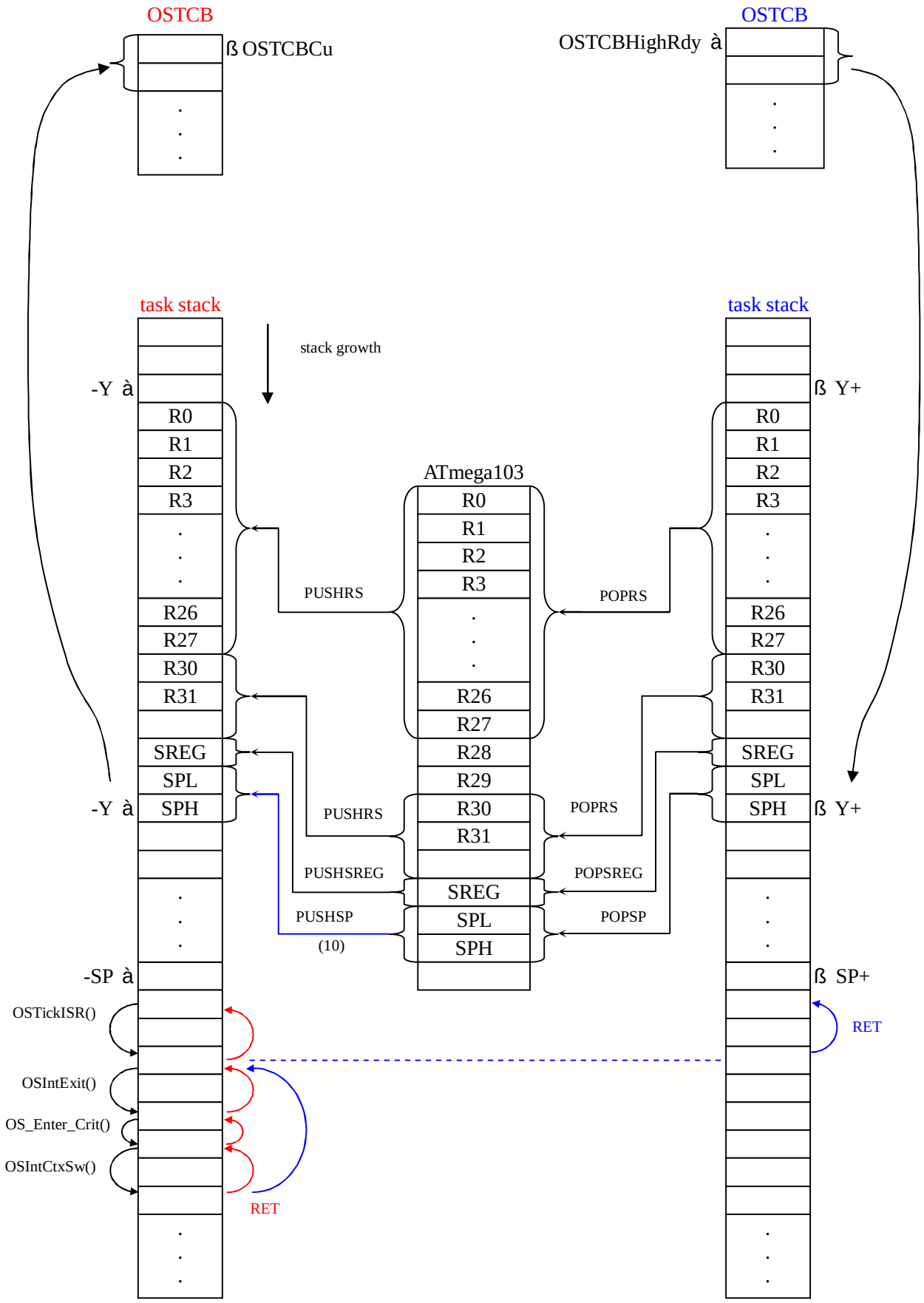
```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        IN    R16 , SREG        ; 保存SREG寄存器
        SBR    R16 , 0x80       ; 使该保存值含有中断允许信息
        ST   -Y, R16            ; 保存SREG到软件堆栈

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//重装时钟节拍读数值（中断间隔＝10ms）晶振＝4MHZ，分频＝1024分频
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        LDI     R16,256-(4000000/100/1024)  ;
```

```
        OUT     TCNT2,R16
//////////////////////////////////////////////////////////////////////////////////////////
//嵌套层数＋1
//////////////////////////////////////////////////////////////////////////////////////////
        LDS     R16,_OSIntNesting   ; 告诉uC/OS-II发生了中断
        INC     R16                 ;
        STS     _OSIntNesting,R16   ; 中断嵌套计数器加1

        CALL    _OSTimeTick         ; Call uC/OS-II's tick updating function

        CALL    _OSIntExit          ; 告诉μC/OS-II即将退出中断，并作一次任务切换

        POP_PSW                     ; 恢复SREG
        POPA                        ; 恢复现场

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
中断服务子程序返回用RET替代了RETI，是因为允许中断由恢复新任务现场完成，而不是由RETI完成。具体原因见前面叙述。
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        RET                         ; 返回到被中断任务的断点处继续执行该任务，
```



--- ohne ContextSwitch, alter Task = neuer Task = rot

--- mit ContextSwitch, alter Task = rot, neuer Task = blau



--- ohne ContextSwitch, alter Task = neuer Task = rot
--- mit ContextSwitch, alter Task = rot, neuer Task = blau
Schritt (10) PUSHSP wird nur durchgeführt, wenn ein Context-Switch erfolgt, ansonsten wird der SP ganz normal zurückgeführt