uCOS-II 在 ATmega128 上的移植 Step by Step

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。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 的内核特点

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 AVR C 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，CodevisionAVR。

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。

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

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，对该区域寻址的指针由用户自己定义，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：

A、全局中断允许位 I 清零。

B、将指向下一条指令的 PC 值压入堆栈，同时堆栈指针 SP 减 2。

C、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。

D、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

A、全局中断允许位 I 置“ 1 ”。

B、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。

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 ICCAVR V6.29 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 4K RAM，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。

2.2.1. NCLUDES.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 L2.2.1INCLUDES.H.

```
#include<iom128v.h>      //  ATmega128的寄存器头文件
#include<macros.h>       //  ICCAVR的宏
#include<stdio.h>
#include<string.h>
#include<ctype.h>
#include<stdlib.h>       //  一些 C语言的标准库

/*
*********************************************************************************
*μ C/OS-II 头文件
*********************************************************************************
*/
#include "G:\Porting\ICCAVR\porting12_8\ATmega128\os_cpu.h"
#include "G:\Porting\ICCAVR\Porting12_8\EX1_mega128\os_cfg.h"
#include "G:\Porting\ICCAVR\Porting12_8\SOURCE\ucos_ii.h"
```

要注意，μ C/OS-II 的 3个头文件的先后顺序是：os_cpu.h，os_cfg.h最后是 ucos_ii.h。

2.2.2. 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,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；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。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 L 2.2.2 OS_CPU.H.

```
#ifdef OS_CPU_GLOBALS
#define OS_CPU_EXT
#else
#define OS_CPU_EXT extern
#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 in r16,0x3F\n cli\n push r16\n
ld r16,y+"); // 关闭中断
#define OS_EXIT_CRITICAL() asm("st -y,r16\n pop r16\n out 0x3F,r16\n ld
r16,y+"); // 打开中断
#endif
#define OS_STK_GROWTH 1 // 堆栈向下生长
#define OS_TASK_SW() OSCtxSw()
```

2.2.3. 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()。其程序清单如 2.2.3 所示。

程序清单 L 2.2.3 OS_CPU_C.C

```
#define OS_CPU_GLOBALS
#include "G:\Porting\ICCAVR\porting12_8\EX1_mega128\includes.h" //包含头文件
/*
*********************************************************************
* 九个接口函数（暂未使用）
*********************************************************************
```

```
*/
#if OS_CPU_HOOKS_EN > 0 && OS_VERSION > 203
void OSInitHookBegin (void)
{
}
#endif
```

```
/*
*********************************************************************
* OSTaskStkInit()
*********************************************************************
*/
OS_STK *OSTaskStkInit (void (*task)(void *pd), void *p_arg, OS_STK *ptos,
INT16U opt)
{
INT8U *psoft_stk;
INT8U *phard_stk; //为操作 AVR 单片机软、硬件堆栈而建立的临时指针
INT16U tmp;

opt = opt; //'opt'未使用,此处可防止编译器的警告
psoft_stk = (INT8U *)ptos; // 载入堆栈指针
phard_stk = (INT8U *)ptos
- OS_TASK_SOFT_STK_SIZE // 任务栈栈空间的大小
L1
+ OS_TASK_HARD_STK_SIZE; // 系统返回的堆栈（硬件堆栈） L2
tmp = *(INT16U const *)task;
*phard_stk-- = (INT8U)tmp;
*phard_stk-- = (INT8U)(tmp >> 8); //把任务入口地址放入硬件堆栈
//******************通用寄存器入栈*************************/
```

```
*psoft_stk-- = (INT8U)0x00; // R0 = 0x00 L3
*psoft_stk-- = (INT8U)0x01; // R1 = 0x01
*psoft_stk-- = (INT8U)0x02; // R2 = 0x02
*psoft_stk-- = (INT8U)0x03; // R3 = 0x03
*psoft_stk-- = (INT8U)0x04; // R4 = 0x04
*psoft_stk-- = (INT8U)0x05; // R5 = 0x05
*psoft_stk-- = (INT8U)0x06; // R6 = 0x06
*psoft_stk-- = (INT8U)0x07; // R7 = 0x07
*psoft_stk-- = (INT8U)0x08; // R8 = 0x08
*psoft_stk-- = (INT8U)0x09; // R9 = 0x09
*psoft_stk-- = (INT8U)0x10; // R10 = 0x10
*psoft_stk-- = (INT8U)0x11; // R11 = 0x11
*psoft_stk-- = (INT8U)0x12; // R12 = 0x12
*psoft_stk-- = (INT8U)0x13; // R13 = 0x13
*phard_stk-- = (INT8U)tmp;
*phard_stk-- = (INT8U)(tmp >> 8); //把任务入口地址放入硬件堆栈
//**********R16、R17 的位置中放置向任务传递的参数*****************/
tmp = (INT16U)p_arg;
*psoft_stk-- = (INT8U)tmp;
*psoft_stk-- = (INT8U)(tmp >> 8);
*psoft_stk-- = (INT8U)0x18; // R18 = 0x18
*psoft_stk-- = (INT8U)0x19; // R19 = 0x19
*psoft_stk-- = (INT8U)0x20; // R20 = 0x20
*psoft_stk-- = (INT8U)0x21; // R21 = 0x21
*psoft_stk-- = (INT8U)0x22; // R22 =
0x22
*psoft_stk-- = (INT8U)0x23; // R23 = 0x23
*psoft_stk-- = (INT8U)0x24; // R24 =
0x24
*psoft_stk-- = (INT8U)0x25; // R25 = 0x25
```

```
*psoft_stk-- = (INT8U)0x26; // R26 = 0x26
*psoft_stk-- = (INT8U)0x27; // R27 = 0x27
/***R28、R29 用作软件堆栈的指针储存在任务控制块 OS_TCB 的 OSTCBStkPtr 中***/
*psoft_stk-- = (INT8U)0x30; // R30 = 0x30
*psoft_stk-- = (INT8U)0x31; // R31 = 0x31L2
*psoft_stk-- = (INT8U)0x80; // SREG = 0x80，开全局中
断
tmp = (INT16U)phard_stk;
*psoft_stk-- = (INT8U)(tmp >> 8); // SPH
*psoft_stk = (INT8U) tmp; // SPL
return ((void *)psoft_stk);
}
```

接下去的工作便是测试移植的代码，具体的测试工作，请参考邵贝贝译的《嵌入式实时操作系统μC/OS-II（第 2 版）》。