基于 STM32 的 UCOSII 移植笔记

By Jamza

2016 年 3 月

目录

一、前言......3
二、准备......3
1、技术资料......3
2、工程文件规划......3
三、移植......4
1、文件夹建立......4
2、项目建立......6
3、移植文件解析......7
3.1、systick.c 文件解析......8
3.2、stm32f10x_it.c 文件解析......10
3.3、 os_cpu.h 文件解析......11
3.4、 os_cpu_a.asm 文件解析......14
3.5、 os_cpu_c.c 文件解析......21
4、主函数文件解析......23
四、结语......24

一、前言

根据网上资料，以及《ARM Cortex-M3 权威指南》、《嵌入式实时操作系统 UCOSII 第二版》等相关书籍的介绍，基于 STM32 官方函数库、ALIENTEK 战舰 STM32F103 开发板、UCOSII 源码，实现 UCOSII 在 STM32 平台上的移植。

二、准备

1、技术资料

在移植之前，阅读相关的技术资料，了解移植背后的细节知识是必须的，否则移植过程中出现的问题很可能让移植无法进行下去。移植过程中用到的参考资料如下：

1）***ARM Cortex-M3 权威指南***：该书对 Cortex-M3 内核进行了详细深入的讲解，对于了解 Cortex-M3 内核有很大的帮助，笔者使用的为清华大学出版社出版，吴常玉、程凯翻译的第 2 版。

2）***嵌入式实时操作系统 UCOSII***：该书对 UCOSII 的源码进行了详细的讲解，对移植过程也有相关介绍，笔者使用的为北京航空航天大学出版社出版，邵贝贝翻译的第 2 版；

3）***STM32F1 开发指南 V3.0 库函数版本***：该资料为 ALIENTEK 战舰 STM32F103 开发板教程，对于移植的硬件平台有非常详细的介绍，移植的结果也是通过该平台的 LED、蜂鸣器等外设来验证的；

4）***STM32 中文参考手册 V10***：该资料为官方参考手册，对于 STM32F10XXX 系统芯片的相关外设、寄存器操作等有详尽的介绍；

5）***嵌入式实时操作系统 UCOSII 原理及应用***：该书对 UCOSII 入门具有很大帮助，基础知识讲解的很详细，作者为任哲；

6）***UCOSII and ARM Cortex-M3 Processors Application Note AN-1018***：该手册在 UCOS 官方网站上可以下载到，手册针对 CM3 内核的移植代码有详尽的讲解，本文中很多的讲解，图片等均出自该手册。

2、工程文件规划

在移植过程中，需要使用到 STM32 开发库、UCOSII 源码、工程应用程序、外设驱动等相关源码程序文件，由于文件种类繁多，没有良好的规划与设计，工程文件繁多容易造成管理混乱，问题排除不易定位，条理不清晰等问题。

本次移植，参考了《STM32F1 开发指南 V3.0 库函数版本》中相关内容，工程文件存放管

理如下：

1) 文件夹 APP：存放应用程序文件，包括 main 文件；

2) 文件夹 CORE：存放 ARM Cortex-M3 内核文件；

3) 文件夹 STM32_Lib：存放 STM32 官方库函数文件；

4) 文件夹 UCOSII：存放嵌入式操作系统 UCOSII 源码、移植文件；

5) 文件夹 OBJ：存放编译过程中生成的链接文件，中间文件、以及 hex 文件等；

6) 文件夹 Project：存放 keil 项目工程文件；

7) 文件夹 HARDWARE：存放外设驱动文件，本移植过程中，主要存放 LED 驱动文件与 Systick 定时器驱动文件。

三、移植

1、文件夹建立

根据“工程文件规划”章节的相关内容，建立工程文件夹。以便存放相关源文件。建立之后的文件夹如下图 1 所示。

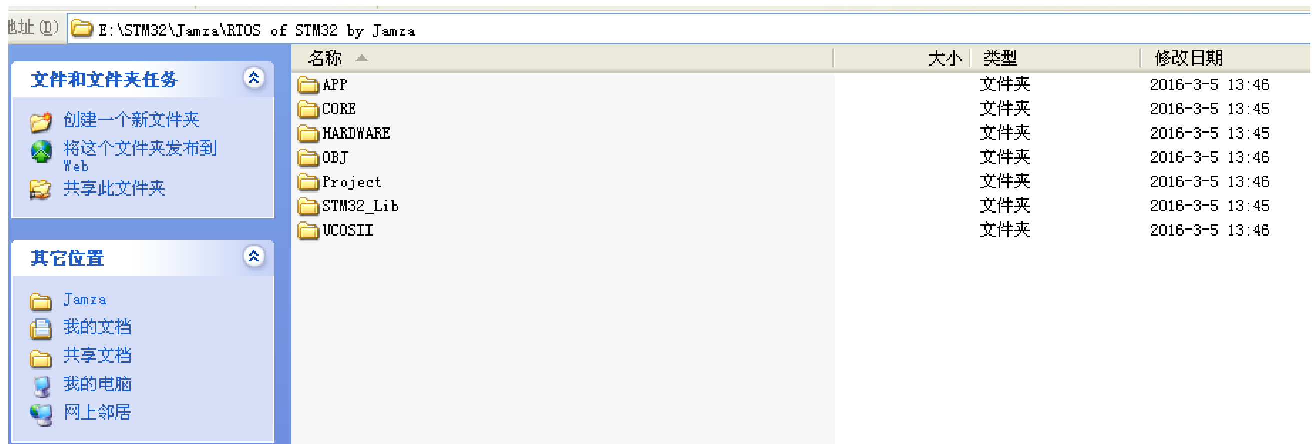

图 1

向 CORE 文件夹存放 ARM Cortex-M3 内核文件，如图 2 所示。

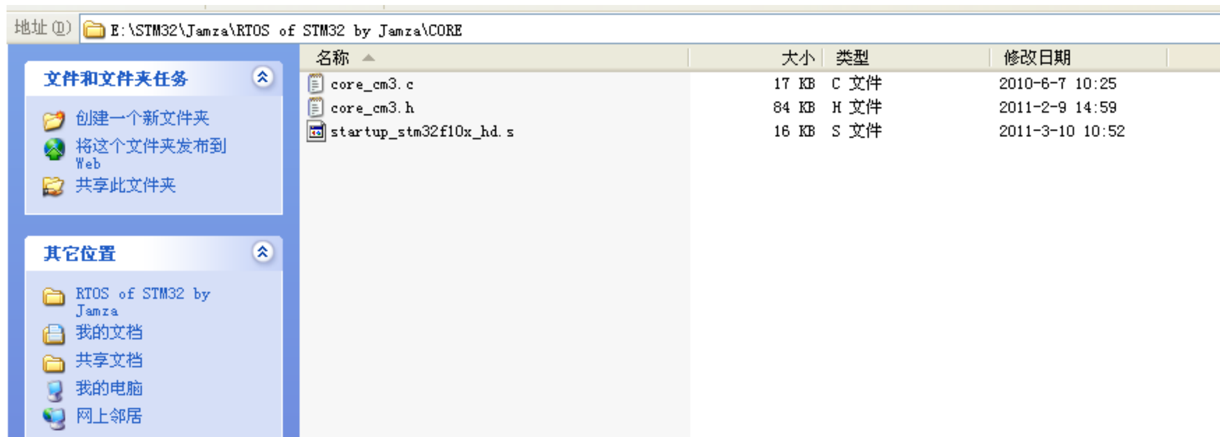

图 2

向 STM32_Lib 文件夹存放 STM32 官方库函数文件，如图 3 所示，包括 inc 文件夹与 src 文件夹，分别存放头文件与源文件，经常使用 STM32 官方库函数的同学应该知道里面的文件。

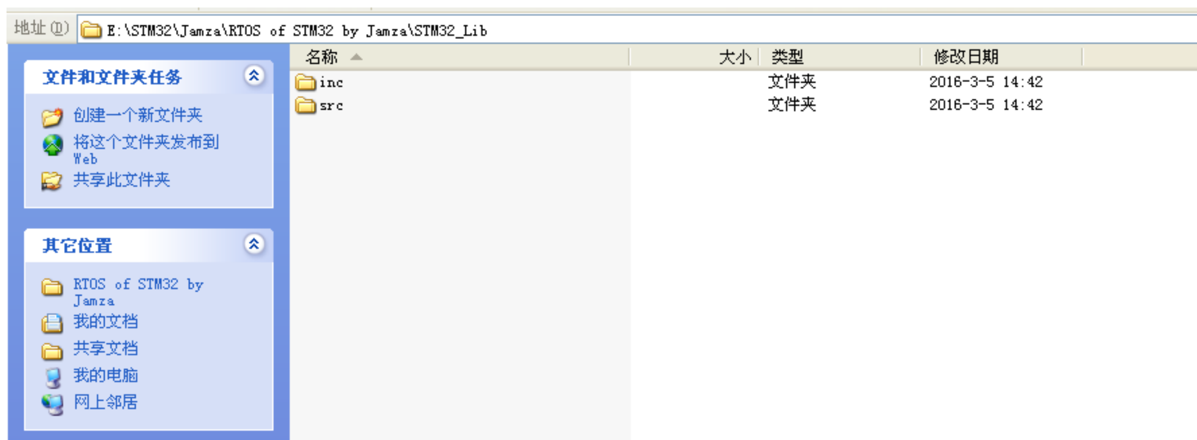

图 3

向 HARDWARE 文件夹存放 LED 驱动文件与 Systick 定时器驱动文件，具体驱动文件内容下文将详细讲解，如图 4 所示。

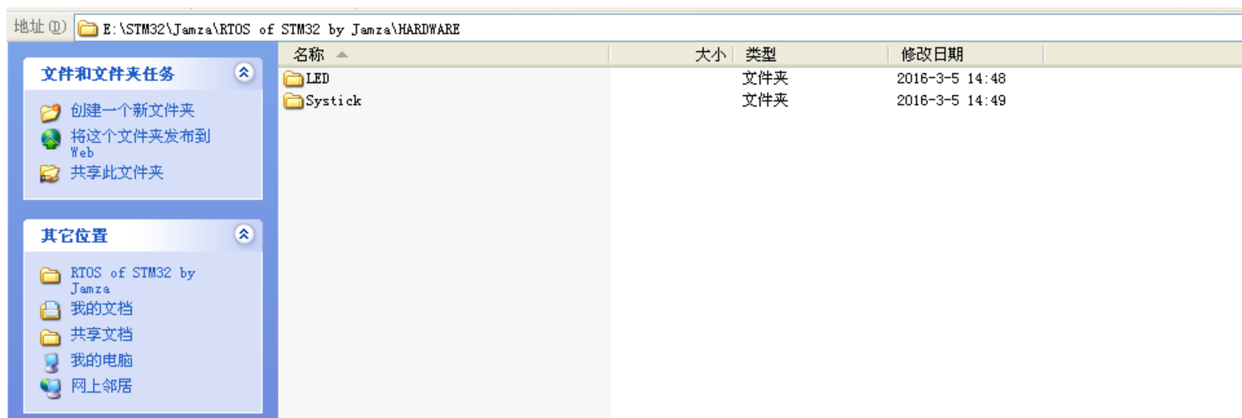

图 4

向 APP 文件夹存放 main 文件、STM32 配置头文件等相关文件，如图 5 所示。

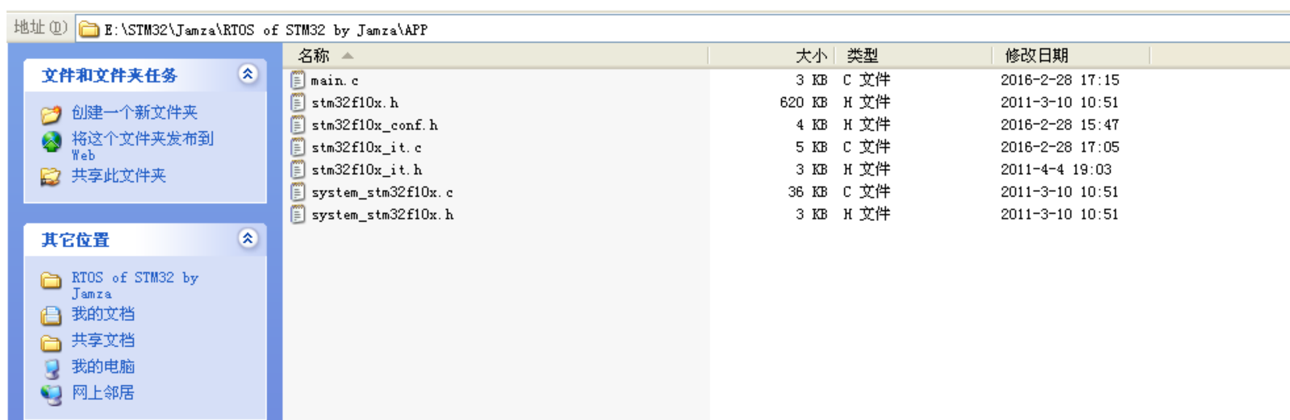

图 5

最后，向 UCOSII 文件夹存放 UCOSII 源文件，移植文件等，其中源文件与移植文件分存在 3 个文件夹中，文件夹 CORE 存放 UCOSII 内核文件、文件夹 CONFIG 存放 UCOSII 配置文件，包括 include.h 文件与 os_cfg.h 文件，文件夹 PORT 存放移植文件，包括 os_cpu.h、os_cpu_a.asm、os_cpu_c.c、os_dbg.c。如图 6 所示。

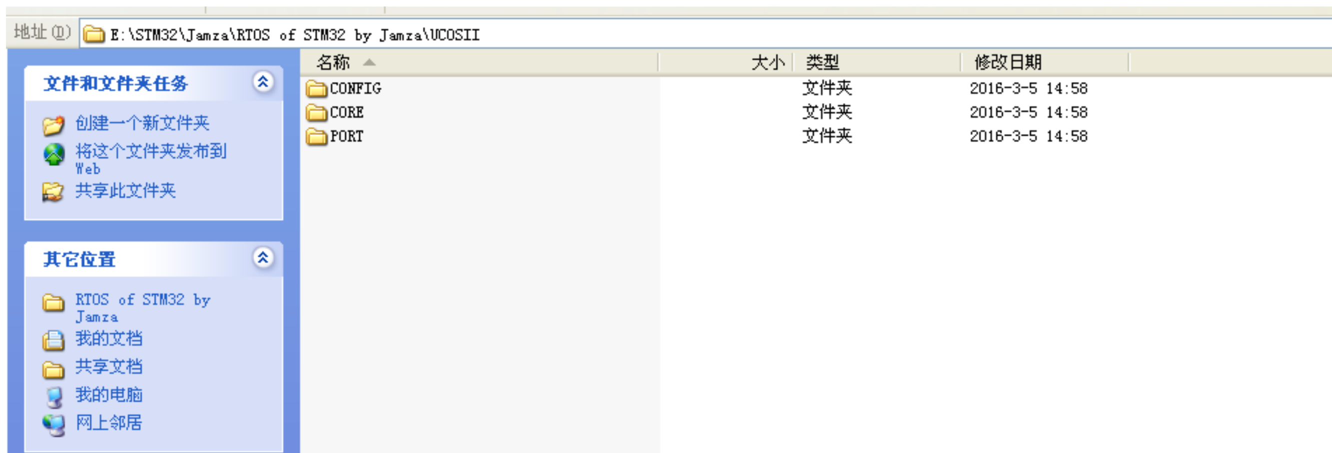

图 6

2、项目建立

打开 keil，建立工程，本次移植使用的软件为 Keil uVision5，建立项目后添加源文件，根据项目文件夹的规划设计，建立的项目分组目录如下图 7 所示。其中在添加 UCOSII 源文件到项目分组中时，需要注意不可以将 ucos_ii.c 文件添加到 UCOSII-CORE 分组中，否则编译后会出现重复定义的错误。

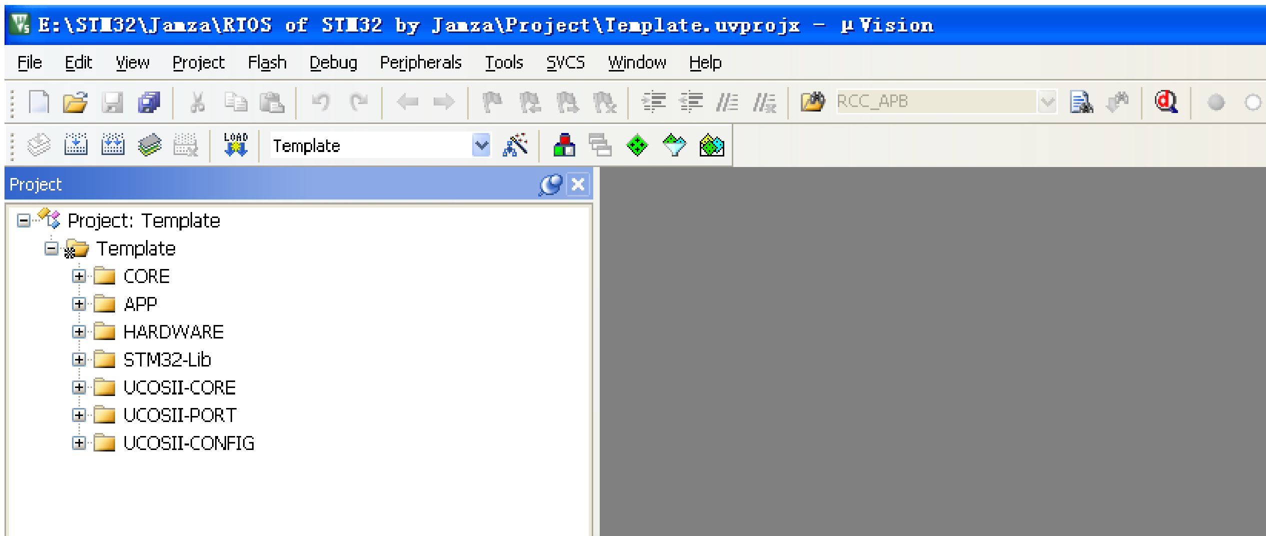

图 7

然后对项目文件进行编译链接处理，在初次编译过程中有很大概率出现错误，基本错误类型为函数未声明、函数重复定义等，根据编译错误信息更改程序，基本可以解决绝大部分问题，由于本次步骤比较简单，这里不详细讲述，具体可参考《STM32F1 开发指南 V3.0 库函数版本》、《STM32F1 UCOS 开发手册 V1.0》中相关的描述，这些资料均为 ALIENTEK 出品，知识点清楚详细，需要认真学习。

3、移植文件解析

移植过程中需要更改，或者需要理解的源码文件主要包括以下内容，下文将分别详细解析。

1) 文件 os_cfg.h，该文件主要为 UCOSII 移植过程中的配置宏定义，可根据工程实际对系统进行配置，本次移植主要修改的为宏定义 OS_TICKS_PER_SEC，该宏定义决定了 UCOSII 系统时钟节拍，其他宏定义笔者未作修改；

2) 文件 os_cpu_c.c，该文件主要为针对 STM32 的移植 C 语言文件，其中定义了几个钩子函数，任务堆栈初始化函数等；

3) 文件 os_cpu.h，还文件为针对 STM32 的移植头文件；

4) 文件 os_cpu_a.asm，该文件为针对 STM32 的移植汇编文件；

5) 文件 stm32f10x_it.c，该文件为 STM 中断处理文件，其中跟移植相关的中断函数为系统时钟 systick 中断处理函数 SysTick_Handler()，另外需要注释掉 PendSV_Handler()。下文将详细介绍；

6) systick.c，该文件为 STM32 的系统节拍时钟 systick 初始化函数，该时钟也作为 UCOSII 运行的心跳时钟。

7) main.c，该文件即为主函数文件，在主函数中创建起始任务，在起始任务中完成 IO 端口初始化，系统节拍时钟 systick 初始化，创建两个 LED 任务，然后 UCOSII 系统开始按照设计周期切换运行两个 LED 任务。

3.1、systick.c 文件解析

系统时钟定时器 systick 是一个 24 位的倒计数定时器，定时器从 RELOAD 寄存器中加载定时器初始值，然后按照时钟频率递减至 0，如果系统时钟定时器 systick 中断使能，将产生一个中断，调用中断服务程序 SysTick_Handler，UCOSII 根据这个中断产生系统运行的心跳信号。

具体系统时钟定时器 systick 的配置寄存器可在《ARM Cortex-M3 权威指南》中找到详细定义。库函数中跟系统时钟定时器 systick 相关的函数为 SysTick_Config，其函数在文件 core_cm3.h 中，注意为 cm3 的 h 文件中，而不是 c 文件中，具体函数代码如下：

```
1678
1679
1680 /* ##################################    SysTick function  ############################################ */
1681
1682 #if (!defined (__Vendor_SysTickConfig)) || (__Vendor_SysTickConfig == 0)
1683
1684 /**
1685  * @brief  Initialize and start the SysTick counter and its interrupt.
1686  *
1687  * @param   ticks   number of ticks between two interrupts
1688  * @return  1 = failed, 0 = successful
1689  *
1690  * Initialise the system tick timer and its interrupt and start the
1691  * system tick timer / counter in free running mode to generate
1692  * periodical interrupts.
1693  */
1694 static __INLINE uint32_t SysTick_Config(uint32_t ticks)
1695 {
1696   if (ticks > SysTick_LOAD_RELOAD_Msk)  return (1);            /* Reload value impossible */
1697
1698   SysTick->LOAD  = (ticks & SysTick_LOAD_RELOAD_Msk) - 1;      /* set reload register */
1699   NVIC_SetPriority (SysTick_IRQn, (1<<__NVIC_PRIO_BITS) - 1);  /* set Priority for Cortex-M0 System Interrupts */
1700   SysTick->VAL   = 0;                                          /* Load the SysTick Counter Value */
1701   SysTick->CTRL  = SysTick_CTRL_CLKSOURCE_Msk |
1702                    SysTick_CTRL_TICKINT_Msk   |
1703                    SysTick_CTRL_ENABLE_Msk;                    /* Enable SysTick IRQ and SysTick Timer */
1704   return (0);                                                  /* Function successful */
1705 }
1706
1707 #endif
1708
```

通过该函数的注释可知道，该函数首先检测了输入的参数 ticks 是否大于重载寄存器的存储最大值，调用中断配置函数对 systick 中断进行了配置，然后使能定时器，使能中断。

如果想使用 AHB/8 为时钟，则可调用库函数 SysTick_CLKSourceConfig 进行修改，该函数在库函数 misc.c 文件中，具体代码如下：

```
/**
  * @brief  Configures the SysTick clock source.
  * @param  SysTick_CLKSource: specifies the SysTick clock source.
  *   This parameter can be one of the following values:
  *     @arg SysTick_CLKSource_HCLK_Div8: AHB clock divided by 8 selected as SysTick clock source.
  *     @arg SysTick_CLKSource_HCLK: AHB clock selected as SysTick clock source.
  * @retval None
  */
void SysTick_CLKSourceConfig(uint32_t SysTick_CLKSource)
{
  /* Check the parameters */
  assert_param(IS_SYSTICK_CLK_SOURCE(SysTick_CLKSource));
  if (SysTick_CLKSource == SysTick_CLKSource_HCLK)
  {
    SysTick->CTRL |= SysTick_CLKSource_HCLK;
  }
  else
  {
    SysTick->CTRL &= SysTick_CLKSource_HCLK_Div8;
  }
}
```

可通过设置 systick 的 CTRL 寄存器的 bit2 来设置时钟来源，若设置为 0，则使用内核时钟，即使用 AHB/8 作为 systick 的时钟源，若 bit2 设置为 1，则使用 AHB 作为 systick 的时钟源。函数 SysTick_CLKSourceConfig 即为设置 systick 的时钟源。

下面介绍 systick.c 文件代码，该代码可通过两种方式来实现。

方式一，代码如下：

```
void Systick_initial(void)
{
    RCC_ClocksTypeDef rcc_clocks;

    SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK);//设置systick时钟源为AHB

    RCC_GetClocksFreq(&rcc_clocks);//获取时钟频率

    SysTick_Config(rcc_clocks.HCLK_Frequency / OS_TICKS_PER_SEC);//配置systick定时器
}
```

代码调用 SysTick_CLKSourceConfig 设置 systick 时钟源为 AHB，调用 RCC_GetClocksFreq 获取系统时钟频率，然后调用 SysTick_Config 对 systick 定时器进行配置初始化。代码中 OS_TICKS_PER_SEC 为 UCOSII 的 os_cfg.h 中的宏，表示每秒钟的 ticks 数，本移植中 OS_TICKS_PER_SEC 为 200，即 1s 中有 200 个 ticks，即每 5ms 产生一个 systick 中断，UCOSII 的系统运行心跳信号周期即为 5ms。

方式二，代码如下：

```
void Systick_initial(void)
{
    SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK);//设置systick时钟源为AHB

    if(SysTick_Config(SystemCoreClock/OS_TICKS_PER_SEC))//配置systick定时器
    {
        while(1);
    }

}
```

代码首先调用 SysTick_CLKSourceConfig 设置 systick 时钟源为 AHB，然后调用 SysTick_Config 对 systick 定时器进行配置初始化，从函数 SysTick_Config 的源码可知，若对 systick 定时器配置错误，则返回 1，若配置正常，则返回 0，利用这个特点，则可使用 if 语言对配置结果进行判断，若配置错误，则软件进入 while 死循环。

代码中使用了变量 SystemCoreClock，该变量在库文件 system_stm32f10x.c 中定义，其具体定义如下：

```
148 /******************************************************************************
149 *  Clock Definitions
150 *******************************************************************************/
151 #ifdef SYSCLK_FREQ_HSE
152   uint32_t SystemCoreClock         = SYSCLK_FREQ_HSE;        /*!< System Clock Frequency (Core Clock) */
153 #elif defined SYSCLK_FREQ_24MHz
154   uint32_t SystemCoreClock         = SYSCLK_FREQ_24MHz;        /*!< System Clock Frequency (Core Clock) */
155 #elif defined SYSCLK_FREQ_36MHz
156   uint32_t SystemCoreClock         = SYSCLK_FREQ_36MHz;        /*!< System Clock Frequency (Core Clock) */
157 #elif defined SYSCLK_FREQ_48MHz
158   uint32_t SystemCoreClock         = SYSCLK_FREQ_48MHz;        /*!< System Clock Frequency (Core Clock) */
159 #elif defined SYSCLK_FREQ_56MHz
160   uint32_t SystemCoreClock         = SYSCLK_FREQ_56MHz;        /*!< System Clock Frequency (Core Clock) */
161 #elif defined SYSCLK_FREQ_72MHz
162   uint32_t SystemCoreClock         = SYSCLK_FREQ_72MHz;        /*!< System Clock Frequency (Core Clock) */
163 #else /*!< HSI Selected as System Clock source */
164   uint32_t SystemCoreClock         = HSI_VALUE;        /*!< System Clock Frequency (Core Clock) */
165 #endif
```

本次移植使用的硬件平台，系统时钟频率为 72MHz，则变量 SystemCoreClock 为 SYSCLK_FREQ_72MHz，即 72000000。

3.2、stm32f10x_it.c 文件解析

在 stm32f10x_it.c 文件中，跟 UCOSII 移植相关的修改只有两处，一处为需要注释掉函数 PendSV_Handler，因为在移植文件 os_cpu_a.asm 中已经定义了函数 PendSV_Handler，只不过该函数使用汇编语言实现。

另一处需要修改的是为函数 SysTick_Handler 编写代码。该函数即为上文介绍的 systick 中断服务函数，当中断来临，UCOSII 在该服务函数中调用 OSTimeTick 函数进行定时任务切换调度。具体代码如下：

```
131 /**
132   * @brief  This function handles SysTick Handler.
133   * @param  None
134   * @retval None
135   */
136 void SysTick_Handler(void)
137 {
138     if(OSRunning==1)                        //OS开始跑了,才执行正常的调度处理
139     {
140         OSIntEnter();                       //进入中断
141         OSTimeTick();                       //调用ucos的时钟服务程序
142         OSIntExit();                        //触发任务切换软中断
143     }
144 
145 }
```

另外需注意，此处的函数 PendSV_Handler 与 SysTick_Handler 的函数名称不可随便更改，中断函数名称在启动文件 startup_stm32f10x_hd.s 中的中断向量表中已经明确规定了，两处的函数名称需保持一致。如下：

```
62  __Vectors       DCD     __initial_sp               ; Top of Stack
63                 DCD     Reset_Handler              ; Reset Handler
64                 DCD     NMI_Handler                ; NMI Handler
65                 DCD     HardFault_Handler          ; Hard Fault Handler
66                 DCD     MemManage_Handler          ; MPU Fault Handler
67                 DCD     BusFault_Handler           ; Bus Fault Handler
68                 DCD     UsageFault_Handler         ; Usage Fault Handler
69                 DCD     0                          ; Reserved
70                 DCD     0                          ; Reserved
71                 DCD     0                          ; Reserved
72                 DCD     0                          ; Reserved
73                 DCD     SVC_Handler                ; SVCall Handler
74                 DCD     DebugMon_Handler           ; Debug Monitor Handler
75                 DCD     0                          ; Reserved
76                 DCD     PendSV_Handler             ; PendSV Handler
77                 DCD     SysTick_Handler            ; SysTick Handler
78
```

3.3、os_cpu.h 文件解析

os_cpu.h 文件为针对 STM32 的移植头文件，下面将分段解析该文件的代码。第一段代码如下所示。

```
20  /*****************************************************************************
21  *                          定义与编译器无关的数据类型
22  *****************************************************************************/
23
24  typedef unsigned char  BOOLEAN;
25  typedef unsigned char  INT8U;              /* Unsigned  8 bit quantity       */
26  typedef signed   char  INT8S;              /* Signed    8 bit quantity       */
27  typedef unsigned short INT16U;             /* Unsigned 16 bit quantity       */
28  typedef signed   short INT16S;             /* Signed   16 bit quantity       */
29  typedef unsigned int   INT32U;             /* Unsigned 32 bit quantity       */
30  typedef signed   int   INT32S;             /* Signed   32 bit quantity       */
31  typedef float          FP32;               /* Single precision floating point*/
32  typedef double         FP64;               /* Double precision floating point*/
33
34  //STM32是32位位宽的,这里OS_STK和OS_CPU_SR都应该为32位数据类型
35  typedef unsigned int   OS_STK;             /* Each stack entry is 32-bit wide*/
36  typedef unsigned int   OS_CPU_SR;          /* Define size of CPU status register*/
37
```

以上代码主要定义了一些数据类型，为了代码移植的方便性，使用 typedef 定义一些数据类型，是良好的编程习惯。其中 OS_STK 是任务堆栈数据类型，OS_CPU_SR 为 CPU 的状态寄存器数据类型。

```
/*
*********************************************************************************************************
*                                          ARM Miscellaneous
*********************************************************************************************************
*/


//定义栈的增长方向.
//CM3中,栈是由高地址向低地址增长的,所以OS_STK_GROWTH设置为1
#define  OS_STK_GROWTH        1      /* Stack grows from HIGH to LOW memory on ARM    */

//任务切换宏,由汇编实现.
#define  OS_TASK_SW()         OSCtxSw()

```

由于 CM3 的堆栈是从高地址向低地址移动的，因此宏定义 OS_STK_GROWTH 需定义成这样的模式。另外定义一个宏 OS_TASK_SW，即任务切换宏。

```
/*
*********************************************************************************************************
*                                          PROTOTYPES
*                                     (see OS_CPU_A.ASM)
*********************************************************************************************************
*/
//OS_CRITICAL_METHOD = 1 :直接使用处理器的开关中断指令来实现宏
//OS_CRITICAL_METHOD = 2 :利用堆栈保存和恢复CPU的状态
//OS_CRITICAL_METHOD = 3 :利用编译器扩展功能获得程序状态字, 保存在局部变量cpu_sr

#define  OS_CRITICAL_METHOD   3     //进入临界段的方法

#if OS_CRITICAL_METHOD == 3
#define  OS_ENTER_CRITICAL()  {cpu_sr = OS_CPU_SR_Save();}
#define  OS_EXIT_CRITICAL()   {OS_CPU_SR_Restore(cpu_sr);}
#endif

void       OSCtxSw(void);
void       OSIntCtxSw(void);
void       OSStartHighRdy(void);

void       OSPendSV(void);

#if OS_CRITICAL_METHOD == 3u                          /* See OS_CPU_A.ASM   */
OS_CPU_SR  OS_CPU_SR_Save(void);
void       OS_CPU_SR_Restore(OS_CPU_SR cpu_sr);
#endif
OS_CPU_EXT INT32U OSInterrputSum;

#endif

/************************ (C) COPYLEFT 2010 Leafgrass ************************/
```

通过以上代码注释可看出，对于程序临界代码的保护可以有三种方法，即分别为直接使用处理器的开关中断指令实现，或者利用堆栈缓冲 CPU 寄存器状态，或者利用编译器扩展功能获得程序状态字，保存在局部变量中。在这里进行知识点扩展下，分别讲解下三种程序临界代码的保护方法。

UCOSII 在处理临界代码时候，即不希望被中断打断的代码段时，需要关闭中断，运行完临

界代码段后，再根据情况打开中断，而关闭与打开中断的实现方式与具体运行 UCOSII 的处理器，编译器相关，不同的处理器实现的方式，指令内容等均不相同，因此这部分内容在移植时候，相关代码需要匹配移植的处理器。

UCOSII 定义了两个宏来实现开中断与关中断操作，分别为 OS_ENTER_CRITICAL 与 OS_EXIT_CRITICAL，在这两个宏中封装了具体的移植处理器对应的开中断与关中断操作指令。

UCOSII 使用宏 OS_CRITICAL_METHOD 来定义三种程序临界代码的保护方法。

第一种，直接使用处理器的开关中断指令，即 OS_CRITICAL_METHOD 为 1。最简单的如 51 单片机的开关中断指令 EA=1 与 EA=0，即实现直接使用处理器的开关中断，但使用该方法的问题是，如果调用了宏 OS_EXIT_CRITICAL 以后，则处理器的中断一定为打开的状态，而无论在进入临界代码段之前处理器的中断是打开或者关闭。但是最理想的状态应该是，在用了宏 OS_EXIT_CRITICAL 以后，处理器的中断打开与否状态，应该跟进入临界代码段之前处理器的中断状态一致。因此就引申出了下面的第二种与第三种方法。

第二种，利用堆栈缓冲 CPU 寄存器状态，即 OS_CRITICAL_METHOD 为 2。即在关闭中断之前，通过堆栈缓存 CPU 的中断寄存器状态，即记住进入临界代码之前 CPU 的中断打开与否的状态，然后在退出临界代码时候，只需将堆栈中的 CPU 中断状态寄存器弹出堆栈。这样即能够实现在用了宏 OS_EXIT_CRITICAL 以后，处理器的中断打开与否状态，跟进入临界代码段之前处理器的中断状态一致。

第三种，利用编译器的扩展功能获得处理器的中断状态状态字，保存在局部变量中，即 OS_CRITICAL_METHOD 为 3。一些处理器可以读取 PSW 寄存器的值，将包含中断开关状态的 PSW 寄存器值保持在 C 语言的局部变量中，在退出临界代码段时，再将局部变量中的值回写到 PSW 寄存器中，也实现了上述功能。幸运的是，STM32 可以通过 MRS 与 MSR 指令读写特殊寄存器，因此移植 UCOSII 到 STM32 选用第三种处理方法。

在上面的代码中，变量 cpu_sr 即为缓存 PSW 的 C 语言的局部变量，OS_CPU_SR_Save 函数通过 MRS 指令读取 PSW 寄存器，关闭中断，并返回读取的 PSW 寄存器数值。OS_CPU_SR_Restore 函数恢复 PSW 寄存器数值。下文将详细讲解这几个函数。

此处的详细相关知识，可参考邵贝贝的《嵌入式实时操作系统 UCOSII》第 3.00 章节的相关内容。

余下的代码为相关函数声明。

3.4、os_cpu_a.asm 文件解析

文件 os_cpu_a.asm 跟 STM32 处理器的汇编级编程密切相关，该文件是移植的重点文件。下面将分段讲述其代码。

```
 9      IMPORT  OSRunning               ; External references
10      IMPORT  OSPrioCur
11      IMPORT  OSPrioHighRdy
12      IMPORT  OSTCBCur
13      IMPORT  OSTCBHighRdy
14      IMPORT  OSIntNesting
15      IMPORT  OSIntExit
16      IMPORT  OSTaskSwHook
17
18      EXPORT  OSStartHighRdy
19      EXPORT  OSCtxSw
20      EXPORT  OSIntCtxSw
21      EXPORT  OS_CPU_SR_Save        ; Functions declared in this file
22      EXPORT  OS_CPU_SR_Restore
23      EXPORT  PendSV_Handler
```

以上代码，使用 IMPORT 定义的表示这个是外部标号，不是在本文件中定义的，EXPORT 表示这些标号是在本文件中定义，可以供外部其他文件调用的。

IMPORT 定义的标号，熟悉 UCOSII 的同学应该都了解其中的含义，不了解的可以参考邵贝贝的《嵌入式实时操作系统 UCOSII》书中相关的介绍。EXPORT 定义的标号均在本文件中，下文将详细讲解。

```
25
26 NVIC_INT_CTRL      EQU     0xE000ED04  ; 中断控制寄存器
27 NVIC_SYSPRI2       EQU     0xE000ED20  ; 系统优先级寄存器(2)
28 NVIC_PENDSV_PRI    EQU     0xFFFF0000  ; PendSV中断和系统节拍中断
29                                        ; (都为最低，0xff).
30 NVIC_PENDSVSET     EQU     0x10000000  ; 触发软件中断的值.
31
```

以上代码通过 EQU 指令定义了四个常量，其中 NVIC_INT_CTRL 与 NVIC_SYSPRI2 是两个 STM32 处理器控制寄存器的地址，NVIC_PENDSV_PRI 与 NVIC_PENDSVSET 是两个掩码。

这里涉及到两个系统寄存器，即中断控制和状态寄存器 NVIC_INT_CTRL，与系统异常优先级寄存器 NVIC_SYSPRI2，并还涉及到 PendSV 中断，在详细讲解这些寄存器的含义与操作方式之前，首先来说说为什么 UCOSII 涉及 PendSV 中断。

现在的 CPU 都是按照程序指针 PC 的指向来运行程序的，当 UCOSII 进行任务切换时候，如果一个高优先级任务就绪，在进行任务调度切换时，将打断正在运行的低优先级任务。为了保证高优先级任务运行完成后，能够回到低优先级任务被打断处继续运行，需要将断点处的程序指针压入堆栈。当高优先级的任务结束后，再从堆栈中将断点程序指针放回程序指针 PC 中，即可实现任务切换无缝衔接。可惜的是现在绝大部分 CPU 都没有对 PC 寄存器压堆栈，弹堆栈的

指令，因此当 UCOSII 进行任务切换时，需要考虑迂回操作，实现对 PC 寄存器的改写。

还好目前 CPU 还有中断处理机制，能够实现对 PC 寄存器的改写，当 CPU 响应到一个中断时，系统会自动将断点指针压入堆栈，而中断返回指令，可自动将堆栈中断点指针弹回到 PC 寄存器中，恢复被中断的程序，实现对断点的保护与恢复。

UCOSII 聪明的使用了这个机制，当需要进行任务切换时候，通过软指令，触发一次中断，间接的保存与恢复了 PC 寄存器，当然 STM32 在响应中断时也会保存与恢复除了 PC 以往的其他寄存器，但不是全部。UCOSII 触发的这个中断即是 PendSV 中断，任务切换时，只需要通过软指令触发 PendSV 中断，在 PendSV 中断服务程序中，保护被中断的任务现场，恢复将要运行的任务现场，同时 CPU 在响应中断过程中自动改写 PC 指针，即通过人为触发中断，欺骗 CPU 改写 PC 寄存器，完美实现任务切换。

相关介绍可在任哲等编著的《嵌入式实时操作系统 UCOSII 原理及应用》第 2 版的 3.4.3 章节处查询到。

既然涉及到 PendSV 中断，就要对中断进行配置，触发中断产生，并设置中断优先级。下面将讲解中断控制和状态寄存器 NVIC_INT_CTRL，与系统异常优先级寄存器 NVIC_SYSPRI2。

中断控制和状态寄存器 NVIC_INT_CTRL，地址为 0xE000ED04，该寄存器可设置一个挂起的 NMI，设置或者清除一个挂起 PendSV，设置或者清除一个挂起 systick，查找挂起异常等功能，该寄存器的具体定义可参考广州周立功单片机发展有限公司的《Cortex-M3 技术参考手册》中第 80 页开始的相关介绍。

对于我们的 UCOSII 移植，相关的为该寄存器的 bit28，即 PENDSVSET，当设置为 1 则挂起 PendSV，设置为 0 则不挂起 PendSV。因此当需要通过软指令触发一个 PendSV 中断时候，可将掩码 NVIC_PENDSVSET 写入中断控制和状态寄存器 NVIC_INT_CTRL 即可。

CM3 内核规定了 10 个系统异常，除了复位、NMI 与硬件错误的优先级是已经规定好了以外，其余的系统异常的优先级是可以编程设置的。系统异常优先级寄存器就是用来设置系统异常的优先级的，其地址范围为 0xE000ED18-0xE000ED23。系统异常优先级寄存器的相关具体定义可参考广州周立功单片机发展有限公司的《Cortex-M3 技术参考手册》中第 87 页开始的相关介绍。

	31 … 24	23 … 16	15 … 8	7 … 0
E000ED18	PRI_7	PRI_6	PRI_5	PRI_4
E000ED1C	PRI_11	PRI_10	PRI_9	PRI_8
E000ED20	PRI_15	PRI_14	PRI_13	PRI_12

图 8

图 8 所示为系统异常优先级寄存器的位分布图，其中各个定义如下：

1) PRI_4：存储器管理错误的优先级；

2) PRI_5：总线错误的优先级；

3) PRI_6：使用错误的优先级；

4) PRI_11：SVC 的优先级；

5) PRI_12：调试监控的优先级；

6) PRI_14：PendSV 的优先级；

7) PRI_15：systick 的优先级；

8) 其他：保留。

跟移植相关的为 PendSV 中断的优先级，在上面的移植代码中，设定系统异常优先级寄存器 NVIC_SYSPRI2，地址为 0xE000ED20，设定掩码 NVIC_PENDSV_PRI 为 0xFFFF0000，将掩码写入地址为 0xE000ED20 的系统异常优先级寄存器，即是将 systick 的优先级与 PendSV 的优先级均设定为 0xFF，即设定优先级为最低。

```
40 ;********************************************************************************************************
41 ;                                  CRITICAL SECTION METHOD 3 FUNCTIONS
42 ;
43 ; Description: Disable/Enable interrupts by preserving the state of interrupts.  Generally speaking you
44 ;              would store the state of the interrupt disable flag in the local variable 'cpu_sr' and then
45 ;              disable interrupts.  'cpu_sr' is allocated in all of uC/OS-II's functions that need to
46 ;              disable interrupts.  You would restore the interrupt disable state by copying back 'cpu_sr'
47 ;              into the CPU's status register.
48 ;
49 ; Prototypes :     OS_CPU_SR  OS_CPU_SR_Save(void);
50 ;                  void       OS_CPU_SR_Restore(OS_CPU_SR cpu_sr);
51 ;
52 ; Note(s)    : 1) These functions are used in general like this:
53 ;
54 ;                 void Task (void *p_arg)
55 ;                 {
56 ;                 #if OS_CRITICAL_METHOD == 3          /* Allocate storage for CPU status register */
57 ;                     OS_CPU_SR  cpu_sr;
58 ;                 #endif
59 ;
60 ;                          :
61 ;                          :
62 ;                     OS_ENTER_CRITICAL();             /* cpu_sr = OS_CPU_SaveSR();                */
63 ;                          :
64 ;                          :
65 ;                     OS_EXIT_CRITICAL();              /* OS_CPU_RestoreSR(cpu_sr);                */
66 ;                          :
67 ;                          :
68 ;                 }
69 ;********************************************************************************************************
```

以上代码为 os_cpu_a.asm 中的一段注释，这段注释告诉我们编写应用程序时候的模板，单独列出来，提醒注意，下面继续看代码。

```
71 OS_CPU_SR_Save
72     MRS     R0, PRIMASK       ;读取PRIMASK到R0,R0为返回值
73     CPSID   I                 ;PRIMASK=1,关中断(NMI和硬件FAULT可以响应)
74     BX      LR                ;返回
75
76 OS_CPU_SR_Restore
77     MSR     PRIMASK, R0       ;读取R0到PRIMASK中,R0为参数
78     BX      LR                ;返回
```

以上两段代码为开关中断的程序段，OS_CPU_SR_Save 段为关闭中断，并返回 PRIMASK 寄存器数值的程序段。OS_CPU_SR_Restore 段为将堆栈中保存的中断开关状态回写到 PRIMASK 寄存器中。

其中 MRS 与 MSR 指令为访问特殊寄存器指令，具体格式如下：

MRS <通用寄存器>，<特殊寄存器> ； 读取特殊寄存器值，存到通用寄存器。

MSR <特殊寄存器>，<通用寄存器> ； 写通用寄存器中的值至特殊寄存器中。

CM3 专门设置了 CPS 指令，用来开关中断与异常，四条 CPS 指令如下：

CPSID I ： 设置 PRIMASK=1，关中断。

CPSIE I ： 设置 PRIMASK=0，开中断。

CPSID F ： 设置 FAULTMASK=1，关异常。

CPSIE F ： 设置 FAULTMASK=0，开异常。

```
81  ;/********************************************************************************
82  ;* 函数名称: OSStartHighRdy
83  ;*
84  ;* 功能描述: 使用调度器运行第一个任务
85  ;*
86  ;* 参    数: None
87  ;*
88  ;* 返 回 值: None
89  ;*******************************************************************************/
90  OSStartHighRdy
91          LDR     R4, =NVIC_SYSPRI2       ; set the PendSV exception priority
92          LDR     R5, =NVIC_PENDSV_PRI
93          STR     R5, [R4]
94
95          MOV     R4, #0                  ; set the PSP to 0 for initial context switch call
96          MSR     PSP, R4
97
98          LDR     R4, =OSRunning          ; OSRunning = TRUE
99          MOV     R5, #1
100         STRB    R5, [R4]
101
102                                         ;切换到最高优先级的任务
103         LDR     R4, =NVIC_INT_CTRL      ;rigger the PendSV exception (causes context switch)
104         LDR     R5, =NVIC_PENDSVSET
105         STR     R5, [R4]
106
107         CPSIE   I                       ;enable interrupts at processor level
108 OSStartHang
109         B       OSStartHang             ;should never get here
110
111 ;/********************************************************************************
```

OSStartHighRdy 程序段被 UCOSII 的 OSStart 调用，程序实现开始运行第一个优先级最高的任务。程序首先设定 PendSV 的优先级，设定 PSP 堆栈指针，设定系统运行标志 OSRunning，然后通过软指令触发 PendSV 中断，在 PendSV 中断服务程序中实现任务现场保护与切换，即完成任务调度，开始运行第一个程序。

程序第一段设定 systick 的优先级与 PendSV 的优先级均设定为 0xFF，即设定优先级为最低，此处在上文以及讲解。

程序第二段设定堆栈指针 PSP 为 0，因为此处是系统运行的第一个程序，在 PendSV 中断服务程序中不需要保存上一个任务的运行环境，因为压根就没有上一个任务。这里设定 PSP 为 0，即是通知 PendSV 中断服务程序不需要保存上一个任务的运行环境。在 PendSV 中断服务程序中将会判断 PSP 的值，下文将讲解。

程序第三段设定系统运行标志 OSRunning 为运行状态。

程序第四段即为通过设定中断控制和状态寄存器 NVIC_INT_CTRL 实现触发 PendSV 中断，然后在 PendSV 中断服务程序中实现上下文切换，在退出中断服务程序时，系统自动将第一个任务的指针写入 PC 寄存器，开始运行第一个任务代码。

程序第五段为中断使能，触发 PendSV 中断。

```
;/********************************************************************************
;* 函数名称: OSCtxSw
;*
;* 功能描述: 任务级上下文切换
;*
;* 参      数: None
;*
;* 返  回  值: None
;********************************************************************************/

OSCtxSw
        PUSH    {R4, R5}
        LDR     R4, =NVIC_INT_CTRL      ;触发PendSV异常 (causes context switch)
        LDR     R5, =NVIC_PENDSVSET
        STR     R5, [R4]
        POP     {R4, R5}
        BX      LR
```

OSCtxSw 程序段为任务级的上下文切换程序段，在 os_cpu.h 中的 OS_TASK_SW()宏封装的即是 OSCtxSw 程序段。OSCtxSw 程序段只是简单的通过设定中断控制和状态寄存器 NVIC_INT_CTRL 实现触发 PendSV 中断，然后返回。

另外需要注意的是，当在 OSCtxSw 程序段中设定触发 PendSV 中断时，PendSV 中断服务程序并不会立即运行，因为在调用 OS_TASK_SW()宏时，中断为关闭状态。当使能中断后，PendSV 中断服务程序才会有机会运行。

经常是 OS_Sched 函数调用 OS_TASK_SW()宏。

```
;/********************************************************************************
;* 函数名称: OSIntCtxSw
;*
;* 功能描述: 中断级任务切换
;*
;* 参      数: None
;*
;* 返  回  值: None
;********************************************************************************/

OSIntCtxSw
        PUSH    {R4, R5}
        LDR     R4, =NVIC_INT_CTRL      ;触发PendSV异常 (causes context switch)
        LDR     R5, =NVIC_PENDSVSET
        STR     R5, [R4]
        POP     {R4, R5}
        BX      LR
        NOP
```

OSIntCtxSw 程序段为中断级的上下文切换程序段，当一个中断服务程序结束后，OSIntExit()查看是否还有优先级更高的程序就绪，若有则调用 OSIntCtxSw 程序段进行任务切换。OSIntCtxSw 程序段也只是简单的通过设定中断控制和状态寄存器 NVIC_INT_CTRL 实现触发 PendSV 中断，然后返回。

OSIntCtxSw 程序段与 OSCtxSw 程序段代码相同，但是意义不同，OSCtxSw 程序段是任务级切换，而 OSIntCtxSw 程序段是中断级切换，是从中断服务程序退出后切换到另一个任务，此时，被中断打断的任务的运行环境已经缓存至堆栈中，因此在从中断切换到新的任务时，无需重新将被中断打断的任务的运行环境入堆栈。

```
148 ;/**********************************************************************************
149 ;* 函数名称: OSPendSV
150 ;*
151 ;* 功能描述: OSPendSV is used to cause a context switch.
152 ;*
153 ;* 参    数: None
154 ;*
155 ;* 返 回 值: None
156 ;**********************************************************************************/
157
158 PendSV_Handler
159     CPSID   I                                   ; Prevent interruption during context switch
160     MRS     R0, PSP                             ; PSP is process stack pointer
161                                                 ;如果在用PSP堆栈,则可以忽略保存寄存器,参考CM3权
162     CBZ     R0, PendSV_Handler_Nosave           ; Skip register save the first time
163
164     SUBS    R0, R0, #0x20                       ; Save remaining regs r4-11 on process stack
165     STM     R0, {R4-R11}
166
167     LDR     R1, =OSTCBCur                       ; OSTCBCur->OSTCBStkPtr = SP;
168     LDR     R1, [R1]
169     STR     R0, [R1]                            ; R0 is SP of process being switched out
170
171                                                 ; At this point, entire context of process has
172 PendSV_Handler_Nosave
173     PUSH    {R14}                               ; Save LR exc_return value
174     LDR     R0, =OSTaskSwHook                   ; OSTaskSwHook();
175     BLX     R0
176     POP     {R14}
177
178     LDR     R0, =OSPrioCur                      ; OSPrioCur = OSPrioHighRdy;
179     LDR     R1, =OSPrioHighRdy
180     LDRB    R2, [R1]
181     STRB    R2, [R0]
182
183     LDR     R0, =OSTCBCur                       ; OSTCBCur  = OSTCBHighRdy;
184     LDR     R1, =OSTCBHighRdy
185     LDR     R2, [R1]
186     STR     R2, [R0]
187
188     LDR     R0, [R2]                            ; R0 is new process SP; SP = OSTCBHighRdy->OSTC
189     LDM     R0, {R4-R11}                        ; Restore r4-11 from new process stack
190     ADDS    R0, R0, #0x20
191     MSR     PSP, R0                             ; Load PSP with new process SP
192     ORR     LR, LR, #0x04                       ; Ensure exception return uses process stack
193     CPSIE   I
194     BX      LR                                  ; Exception return will restore remaining conte
195
196  end
```

PendSV_Handler 程序段即为上文多次提到的 PendSV 中断服务程序，由汇编语言实现上下午切换。因为 CM3 架构处理器在响应中断时候，自动将 xPSR、PC、LR、R12、R3、R2、R1、R0 这 8 个寄存器压入堆栈，在退出中断时，自动将 xPSR、PC、LR、R12、R3、R2、R1、R0 这 8 个寄存器弹出堆栈，其余 R4~R11 寄存器就需要程序压入堆栈。因此 PendSV 中断服务程序需要将 R4~R11 寄存器压入堆栈以及弹出堆栈。

在详细解析源代码之前，先看看 PendSV_Handler 程序段的伪代码：

```
OS_CPU_PendSVHandler:
        if (PSP != NULL) {                                  (1)
                Save R4-R11 onto task stack;                (2)
                OSTCBCur->OSTCBStkPtr = SP;                 (3)
        }
        OSTaskSwHook();                                     (4)
        OSPrioCur = OSPrioHighRdy;                          (5)
        OSTCBCur  = OSTCBHighRdy;                           (6)
        PSP       = OSTCBHighRdy->OSTCBStkPtr;              (7)
        Restore R4-R11 from new task stack;                 (8)
        Return from exception;                              (9)
```

首先检查 PSP 堆栈指针是否为 NULL，即是否为 0，若 PSP 为 0，则说明即将运行的为系统第一个任务，无需缓存 R4~R11 寄存器，否则，即需缓存 R4~R11 寄存器，即前一个运行任务的运行环境，前一运行任务的 xPSR、PC、LR、R12、R3、R2、R1、R0 已经在响应 PendSV 中断时由处理器自动压入堆栈。将前一任务的堆栈指针存入任务控制块中。

然后调用任务切换钩子函数 OSTaskSwHook。

然后设定 OSPrioCur 为当前就绪任务中优先级最高的任务优先级，设定 OSTCBCur 指向当前就绪任务中优先级最高的任务的控制块，设定 PSP 指向当前就绪任务中优先级最高的任务的堆栈。

然后从当前就绪任务中优先级最高的任务的堆栈中弹出 R4~R11 寄存器，即恢复运行缓存。退出中断，处理器自动将 xPSR、PC、LR、R12、R3、R2、R1、R0 从当前就绪任务中优先级最高的任务的堆栈中弹出，即实现任务上下文切换。

根据伪代码的含义去理解汇编源代码会容易很多。汇编的程序的操作流程基本上是按照伪代码的流程进行，其中相关的汇编指令编码查看相关手册。当对汇编指令编码理解后，结合伪代码，这段程序会很好理解，这里不再详细讲解每一句汇编源码。

3.5、os_cpu_c.c 文件解析

os_cpu_c.c 文件主要包括一些钩子函数，以及任务堆栈初始化函数，通常 UCOSII 只会调用任务堆栈初始化函数 OSTaskStkInit，其他的钩子函数留由用户扩展程序功能时候使用。os_cpu_c.c 文件中包含的函数主要包括以下：

1) OSInitHookBegin 函数：初始化启动钩子函数，被 OSInit()函数在运行起始段调用；
2) OSInitHookEnd 函数：初始化结束钩子函数，被 OSInit()函数在运行结束段调用；
3) OSTaskCreateHook 函数：任务创建钩子函数；

4) OSTaskDelHook 函数：任务删除钩子函数；

5) OSTaskIdleHook 函数：空闲任务钩子函数；

6) OSTaskStatHook 函数：统计任务钩子函数；

7) OSTaskSwHook 函数：任务切换钩子函数；

8) OSTCBInitHook 函数：任务控制块初始化钩子函数，被 OS_TCBInit()在设定了大部分任务控制块后调用；

9) OSTimeTickHook 函数：滴答时钟钩子函数；

10) OSTaskStkInit 函数：任务堆栈初始化函数。

对于钩子函数不再作详细介绍，重点介绍 OSTaskStkInit 函数。

```
192  OS_STK *OSTaskStkInit (void (*task)(void *p_arg), void *p_arg, OS_STK *ptos, INT16U opt)
193  {
194      OS_STK *stk;
195
196
197      (void)opt;                                  /* 'opt' is not used, prevent warning                */
198      stk         = ptos;                         /* Load stack pointer                                */
199
200                                                  /* Registers stacked as if auto-saved on exception   */
201      *(stk)    = (INT32U)0x01000000L;            /* xPSR                                              */
202      *(--stk)  = (INT32U)task;                   /* Entry Point                                       */
203      *(--stk)  = (INT32U)0xFFFFFFFEL;            /* R14 (LR)  (init value will cause fault if ever used)*/
204      *(--stk)  = (INT32U)0x12121212L;            /* R12                                               */
205      *(--stk)  = (INT32U)0x03030303L;            /* R3                                                */
206      *(--stk)  = (INT32U)0x02020202L;            /* R2                                                */
207      *(--stk)  = (INT32U)0x01010101L;            /* R1                                                */
208      *(--stk)  = (INT32U)p_arg;                  /* R0 : argument                                     */
209
210                                                  /* Remaining registers saved on process stack        */
211      *(--stk)  = (INT32U)0x11111111L;            /* R11                                               */
212      *(--stk)  = (INT32U)0x10101010L;            /* R10                                               */
213      *(--stk)  = (INT32U)0x09090909L;            /* R9                                                */
214      *(--stk)  = (INT32U)0x08080808L;            /* R8                                                */
215      *(--stk)  = (INT32U)0x07070707L;            /* R7                                                */
216      *(--stk)  = (INT32U)0x06060606L;            /* R6                                                */
217      *(--stk)  = (INT32U)0x05050505L;            /* R5                                                */
218      *(--stk)  = (INT32U)0x04040404L;            /* R4                                                */
219
220      return (stk);
221  }
```

以上为 OSTaskStkInit 函数的代码，函数有多个输入参数，返回堆栈指针，如下：

1) 输入参数 task：指向任务代码的指针；

2) 输入参数 p_arg：用户输入参数数据块指针；

3) 输入参数 ptos：栈顶指针，指向堆栈顶；

4) 输入参数 opt：控制选项，本函数不使用；

5) 返回：函数返回新的堆栈栈顶指令。

函数代码在任务创建时初始化任务堆栈，由于需要输入用户参数，因此寄存器 R0 用来传递输入用户参数 p_arg，由于任务初始运行时候，其对应的 CPU 寄存器值并不重要，为了调试方便，移植时对堆栈中各个 CPU 寄存器赋予的初值为寄存器号，初始化后堆栈情况如下图：

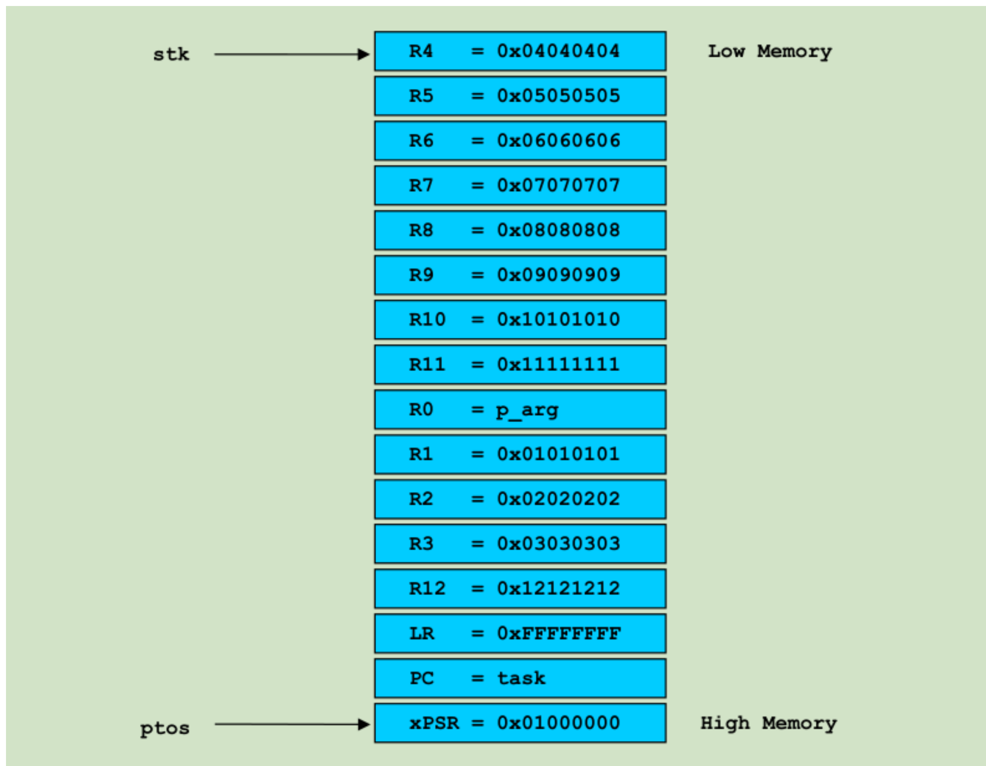

图 9

CM3 内核使用了一种满递减的堆栈操作模式，即初始堆栈为高地址处，随着堆栈的使用，堆栈指针移向低地址处。ptos 指向任务堆栈的栈顶，而 stk 指向当前堆栈的使用位置处。

4、主函数文件解析

为了检测移植的效果，在 main 主函数中创建开始任务，在开始任务中对 LED 外设进行初始化设置，对 systick 定时器进行初始化设置，开启统计任务，创建 LED0 任务，创建 LED1 任务，随后删除开始任务，开始系统运行，LED0 任务与 LEED1 任务各自按照自己的频率进行 LED 流水灯闪烁。移植成功。

其中需要注意的是 systick 定时器初始化设置一定要再统计任务初始化 OSStatInit()之前，否则系统将会进入死循环，因为在 OSStatInit()中会调用 OSTimeDly()函数，而若此时 systick 定时器还未完成设置，则 OSTimeDly()函数永远也结束不了，系统进入死机了，后续的 LED0 与 LED1 任务得不得运行。具体代码如下：

```c
int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //中断分组配置
    OSInit();                  //UCOS初始化

    OSTaskCreate(start_task,(void*)0,
                (OS_STK*)&START_TASK_STK[START_STK_SIZE-1],
                START_TASK_PRIO); //创建开始任务

    OSStart();                 //开始任务
    return 0;
}

//开始任务
void start_task(void *pdata)
{
    OS_CPU_SR cpu_sr=0;
    pdata=pdata;
    OS_ENTER_CRITICAL();          //进入临界区(关闭中断)
    led_initial();                //LED初始化
    Systick_initial();            //节拍时钟初始化

    OSStatInit();                 //开启统计任务

    OSTaskCreate(led0_task,(void*)0,
                (OS_STK*)&LED0_TASK_STK[LED0_STK_SIZE-1],
                LED0_TASK_PRIO);//创建LED0任务

    OSTaskCreate(led1_task,(void*)0,
                (OS_STK*)&LED1_TASK_STK[LED1_STK_SIZE-1],
                LED1_TASK_PRIO);//创建LED1任务

    OSTaskDel(OS_PRIO_SELF);

    OS_EXIT_CRITICAL();   //退出临界区(开中断)
}

//LED0任务
void led0_task(void *pdata)
{
    while(1)
    {
        LED0_OFF;
        OSTimeDlyHMSM(0,0,0,80);
        LED0_ON;
        OSTimeDlyHMSM(0,0,0,400);
    }
}

//LED1任务
void led1_task(void *pdata)
{
    while(1)
    {
        LED1_OFF;
        OSTimeDlyHMSM(0,0,0,1000);
        LED1_ON;
        OSTimeDlyHMSM(0,0,0,1000);
    }
}
```

四、结语

本文主要是参考第二章节所列的参考资料，根据自己经验，对 UCOSII 进行的简单的移植实

验所得的笔记，文中所有的代码，图片等均来自第二章节所列的参考资料，版权归原著所有，特此声明。文中有错误之处，请各位同学朋友指正批评，多谢。

笔者的电子邮箱为 jiangzhangha@163.com，欢迎同学朋友们多多交流，互相学习，共同进步。